ATMTools: A Toolbox for Atmospheric Propagation and Engagement Geometry Modeling

User's Guide

Version 2018b.1953

Eric P. Magee, Ph.D
Amy M. Ngwele

MZA Associates Corporation
1360 Technology Ct, Ste 200, Dayton, OH 45430-2211
http://www.mza.com
(937) 684-4100, atmtools@mza.com

The revision date is August 28, 2018

Contents

What's New . . . 13

1 Introduction **17**
1.1 Layout . . . 17
1.2 Use of the API in MATLAB® . . . 17

2 EngagementTools **18**
2.1 Layout . . . 18
2.2 Getting Started . . . 18
2.2.1 Using the Toolbox . . . 21
2.3 Function Descriptions . . . 22
2.3.1 Geometry and Engagement Structures . . . 22
2.3.1.1 GEOMSTRUCT . . . 22
2.3.1.2 ENGAGEMENTSTRUCT . . . 29
2.3.2 Propagation Geometry . . . 33
2.3.2.1 CHANGERD . . . 33
2.3.2.2 CLOUDFREEHT . . . 36
2.3.2.3 COMMONSITES . . . 37
2.3.2.4 EARTHALT . . . 38
2.3.2.5 ECF2ECI . . . 41
2.3.2.6 ECF2ENU . . . 43
2.3.2.7 ECF2LLA . . . 45
2.3.2.8 ECI2ECF . . . 48
2.3.2.9 ENU2ECF . . . 50
2.3.2.10 FACETNORMS . . . 52
2.3.2.11 FLYOUTGEOM . . . 53
2.3.2.12 FWRECKON . . . 56
2.3.2.13 GRDALTPROFILE . . . 58
2.3.2.14 ISGOODGEOM . . . 59
2.3.2.15 ISGOODHITPOINT . . . 60
2.3.2.16 LLA2ECF . . . 61
2.3.2.17 LOCALEARTHRADIUS . . . 63
2.3.2.18 LOSANGLE . . . 65
2.3.2.19 OBJECTINCIDENCE . . . 66
2.3.2.20 OBJECTINTERSECT . . . 69
2.3.2.21 PHYSICALCONST . . . 71
2.3.2.22 REVERSEGEOM . . . 72
2.3.2.23 ROTATEOBJECT . . . 73
2.3.2.24 SCREENECF . . . 75
2.3.2.25 SIMPLEGEOM . . . 76
2.3.2.26 SLANT2DOWN . . . 79
2.3.2.27 SLANT2DOWNEL . . . 80
2.3.2.28 TARGCOORDVEC . . . 81
2.3.2.29 TARGPTANGLE . . . 83
2.3.2.30 TERRAINPROFILER . . . 84
2.3.2.31 TRANSVELOCITY . . . 85
2.3.2.32 VTP2VXY . . . 88
2.3.2.33 WHICHCOMMONSITE . . . 90
2.3.3 Functions Supporting Scaling Law Analysis . . . 90
2.3.3.1 CASESTRUCT . . . 90
2.3.3.2 G_INTERFACE . . . 93

2.3.3.3 getTargetSize . . . 95
2.3.3.4 LaserField . . . 95
2.3.3.5 ParamStruct . . . 98
2.3.3.6 plotEng . . . 100
2.3.3.7 plotGeom . . . 101
2.3.3.8 plotObject . . . 102
2.3.3.9 TargGUI . . . 103
2.3.3.10 TargStruct . . . 103
2.3.4 Functions for External Data . . . 109
2.3.4.1 AddPolyFit . . . 109
2.3.4.2 bilLoad . . . 110
2.3.4.3 Boost2Flyout . . . 111
2.3.4.4 BoostLoad . . . 111
2.3.4.5 FortRead . . . 112
2.3.4.6 LoadStampParams . . . 112
2.3.4.7 TALOEngage . . . 112
2.3.5 Functions for Satellite Propagation (TLE) . . . 112
2.3.5.1 ConvertTZ . . . 113
2.3.5.2 Gregorian2Julian . . . 113
2.3.5.3 Julian2Gregorian . . . 114
2.3.5.4 LOS . . . 114
2.3.5.5 MoonPhase . . . 115
2.3.5.6 MoonPos . . . 115
2.3.5.7 propTLE . . . 116
2.3.5.8 SDP4 . . . 117
2.3.5.9 SGP4 . . . 117
2.3.5.10 SGPPrep . . . 118
2.3.5.11 SiderealTime . . . 119
2.3.5.12 SunPos . . . 119
2.3.5.13 SunRiseSet . . . 120
2.3.5.14 TLEExtract . . . 121
2.3.5.15 TLEParse . . . 122
2.3.6 API Support Functions . . . 123
2.3.6.1 APImessage . . . 123
2.3.6.2 APIstruct . . . 123
2.3.6.3 APIwarning . . . 126
2.3.6.4 loadH5 . . . 126
2.3.6.5 loadMZAScalingCodeAPI . . . 126
2.3.6.6 ParamPath . . . 126
2.3.6.7 saveH5 . . . 127
2.3.6.8 TargDataPath . . . 127
2.3.6.9 unloadMZAScalingCodeAPI . . . 127
2.3.7 Programming Utility Functions . . . 127
2.3.7.1 cell2csv . . . 127
2.3.7.2 cell2Excel . . . 128
2.3.7.3 clickableLegend . . . 128
2.3.7.4 CompStruct . . . 130
2.3.7.5 Extract . . . 130
2.3.7.6 FindStructNaN . . . 132
2.3.7.7 hline . . . 132
2.3.7.8 inpaint_nans . . . 133
2.3.7.9 IsClose . . . 134
2.3.7.10 isDaylight . . . 134
2.3.7.11 LineIntersect . . . 135

2.3.7.12 MarkFig . . . 135
2.3.7.13 mcodeHelp . . . 136
2.3.7.14 pdfOpen . . . 136
2.3.7.15 progressbar . . . 137
2.3.7.16 PSDGenRandom . . . 138
2.3.7.17 rotateticklabel . . . 139
2.3.7.18 struct2vec . . . 139
2.3.7.19 structArray2arrayStruct . . . 140
2.3.7.20 syntaxHelp . . . 140
2.3.7.21 VaryStruct . . . 141
2.3.7.22 VectorArgCheck . . . 141
2.3.7.23 vline . . . 142

3 ATMTools **144**
3.1 Layout . . . 144
3.2 Getting Started . . . 144
3.3 Using the Toolbox . . . 145
3.3.0.24 M-file or Command-Line Computations . . . 145
3.3.0.25 Using the Graphical User Interface . . . 146
3.3.0.26 Using ATMTools with WaveTrain™ . . . 154
3.4 Function Descriptions . . . 161
3.4.1 Atmosphere Structure . . . 161
3.4.1.1 AtmStruct . . . 161
3.4.2 Atmospheric Characterization . . . 167
3.4.2.1 AnisoplanaticJitter . . . 168
3.4.2.2 AnisoplanaticStrehl . . . 169
3.4.2.3 AtmosphericOTF . . . 171
3.4.2.4 BeaconD0 . . . 173
3.4.2.5 BeamIrradianceCovariance . . . 174
3.4.2.6 BeamSpread . . . 175
3.4.2.7 BoundaryAlt . . . 176
3.4.2.8 CircFresnel . . . 178
3.4.2.9 DistortionNumber . . . 180
3.4.2.10 FocusStrehl . . . 183
3.4.2.11 GreenwoodFreq . . . 185
3.4.2.12 HigherOrderPSD . . . 186
3.4.2.13 IrradianceCovariancePathWeight . . . 189
3.4.2.14 NollMatrix . . . 191
3.4.2.15 OpenLoopJitter . . . 191
3.4.2.16 OpenLoopStrehl . . . 192
3.4.2.17 PlaneD0 . . . 194
3.4.2.18 PlaneIrradianceCovariance . . . 194
3.4.2.19 PlaneR0 . . . 195
3.4.2.20 PlaneRytov . . . 196
3.4.2.21 PlaneWaveStrFcn . . . 197
3.4.2.22 PropParams . . . 198
3.4.2.23 PupilProp . . . 199
3.4.2.24 RefractAlt . . . 200
3.4.2.25 ScreenR0 . . . 201
3.4.2.26 SphericalIrradianceCovariance . . . 202
3.4.2.27 SphericalIrradianceVariance . . . 203
3.4.2.28 SphericalLogIrrCovariance . . . 204
3.4.2.29 SphericalLogIrrPathWeight . . . 205
3.4.2.30 SphericalR0 . . . 206

3.4.2.31 SphericalRytov . . . 208
3.4.2.32 SphericalTheta0 . . . 210
3.4.2.33 SphericalWaveStrFcn . . . 213
3.4.2.34 StrehlEquivFocus . . . 213
3.4.2.35 TBWaveCalc . . . 215
3.4.2.36 ThermalBlooming . . . 219
3.4.2.37 TiltAngCovScale . . . 224
3.4.2.38 TiltCovariance . . . 226
3.4.2.39 TiltCovariancePathWeight . . . 227
3.4.2.40 TiltTempCovScale . . . 229
3.4.2.41 TiltVariance . . . 231
3.4.2.42 Transmission . . . 232
3.4.2.43 TurbConst . . . 233
3.4.2.44 TylerFreq . . . 234
3.4.2.45 WaveStrFcnPathWeight . . . 236
3.4.2.46 WeightFunctionRef . . . 237
3.4.2.47 WeightMat . . . 238
3.4.2.48 WindDirCos . . . 239
3.4.2.49 wtBLS900 . . . 241
3.4.2.50 ZernikeCoeffCov . . . 241
3.4.2.51 ZernikeCoefficient . . . 242
3.4.2.52 ZernikeCoord . . . 243
3.4.2.53 ZernikeCovariancePathWeight . . . 243
3.4.2.54 ZernikePolynomial . . . 245
3.4.2.55 ZernikeReconstruct . . . 245
3.4.2.56 ZTiltPSD . . . 246
3.4.3 Atmospheric Model Functions . . . 249
3.4.3.1 AFGLAMOS . . . 249
3.4.3.2 AFRLDE_Atmos . . . 250
3.4.3.3 AIRS . . . 251
3.4.3.4 ApparentGravity . . . 252
3.4.3.5 AtmModels . . . 253
3.4.3.6 AverageAtm . . . 255
3.4.3.7 BoundaryAtm . . . 256
3.4.3.8 Bufton . . . 257
3.4.3.9 CalcHRange . . . 259
3.4.3.10 ChangeAtm . . . 260
3.4.3.11 Clear1Night . . . 263
3.4.3.12 Clear2Night . . . 264
3.4.3.13 Clear2Wind . . . 264
3.4.3.14 dewPt2RH . . . 265
3.4.3.15 DLRmodHV57 . . . 266
3.4.3.16 dynamicPiecewiseMaritimeAIRS . . . 267
3.4.3.17 DynamicViscosity . . . 268
3.4.3.18 EdlenOwens67 . . . 268
3.4.3.19 EqualCn2Screens . . . 269
3.4.3.20 FASCODE . . . 270
3.4.3.21 GADSAtmos . . . 271
3.4.3.22 GreenwoodCn2 . . . 274
3.4.3.23 HufnagelValley . . . 275
3.4.3.24 HV57 . . . 276
3.4.3.25 InnerScale . . . 276
3.4.3.26 LEEDRAtm . . . 278
3.4.3.27 LEEDRAtmos . . . 279

3.4.3.28 LEEDRWxCube . . . 282
3.4.3.29 LEEDRWxCubeAtm . . . 285
3.4.3.30 LogScaling . . . 286
3.4.3.31 LUTAtm . . . 286
3.4.3.32 Maui3Cn2 . . . 287
3.4.3.33 MIEV0 . . . 288
3.4.3.34 MODFAS . . . 291
3.4.3.35 ModHufnagelValley . . . 294
3.4.3.36 MODTRAN . . . 295
3.4.3.37 NSLOT_v2 . . . 297
3.4.3.38 OceanAtmos . . . 298
3.4.3.39 OuterScale . . . 299
3.4.3.40 processAIRShdf . . . 300
3.4.3.41 PathDisp . . . 302
3.4.3.42 RandCn2 . . . 306
3.4.3.43 RefIndex . . . 308
3.4.3.44 SLCDay . . . 309
3.4.3.45 SLCNight . . . 310
3.4.3.46 SOR2miAtm . . . 311
3.4.3.47 SORWind . . . 311
3.4.3.48 SpeedOfSound . . . 312
3.4.3.49 TerrainAtm . . . 314
3.4.3.50 UniformAtm . . . 315
3.4.3.51 UniformCn2 . . . 315
3.4.3.52 US_Standard76 . . . 317
3.4.3.53 VTP2VLLA . . . 317
3.4.3.54 WSMRatm . . . 320
3.4.3.55 WSMRCn2 . . . 321
3.4.3.56 WSMRWind . . . 322
3.4.3.57 WSMRWindHeading . . . 324
3.4.4 LEEDR Atmospheric Model Functions . . . 324
3.4.4.1 Using LEEDR in ATMTools . . . 324
3.4.4.2 setLEEDRpath . . . 326
3.4.4.3 buildLEEDRAtmos . . . 326
3.4.4.4 buildLEEDRWxCube . . . 327
3.4.4.5 CFLOSSites . . . 329
3.4.4.6 downloadGrbData . . . 330
3.4.4.7 LEEDRLUT . . . 330
3.4.4.8 LEEDRSites . . . 332
3.4.4.9 LEEDRWXCUBELUT . . . 332
3.4.4.10 loadLEEDRAtmos . . . 335
3.4.4.11 loadLEEDRWxCube . . . 335
3.4.4.12 meanRefract . . . 336
3.4.4.13 mergeLEEDRAtmos . . . 336
3.4.4.14 pCFLOS . . . 337
3.4.4.15 saveLEEDRAtmos . . . 338
3.4.4.16 setLEEDRboundaryLayerAlt . . . 338
3.4.4.17 setLEEDRInputs . . . 339
3.4.4.18 updateLEEDRAtmos . . . 339
3.4.4.19 updateLEEDRInputs . . . 340
3.4.5 Functions to Support Turbulence Prediction Modeling . . . 340
3.4.5.1 MZAneuralNetCn2Scaling . . . 340
3.4.5.2 MZAsurfCn2 . . . 341
3.4.5.3 readMetData . . . 342

3.4.6 Specialized Mathematical Functions . . . 344
3.4.6.1 FresnelInt . . . 344
3.4.6.2 HYPER . . . 345
3.4.6.3 hyper2f3as . . . 346
3.4.6.4 Lommel . . . 346
3.4.6.5 sgamma . . . 348
3.4.6.6 ZernikeOrder . . . 348
3.4.7 API Support Functions . . . 349
3.4.7.1 ATMLookupPath . . . 349
3.4.7.2 LEEDRLookupPath . . . 349
3.4.7.3 LEEDRWxCubeLookupPath . . . 349
3.4.7.4 LookupPath . . . 350
3.4.7.5 saveMODFASAsH5 . . . 350
3.4.8 General Utility Functions and User Interfaces . . . 350
3.4.8.1 addModelType . . . 350
3.4.8.2 AeroOpticsScaling . . . 352
3.4.8.3 AreaAdditionStrehl . . . 354
3.4.8.4 AtmStruct_Interface . . . 354
3.4.8.5 AtmStruct2Cell . . . 355
3.4.8.6 ExtractBaseModel . . . 356
3.4.8.7 FAS_TAPE6 . . . 356
3.4.8.8 getAlpha . . . 357
3.4.8.9 mkFuncStr . . . 358
3.4.8.10 MOD_TAPE6 . . . 358
3.4.8.11 PhasorSumStrehl . . . 359
3.4.8.12 plotAtm . . . 360
3.4.8.13 PropConfig . . . 361
3.4.8.14 PropControl_Interface . . . 362
3.4.8.15 PropMeshParams . . . 364
3.4.8.16 RecommendedMesh . . . 367
3.4.8.17 ReverseAreaAddition . . . 368
3.4.8.18 ReverseAtm . . . 369
3.4.8.19 ScreenData . . . 370
3.4.8.20 Strehl2wfe . . . 372
3.4.8.21 wfe2Strehl . . . 372
3.4.8.22 Wind2Vxy . . . 373

4 Application Programming Interface Functions 375
4.1 Functions Available in MATLAB® . . . 375
4.1.1 Geometry . . . 375
4.1.1.1 ComputeBILLPoint . . . 375
4.1.1.2 CreateScreenLocations . . . 376
4.1.1.3 CreateScreenLocationsEcf . . . 377
4.1.1.4 CreateScreenNormPositions . . . 378
4.1.1.5 EarthAltEcf . . . 379
4.1.1.6 FlyoutGeomGC . . . 380
4.1.2 Atmosphere . . . 381
4.1.2.1 AtmEval . . . 381
4.1.2.2 extCoeffs . . . 383
4.1.2.3 boundaryAtmEval . . . 383
4.1.2.4 checkGenericModels . . . 384
4.1.2.5 leedrAtmosEval . . . 385
4.1.2.6 leedrWxCubeEval . . . 386
4.1.2.7 logScalingEval . . . 387

4.1.2.8 uniformAtmEval . . . 387
4.1.2.9 cn2Eval . . . 388
4.1.2.10 coeffEval . . . 389
4.1.2.11 densEval . . . 389
4.1.2.12 dewPtEval . . . 390
4.1.2.13 innerScaleEval . . . 391
4.1.2.14 outerScaleEval . . . 391
4.1.2.15 pressEval . . . 392
4.1.2.16 rhEval . . . 393
4.1.2.17 tempEval . . . 394
4.1.2.18 windEval . . . 394
4.1.2.19 windHeadingEval . . . 395
4.1.2.20 wlString . . . 396
4.1.2.21 OpenLoopStrehlReverseLookup . . . 396
4.1.3 LEEDR Atmosphere . . . 397
4.1.3.1 LEEDRAtmos_GetLUTFN . . . 397
4.1.3.2 LEEDRAtmos_GetLUTFNLength . . . 397
4.1.3.3 LEEDR_clearModelData . . . 397
4.1.3.4 LEEDR_setModelData . . . 398
4.1.3.5 LEEDRSiteAltitude . . . 398
4.1.3.6 LEEDRSiteLatLon . . . 399
4.1.3.7 LEEDRWxCube_clearModelData . . . 399
4.1.3.8 LEEDRWxCube_GetAlts . . . 399
4.1.3.9 LEEDRWxCube_GetData . . . 400
4.1.3.10 LEEDRWxCube_GetLUTInfo . . . 400
4.1.3.11 LEEDRWxCube_GetLUTInfoLengths . . . 401
4.1.3.12 LEEDRWxCube_GetNumWavelengths . . . 401
4.1.3.13 LEEDRWxCube_GetWavelength . . . 401
4.1.3.14 LEEDRWxCube_GetSurfaceAlts . . . 402
4.1.3.15 LEEDRWxCube_GetSurfaceData . . . 402
4.1.3.16 LEEDRWxCube_LoadLookupTable . . . 403
4.1.4 Parameter Structure . . . 403
4.1.4.1 ParamStructClearLast . . . 404
4.1.4.2 ParamStructCreate . . . 404
4.1.4.3 ParamStructDestroy . . . 404
4.1.4.4 ParamStructGetChar . . . 405
4.1.4.5 ParamStructGetCharLen . . . 405
4.1.4.6 ParamStructGetDbl . . . 406
4.1.4.7 ParamStructGetDblArray . . . 406
4.1.4.8 ParamStructGetInt . . . 407
4.1.4.9 ParamStructGetIntArr . . . 407
4.1.4.10 ParamStructGetXML . . . 408
4.1.4.11 ParamStructGetXMLLen . . . 408
4.1.4.12 ParamStructRemoveField . . . 409
4.1.4.13 ParamStructSetChar . . . 409
4.1.4.14 ParamStructSetDbl . . . 410
4.1.4.15 ParamStructSetInt . . . 410
4.1.5 General Utility Functions . . . 411
4.1.5.1 GetFieldDataPath . . . 411
4.1.5.2 GetFieldDataPathLength . . . 411
4.1.5.3 SetFieldDataPath . . . 412
4.1.5.4 GetLookupPath . . . 412
4.1.5.5 GetLookupPathLength . . . 413
4.1.5.6 SetLookupPath . . . 413

4.1.5.7 GetParamPath . . . 414
4.1.5.8 GetParamPathLength . . . 414
4.1.5.9 SetParamPath . . . 414
4.1.5.10 GetTargDataPath . . . 415
4.1.5.11 GetTargDataPathLength . . . 415
4.1.5.12 SetTargDataPath . . . 415
4.1.5.13 errorCheck . . . 416
4.1.5.14 getWarning . . . 416
4.1.5.15 outputOn . . . 416
4.1.5.16 seeOut . . . 416
4.1.5.17 setWarning . . . 417
4.1.6 MzaSciComp General Utility Functions . . . 418
4.1.6.1 crop . . . 418
4.1.6.2 pad . . . 418
4.1.6.3 Unwrap . . . 419
4.2 Functions Only in API . . . 420
4.2.1 Parameter Structure Methods . . . 420
4.2.1.1 addDouble . . . 420
4.2.1.2 addInt . . . 420
4.2.1.3 addString . . . 420
4.2.1.4 clear . . . 421
4.2.1.5 getDouble . . . 421
4.2.1.6 getDoubleNames . . . 421
4.2.1.7 getInt . . . 422
4.2.1.8 getIntNames . . . 422
4.2.1.9 getString . . . 422
4.2.1.10 getStringNames . . . 423
4.2.1.11 getXMLtext . . . 423
4.2.1.12 isDouble . . . 423
4.2.1.13 isInt . . . 423
4.2.1.14 isString . . . 423
4.2.1.15 loadXML . . . 424
4.2.1.16 print . . . 424
4.2.1.17 printAll . . . 424
4.2.1.18 remove . . . 424
4.2.1.19 removeGroup . . . 425
4.2.2 General Utility Functions . . . 425
4.2.2.1 getFullFilePath . . . 425
4.2.2.2 LowerCase . . . 425
4.2.3 Sci Comp Numerics . . . 426
4.2.3.1 IsInfinite . . . 426
4.2.3.2 IsNan . . . 426
4.2.3.3 Nan . . . 426
4.2.3.4 NegativeInfinity . . . 426
4.2.3.5 One . . . 426
4.2.3.6 PositiveInfinity . . . 427
4.2.3.7 Zero . . . 427
4.2.4 Scalar Functions . . . 427
4.2.4.1 BesselJ . . . 427
4.2.4.2 CartesianToPolar . . . 428
4.2.4.3 CartesianToSpherical . . . 428
4.2.4.4 ComplexDivide . . . 429
4.2.4.5 ComplexMultiply . . . 429
4.2.4.6 DegreesToRadians . . . 430

4.2.4.7 ErrorFunction . . . 430
4.2.4.8 Fix . . . 430
4.2.4.9 Gamma . . . 430
4.2.4.10 Gammaln . . . 431
4.2.4.11 getNextPower2 . . . 431
4.2.4.12 PolarToCartesian . . . 431
4.2.4.13 RadiansToDegrees . . . 432
4.2.4.14 Remainder . . . 432
4.2.4.15 Round . . . 432
4.2.4.16 Sign . . . 433
4.2.4.17 SphericalToCartesian . . . 433
4.2.5 Vector Functions . . . 433
4.2.5.1 AbsElementwise . . . 433
4.2.5.2 AcosElementwise . . . 434
4.2.5.3 arrayFillD . . . 434
4.2.5.4 arrayFillI . . . 435
4.2.5.5 AsinElementwise . . . 435
4.2.5.6 AtanElementwise . . . 436
4.2.5.7 Atan2Elementwise . . . 436
4.2.5.8 ComplexElementwiseDivide . . . 436
4.2.5.9 ComplexElementwiseMultiply . . . 437
4.2.5.10 CopyVector . . . 438
4.2.5.11 CosElementwise . . . 438
4.2.5.12 CrossProduct . . . 439
4.2.5.13 CrossProductArray . . . 439
4.2.5.14 DegreesToRadiansVector . . . 440
4.2.5.15 Difference . . . 440
4.2.5.16 DifferenceElementwise . . . 440
4.2.5.17 DotProduct . . . 441
4.2.5.18 ErfElementwise . . . 441
4.2.5.19 ExpElementwise . . . 442
4.2.5.20 GammaElementwise . . . 442
4.2.5.21 GammalnElementwise . . . 443
4.2.5.22 GenerateRandomNormal . . . 443
4.2.5.23 InverseElementwise . . . 444
4.2.5.24 Laguerre . . . 444
4.2.5.25 LnElementwise . . . 445
4.2.5.26 Log10Elementwise . . . 445
4.2.5.27 LSPolyFit . . . 445
4.2.5.28 Maximum . . . 446
4.2.5.29 MaximumV . . . 446
4.2.5.30 MaximumWithIndex . . . 447
4.2.5.31 MaxOfColumns . . . 447
4.2.5.32 MaxOfRows . . . 448
4.2.5.33 Mean . . . 448
4.2.5.34 MeanOfColumns . . . 449
4.2.5.35 MeanOfRows . . . 449
4.2.5.36 Minimum . . . 450
4.2.5.37 MinimumV . . . 450
4.2.5.38 MinimumWithIndex . . . 451
4.2.5.39 MinOfColumns . . . 451
4.2.5.40 MinOfRows . . . 452
4.2.5.41 Mode . . . 452
4.2.5.42 mza_sort . . . 453

4.2.5.43 mza_vdPackM . . . 453
4.2.5.44 mza_vdPackV . . . 454
4.2.5.45 mza_vdUnpackM . . . 454
4.2.5.46 mza_vdUnpackV . . . 455
4.2.5.47 Negate . . . 455
4.2.5.48 Normalize . . . 456
4.2.5.49 NormalizeInPlace . . . 456
4.2.5.50 NormalizeRows . . . 456
4.2.5.51 NormalizeRowsInPlace . . . 457
4.2.5.52 PowElementwise . . . 457
4.2.5.53 Pow2o3Elementwise . . . 458
4.2.5.54 Product . . . 458
4.2.5.55 ProductElementwise . . . 459
4.2.5.56 ProductElementwiseIncWithScalarIP . . . 459
4.2.5.57 ProductElementwiseWithScalar . . . 460
4.2.5.58 ProductElementwiseWithScalarIP . . . 460
4.2.5.59 QuotientElementwise . . . 461
4.2.5.60 RSSElementwise . . . 461
4.2.5.61 RectangularGrid . . . 462
4.2.5.62 SinElementwise . . . 462
4.2.5.63 SpaceEvenly . . . 463
4.2.5.64 SqrtElementwise . . . 463
4.2.5.65 SquareElementwise . . . 464
4.2.5.66 Sum . . . 464
4.2.5.67 SumElementwise . . . 465
4.2.5.68 SumElementwiseWithScalar . . . 465
4.2.5.69 SumOfColumns . . . 466
4.2.5.70 SumOfRows . . . 466
4.2.5.71 TanElementwise . . . 467
4.2.5.72 VectorScalarProductWithSumIP . . . 467
4.2.5.73 vector2Norm . . . 468
4.2.6 Matrix Functions . . . 468
4.2.6.1 Chol . . . 468
4.2.6.2 Fft . . . 469
4.2.6.3 FftShift . . . 469
4.2.6.4 FftShiftInPlace . . . 470
4.2.6.5 IdentityMatrix . . . 470
4.2.6.6 InverseLU . . . 471
4.2.6.7 LinearInterpolate1D . . . 471
4.2.6.8 LinearInterpolate2D . . . 472
4.2.6.9 LinearInterpolate2DVectorOut . . . 473
4.2.6.10 LinearInterpolate3DVectorOut . . . 474
4.2.6.11 LinearSolve . . . 475
4.2.6.12 MatrixMultiply . . . 475
4.2.6.13 MatrixMultiplyAndTranspose . . . 476
4.2.6.14 PiecewiseCubicHermiteInterpolate1D . . . 477
4.2.6.15 QRDecomp . . . 478
4.2.6.16 SingularValueDecomp . . . 478
4.2.6.17 Transpose . . . 479
4.2.6.18 TransposeInPlace . . . 479

Index **481**

List of Figures

2.1 ECF, LLA, and ENU coordinate systems . . . 24
2.2 Geometric versus Geodetic earth model . . . 46
2.3 IsGoodGeom Illustration . . . 60
2.4 Plots of geometries from Example 2.3.14, (a) `Gin` and (b) `G`. . . . 73
2.5 Geometry structure interface with the default geometry structure from GeomStruct. . . . 94
2.6 Plots of engagement geometry from plotGeom . . . 102
2.7 TargGUI with default target structure loaded. . . . 104
2.8 TargGUI after specifying the class as a box. . . . 105

3.1 Atmospheric Structure Interface . . . 147
3.2 PropConfig Main Tab . . . 148
3.3 PropConfig Geometry Tab . . . 149
3.4 PropConfig Atmosphere Tab . . . 150
3.5 PropConfig Screens Tab and Table . . . 151
3.6 PropConfig Mesh Tab . . . 152
3.7 PropConfig Simulation Tab . . . 153
3.8 PropConfig XY Geometry Setup . . . 154
3.9 Baseline Adaptive Optics and Tracking (BLAT) Model . . . 155
3.10 Standard run set for BLAT that does not use data files from PropConfig. . . . 156
3.11 Runset for BLAT that loads selected data from a PropConfig data file. . . . 157
3.12 Runset for BLAT that uses **PropConfigSpec** . . . 158
3.13 HigherOrderPSD . . . 189
3.14 ZTiltPSD . . . 249
3.15 AFGL AMOS Turbulence Profile . . . 250
3.16 AFRL/DE Absorption and Scattering Model . . . 251
3.17 Bufton Wind Profile . . . 259
3.18 Clear-1 Night Turbulence Profile . . . 263
3.19 Clear-2 Night Turbulence Profile . . . 265
3.20 Clear-2 Wind Profile . . . 265
3.21 Greenwood Turbulence Profile . . . 274
3.22 Hufnagel Valley Turbulence Profile . . . 277
3.23 Hufnagel Valley (5/7) Turbulence Profile . . . 277
3.24 Maui 3 Turbulence Profile . . . 288
3.25 `MODFAS` Absorption and Scattering Model for 1.064 μm . . . 294
3.26 `MODFAS` Absorption and Scattering Model for 1.31521 μm . . . 295
3.27 Modified Hufnagel Valley Turbulence Profile . . . 295
3.28 SLC Daytime Turbulence Profile . . . 309
3.29 SLC Nighttime Turbulence Profile . . . 310
3.30 SOR wind profile for different seasons . . . 312
3.31 1976 US Standard Temperature, Pressure and Density Model . . . 318
3.32 WSMR C_n^2 Profile . . . 323
3.33 WSMR Wind Profile . . . 323
3.34 WSMR Wind Heading Profile . . . 324
3.35 Examples of plots from plotAtm. . . . 361
3.36 PropControl_Interface with default parameter settings. . . . 363

List of Tables

2.1 List of currently available sites accessible through COMMONSITES 38
2.2 Physical Constants 72

3.1 *HRANGE* for different altitudes. 259
3.2 FASCODE parameters for extracting different extinction coefficients. 271
3.3 LEEDR Wavelength Indexing 272
3.4 LEEDR Time of Day Indexing 273
3.5 LEEDR Percentile Indexing 273
3.6 Possible terrain descriptions and corresponding albedo and evaporative constant values. 342

What's New

The biggest change with the 2018a release is that much of the MATLAB® code has been converted into an API that can be used outside of Matlab. This user guide includes information about both the C++ functions and the Matlab functions, including functions from EngagementTools, which were previously in a separate document. Additionally, there are quick reference guides for both Matlab and the C++ for help with syntax. The Matlab toolbox functions that have been converted to C++ have been updated to call the API routines. After loading the API libraries into Matlab, the user can call the functions as usual. Refer to Section 1.2 for more information on using the API in Matlab.

Compatibility Considerations

This release of ATMTools has been tested in Matlab 9.1 (R2016b) and will be compatible with that version and newer. Because of changes made to the code to take advantage of improved functionality, some functions in the toolbox will not be compatible with older versions of Matlab. In this release, we are using a Matlab function contains introduced in Matlab 2016b for improved efficiency. With the 2011b release, operation on Mac and Linux is supported.

Beginning with the 2018a release of ATMTools, much of the Matlab code uses an application programming interface (API). Development of the API allows ATMTools to be used in applications other than Matlab. Many of the Matlab functions call API routines in order to avoid code duplication. In Windows the user will need to have installed the Visual C++ Redistributable Packages for Visual Studio 2013 (x64), if not already installed. These are available from http://www.microsoft.com/en-us/download/details.aspx?id=40784. On a 64-bit machine, both 32-bit (x86) and 64-bit (x64) packages will need to be installed. The user will then need to load the API in Matlab prior to using the toolbox.

ATMTools

The following changes have been made to functions in ATMTools:

- In the API, system parameters, target parameters, geometry, and atmospheric specification are imported from xml files. Updates have been made to the Matlab functions to support using the same xml files. ParamStruct, GeomStruct, AtmStruct, and TargStruct now support an xml file name input. Additionally, the new function APIStruct creates a Matlab object that supports interaction with the API ParamStruct. Functions available in the API for loading and getting data from ParamStructs can be found in Section 4.1.4. (Release 2018a)
- LEEDR weather cube functionality has been added to ATMTools, both for Matlab and C++. Instead of a simple lookup table as a function of altitude, LEEDR weather cubes provide atmospheric conditions as a function of latitude, longitude, and altitude. More information can be found in Section 3.4.4. (Release 2018a)
- Surface layer C_n^2 modeling capability has been added to the Matlab toolbox. See Section 3.4.5 for more information. (Release 2018a)
- Additional irradiance variance and covariance functions, listed below. (Release 2018a)
- PropConfig has been updated to allow output of the data that would be saved to a file if the file name is not input or is empty. The data is returned in a single structure. Additionally, the default values have been modified to use geometry and atmosphere based on defaults from GeomStruct and AtmStruct. On the Simulation tab, there is now an option to use a clipped Gaussian, clipped at the specified platform diameter. (Release 2015a)
- The code for the graphical user interfaces (GUIs) and plotting functions has been updated to work with changes to graphics in MATLAB® 2014b. This includes updates to AtmStruct_Interface, PropConfig, TBWaveCalc. (Release 2015a)

The following is a list of functions that have been added with the most recent additions being first:

- Functions supporting LEEDR weather cube data
  - LEEDRWXCUBELUT creates a MATLAB® object for interfacing with LEEDR weather cube data. (Release 2018b)
  - DOWNLOADGRBDATA facilitates downloading forecast data for building LEEDR weather cubes. (Release 2018b)
  - BUILDLEEDRWXCUBE creates LEEDR Wx cube data using LEEDR. (Release 2018a)
  - LEEDRWXCUBE implements a lookup table into LEEDR weather data. (Release 2018a)
  - LEEDRWXCUBEATM returns an Atm structure that uses LEEDR Wx cube data. (Release 2018a)
  - LOADLEEDRWXCUBE loads LEEDR Wx cube data for use within ATMTools. (Release 2018a)
- Functions supporting API functionality (Release 2018a)
  - LOOKUPPATH sets default directories in the API for various lookup table data.
  - SAVEMODFASASH5 saves MODFAS LUT data to an h5 format for use in the API.
- Functions for surface layer C_n^2 modeling (Release 2018a)
  - MZASURFCN2 applies bulk modeling methods to estimate the ground level C_n^2 value
  - MZANEURALNETCN2SCALING applies neural network to the inputs and return the resulting scale factor.
  - READMETDATA reads meteorological data from a file.
- Functions for computing irradiance covariance (Release 2018a)
  - BEAMIRRADIANCECOVARIANCE
  - IRRADIANCECOVARIANCEPATHWEIGHT
  - PLANEIRRADIANCECOVARIANCE
  - SPHERICALIRRADIANCECOVARIANCE
  - SPHERICALIRRADIANCEVARIANCE
  - SPHERICALLOGIRRCOVARIANCE
  - SPHERICALLOGIRRPATHWEIGHT
  - SPHERICALLOGIRRWEIGHTFCNSLUT
- The new function DYNAMICPIECEWISEMARITIMEAIRS computes C_n^2 using AFIT's dynamic piecewise model based on AIRS maritime observations. (Release 2015a)
- The new function PLOTATM creates plots of atmospheric data from an input Atm structure. It also adds a menu to the resulting figure window for additional plot manipulation. (Release 2015a)
- The new function UPDATELEEDRATMOS can update existing LEEDR 3.3 data to LEEDR 4.0 by converting the inputs and recomputing the atmospheric data using LEEDR 4. (Release 2015a)
- The functions AIRS and PROCESSAIRSHDF have been added for using measurements from the Atmospheric InfraRed Sounder (AIRS) for atmospheric model data. (Release 2015a)
- The functions SPHERICALWAVESTRFCN and PLANEWAVESTRFCN compute the spherical and plane wave, respectively, structure functions using path dependent inner and outer scale models. To support the calculation, the new function WAVESTRFCNPATHWEIGHT computes the path weighting of turbulence for the structure function. (Release 2015a)
- The new function INNERSCALE computes inner scale for turbulence using a physical model. Additionally, the function OUTERSCALE has been updated to include the option of computing outer scale using a physical model as opposed to a simple scale factor on altitude. (Release 2015a)

The following is a list of bugs that have been fixed in ATMTools functions:

- An error in PropConfig when changing the display after a simulation has been run with a single time step has been fixed. (Release 2018a)
- A bug in RandCn2 that caused the function to output profiles with no variability along the path has been fixed. (Release 2018a)
- When specifying long ranges such as that for modeling satellite geometries with a geodetic earth model using a max altitude for atmospheric modeling, the calculation of altitudes is incorrect resulting in bad geometries and NaNs for propagation parameters. This has been fixed. (Release 2018a)
- An error in PropConfig that occurred in Matlab® 2014b and newer when switching to geodetic earth model has been fixed. (Release 2018a)
- An issue that causes PropConfig to fail when a user input Atm structure does not have an extinction model has been fixed. (Release 2018a)
- An issue in the functions GADSAtmos and OceanAtmos causing them to crash with a scalar altitude input of zero, i.e. h = 0, has been fixed. (Release 2018a)
- The function MODTRAN would crash when the user specified an HRANGE input. This has been fixed and FASCODE has been updated for consistency. (Release 2018a)

EngagementTools

The following are changes that have been made recently to EngagementTools:

- The function TargStruct now sets *Targ.StageLength* based on the actually length of the target (along the axis of motion). The new function getTargetSize supports computing the target size from the object vertices. Additionally, functionality has been added to TargStruct to support loading object wavefront files to support custom target object information. (Release 2018b)
- The new function isDaylight returns whether a specified time at a particular location is within daylight hours. (Release 2018b)
- New functions have been added to support the API in Matlab
  - TargDataPath returns the path for loading target object data. (Release 2018b)
  - APImessage creates a Matlab object used for error handling in the API. (Release 2018a)
  - APIstruct creates a Matlab object for interacting with the ParamStruct pointer in the API. (Release 2018a)
  - APIwarning controls API warning message display. (Release 2018a)
  - loadMZAScalingCodeAPI and unloadMZAScalingCodeAPI load and unload the scaling code API libraries in Matlab. (Release 2018a)
  - ParamPath sets the path for parameter xml files. (Release 2018a)
  - saveH5 and loadH5 saves Matlab® data to an h5 format for use in the API. (Release 2018a)
  - structArray2arrayStruct supports loading condensed H5 data with arrays of structures. (Release 2018a)
- The new function mcodeHelp can display m-code for Matlab functions that are using the API and for which we have added an m-code section in the file. This could be useful for understanding the computations made within the function. Not all Matlab toolbox functions converted to API have m-code help available.
- TargStruct now supports a cylinder with a tip. Specify the object type as the tip with a three element input for the size where the third element is the length of the cylinder.

- LaserField now supports a Laguerre-Gaussian type. (Release 2018a)
- FlyoutGeom has been updated to support great circle reckoning for an input scalar geometry. Previously only rhumb line could be used. (Release 2018a)
- Additional sites have been added to the function CommonSites. (Release 2015a)
- The functions hline and vline have been updated so that line data adjusts with changes to axes limits. In previous versions, expanding axis limits would result in horizontal/vertical lines that still spanned the original axis limits. Additionally, the function MarkFig has been updated to use normalized units for the resulting text box so that it stays in the same relative location independent of axes limits. (Release 2015a)
- The function VaryStruct has been modified to set fields in the input structure in the order that they are input. In previous versions, the field names were sorted alphabetically and setting several fields in one call would result in a different field ordering than setting the fields in separate calls. (Release 2015a)
- The function xinterpn has been removed. Use MATLAB®'s function interpn instead. (Release 2015a)
- Functions which make use of MATLAB® graphics, such as G_Interface, have been updated for compatibility with MATLAB® 2014b and newer. (Release 2015a)

The following is a list of bugs that have been fixed recently in EngagementTools functions:

- The function ObjectIncidence has been modified to compute the hit point in target P/T coordinates from the actual hit point instead of from the aimpoint. The two can be different if the aimpoint is specified as a point internal to the target object. (Release 2015a)
- The function ChangeRd has been updated to ensure that altitude remains unchanged when vectors of positions are input. This was not an issue for the case when a geometry structure is input. (Release 2015a)
- Inconsistencies with use of phase screen information when a continuous atmospheric structure is passed to the function TransVelocity have been fixed. (Release 2015a)
- The functions TLEExtract and TLEParse have been updated to work with the new data format from Space Track. (Release 2014a & 2015a)

1.Introduction

ATMTools is a toolbox for characterizing atmospheric propagation paths and provides extended atmospheric propagation models. The utilities in ATMTools support the Scaling for High energy laser and Relay Engagement (SHaRE) toolbox. ATMTools includes the supporting toolbox EngagementTools for propagation geometry and engagement modeling. EngagementTools provides tools for computing engagement parameters and extended geometry tools for laser weapon applications.

Function help for ATMTools and EngagementTools MATLAB® functions is available in html format. Type `syntaxHelp FunctionName` at the MATLAB® command prompt to see function syntax and a link to information in the MATLAB® help browser. The user's guides for EngagementTools and ATMTools are also available online at http://scalingcodes.mza.com/.

1.1 Layout

Each of the next three chapters provides more detailed information about EngagementTools, ATMTools, and the API. Chapter 2 describes EngagementTools and it's functions. Section 2.2 contains tips for getting started with EngagementTools and Section 2.3 contains detailed descriptions of each of the MATLAB® functions. The detailed descriptions include the MATLAB® function syntax and, if an equivalent API function is available, the C syntax. Chapter 3 describes the ATMTools toolbox in more detail. Section 3.2 includes tips for getting started with ATMTools. Section 3.3 has more detailed information and examples illustrating uses of ATMTools. Section 3.4 contains detailed function descriptions for ATMTools MATLAB® functions. Chapter 4 describes functions that are in the API and do not have MATLAB® equivalents in ATMTools. Section 4.1 contains functions accessible from MATLAB® and Section 4.2 contains functions API routines not accessible in MATLAB® but could be accessible when using the API from other applications.

At the end of this document, there is an index that contains an alphabetical list of all functions available followed by a list of references.

1.2 Use of the API in Matlab®

In order to use this version of ATMTools, the user will need to load the API library into MATLAB®. The user will need to have installed the Visual C++ Redistributable Packages for Visual Studio 2013 (x64), if not already installed. These are available from https://www.microsoft.com/en-us/download/details.aspx?id=40784. On a 64-bit machine, both 32-bit (x86) and 64-bit (x64) packages will need to be installed. In order to load the library, we have provide the function LOADMZASCALINGCODEAPI. The first argument is the path to the ScalingCodeAPI directory. This function can be called manually every time MATLAB® is opened before using ATMTools functions, or the user can add the call to the MATLAB® startup.m file in order to have the library loaded upon startup.

If the user has the SHaRE API, the SHaRE library will be loaded by LOADMZASCALINGCODEAPI. The SHaRE library includes all ATMTools API functions as well as SHaRE functions. The function UNLOADMZASCALINGCODEAPI can be used to unload the library if needed.

2.EngagementTools

The EngagementTools functions can separated into 6 sections, geometry and engagement structures, propagation geometry, functions supporting scaling law analysis, functions for satellite propagation, and programming utility functions. The geometry and engagement structures category includes the primary functions for engagement modeling. The propagation geometry category includes functions for coordinate conversion and decomposition. Functions for supporting scaling law analysis includes some plotting utilities and general utilities. The functions for working with external data include functions for working with data from other sources, e.g. ISAAC, STAMP, and DTED. Functions for satellite propagations include utilities for working with two and three-line element (TLE) sets and for computation of satellite positions as a function of time as well as computations for solar and lunar positions. Finally, the programming utility functions category includes some utilities for data handling in MATLAB® as well as some additional plotting utilities.

2.1 Layout

The remainder of the introduction contains some examples of how to use this toolbox. Section 2.2 shows the user how to generate the main components used by the toolbox. Section 2.2.1 goes through a sample of calculations that one can perform with EngagementTools.

Section 2.3 gives detailed descriptions of the functions available in EngagementTools and should primarily be used as a reference if the user needs more detailed information about specific functions. The functions in this section have been separated into sections with the geometry and engagement structures in Section 2.3.1, propagation geometry functions in Section 2.3.2, functions supporting scaling law analysis in Section 2.3.3, functions for working with data from other sources in Section 2.3.4, functions for satellite propagation in Section 2.3.5, and general utility functions Section 2.3.7. In the propagation geometry category there are functions for generating propagation paths and for coordinate transformations.

2.2 Getting Started

EngagementTools is built on the generation of three MATLAB® structures.[1] The first structure defines the geometry for a scenario and gives platform and target position and velocities. This structure can be used to generate the other two structures used in ATMTools the engagement structure and the atmospheric structure. The engagement structure contains more detailed information about the propagation, including velocity components for platform, target, and wind. The atmospheric structure contains top level information about the propagation path (geometry) and the parameters of interest along the propagation path, such as the index of refraction structure coefficient and the absorption coefficient. With the geometry and engagement structures and the atmospheric structure, the propagation path is fully characterized.

The geometry structure used by most functions can be generated by using the function GEOMSTRUCT [see Section 2.3.1.1]. GEOMSTRUCT can take inputs for three different geometry specifications, 'ECF,' 'LLA,' and 'Simple'. Several assumptions are made by the 'Simple' geometry and they are as follows:

1. The target is placed at 0 degrees latitude (Equator) and 0 degrees longitude (Prime Meridian).
2. The platform is placed at 0 degrees longitude and south of the target.
3. Platform and target motion is assumed to be straight and level (parallel to the surface of the earth).
4. A Geometric, or spherical, earth is assumed.

For the examples in the introduction the 'Simple' geometry will be used. If the above assumptions are too restrictive, the user can refer to Section 2.3.1.1 for examples of how to use the 'ECF' and 'LLA' geometry specifications.

[1] For help on MATLAB® structures type `HELP STRUCT` at the MATLAB® prompt.

The only required inputs to GeomStruct for a 'Simple' geometry are the geometry identifier, `'Simple'`, platform altitude, *hp*, the target altitude, *ht*, and the ground range, *rd*. For a platform at 1000 m, a target at 10 m and 20 km ground range between them, the geometry structure would be generated as in Example 2.2.1.

Example 2.2.1 ('Simple' geometry structure) *This example shows how to generate a simple geometry structure from* GeomStruct *for a platform at 1000 m, a target at 10 m and 20 km ground range between them.*

```
>> G = GeomStruct('Simple',1000,10,20000)

G =

    Coordinates: 'ECF'
             RP: [6.3720e+006 0 -2.0003e+004]
             VP: [0 2.2204e-016 0]
             RT: [6.3710e+006 0 0]
             VT: [0 2.2204e-016 0]
           TALO: []
    EarthRadius: 6.3710e+006
```

Notice that the platform and target velocities are zero (*eps*). Also notice that the position and velocity of platform and target are given in Earth Centered Fixed (ECF) coordinates [see Section 2.3.1.2 for a description of ECF coordinates and see Sections 2.3.2.7 and 2.3.2.16 for a description of conversion between ECF coordinates and a latitude, longitude, altitude, velocity, and heading description]. One could also input target velocity, *vt*, platform velocity, *vp*, platform heading, and target heading as in Example 2.2.2.

Example 2.2.2 (Simple geometry with nonzero velocities) *This function shows how to generate a simple geometry structure with velocity and headings specified.*

```
>> G = GeomStruct('Simple',1000,10,20000,100,10,90,180)

G =

    Coordinates: 'ECF'
             RP: [6.3720e+006 0 -2.0003e+004]
             VP: [0 100 0]
             RT: [6.3710e+006 0 0]
             VT: [0 0 -10]
           TALO: []
    EarthRadius: 6.3710e+006
```

Platform with $vp = 100$ *m/s heading East and a target with* $vt = 10$ *m/s heading South.*

EngagementStruct can then be used with the output of GeomStruct to calculate more engagement specific parameters such as azimuth and elevation angle of the target relative to the platform and components of velocity in a target based coordinate system [see Section 2.3.1.2 for more details]. Example 2.2.3 shows the output of EngagementStruct with the simple geometry structure of Example 2.2.2.

Example 2.2.3 (Calculation of engagement structure) *This example shows how to generate an engagement structure given a geometry structure from* GeomStruct.

```
>> S = EngagementStruct(G)

S =

          G: [1x1 struct]
         hp: 1000
         ht: 10
```

```
        rd: 2.0000e+004
         L: 2.0026e+004
        az: -1.5708
        el: -0.0510
 LOSangle: 1.5708
     TALO: []
        vp: 100
        vt: 10
       tia: 1.5229
    V_TTP: 0.4787
    V_PTP: 0
    V_PTT: 100.0000
```

Field `S.az` *is the azimuth,* `S.el` *is the elevation,* `S.LOSangle` *is the magnitude of the full angle between the platform velocity and propagation vectors (rad), and* `S.tia` *is the target incidence angle.* `S.V_TTP` *is the component of target velocity transverse to the line of site and* `S.V_PTP` *and* `S.V_PTT` *are the components of platform velocity transverse to the line of site, but parallel and transverse to the target motion, respectively.*

The atmospheric structure is generated by the function AtmStruct [see Section 3.4.1.1 for more information about AtmStruct]. The only information required by this function is information about the geometry (*hp*, *ht*, and *rd*) and information about the phase screens (number and location). Example 3.2.1 shows how to use the geometry structure to generate *Atm*

Example 2.2.4 (Generating *Atm* with input geometry structure) *This example shows how to generate an atmospheric structure by inputting the geometry structure of Example 2.2.2.*

```
>> Atm = AtmStruct(G,10)

Atm =

    hp: 1.0000e+003
    ht: 10
    rd: 2.0000e+004
     L: 2.0026e+004
     z: [10x1 double]
    dz: [10x1 double]
     h: [10x1 double]
```

Atmospheric structure with ten equally-spaced, equal-thickness phase screens.

The atmospheric structure can also contain information about C_n^2 , absorption, scattering, wind, temperature, pressure, and density. The inputs for each desired model would include first a string to identify the desired model type, 'Cn2', 'Wind,' 'Abs,' 'Scat,' etc. The next input is the name of the function to be used to calculate the profile. This can be any function on the MATLAB® path with altitude as the first argument. Available models in ATMTools can be found in Section 3.4.3. After the function name should be a list of any parameters (other than altitude) required by that function. Example 3.2.2 shows how to generate an *Atm* with profiles for C_n^2 and wind.

Example 2.2.5 (Generating *Atm* with AtmStruct) *This example shows the generation of an atmospheric structure with profiles of structure constant and wind.*

```
>> Atm = AtmStruct(1000,10,20000,8,'Cn2','HV57','Wind','UniformAtm',5,'WindHeading','UniformAtm',-90)

Atm =

                hp: 1000
```

```
              ht: 10
              rd: 2.0000e+04
               L: 2.0026e+04
         LatLong: [11 struct]
          MaxAlt: Inf
          xRange: [0 1]
               z: [81 double]
              dz: [81 double]
               h: [81 double]
             Cn2: [81 double]
         Cn2Eval: {'HV57'  'h'}
            Wind: [81 double]
        WindEval: {'UniformAtm'  'h'  [5]}
     WindHeading: [81 double]
WindHeadingEval: {'UniformAtm'  'h'  [-90]}
```

Atmospheric structure with 8 equally-spaced, equal-thickness phase screens. Notice the additional parameter required by UniformAtm *for constant wind speed and heading.*

2.2.1 Using the Toolbox

After the atmospheric structure and the geometry and engagement structures have been calculated, one can perform any number of calculations using the functions in this toolbox. Section 3.4.2 has functions for computing atmospheric propagation parameters, such as the Fried parameter (r_0), Rytov number (σ_χ^2), and isoplanatic angle (θ_o) and Greenwood and Tyler frequencies. There are also functions in Section 3.4.2 for calculating PSD's of phase abberations and for calculating irradiance after propagation. Example 3.3.1 shows how to calculate Fried parameter and Greenwood frequency.

Example 2.2.6 (Calculation of atmospheric propagation parameters) *This example shows how to calculate r_0 and the Greenwood frequency*

```
>> r0 = SphericalR0(1, 1.064e-6, G,  Atm)

r0 =

    0.1187
```

Fried parameter calculated with turbulence profile multiplier of one and wavelength of 1.064 μm, and geometry structure (`G`) as given in Example 2.2.2 and atmospheric structure (`Atm`) as given in Example 3.2.2. See Section 3.4.2.30 for a detailed description of SphericalR0.

```
>> fG = GreenwoodFreq(1, 1.064e-6,S,Atm)

fG =

  430.3621
```

Greenwood frequency calculated with turbulence profile multiplier of one and wavelength of 1.064 μm and `S` given in Example 2.2.3. See Section 3.4.2.11 for a detailed description of GreenwoodFreq.

Section 2.3.2 lists functions for calculating propagation geometry, such as SimpleGeom which works like GeomStruct with a 'Simple' geometry and ecf2LLA and LLA2ecf which convert between Earth Centered Fixed Coordinates and latitude, longitude, altitude descriptions. There are also the functions TurbConst and PhysicalConst which give exact values of constants commonly used in atmospheric propagation calculations. Section 2.3.2 are also lists functions for programming utilities, such as CompStruct which compares two structures to see if they have the same fields. Example 2.2.7 shows how to use the function EarthAlt to calculate altitude vector, elevation angle, and slant range from the platform to the target.

Example 2.2.7 (EarthAlt) *This example shows the use of* EarthAlt *to calculate altitude, slant range and elevation angle.*

```
>> [h,L,el] = EarthAlt([0.05:0.1:1]',G)

h =

  949.0089
  847.4974
  746.6137
  646.3579
  546.7299
  447.7298
  349.3575
  251.6133
  154.4970
   58.0086


L =

  2.0026e+004


el =

   -0.0510
```

The first input is the path normalized position vector from platform (0) to target (1) and represents (in `Atm`*) the position of the phase screens.* `G` *is from Example 2.2.2.*

2.3 Function Descriptions

2.3.1 Geometry and Engagement Structures

The backbone for most of the functions in EngagementTools are the functions GeomStruct and EngagementStruct and they will be described in more detail in this section.

2.3.1.1 GeomStruct

Syntax

G = GeomStruct(*'Simple', hp, ht, rd, [vp], [vt], ... [PlatformHeading], [TargetHeading], [EarthRadius]*)

G = GeomStruct(*'LLA', RP, RT, [VP], [VT], [EarthModel or EarthRadius]*)

G = GeomStruct(*'ECF', RP, RT, [VP], [VT], [EarthModel or EarthRadius]*)

G = GeomStruct(*'ENU', RP, RT, [VP], [VT], [RefLLA]... [EarthModel or EarthRadius]*)

G = GeomStruct(*XMLFile, [TALO]*)

Overloaded Syntax

```
G = GeomStruct(G, [hp], [ht], [rd], [vp], [vt], ...
    [PlatformHeading], [TargetHeading])
```

This function returns a MATLAB® structure of geometry parameters, including platform and target position and velocity, given a list of input geometry specifications which depend on the geometry identifer. Currently supported geometry identifiers are 'Simple,' 'LLA,' 'ECF,' or 'ENU.' [The geometry identifier may also be a case code from the SCALE or SHaRE toolboxes, if available, in which case no other inputs are needed.] The Latitude, Longitude, Altitude (LLA) coordinate system is a curvilinear coordinate system in which it is necessary to specify the earth model. The Earth Centered Fixed (ECF) Coordinate system is a Cartesian coordinate system defined with the $z-$axis through the North pole, the $x-$axis through 0 degrees longitude, and the $y-$axis through 90 degrees East longitude as shown in Figure 2.1(a) [see Section 2.3.2.7 and Section 2.3.2.16 for descriptions of the conversion from LLA to ECF and velocity/heading to ECF (and vice-versa)]. The East North Up (ENU) coordinate system is a Cartesian coordinate system formed by a plane tangent to the Earth's surface fixed to a specific reference location as illustrated in 2.1(c) [see Section 2.3.2.9 and Section 2.3.2.6 for descriptions of the conversion from ENU positions and velocities to ECF coordinates (and vice-versa)]. Two typical earth models are a spherical or Geometric earth model and an oblate or Geodetic earth model. Longitude is the angle from the Prime Meridian in the plane perpendicular to the polar axis ($z-$axis) with -180 to 0 degrees defining the Western hemisphere and 0 to 180 degrees defining the Eastern hemisphere and will be the same in both models due to rotational symmetry. Latitude is measured as the angle between a line perpendicular to the tangent at the surface and a line perpendicular to the polar axis. Therefore, in both models, the Equator is at zero latitude with -90 degrees at the South Pole and +90 at the North Pole. Altitude is defined as the distance to the ellipsoid (sphere) along the normal to the surface. Figure 2.1(b) illustrates the difference in latitude and altitude between the two models for the same point (the scale has been exaggerated). The geodetic altitude and latitude are indicated by the subscript 'G.'

Velocity is simply the speed in a plane parallel to the earth's surface for straight and level flight and heading is measured clockwise from true North. As an example consider a target at 1 km above sea level at the intersection of the Prime Meridian and the Equator heading East with velocity 100 m/s. This target would have *Lat* = 0, *Long* = 0, *ht* = 1000 m, *Heading* = 90, *vt* = 100 m/s. The same target would have ECF coordinates of $\mathbf{R} = [r_{ee}+1000\ 0\ 0]$ m and $\mathbf{V} = [0\ 100\ 0]$ m/s, where r_{ee} is the radius of the earth at the equator.

There are four available geometry identifiers in GeomStruct, 'Simple,' 'ECF,' 'LLA,' or an existing geometry structure to be modified. The calling sequence depends on the input identifier.

1. **'Simple' -** For a simple geometry the inputs are

hp [scalar]	Altitude of platform above curved earth surface (m).
ht [scalar]	Altitude of target above curved earth surface (m).
rd [scalar]	Ground range of target from platform along earth surface (m).
vp [scalar]	(Optional) Speed of platform, defaults to zero (m/s).
vt [scalar]	(Optional) Speed of target, defaults to zero (m/s).
PlatformHeading [scalar]	(Optional) Heading clockwise from due north, defaults to 90 (East) (deg).
TargetHeading [scalar]	(Optional) Heading clockwise from due north, defaults to 90 (East) (deg).
EarthRadius [scalar]	(Optional) Spherical Earth Radius (m).

For a simple geometry, the target will be placed at Lat = 0, Long = 0 at the specified altitude, *ht*, and the platform will be located on the prime meridian (Long = 0) in the southern hemisphere at a latitude

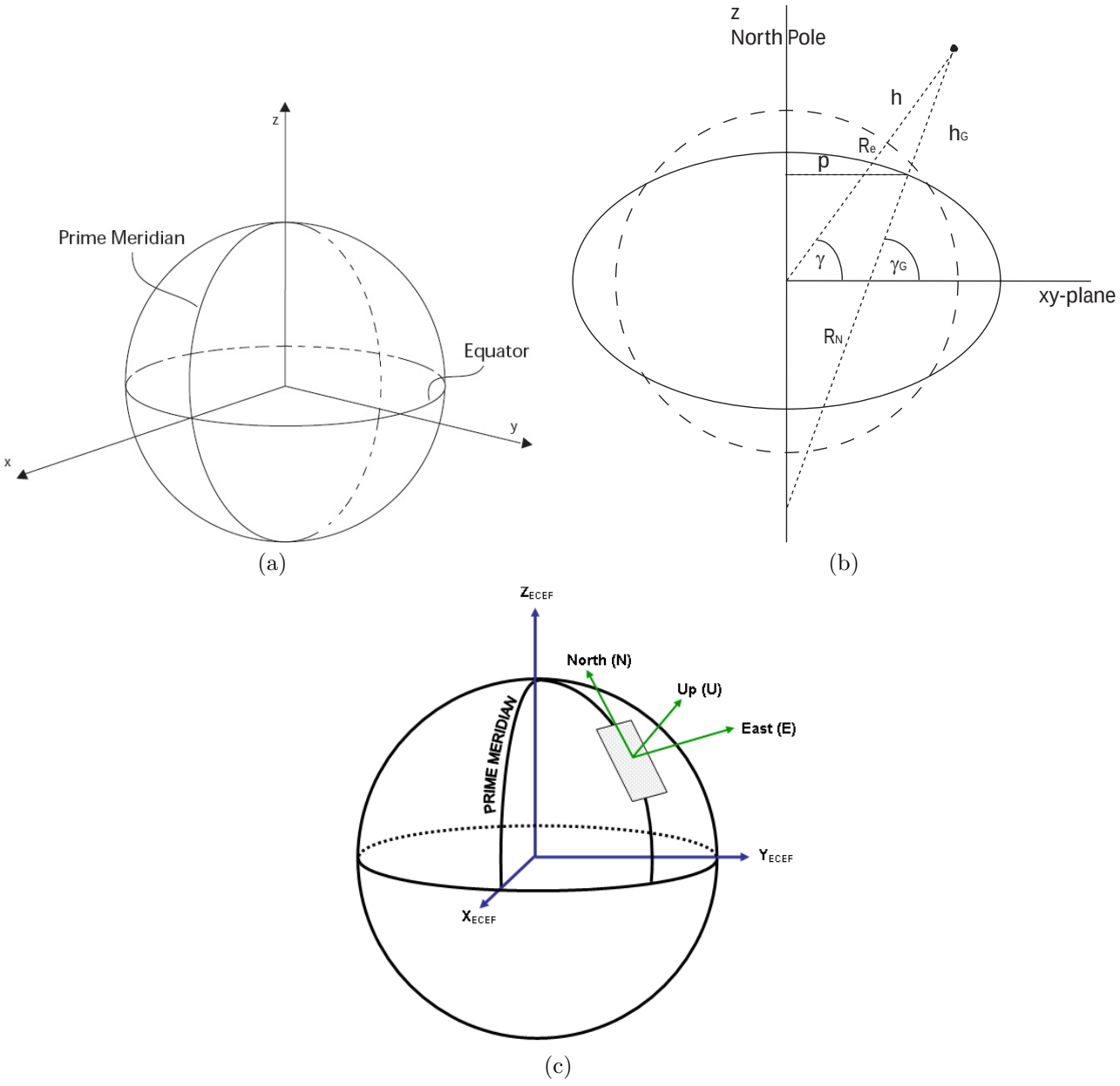


Figure 2.1: (a) Three dimensional view of ECF and LLA coordinates. (b) Two dimensional view of Geometric and Geodetic LLA coordinates. (c) Three dimensional view of ENU coordinates.

consistent with the input platform-to-target ground range, ***rd***. This geometry assumes level flight (vertical velocity = 0) of platform and target at (optional) specified headings. A Geometric (spherical) earth is assumed.

Example 2.3.1 ('Simple') *This example shows how to generate a simple geometry structure using* GeomStruct *for a platform at 3000 m, a target at 50 m and 10 km ground range between them.*

```
>> G = GeomStruct('Simple',3000,50,10000)

G =

    Coordinates: 'ECF'
             RP: [6.3740e+006 0 -1.0005e+004]
             VP: [0 2.2204e-016 0]
             RT: [6.3711e+006 0 0]
             VT: [0 2.2204e-016 0]
           TALO: []
    EarthRadius: 6.3710e+006
```

2. **'LLA' -** The inputs for Latitude, Longitude, Altitude (LLA) geometry specification are

RP [vector]	3-element LLA position vector of platform, i.e. [lat, long, alt].
RT [vector]	3-element LLA position vector of target.
VP [vector]	(Optional) 3-element LLA velocity vector of platform, i.e. [level speed, heading, vertical speed].
VT [vector]	(Optional) 3-element LLA velocity vector of target.
EarthModel [string]	(Optional) Earth model, 'Geometric' (default) for a spherical earth or 'Geodetic' for oblate earth.
EarthRadius [scalar/vector]	(Optional) If scalar - spherical Earth radius (***EarthModel*** should be Geometric), if vector - [a b], where a is the equatorial radius and b is the polar radius.

Only one of ***EarthModel*** or ***EarthRadius*** is needed. If both are passed in ***EarthRadius*** will be used.

Example 2.3.2 ('LLA') *This example shows how to generate a geometry structure using* GeomStruct *by inputting LLA coordinate vectors.*

```
>> G = GeomStruct('LLA',[10 0 2000],[5 0 10],...
                  [100 180 0],[20 90 0],'Geodetic')

G =

    Coordinates: 'ECF'
```

```
            RP: [6.2838e+006 0 1.1006e+006]
            VP: [17.3648 0 -98.4808]
            RT: [6.3540e+006 0 5.5218e+005]
            VT: [0 20 0]
          TALO: []
    EarthRadius: [6378137 6.3568e+006]

>> G = GeomStruct('LLA',CommonSites('NOP'),CommonSites('BECK'))

G =

    Coordinates: 'ECF'
            RP: [-1.4937e+006 -5.0843e+006 3.5411e+006]
            VP: [3.4774e-017 1.1837e-016 1.8462e-016]
            RT: [-1.5027e+006 -5.0827e+006 3.5380e+006]
            VT: [3.4950e-017 1.1822e-016 1.8468e-016]
          TALO: []
    EarthRadius: 6.3710e+006
```

Use the output of the function CommonSites *to generate a geometry structure using LLA position vectors.*

3. **'ECF' -** The inputs for Earth Centered Fixed (ECF) geometry specification are

RP [vector]	ECF position vector for platform.
RT [vector]	ECF position vector for target.
VP [vector]	(Optional) ECF velocity vector for platform.
VT [vector]	(Optional) ECF velocity vector for target.
EarthModel [string]	(Optional) Earth model, 'Geometric' (default) for a spherical earth or 'Geodetic' for oblate earth.
EarthRadius [scalar/vector]	(Optional) If scalar - spherical Earth radius (*EarthModel* should be Geometric), if vector - [a b], where a is the equatorial radius and b is the polar radius.

VP and *VT* may be omitted to specify the earth model as shown in Example 2.3.3. However, if either *VP* or *VT* is specified, the other must also be specified and may be passed in as `[]`. Only one of *EarthModel* or *EarthRadius* is needed. If both are passed in *EarthRadius* will be used.

Example 2.3.3 ('ECF') *This example shows how to generate a geometry structure using* GeomStruct *by inputting ECF coordinates.*

```
>> G = GeomStruct('ECF',[PhysicalConst('re')+4000 0 20000],...
                  [PhysicalConst('re')+50 0 0],'Geodetic')

G =
```

```
    Coordinates: 'ECF'
             RP: [6382008 0 20000]
             VP: [0 0 0]
             RT: [6378058 0 0]
             VT: [0 0 0]
           TALO: []
    EarthRadius: [6378137 6.3568e+006]
```

4. **'ENU' -** The inputs for East North Up (ENU) geometry specification are

RP [vector]	3-element ENU position vector of platform, i.e. [East, North, Up] from reference location *RefLLA*.
RT [vector]	3-element ENU position vector of target.
VP [vector]	(Optional) 3-element ENU velocity vector of platform.
VT [vector]	(Optional) 3-element ENU velocity vector of target.
RefLLA [vector]	(Optional) 3-element LLA position vectors [Lat Long Alt] for reference location. Lat is the Latitude in degrees (-90 to 90), Long is the longitude positions in degrees (-180 to 180), and Alt is the altitude (m). Defaults to [0 0 0].
EarthModel [string]	(Optional) Earth model, 'Geometric' (default) for a spherical earth or 'Geodetic' for an oblate earth.
EarthRadius [scalar/vector]	(Optional) If scalar - spherical Earth radius (*EarthModel* should be Geometric), if vector - [a b], where a is the equatorial radius and b is the polar radius.

VP, *VT* and *RefLLA* may be omitted to specify the earth model as for 'ECF' or 'LLA' specifications. However, if any of *VP*, *VT*, and *RefLLA* are to be specified, the others must also be specified, but may be passed in as `[]` as in Example 2.3.4. Only one of *EarthModel* or *EarthRadius* is needed. If both are passed in *EarthRadius* will be used.

Example 2.3.4 ('ENU') *This example shows how to generate a geometry structure using* GeomStruct *by inputting ENU coordinates.*

```
>> G = GeomStruct('ENU',[0 10000 0],[5000 10000 0],[],[],[0 0 0],'Geodetic')

G =

    Coordinates: 'ECF'
             RP: [6378137 0 10000]
             VP: [0 0 0]
```

```
                RT: [6378137 5000 10000]
                VT: [0 0 0]
              TALO: []
      EarthRadius: [6378137 6.3568e+006]
```

5. **XMLFile -** Geometry will be imported from a specified xml file as saved by APIstruct.struct2xml (see APIstruct). If the xml file has an element LUTFile with an h5 file name specified, the geometry will be loaded from the h5 file and the output will be an array of geometries. The default xml file location is the directory returned from ParamPath. One can optionally pass in *TALO*, an array of engagement times to project or interpolate the geometry specified in the xml file. If not input, will use value in the h5 file.

6. **Geom -** The inputs for modification of an existing geometry structure are

G [struct]	Geometry structure from GeomStruct.
hp [scalar]	Altitude of platform above curved earth surface (m).
ht [scalar]	Altitude of target above curved earth surface (m).
rd [scalar]	Ground range of target from platform along earth surface (m).
vp [scalar]	Speed of platform (m/s).
vt [scalar]	Speed of target (m/s).
PlatformHeading [scalar]	Heading clockwise from due north (deg).
TargetHeading [scalar]	Heading clockwise from due north (deg).

Inputs must be given in this order, but all inputs are not necessary. Inputs that are passed as [ ] will not be changed.

Example 2.3.5 (Geom) *This example shows how to change an existing geometry structure using* GeomStruct*.*

```
>> G = GeomStruct(G,[],10,0)

G =

    Coordinates: 'ECF'
             RP: [6.3560e+006 0 5.5236e+005]
             VP: [8.7156 0 -99.6195]
             RT: [6.3540e+006 0 5.5218e+005]
             VT: [0 20 0]
           TALO: []
    EarthRadius: [6378137 6.3568e+006]
```

Leave the platform altitude unchanged while changing the target altitude and ground range of an existing geometry structure.

For all geometries if the speed of the platform or target is not specified, velocities will be zero, as in Examples 2.3.1 and 2.3.3. If heading is not specified for the platform and/or the target, the heading will be 90 degrees (due East). The output is the geometry structure, *G*, as follows:

G [struct]	Geometry structure to be input to EngagementStruct or to update engagement case definition.
G.Coordinates [string]	Specifies the coordinate system used for defining the geometry - always 'ECF'.
G.RP [vector]	Platform position in ECF Coords (m).
G.VP [vector]	Platform velocity in ECF Coords (m/s).
G.RT [vector]	Target position in ECF Coords (m).
G.VT [vector]	Target velocity in ECF Coords (m/s).
G.TALO [empty]	Placeholder for target Time-After-Lift-Off.
G.EarthRadius [scalar/vector]	Earth radius used (m).

Note that the output is always in ECF coordinates.

2.3.1.2 EngagementStruct

Syntax

S = EngagementStruct(*G*)

```
SCALING_CODE_API int EngagementStruct(char** errorChain,
    const char geomType, const double* posPlat, const double* posTarg,
    const double* velPlat, const double* velTarg, const int nGeoms,
    const double* earthRadius, const int nEarthRadii, double* platAlt,
    double* targAlt, double* downRange, double* slantRange,
    double* platSpeed, double* targSpeed, double* v_ttp, double* v_ptp,
    double* v_ptt, double* azimuth, double* elevation,
    double* targIncidenceAngle, double* lineOfSightAngle)
```

This function will generate a MATLAB® structure of engagement parameters for custom scenarios, given platform position and velocity vectors and target position and velocity vectors in a defined coordinate system. The input can be a geometry structure from GeomStruct (in ECF or LLA coordinates) or a case code string [refer to SHaRE and/or SCALE toolboxes, if available]. If *G* is not in ECF coordinates, it will be converted to ECF.

The output is a structure, *S*, called the engagement structure which contains the engagement parameters for the defined path.

S [struct]	Structure containing engagement parameters.
S.G [struct]	Engagement geometry definition.
S.TALO [scalar]	Time after lift off (s).

S.hp [scalar]	Altitude of platform (m).
S.ht [scalar]	Altitude of target (m).
S.rd [scalar]	Range from platform to target along earth surface (m).
S.L [scalar]	Slant range from platform to target (m).
S.vp [scalar]	Platform speed (m/s).
S.vt [scalar]	Target speed (m/s).
S.V_TTP [scalar]	Path-transverse target velocity, in target "parallel" direction (m/s).
S.V_PTP [scalar]	Path-transverse platform velocity, in target "parallel" direction (m/s).
S.V_PTT [scalar]	Path-transverse platform velocity, in target "transverse" direction (m/s).
S.az [scalar]	Azimuth of target relative to platform (rad).
S.el [scalar]	Elevation of target relative to platform (rad).
S.tia [scalar]	Target incidence angle (rad).
S.LOSangle [scalar]	Magnitude of the full angle between platform velocity and propagation vector (rad).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages, deepest error is first.
geomType [in]	The type of coordinates input, 'L' for LLA or 'E' for ECF.
posPlat [in]	Platform position in either ECF or LLA coordinates; [x1,y1,z1,...,zN,yN,zN] where n = *nGeoms*; must be size 3 x *nGeoms*.
posTarg [in]	Target position in either ECF or LLA coordinates; must be size 3×*nGeoms*.
velPlat [in]	Velocity of platform in either ECF or LLA coordinates; must be size 3×*nGeoms*.
velTarg [in]	Velocity of target in either ECF or LLA coordinates; must be size 3×*nGeoms*.
nGeoms [in]	Number of independent geometry specifications input.
earthRadius [in]	Either radius pairs for an oblate earth model or scalar for spherical model, one for each input geometry; should be size *nEarthRadii* ×*nGeoms*

nEarthRadii [in]	Must be either 1 for spherical earth or 2 for oblate.
platAlt [out]	Altitude of platform (m); length *nGeoms*.
targAlt [out]	Altitude of target (m); length *nGeoms*.
downRange [out]	Range from platform to target along earth surface (m); length *nGeoms*.
slantRange [out]	Slant range from platform to target (m); length *nGeoms*.
platSpeed [out]	Platform speed (m/s); length *nGeoms*.
targSpeed [out]	Target speed (m/s); length *nGeoms*.
v_ttp [out]	Path-transverse target velocity, in target "parallel" direction (m/s); length *nGeoms*.
v_ptp [out]	Path-transverse platform velocity, in target "parallel" direction (m/s); length *nGeoms*.
v_ptt [out]	Path-transverse platform velocity, in target "transverse" direction (m/s); length *nGeoms*.
azimuth [out]	Azimuth of target relative to platform (rad); length *nGeoms*.
elevation [out]	Elevation of target relative to platform (rad); length *nGeoms*.
targIncidenceAngle [out]	Target incidence angle (rad); length *nGeoms*.
lineOfSightAngle [out]	Magnitude of the full angle between platform velocity and propagation vector (rad); length *nGeoms*.

S.G is the input geometry structure in ECF coordinates. Example 2.3.6 shows how to create an engagement structure.

Example 2.3.6 (Engagement structure) *This example shows how to create an engagement structure. G is the LLA geometry structure from Example 2.3.2.*

```
>> S = EngagementStruct(G)

S =

           G: [1x1 struct]
        TALO: []
          hp: 2.0000e+003
          ht: 10.0000
          rd: 5.4940e+005
           L: 5.5289e+005
          vp: 100
          vt: 20
       V_TTP: 20
       V_PTP: 0
       V_PTT: -4.7210
```

```
            az: 0
            el: -0.0472
           tia: 0
     LOSangle: 0.0472

>> S.G

ans =

    Coordinates: 'ECF'
             RP: [6.2838e+006 0 1.1006e+006]
             VP: [17.3648 0 -98.4808]
             RT: [6.3540e+006 0 5.5218e+005]
             VT: [0 20 0]
           TALO: []
    EarthRadius: [6378137 6.3568e+006]
```

With the ECF vectors for position ($\mathbf{R}$) and velocity ($\mathbf{V}$) of both platform (P) and target (T), the engagement parameters are computed using the equations below [1].

- Down range along the Earth's surface (rd) assuming a spherical Earth with radius R_e.

$$\texttt{rd} = R_e \cos^{-1}\left(\frac{\mathbf{R}_P \bullet \mathbf{R}_T}{|\mathbf{R}_P|\,|\mathbf{R}_T|}\right)$$

- Slant range (L).

$$\texttt{L} = |\mathbf{L}| = |\mathbf{R}_T - \mathbf{R}_P|$$

- Platform speed (vp).

$$\texttt{vp} = |\mathbf{V}_P|$$

- Target speed (vt).

$$\texttt{vt} = |\mathbf{V}_T|$$

- Path-transverse target velocity, in the target "parallel" direction (V_TTP). The platform, target, and wind velocities are rotated into a coordinate system such that the target has only one velocity component transverse to the propagation direction – the target "parallel" direction.

$$\texttt{V\_TTP} = \left|\mathbf{V}_T - \frac{[(\mathbf{R}_T - \mathbf{R}_P) \bullet \mathbf{V}_T]\,(\mathbf{R}_T - \mathbf{R}_P)}{\texttt{L}^2}\right|$$

- Path-transverse platform velocity, in the target "parallel" direction (V_PTP).

$$\begin{aligned}\hat{\rho}_P &= \frac{1}{\texttt{V\_TTP}}\left[\mathbf{V}_T - \frac{[(\mathbf{R}_T - \mathbf{R}_P) \bullet \mathbf{V}_T]\,(\mathbf{R}_T - \mathbf{R}_P)}{\texttt{L}^2}\right] \\ \texttt{V\_PTP} &= \mathbf{V}_P \bullet \hat{\rho}_P\end{aligned}$$

- Path-transverse platform velocity, in the target "transverse" direction (V_PTT).

$$\begin{aligned}\hat{\rho}_T &= \frac{\hat{\rho}_P \times (\mathbf{R}_T - \mathbf{R}_P)}{\texttt{L}} \\ \texttt{V\_PTT} &= \mathbf{V}_P \bullet \hat{\rho}_T\end{aligned}$$

- Azimuth angle (`az`) to the target (measured clockwise from platform heading, ϕ_p).

$$\begin{aligned} \mathbf{K}_{xy} &= \frac{\mathbf{L}_{xy}}{\mathtt{L}} \\ \mathbf{K}_{H_{xy}} &= \mathbf{K}_{xy} - (\mathbf{K}_{xy} \bullet \hat{\mathbf{z}})\,\hat{\mathbf{z}} \\ \mathtt{az} &= \frac{\mathbf{K}_{H_{xy}} \bullet \hat{\mathbf{x}}}{|\mathbf{K}_{H_{xy}} \bullet \hat{\mathbf{x}}|} \cos^{-1}\left(\frac{\mathbf{K}_{H_{xy}} \bullet \hat{\mathbf{y}}}{|\mathbf{K}_{H_{xy}}|}\right) - \phi_p \end{aligned}$$

  where $\mathbf{K}_{H_{xy}}$ is the projection of $\mathbf{K}_H$ into the xy plane (not a unit vector), $\mathbf{L}_{xy}$ is the vector $\mathbf{L}$ (in ECF coordinates) rotated into the aircraft coordinate frame (xy) with $\hat{x}$ directed east, $\hat{y}$ to the north, and the $\hat{z}$ vertical using the rotation matrix,

$$R = \begin{bmatrix} -\sin\Phi_A & \cos\Phi_A & 0 \\ -\cos\Theta_A\cos\Phi_A & -\cos\Theta_A\sin\Phi_A & \sin\Theta_A \\ \sin\Theta_A\cos\Phi_A & \sin\Theta_A\sin\Phi_A & \cos\Theta_A \end{bmatrix}$$

  Φ_A is the latitude and Θ_A is 90 minus the longitude.

- Elevation angle (`el`) to the target (measured from the local horizon of the platform).

$$\mathtt{el} = \frac{\pi}{2} - \cos^{-1}(\mathbf{K}_{xy} \bullet \hat{\mathbf{z}})$$

- Target incidence angle (`tia`) – angle between the propagation vector and the normal to the target surface (90 degrees minus the angle between the propagation vector ($\mathbf{R}_T - \mathbf{R}_P$) and the target velocity vector).

$$\mathtt{tia} = \frac{\pi}{2} - \cos^{-1}\left(-\frac{(\mathbf{R}_T - \mathbf{R}_P) \bullet \mathbf{V}_T}{\mathtt{L} \cdot \mathtt{vt}}\right)$$

- Line of sight angle (`LOSangle`) – full angle between platform velocity vector and propagation vector

$$\mathtt{LOSangle} = \cos^{-1}\left(\frac{(\mathbf{R}_T - \mathbf{R}_P) \bullet \mathbf{V}_P}{\mathtt{L} \cdot \mathtt{vp}}\right)$$

2.3.2 Propagation Geometry

The functions in this section provide the user extended geometry tools for laser weapon applications.

2.3.2.1 ChangeRd

Syntax

RNew = ChangeRd(*RPecf, RTecf, [EarthRadius], rd*)

RNew = ChangeRd(*Geom, rd*)

```
SCALING_CODE_API int ChangeRd(char** errorChain, const char geomType,
    const double* posPlat, const double* posTarg, const int nGeoms,
    const double* earthRadius, int nEarthRadii, int nEarthSpecs,
    const double* rd, const int nRd, double* newPosPlat)
```

This function changes the down range along a spherical earth surface between two ECF position vectors. The platform will be moved to a new location corresponding to the desired down range, ***rd***, while keeping the platform altitude fixed. The Matlab inputs are as follows:

Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list (*RPecf*, *RTecf*, *EarthRadius*, ...).
RPecf [vector]	Platform ECF position (m).
RTecf [vector]	Target ECF position (m).
EarthRadius [string]	(Optional) Earth radius to use (m). Defaults to the mean Earth radius.
rd [scalar]	Desired down range along the spherical Earth surface (m).

Geom as a structure from GeomStruct can be in any supported coordinates, in which case it will be converted to ECF. *Geom* as a list must be specified in ECF coordinates. The output is the new platform ECF position. If *Geom* is input as a structure *RNew* will be a new geometry structure with the updated platform position. If *Geom* is input as a list, *RNew* will be a vector. To move the target instead of the platform, use ReverseGeom.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages, deepest error is first.
geomType [in]	The type of coordinates input, 'L' for LLA or 'E' for ECF.
posPlat [in]	Position of platform in *geomType* coords (m); must be length 3×*nGeoms* with leading dimension length 3.
posTarg [in]	Position of target in *geomType* coords (m); must be length 3×*nGeoms* with leading dimension length 3.
nGeoms [in]	Number of geometry specifications to compute heights over; must be *nRd* if *nRd* ¿ 1.
earthRadius [in]	(NULL OK) Spherical or oblate earth radius (m); if passed in must be length *nEarthRadii*×*nGeoms* with *nEarthRadii* being the length of the leading dimension; the default value is the mean earth radius.
nEarthSpecs [in]	Must be 1 or equal to *nGeoms* or *nRd* to represent the number of earth radius specifications input.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate; *earthRadius* length will be *nEarthRadii*×*nEarthSpecs*.
rd [in]	Desired down range along the spherical Earth surface (m); must be length *nRd*.
nRd [in]	Number of down range specifications; must be *nGeoms* if *nGeoms* ¿ 1.

newPosPlat [out]	New platform position vector in *geomType* coordinates (m); Must be 3×*nGeoms* OR 3×*nRd*.

If $\mathbf{V}$ and $\mathbf{W}$ are the position vectors of the platform and target respectively in earth centered coordinates, then the angle between these vectors is $\theta = \cos^{-1}(\widehat{\mathbf{V}} \bullet \widehat{\mathbf{W}})$, where $\bullet$ is the dot product operator and $\widehat{\cdot}$ indicates unit vector. The angle by which the target position must be rotated to give a down range of r_d is

$$\alpha = \frac{r_d}{r_e} - \theta$$

where r_e is the earth radius as specified by *EarthRadius* or, in the case of an oblate earth, as computed using the function LocalEarthRadius. A new coordinate frame can be set up such that the x-direction is $\widehat{\mathbf{W}}$, the z-direction is normal to the plane formed by $\mathbf{V}$ and $\mathbf{W}$, $\widehat{\mathbf{n}} = \widehat{\mathbf{V}} \times \widehat{\mathbf{W}}$, and the y-direction completes the right-hand coordinate frame, $\widehat{\mathbf{z}} = \widehat{\mathbf{x}} \times \widehat{\mathbf{y}}$. The target and platform position vectors can then projected into the new coordinate frames using the projection operator

$$\begin{aligned}
\mathbf{V}' &= (\mathbf{V} \bullet \widehat{\mathbf{x}}) \cdot \widehat{\mathbf{x}} + (\mathbf{V} \bullet \widehat{\mathbf{y}}) \cdot \widehat{\mathbf{y}} + (\mathbf{V} \bullet \widehat{\mathbf{x}}) \cdot \widehat{\mathbf{x}} \\
&= \mathbf{A} \cdot \mathbf{V} \\
\mathbf{U}' &= (\mathbf{U} \bullet \widehat{\mathbf{x}}) \cdot \widehat{\mathbf{x}} + (\mathbf{U} \bullet \widehat{\mathbf{y}}) \cdot \widehat{\mathbf{y}} + (\mathbf{U} \bullet \widehat{\mathbf{x}}) \cdot \widehat{\mathbf{x}} \\
&= \mathbf{A} \cdot \mathbf{U}
\end{aligned}$$

where $\mathbf{A}$ is the matrix formed by

$$\mathbf{A} = \begin{bmatrix} \widehat{x}_i & \widehat{x}_j & \widehat{x}_k \\ \widehat{y}_i & \widehat{y}_j & \widehat{y}_k \\ \widehat{z}_i & \widehat{z}_j & \widehat{z}_k \end{bmatrix}$$

Note that $\mathbf{A}$ is an orthogonal matrix so $\mathbf{A}^{\mathrm{T}} = \mathbf{A}^{-1}$. The operator for rotation about the z-axis can is

$$\mathbf{R} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and the new platform position vector, $\mathbf{U}$, can be computed using

$$\begin{aligned}
\mathbf{U}' &= \mathbf{R} \cdot \mathbf{V}' \\
\mathbf{A} \cdot \mathbf{U} &= \mathbf{R} \cdot \mathbf{A} \cdot \mathbf{V} \\
\mathbf{U} &= \mathbf{A}^{\mathrm{T}} \cdot \mathbf{R} \cdot \mathbf{A} \cdot \mathbf{V}
\end{aligned}$$

Example 2.3.7 (Change down range) *This example shows how to use* ChangeRd *to change the platform position based on a desired down range.*

```
>> GNew = ChangeRd(GeomStruct,25e3)
```

Compute default geometry from GeomStruct *at 25 km ground range, moving the platform.*

```
>> GNew = ReverseGeom(ChangeRd(ReverseGeom(GeomStruct),25e3));
```

Compute default geometry from GeomStruct *at 25 km ground range, using* ReverseGeom *so that the target is moved.*

```
>> G = GeomStruct;
>> RNew = ChangeRd(G.RP,G.RT,[20:10:50]*1e3)
```

Compute new platform position for several ranges.

```
>> re = [PhysicalConst('reequator') PhysicalConst('repole')];
>> RNew = ChangeRd(G.RP,G.RT,re,[20:10:50]*1e3)
```

Compute new platform position for several ranges assuming oblate earth.

2.3.2.2 CLOUDFREEHT

Syntax

htEngage = CLOUDFREEHT(*hp, rd, [CloudFreeAlt], [EarthRadius]*)

```
SCALING_CODE_API int CloudFreeHT(char** errorChain, const double* hp,
    const int nHp, const double* rd, const int nRd, const double* cfAlt,
    int nCfAlt, const double* re, int nRe, double* cfHt)
```

This function computes the target altitude given platform altitude at which there is a clear line of sight from the platform to the target. The Matlab inputs are:

hp [vector]	Platform altitude (m).
rd [vector]	Down range along a spherical Earth (m).
CloudFreeAlt [scalar]	(Optional) Altitude of cloud tops (m) - defaults to 10000 m.
EarthRadius [scalar]	(Optional) Earth radius to use (m) - assumes spherical earth and defaults to WGS84 mean earth radius.

The output ***htEngage*** is the target altitude for cloud-free engagement in meters. Given a platform altitude, h_p, spherical earth radius, r_e, the altitude of the cloud tops h_c, and down range r_d, the minimum target altitude for clear line-of-sight is computed as detailed below. To calculate the minimum target altitude, we must calculate the angles shown in Figure 2.3 but replacing the value of r_e with $r_e + h_c$. The first angle, ϕ, is the angle from the platform position to the position where a line connecting the target to the platform is tangent to the top of the cloud deck, as specified by ***CloudFreeAlt***. The second angle, Δ, is the angle from the position where a line connecting the target to the platform is tangent to the top of the cloud deck and the target position.

$$\begin{aligned} \phi &= \cos^{-1}\left(\frac{r_e + h_c}{h_p + r_e}\right) \\ \Delta &= \frac{r_d}{r_e} - \phi \\ &= \cos^{-1}\left(\frac{r_e + h_c}{r_e + h_t^{\min}}\right) \end{aligned}$$

where $h_t^{\min}$ is the minimum target altitude for clear line of sight. The equality for Δ can be solved to find $h_t^{\min}$.

$$h_t^{\min} = \begin{cases} h_c & r_d \le r_d^{\max} \\ \frac{r_e + h_c}{\cos\Delta} - r_e & r_d > r_d^{\max} \end{cases}$$

where $r_d^{\max} = r_e \cdot \phi$ is down range from the platform to the point where the line connecting the target to the platform is tangent to the top of the cloud deck.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages, deepest error is first.
hp [in]	Platform altitude range (m); length *nHp*.
nHp [in]	Number of elements in vector hp; must be 1 or size *N* where *N* = max(*nHp*,*nRd*).
rd [in]	Down range (m); length *nRd*.
nRd [in]	Number of elements in vector rd; must be 1 or size *N*.
cfAlt [in]	Altitude of cloud tops (m); length *nCfAlt*.
nCfAlt [in]	Number of elements in vector *cfAlt*; must be 1 or size *N*.
re [in]	(NULL OK) Spherical earth radius (m); Defaults to WGS84 mean earth radius; length *nRe*.
nRe [in]	Number of elements in vector re; must be 1 or size *N*.
cfHt [out]	Cloud free target altitude (m); must allocate length *N*.

2.3.2.3 CommonSites

Syntax

[LLA, SiteInfo] = CommonSites(*SiteID*)

This function returns an LLA location vector for common sites. If *SiteID* is unspecified, i.e. CommonSites, available sites are displayed in the MATLAB® command window. Also returns a cell array of available sites when *SiteID* is unspecified. Thus, cell2Excel(CommonSites) produces an Excel spreadsheet of available common sites. The altitude returned is consistent with that returned from GrdAltProfile using 3-arcsecond resolution DTED data. The input is

SiteID [string]	String identifier for site. If *SiteID* is not recognized, will return the location for 'MZA-DAY'.

The outputs are

LLA [vector]	[lat long alt] of specified site.
SiteInfo [cell]	Information about the specified site stored in a cell array of {'SITE ID', 'NAME', 'LOCATION', 'TIMEZONEID'}.

Table 2.1 shows the available sites.

Site ID	Name	Location
2-MILE	2-Mile Site	SOR, Kirtland AFB, NM
2KM	2 km Site S. of Lamberson hangar	Kirtland AFB, NM
ABD	Aux. Beam Director	SOR, Kirtland AFB, NM
AFIT	AF Inst. of Tech.	Wright-Patt AFB, OH
BECK	Beck Site	WSMR, NM
CETFAC	Dahlgren CETFAC	Dahlgren, VA
CROWS-NEST	Crow nest Lamberson hangar	Kirtland AFB, NM
ELTF	Environmental Laser Test Facility	Kirtland AFB, NM
HANGAR760	2km Profiler Setup	Kirtland AFB, NM
HELSTF	HELSTF	WSMR, NM
KAFB-1KM	Arroyo 1 km S of Lamberson hangar	Kirtland AFB, NM
KAFB-4.5KM	Berm 4.5 km S of Lamberson hangar	Kirtland AFB, NM
KAFB-5.8KM	Berm 5.8 km S of Lamberson hangar	Kirtland AFB, NM
KAFB-6.7KM	Berm 6.7 km S of Lamberson hangar	Kirtland AFB, NM
KAFB-7.6KM	Berm 7.6 km S of Lamberson hangar	Kirtland AFB, NM
MILLER	Miller's Watch Site	WSMR, NM
MINE	Mine Site	WSMR, NM
MZA-ABQ	MZA Associates Corp.	Albuquerque, NM
MZA-DAY	MZA Associates Corp.	Dayton, OH
NDEC	Dahlgren NDEC	Dahlgren, VA
NOP	North Oscura Peak	WSMR, NM
RACHL	RACHL Site	SOR, Kirtland AFB, NM
SALINAS	Salinas Peak	WSMR, NM
SANDIA	North Sandia Peak	Albuquerque, NM
SULF	SULF Site	WSMR, NM
UD-CPC	U of Dayton (CPC 5th floor)	Dayton, OH
VA-HOSPITAL	VA hospital roof (10th floor)	Dayton, OH
WP-B620	WPAFB Bldg. 620 (11th floor)	Wright-Patt AFB, OH
WP-B622	WPAFB Bldg. 622	Wright-Patt AFB, OH
WP-RUNB	WPAFB Area B Runway	Wright-Patt AFB, OH

Table 2.1: List of currently available sites accessible through CommonSites. The first column is the value of the input *SiteID*.

2.3.2.4 EarthAlt

Syntax

[h, L, el] = EarthAlt(*x, Geom*)

[h, L, el] = EarthAlt(*x, hp, ht, rd, [re]*)

```
SCALING_CODE_API int EarthAlt(char** errorChain, const double* x,
    const int nX, const double* hp, const double* ht, const double* rd,
    const double* re, const int nGeoms, double* h, double* l,
    double* el)
```

This function computes the altitude above the surface of a spherical earth for a ray going from ***hp*** to ***ht*** at downrange ***rd*** for normalized position ***x*** (***x*** = 0 at ***hp***, ***x*** = 1 at ***ht***). The Matlab inputs are:

x [vector]	Normalized path position from platform (0) to target (1).
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma-separated list of (..., *hp*, *ht*, *rd*, [*re*]).
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along curved earth surface (m).
re [scalar]	(Optional) Spherical Earth radius (m). If not passed, uses mean Earth radius. If passed as a vector (as for Geodetic Earth) uses the mean value. Not required for geometry structure input. When re is passed in as a 2x1 and length(G) (number of geometries) is also 2, re will be treated as spherical radii, one for each geometry. To force use of re as geoid, pass in as NGx2.

Note that if geometry is input as a list, the earth model is assumed to be spherical using the mean earth radius. If the geometry is input as a structure, an oblate earth model can be used and altitudes will be above the oblate earth surface.

The output *h* is a vector the size of *x* with the altitude at various positions along the path in meters. The function also returns the slant range (*L*) in meters and elevation angle (*el*) in radians.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Normalized path position from platform (0) to target (1); must be length *nX*×*nGeoms* with the leading dimension being length *nX*.
nX [in]	Number of elements along path to compute altitude at.
hp [in]	Altitude of transmit/receive platform (m); must be length *nGeoms*.
ht [in]	Altitude of target (m); must be length *nGeoms*.
rd [in]	Downrange of target along curved earth surface (m); must be length *nGeoms*.
re [in]	(NULL OK) Spherical earth radius (m); if passed in must be length *nGeoms*; the default value is the mean earth radius.
nGeoms [in]	Number of geometry specifications to compute heights over.
h [out]	Altitude above earth surface for position *x* (m); must be length *nX* x *nGeoms* with the leading dimension being length *nX*.

`l` [out]	Slant range of each engagement (m); must be length `nGeoms`.
`el` [out]	Elevation angle of each engagement (rad); must be length `nGeoms`.

Given the platform position, $\mathbf{R}_p$, and target position, $\mathbf{R}_t$, in earth centered coordinates, the unit vector from the platform to the target is $\widehat{\mathbf{K}} = (\mathbf{R}_t - \mathbf{R}_p)/L$, where L is the slant range. Slant range and elevation angles are computed from these position vectors as described in EngagementStruct. Given the input normalized path position, x, the earth centered coordinates of the i^{th} position along the path can be found by

$$\mathbf{R}_p^{(i)} = \mathbf{R}_p + \widehat{\mathbf{K}} \cdot L \cdot x^{(i)}$$

Converting each of these position vectors to latitude, longitude, altitude format using ecf2LLA yields the altitude of each position along the path.

Given the altitude of the platform, h_p, and target, h_t, along with the down range along the curved earth surface, r_d, and the earth radius, R_e, the angle between the platform and the target is $\theta = r_d/R_e$. The slant range, L, and elevation angle, ϕ,m are then given by

$$\begin{aligned} L &= \sqrt{[(R_e + h_t)\sin\theta]^2 + [(R_e + h_t)\cos\theta - (R_e + h_p)]^2} \\ \phi &= \tan^{-1}\left\{\frac{R_e + h_t)\cos\theta - (R_e + h_p)}{(R_e + h_t)\sin\theta}\right\} \end{aligned}$$

where $\tan^{-1}$ is the four quadrant inverse tangent, `atan2` in Matlab. Given the input normalized path position, x, the earth centered coordinates of the i^{th} position along the path can be found by

$$h_p^{(i)} = \sqrt{(L \cdot x^{(i)} \cdot \cos\phi)^2 + (R_e + h_p + L \cdot x^{(i)} \sin\phi)^2 - R_e}$$

Example 2.3.8 (Calculation of `h`, `L`, and `el`) *This example shows how to calculate the altitude along a path, slant range and elevation angle using* EarthAlt.

```
>> [h,L,el] = EarthAlt([0 .25 .5 .75 1],1000,10,20000,PhysicalConst('re'))

h =

  1.0e+003 *

    1.0000    0.7466    0.4972    0.2516    0.0100


L =

  2.0026e+004


el =

   -0.0510
```

Calculation with 5 equally spaced points along the path from `hp` *= 1000 m to* `ht` *= 10 m.*

```
>> [h,L,el] = EarthAlt([0.1:0.2:0.9]',G)
```

```
h =

  898.1747
  696.4073
  497.1513
  300.4069
  106.1743


L =

  2.0026e+004


el =

   -0.0510
```

Calculation using a geometry structure where `G` *is the structure from Example 2.2.1.*

2.3.2.5 ECF2ECI

Syntax

G = ECF2ECI(*Gecf, [eRate]*)

[R, V] = ECF2ECI(*TALO, Recf, [Vecf], [eRate]*)

```
SCALING_CODE_API int ECF2ECI(char** errorChain, const double* talo,
     const double* posEcf, const double* velEcf, const int nCoords,
     const double* earthRotRate, double* posEci, double* velEci)
```

This function is used to convert position and velocity in earth-center-fixed (ECF) coordinates, in which the coordinate frame is fixed to the rotating earth, to earth-center-inertial (ECI) coordinate system, in which the coordinate frame is fixed at a specific time. The Matlab inputs are:

Gecf [struct]	ECF geometry structure or comma-separated list of (*TALO*, *Recf*, [*Vecf*], ...).
TALO [vector]	Column vector (Nx1) of the sample times (sec).
Recf [Nx3 matrix]	N 3-element ECF position vectors (m).
Vecf [Nx3 matrix]	(Optional) N 3-element ECF velocity vectors (m/s).
eRate [scalar]	(Optional) Earth rotation rate (rad/s). Must pass in *Vecf* (can be []) to specify earth rotation rate.

The outputs are

`G` [struct]	If `Gecf` is a structure, the output `G` will be a structure with position and velocity vectors in ECI coordinates.
`R` [Nx3 matrix]	ECI position vectors (m).
`V` [Nx3 matrix]	ECI velocity vectors (m/s).

The API function returns system error codes and the arguments are:

`errorChain` [in,out]	Holds error and warning messages, deepest error is first.
`talo` [in]	Column vector of sample times (sec); must have `nCoords` elements.
`posEcf` [in]	ECF position vectors (m); should be ordered as [x1 y1 z1 ... xn yn zn] where `n` = `nCoords` and the total size of the array is 3 x `nCoords`.
`velEcf` [in]	(NULL OK) ECF velocity vector (m/s); if used should be the same size as `posEcf`.
`nCoords` [in]	Number elements in `talo` and number of 3-element vectors in `posEci`, `velEci`, `posEcf`, and `velEcf`.
`earthRotRate` [in]	(NULL OK) Earth Rotation Rate (rad/s); assumed to be a single element; defaults to standard PhysicalConstStruct value.
`posEci` [out]	ECI position vector (m); must be the same size as `posEcf`.
`velEci` [out]	(NULL OK IF `velEcf`=NULL) ECI velocity vector (m/s); must be the same size as `velEcf`.

Given an ECF position vector $\mathbf{R}_f$, the conversion from ECF to ECI is a simple rotation about the z-axis

$$\mathbf{R}_i = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \mathbf{R}_f$$

where $\theta = \dot{\alpha} \cdot t$, $\dot{\alpha}$ is the earth rotation rate as specified by `eRate`, and t is the time since $t = 0$. The conversion of velocity vector, $\mathbf{V}_f$, can be found using the relation $\mathbf{V}_i = \dot{\mathbf{R}}_i$ and the chain rule for differentiation

$$\begin{aligned} \mathbf{V}_i &= \dot{\mathbf{A}} \cdot \mathbf{R}_f + \mathbf{A}\dot{\mathbf{R}}_f \\ &= \begin{bmatrix} -\dot{\alpha}\sin\theta & -\dot{\alpha}\cos\theta & 0 \\ \dot{\alpha}\cos\theta & -\dot{\alpha}\sin\theta & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \mathbf{R}_f + \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \mathbf{V}_f \end{aligned}$$

where $\mathbf{A}$ is the rotation matrix.

2.3.2.6 ECF2ENU

Syntax

G = ECF2ENU(*RefLLA, Gecf*)

[R, V] = ECF2ENU(*RefLLA, Recf, [Vecf], [EarthModel]*)

```
SCALING_CODE_API int ECF2ENU(char** errorChain,
     const double* refPosLla, const int nRefCoords, const double* posEcf,
     const double* velEcf, int nCoords, const double* earthRadius,
     int nEarthSpecs, int nEarthRadii, double* posEnu, double* velEnu)
```

This function converts position and velocity vectors from earth-center-fixed (ECF) coordinates, in which the coordinate frame is fixed to the rotating earth, to the local East-North-Up (ENU) coordinate system based on a fixed reference location [see Section 2.3.1.1 for a discussion of ECF and ENU coordinates]. The Matlab inputs are:

RefLLA [vector]	3-element LLA position vectors [Lat Long Alt] for reference location. Lat is the Latitude in degrees (-90 to 90), Long is the longitude positions in degrees (-180 to 180), and Alt is the altitude (m).
Gecf [struct/list]	Geometry parameters. Can be a structure (or array of structures) from GEOMSTRUCT or a comma separated list of (..., *Recf*, [*Vecf*], [*EarthModel*]).
Recf [Nx3 matrix]	N 3-element ECF position vectors (m).
Vecf [Nx3 matrix]	(Optional) N 3-element ECF velocity vectors (m/s).
EarthModel [string/numeric]	(Optional) If string, Geodetic (default) or Geometric. Use Geometric for a spherical earth and Geodetic for an oblate earth. If numeric and scalar - spherical Earth radius, if vector - [a b], where a is the equatorial radius and b is the polar radius. Must pass in *Vecf* (can be []) to specify earth model.

The Matlab outputs are:

G [struct]	If *Gecf* is a structure, *G* will be a structure with position and velocity vectors in ENU coordinates.
R [Nx3 matrix]	ENU position vectors (m).
V [Nx3 matrix]	ENU velocity vectors (m/s).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages, deepest error is first.
refPosLla [in]	Series of 3-element LLA position vectors [Lat1 Long1 Alt1 ... LatN LonN AltN] for reference location; length can be either 3 x 1 or 3 x *nCoords*.
nRefCoords [in]	Number of coordinates in *refPosLla*; should be either 1 or *nCoords*.
posEcf [in]	Series of 3-element ECF position vectors (m); should be ordered as [x1 y1 z1 ... xn yn zn] where n = *nCoords* and the total size of the array is 3 x *nCoords*.
velEcf [in]	(NULL OK) Series of 3-element ECF velocity vectors (m/s); if used should be the same size as *posEcf*.
nCoords [in]	Number of 3-element vectors in *posEcf*, *velEcf*, *posEnu*, and *velEnu*.
earthRadius [in]	Numeric and scalar if spherical Earth radius, or vector [a b] where *a* is the equatorial radius and *b* is the polar radius; also permitted to input *nCoords* radii all of the same type, i.e. [a1 a2 ... an] or [a1 b1 a2 b2 ... an] where n=*nCoords*.
nEarthSpecs [in]	Must be 1 or equal to *nCoords* to represent the number of earth radius specifications input.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate; *earthRadius* length will be *nEarthRadii* x *nEarthSpecs*.
posEnu [out]	ENU position vector (m); must be the same size as *posEcf*.
velEnu [out]	(NULL OK if *velEcf*=NULL) ENU velocity vector (m/s); must be the same size as *velEcf*.

Given position of the reference location $[\gamma, \phi, h_r]$, where γ is the latitude in degrees (negative for south latitudes), ϕ is the longitude in degrees (negative for west latitudes), and h_r is the altitude in meters, the reference position in ECF coordinates, $\mathbf{R}_r$, is determined using LLA2ECF. Then the vector (in ECF coordinates) pointing from the input ECF position vector, $\mathbf{R}_e$, to $\mathbf{R}_r$ is $\mathbf{R} = \mathbf{R}_e - \mathbf{R}_r = [x_R, y_R, z_R]$. The ENU position can then be found by rotating $\mathbf{R}$ about the ECF z-axis by the angle equal to the 90 plus reference longitude, ϕ, then rotating into the ECF xy-plane by an angle $\pi/2 - \gamma'$ [see Figure 2.1(c)], where

$$\gamma' = \tan^{-1}\left[\frac{\sqrt{x_R^2 + y_R^2}}{z_R}\right]$$

is the angle between the ECF xy-plane and $\mathbf{R}_e$, which for a spherical earth is equal to the latitude, γ. The ENU vector, $\mathbf{R}_n$, is given by

$$\mathbf{R}_n = \begin{bmatrix} -\sin\gamma & \cos\gamma & 0 \\ -\sin\phi\cos\gamma & -\sin\phi\sin\gamma & \cos\phi \\ \cos\phi\cos\gamma & \cos\phi\sin\gamma & \sin\phi \end{bmatrix} \cdot \mathbf{R}_e$$

Likewise, the ECF velocity vector, $\mathbf{V}_e$, is also converted to ENU via the same rotation operator

$$\mathbf{V}_n = \begin{bmatrix} -\sin\gamma & \cos\gamma & 0 \\ -\sin\phi\cos\gamma & -\sin\phi\sin\gamma & \cos\phi \\ \cos\phi\cos\gamma & \cos\phi\sin\gamma & \sin\phi \end{bmatrix} \cdot \mathbf{V}_e$$

2.3.2.7 ECF2LLA

Syntax

G = ECF2LLA(*Gecf*)

[R, V] = ECF2LLA(*Recf, [Vecf], [EarthModel]*)

```
SCALING_CODE_API int ECF2LLA(char** errorChain, const double* posEcf,
    const double* velEcf, const int nCoords, const double* earthRadius,
    int nEarthSpecs, int nEarthRadii, double* posLla, double* velLla)
```

This function converts position and velocity vectors in earth-centered-fixed (ECF) coordinates, in which the coordinate frame is fixed to the rotating earth, to a latitude, longitude, altitude (LLA), heading, velocity description assuming a spherical or oblate Earth [see Section 2.3.1.1 for a discussion of LLA and ECF coordinates]. The Matlab inputs are:

Gecf [struct/list]	Geometry parameters. Can be a structure (or array of structures) from GEOMSTRUCT or a comma separated list of (*Recf*, [*Vecf*], [*EarthModel*]).
Recf [Nx3 matrix]	N 3-element ecf position vectors (m).
Vecf [Nx3 matrix]	(Optional) N 3-element ecf velocity vectors (m/s).
EarthModel [string/numeric]	(Optional) If string, Geometric (default) or Geodetic. Use Geometric for a spherical earth and Geodetic for an oblate earth. If numeric and scalar - spherical Earth radius, if vector - [a b], where a is the equatorial radius and b is the polar radius. Must pass in *Vecf* (can be []) to specify earth model.

Notice *EarthModel* can be either a string representing the earth model to use or a numeric value for the earth radius. For conversion additional information is needed which depends on the specified *EarthModel*. For a geometric model, only the earth radius is needed, but for a geodetic model, one needs to specify the radius of earth at the equator (semi-major axis of ellipse, a) and the flattening, $f = (a\text{-}b)/a$, where b is the semi-minor axis (all of which can be extracted from PHYSICALCONST). Figure 2.2 illustrates the difference in latitude and altitude between the two models for the same point (the scale has been exaggerated). The geodetic altitude and latitude (γ) are indicated by the subscript 'G.'

Conversion from a cartesian coordinate reference frame to geodetic latitude and ellipsoidal altitude requires more elaborate treatments (geodetic and geocentric longitudes are identical). Common solutions use iterative techniques; and while several exact solvers have existed for just as long, their implementations span from simple to complex. Borkowski's treatment is particularly well suited for code implementation [2] and is premised on using a fourth-order polynomial expression to calculate the reduced latitude [3]. Computationally, this calculation starts with the radial distance

$$p = \sqrt{X^2 + Y^2}$$

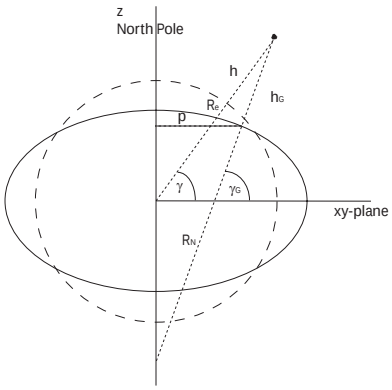


Figure 2.2: Illustration of the difference between Geometric and Geodetic earth models. Geometric latitude and altitude are γ and h, respectively and γ_G and h_G are the Geodetic latitude and altitude, respectively.

which, along with the semi minor and major radii, a and b, is the input to a series of intermediate calculations

$$\begin{aligned}
E &= \frac{bZ - \left(a^2 - b^2\right)}{ap} \\
F &= \frac{bZ + \left(a^2 - b^2\right)}{ap} \\
P &= \frac{4\left(EF + 1\right)}{3} \\
Q &= 2\left(E^2 - F^2\right) \\
D &= P^3 + Q^2 \\
\nu &= \left(D^{1/2} - Q\right)^{1/3} - \left(D^{1/2} + Q\right)^{1/3} \\
G &= \frac{1}{2}\left(E + \sqrt{E^2 + \nu}\right) \\
t &= \sqrt{G^2 + \frac{F - \nu G}{2G - E}} - G
\end{aligned}$$

The latitude is found by

$$\gamma_G = \tan^{-1}\left(a\frac{1 - t^2}{2bt}\right)$$

And the ellipsoidal height is

$$h = (r - at)\cos(\gamma_G) + (Z - b)\sin(\gamma_G)$$

In the unlikely case where $D < 0$, which applies to locations less than 45km from the Earth's center, ν should be calculated using

$$\nu = 2\sqrt{-P}\cos\left(\frac{1}{3}\cos^{-1}\left(\frac{Q}{P\sqrt{-P}}\right)\right)$$

which will prevent complex numbers. Additionally, in order to obtain the correct sign — this process does use a forth-order polynomial — the sign of b is set to that of Z.

If `Vecf` is specified, it will be converted to a velocity heading description and this conversion depends on position. `Vecf` must be converted to a coordinate system where $z-$axis is aligned with the normal to the surface and the $x-$ and $y-$axes lie on a plane tangent to the surface at the point where the normal intersects the surface with the $y-$axis pointing North. The transformation is given by [see Section 2.3.1.2]

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} -\sin(\phi_{ecf}) & \cos(\phi_{ecf}) & 0 \\ -\cos(\theta_{ecf})\cos(\phi_{ecf}) & -\cos(\theta_{ecf})\sin(\phi_{ecf}) & \sin(\theta_{ecf}) \\ \sin(\theta_{ecf})\cos(\phi_{ecf}) & \sin(\theta_{ecf})\sin(\phi_{ecf}) & \cos(\theta_{ecf}) \end{bmatrix} \begin{bmatrix} V_{Xecf} \\ V_{Yecf} \\ V_{Zecf} \end{bmatrix}$$

where ϕ_{ecf} and θ_{ecf} are spherical coordinates for ***Recf***, ϕ is measured in the $xy-$plane from the $+x-$axis and theta is measure from the $z-$axis [$\phi_{ecf} = \gamma$ and $\theta_{ecf} = 90 - \phi$. The components of the output ***V*** are then calculated by

$$\begin{aligned} \text{xy-Velocity, } v_{p1} &= \sqrt{v_x^2 + v_y^2} \\ \text{Heading, } \phi_{xy} &= \frac{v_y}{\sqrt{v_x^2 + v_y^2}} \\ \text{Vertical Velocity, } v_{p2} &= v_z \end{aligned}$$

If the input ***Gecf*** is a structure or an array of structures from GEOMSTRUCT, the output will be a new geometry structure or array of structures with all position and velocity vectors in LLA coordinates and field ***Coordinates*** = 'LLA.' If ***Gecf*** is input as a list, the outputs are N 3-element LLA position vectors ***R*** = [γ (deg) ϕ (deg) h (m)] and N 3-element LLA velocity vector ***V*** = [v_{p1} (m/s) ϕ_{xy} (deg) v_{p2} (m/s)] where N is the number of input position and velocity vectors in the input matrices.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages, deepest error is first.
posEcf [in]	Series of 3-element ECF positions one after another, that is [x y z x y z ...] (m).
velEcf [in]	(NULL OK) Series of 3-element ECF velocity vectors, same size as ***posEcf*** (m/s).
nCoords [in]	Number of coordinate tuples such that size of input vectors will be 3 x ***nCoords***.
earthRadius [in]	(NULL OK) Numeric and scalar if spherical Earth radius, or vector [a b] where a is the equatorial radius and b is the polar radius; also permitted to input ***nCoords*** radii all of the same type, i.e. [a1 a2 ... an] or [a1 b1 a2 b2 ... an] where n=***nCoords***. Defaults to WGS84 mean earth radius.
nEarthSpecs [in]	Must be 1 or equal to ***nCoords*** to represent the number of earth radius specifications input.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate; ***earthRadius*** length will be ***nEarthRadii*** x ***nEarthSpecs***.
posLla [out]	LLA Series of 3-element position vectors, same size and ordering as ***posEcf***, i.e. [Lat (deg) Long (deg) Alt (m) ...].
velLla [out]	(NULL OK if ***velEcf***=NULL) Series of LLA 3-element velocity vectors, i.e. [xy-Velocity (m/s) Heading (deg) vertical Velocity (m/s) ...].

Example 2.3.9 (Conversion from ECF to LLA) *This example shows how to use* ECF2LLA *to convert from earth-centered fixed coordinates to latitude longitude altitude, velocity and heading.*

```
>> [R,V] = ecf2LLA([0 PhysicalConst('reequator') 0],[40 0 40],'Geodetic')

R =

     0    90     0


V =

   56.5685  -45.0000         0
```

Platform at Lat = 0, Long = 90 on the surface of the earth in a geodetic (oblate) earth model with a speed of 56.5685 m/s and heading of -45 degrees with level flight.

```
>> Glla = ecf2LLA(G)

Glla =

    Coordinates: 'LLA'
             RP: [-0.1799 0 1000]
             VP: [100 90 0]
             RT: [0 0 10]
             VT: [10 180 0]
           TALO: []
    EarthRadius: 6.3710e+006
```

Conversion of geometry structure from ECF to LLA. `G` is the geometry structure from Example 2.2.2.

2.3.2.8 ECI2ECF

Syntax

G = ECI2ECF(*Geci*)

[R, V] = ECI2ECF(*TALO, Reci, [Veci], [eRate]*)

```
SCALING_CODE_API int ECI2ECF(char** errorChain, const double* talo,
     const double* posEci, const double* velEci, const int nCoords,
     const double* earthRotRate, double* posEcf, double* velEcf)
```

This function is used to convert position and velocity vectors in earth-center-inertial (ECI) coordinates, in which the coordinate frame is fixed at a specific time, to earth-center-fixed (ECF) coordinate system, in which the coordinate frame is fixed to the rotating earth. The Matlab inputs are:

Geci [struct]	ECI geometry structure or comma-separated list of (*TALO*, *Reci*, [*Veci*], [*eRate*]).
TALO [vector]	Column vector (Nx1) of the sample times (sec).
Reci [Nx3 matrix]	N 3-element ECI position vectors (m).

Veci [Nx3 matrix]	(Optional) N 3-element ECI velocity vectors (m/s).
eRate [scalar]	(Optional) Earth rotation rate (rad/s). Must pass in *Veci* (can be []) to specify earth rotation rate.

The Matlab output is:

G [struct]	If *Geci* is a structure, the output *G* will be a structure with position and velocity vectors in ECF coordinates.
R [Nx3 matrix]	ECF position vectors (m).
V [Nx3 matrix]	ECF velocity vectors (m/s).

The conversion from ECI to ECF can found found by using the function ECF2ECI with a negative earth rotation rate.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages, deepest error is first.
talo [in]	Column vector of the sample time (sec); must have *nCoords* elements.
posEci [in]	ECI position vectors (m); should be ordered as [x1 y1 z1 ...xn yn zn] where n = *nCoords* and the total size of the array is 3 x *nCoords*.
velEci [in]	(NULL OK) ECI velocity vector (m/s); if used should be the same size as *posEci*.
nCoords [in]	Number elements in *talo* and number of 3-element vectors in *posEci*, *velEci*, *posEcf*, and *velEcf*.
earthRotRate [in]	(NULL OK) Earth Rotation Rate (rad/s); assumed to be a single element; defaults to standard *PhysicalConstStruct* value.
posEcf [out]	3-element ECF position vector (m); must be the same size as *posEci*.
velEcf [out]	(NULL OK IF *velEci* =NULL) 3-element ECF velocity vector (m/s); must be the same size as *velEci*.

2.3.2.9 ENU2ECF

Syntax

G = ENU2ECF(*Genu*)

[R, V] = ENU2ECF(*RefLLA, Renu, [Venu], [EarthModel]*)

```
SCALING_CODE_API int ENU2ECF(char** errorChain,
     const double* refPosLla, const int nRefCoords, const double* posEnu,
     const double* velEnu, const int nCoords, const double* earthRadius,
     int nEarthSpecs, int nEarthRadii, double* posEcf, double* velEcf)
```

This function converts local East-North-Up (ENU) geometry specification relative to a fixed location on the earth to earth-center-fixed (ECF) coordinate system, in which the coordinate frame is fixed to the rotating earth [see Section 2.3.1.1 for a discussion of ENU and ECF coordinates]. The Matlab inputs are:

Genu [struct/list]	Geometry parameters. Can be a structure (or array of structures) from GEOMSTRUCT or a comma separated list of (*RefLLA*, *Renu*, [*Venu*], [*EarthModel*]).
RefLLA [vector]	3-element LLA position vectors [Lat Long Alt] for reference location. Lat is the Latitude in degrees (-90 to 90), Long is the longitude positions in degrees (-180 to 180), and Alt is the altitude (m).
Renu [Nx3 matrix]	N 3-element ENU position vectors (m).
Venu [Nx3 matrix]	(Optional) N 3-element ENU velocity vectors (m/s).
EarthModel [string/numeric]	(Optional) If string, Geodetic (default) or Geometric. Use Geometric for a spherical earth and Geodetic for an oblate earth. If numeric and scalar - spherical Earth radius, if vector - [a b], where a is the equatorial radius and b is the polar radius. Must pass in *Venu* (can be []) to specify earth model.

The Matlab outputs are:

G [struct]	If *Genu* is a structure, *G* will be a structure with position and velocity vectors in ECF coordinates.
R [Nx3 matrix]	ECF position vectors (m).
V [Nx3 matrix]	ECF velocity vectors (m/s).

The conversion from ENU to ECF follows directly from the derivation in ECF2ENU and noting the properties of rotation matrices, namely $\mathbf{A}^{-1}(\alpha) = \mathbf{A}(-\alpha) = \mathbf{A}^{-\mathrm{T}}(\alpha)$.

$$\mathbf{R}_e = \begin{bmatrix} -\sin\gamma & \cos\gamma & 0 \\ -\sin\phi\cos\gamma & -\sin\phi\sin\gamma & \cos\phi \\ \cos\phi\cos\gamma & \cos\phi\sin\gamma & \sin\phi \end{bmatrix}^{\mathrm{T}} \cdot \mathbf{R}_n$$

and

$$\mathbf{V}_e = \begin{bmatrix} -\sin\gamma & \cos\gamma & 0 \\ -\sin\phi\cos\gamma & -\sin\phi\sin\gamma & \cos\phi \\ \cos\phi\cos\gamma & \cos\phi\sin\gamma & \sin\phi \end{bmatrix}^{\mathrm{T}} \cdot \mathbf{V}_n$$

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages, deepest error is first.
refPosLla [in]	Series of 3-element LLA position vectors [Lat1 Long1 Alt1 ... LatN LonN AltN] for reference location; length can be either 3 x 1 or 3 x *nCoords*.
nRefCoords [in]	Number of coordinates in *refPosLla*; must be either 1 or *nCoords*.
posEnu [in]	Series of 3-element ENU position vectors (m); should be ordered as [E1 N1 U1 ... En Nn Un] where n = *nCoords* and the total size of the array is 3 x *nCoords*.
velEnu [in]	(NULL OK) Series of 3-element ENU velocity vectors (m/s); if used should be the same size as *posEcf*.
nCoords [in]	Number of 3-element vectors in *posEcf*, *velEcf*, *posLla*, and *velLla*.
earthRadius [in]	(NULL OK) Numeric and scalar if spherical Earth radius, or vector [a b] where a is the equatorial radius and b is the polar radius; also permitted to input *nCoords* radii all of the same type, i.e. [a1 a2 ... an] or [a1 b1 a2 b2 ... an] where n=*nCoords*; default value is mean spherical earth radius from *PhysicalConst*.
nEarthSpecs [in]	Must be 1 or equal to *nCoords* to represent the number of earth radius specifications input.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate; *earthRadius* length will be *nEarthRadii* x *nEarthSpecs*.
posEcf [out]	ECF position vector (m); must be the same size as *posEnu*.
velEcf [out]	(NULL OK if *velEnu*=NULL) ECF velocity vector (m/s); must be the same size as *velEnu*.

2.3.2.10 FacetNorms

Syntax

Norm = FacetNorms(*vertex, facet*)

```
SCALING_CODE_API int FacetNorms(char** errorChain,
    const double* vertices, const int nVertices, const int* facets,
    const int nFacets, const int nVerticesPerFacet,
    double* facetNormals)
```

This function calculates the normal of each facet from a list of vertex points and associated facets. The inputs are

vertex [array]	Vertex points.
facet [array]	Facet list.

The output is an array of facet normals. The size of the array of vertex points should be the total number of vertices by 3. The size of the facet list array should be of size number of facets by number of vertices per facet. The output array of facet normals will then be of size number of facets by 3.

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
vertices [in]	Points in three dimensions where object vertices are located; stored with each coordinate together, i.e. [v1x, v1y, v1z, v2x, v2y, v2z, ..., vNx, vNy, vNz] where N = *nVertices*.
nVertices [in]	Number of three-element coordinates in vertices; the vertices array must have 3×*nVertices* elements.
facets [in]	Zero-based index indicating which vertices make up a facet (i.e. the flat face of the object); each facet must have at least three non-collinear vertices, but it could be more (only the first three are considered in this function); must be stored such that indices belonging to the same facet are together; the total array length is *nFacets*×*nVerticesPerFacet* with *nVerticesPerFacet* being the length of the leading dimension.
nFacets [in]	Number of facets to find the normal for.
nVerticesPerFacet [in]	Number of vertices that make up a facet; must be three or more and only the first three are considered.
facetNormals [out]	Vectors in three dimensions normal to each facet; will be length *nFacets*×3; stored in the same way as vertices.

All facets will have at least 3 vertices and all vertices will be co-planar. To find the unit vector that is normal to the surface of the plane defined by the facet vertices, use the cross product of the two vectors lying in the same plane as the vertices. If the n^{th} vertex is given by $v^{(n)} = [v_x^{(n)}, v_y^{(n)}, v_z^{(n)}]$, then we can form two vectors $\mathbf{U}$ and $\mathbf{V}$ in the facet plane

$$\begin{aligned} \mathbf{U} &= \left(v_x^{(i)} - v_x^{(j)}\right)\hat{x} + \left(v_y^{(i)} - v_y^{(j)}\right)\hat{y} + \left(v_z^{(i)} - v_z^{(j)}\right)\hat{x} \\ \mathbf{V} &= \left(v_x^{(k)} - v_x^{(j)}\right)\hat{x} + \left(v_y^{(k)} - v_y^{(j)}\right)\hat{y} + \left(v_z^{(k)} - v_z^{(j)}\right)\hat{x} \end{aligned}$$

where i, j, and k are any combination of three vertex points for the facet. Then the surface normal can be found by

$$\widehat{\mathbf{N}} = \widehat{\mathbf{U}} \times \widehat{\mathbf{V}}$$

where $\widehat{\cdot}$ denotes a unit vector.

2.3.2.11 FlyoutGeom

Syntax

G = FlyoutGeom(*C, TALO, [RPFlag], [trackStr]*)

```
SCALING_CODE_API int FlyoutGeom(char** errorChain,
    const double* requestedTime, const int nRequestedTimes,
    const double* geomValidTime, const int movePlatform,
    const double* posPlat, const double* posTarg, const double* velPlat,
    const double* velTarg, const int nGeoms, const double* earthRadius,
    int nEarthRadii, double* newPosPlat, double* newPosTarg,
    double* newVelPlat, double* newVelTarg)
```

FlyoutGeom returns a structure or an array of structures with geometry information for position and velocity of a target as a function of time. The inputs are

C [struct]	Engagement geometry structure. Can be an engagement structure from ABLCase [see SCALE User's Guide], LinkCase [see SHaRE User's Guide], or EngagementStruct, a geometry structure from GeomStruct, a target structure with a Flyout substructure, or a scalar Flyout structure.
TALO [vector]	Time after lift off (sec).
RPFlag [logical]	(Optional) Flag for repositioning the platform based on *VP* and *TALO*. If true, platform will be moved assuming the position at *TALO(1)* is set by the value in *C.G.RP*. Default is false unless the input is an array of geometry structures.
trackStr [string]	(Optional) Identifier for how the track is computed, either 'gc' for great circle (default) or 'rl' for rhumb line. Only used when trajectory type is altitude and down range polynomials or a single geometry structure.

Position and velocity information for the platform will be taken from input *C.G*. Target information will be taken from *C.Targ* unless *C.Targ.Flyout* is empty, in which case target information will be taken from *C.G* and constant velocity (speed and heading) will be assumed.

The API function outputs system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
requestedTime [in]	Points in time (with respect to *geomValidTime*) that geometry is requested (s); must be length *nRequestedTimes*.
nRequestedTimes [in]	Length of *requestedTime* vector.
geomValidTime [in]	(NULL OK) Points in time that the geometry points (specified next) are associated with (s), defaults to 0.0; must be length *nGeoms*; for interpolation mode must be sorted.
movePlatform [in]	If true platform moves according to velocity and requested time; if false only the target is moved.
posPlat [in]	(NULL OK) Platform position in ECF coordinates (m); [x1,y1,z1,...,zN,yN,zN] where N = *nGeoms*; must be size 3×*nGeoms*.
posTarg [in]	(NULL OK) Target position in ECF coordinates (m); must be size 3×*nGeoms*.
velPlat [in]	Velocity of platform in ECF coordinates (m/s); must be size 3×*nGeoms*.
velTarg [in]	Velocity of target in ECF coordinates (m/s); must be size 3×*nGeoms*.
nGeoms [in]	Number of geometry specifications input; if equal to 1, then forward reckoning is used for the requested times; otherwise, interpolation between *geomValidTime* is used.
earthRadius [in]	(NULL OK) Numeric and scalar if spherical Earth radius, or 2-element vector [a b] where a is the equatorial radius and b is the polar radius; is mean spherical earth radius from PHYSICALCONST.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate.
newPosPlat [out]	(NULL OK) Platform positions evaluated at *requestedTime*; size is 3×*nRequestedTime*.
newPosTarg [out]	(NULL OK) Target positions evaluated at *requestedTime*.
newVelPlat [out]	Platform velocity positions evaluated at *requestedTime*.
newVelTarg [out]	Target velocity positions evaluated at *requestedTime*.

This function supports three trajectory "types" in the Targ.Flyout structure:

- Polynomials for altitude and down range from an initial point at a given direction; values are computed using the Matlab function `polyval`. Supports great circle or rhumb-line (constant heading) trajectories as specified by ***trackStr***. Velocities are computed using the derivatives of the polynomials via the Matlab functions `polyval` and `polyder`. For great circle trajectories, the function FWRECKON is used to compute positions and headings as a function of time

$$[\gamma(t), \phi(t), \alpha(t)] = \texttt{FWreckon}\left(\gamma(t-\Delta t), \phi(t-\Delta t), \alpha(t-\Delta t), \Delta r_d(t-\Delta t)\right)$$

  where γ is the latitude, ϕ is the longitude, α is the heading, Δt is the time sampling interval, and Δr_d is the change in down range.

- Matrices of ECF positions and velocity as a function of time - positions and velocities are interpolated (linearly) at the input TALOs.

- Polynomial fit to ECF positions as a function of time - positions are computed from the polynomials and velocities are computed using the derivatives of the polynomials using the Matlab functions `polyval` and `polyder`.

The output is

G [struct]	Geometry structure, or array of structures.
G.Coordinates [string]	Specifies the coordinate system used for defining the geometry - always 'ECF'.
G.RP [vector]	Platform position in ECF Coords (m).
G.VP [vector]	Platform velocity in ECF Coords (m/s).
G.RT [vector]	Target position in ECF Coords (m).
G.VT [vector]	Target velocity in ECF Coords (m/s).
G.TALO [scalar]	Time-After-Lift-Off (s).
G.EarthRadius [string]	Specification of earth radius.

The input ***C*** can also be either a geometry structure or a target structure. In the event that a scalar geometry structure is input, constant speed and heading will be assumed for the target and, optionally, the platform. With a scalar geometry, the user can also specify the ***trackStr*** for the track type to use. If an array of geometry structures is input, positions and velocities are interpolated onto the new TALOs using linear interpolation of the ECF positions and velocities. This will return `NaN` for input TALOs outside the range of TALOs in the geometry structure. If a target structure is input, the output array of geometry structures will have empty fields ***RP*** and ***VP*** since information about the platform is unknown.

2.3.2.12 FWReckon

Syntax

[lat2, long2, BAZ] = FWReckon(*lat1, long1, FAZ, rd, [EarthRadius]*)

```
SCALING_CODE_API int FWReckon(char** errorChain, const double* lat,
    const int nLat, const double* lon, const int nLon,
    const double* fwAz, const int nFwAz, const double* downRange,
    const int nDownRange, const double* earthRadius, int nEarthSpecs,
    int nEarthRadii, double* newLat, double* newLon, double* bkAz)
```

This function computes a new LLA position given an LLA starting position, the forward Azimuth, and the down range using forward reckoning and great circles. The inputs are

lat1 [vector]	Starting latitude in degrees.
long1 [vector]	Starting longitude in degrees.
FAZ [vector]	Forward Azimuth in degrees (Azimuth from starting position to the new position).
rd [vector]	Desired down range (m).
EarthRadius [scalar/vector]	(Optional) Earth radius, single geometric earth radius (m) or 2 element vector, [Equatorial radius, Polar radius], of the geodetic earth radii (m). Defaults to mean earth radius from PhysicalConst.

The outputs are

lat2 [vector]	New latitude in degrees.
long2 [vector]	New longitude in degrees.
BAZ [vector]	Back azimuth in degrees (Azimuth from new position to starting position).

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lat [in]	Starting latitude in degrees.
nLat [in]	Length of *lat* vector; can be 1 or max(*nLat*,*nLon*,*nFwAz*,*nDownRange*,*nEarthSpecs*).

lon [in]	Starting longitude in degrees.
nLon [in]	Length of *lon* vector; can be 1 or max(*nLat*,*nLon*,*nFwAz*,*nDownRange*,*nEarthSpecs*).
fwAz [in]	Forward Azimuth in degrees (azimuth from starting position to the new position).
nFwAz [in]	Length of *fwAz* vector; can be 1 or max(*nLat*,*nLon*,*nFwAz*,*nDownRange*,*nEarthSpecs*).
downRange [in]	Desired down range (m).
nDownRange [in]	Length of *downRange* vector; can be 1 or max(*nLat*,*nLon*,*nFwAz*,*nDownRange*,*nEarthSpecs*).
earthRadius [in]	Numeric and scalar if spherical Earth and b is the polar radius; also permitted to input multiple radii all of the same type, i.e. [a1 a2 ... an] or [a1 b1 a2 b2 ... an] where n=max(*nLat*,*nLon*,*nFwAz*,*nDownRange*,*nEarthSpecs*).
nEarthSpecs [in]	Must be 1 or equal to max(*nLat*,*nLon*,*nFwAz*,*nDownRange*,*nEarthSpecs*) to represent the number of earth radius specifications input.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate; *earthRadius* length will be *nEarthRadii* × *nEarthSpecs*.
newLat [out]	New latitude in degrees; must be allocated to length max(*nLat*,*nLon*,*nFwAz*,*nDownRange*,*nEarthSpecs*).
newLon [out]	New longitude in degrees; must be allocated to length max(*nLat*,*nLon*,*nFwAz*,*nDownRange*,*nEarthSpecs*).
bkAz [out]	(NULL OK) Back Azimuth in degrees (azimuth from new position to starting position); must be allocated to length max(*nLat*,*nLon*,*nFwAz*,*nDownRange*,*nEarthSpecs*).

The algorithm used in this function can be found in [4]. Example 2.3.10 shows how to compute a new platform position relative to a given target location.

Example 2.3.10 (FWReckon) *Compute platform position* 30° *from North and 10 km ground range from a target with location 0 latitude and* 10° *longitude*

```
>> latT = 0; longT = 10; az = 30;   % all in degrees
>> re = PhysicalConst('remean');
>> [latP,longP,BAZ] = FWReckon(latT,longT,az,100000,re)

latP =

    0.7788
```

```
longP =

   10.4497

BAZ =

 -149.9969

>> G = GeomStruct('LLA',[latP, longP, 10000], [latT, longT, 5000], [], [], re);
```

You can verify the geometry by typing `S = EngagementStruct(G)` *and examining the output structure for ground range and azimuth. Note that the azimuth output by* EngagementStruct *is the azimuth from the platform (*`BAZ`*) where the* 30° *specified in the example was from the target to the platform (*`az`*).*

2.3.2.13 GrdAltProfile

Syntax

Alt = GrdAltProfile(*[AltMap], pLat, pLong, [pixBuff], [dirName], ... [dispMap]*)

Alt = GrdAltProfile(*[gAlt], [yLat], [xLong], pLat, pLong, ... [pixBuff], [dirName], [dispMap]*)

This function takes an altitude grid and Lat/Long vectors and Lat/Long points of interest and returns their respective altitudes. The inputs are

AltMap [struct]	(Optional) Structure of elevation data as from bilLoad or a comma-separated list of (*gAlt*, *yLat*, *xLong*, ...).
gAlt [array]	Vertical deviation from WGS84/EGM96 geoid (m).
yLat [vector]	Decimal latitude of *Alt* pixels' centers (deg).
xLong [vector]	Decimal longitude of *Alt* pixels' centers (deg).
pLat [vector]	Decimal latitude of points of interests (deg).
pLong [vector]	Decimal longitude of points of interests (deg).
pixBuff [scalar]	(Optional) Width of buffer (pixels). Defaults to 1.5.
dirName [string]	(Optional) Directory location of DTED data files. Defaults to EngagementTools/DTED.
dispMap [bool]	(Optional) Flag controlling whether image of the data is displayed. Default is false.

The input *AltMap* is optional. If not input, GrdAltProfile will determine which data is needed from the inputs *pLat* and *pLong* and loads all necessary data files.

The output is

Alt [vector]	Points of interests' vertical deviation from WGS84/EGM96 geoid (m).

2.3.2.14 IsGoodGeom

Syntax

t = IsGoodGeom(*Geom, [hGnd], [dirName], [dispMap]*)

This function returns true if the slant path from the platform to the target does not intersect the surface of the earth, as specified by the optional input ground altitude. The input can be either a geometry structure or a comma-separated list of geometry as described below.

Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list (*hp*, *ht*, *rd*, [*re*]).
hp [scalar/vector]	Altitude of transmit/receive platform (m).
ht [scalar/vector]	Altitude of target (m).
rd [scalar/vector]	Downrange of target along curved earth surface (m).
re [scalar/vector]	(Optional) Earth radius to use (m). Defaults to value from PhysicalConst. If passed in as a vector (as for Geodetic Earth) uses the mean of input Earth radius.
hGnd [vector/logical]	(Optional) Altitude of ground above mean sea level (m). Defaults to 0. Use logical true with *Geom* as a structure to load terrain data and check if path intersects terrain. If *Geom* is not a structure, *hGnd* must be scalar. Not required if specifying *dirName*.
dirName [string]	(Optional) Directory location of DTED data files for loading terrain data. Defaults to EngagementTools/DTED.
dispMap [boolean]	(Optional) Flag indicating whether to display image of terrain data. Default is false. Only used if using terrain data.

Given a platform altitude, h_p, spherical earth radius, r_e, and down range r_d, the minimum target altitude for clear line-of-sight is computed as detailed below. If the input target altitude is greater than this minimum altitude, IsGoodGeom returns true. To calculate the minimum target altitude, we must calculate the angles shown in Figure 2.3. The first angle, θ is the angle between the platform position and the target position.

$$\theta = \frac{r_d}{r_e}$$

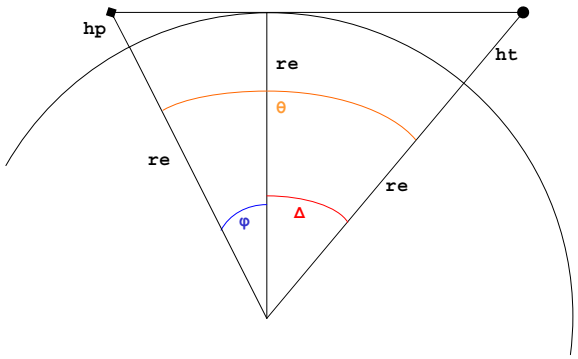


Figure 2.3: Illustration of angles used to calculate the minimum target altitude (*ht*) required for line-of-sight given platform altitude (*hp*), down range (*rd*), and earth radius (*re*).

The next angle ϕ is the angle from the platform position to the position where a line connecting the target to the platform is tangent to the surface of the earth.

$$\phi = \cos^{-1}\left(\frac{r_e + h_G}{h_p + r_e}\right)$$

where h_G is the altitude of the ground. Δ is the difference between these two angles, $(\theta - \phi)$. If Δ is less than zero and the target altitude is greater than the ground altitude, clear line-of-sight can always be established. If Δ is greater than 90°, the earth will block the view of the target. If Δ is greater than 0° and less than 90°, the following equation can be used to calculate the minimum target altitude required for clear line-of-sight.

$$h_t^{\min} = \frac{r_e}{\cos\Delta} - r_e$$

If ***ht*** is greater than $h_t^{\min}$ and ***hp*** is greater than the ground altitude, then the geometry is good.

2.3.2.15 IsGoodHitPoint

Syntax

I = IsGoodHitPoint(*G*, *Targ*)

This function determines whether the hit point on an object is close to the specified aimpoint given the geometry. The function compares *Targ.Object.hitPoint* (or *Targ.HELpt* if *Targ.Object.hitPoint* is empty) to the output of ObjectIncidence. The inputs are

G [struct]	Geometry structure as from GeomStruct.
Targ [struct]	Target structure as from TargStruct with object information.

The function will return false where the aimpoint is not within line of site from the platform, including the case where an aimpoint is specified internal to the object. If the input *Targ* structure does not contain object information, IsGoodHitPoint will return true. Otherwise, the function returns a logical array that is true if the hit point lies on a visible surface and false if the hit point is not visible.

2.3.2.16 LLA2ECF

Syntax

G = LLA2ECF(*Glla*)

[R, V] = LLA2ECF(*Rlla, [Vlla], [EarthModel]*)

```
SCALING_CODE_API int LLA2ECF(char** errorChain, const double* posLla,
    const double* velLla, const int nCoords, const double* earthRadius,
    int nEarthSpecs, int nEarthRadii, double* posEcf, double* velEcf)
```

This function converts from a latitude, longitude, altitude (LLA), speed, heading coordinate description to earth-centered-fixed coordinates, in which the coordinate frame is fixed to the rotating earth [see Section 2.3.1.1 for a discussion of LLA and ECF coordinates]. The inputs are

Glla [struct/list]	Geometry parameters. Can be a structure (or array of structures) from GEOMSTRUCT or a comma separated list of (*Rlla*, [*Vlla*], [*EarthModel*]).
Rlla [Nx3 matrix]	N 3-element LLA position vectors [Lat Long Alt]. Lat is the Latitude in degrees (-90 to 90), Long is the longitude positions in degrees (-180 to 180), and Alt is the altitude (m).
Vlla [Nx3 matrix]	(Optional) N 3-element LLA velocity vectors ([Vxy Heading Vz]). Vxy is the velocity in a cylindrical coordinate frame with the XY plane tangent to the Earth's surface (m/s). Heading is the heading in degrees clockwise (from North). Vz is the velocity in the vertical direction (m/s).
EarthModel [string/numeric]	(Optional) If string, Geometric (default) or Geodetic. Use Geometric for a spherical earth and Geodetic for an oblate earth. If numeric and scalar - spherical Earth radius, if vector - [a b], where a is the equatorial radius and b is the polar radius. Must pass in *Vlla* (can be []) to specify earth model.

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
posLla [in]	Series of 3-element LLA position vectors [Lat1 Long1 Alt1 ... LatN LongN AltN] (where N=*nCoords*) for reference location.
velLla [in]	(NULL OK) Series of LLA 3-element velocity vectors, i.e. [xy-Velocity (m/s) Heading (deg) vertical Velocity (m/s) ...].

nCoords [in]	Number of 3-element vectors in *posEcf*, *velEcf*, *posLla*, and *velLla*.
earthRadius [in]	(NULL OK) Numeric and scalar if spherical Earth radius, or vector [a b] where a is the equatorial radius and b is the polar radius; also permitted to input *nCoords* radii all of the same type, i.e. [a1 a2 ... an] or [a1 b1 a2 b2 ... an] where n=*nCoords*; default value is mean spherical earth radius from PHYSICALCONST.
nEarthSpecs [in]	Must be 1 or equal to *nCoords* to represent the number of earth radius specifications input.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate; *earthRadius* length will be *nEarthRadii* × *nEarthSpecs*.
posEcf [out]	Series of 3-element ECF position vectors (m); must be same size as *posLla*.
velEcf [out]	(NULL OK IF *velLla*=NULL) 3-element ECF velocity vector (m/s); must be same size as *velLla*.

Notice *EarthModel* can be either a string representing the earth model to use or a numeric value for the earth radius. See Section 2.3.2.7 for more information about the difference between Geometric and Geodetic earth models and a description of parameters involved. If the input is a geometry structure or array of structures, the output will be a new geometry structure or array of structures in which the position and velocity vectors have been converted to ECF coordinates as described below. If the input is in the form of a list, the outputs are N ECF position vectors, *R*, and, if *Vlla* is specified, N ECF velocity vectors, *V*, where N is the number of input position and velocity vectors in the input matrices. The components of *R* are given by

$$\begin{aligned} X &= (hp + R_N)\sin(\theta)\cos(\phi) \\ Y &= (hp + R_N)\sin(\theta)\sin(\phi) \\ Z &= [hp + R_N(1-e^2)]\cos(\theta) \end{aligned}$$

where θ is the zenith angle from the polar axis (90-latitude or 90-γ), ϕ is the azimuth angle (longitude), e is the eccentricity $e^2 = 2f - f^2$, f is the flattening $f = (r_{ee} - r_{ep})/r_{ee}$, r_{ee} is the radius of the earth at the equator, r_{eq} is the radius of the earth through the poles, and R_N is the radius of curvature at the prime vertical and is given by

$$R_N = \frac{r_{ee}}{\sqrt{1 - e^2\sin^2(\gamma)}}$$

For the Geometric model R_N is simply equal to r_{ee}. The components of *V* are given by

$$\begin{aligned} V_{Xecf} &= -v_{p1}\sin(\phi_{xy})\sin(\phi) - v_{p1}\cos(\phi_{xy})\cos(\phi)\cos(\theta) + v_{p2}\cos(\phi)\sin(\theta) \\ V_{Yecf} &= v_{p1}\sin(\phi_{xy})\cos(\phi) - v_{p1}\cos(\phi_{xy})\sin(\phi)\cos(\theta) + v_{p2}\sin(\phi)\sin(\theta) \\ V_{Zecf} &= v_{p1}\cos(\phi)\sin(\theta) + v_{p2}\cos(\theta) \end{aligned}$$

where ϕ_{xy} is the heading measured clockwise from true North, v_{p1} is the speed parallel to the surface and v_{p2} is the vertical speed.

Example 2.3.11 (Conversion from LLA to ECF) *This example shows how to use* LLA2ECF *to convert from latitude longitude altitude, velocity and heading to earth-centered fixed coordinates.*

```
>> [R,V] = LLA2ecf([0,0,1000],[],'Geometric')

R =

  1.0e+006 *

    6.3720         0         0


V =

     []
```

Convert an LLA position vector to ECF. Must pass `Vlla` *to specify earth model. Since no velocities were input, the output* `V` *is empty.*

```
>> [R,V] = LLA2ecf([10,0,1000],[100,90,0],'Geodetic')
```

Convert LLA position and velocity assuming geodetic earth model.

```
>> Glla = struct('Coordinates','LLA','RP',[0.25 0 1000], ...
           'RT',[0 0 10],'VP',[100 90 0],'VT',[10 0 0], ...
           'EarthRadius',PhysicalConst('remean'));
>> Gecf  = LLA2ecf(Glla);
```

Convert a geometry structure in LLA coordinates to ECF

2.3.2.17 LocalEarthRadius

Syntax

[re, NSre, EWre] = LocalEarthRadius(*G*)

[re, NSre, EWre] = LocalEarthRadius(*R1, [R2], [re]*)

```
SCALING_CODE_API int LocalEarthRadius(char** errorChain,
     const double* lla1, const double* lla2, const int nCoords,
     const double* earthRadius, int nEarthSpecs, int nEarthRadii,
     double* locEarthRadSp, double* locEarthRadNs, double* locEarthRadEw)
```

Computes the effective spherical Earth radius as function of latitude [5]. The inputs are

G [struct]	Geometry structure from GeomStruct or a comma-separated list of (*R1*, [*R2*], [*re*]).
R1 [Nx3 vector]	LLA position vector 1.
R2 [Nx3 vector]	(Optional) LLA position vector 2. Can be empty but must be passed if passing *re*. If not passed in or passed in as empty, *re* is (*NSre*+*EWre*)/2.

re [vector]	(Optional) 2 element vector, [Equatorial radius, Polar radius], of the geodetic earth radii (m). Defaults to [6378137, 6356752.314] from PHYSICALCONST.

The outputs are

re [vector]	Vector with same dimension as *R1* & *R2* with the effective spherical Earth radius (m).
NSre [vector]	Vector with same dimension as *R1* & *R2* with the effective North/South spherical Earth radius (m).
EWre [vector]	Vector with same dimension as *R1* & *R2* with the effective East/West spherical Earth radius (m).

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
llaA [in]	LLA position vector 1; takes the form [Lat1 Long1 Alt1 ... LatN LongN AltN] (where N=*nCoords*).
llaB [in]	(NULL OK) LLA position vector 2; if not passed in re is (*locEarthRadNs* + *locEarthRadEw*) / 2; must be same size as *llaA*.
nCoords [in]	The number of 3-element coordinates passed in; *llaA* must be length 3×*nCoords*.
earthRadius [in]	Numeric and scalar if spherical Earth radius, or vector [a b] where a is the equatorial radius and b is the polar radius; also permitted to input multiple radii all of the same type, i.e. [a1 a2 ... an] or [a1 b1 a2 b2 ... an] where n=*nCoords*.
nEarthSpecs [in]	Must be 1 or *nCoords* to represent the number of earth radius specifications input.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate; *earthRadius* length will be *nEarthRadii* ×*nEarthSpecs*.
locEarthRadSp [out]	Effective spherical Earth radius (m); must allocate length *nCoords*.
locEarthRadNs [out]	Effective north/south spherical Earth radius (m); must allocate length *nCoords*.

locEarthRadEw [out]	Effective east/west spherical Earth radius (m); must allocate length *nCoords*.

2.3.2.18 LOSANGLE

Syntax

[phiLOS, phiLOSaz, phiLOSel] = LOSANGLE(*G*)

```
SCALING_CODE_API int LOSANGLE(char** errorChain, const char geomType,
    const double* posPlat, const double* posTarg, const double* velPlat,
    const double* velTarg, const int nGeoms, const double* earthRadius,
    const int nEarthRadii, double* phiLos, double* phiAz, double* phiEl)
```

This function computes the full angle between the platform velocity vector and the propagation vector (line-of-sight) from platform to target given *G*, a geometry structure from GEOMSTRUCT. The outputs are

phiLOS [scalar]	Full angle between platform velocity and propagation vector from platform to target (rad).
phiLOSaz [scalar]	Azimuthal component of *phiLOS* (rad).
phiLOSel [scalar]	Elevation component of *phiLOS* (rad).

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
geomType [in]	The type of coordinates input, 'L' for LLA or ''E' for ECF.
posPlat [in]	Platform position in either ECF or LLA coordinates; [x1,y1,z1,...,zN,yN,zN] where n = *nGeoms*; must be size 3×*nGeoms*.
posTarg [in]	Target position in either ECF or LLA coordinates; must be size 3×*nGeoms*.
velPlat [in]	Velocity of platform in either ECF or LLA coordinates; must be size 3×*nGeoms*.
velTarg [in]	Velocity of target in either ECF or LLA coordinates; must be size 3×*nGeoms*.
nGeoms [in]	Number of independent geometry specifications input.

earthRadius [in]	Either radius pairs for an oblate earth model or scalar for spherical model, one for each input geometry; should be size *nEarthRadii* × *nGeoms*.
nEarthRadii [in]	Must be either 1 for spherical earth or 2 for oblate.
phiLos [out]	(NULL OK) Full angle between platform velocity and propagation vector from platform to target (rad); length *nGeoms*; if NULL this output won't be returned.
phiAz [out]	(NULL OK)
Azimuthal component of *phiLOS* (rad); length *nGeoms*; if NULL this output won't be returned. *phiEl* [out]	(NULL OK) Elevation component of *phiLOS* (rad); length *nGeoms*; if NULL this output won't be returned.

Equations for calculating these outputs can be found in Section 2.3.1.2.

2.3.2.19 ObjectIncidence

Syntax

[hitPt_PT, tia, incPos, tiaPT, facetIdx] = ObjectIncidence(*G, ... Targ, [hitPt], [dispPlot]*)

```
SCALING_CODE_API int ObjectIncidence(char** errorChain,
    const double* posPlatEcf, const double* posTargEcf,
    const double* velTargEcf, const double* earthRadius,
    const int nEarthRadii, const int nGeoms, const double* vertices,
    const int nVertices, const int* facets, const int nFacets,
    const int nVerticesPerFacet, const double* facetNormals,
    const double* objMotion, const double* objUp,
    const double* objRefLoc, const double* hitPoint, int hitPointDim,
    double* targIncAngle, double* incPos, double* targIncAnglePt,
    double* incPosPt, int* facetIdx, double* RayDir)
```

This function computes incidence angle for a given target and geometry. If *Targ* has a structure field *Object* containing facets and vertices, the incidence angle is computed based on those. Otherwise, the incidence angle is computed assuming a missile. The inputs are

G [struct]	Geometry structure from GeomStruct.
Targ [struct]	Target structure from TargStruct.
hitPt [array]	(Optional) Point on the target for which to compute incidence angle. If size NGeom x 2 it is assumed to be in target P/T coordinates. If size NGeom x 3 it is assumed to be in object coordinates. If not input, uses *Targ.Object.hitPoint* or *Targ.HELpt*.

dispPlot [logical/char]	(Optional) Flag indicating whether to plot the object with the incident ray orientation or an avi file name to write the plot(s) to an avi file. Default is false. Only displayed if *Targ.Object* exists and the ray is actually incident.

The outputs are

hitPt_PT [array]	Hit point in target P/T coordinates.
tia [vector]	Target incidence angle (rad).
incPos [array]	Position vector on the target at which the ray is incident in the target object (x,y,z) coordinate frame.
tiaPT [array]	Target incidence angle resolved in target P/T coordinate frame (rad).
facetIdx [array]	Index into *Targ.Object.facets* for the facet on which the ray is incident for each geometry.

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
posPlatEcf [in]	Platform position in ECF coordinates (m); [x1,y1,z1,...,zN,yN,zN] where n = *nGeoms*; must be size 3×*nGeoms*.
posTargEcf [in]	Target position in ECF coordinates (m); must be same size and layout as *posPlatEcf*.
velTargEcf [in]	Target velocity in ECF coordinates (m); must be same size and layout as *posPlatEcf*.
earthRadius [in]	Specify the earth radius belonging to each geometry; must be *nEarthRadii* x *nGeoms* with each radius belonging to one geometry specification consecutive.
nEarthRadii [in]	Specify 1 for spherical earth or 2 for oblate earth; must be 1 or 2.
nGeoms [in]	Number of geometries over which to compute the incidence angle.
vertices [in]	(NULL OK) Points in three dimensions where target's vertices are located; stored with each coordinate together, i.e. [v1x, v1y, v1z, v2x, v2y, v2z, ..., vNx, vNy, vNz] where N = *nVertices*; if NULL assumes an ideal missile target shape.

nVertices [in]	Number of three-element coordinates in vertices; the vertices array must have 3×*nVertices* elements; ignored if vertices or facets is NULL.
facets [in]	(NULL OK) Zero-based indices into the list of vertices which specifies the target's flat faces; must refer to at least 3 non-collinear points where additional points are allowed to define the polygon (exactly how many is specified with *nVerticesPerFacet*); must be size *nFacets*×*nVerticesPerFacet*; indices defining the same polygon are assumed to be consecutive in memory; if NULL assumes an ideal missile target shape.
nFacets [in]	The number of flat faces that make up the target; ignored if facets or vertices is NULL.
nVerticesPerFacet [in]	The number of vertices that make up one facet; should be 3 or more; ignored if vertices or facets is NULL.
facetNormals [in]	(NULL OK) If vectors normal to the facet surfaces are already computed, they can be passed in here to save time; if NULL is passed in then they are computed internally; if passed in must be size 3×*nFacets* stored in the same way as vertices; values from FacetNorms accepted; ignored if vertices or facets is NULL.
objMotion [in]	(NULL OK if vertices or facets is NULL) Unit vector specifying which direction in the vertices is to be associated with the target velocity; must be length 3.
objUp [in]	(NULL OK if vertices or facets is NULL) Unit vector specifying which direction in the target is "up" for purposes of determining the 2D target parallel/transverse coordinate bases; must be length 3.
objRefLoc [in]	(NULL OK) Point in target object space at which the *hitPoint* is in relation to; if input must be length 3; if NULL defaults to [0,0,0].
hitPoint [in]	(NULL OK) Point on the object from which to calculate the incidence angle; if NULL uses *objRefLoc*; must be length *hitPointDim*×*nGeoms* with each point's coordinates consecutive in memory.
hitPointDim [in]	Must be 2 or 3 specifying the length of each *hitPoint*; if 2 then the hit point is in target parallel/transverse coordinates; if 3 then the hit point is relative to the standard bases in target object space; ignored if *hitPoint* is NULL.
targIncAngle [out]	Target incidence angle (rad); relative to *facets* if facets are input; must be length *nGeoms*.
incPos [out]	Position vector on the target at which the ray is incident in the target object (x,y,z) coordinate frame; must be length 3×*nGeoms* where each point's coordinates are consecutive in memory.

targIncAnglePt [out]	Target incidence angle along target parallel and transverse axes (rad); must be length 2×*nGeoms*; stored [thetaP1, thetaT1, thetaP2, thetaT2, ..., thetaPn, thetaTn] where P is parallel and T is transverse and n is *nGeoms*.
incPosPt [out]	Hit point in target P/T coordinates; must be length 2×*nGeoms* stored similarly to *targIncAnglePt*.
facetIdx [out]	Facet index indicated the facet upon which the HEL is incident; NULL OK; otherwise allocate to length *nGeoms*.
RayDir [out]	Ray direction; NULL OK; otherwise allocate to length 3×*nGeoms*.

Example 2.3.12 (Compute incidence angle on an object) *This example shows how to call* ObjectIncidence *to compute the incidence angle on the target.*

```
>> Targ = TargStruct('cone',[0.5,2,4]);
>> [~,tia] = ObjectIncidence(GeomStruct,Targ,[0.25 0])
```

Compute angle of incidence for a point 0.25m from the tip in P/T coordinates (halfway down the cone).

```
>> [trkpt,tia] = ObjectIncidence(GeomStruct,Targ,Extract(Targ,'Object.trkPoint'))
```

Compute the object track point in target P/T coordinates and the angle of incidence.

2.3.2.20 ObjectIntersect

Syntax

[Inter, z, Ri, facet] = ObjectIntersect(*RayOrig, RayDir, ... v, f, fn, FacetCheck*)

```
SCALING_CODE_API int ObjectIntersect(char** errorChain,
    const double* rayOrg, const double* rayDir, const double* vertices,
    const int nVertices, const int* facets, const int nFacets,
    const int nVerticesPerFacet, const double* facetNormals,
    const int facetCheck, int* intersectFace, double* intersectPoint,
    double* distToFacet)
```

This function finds if and where a ray intersects an object. The inputs are

RayOrig [array]	Ray origin.
RayDir [array]	Ray direction.
v [array]	Location of object vertices.
f [array]	Facet list.

fn [array]	(Optional) Facet normals. If not input, these are computed internally.
FacetCheck [logical]	(Optional) Runs bounding box check for each *facet*.

The outputs are

Inter [logical]	Flag indicating whether an intersection occurs.
z [vector]	Distance to intersect for each facet intersected.
Ri [array]	Point of intersection for each facet.
facet [vector]	Facet(s) intercepted.

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
rayOrg [in]	A point from where a ray originates; must be length 3.
rayDir [in]	A unit vector indicating the ray's direction; must be length 3.
vertices [in]	Points in 3D, which together with the facets array, define the polygons that makes up the object; must be length 3×*nVertices*; stored [v1x, v1y, v1z, ..., vnx, vny, vnz] where n = *nVertices*.
nVertices [in]	The number of points in the vertices input.
facets [in]	Zero-based indices into the list of vertices defining a polygon in 3D; must refer to at least 3 non-collinear points where additional points are allowed to define the polygon (exactly how many is specified with *nVerticesPerFacet*); must be size *nFacets*×*nVerticesPerFacet*; indices defining the same polygon are assumed to be consecutive in memory.
nFacets [in]	The number of facets or polygons that make up the object.
nVerticesPerFacet [in]	The number of vertices that make up one facet.

facetNormals [in]	(NULL OK) If vectors normal to the facet surfaces are already computed, they can be passed in here to save time; if NULL is passed in then they are computed internally; if passed in must be size 3×*nFacets* stored in the same way as vertices; values from FacetNorms accepted.
facetCheck [in]	This flag can be used to improved performance if the right value is selected, but either way the result should be the same; set to 1 to place a bounding box around each facet and quickly eliminate some from possible intersection; set to 0 to avoid this check; if there are a lot of facets with few intersections, setting to 1 should be better.
intersectFace [out]	0 if no intersection takes place; 1 if the intersection is from the front and -1 if from the back (relative to *facetNormals*); must be length *nFacets*.
intersectPoint [out]	Points at which the ray intersects each facet that is intersected; must be size 3×*nFacets*; value undefined for facets that do not have an intersection (i.e. where *intersectFace* = 0).
distToFacet [out]	Euclidean distance from the ray origin the point at which the ray intersects the facet; must be length *nFacets*; values undefined where *intersectFace* = 0.

2.3.2.21 PhysicalConst

Syntax

const = PhysicalConst(*ConstName*)

```
SCALING_CODE_API int PhysicalConst( PhysicalConstStruct* outStruct)
```

This function returns the value of a physical constant given the name identifier. Table 2.2 shows a description of available constants, the input strings and the output values from [6].

PhysicalConst with no inputs and no more than one output will return a structure that contains all of the possible output values.

In the API, the function PhysicalConst returns a struct of custom type PhysicalConstStruct. This custom structure contains all the possible output values and each can be referenced using the same input strings as are used for the MATLAB® function.

Example 2.3.13 (PhysicalConst C++) *This example illustrates use of* PhysicalConst *in the C++ API.*

```
PhysicalConstStruct PCstruct;
int errorCode = PhysicalConst(&PCstruct);
double re = PCstruct.remean;
```

Use PhysicalConst *to get the value of the WGS84 mean earth radius.*

Description	Input String	Output Value
Avogadro's number	'avogadro'	6.022e23 (1/mol)
Conversion from degrees to radians	'deg2rad'	$\pi/180$
Ellipsoid (WGS84)	'ellipsoid'	[reequator eccentricity]
Geodetic earth radius (WGS84) (m)	'geoid'	[reequator repole]
Gravitational acceleration (m/s^2)	'gravityacc'	9.80665
Ideal gas constant (J/Kg/K)	'idealgas'	287.0583
Planck's constant (Js)	'planck'	6.626196×10^{-34}
Conversion from radians to degrees	'rad2deg'	$180/\pi$
ABLE Radius of earth[a]	're'	6378008
Equatorial radius (m) (WGS84)	'reequator'	6378137
Mean radius (m) (WGS84)	'remean'	6371009
Polar radius (m) (WGS84)	'repole'	$6378137(1 - 1/298.257223573)$
Flattening of the Earth [repole=reequator*(1-rf)] (WGS84)	'rf'	$1/298.257223573$
Earth rotation rate (rad/sec) (WGS84)	'rw'	$7292115.0 \times 10^{-11}$
Specific heat of dry air (J/Kg/K)	'specificheat'	1004.67
Speed of light (m/s) (WGS84)[b]	'speedoflight'	299792458
Structure with WGS72 parameters	'wgs72'	[structure]
von Karman constant	'vonkarman'	0.4

[a] As used by Bill Burckle and Paul Berger
[b] World Geodetic System 84

Table 2.2: Input strings and output values from PhysicalConst.

2.3.2.22 REVERSEGEOM

Syntax

G = REVERSEGEOM(*Gin*)

This function reverses the input geometry making the target position and velocity the platform position and velocity and vice versa. The only input is a geometry structure and the output is the reversed geometry structure.

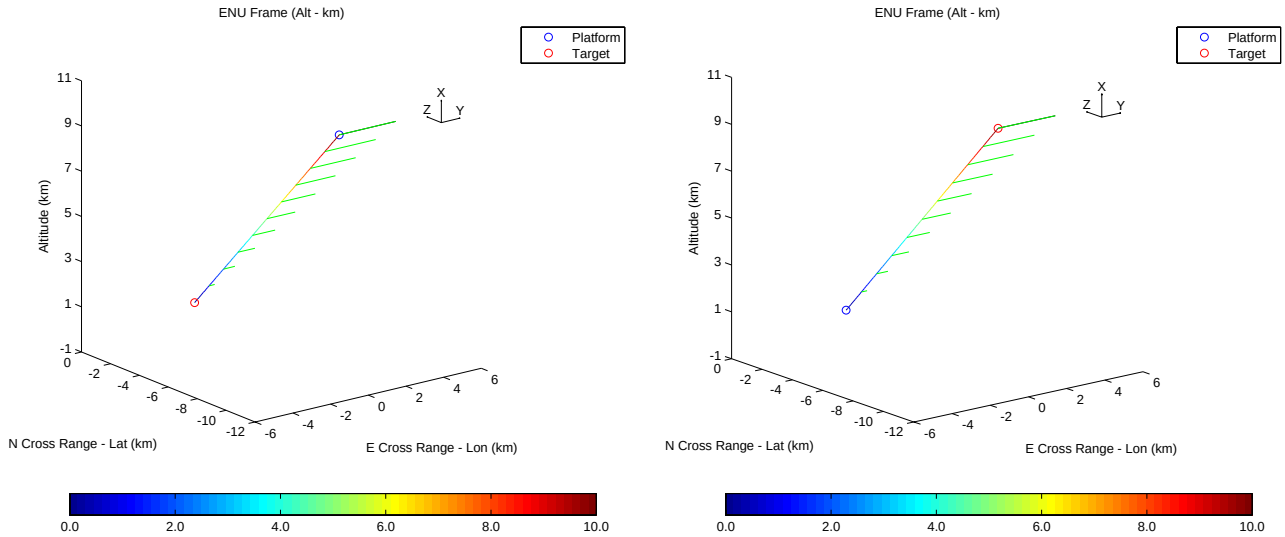
Example 2.3.14 (ReverseGeom) *This examples illustrates the functionality of* REVERSEGEOM.

```
>> Gin = GeomStruct('Simple',10000,0,10000,200,0);
>> G = ReverseGeom(Gin);
```

Create a geometry structure with a high-altitude moving platform and stationary ground target. Then reverse the geometry for a stationary, ground platform to a high-altitude moving target.

```
>> plotEng(Gin,'ENU'); plotEng(G,'ENU');
```

Plot the two geometries using PLOTENG *in 'ENU' units. Figure 2.4 shows the two geometry structures plotted. The only thing that changes is the direction of propagation under consideration.*



(a) (b)

Figure 2.4: Plots of geometries from Example 2.3.14, (a) `Gin` and (b) `G`.

2.3.2.23 RotateObject

Syntax

[Obj R] = RotateObject(*Obj0, upDir, VT*)

```
SCALING_CODE_API int RotateObject(char** errorChain,
     const double* vertices, const int nVertices, const double* motion,
     const double* up, double* newMotion, double* newUp,
     double* newVertices, double* rotation)
```

This function rotates an object as defined in Targ.Object from TargStruct based on a desired direction of motion and zenith direction. The inputs are

Obj0 [struct]	Original object structure with fields vertices and facets.
upDir [vector]	Desired direction for "up".
VT [vector]	Desired direction of motion.

The outputs are

Obj [struct]	Object structure with any computed rotation applied.
R [matrix]	Rotation matrix applied.

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
vertices [in]	Points in 3D, which together with facets, define the object to be rotated; must be length 3×*nVertices*; stored [v1x, v1y, v1z, ..., vnx, vny, vnz] where n = *nVertices*.
nVertices [in]	The number of points in the *vertices* and *newVertices* arrays.
motion [in]	(NULL OK) Unit vector specifying which direction in the object is be associated with the velocity of the object; if NULL then assumed to already be rotated into *newMotion*; must be length 3.
up [in]	(NULL OK) Unit vector specifying which direction in the object is "up"; this is used for the second part of the rotation where, in the first part, the object is rotated into *newMotion* and where in the second part, the object is rotated about *newMotion* to line up with *newUp*; expected to be orthogonal to motion vector; must be length 3; if NULL then this second rotation is not performed.
newMotion [in,out]	Use this to specify the new direction of motion that is desired for the object; should be a unit vector of length 3 orthogonal to *newUp*; is also an output because this value may be modified into a unit vector if it is not already, made orthogonal to *newUp*, or otherwise minorly changed to reflect the actual rotation performed.
newUp [in,out]	Use this to specify the new up direction that is desired for the object; should be a unit vector of length 3 and orthogonal to *newMotion*; is an output for similar reasons that *newMotion* is an output.
newVertices [out]	The object vertices after the rotation has been applied; must be length *nVertices*; stored in the same way as vertices.
rotation [out]	Matrix representing linear transform applied to motion, up, and vertices to get *newMotion*, *newUp*, and *newVertices*; the matrix is 3x3 stored in row-major format (i.e. length of leading dimension will be the number of columns).

2.3.2.24 ScreenECF

Syntax

R = ScreenECF(*x, G*)

R = ScreenECF(*Atm, [G]*)

```
SCALING_CODE_API int ScreenECF(char** errorChain,
     const double* normScreenPos, const double* posPlatEcf,
     const double* posTargEcf, const int nScreenPositions,
     const int nGeometries, double* ecfScreenPos)
```

This function returns position vectors in earth-centered fixed (ECF) coordinates of specific locations along a path between a platform and a target. The inputs are

x [vector]	Normalized screen locations or number of screens.
Atm [struct]	A discrete atmospheric structure as from AtmStruct. If *Atm* includes the platform and target locations in *Atm.LatLong*, *G* is not required.
G [struct]	Structure or array of structures from GeomStruct.

The output is

R [array]	Screen locations in ECF coordinates. If *G* is an array of geometry structures, the output will be a cell array where *R*{ii} contains the screen locations for *G*(ii).

Example 2.3.15 (Screen Locations) *Examples illustrating use of* ScreenECF *to compute phase screen locations in ECF coordinates.*

```
>> G = GeomStruct; Atm = AtmStruct;
>> R = ScreenECF(Atm.z/Atm.L,G)
```

Return screen locations along the propagation path.

```
>> R = ScreenECF(Atm)
```

Return screen locations along the propagation path with locations specified in the input Atm *structure.*

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
normScreenPos [in]	Normalized screen locations; must be length *nScreenPositions*×*nGeometries*; screen positions are sequential.
posPlatEcf [in]	Position of platform in ECF coordinates; must be length 3×*nGeometries*; of the form [x1,y1,z1,,x2,y2,z2,...,xN,yN,zN].
posTargEcf [in]	Position of target in ECF coordinates; must be length 3×*nGeometries*; of the form [x1,y1,z1,,x2,y2,z2,...,xN,yN,zN].
nScreenPositions [in]	Number of elements in *normScreenPos*.
nGeometries [in]	Number of paths to compute positions for.
ecfScreenPos [out]	Screen locations along path for each geometry input; must be length 3×*nScreenPositions*×*nGeometries*; has the form [g1, g2, ..., gM] where gX = [x1,y1,z1,x2,y2,z2,...,xN,yN,zN], M=*nGeometries*, N=*nScreenPositions*.

2.3.2.25 SimpleGeom

Syntax

G = SimpleGeom(*hp, ht, rd, [vp], [vt], ...*
[PlatformHeading], [TargetHeading], [EarthRadius])

```
SCALING_CODE_API int SimpleGeom(char** errorChain, const double* hp,
    const int nHp, const double* ht, const int nHt, const double* rd,
    const int nRd, const double* vp, int nVp, const double* vt, int nVt,
    const double* ph, int nPh, const double* th, int nTh,
    const double* re, int nRe, double* posPlatEcf, double* posTargEcf,
    double* velPlatEcf, double* velTargEcf, double* earthRadius)
```

This function returns a geometry structure with platform and target position and velocity in ECF coordinates assuming a spherical earth given a simplified list of input geometry specifications [see Section 2.3.1.1 for a discussion of ECF and LLA coordinates and Section 2.3.2.7 for more information on conversions between the two]. The inputs are

hp [scalar]	Altitude of platform above curved earth surface (m).
ht [scalar]	Altitude of target above curved earth surface (m).
rd [scalar]	Ground range of target from platform along earth surface (m).

vp [scalar]	(Optional) Speed of platform (default is zero) (m/s).
vt [scalar]	(Optional) Speed of target (default is zero) (m/s).
PlatformHeading [scalar]	(Optional) Heading clockwise from due north (default is 90) (deg).
TargetHeading [scalar]	(Optional) Heading clockwise from due north (default is 90 unless *vt* is 0, then the default is *PlatformHeading*) (deg).
EarthRadius [scalar]	(Optional) Spherical Earth Radius (defaults to value from PhysicalConst) (m).

The target will be placed at Lat = 0, Long = 0 at the specified altitude, and the platform will be located on the prime meridian (Long = 0) in the southern hemisphere at a latitude consistent with the input platform-to-target ground range, *rd*. Level flight of the platform and target at (optional) specified headings is assumed. If the speed of the platform or target is not specified, *vp* and *vt* default to zero. If heading is not specified for the platform, target, or wind, the default heading is 90 degrees (due East). The output is a geometry structure that can be used as an input to EngagementStruct and has the following fields:

Coordinates [string]	Coordinate frame for geometry specification - always 'ECF'.
RP [vector]	Platform position in ECF Coords (m).
VP [vector]	Platform velocity in ECF Coords (m/s).
RT [vector]	Target position in ECF Coords (m).
VT [vector]	Target velocity in ECF Coords (m/s).
TALO [empty]	Placeholder for target Time-After-Lift-Off.
EarthRadius [scalar]	Earth radius used (m).

Example 2.3.16 (Simple geometry structure) *This example shows how to construct a geometry structure using* SimpleGeom.

```
>> G = SimpleGeom(1000,10,20000)

G =

    Coordinates: 'ECF'
             RP: [6.3720e+006 0 -2.0003e+004]
             VP: [0 2.2204e-016 0]
             RT: [6.3710e+006 0 0]
             VT: [0 2.2204e-016 0]
           TALO: []
    EarthRadius: 6.3710e+006
```

Input only `hp`, `ht`, *and* `rd` *and everything else takes default values. Note that target is at Lat = 0 and Long = 0 with platform to the south and that both platform and target are stationary. The same output could be gotten from the command* `G = GeomStruct('Simple',1000,10,20000)`.

```
>> G = SimpleGeom(1000,10,20000,100,10,90,180,-90)

G =

    Coordinates: 'ECF'
             RP: [6.3720e+006 0 -2.0003e+004]
             VP: [0 100 0]
             RT: [6.3710e+006 0 0]
             VT: [0 0 -10]
           TALO: []
    EarthRadius: 6.3710e+006
```

Platform is heading East, target is heading South and wind is headed West. Note that target is at Lat = 0 and Long = 0 with platform to the south.

The API function returns system error codes and the arguments are

`errorChain` [in,out]	Holds error and warning messages- deepest error is first.
`hp` [in]	Altitude of platform above curved earth surface (m).
`nHp` [in]	Number of elements in vector `hp`; must be 1 or N where N = max(`nHp`,`nHt`,`nRd`,`nVp`,`nVt`,`nPh`,`nTh`,`nRe`).
`ht` [in]	Altitude of target above curved earth surface (m).
`nHt` [in]	Number of elements in vector `ht`; must be 1 or N.
`rd` [in]	Ground range of target from platform along earth surface (m).
`nRd` [in]	Number of elements in vector `rd`; must be 1 or N.
`vp` [in]	(NULL OK) Speed of platform (m/s); default value is zero.
`nVp` [in]Number of elements in vector `vp`; can be 0, 1, or N. `vt` [in](NULL OK) Speed of target (m/s); default value is zero. `nVt` [in]Number of elements in vector `vt`; can be 0, 1, or N. `ph` [in]	(NULL OK) Platform heading clockwise from due north (deg); default is 90.
`nPh` [in]	Number of elements in vector `ph`; can be 0, 1, or N.
`th` [in]	(NULL OK) Target heading clockwise from due north (deg); default is 90 unless `vt` is 0, then the default is `ph`.
`nTh` [in]	Number of elements in vector `th`; can be 0, 1, or N.

re [in]	(NULL OK) Spherical Earth Radius (m); defaults to value from PHYSICALCONST.
nRe [in]	Number of elements in vector *re*; can be 0, 1, or N.
posPlatEcf [out]	Platform position vector in ECF Coords (m); must allocate 3×N elements; [x1,y1,z1,x2,y2,z2,...].
posTargEcf [out]	Target position vector in ECF Coords (m/s); must allocate 3×N elements.
velPlatEcf [out]	Platform velocity vector in ECF Coords (m); must allocate 3×N elements.
velTargEcf [out]	Target velocity vector in ECF Coords (m/s); must allocate 3×N elements.
earthRadius [out]	Earth radius used (m); must allocate N elements.

2.3.2.26 SLANT2DOWN

Syntax

DownRange = SLANT2DOWN(*L, hp, ht, [EarthRadius]*)

```
SCALING_CODE_API int SLANT2DOWN(char** errorChain,
     const double* slantRange, const int nL, const double* hp,
     const int nHp, const double* ht, const int nHt, const double* re,
     int nRe, int nEarthRadii, double* rd)
```

This function converts a slant range to a downrange along the curved-earth surface. The inputs are

L [scalar]	Target downrange along curved earth surface (m).
hp [scalar]	Platform altitude above surface of the earth (m)
ht [scalar]	Target altitude above surface of the earth (m)
EarthRadius [scalar]	(Optional) Earth radius to use (m) - assumes spherical Earth and defaults to WGS84 mean Earth radius.

The output is the target downrange *DownRange* obtained by first computing the angle between rays from the center of the Earth to the platform and target locations, respectively, using the law of cosines as follows:

$$\cos(\theta) = \frac{(r_e + ht)^2 + (r_e + hp)^2 - L^2}{2(r_e + ht)(r_e + hp)}$$

The downrange is computed as the arc length given by the angle θ

$$rd = r_e\theta$$

where r_e is the radius of the earth in meters.

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
slantRange [in]	Slant range (m); length *nL*.
nL [in]	Number of elements in vector *l*; must be 1 or N where N = max(*nL*,*nHp*,*nHt*).
hp [in]	Platform altitude above surface of the earth (m); length *nHp*.
nHp [in]	Number of elements in vector *hp*; must be 1 or N.
ht [in]	Target altitude above surface of the earth (m); length *nHt*.
nHt [in]	Number of elements in vector *ht*; must be 1 or N.
re [in]	(NULL OK) Earth radius (m); Defaults to WGS84 mean earth radius; must be *nEarthRadii* × *nRe*.
nRe [in]	Must be 1 or N to represent the number of earth radius specifications input.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate; *re* length will be *nEarthRadii* × *nRe*.
rd [out]	Down range along curved-earth surface (m); must allocate length N.

2.3.2.27 SLANT2DOWNEL

Syntax

[DownRange ht] = SLANT2DOWNEL(*L, hp, el, [EarthRadius]*)

```
SCALING_CODE_API int SLANT2DOWNEL(char** errorChain,
     const double* slantRange, const int nL, const double* hp,
     const int nHp, const double* el, const int nEl, const double* re,
     int nRe, int nEarthRadii, double* rd, double* ht)
```

This function converts a slant range at a given elevation angle to a downrange along the curved-earth surface and a target altitude. The inputs are

L [scalar]	Target slant range (m).
hp [scalar]	Platform altitude above surface of the earth (m).
el [scalar]	Elevation angle from platform to target measured from horizontal (rad). Up is positive and down is negative.
EarthRadius [scalar]	(Optional) Earth radius to use (m) - assumes spherical Earth and defaults to WGS84 mean Earth radius.

The function outputs are

DownRange [scalar]	Target downrange along curved-earth surface.
ht [scalar]	Target altitude.

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
slantRange [in]	Slant range (m); length *nL*.
nL [in]	Number of elements in vector *l*; must be 1 or N where N = max(*nL*,*nHp*,*nHt*).
hp [in]	Platform altitude above surface of the earth (m); length *nHp*.
nHp [in]	Number of elements in vector *hp*; must be 1 or N.
el [in]	Elevation angle from platform, to target measured from horizontal (rad); length *nEl*.
nEl [in]	Number of elements in vector *el*; must be 1 or N.
re [in]	(NULL OK) Earth radius (m); Defaults to WGS84 mean earth radius; must be *nEarthRadii* × *nRe*.
nRe [in]	Must be 1 or N to represent the number of earth radius specifications input.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate; *re* length will be *nEarthRadii* × *nRe*.
rd [out]	Down range along curved-earth surface (m); must allocate length N.
ht [out]	Target altitude (m); must allocate length N.

2.3.2.28 TargCoordVec

Syntax

[K, rhoP, rhoT, upECF] = TargCoordVec(*G*)

```
SCALING_CODE_API int TargCoordVec(char** errorChain,
     const double* posPlatEcf, const double* posTargEcf,
     const double* velTargEcf, const int nGeoms, double* lineOfSight,
     double* rhoP, double* rhoT, double* up)
```

This function returns unit vectors in earth-centered fixed (ECF) coordinate system defining a target-based coordinate system from a platform to target given an input structure *G*, like that output from GeomStruct with fields *RT*, *RP*, *VT*, and *VP*. The outputs are

K [vector]	Unit vector for propagation from platform to target.
rhoP [vector]	Unit vector parallel to target velocity, transverse to *K*.
rhoT [vector]	Unit vector transverse to target velocity, transverse to *K*.
upECF [vector]	Unit vector for the zenith at the target in ECF coordinates.

The output *upECF* is simply the unit vector in ECF coordinates for the ray pointing from the center of the earth (0,0,0) to the target location. Mathematical descriptions of the other quantities are given in Section 2.3.1.2.

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
posPlatEcf [in]	Platform position in ECF coordinates (m); [x1,y1,z1,...,zN,yN,zN] where n = *nGeoms*; must be size 3×*nGeoms*.
posTargEcf [in]	Target position in ECF coordinates (m); must be size 3×*nGeoms*.
velTargEcf [in]	(NULL OK IF *rhoP*, *rhoT* AND *up* ARE NULL) Target velocity in ECF coordinates (m); must be size 3×*nGeoms*.
nGeoms [in]	Number of geometry specifications input.
lineOfSight [out]	Unit vector for propagation from platform to target; must be size 3×*nGeoms*.
rhoP [out]	(NULL OK IF *rhoT* AND *up* ARE NULL) Unit vector parallel to target velocity, transverse to *lineOfSight*; must be size 3×*nGeoms*.
rhoT [out]	(NULL OK IF *up* IS NULL) Unit vector transverse to target velocity, transverse to *lineOfSight*; must be size 3×*nGeoms*.
up [out]	(NULL OK) Unit vector for the zenith at the target in ECF coords; must be size 3×*nGeoms*.

2.3.2.29 TargPtAngle

Syntax

[betaP betaT beta] = TargPtAngle(*G, pt1, pt2*)

```
SCALING_CODE_API int TargPtAngle(char** errorChain,
    const double* posPlatEcf, const double* posTargEcf,
    const double* velTargEcf, const int nGeoms, const double* pointA,
    const double* pointB, double* betaP, double* betaT, double* beta)
```

This function computes the angular separation between two points on a target as seen from the platform. The input points should be specified in target P/T coordinate frame. The inputs are

G [struct]	Geometry structure as from GeomStruct.
pt1 [array]	First point on the target (m). Array size should be NGx2.
pt2 [array]	Second point [NGx2] on the target (m).

The outputs are

betaP [vector]	Angular separation between the points in the target P-axis (rad).
betaT [vector]	Angular separation between the points in the target T-axis (rad).
beta [vector]	Full angular separation between points (rad).

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
posPlatEcf [in]	Platform position in ECF coordinates (m); [x1,y1,z1,...,zN,yN,zN] where N = *nGeoms*; must be size 3×*nGeoms*.
posTargEcf [in]	Target position in ECF coordinates (m); must be size 3×*nGeoms*.
velTargEcf [in]	Target velocity in ECF coordinates (m); must be size 3×*nGeoms*.
nGeoms [in]	Number of geometry specifications input.

pointA [in]	First point on the target (m); stored [p1,t1,p2,t2,...,pN,tN] where p and t are coordinates in target parallel and transverse axes respectively and N = *nGeoms*; total length must be 2×*nGeoms*.
pointB [in]	Second point on the target (m); stored just as *pointA*; total length must be 2×*nGeoms*.
betaP [out]	Angular separation between the points in the target P-axis (rad); must be length *nGeoms*.
betaT [out]	Angular separation between the points in the target T-axis (rad); must be length *nGeoms*.
beta [out]	(NULL OK) Full angular separation between points (rad); if NULL this angle is not computed; if not NULL must be length *nGeoms*.

2.3.2.30 TerrainProfiler

Syntax

[h, hTerr, RLLA] = TerrainProfiler(*screens, G, [dirName], [dispMap]*)

This function computes altitude above terrain along a given path. Also returns the terrain altitude and the locations of the phase screens in LLA coordinates. Then inputs are

screens [vector]	Normalized screen locations or number of *screens*.
G [struct]	Geometry structure as from GeomStruct or comma-separated list of (..., *RP*, *RT*, *re*, ...).
RP [array]	LLA position vector of platform, i.e. [lat, long, alt] in degrees and meters.
RT [array]	LLA position vector of target.
re [vector]	(Optional) If scalar - spherical Earth radius, if vector - [a b], where a is the equatorial radius and b is the polar radius.
dirName [string]	(Optional) Directory location of DTED data files. Defaults to EngagementTools/DTED.
dispMap [boolean]	(Optional) Flag indicating whether to display image of terrain data. Default is false.

The outputs are

h [vector]	Altitude of *screens* above the terrain (m).
hTerr [vector]	Altitude of the terrain (m).
RLLA [array]	LLA locations of *screens*.

2.3.2.31 TransVelocity

Syntax

[V, V_TP, V_TT] = TransVelocity(*[x], S, [Atm]*)

```
SCALING_CODE_API int TransVelocity(char** errorChain,
    const double* windHeading, const double* horzWindSpeed,
    const double* vertWindSpeed, const double* screenPositions,
    const double* pathLength, const double* posPlatEcf,
    const double* posTargEcf, const double* velPlatEcf,
    const double* velTargEcf, const double* earthRadius,
    const int nEarthRadii, const int nScreens, const int nGeoms,
    double* windVel, double* windVelParallel, double* windVelTransverse)
```

This function computes the line-of-sight velocity transverse to the propagation direction along the path for platform motion, target motion, and natural wind. It is assumed that the coordinate system has been rotated to be aligned with the target transverse velocity vector. The inputs are

x [vector]	(Optional) Position along path, normalized to slant range (0-1). Not required if *Atm* is a discrete structure from AtmStruct - ignored if passed in and *Atm* is discrete.
S [struct]	Engagement parameters. Can be a structure from EngagementStruct or GeomStruct. If a structure from GeomStruct is passed in as *S*, the engagement structure will be computed.
Atm [struct]	Atmospheric modeling parameters from AtmStruct. *Atm* must have wind profile if calculation with wind is desired.

The outputs are

V [vector]	Magnitude of path-transverse velocity (m/s).
V_TP [vector]	Path-transverse velocity component in the "parallel" direction (m/s).

V_TT [vector]	Path-transverse velocity component in the "transverse" direction (m/s).

and are given by

$$\begin{aligned} V(x) &= [V_{\mathrm{TP}}(x)^2 + V_{\mathrm{TT}}(x)^2]^{1/2}, \\ V_{\mathrm{TP}}(x) &= -V_{\mathrm{TTP}} \cdot x - V_{\mathrm{PTP}} \cdot (1-x) + W_{\mathrm{TP}}(x) \\ V_{\mathrm{TT}}(x) &= -V_{\mathrm{PTT}} \cdot (1-x) + W_{\mathrm{TT}}(x) \end{aligned}$$

where x is the normalized position along the propagation path ($x = 0$ is at the platform and $x = 1$ is at the target), V_{TTP} is the transverse target velocity in the P-direction, V_{PTP} is the transverse platform velocity in the P-direction, and V_{PTT} is the transverse platform velocity in the T-direction. The wind components (computed from information in *Atm*), $W_{\mathrm{TP}}(x)$ and $W_{\mathrm{TT}}(x)$, are given by

$$\begin{aligned} W_{\mathrm{TP}}(x) &= w_{\mathrm{TP}}(x) \cdot V_{\mathrm{w}}(x) \\ W_{\mathrm{TT}}(x) &= w_{\mathrm{TT}}(x) \cdot V_{\mathrm{w}}(x) \end{aligned}$$

where $w_{\mathrm{TP}}(x)$ and $w_{\mathrm{TT}}(x)$ are the wind direction cosines and V_{w} is the wind speed in m/s. If a wind model is not specified, V_{w} is assumed to be zero. The following examples illustrates the calculation of transverse velocity.

Example 2.3.17 (*S* as a structure) *This example shows how to calculate the transverse velocity given the engagement structure, S, from Example 2.2.3.*

```
>> G = GeomStruct('Simple',10,12000,50000,0,250,0,90);
>> [V,V_TP,V_TT] = TransVelocity(0.1:0.2:1,G)

V =

   25.0000   75.0000  125.0000  175.0000  225.0000


V_TP =

  -25.0000  -75.0000 -125.0000 -175.0000 -225.0000


V_TT =

   1.0e-16 *

    0.4580    0.3562    0.2544    0.1527    0.0509
```

Compute the line-of-sight velocity at 5 equally spaced points along the path with no natural wind.

```
>> AtmC = ChangeAtm(AtmStruct,'Screens',inf);
>> [V,V_TP,V_TT] = TransVelocity(0.1:0.2:1,G,AtmC)

V =

   18.3894   63.1529  102.0720  141.3712  192.3910
```

```
V_TP =

  -18.3894  -63.1529 -102.0720 -141.3712 -192.3910


V_TT =

   1.0e-16 *

    0.4580    0.3562    0.2544    0.1527    0.0509
```

Compute the line-of-sight velocity at 5 equally spaced points along the path with a wind model from `Atm` *from* AtmStruct *with wind model from* Bufton*. Note that* `x` *is not needed if* `Atm` *is a discrete model structure. In this case,* `G` *and geometry in* `AtmC` *are not consistent and AtmC will be updated.*

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
windHeading [in]	Natural wind heading for the phase screens (deg from North); must be *nScreens* × *nGeoms* with the value for each screen at consecutive addresses.
horzWindSpeed [in]	Natural wind speed values for the phase screens (m/s); must be *nScreens* × *nGeoms*.
vertWindSpeed [in]	Natural wind speed values orthogonal to North-South plane and in "up" direction (m/s); must be *nScreens* × *nGeoms*.
screenPositions [in]	Phase screen distances from transmitter (m); must be *nScreens* × *nGeoms*.
pathLength [in]	Slant range from platform to target (m); must be length *nGeoms*.
posPlatEcf [in]	ECF position vector(s) for platform (m); must be length 3×*nGeoms* with each coordinate belonging to one geometry specification consecutive.
posTargEcf [in]	ECF position vector(s) for target (m); must be length 3×*nGeoms*.
velPlatEcf [in]	ECF velocity vector(s) for platform (m); must be length 3×*nGeoms*.
velTargEcf [in]	ECF velocity vector(s) for target (m); must be length 3×*nGeoms*.
earthRadius [in]	Specify the earth radius belonging to each geometry; must be *nEarthRadii* × *nGeoms* with each radius belonging to one geometry specification consecutive.
nEarthRadii [in]	Specify 1 for spherical earth with that radius or 2 for oblate earth; must be 1 or 2.

nScreens [in]	Number of phase screens specified for each geometry.
nGeoms [in]	Number of geometries over which to compute wind direction cosines.
windVel [out]	Magnitude of path-transverse velocity (m/s); will be length *nScreens*×*nGeoms* with the value for each screen at consecutive addresses.
windVelParallel [out]	Path-transverse velocity component in "parallel" direction (m/s); must be length *nScreens*×*nGeoms*.
windVelTransverse [out]	Path-transverse velocity component in "transverse" direction (m/s); must be length *nScreens*×*nGeoms*.

2.3.2.32 VTP2Vxy

Syntax

[Vx Vy] = VTP2Vxy(*Vtp, upTP, upSpec*)

[Vx Vy] = VTP2Vxy(*G, [Atm], upSpec*)

```
SCALING_CODE_API int VTP2Vxy(char** errorChain, const double* Vtp,
    const int nEvals, const double* upTP, const double* upSpec,
    double* Vxy)
```

This function rotates velocity vector from target T/P coordinates, or some other coordinate frame, into the desired coordinate frame. Rotation is based on the angle between the projection of the zenith direction onto the target plane, i.e. the plane transverse to the propagation direction, in target T/P coordinates and the user-specified zenith projection. The inputs are

Vtp [vector]	Velocity vector in target T/P coordinates [V_TT V_TP] of size Nx2 (m/s).
upTP [vector]	Projection of the "up" direction onto the target plane in target T/P coordinates of size 1x2.
upSpec [vector]	Desired projection of the "up" direction.

Note that the inputs *upTP* and *upSpec* need not be normalized, i.e. the magnitude need not represent the magnitude of the component of the zenith projection in the target plane. The only thing that matters is the angle between the two in the target plane. Use of the user-specified "up" direction is under the assumption that it is defined in a right-handed coordinate frame where z is the direction of propagation from the platform to the target.

The outputs are

Vx [vector]	Component of transverse velocity in the x-direction (m/s).
Vy [vector]	Component of transverse velocity in the y-direction (m/s).

Example 2.3.18 (Rotation of Velocity) *This example illustrates rotation of velocity components into an alternate coordinate basis.*

```
>> G = GeomStruct('simple',10,12000,50000,0,250,0,90);
>> [K,rhoP,rhoT,upECF] = TargCoordVec(G);
>> upTP = upECF*[rhoT' rhoP']

upTP =

    0.9716         0

>> S = EngagementStruct(G);
>> [Vx Vy] = VTP2Vxy([S.V_PTT S.V_PTP],upTP,[0 1])

Vx =

     0


Vy =

  -5.0884e-17
```

The default assumption of GeomStruct‡ *is that target and platform velocity are both horizontal, i.e. no vertical components. Therefore, the target P direction is horizontal to the earth and the T direction is vertical. The platform velocity is computed in the alternate coordinate frame where* ***Vx*** *represents horizontal component and* ***Vy*** *represents vertical component. In this case, because the platform is lower altitude than the target and headed North, directly toward the target, the platform has an apparent negative vertical velocity.*

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
Vtp [in]	Velocity vector in target T/P coordinates V_TT, V_TP of size 2×*nEvals* (m/s).
nEvals [in]	Number of evaluations.
upTP [in]	Projection of the "up" direction onto the target plane in target T/P coordinates; length 2.

upSpec [in]	Desired projection of the "up" direction; length 2.
Vxy [out]	Transverse velocity. Allocate to 2×*nEvals* where first *nEvals* are x-comp and second *nEvals* are y-comp.

2.3.2.33 WhichCommonSite

Syntax

[SiteP, SiteT] = WhichCommonSite(*G*)

This function returns site information for the location from CommonSites which matches the input geometry information. If no match, returns 'UNKNOWN'. If a *G* structure is passed in, information for platform and target are returned. If an LLA position vector is passed in, a single site is returned. The inputs are

G [struct/vector]	Geometry structure or LLA coordinate vector.

The function returns

SiteP [cell]	Site information for platform.
SiteT [cell]	Site information for target. Only if the input is a geometry structure.

2.3.3 Functions Supporting Scaling Law Analysis

2.3.3.1 CaseStruct

Syntax

Cstruct = CaseStruct(*P, G, Targ, [Atm], [lambda]*)

Cstruct = CaseStruct(*C, [FieldName1], [FieldValue1], ...*
[FieldName2], [FieldValue2], ...)

This function returns a structure or array of structures of engagement case parameters for laser engagement scenarios. The output of CaseStruct is one of the main structures used in SCALE and SHaRE toolboxes. The inputs are

C [struct]	Can be a structure with fields *P*, *G*, *Targ*, and optionally *Atm* or a comma-separated list of (*P*, *G*, *Targ*, [*Atm*], [*lambda*]). Can also be a complete *C* structure followed by a list of parameters to change within that *C* structure.
P [struct]	System parameters, as from ABLParams (SCALE) or LinkParams (SHaRE). Must have fields 'name', 'HEL', 'Tx', and 'Rx'.
G [struct]	Geometry description from GeomStruct.
Targ [struct]	Target parameters, as from TargStruct. Must have fields 'trackpt', 'HELpt', 'TargetRadius', 'Flyout', 'Lethality', and 'HELPointing'.
Atm [struct]	(Optional) Atmospheric description from AtmStruct.
lambda [scalar]	(Optional) Wavelength of laser. Only used in the case where *P.HEL* is empty. Must pass in *Atm* to specify wavelength, can pass [] for *Atm*.
FieldName [string]	(Optional) Name of field to be updated (some fields are restricted).
FieldValue [varies]	(Optional) Value for *FieldName* parameters. There must be a field value for each field name specified.

When using the calling sequence with *FieldName* and *FieldValue*, the input *C* structure must be complete - an output from CaseStruct. *FieldName* can be any unrestricted field in the input *C* or any of 'alpha,' 'HELpt' and 'BILLpt.' Restricted fields are those which are computed based on other parameters and are indicated as restricted in the description of the case structure fields below.

The output is an engagement case structure.

C [struct]	Structure of engagement case parameters.
C.code [string]	(Restricted) Identifier for the engagement case.
C.type [string]	(Restricted) Identifier for propagation case type.
C.ver [struct]	(Restricted) Structure of MATLAB® version used to generate *C* (*C.ver*=*ver*;).
C.P [struct]	Structure of system design parameters.
C.G [struct]	Structure of geometry parameters as from GeomStruct.
C.Targ [struct]	Structure of target parameters.
C.Atm [struct]	Structure of atmospheric parameters from AtmStruct (turbulence, wind, etc) -Can only change Model parameters in *Atm*, not geometry parameters.

C.wavelength [scalar]	Wavelength of propagated beam (m).
C.hp [scalar]	Altitude of source platform (m).
C.ht [scalar]	Altitude of target platform (m).
C.rd [scalar]	Range from source to target along earth surface (m).
C.L [scalar]	(Restricted) Slant range from source to target (m).
C.vp [scalar]	Platform speed (m/s).
C.vt [scalar]	Target speed (m/s).
C.V_TTP [scalar]	(Restricted) Path-transverse target velocity, in missile "parallel" direction (m/s).
C.V_PTP [scalar]	(Restricted) Path-transverse platform velocity, in missile "parallel" direction (m/s).
C.V_PTT [scalar]	(Restricted) Path-transverse platform velocity, in missile "transverse" direction (m/s).
C.az [scalar]	(Restricted) Azimuth of target relative to platform (rad).
C.el [scalar]	(Restricted) Elevation of target relative to platform (rad).
C.tia [scalar]	(Restricted) Target incidence angle (rad).
C.LOSangle [scalar]	(Restricted) Magnitude of the full angle between platform velocity and propagation vector (rad).
C.Fresnel [scalar]	(Restricted) Propagation *Fresnel* number, $D^2/(\lambda L)$.
C.Rytov [scalar]	(Restricted) Spherical wave *Rytov* number for engagement.
C.r0 [scalar]	(Restricted) Spherical wave coherence diameter for engagement (m).
C.theta0 [scalar]	(Restricted) Spherical wave isoplanatic angle for engagement (rad).
C.fG [scalar]	(Restricted) Greenwood frequency (Hz).
C.fT [scalar]	(Restricted) Tyler frequency (Hz).
C.Nd [scalar]	(Restricted) Uniform wave distortion number.

If one does not have SCALE or SHaRE on the path, CaseStruct can still be used as shown in Example 2.3.19.

Example 2.3.19 (Minimum requirements for CaseStruct) *This example shows how to generate an engagement case structure without the use of SCALE and SHaRE.*

```
>> P.name = 'MyParams';
>> P.HEL = [];
>> P.Tx.D = 0.50;
>> P.Rx = [];
```

```
>> G = GeomStruct;

>> Targ = TargStruct;

>> C = CaseStruct(P,G,Targ)

C =

          code: 'CUSTOM'
          type: 'TARGETLINK'
           ver: [1x6 struct]
             P: [1x1 struct]
             G: [1x1 struct]
          Targ: [1x1 struct]
           Atm: []
    wavelength: 1.0000e-006
            hp: 12000
            ht: 0
            rd: 5.0000e+004
             L: 5.1465e+004
            vp: 0
            vt: 2.2204e-016
         V_TTP: 2.2204e-016
         V_PTP: 2.2204e-016
         V_PTT: 0
            az: -1.5708
            el: -0.2393
           tia: 0
      LOSangle: 1.5708
       Fresnel: 4.8576
         Rytov: []
            r0: []
      theta0: []
            fG: []
            fT: []
            Nd: []
```

Create a parameter structure with minimum fields required. One could also set `P.HEL.Wavelength` *and* `P.HEL.Power` *to desired values. Because no atmospheric structure was specified, the atmospheric parameters are empty.*

2.3.3.2 G_Interface

Syntax

GNew = G_Interface(*[GOld], [CoordFormat]*)

G_Interface is a Graphical User Interface (GUI) for editing a single G structure within a scenario. Optional inputs are

GOld [struct]	(Optional) Starting geometry structure from GeomStruct. If not declared, *GOld* is the default structure output from GeomStruct.
CoordFormat [string]	(Optional) Output geometry structure coordinates format. Can be 'LLA' or 'ECF' by default.

The output *GNew* is the modified geometry structure in ECF coordinates, unless LLA coordinates are specified through use of *CoordFrame*. If *GNew* is not specified at the command prompt, the variable *G* will be assigned to the base workspace when the "Commit & Close" button is pushed. While the GUI window is open, execution of commands at the MATLAB® command prompt will be blocked. Figure 2.5 shows the geometry structure interface with default geometry from GeomStruct. The function plotEng is used to generate the display in the bottom half of the figure window. The pulldown menus to the right of the plot can be used to change the coordinate frame and the view for the plot.

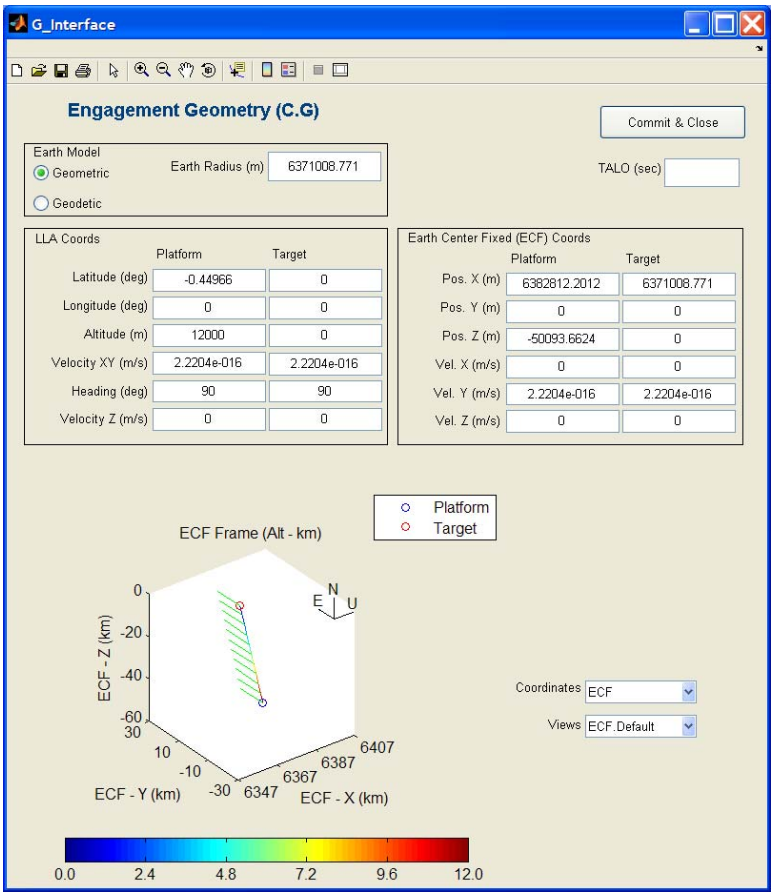


Figure 2.5: Geometry structure interface with the default geometry structure from GeomStruct.

2.3.3.3 getTargetSize

Syntax

[TargLen, TargWidth, TargHeight] = getTargetSize(*Targ, G, hitPt*)

```
SCALING_CODE_API int getTargetSize(char** errorChain, int nVertices,
    const double* objVertices, const double* objMotion,
    const double* objZenith, int nGeoms, const double* posPlatEcf,
    const double* posTargEcf, const double* velTargEcf, int nHitPoint,
    const double* hitPoint, double* objLength, double* objWidth,
    double* objHeight)
```

This function returns the maximum target length and the width and height at the point determined by *hitPt*. The output *TargHeight* can be interpreted as twice the target radius. Any effect on target size due to incidence angle is not included. The inputs are

Targ [struct]	Target structure from TargStruct.
G [struct]	(Optional) Geometry structure from GeomStruct. If not input or [], assumes target is rotationally symmetric about its motion.
hitPt [array]	(Optional) Point on the target for which to compute size in target P/T coordinates. Must be size 2 x 1 or 2 x nGeoms. If not input, returns the maximum dimensions.

The outputs are

TargLen [scalar]	Maximum target length (m).
TargWidth [scalar]	Target width at *hitPt* (m).
TargHeight [scalar]	Target height at *hitPt* (m).

2.3.3.4 LaserField

Syntax

Laser = LaserField(*x, y, LaserType, TotalPower, lambda, ... FocusDistance, [p1, p2, ...pN]*)

```
SCALING_CODE_API int LaserField(char** errorChain, const double* x0,
    const int nx0, const double* x1, const int nx1, const char laserType,
    const double totalPower, const double wavelength,
    const double focusDistance, const double* param1,
    const double* param2, const double* param3, const double* param4,
    double* dimension, double* fieldRe, double* fieldIm)
```

This function returns a structure representing a Laser field for use in wave-optics simulation. The inputs are

`x` [vector]	X centers (vertical axis) for laser field evaluation (m).
`y` [vector]	Y centers (horizontal axis) for laser field evaluation (m).
`LaserType` [string]	Identifier for type of laser to model. Supports 'UNIFORM', 'GAUSSIAN', 'LAGUERRE-GAUSS', and 'RECTANGULAR' (see comments below).
`TotalPower` [scalar]	Output power for the laser (W).
`lambda` [scalar]	Wavelength of the laser (m).
`FocusDistance` [scalar]	Radius of curvature for beam (m).
`p1...pN` [list]	Parameter values for laser model options.

The parameters `p1` to `pN` depend on `LaserType` as follows:

UNIFORM

`p1` = Diameter of aperture (m).
`p2` = Diameter of central obscuration (m).

GAUSSIAN

`p1` = Sigma for irradiance of form $\exp(-(r/\sigma)^2)$ (m).
`p2` = Diameter of clipping aperture (m).

LAGUERRE-GAUSS

`p1` = Sigma for irradiance of form $\exp(-(r/\sigma)^2)$ (m).
`p2` = Diameter of clipping aperture (m).
`p3` = Laguerre radial mode.
`p4` = Laguerre angular mode.

RECTANGULAR

`p1` = Outer side length in `x` axis (m).
`p2` = Outer side length in `y` axis (m).
`p3` = (Optional) Obscuration side length in `x` axis (m).
`p4` = (Optional) Obscuration side length in `y` axis (m).

The output structure contains a complex grid for the laser field consistent with the input parameters.

`Laser` [struct]	Output Laser structure.
`Laser.x` [vector]	`x` locations of laser field grid points (m).
`Laser.y` [vector]	`y` locations of laser field grid points (m).
`Laser.g` [matrix]	Complex grid of laser field values ($\sqrt{W}/m$).
`Laser.LaserType` [string]	Identifier for laser field specification.

Laser.TotalPower [scalar]	Total integrated power for field (W).
Laser.Wavelength [scalar]	Wavelength of laser (m).
Laser.FocusDistance [scalar]	Radius of curvature for beam (m).
Laser.Dimension [scalar]	Characteristic size of beam (m)
Laser.Params [cell array]	Model parameters input to function.

Laser.g is a complex laser field calculated as follows for Gaussian, Uniform, or rectangular beams.

$$g = A' e^{-i\theta}$$

where θ is the focus phase given by

$$\theta = \begin{cases} \frac{k}{2R} r^2 & r \leq \mathit{FocusLimit} \\ 0 & r > \mathit{FocusLimit} \end{cases}$$

where *FocusLimit* is given by D/2 for a Uniform or Rectangular beam and by $5\sigma_{Irr}$ for a Gaussian beam, and R is the focus distance or phase curvature. A' is given by

$$A' = \sqrt{P} \frac{P'(x,y)}{\sqrt{\int P'(x,y)^2 dA}}$$

where $P'(x,y)$ is related to the pupil function, $P(x,y)$, by

$$P'(x,y) = \begin{cases} P(x,y) & \text{Uniform or Rectangular} \\ P(x,y)\sqrt{\exp(-r^2/\sigma_{Irr}^2)} & \text{Gaussian} \end{cases}$$

For the Laguerre-Gauss beam, the laser field is given by [7]

$$g = \left(-\frac{r}{\sigma_{Irr}}\right)^m \exp(i\, m\, \theta) \exp\left[-\frac{r^2}{2\sigma_{Irr}^2}\right] L_n^{(m)}(r^2/\sigma_{Irr}^2)$$

where $L_n^{(m)}$ is the associated Laguerre polynomial [8] given by

$$L_n^{(m)}(x) = \sum_{k=0}^{n} \frac{(-1)^k (n+m)!\, x^k}{(n-k)!(m+k)!k!}$$

and n and m are the radial and angular mode numbers respectively.

Example 2.3.20 (LaserField) *This example shows how to calculate a laser field for a uniform beam and a Gaussian beam.*

```
>> Laser = LaserField([-1:.01:1],[-1:.01:1],'uniform',...
                      100000,1.064e-6,20000,0.5,.1)

Laser =

                x: [201x1 double]
                y: [201x1 double]
                g: [201x201 double]
        LaserType: 'UNIFORM'
       TotalPower: 100000
       Wavelength: 1.0640e-006
```

```
    FocusDistance: 20000
        Dimension: 0.5000
           Params: {[0.5000]  [0.1000]}
```

Laser field, `Laser.g`, for a uniform beam with 100 kW power and wavelength of 1.064 μm and 20 km focus range. The uniform beam has aperture diameter of 0.5 m and obscuration of 0.1 m. This field can be plotted by `imagesc(Laser.x,Laser.y,abs(Laser.g.*Laser.g))`.

```
>> Laser = LaserField([-1:.01:1],[-1:.01:1],'gaussian',...
                      100000,1.064e-6,20000,0.25,.5)

Laser =

                x: [201x1 double]
                y: [201x1 double]
                g: [201x201 double]
        LaserType: 'GAUSSIAN'
       TotalPower: 100000
       Wavelength: 1.0640e-006
    FocusDistance: 20000
        Dimension: 0.7071
           Params: {[0.2500]  [0.5000]}
```

Laser field, `Laser.g`, for a Gaussian beam with 100 kW power and wavelength of 1.064 μm and 20 km focus range. The Gaussian beam has σ of 0.25 m and the diameter of the clipping aperture is 0.5 m. This field can be plotted by `imagesc(Laser.x,Laser.y,abs(Laser.g.*Laser.g))`.

2.3.3.5 ParamStruct

Syntax

P = ParamStruct(*lambda, TxD, RxD*)

P = ParamStruct(*XMLFile*)

ParamStruct returns a basic parameter structure, as from SourceParams or RelayParams in SHaRE, with ideal system parameter settings. The inputs are

lambda [scalar]	Wavelength of laser (m). Leave empty to return parameters for a relay system.
TxD [scalar]	(Optional) Diameter of system transmitter (m). Defaults to 1.0 m.
RxD [scalar]	(Optional) Diameter of system receiver (m). If *lambda* is empty, defaults to value of *TxD* if not input or input as empty.

One can also specify the name of an xml file to load. System parameters will be imported from the specified xml file as saved by APIstruct.struct2xml (see APIstruct). The default xml file location is the directory returned from ParamPath.

The output is

P [struct]	System parameter structure as from either SourceParams or RelayParams depending on *lambda*.
P.name [string]	Name of relay system.
P.HEL [struct]	Structure containing *HEL* parameters. If *lambda* is empty, *P.HEL* is empty indicating a relay system.
P.Tx [struct]	Structure containing transmit parameters.
P.Rx [struct]	Structure containing receiver parameters. If *lambda* is not empty, *P.Rx* is empty indicating a laser source system.

Example 2.3.21 (General Parameter Structure) *Generate some basic parameter structure with ideal system settings.*

```
>> P = ParamStruct(1e-6,0.5);

P =

    name: 'CUSTOM'
     HEL: [1x1 struct]
      Tx: [1x1 struct]
      Rx: []
```

Parameter structure for a source at 1 micron and 50 cm transmitter. Note that P.Rx is empty.

```
>> P = ParamStruct([],0.5);

P =

    name: 'CUSTOM'
     HEL: []
      Tx: [1x1 struct]
      Rx: [1x1 struct]
```

Parameter structure for a relay with 50 cm transmitter and receiver. Note that P.HEL is empty.

2.3.3.6 plotEng

Syntax

[ax refs] = plotEng(*G, [Atm], [coord], [model], [figHandle]*)

This function generates a 3-dimensional plot display of the engagement geometry in the specified coordinate frame. The plot includes indicators of beam slew rate and direction at multiple locations along the beam path. The line connecting the platform and target is color-coded by the specified *model* values along the path. The inputs are

G [struct]	Geometry structure from GeomStruct or a C structure from CaseStruct. If a C structure is passed in, the *Atm* structure is not needed and will be ignored if passed in.
Atm [struct]	(Optional) Atmospheric parameter structure from AtmStruct. If not passed in, the *model* will be set to the altitude along the path and no effects of wind are included in the beam slew.
coord [string]	(Optional) Desired coordinate frame, one of '*ECF*' (default), 'LLA' (or '*ENU*' - east-north-up), or 'PTK'.
model [string]	(Optional) Model values to use for color-coding along the path. Must be a *model* in the *Atm* structure or 'h' for altitude (default). Can also specify using the log of the *model* values by prepending the *model* name with 'log'.
figHandle [handle]	(Optional) Handle for plot axes.

The outputs are

ax [scalar]	Handle to the plot axes.
refs [struct]	Contains the unit vectors for the three coordinate frames in the plotted coordinate frame. Also contains axis setting to generate views along the various coordinate axes frames - see examples.
refs.ECF [struct]	*ECF* x, y, and z unit vectors in the plotted coords.
refs.ENU [struct]	*ENU* x, y, and z unit vectors in the plotted coords.
refs.PKT [struct]	*PKT* x, y, and z unit vectors in the plotted coords.
refs.View [struct]	Camera view setting for the axes. *View.ECF*, *View.ENU*, and *View.PKT* are structures with the settings to change the plot view to the XY, XZ, YZ, or Default view for that coordinate frame.

Example 2.3.22 (plotEng) *This example illustrates the use of* plotEng.

```
>> G = GeomStruct; Atm = AtmStruct;
>> [AX REFS] = plotEng(G,Atm,'enu','logAbs')
```

Plots the geometry in ENU coordinates using log of the absorption profile for color.

```
>> set(AX,REFS.View.ECF.XY)
```

Sets the view for the plot to the ECF XY coordinate plane.

```
>> set(AX,REFS.View.ECF.Default)
```

Sets the view for the plot to the default ECF view - the view as if the geometry were plotted in the ECF frame.

2.3.3.7 PLOTGEOM

Syntax

PLOTGEOM(*[ax], S, [hGnd], [dirName], [dispMap]*)

Produces a 2-dimensional plot of the engagement geometry with a color-coded background indicating air, earth below mean sea level and ground level if input. The inputs are

ax [scalar]	(Optional) Axis handle for plot. Creates new figure if *ax* is not passed in.
S [struct]	Engagement geometry structure as from ENGAGEMENTSTRUCT.
hGnd [vector/logical]	(Optional) Altitude of ground above mean sea level (m). Defaults to 0. Use logical true to load terrain data and add to plot. Not required if specifying *dirName*.
dirName [string]	(Optional) Directory location of DTED data files for loading terrain data. Defaults to EngagementTools/DTED.
dispMap [boolean]	(Optional) Flag indicating whether to display image of terrain data. Default is false. Only used if using terrain data.

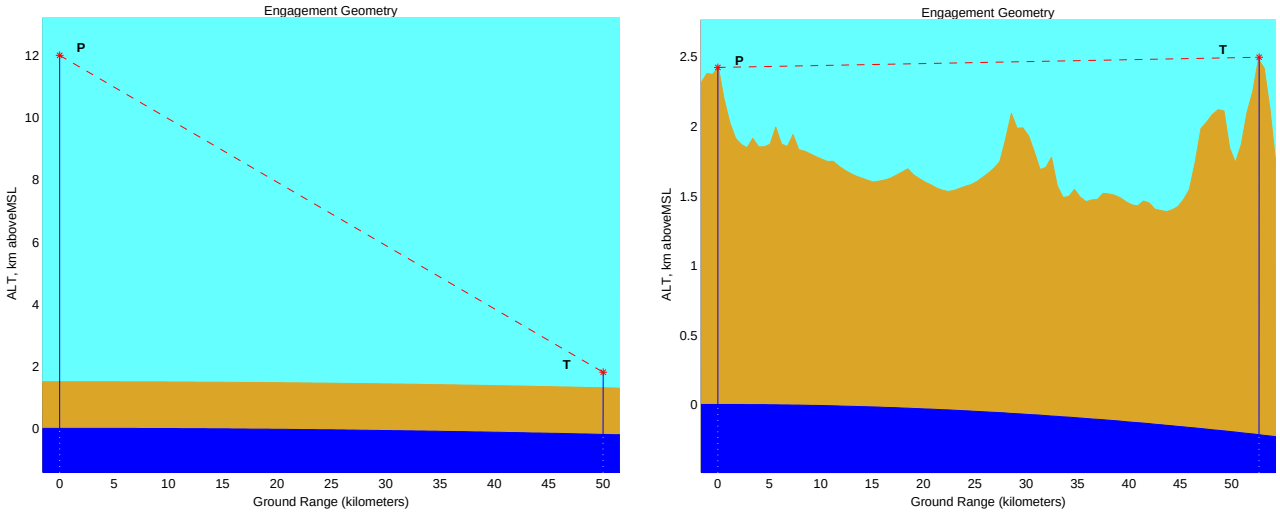
Example 2.3.23 (plotGeom) *Example of a plot of engagement geometry.*

```
>> G = GeomStruct('Simple',12000,2000,50000);
>> plotGeom(G,1500)
```

Plot engagement geometry for airborne platform and target 500 m above ground level. Figure 2.6(a) shows the plot produced by PLOTGEOM.

```
>> G = GeomStruct('LLA',CommonSites('NOP'),CommonSites('SALINAS'));
>> plotGeom(G,true)
```

Plot engagement geometry with terrain data, as in Figure 2.6(b). Requires that the user download the necessary DTED files. See Section 2.3.4.2 for information regarding downloads of DTED files. Table 2.1 contains latitude and longitude information for sites databased in COMMONSITES.



(a) (b)

Figure 2.6: Plots of engagement geometry for Ex. 2.3.23 (a) with fixed ground altitude and (b) with terrain data.

2.3.3.8 PLOTOBJECT

Syntax

PLOTOBJECT(*Targ, [G]*)

Displays a three dimensional plot of a given object in the object's coordinate frame. Includes vertices, unit vectors in direction of motion and zenith, *hitPoint* and *trackPoint* if specified. Use ROTATEOBJECT to plot the object in alternate coordinate frames. The inputs are

Targ [struct]	Target structure form TARGSTRUCT with an object defined or object structure as from *Targ.Object*.
G [struct]	(Optional) Geometry structure from GEOMSTRUCT. Only used when *Targ* is a target structure to convert *Targ.HELpt* to the object coordinate frame when *Targ.Object.hitPoint* is empty.

Example 2.3.24 (plotObject) *This example illustrates object visualization using* PLOTOBJECT.

```
>> Targ = TargStruct('cone',[0.5,2,4]);
>> plotObject(Targ.Object)
```

Display the object in the object (x,y,z) coordinate frame.

```
>> plotObject(RotateObject(Targ.Object,[0 0 1],[1 0 0]))
```

Display object rotated such that the direction of motion is in the x-axis and the zenith direction is in the z-axis.

2.3.3.9 TargGUI

Syntax

Targ = TargGUI(*[TargIn]*)

TargGUI is a user interface for displaying target data. It also allows for some modifications to the target structure to be applied. The optional input is

TargIn [struct]	(Optional) A target structure, as from TargStruct.

The output is

Targ [struct]	Target structure with any modifications applied, or the default target structure if no inputs.

While open, TargGUI blocks the execution of commands using UIWAIT until it is closed.

Figure 2.7 shows the target interface window with the default target structure loaded. Most parameters can be modified within the GUI. Any vector parameters cannot be made shorter or longer than they are upon input.

TargGUI also allows the user to specify a target object type in the class field, although additional parameters that could be input for a specific object are set to the default values. See Section 2.3.3.10 for more information. Figure 2.8 shows the target interface window after specifying the class as a box. The object model defaults to a 1 m cube with a single facet on each side. The user can then use the button Plot Object to see the object model with unit vectors for the direction of motion, the zenith direction, and the hit point and track point, if specified.

2.3.3.10 TargStruct

Syntax

Targ = TargStruct(*[TargDesc], [objSize], [objRes]*)

Targ = TargStruct(*[class], [Stage], [DataClassification]*)

Targ = TargStruct(*XMLFile*)

```
SCALING_CODE_API int TargStruct(char** errorChain,
    const char* targDesc, const double* param1, const int nParam1,
    const int* param2, int* nVertices, int* nFacets,
    int* nVerticesPerFacet, double* vertices, int* facets,
    double* zenith, double* motion, double* hitPoint, double* trkPoint,
    double* refLoc, double* targRadius, double* targLength)
```

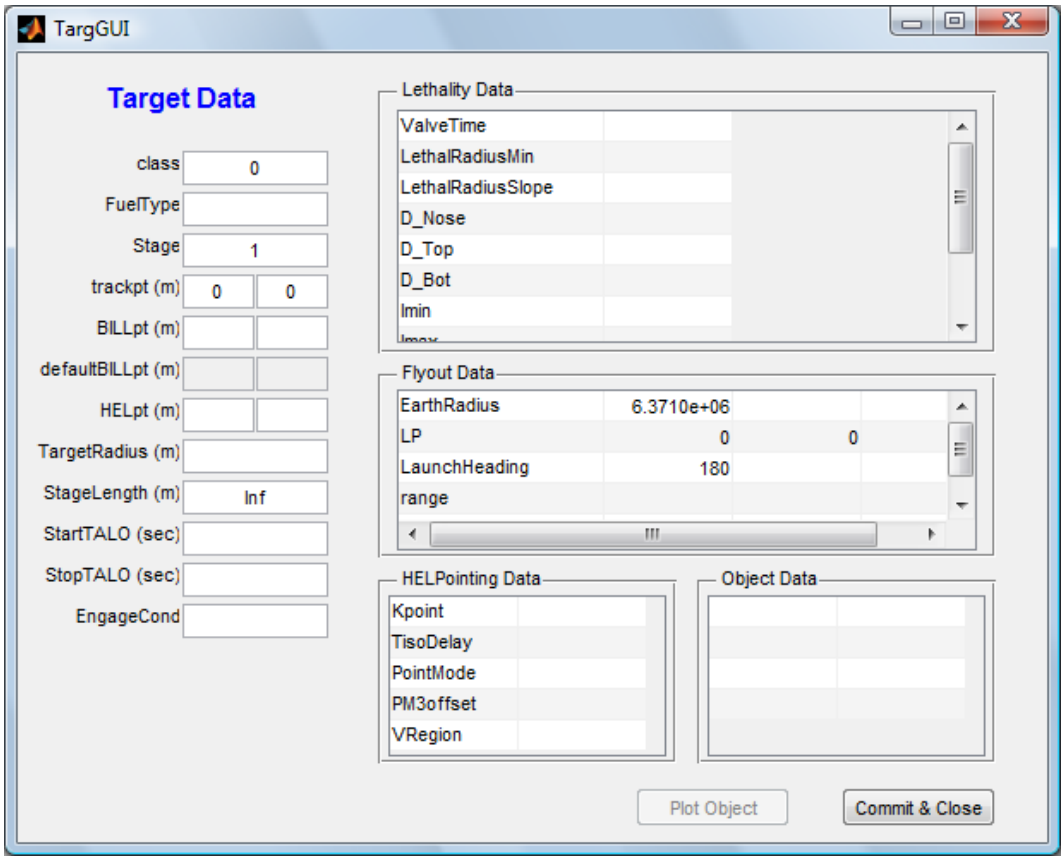


Figure 2.7: TARGGUI **with default target structure loaded.**

This function returns a target structure for SHaRE and SCALE. The inputs are

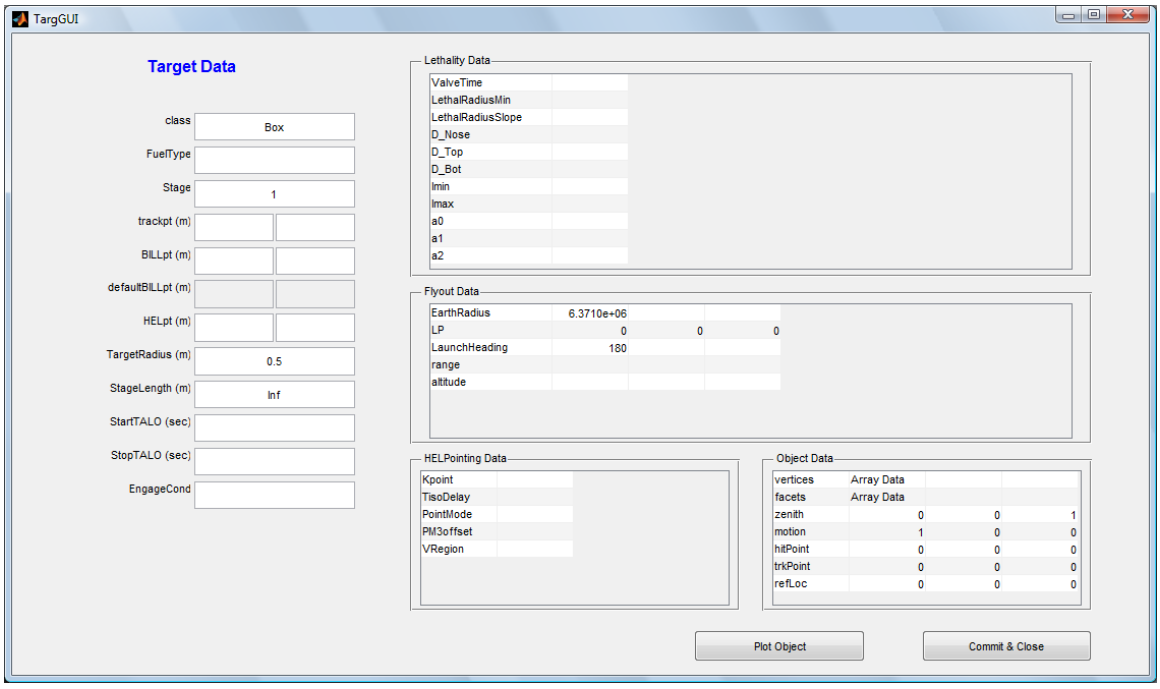


Figure 2.8: TargGUI after specifying the class as a box.

TargDesc [string]	(Optional) Name of a function or function handle to a function that returns a target structure or a string indicating one of the target shapes available ('cylinder,' 'box,' or 'targetboard').
objSize [array]	Size of the object. Interpretation depends on the target shape (see below).
objRes [scalar]	Number of facets to model. Interpretation depends on the target shape (see below).
class [scalar]	(Optional) Identifier for target *class* (see ABLCaseTargData).
Stage [scalar]	(Optional) Missile stage (1 or 2) (see ABLCaseTargData).
DataClassification [string]	(Optional) Classification of the data. Can be either 'U' for unclassified or 'S' for secret.
XMLFile [string]	Name of xml file. Target parameters will be imported from the specified xml file as saved by APIstruct.struct2xml (see APIstruct). The default xml file location is the directory returned from ParamPath.

If no inputs are given or if the first input is *class* and SCALE is not available, a default target structure is returned with *BILLpt* = [], *trackpt* = [0;0], and all other fields empty. The output target structure has fields as follows

Targ [struct]	Structure of target parameters.
Targ.class [scalar]	Target *class*.
Targ.FuelType [string]	Fuel type for the target booster.
Targ.Stage [scalar]	Current selected stage - used for engagement modeling. Default value is 1.
Targ.trackpt [vector]	Location of track point relative to missile nose (m).
Targ.BILLpt [vector]	Location of BILL point relative to missile nose (m). If it is empty, the BILL point is the default *BILLpt* in *Targ.defaultBILLpt*. This field is almost always empty unless a specific *BILLpt* is desired. To force the use of a specific *BILLpt*, use CaseStruct to change '*BILLpt*'.
Targ.defaultBILLpt	The minimum of the isoplanatic BILL point and the Beacon Margin (P.Tx.AO.BeaMargin + *Targ.trackpt*).
Targ.HELpt [vector]	Location of HEL relative to missile nose (m). If it is empty, the HEL point for an engagement will be computed based on information in *HELPointing*.
Targ.TargetRadius [scalar]	Radius of target body.
Targ.StageLength [scalar]	Physical length of each stage body (m).
Targ.StartTALO [scalar]	Start time of the current stage (s) after lift-off.
Targ.StopTALO [scalar]	Stop time of the current stage (s) after lift-off.
Targ.EngageCond [scalar]	Flag of 0, 1, or 2 indicating which engagement condition to use: CloudFreeAlt, Co-Altitude, Elevation angle>=0, respectively.
Targ.Flyout [struct]	Structure containing target trajectory data.
Targ.Lethality [struct]	Structure of target lethality data.
Targ.HELPointing [struct]	Parameters for computing aimpoint.
Targ.Object [struct]	Parameters for the target when *TargDesc* is one of the standard object models.

For the standard object models, the optional parameters are as follows:

TargDesc	'cylinder,' models a cylinder centered around the x-axis with motion in the x-axis.
objSize [array]	(Optional) Radius and length of the cylinder. The radius defaults to 1 m. If not input or scalar, length will default to twice the radius.

objRes [scalar]	(Optional) Number of facets around the diameter to model. Defaults to 20.
TargDesc	'box,' models a 3 dimensional box with motion in the x-axis and a zenith direction in the z-axis.
objSize [array]	(Optional) Lengths of the sides. If scalar, will model a cube. Defaults to 1 m cube.
objRes [scalar]	(Optional) Square root of the number of facets on a side. Defaults to a single facet per side.
TargDesc	'targetboard,' models 2 dimensional targetboard in the xy-plane with motion in the x-axis and zenith direction in the z-axis.
objSize [array]	(Optional) Lengths of the sides. If scalar will model a square. Defaults to 1 m square.
objRes [scalar]	(Optional) Square root of the total number of facets. Defaults to a single two-sided facet.
TargDesc	'cone', 'ogive', 'hemisphere', 'ellipsoid' models the specified shape with the x-axis being the axis of rotation, motion in the x-axis and the zenith direction in the z-axis.
objSize [array]	Bottom radius and height of the shape. If scalar, assumes the height is 1 m unless 'hemisphere' is specified.If 3 elements, Bottom radius, height, and cylinder length, a cylinder will be added to the object.
objRes [scalar]	(Optional) Number of facets around the diameter to model. Defaults to 20.

Example 2.3.25 (TargStruct) *The following illustrates different target objects*

```
>> Targ = TargStruct('cylinder',[0.5,4])
```

Cylinder target with radius 0.5m and length 4m.

```
>> Targ = TargStruct('cone',[0.5,2,4])
```

Cone target with bottom radius 0.5m, cone length 2m, and cylinder length 4m.

```
>> plotObject(Targ)
```

View the resulting target object.

The API function arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.

targDesc [in]	A null terminating character array with a description of the target object to create; available options include targetboard, box, cube, cylinder, cone, ogive, hemisphere, and ellipsoid (all lower case). In the case of a conic (cone, ogive, hemisphere, and ellipsoid), if *nParam1* = 3, the third parameter is interpreted as the length of a cylinder to be added to the conic. Can also be the name of a .obj file in the directory returned by GETTARGDATAPATH to load a custom object in which case *param1* and *param2* are ignored.
param1 [in]	(NULL OK) A parameter having a different meaning for different target object classes; must be length *nParam1*; for targetboard, box, cube - lengths of each side, defaults to 1 in each direction; for cylinder - [radius, length] of cylinder, defaults to [1, 2∗radius]; for cone, ogive, hemisphere, ellipsoid - [bottom radius, height/length] of object, defaults to [1, 1], OR [bottom radius, height/length, cylinder length] of object which will add a cylinder to the conic.
nParam1 [in]	The number of elements in *param1*; can be 3 for a box or conic or 2 for anything else or 1 or 0 to take on some default value.
param2 [in]	(NULL OK) Another parameter having a different meaning for different object classes; always assumed to be length 1; for targetboard - square root of the total number of facets, defaults to 1; for box, cube - square root of the number of facets on a side, defaults to 1; for cylinder, cone, ogive, hemisphere, ellipsoid - number of facets around the diameter, defaults to 20.
nVertices [in,out]	Number of vertices allocated in output; if this number is incorrect, the right value will be stored here and the function will return, with an error if *vertices* and *facets* are not NULL.
nFacets [in,out]	Number of facets allocated in output; if this number is incorrect, the right value will be stored here and the function will return, with an error if *vertices* and *facets* are not NULL.
nVerticesPerFacet [in,out]	Number of co-planar vertices that each facet consists of; usually 4; if incorrect value input the correct value will be stored here and the function will return, with an error if *vertices* and *facets* are not NULL.
vertices [out]	Points in 3D, which together with the facets array, define the polygons that makes up the object; must be length 3 × *nVertices*; stored [v1x, v1y, v1z, ... vnx, vny, vnz] where n = *nVertices*.
facets [out]	Zero-based indices into the list of vertices defining a polygon in 3D; must be size *nFacets* × *nVerticesPerFacet*; indices defining the same polygon are assumed to be consecutive in memory.

zenith [out]	The direction designated as "up" or the opposite of gravity in the object's space; allocate to length 3.
motion [out]	The direction of motion of the target object in object space; allocate to length 3.
hitPoint [out]	The point (in object space) intended to be hit in further parts of the simulation; allocate to length 3.
trkPoint [out]	The point (in object space) intended to be tracked by a tracking sensor; allocate to length 3.
refLoc [out]	This will always be [0 0 0] in order for the above descriptions to make sense; allocate to length 3.
targRadius [out]	Gives a single measure for the approximate size of the target; must be length 1.
targLength [out]	Length of the target; must be length 1.

The API function returns system error codes; -1 for error, 0 for success. Within the API, call TargStruct with vertices and facets set to NULL to return *nVertices*, *nFacets*, and *nVerticesPerFacet* for allocation.

2.3.4 Functions for External Data

The functions in this section work with other toolboxes or data from other databases, not distributed with EngagementTools. This section has a description of the function CaseStruct, which generates the main structure used in SCALE and SHaRE. Some of the functions in this section are available for users who wish to use data from ISAAC and STAMP within the EngagementTools toolbox. Also in this section is TALOEngage for working with SCALE generated structures.

2.3.4.1 AddPolyFit

Syntax

Traj = AddPolyFit(*Traj, Order*)

This function adds polynomial fitting coefficients to an existing trajectory structure. The inputs are

Traj [struct]	Structure of Trajectory information from Boost2Flyout or a flyout structure from ABLPAT with LP (launch point) and trajectory data points.
Order [scalar]	Polynomial order.

The output, *Traj*, is a structure of Trajectory information with fields for the polynomial fit or a flyout structure - depending on the input.

This utility makes use of the Matlab function `polyfit` to generate a polynomial expansion of the altitude and ground range (from the launch point) as a function of time.

$$\begin{aligned} h(t) &= p_1^{(\mathrm{h})} t^n + p_2^{(\mathrm{h})} t^{n-1} + \ldots + p_{n+1}^{(\mathrm{h})} \\ r(t) &= p_1^{(\mathrm{r})} t^n + p_2^{(\mathrm{r})} t^{n-1} + \ldots + p_{n+1}^{(\mathrm{r})} \end{aligned}$$

where $h(t)$ is the altitude of the trajectory as a function of time, $r(t)$ is the down range of the trajectory from the launch point as a function of time, $\boldsymbol{p}^{(\mathtt{ht})}$ are the $n+1$ polynomials for altitude, $\boldsymbol{p}^{(\mathtt{rd})}$ are the $n+1$ polynomials for down range, and n is the polynomial order as set by *Order*. See the Matlab documentation for more information on `polyfit`.

2.3.4.2 BILLOAD

Syntax

AltMap = BILLOAD(*loadStr, [dispMap], [interpFlag]*)

This function loads digital terrain elevation data (DTED) from a Band Interleaved by Line (BIL) format file (Used by the SRTM mission). BIL formatted digital elevation data can be found at http://earthexplorer.usgs.gov/ by selecting the "SRTM" database under "Digital Elevation." Data sets are 1x1 degree and can be downloaded in either 1 or 3 arcsecond resolution. Using 3-arcsecond resolution is more time-efficient while 1-arcsecond data is more accurate. The inputs are

loadStr [string]	Location of the bil file or zip
dispMap [bool]	(Optional) Flag controlling whether image of the data is displayed. Default is false.
interpFlag [bool]	(Optional) Flag indicating whether to return interpolated map data. Default is false.

The output is

AltMap [struct]	Structure of elevation data from the input file
AltMap.Lat [vector]	Decimal Latitude of pixels' centers (deg)
AltMap.Long [vector]	Decimal Longitude of pixels' centers (deg)
AltMap.Alt [array]	Vertical deviation from WGS84/EGM96 geoid (m)
AltMap.hdr [struct]	Header information

Within the data files, there are some points that simply have no data for whatever reason. When the input *interpFlag* is true, the data will be interpolated using INPAINT_NANS to fill in the holes. To get back the original, un-interpolated data issue the command

```
>> AltMap.Alt(AltMap.hdr.isBad) = NaN;
```

2.3.4.3 Boost2Flyout

Syntax

[Flyout Traj] = Boost2Flyout(*StampDir*)

[Flyout Traj] = Boost2Flyout(*Booster*)

Boost2Flyout is a function to convert STAMP boost data to a Flyout structure for incorporation into a Targ structure in SCALE. To obtain geometry structures, use FlyoutGeom. There are two calling sequences. The only input is one of

StampDir [string]	STAMP/TGx Directory or boost data file.
Booster [struct]	STAMP/TGx Boost data from BoostLoad.

The outputs are

Flyout [struct]	Flyout structure.
Traj [struct]	Structure of Trajectory information - 1xNumStages.

2.3.4.4 BoostLoad

Syntax

Booster = BoostLoad(*fileName*)

This function reads a STAMP/TGx JNTF ASCII file and returns the metric data as a structure. The input, *fileName*, must include the path and the file name. The output is a structure containing STAMP Boost metrics as follows:

Booster [struct]	Structure containing STAMP/TGx Boost metrics.
Booster.stage [struct]	Contains stage metrics per booster.
stage.time [array]	Discrete times steps (sec).
stage.event [cell]	Discrete event table.
stage.pos [array]	Discrete ECI position vector (m).
stage.vel [array]	Discrete ECI velocity vector (m/s).
stage.acc [array]	Discrete ECI acceleration vector (m/s^2).
stage.thrust [array]	Discrete ECI *thrust* unit vector.

2.3.4.5 FortRead

Syntax

data = FortRead(*fileName*)

This function reads a fortran ASCII input file into a cell array while preserving white spaces. This routine allows the user to load an entire input file for easy parsing for Fortran data inputs/outputs, while maintaining new line formatting. The input *fileName* is the name of the Fortran data file with full path info. The output *data* is a cell array of data from the file.

2.3.4.6 LoadStampParams

Syntax

M = LoadStampParams(*ParamsFile*)

This function loads modeling parameters for STAMP from a given *ParamsFile* and returns them in the structure *M*.

2.3.4.7 TALOEngage

Syntax

TALO = TALOEngage(*C, [CloudFreeAlt], [condition]*)

This function computes the minimum time after lift off (*TALO*) for cloud-free engagement for a given geometry. The inputs are

C [struct]	Case structure from CaseStruct.
CloudFreeAlt [scalar]	(Optional) Altitude at which target breaks through clouds (m) - defaults to 0 m or the value in *C.Atm.CloudFreeAlt* if it exists. Care should be taken when using 0 for the cloud free altitude as round-off error may cause IsGoodGeom to return false when using the position corresponding to the *TALO* returned.
condition [scalar]	(Optional) 0 or empty for *CloudFreeAlt*, 1 for co-altitude with platform, 2 for elevation angle >=0.

The output *TALO* is the minimum time after lift off to engage the target in seconds. TALOEngage returns *Inf* if the platform is below the cloud deck.

2.3.5 Functions for Satellite Propagation (TLE)

The functions in this section are for satellite propagation, including the Sun and moon. Some of these functions were designed specifically for use with the http://www.space-track.org satellite catalog. There is an example script using some of these functions that is installed with this toolbox in EngagementTools/Examples.

2.3.5.1 ConvertTZ

Syntax

dvout = ConvertTZ(*TZin, dv, TZout, [fmt]*)

This function converts a given date and time from one time zone to another or to universal time [9]. If the output time zone is not specified, Universal Time is returned. Daylight saving time is handled automatically in the code, although care should be taken here as the start and end dates vary with location and date. The inputs are

TZin [cell]	Time zone of date, supported times zones are: UTC: 0 hr, Atlantic: UTC - 4hr (3hr DLS), Eastern: UTC - 5hr (4hr DLS), Indiana: UTC - 5hr (5hr DLS), Central: UTC - 6hr (5hr DLS), Mountain: UTC - 7hr (6hr DLS), Pacific: UTC - 8hr (7hr DLS), Alaska: UTC - 9hr (8hr DLS), Hawaii: UTC - 10hr (10hr DLS).
dv [vector/cell]	Date value in any acceptable MATLAB® date format - date number, date vector, or date string - see datevec and datenum. If a string (or cell of strings) is passed in and no format is defined, the format must be in one of the date formats 0, 1, 2, 6, 13, 14, 15, 16, 23 as defined by datestr. If passing in date numbers, or date vectors, each row is considered to be a different time/date.
TZout [string]	Desired output time zone.
fmt [string/scalar]	(Optional) Format string or format identifier for output time.

The output *dvout* is the time vector.

2.3.5.2 Gregorian2Julian

Syntax

JD = Gregorian2Julian(*GD, [JulCalFlag]*)

This function converts an epoch Gregorian date into the Julian date [9]. The code handles date vectors and date strings supported by MATLAB®. The inputs are

GD [varies]	Gregorian date in any acceptable MATLAB® date format - see datevec and/or datenum.
JulCalFlag [logical]	(Optional) Date is Julian, not Gregorian (rare).

The output *JD* is the corresponding Julian date.

2.3.5.3 JULIAN2GREGORIAN

Syntax

GD = JULIAN2GREGORIAN(*JD*)

This function converts a Julian date into the epoch Gregorian date [9]. The code handles date vectors and date strings supported by MATLAB®. The input is

JD [vector]	Date number(s) for Julian date(s).

The output is

GD [array]	Gregorian date vector - see datevec. Output is length(*JD*)x6.

2.3.5.4 LOS

Syntax

Site = LOS(*r1, r2*)

This function determines if there is line of site between two satellites assuming a circular earth [9]. This algorithm should only be used for altitudes above 100 km, For altitudes below 100km, use GEOMSTRUCT or SUNRISESET. Satellite positions should be in the ECF coordinate frame. LOS can determine if an asset is sunlit by passing in the satellite ECF position vector in as *r1* and the Julian date as *r2*. In that case *r2* must have the same dimensions as *r1*, except for the last dimension that contain the position vector. The inputs are

r1 [array]	Satellite 1's position vector in ECF coordinates (m).
r2 [array]	Satellite 2's position vector (m) or the Julian date for sunlit calculations.

The output is

Site [array]	0 - no line-of-site or in earth's shadow, or 1 - line-of-site or sunlit.

2.3.5.5 MoonPhase

Syntax

[Phase, DiskFrac] = MoonPhase(*JD*)

This function returns the lunar phase and the fraction of the lunar disk that is visible given a Julian date [9]. The input is

JD [array]	Julian date.

The outputs are

Phase [array]	Lunar phase (rad).
DiskFrac [array]	Fraction of lunar disk visible.

2.3.5.6 MoonPos

Syntax

[RECF, rM, RtAsc, Dec] = MoonPos(*JD*)

This function determines the moon position, right ascension and declination, and ECF position [9]. The input is

JD [vector]	Vector (N elements) of Julian Dates (days).

The outputs are

RECF [Nx3 matrix]	Lunar position in ECF coordinates (m).
rM [Nx3 matrix]	Lunar position in ECI coordinates (m).
RtAsc [vector]	Lunar Right Ascension (deg).
Dec [vector]	Lunar Declination (deg).

2.3.5.7 PROPTLE

Syntax

[G ECI] = PROPTLE(*TLE, startTime, elapsedTime, [GEOM]*)

[pos vel time] = PROPTLE(*TLE, startTime, elapsedTime*)

This function returns a geometry structure of ECF positions and velocities by propagating the input *TLE* set using the NORAD SGP4/SDP4 propagator [10]. The inputs are

TLE [struct]	*TLE* structure from TLEEXTRACT.
startTime [date]	Start time (in UT1) - can be any acceptable MATLAB® date format. If empty, *startTime* is the *TLE* Epoch Date.
elapseTime [vector]	Time vector for propagation from *startTime* or length of time to propagate from *startTime* (sec).
GEOM [struct/list]	(Optional) Geometry structure defining observer position and velocity as from GEOMSTRUCT or a list of (RPLLA,[VPLLA],[RE]). Defaults to LLA position and velocity equal [0 0 0].
RPLLA [vector]	(Optional) Observer LLA position [Lat Long Alt].
VPLLA [vector]	(Optional) Observer LLA velocity [Vxy Heading Vz].
RE [scalar/vector]	(Optional) Earth radius, defaults to PHYSICALCONST('reequator', 'repole').

The outputs are

G [struct]	Array of ECF geometry structures.
ECI [struct]	Contains the satellite position, velocity, and time data.
ECI.pos [array]	X, Y, Z position data (m).
ECI.vel [array]	X, Y, Z velocity data (m/s).
ECI.time [array]	Time required for conversion from *ECI* to ECF (sec).
pos [array]	X, Y, Z position data in *ECI* coordinates (m).
vel [array]	X, Y, Z velocity data in *ECI* coordinates (m/s).
t [array]	Time required for conversion from *ECI* to ECF (sec).

This code runs much faster when using the calling sequence with three outputs: matrices for position and velocity in ECI coordinates and time.

2.3.5.8 SDP4

Syntax

[SGPpos, SGPvel, SGPvals] = SDP4(*SGPinp, elapseTime*)

This function implements the NORAD deep space satellite position and velocity propagator [10]. Time samples are vectorized for quick generation, but the satellite ephemeris data is not. The inputs are

SGPinp [struct]	Processed TLE data, the output of SGPPREP.
elapseTime [array]	Elapse Time from ephemeris epoch (min).

The outputs are

SGPpos [struct]	Satellite's Cartesian position (ER).
SGPvel [array]	Satellite's Cartesian velocity (ER/min).
SGPvals [array]	Contains useful deepspace quantities.
SGPvals.n0dprime [scalar]	Recovered original mean motion (rad).
SGPvals.a0dprime [scalar]	Recovered original semimajor axis (ER).
SGPvals.perigee [scalar]	Perigee (km).

2.3.5.9 SGP4

Syntax

[SGPpos, SGPvel, SGPvals] = SGP4(*SGPinp, elapseTime*)

This function implements the NORAD low orbit satellite position and velocity propagator [10]. Time samples are vectorized for quick generation, but the satellite ephemeris data is not. The inputs are

SGPinp [struct]	Processed TLE data, the output of SGPPREP.
elapseTime [array]	Elapse Time from ephemeris epoch (min).

The outputs are

SGPpos [struct]	Satellite's Cartesian position (ER).
SGPvel [array]	Satellite's Cartesian velocity (ER/min).
SGPvals [array]	Contains useful deepspace quantities.
SGPvals.n0dprime [scalar]	Recovered original mean motion (rad).
SGPvals.a0dprime [scalar]	Recovered original semimajor axis (ER).
SGPvals.perigee [scalar]	Perigee (km).

2.3.5.10 SGPPREP

Syntax

SGPinp = SGPPREP(*tle*)

This function prepares extracted TLE data, such as output by TLEEXTRACT, for use by the NORAD SGP propagators (old FORTRAN convention) [10]. It also determines which SGP model to use based on the period. In general SGP requires these units: time in minutes, distance in units of earth radii, and angular quantities in radians. The input is

tle [struct]	Contains the TLE ephemeris data (See TLEEXTRACT).

The output is

SGPinp [struct]	Processed TLE data, the output of SGPPREP.
SGPinp.SSC [string]	Space Surveillance Catalog Number.
SGPinp.node0 [scalar]	Right ascension (rad).
SGPinp.omega0 [scalar]	Argument of perigee (rad).
SGPinp.M0 [scalar]	Mean anomaly (rad).
SGPinp.i0 [scalar]	Inclination (rad).
SGPinp.n0 [scalar]	Mean motion (rad/day).
SGPinp.e0 [scalar]	Eccentricity.
SGPinp.n0dotOvr2 [scalar]	First time derivative of Mean Motion /2.
SGPinp.n0ddotOvr6 [scalar]	Second time derivative of Mean Motion /6.
SGPinp.BSTAR [scalar]	*BSTAR* drag term.

SGPinp.EpoJD [scalar]	Epoch Julian Date.
SGPinp.Period [scalar]	*Period* (min).
SGPinp.DEEP [bool]	Deep space propagator flag.

2.3.5.11 SiderealTime

Syntax

[GMST, LST] = SiderealTime(*JD, [Long]*)

```
SCALING_CODE_API int SiderealTime(char** errorChain,
    const int nInputs, const double* jd, const double* lon, double* gmst,
    double* lst)
```

This function converts the Julian date into Greenwich Mean & Local Mean Sidereal Time.[9] Uses Julian date format given in Seidelmann, valid 1901-2199 AD.[11] The inputs are

JD [array]	Julian Date.
Long [array]	Longitude position (deg).

The outputs are

GMST [array]	Greenwich Mean Sidereal Time (deg).
LST [array]	(Optional) Local Mean Sidereal Time, defaults to 0 (deg).

2.3.5.12 SunPos

Syntax

[RECF, rS, RtAsc, Dec] = SunPos(*JD*)

This function uses the low precision formulas for the sun presented in section C of The Astronomical Almanac for determining the sun position, right ascension and declination.[11, 9] The precision there is given as one minute of arc, but the solar mean anomaly, longitude, etc. use the IAU 2000 methods, increasing the precision. Valid for years 1950-2050 AD. The input is

JD [vector]	Vector (N elements) of Julian Dates (days).

The outputs are

RECF [Nx3 matrix]	Solar position in ECF coordinates (m).
rS [Nx3 matrix]	Solar position in ECI coordinates (m).
RtAsc [vector]	Solar Right Ascension (deg).
Dec [vector]	Solar Declination (deg).

2.3.5.13 SunRiseSet

Syntax

Sun = SunRiseSet(*JD, Lat, Long, illumDef*)

This function computes the sun position, right ascension and declination, equation of *time*, rise and set times, etc. for an given date [9]. The inputs are

JD [array]	Julian Date.
Lat [array]	Latitude position, valid between -75 to 75 (deg).
Long [array]	Longitude position (deg) - uses military convention which has east longitudes as > 0 and west longitudes as < 0.
illumDef [string]	(Optional) Definition of solar illumination other than the standard definition (90 deg 50 min). Choose between 'Civil' - 96 deg (twilight), 'Nautical' - 102 deg, or 'Astro' - 108 deg.

The outputs are

Sun [struct]	Contains the sun vector information.
Sun.EqOfTime [array]	Solar Equation of Time (deg).
Sun.RtAsc [array]	Solar Right Ascension (deg).

Sun.Dec [array]	Solar Declination (deg).
Sun.Dist [array]	Solar Distance (m).
Sun.rS [array]	Solar position (m).
Sun.ECF [array]	Solar position in *ECF* reference (m).
Sun.Az [array]	Solar Azimuth (deg).
Sun.El [array]	Solar True Elevation (deg).
Sun.AtmEl [array]	Solar Refracted Elevation (deg).
Sun.cosZen [array]	Cosine of Zenith.
SolarNoon.time [array]	Solar Noon [UT hr, min, sec].
SolarNoon.JD [array]	Solar Noon [UT Julian date].
Rise.time [array]	*Sun* Rise [UT hr, min, sec].
Rise.JD [array]	*Sun* Rise [UT Julian date].
Set.time [array]	*Sun* Set [UT hr, min, sec].
Set.JD [array]	*Sun* Set [UT Julian date].
Sun.SNEqOfTime [array]	Solar Noon Equation of Time (deg).
Sun.SNRtAsc [array]	Solar Noon Right Ascension (deg).
Sun.SNDec [array]	Solar Noon Declination (deg).

2.3.5.14 TLEExtract

Syntax

[tle, tleMessage] = TLEExtract(*TLEstr, [SearchList]*)

This function extracts the Two Line Element (TLE) data from a TLE (two or three line) data array. Since the input is a data array, the input should be a 2x69 or 3X69 string array. If the check sum error flag is triggered, a note will be made in *tleMessage*. The inputs are

TLEstr [string/cell]	Two or three-line TLE or the location of the space-track.org satellite catalog.
SearchList [string/scalar]	(Optional) If a string, returns all TLE (3-line only) where the satellite names begins with SearchString. If a number, returns all TLEs (2 or 3-line) where the satellite number is SearchString. Only used if *TLEstr* is a TLE catalog file name or directory location.

The output is

tle [struct]	Structure containing the TLE data.
tle.SSC [string]	Space Surveillance Catalog Number.
tle.INTERNATIONAL_DESIGNATOR [string]	International designator. First two characters designate the launch year; the next 3 characters indicate the launch number, starting from the beginning of the year; the remainder of the characters (1 to 3 characters) indicate the piece of the launch.
tle.EPOCH_DATE [vector]	Epoch date of the element set in the form [YYYY MO DD HH MM SS.xxxx].
tle.MEAN_MOTION_DOT [scalar]	1st derivative of the mean motion with respect to time.
tle.MEAN_MOTION_DOT_DOT [scalar]	2nd derivative of the mean motion with respect to time.
tle.BSTAR [scalar]	B∗ drag term.
tle.ELEMENT_NUMBER [scalar]	Element number.
tle.INCLINATION [scalar]	Orbit inclination (deg).
tle.RIGHT_ASCENSION_OF_NODE [scalar]	Right ascension of ascending node (deg).
tle.ECCENTRICITY [scalar]	Orbit eccentricity.
tle.ARGUMENT_OF_PERIGEE [scalar]	Argument of perigee (deg).
tle.MEAN_ANOMOLY [scalar]	Mean anomaly (deg).
tle.MEAN_MOTION [scalar]	Mean motion (rev/day).
tle.EPOCH_REV [scalar]	Revolution number at epoch.
tle.CLASSIFICATION [string]	Classification of element set.
tle.TLE [string]	The two or three line element set.
tleMessage [string]	Notice of either - Satellite number mismatch between lines - Internal summation doesn't agree with TLE checksum value Either flag is a good indicator that the TLE isn't valid, but both are very rare.

2.3.5.15 TLEParse

Syntax

[TLEstr, TLEform] = TLEParse(*CatalogLoc, [SearchList]*)

This function loads TLEs from the given catalog file or the most recent data if a catalog directory is specified. It can also parse a TLE string array or file location into a structure of strings. It also pads any lines that fall

short of the needed 69 characters and handles two and three line element sets. If no file is passed in with the location string, the code will grab the latest file that has the word "catalog" in it (after the space-track.org convention) and you may pass in a gzip file directly. SearchParam is set for either "Name" or "SatNum" and the *SearchList* is a STRING array of the satellites of interest to extract and return in a smaller string array. This code was designed specifically for use with the space-track.org satellite catalog. The inputs are

CatalogLoc [string/cell]	A string with the catalog name and/or location. Can be a cell array of catalog files/locations.
SearchList [string/scalar]	(Optional) If a string, returns all TLE (3-line only) where the satellite names begins with SearchString. If a number, returns all TLEs (2 or 3-line) where the satellite number is SearchString.

The outputs are

TLEstr [string array]	Parsed TLE list.
TLEform [scalar]	Report a two or three line convention.

2.3.6 API Support Functions

This section describes functions in EngagementTools that support the MATLAB® interface with the ScalingCode API.

2.3.6.1 APImessage

Syntax

apimsg = APImessage

This function creates an API message object for logging error and warning messages from the Scaling Code API. It has no inputs. Upon return from a call to an API function, the API message object will contain any error or warning messages from the API which can be displayed in MATLAB®. The deepest error or warning will be first in the message. The error/warning can be displayed with the following command.

```
apimsg.throw(errorCode);
```

2.3.6.2 APIstruct

Syntax

obj = APIstruct(*xmlFile*)

This function returns an APIstruct object for interacting with the API models. This function can read in xml files for ParamStruct, TargStruct, AtmStruct, and GeomStruct. The input is

xmlFile [char]	Name of xml file to read in. Defaults to directory returned by PARAMPATH if path is not specified.

The output is

obj [APIstruct]	APIstruct object.
structType [char]	Type of structure from the xml file.
pointer [libpointer]	Lib pointer object required by direct calls to API functions.
xmlFileName [char]	Name of xml file loaded.

The following highlights some of the methods available on an APIstruct object.

APIstruct.get/setChar

Syntax

val = GETCHAR(*obj*, *paramName*)

SETCHAR(*obj*, *paramName*, *val*)

Methods used to get and set char array values within an API struct.

APIstruct.get/setDbl

Syntax

val = GETDBL(*obj*, *paramName*)

SETDBL(*obj*, *paramName*, *val*)

Methods used to get and set double values within an API struct.

APIstruct.get/setInt

Syntax

val = GETINT(*obj*, *paramName*)

SETINT(*obj*, *paramName*, *val*)

Methods used to get and set char array values within an API struct.

APIstruct.getXML

Syntax

val = GETXML(*obj*)

This method returns the XML text associated with pointer. This will include any changes to the structure made through SETCHAR, SETDBL, or SETINT, described above.

APIstruct.struct

Syntax

val = STRUCT(*obj*)

An overloaded version of Matlab function struct to convert APIstruct objects to an ATMTools structure.

APIstruct.writeXML

Syntax

WRITEXML(*obj, xmlFile*)

Write the data from the APIstruct object to an xml file. The file will be saved in the directory returned from PARAMPATH unless the a directory is specified in the input file name.

APIstruct.struct2xml

Syntax

S = APISTRUCT.STRUCT2XML(*S, xmlFile, [LEEDRLUTFile]*)

This is a static method to write data from a parameter structure *S* (of type P, G, Targ, or Atm) to an xml file for use with the API models. The optional input *LEEDRLUTFile* is used when the input structure is an Atm struct using LEEDR LUT data. The default directory to save the file is the directory returned from PARAMPATH. Pass the full file name to save in a different directory.

APIstruct.xml2struct

Syntax

S = APISTRUCT.XML2STRUCT(*xmlFile*)

This is a static method for importing xml files into a structure format. The default directory to load the file is the directory returned from PARAMPATH. Pass the full file name to load from a different directory.

Example 2.3.26 (APIstruct) *This example illustrates use of an APIstruct.*

```
>> ptr = APIstruct('source_GBTL.xml')
```

Loads the input xml file from default parameter structure path.

```
>> ptr.getDbl('HEL.Wavelength')
```

Return the value of the HEL wavelength

```
>> ptr.setDbl('Tx.TRK.BW',100)
```

Change the track bandwidth to 100

```
>> PS = struct(ptr)
```

Convert the APISTRUCT *object to a structure*

2.3.6.3 APIwarning

Syntax

S = APIwarning(*setting, warningID*)

Use this function to control and query warning message display from the Scaling Code API. This function works similarly to Matlab®'s function warning. The inputs are the desired setting and the warning message identifer. The output, if requested, is a structure with fields 'identifier' and 'state' with the state of the warning before calling the function. The input setting can be either 'on', 'off', or 'query'. Some warnings will need to be controlled using both APIwarning and warning as some Matlab® functions also check certain conditions. For example,

```
warning('off','Geom:IsGood')
APIwarning('off','Geom:IsGood')
```

turns off warnings about propagation geometry intersecting the surface of Earth both from Matlab® and from the API.

2.3.6.4 loadH5

Syntax

S = loadH5(*fileName, [var1], ..., [varN]*)

This function loads all available data from the specified h5 file and returns the data in a structure, *S*. The inputs are

fileName [string]	Name of h5 file to load. Must include the full path to the file if it is not on the Matlab® search path.
var1..varN [list]	(Optional) Names of variables to load. If omitted, all data will be loaded.

If an output is not specified, the data is put into the caller workspace.

2.3.6.5 loadMZAScalingCodeAPI

Syntax

[binDir, APIfolder] = loadMZAScalingCodeAPI(*APIfolder, [Ver]*)

This function loads the MZA Scaling Code API libraries for use in Matlab®. The input *APIfolder* should be the top level directory of the complied libraries and data directories. The user can specify to load either Release or Debug versions, but debugging only works with Visual Studio 2013.

2.3.6.6 ParamPath

Syntax

p = ParamPath(*[paramPath]*)

Use to set or return the default path to parameter (P, G, Targ, and Atm) xml files. To set the directory for xml file location, the input *paramPath* must be the full path to the directory. The output *p* will be the full path to the directory.

2.3.6.7 saveH5

Syntax

saveH5(*fileName, S*)

Save all data from the input structure to the specified .h5 file. Any existing file with the same name will be overwritten. The inputs are

fileName [string/cell]	Name of file to save. Can be name of a mat or cell array of mat files to convert to an h5 file with the same name.
S [struct]	(Optional) Structure of data to save. If not input, assumes fileName is a mat file to convert to h5.
locOpt [string]	(Optional) Input '-inplace' to save the h5 file in the same directory as the mat file. If data is input with '-inplace' option, h5 file will be saved in the working directory. Otherwise, h5 file will be saved in the default API directory.

2.3.6.8 TargDataPath

Syntax

targPath = TargDataPath(*[targPath0]*)

This function returns the default target data path. Can also be used to set or change the path. The target data path is used for loading target objects from .obj files with TargStruct. To set the directory for target object data, the input *targPath0* must be the full path to the desired directory. The output *targPath* will be the full path to the current directory.

2.3.6.9 unloadMZAScalingCodeAPI

Syntax

unloadMZAScalingCodeAPI

This function unloads all MZA Scaling Code API libraries from Matlab®.

2.3.7 Programming Utility Functions

The following functions are general utility functions contained in EngagementTools.

2.3.7.1 cell2csv

Syntax

cell2csv(*c, csvFile*)

This function writes a cell array of data to a .csv file. The inputs are

`c` [cell]	Cell array of data to write to file. Can be a cell array of cell arrays where each gets written to a separate file.
csvFile [char/cell]	File name or cell array of file names (if `c` is a cell array of cell arrays).

This function can be used in place of CELL2EXCEL when Microsoft Excel is not available.

2.3.7.2 CELL2EXCEL

Syntax

```
CELL2EXCEL(CellSheets, [sheetNames])
```

CELL2EXCEL loads data from a cell array into a Microsoft Excel®[2] Worksheet. The inputs are

CellSheets [cell]	Cell array to be displayed in Excel. If *CellSheets* is an array of cell arrays, each cell array will be displayed on a separate worksheet.
sheetNames [cell]	(Optional) Name for each sheet. Must be same length as *CellSheets* if *CellSheets* is an array of cell arrays.

The function optionally will output *SheetHandle* which is a handle to the Excel sheet. Use the function CELL2CSV when working on a Mac or when Microsoft Excel is not available.

2.3.7.3 CLICKABLELEGEND

Syntax

```
[LEGH, OBJH, OUTH, OUTM] = CLICKABLELEGEND(AX, {[string1], ...
    [stringN]}, 'group', grps, 'displayedLines', lines, ...)
```

CLICKABLELEGEND is a wrapper around the MATLAB® function LEGEND that provides the added functionality to turn on and off (hide or show) a graphics object (line or patch) by clicking on its text label in the legend. Call with no inputs to place a CLICKABLELEGEND on the current plot. Its usage is the same as the MATLAB® function. For further information please see the LEGEND documentation. Notes:

1. If you save the figure and re-load it, the toggling functionality is not automatically re-enabled. To restore it, simply call CLICKABLELEGEND with no arguments.
2. To prevent the axis from automatically scaling every time a line is turned on and off, issue the command: axis manual.

The inputs are

[2] Excel is a registered trademark of Microsoft Corporation in the United States and/or other countries.

AX [scalar]	(Optional) Axes handle for legend.
string1..stringN [string]	(Optional) Strings for legend labels. Also string1 could be a cell array containing all labels.
grps [vector]	(Optional) Must be preceded by keyword 'groups'. A vector specifying the group membership for every line object. The grouping parameter lets you have a group of *lines* represented and controlled by one entry in the legend.
lines [vector]	(Optional) Must be preceded by the keyword 'displayedLines'. A vector of indices corresponding to the *lines* or groups that should be displayed initially.

Additional inputs accepted by LEGEND, such as location, can be input after the input legend labels. The outputs are the same as for LEGEND.

LEGH [scalar]	A handle to the legend axes.
OBJH [vector]	A vector containing handles for the text, lines, and patches in the legend.
OUTH [vector]	A vector of handles to the lines and patches in the plot.
OUTM [cell]	A cell array containing the text in the legend.

The function CLICKABLELEGEND is currently used by VARYLINKPLOT and VARYLINKCOMPARE in the SHARE toolbox and by *EI* in the SCALE toolbox.

Example 2.3.27 (clickableLegend) *This example illustrates use of* CLICKABLELEGEND.

```
>> z = peaks(100);
>> plot(z(:,26:5:50)); grid on; axis manual;
>> clickableLegend({'Line1','Line2','Line3','Line4','Line5'}, ...
              'Location', 'NorthWest');

>> f = plot([1:10;1:2:20]','x'); hold on;
>> g = plot(sin([1:10;1:2:20]'),'r-');
>> h = plot(11:20,rand(5,10)*5,'b:');
>> clickableLegend([f;g;h], {'Line1','Line2','Line3'},...
               'groups', [1 1 2 2 3 3 3 3 3], 'displayedLines', [2 3]);
```

Use CLICKABLELEGEND *to group the two lines in* `f`*, the two lines in* `g` *and the five lines in* `h` *so that only three lines appear in the legend and all lines of the corresponding group are hidden when the legend entry is clicked.*

2.3.7.4 CompStruct

Syntax

[I Fnames Istruct] = CompStruct(*S1, S2, [TOL]*)

This function compares the content of 2 scalar structures with identical fields. The inputs are

S1 [struct]	First structure to be compared.
S2 [struct]	Second structure to be compared.
TOL [scalar]	(Optional) Tolerance for comparing numerical values. Defaults to *eps*.

The outputs are

I [logical vector]	Indicates fields where components are equal. Returns *NaN* if structures do not have identical fields. If field is a structure, returns true if the structures are identical, regardless of TOL.
Fnames [cell array]	Field names corresponding to the order of comparison in the output logical vector *I*. Always the order of the fields in the first input structure. Returns the set of field names that are not in both input structures if the inputs do not have identical fields.
Istruct [logical vector]	True if the field is a structure.

If the structures have the same fields, *I* is a vector whose length is the number of fields and consists of ones and zeros indicating the fields where the contents are equal within an optional specified tolerance using IsClose. The default tolerance is the machine epsilon (*eps*). *Fnames*{∼*I*} returns the field names where the contents of the two structures are not equal. This function tests substructures, but only returns true of they are identical rather than close within the specified tolerance. *Fnames*(*Istruct*) returns the field names that are substructures of the input structure.

2.3.7.5 Extract

Syntax

[Data, Vars] = Extract(*StructArray, [Var1], [Var2], ..., [VarN]*)

Extracts specified variables from the input *StructArray* and returns a single matrix with variables in each column. The inputs are

StructArray [struct]	Array of structures. All sub structures must be identical.
Var1..VarN [string/cell]	Name of variable to be extracted from *StructArray*. Any number of variables can be extracted. May be a cell array containing all variables to be extracted.

The outputs are

Data [double]	Matrix of extracted data. Each column is data for one variable. Size of *Data* will be length(*StructArray*)xN. If variable does not exist as a field, *Data* will be empty ([]).
Vars [cell]	Cell array of variables returned in *Data*.

If vector fields are to be extracted, each vectors will be a row vector in the output. Therefore fields that are row vectors in the input will be vertically concatenated for output. Fields that are column vectors will be transposed and then vertically concatenated.

Example 2.3.28 (Extract) *This example shows the use of* Extract *for getting data from an engagement structure.*

```
>> G = GeomStruct('Simple',[1000, 3000, 5000, 7000, 10000],50,10000);
>> S = EngagementStruct(G);
>> [Data,Vars] = Extract(S,'S.hp','S.L')

Data =

  1.0e+004 *

    0.1000    1.0046
    0.3000    1.0428
    0.5000    1.1162
    0.7000    1.2182
    1.0000    1.4112


Vars =

    'S.hp'    'S.L'
```

Extract platform altitude and slant range. The data can be visualized using the following commands:

```
    plot(Data(:,1),Data(:,2));
    xlabel(Vars{1}); ylabel(Vars{2})

>> RP = Extract(S,'G.RP')

RP =

  1.0e+006 *

    6.3720         0   -0.0100
    6.3740         0   -0.0100
```

```
6.3760         0   -0.0100
6.3780         0   -0.0100
6.3810         0   -0.0100
```

Example of extracting vector fields using EXTRACT *with* *S* *above.*

```
>> P = VaryStruct(ParamStruct,'Tx.D',(0.3:0.1:1.0), ...
      'Tx.DObs',0.1*(0.3:0.1:1.0));
>> Data = Extract(P,'Tx.D','Tx.DObs')
```

Extract aperture and obscuration diameter from a parameter structure.

This function is useful for extracting beam metrics from the structures used in the SHaRE and SCALE toolboxes. [Refer to the SHaRE and/or SCALE Users' Guides for information on the beam metrics structures.]

2.3.7.6 FINDSTRUCTNAN

Syntax

NaNFields = FINDSTRUCTNAN(*TestStruct*)

This function returns the names of fields within the input structure, *TestStruct* containing *NaN* as the field value. It does not test substructures. The output is a cell array containing the names of the *NaN* fields.

2.3.7.7 HLINE

Syntax

h = HLINE(*[ax], y, [linetype], [lbl]*)

This function draws a horizontal line on the current axes at the location specified by *y*. This function was written by Brandon Kuczenski and obtained from the Mathworks File Exchange. The inputs are

ax [handle]	(Optional) Axis handle. Defaults to current axes.
y [vector]	Y-axis values for horizontal lines.
linetype [char/cell]	(Optional) Simple line spec for the line color, style and symbol. Defaults to 'r:'. Can pass in a cell array of strings if *y* is a vector.
lbl [char/cell]	(Optional) Text label for each line. Will be the same color as the line. If not passed in, lines will have no label.

The line is held on the current axes, and after plotting the line, the function returns the axes to its prior hold state. The HandleVisibility property of the line object is set to "off", so not only does it not appear on legends, but it cannot be found by using FINDOBJ. Specifying an output argument causes the function to return a handle to the line, so it can be manipulated or deleted.

h [handle]	Handles for each line plotted.

Example 2.3.29 (Drawing horizontal lines) `>> h = hline(42,'g','The Answer')`

Returns a handle to a green horizontal line on the current axes at y =42, and creates a text object on the current axes, close to the line, which reads "The Answer".

```
>> hline([4 8 12],{'g','r','b'},{'l1','lab2','LABELC'})
```

Draws three horizontal lines with the corresponding labels and colors.

2.3.7.8 INPAINT_NANS

Syntax

B = INPAINT_NANS(*A, [method]*)

This function solves approximation to one of several pdes to interpolate and extrapolate holes in an array. This function was written by John D'Errico and obtained from the Mathworks File Exchange. The inputs are

A [array]	NxM array with some NaNs to be filled in
method [scalar]	(Optional) Scalar numeric flag - specifies which approach (or physical metaphor) to use for the interpolation. All methods are capable of extrapolation, some are better than others. There are also speed differences, as well as accuracy differences for smooth surfaces.

The available methods are described below.

- **method == 0** → (DEFAULT) see method 1, but this method does not build as large of a linear system in the case of only a few NaNs in a large array. Extrapolation behavior is linear.
- **method == 1** → simple approach, applies del^2 over the entire array, then drops those parts of the array which do not have any contact with NaNs. Uses a least squares approach, but it does not modify known values. In the case of small arrays, this method is quite fast as it does very little extra work. Extrapolation behavior is linear.
- **method == 2** → uses del^2, but solving a direct linear system of equations for nan elements. This method will be the fastest possible for large systems since it uses the sparsest possible system of equations. Not a least squares approach, so it may be least robust to noise on the boundaries of any holes. This method will also be least able to interpolate accurately for smooth surfaces. Extrapolation behavior is linear.
- **method == 3** → See method 0, but uses del^4 for the interpolating operator. This may result in more accurate interpolations, at some cost in speed.

- **method == 4** → Uses a spring metaphor. Assumes springs (with a nominal length of zero) connect each node with every neighbor (horizontally, vertically and diagonally) Since each node tries to be like its neighbors, extrapolation is as a constant function where this is consistent with the neighboring nodes.
- **method == 5** → See method 2, but use an average of the 8 nearest neighbors to any element. This method is NOT recommended for use.

2.3.7.9 IsClose

Syntax

T = IsClose(*A, B, [tol]*)

```
SCALING_CODE_API int IsClose(char** errorChain, const double* a,
    const int na, const double* b, const int nb, const double* tolerance,
    int* result)
```

This function checks two vectors to see if they are close within some level of tolerance. The inputs are

A [vector]	First value to check.
B [vector]	Second value to check.
tol [scalar]	(Optional) Tolerance. Defaults to *eps*. Use 0 for ==.

The default tolerance is the machine epsilon, *eps*. The output *T* will be one if A and B are close within *tol*, 0 if they are not. If one input is a scalar and the other is a vector, each element of the vector is compared to the scalar value and a logical vector is returned. If both are vectors and size(*A*) $\neq$ size(*B*), a logical scalar 0 is returned.

2.3.7.10 isDaylight

Syntax

[ID, SRS] = isDaylight(*dt, Lat, Long, [illumDef]*)

This function returns true if the input time is during daylight hours for the specified location. The inputs are

dt [datetime]	Date of observation. Can be a Matlab datetime object of a date vector. If not a datetime object, GMT is assumed.
Lat [array]	Latitude position, valid between -75 to 75 (deg).
Long [array]	Longitude position (deg) - uses military convention which has east longitudes as ¿ 0 and west longitudes as ¡ 0.
illumDef [string]	(Optional) Definition of solar illumination other than the standard definition (90 deg 50 min). Choose between 'Civil' - 96 deg (twilight), 'Nautical' - 102 deg, or 'Astro' - 108 deg.

The outputs are

ID [bool]	True if input time is during daylight hours.
SRS [struct]	Contains the sun vector information (see SunRiseSet).

2.3.7.11 LineIntersect

Syntax

[Int Px Py Len Para] = LineIntersect(*P1x, P1y, P2x, P2y*)

This function determines if and where 2D polylines intersect [12]. The inputs are

P1x [vector]	x axis locations of first polyline.
P1y [vector]	y axis locations of first polyline.
P2x [vector]	x axis locations of second polyline.
P2y [vector]	y axis locations of second polyline.

The outputs are

Int [P1xP2 logical array]	Intersections tag per polyline segment.
Px [P1xP2 array]	X axis intersect points.
Py [P1xP2 array]	Y axis intersect points.
Len [P1xP2 array]	Length to intersect.
Para [P1xP2 array]	Parallel tag per polyline segment.

2.3.7.12 MarkFig

Syntax

h = MarkFig(*[ax], MarkText, [Name1], [Value1], ..., [NameN], [ValueN]*)

This function adds text markings to figures. The inputs are

ax [scalar]	(Optional) Figure or axis handle to which to add text. Defaults to the MATLAB® current figure (gcf).
MarkText [string]	Text to add to figure.
NameN [char]	Property name for input to MATLAB® function text. Can specify things like 'Color' or 'FontSize'.
ValueN [varied]	Value associated with *NameN* for input to MATLAB®function text.

The function can return the object handle to the text box for use in changing font size/color, etc.

Example 2.3.30 (Marking figures) *The examples below show different ways to call* MARKFIG *to mark figures.*

```
>> MarkFig('FOUO')
```

Mark the current figure as FOUO with the default color and font size.

```
>> MarkFig('My label','Color','b','FontSize',10)
```

Mark the current figure specifying the color and font size.

2.3.7.13 MCODEHELP

Syntax

MCODEHELP(*functionName*)

This function displays the m-code equivalent of the calculations made in the API library for a specified function if the m-code has been made available. With no output, the code is displayed in the MATLAB® command window.

2.3.7.14 PDFOPEN

Syntax

PDFOPEN(*fileName, sectName*)

Open a Adobe pdf document and display the page referred to by a named destination in the document. If *sectName* is not specified, the MATLAB® function open is used. If used to open any of the user's guides for the scaling code toolboxes, a prefix is required on the function name. An 's' will link to the section in the main part of the document and a prefix of 'm' will link to the section in the Appendix.

fileName [string]	File name with full or relative path information. May contain section name (see example below).
sectName [string]	(Optional) Name of destination in pdf document.

2.3.7.15 PROGRESSBAR

Syntax

[flag, h, callbackFcn] = PROGRESSBAR(*v, h, title*)

This function can be used to create and update a progress bar using MATLAB®'s waitbar function. Includes a cancel button. The inputs are

v [vector]	Vector containing the index of the current iteration and the total number of iterations. May contain multiple index/total pairs. Pass as [] to initialize a new progress bar.
h [handle]	Wait bar handle to update. Only required if *v* is not empty. Must be created using MATLAB® function waitbar. Pass as empty ([]) to specify window *title*.
title [string]	(Optional) Title for progress bar window. Can include '%' to be replaced with the percentage complete based on the input vector.

The outputs are

flag [logical]	Flag indicating whether update of the waitbar was successful.
h [handle]	Object handle for the waitbar.
callbackFcn [function_handle]	Function handle for updating the waitbar that can be passed to other functions that allow a callback function input.

Outputs a function handle that can be used as a callback function.

Example 2.3.31 (Progress Bar) *The following examples illustrate use of the function* PROGRESSBAR.

```
>> [flag, h, callbackFcn] = progressbar([], [], 'Running (%)');
```

Initialize a progress bar. The output handle h *can be used to modify the progress bar in any way.*

```
>> flag = progressbar([2,10],h,'Running (%)');
```

Pass loop indices to update the progress bar. Output flag *will be false if the user pushed the cancel button.*

```
>> flag = callbackFcn([2 10 2 3])
```

Same as above, except that it uses the function handle to access the progress bar. Also includes indices for multiple/nested loops.

```
>> RunMySimulation(parameters, callbackFcn)
```

Pass the function handle for updating the progress bar to a separate MATLAB® *function that checks and updates the progress bar.*

2.3.7.16 PSDGenRandom

Syntax

[Xre, Xim] = PSDGenRandom(*t, tc, SigSTD*)

This function generates a temporally-correlated random signal from a PSD. The inputs are

t [vector]	Time (s).
tc [scalar]	Temporal correlation length (s).
SigSTD [scalar]	Standard deviation of the signal.

The outputs are

Xre [vector]	Real part of the random signal. *Xre* and *Xim* will have the same length as *t* with same units as *SigSTD*.
Xim [vector]	Imaginary part of the random signal.

Example 2.3.32 (Generating Random Signals) *This example illustrates the generation of both 1 and 2 variable random signals.*

```
>> Xre = PSDGenRandom(0:10,1,1e-6)
```

Generate a single variable random signal.

```
>> [x,y] = PSDGenRandom(0:100,5,1e-6)
```

Generate a dual variable random signal, such as a jitter.

2.3.7.17 ROTATETICKLABEL

Syntax

th = ROTATETICKLABEL(*h, [rot], demo*)

This function rotates the x-axis tick labels by a specified angle. It was written by Andrew Bliss and obtained from the Mathworks File Exchange.The inputs are

h [scalar]	Handle to the axis that contains the x-tick labels that are to be rotated.
rot [scalar]	(Optional) Rotation angle to apply (deg). Defaults to 90.
demo [char]	String '*demo*' to show a demo figure.

For long strings such as those produced by DATETICK, you may have to adjust the position of the axes so the labels don't get cut off.

th [vector]	Handles to the text objects created.

2.3.7.18 STRUCT2VEC

Syntax

[v, names] = STRUCT2VEC(*S*)

This function converts field values in *S*, where *S* is any input structure, to a vector of values. The function will also return a cell array of the field names. If anything but a structure is passed in, the output will be the same as the input.

Example 2.3.33 (Structure to vector) *This example shows how to use* ***struct2vec*** *to convert a structure to a vector.*

```
>> S = struct('a',1,'b',2,'c',3)

S =

    a: 1
    b: 2
    c: 3

>> [v,c] = struct2vec(S)

v =
```

```
     1
     2
     3


c =

    'a'
    'b'
    'c'
```

2.3.7.19 structArray2arrayStruct

Syntax

S = structArray2arrayStruct(*s, [depth]*)

This function converts a scalar structure with array fields to an array of structures. This is used by loadH5 to decompress Atm and Geom structures saved to h5 files. The inputs are

s [struct]	Scalar structure with array fields.
depth [scalar]	(Optional) Depth to decompress. If *depth*=1, assumes *s* contains array fields to decompress. if *depth*=2, assumes *s* contains structure fields that are to be decompressed.

2.3.7.20 syntaxHelp

Syntax

syntaxHelp(*functionName*)

This function will display the syntax (first few lines of the help) given the input function name. For more information about the function's inputs, use the MATLAB® command `help`. If the comment format is not consistent with this toolbox, the first 5 lines of comments are displayed. The input is

functionName [string]	Name of function for which to display syntax.

Example 2.3.34 (Display Syntax) *As an example, the syntax for a couple of functions is displayed.*

```
>> syntaxHelp CaseStruct
  SYNTAX:
  Cstruct = CaseStruct(P,G,Targ,[Atm],[lambda])
     or to update an existing output:
  Cstruct = CaseStruct(C,[FieldName1],[FieldValue1],...
```

```
                [FieldName2],[FieldValue2],...)

    Reference page in Help browser
      doc CaseStruct

  >> syntaxHelp FlyoutGeom
    SYNTAX:
    G = FlyoutGeom(C,TALO,[RPFlag],[trackStr])

    Reference page in Help browser
      doc FlyoutGeom
```

2.3.7.21 VaryStruct

Syntax

Strct = VaryStruct(*StrctIn, Var1, Var1Values, Var2, Var2Values, ...*)

This function will create an array of structures from the input structure given a vector of values to be set, or set fields in an array of input structures. Any substructures referenced in *VarN* must be present in the input structure, but scalar fieldnames need not be present. All variables must have the same number of values, and must be the same length as the input structure, if it is an array. The inputs are

StrctIn [struct/array]	Structure or array of structures.
Var1...VarN [string/cell]	String or cell array of strings identifying fields within *StrctIn* to be changed.
VarNValues [array/cell array]	Values for the specified fields in the input structure.

The output is

Strct [struct array]	Array of structures, one element per value specified.

2.3.7.22 VectorArgCheck

Syntax

argsOut = VectorArgCheck(*['ndims', ndims], argsIn*)

This function takes in a list of arguments and makes them all the same size. If the optional input *ndims* is one, the outputs will all be either rows or columns based on the first vector input. If *ndims* is two or unspecified, it will look at the maximum size in dimension 1 (N1) and the max size in dimension 2 (N2) of all the inputs, with the exception of strings. The outputs will be of size N1xN2. Character arrays will not be repeated. If only a single output is specified the outputs will be returned as a cell array.

Example 2.3.35 (VectorArgCheck) *Some examples of using* VectorArgCheck.

```
>> NumGeoms = 10;
>> G = FlyoutGeom(GeomStruct,linspace(0,100,NumGeoms));
>> [RP,re] = VectorArgCheck(vertcat(G.RP),LocalEarthRadius(G))
```

Outputs will be NumGeoms x 3.

```
>> Atm = AtmStruct;
>> [alpha,L,z,Cn2] = VectorArgCheck([1 2 3],Atm.L,Atm.z,Atm.Cn2)
```

Atm is a scalar structure from AtmStruct. *The outputs will be length(Atm.z) × 3. This is the same as VectorArgCheck('ndims',2,[1,2,3],Atm.L,Atm.z,Atm.Cn2)*

```
>> Diam = 1:3; DObs = .25; r0 = [.1 .2 .3]';
>> [Diam,DObs,r0] = VectorArgCheck('ndims',1,Diam,DObs,r0);
```

Output will have one non-singleton dimension, i.e. the size will be 1xN or Nx1 depending on the size of the first vector input.

2.3.7.23 vline

Syntax

h = vline(*[ax], x, [linetype], [lbl]*)

This function draws a vertical line on the current axes at the location specified by *x*. This function was written by Brandon Kuczenski and obtained from the Mathworks File Exchange. The inputs are

ax [handle]	(Optional) Axis handle. Defaults to current axes.
x [vector]	X-axis values for vertical lines.
linetype [char/cell]	(Optional) Simple line spec for the line color, style and symbol. Defaults to 'r:'. Can pass in a cell array of strings if *x* is a vector.
lbl [char/cell]	(Optional) Text label for each line. Will be the same color as the line. If not passed in, lines will have no label.

The line is held on the current axes, and after plotting the line, the function returns the axes to its prior hold state. The HandleVisibility property of the line object is set to "off", so not only does it not appear on legends, but it cannot be found by using findobj. Specifying an output argument causes the function to return a handle to the line, so it can be manipulated or deleted. The output is

`h` [handle] Handles for each line plotted.

Example 2.3.36 (Drawing vertical lines) `>> h = vline(42,'g','The Answer')`

Returns a handle to a green vertical line on the current axes at x*=42, and creates a text object on the current axes, close to the line, which reads "The Answer".*

```
>> vline([4 8 12],{'g','r','b'},{'l1','lab2','LABELC'})
```

Draws three vertical lines with the corresponding labels and colors.

3. ATMTools

The ATMTools functions can be separated into 4 categories, atmospheric characterization, atmospheric modeling, specialized mathematical functions, and tempus™mex system functions. The atmospheric characterization category includes functions for computing parameters such as the Fried parameter (r_0), Rytov number (σ_χ^2), isoplanatic angle (θ_0). The atmospheric modeling category includes functions for generating profiles of various atmospheric parameters such as the refractive index structure constant (C_n^2), absorption, scattering, wind, temperature, pressure, and density. The specialized mathematical function category includes general functions used by other functions within the toolbox, such as the generalized hypergeometric function ${}_pF_q(a_1,\ldots,a_p;b_1,\ldots,b_q;z)$. The final category is the tempus™mex system files. This category includes functions for interfacing tempus™mex system wave-optics simulations generated using WaveTrain™.

3.1 Layout

The remainder of the introduction contains some examples of how to use this toolbox. Section 3.2 shows the user how to generate the main component used by the toolbox, the atmospheric structure. Section 3.3 goes through a sample of calculations that one can perform with ATMTools.

Section 3.4 gives detailed descriptions of the functions available in ATMTools and should primarily be used as a reference if the user needs more detailed information about specific functions. The functions in this section have been separated into sections with the atmosphere structure in Section 3.4.1, atmospheric characterization functions in Section 3.4.2, atmospheric modeling functions in Section 3.4.3, functions for working with LEEDR in Section 3.4.4, specialized mathematical functions in Section 3.4.6, and general utility functions in Section 3.4.8.

3.2 Getting Started

The toolbox is built on the generation of a MATLAB® structure[1] for atmospheric characterization. The atmospheric structure contains top level information about the propagation path (geometry) and the parameters of interest along the propagation path, such as the index of refraction structure coefficient and the absorption coefficient. The atmospheric structure is generated by the function ATMSTRUCT [see Section 3.4.1.1 for more information about ATMSTRUCT]. The only information required by this function is information about the geometry (*hp*, *ht*, and *rd*) and information about the phase screens (number and location). Example 3.2.1 shows how to generate *Atm*

Example 3.2.1 (Generating *Atm* with input list of geometry) *This example shows how to generate an atmospheric structure by inputting the geometry as a list.*

```
>> Atm = AtmStruct(1000,10,20000,10)

Atm =

    hp: 1.0000e+003
    ht: 10
    rd: 2.0000e+004
     L: 2.0026e+004
     z: [10x1 double]
    dz: [10x1 double]
     h: [10x1 double]
```

Atmospheric structure with ten equally-spaced, equal-thickness phase screens.

[1] For help on MATLAB® structures type HELP STRUCT at the MATLAB® prompt.

The atmospheric structure can also contain information about C_n^2 , absorption, scattering, wind, temperature, pressure, and density. The inputs for each desired model would include first a string to identify the desired model type, 'Cn2', 'Wind,' 'Abs,' 'Scat,' etc. The next input is the name of the function to be used to calculate the profile. This can be any function on the MATLAB® path with altitude as the first argument. Available models in ATMTools can be found in Section 3.4.3 or by issuing the command `AtmModels` at the MATLAB® command prompt. After the function name should be a list of any parameters (other than altitude) required by that function. Example 3.2.2 shows how to generate an *Atm* with profiles for C_n^2 and wind.

Example 3.2.2 (Generating *Atm* with AtmStruct) *This example shows the generation of an atmospheric structure with profiles of structure constant and wind.*

```
>> Atm = AtmStruct(1000,10,20000,8,'Cn2','HV57','Wind','UniformAtm',5)

Atm =

          hp: 1000
          ht: 10
          rd: 2.0000e+04
           L: 2.0026e+04
    LatLong: [1x1 struct]
     MaxAlt: Inf
     xRange: [0 1]
           z: [8x1 double]
          dz: [8x1 double]
           h: [8x1 double]
         Cn2: [8x1 double]
    Cn2Eval: {'HV57'  'h'}
        Wind: [8x1 double]
   WindEval: {'UniformAtm'  'h'  [5]}
```

Atmospheric structure with 8 equally-spaced, equal-thickness phase screens. Notice the additional parameter required by UniformAtm.

This structure can also be generated using a geometry structure from GeomStruct‡ [see Example 3.4.1 in Section 3.4.1.1].

3.3 Using the Toolbox

After the atmospheric structure and the geometry and engagement structures have been calculated, one can perform any number of calculations using the functions in this toolbox. Section 3.3.0.24 shows how to perform some basic atmospheric calculations, such as Fried coherence length (r_0) and Greenwood frequency (f_G). Section 3.3.0.25 describes briefly how to use the graphical user interfaces available in ATMTools. Section 3.3.0.26 illustrates use of ATMTools outputs in WaveTrain™ simulations.

3.3.0.24 M-file or Command-Line Computations

Section 3.4.2 has functions for computing atmospheric propagation parameters, such as the Fried parameter (r_0), Rytov number (σ_χ^2), and isoplanatic angle (θ_0) and Greenwood and Tyler frequencies. There are also functions in Section 3.4.2 for calculating PSD's of phase abberations and for calculating irradiance after propagation. The function TurbConst gives exact values of constants commonly used in atmospheric propagation calculations. Example 3.3.1 shows how to calculate Fried parameter and Greenwood frequency.

Example 3.3.1 (Calculation of atmospheric propagation parameters) *This example shows how to calculate r_0 and the Greenwood frequency*

```
>> r0 = SphericalR0(1, 1.064e-6,  Atm)

r0 =

    0.1187
```

Fried parameter calculated with turbulence profile multiplier of one and wavelength of 1.064 μm, and atmospheric structure (`Atm`) as given in Example 3.2.2. See Section 3.4.2.30 for a detailed description of SphericalR0.

```
>> fG = GreenwoodFreq(1, 1.064e-6,S,Atm)

fG =

  430.3621
```

Greenwood frequency calculated with turbulence profile multiplier of one and wavelength of 1.064 μm and `S` is an engagement structure from EngagementStruct‡. *See Section 3.4.2.11 for a detailed description of* GreenwoodFreq.

3.3.0.25 Using the Graphical User Interface

ATMTools comes with two graphical user interfaces (GUIs) for setting up atmospheric parameters and propagations. AtmStruct_Interface supports atmospheric setup within the GUI framework for SHaRE and SCALE and contains fields for setting atmospheric profiles and basic geometry parameters. To load with default parameters type `AtmStruct_Interface` at the MATLAB® command prompt. Figure 3.1 shows a screen shot of AtmStruct_Interface with the default settings. As illustrated in the figure, tooltips show the calling sequence for the specified model function and additional function help can be obtained via a right+click context menu on the model name field. The model list drop down menus contain all functions available in ATMTools for the specified model type, however, any function available on the MATLAB® path that takes in a vector of altitudes as the first argument can be specified by typing the function name in the model name box and any additional parameters in the model parameters box for the given model type. Section 3.4.1.1 explains how to call AtmStruct_Interface with a pre-computed atmospheric structure.

PropConfig contains all the functionality of the AtmStruct_Interface and much more, including more advanced geometry and atmospheric setup as well as the ability to provide guidance in setting mesh parameters for wave optics simulations. PropConfig can be used to simply modify existing atmosphere and geometry structures, but the primary output of PropConfig is a MATLAB® data file that contains all the atmosphere and geometry information as well as mesh parameters and phase screen data for use in a wave-optics simulation. One can also save a data file without opening the GUI, using default options for computing mesh parameters by passing as the first argument the string `'SaveFile'` followed by a `C` structure and the name of the file to save.

```
>> PropConfig('SaveFile',C,'mypropconfig.mat')
```

The file will be saved to the current working directory unless the input file name includes the path.

Figures 3.2 through 3.7 show different aspects of the PropConfig interface. Some additional help on getting started with PropConfig is available in `PropConfigTutorial.pdf` in the ATMTools/doc directory.

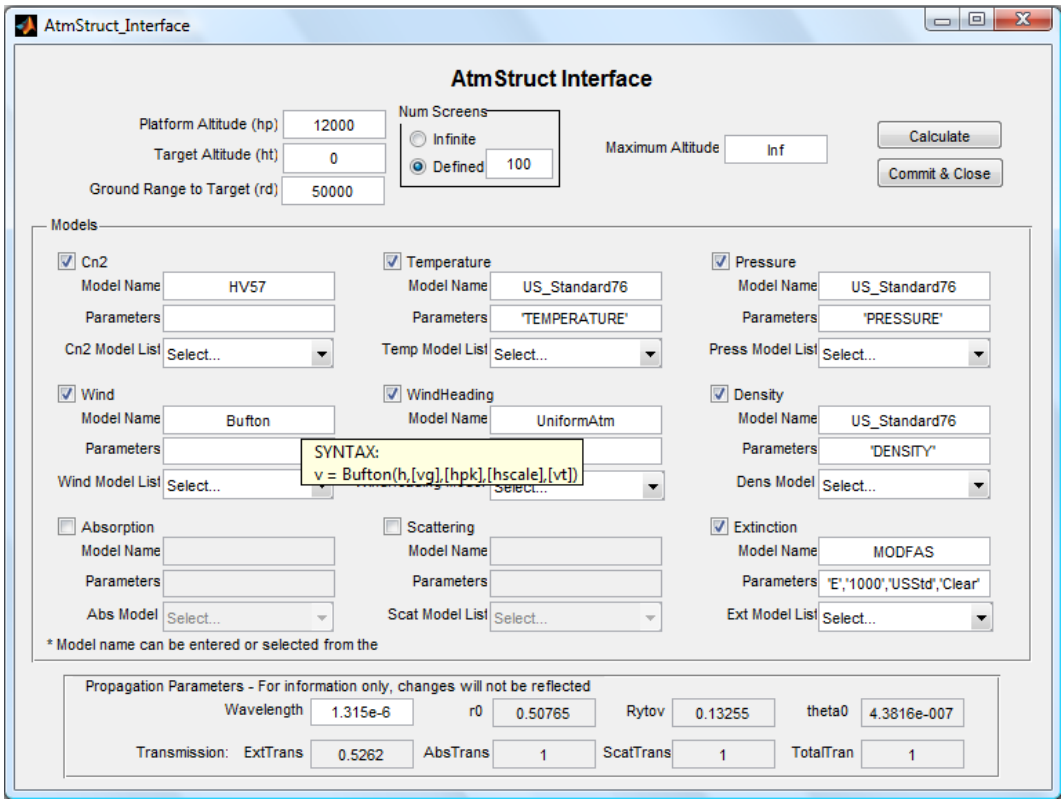


Figure 3.1: User interface for the atmospheric structure used in ATMTools.

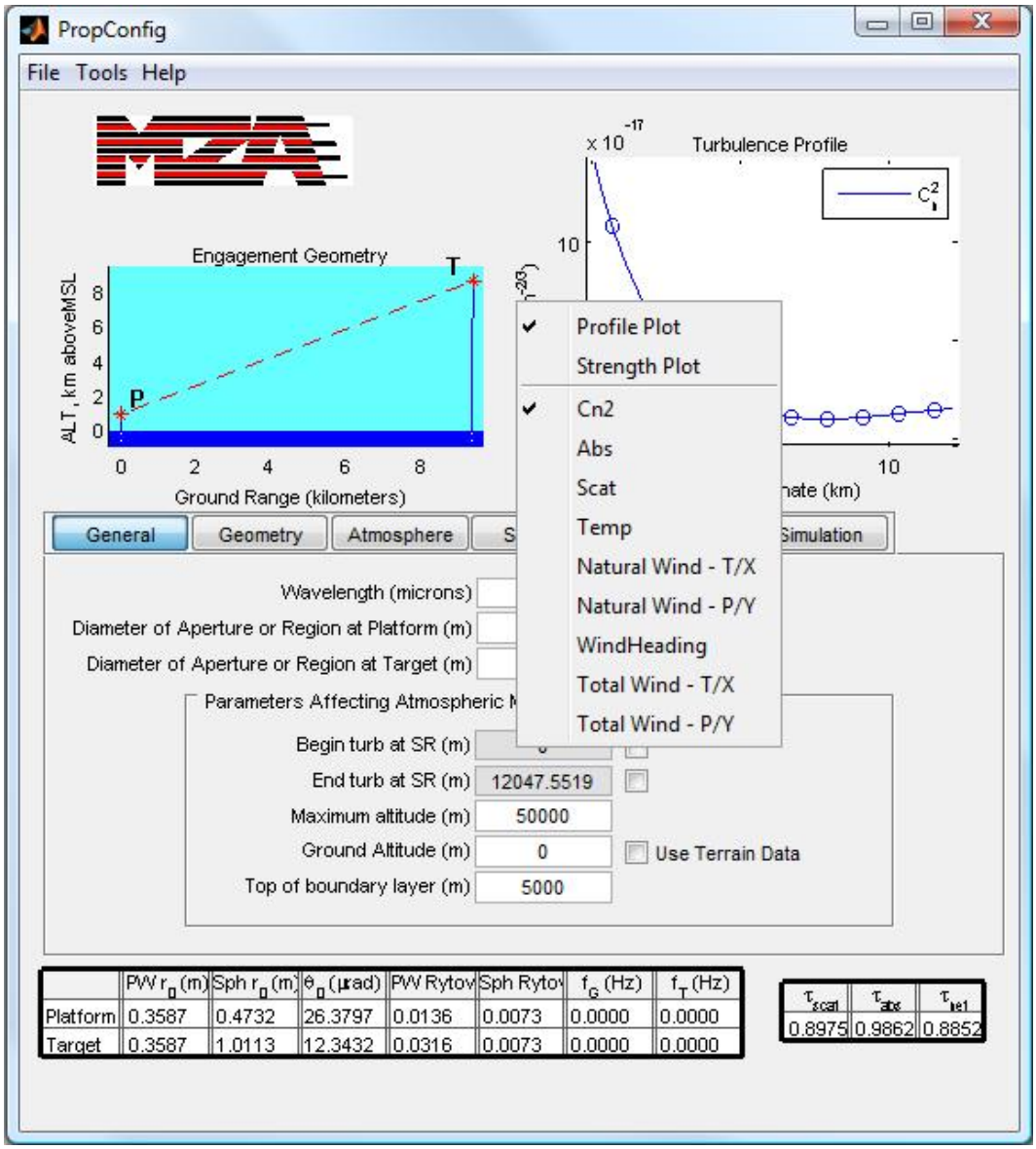


Figure 3.2: PropConfig **window open in the "General" tab with default parameters. Shows a context menu for the profile plot available via right+click on the y-axis label.**

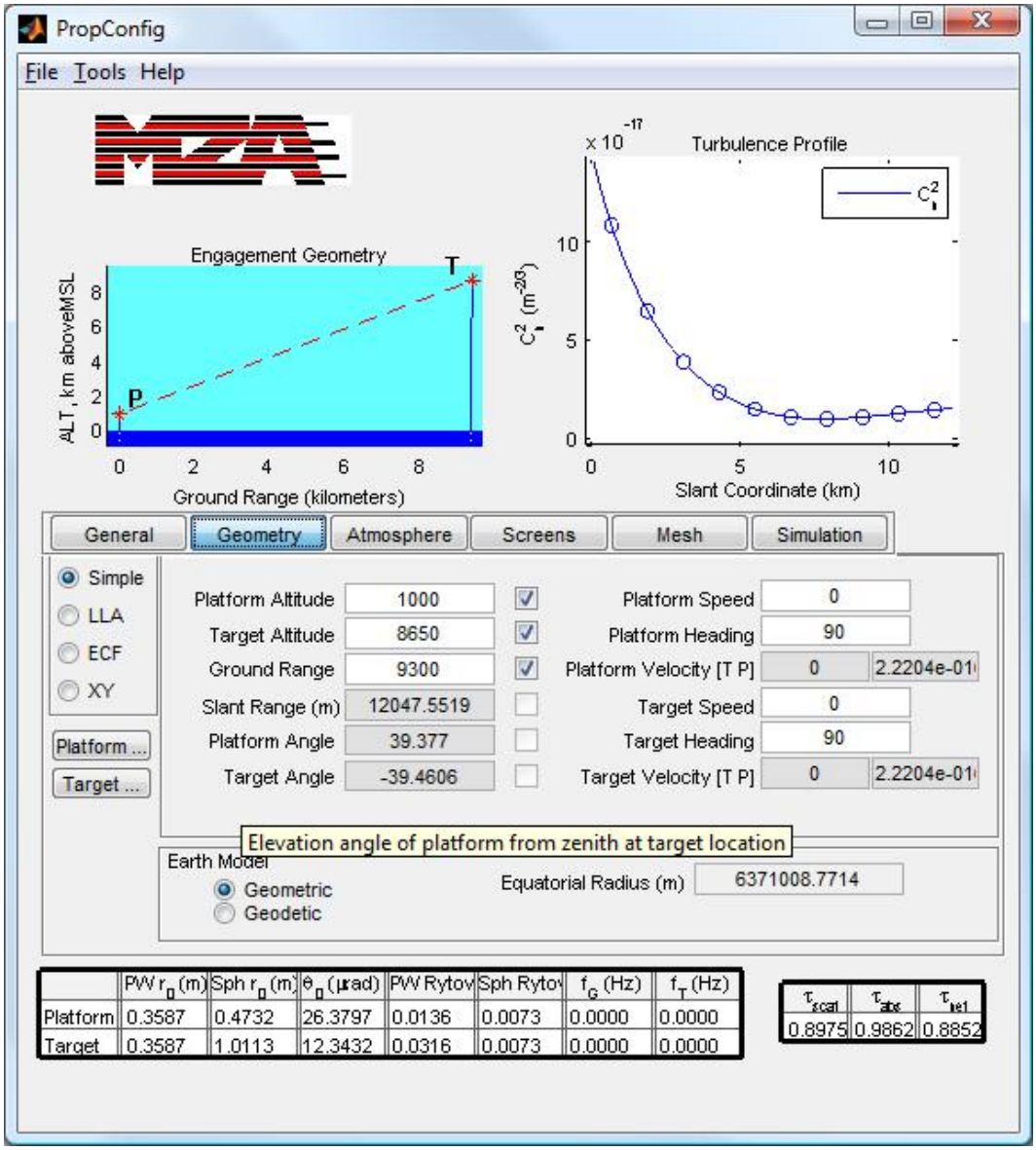


Figure 3.3: PropConfig **window open in the "Geometry" tab with default parameters. Shows tooltips available on most fields.**

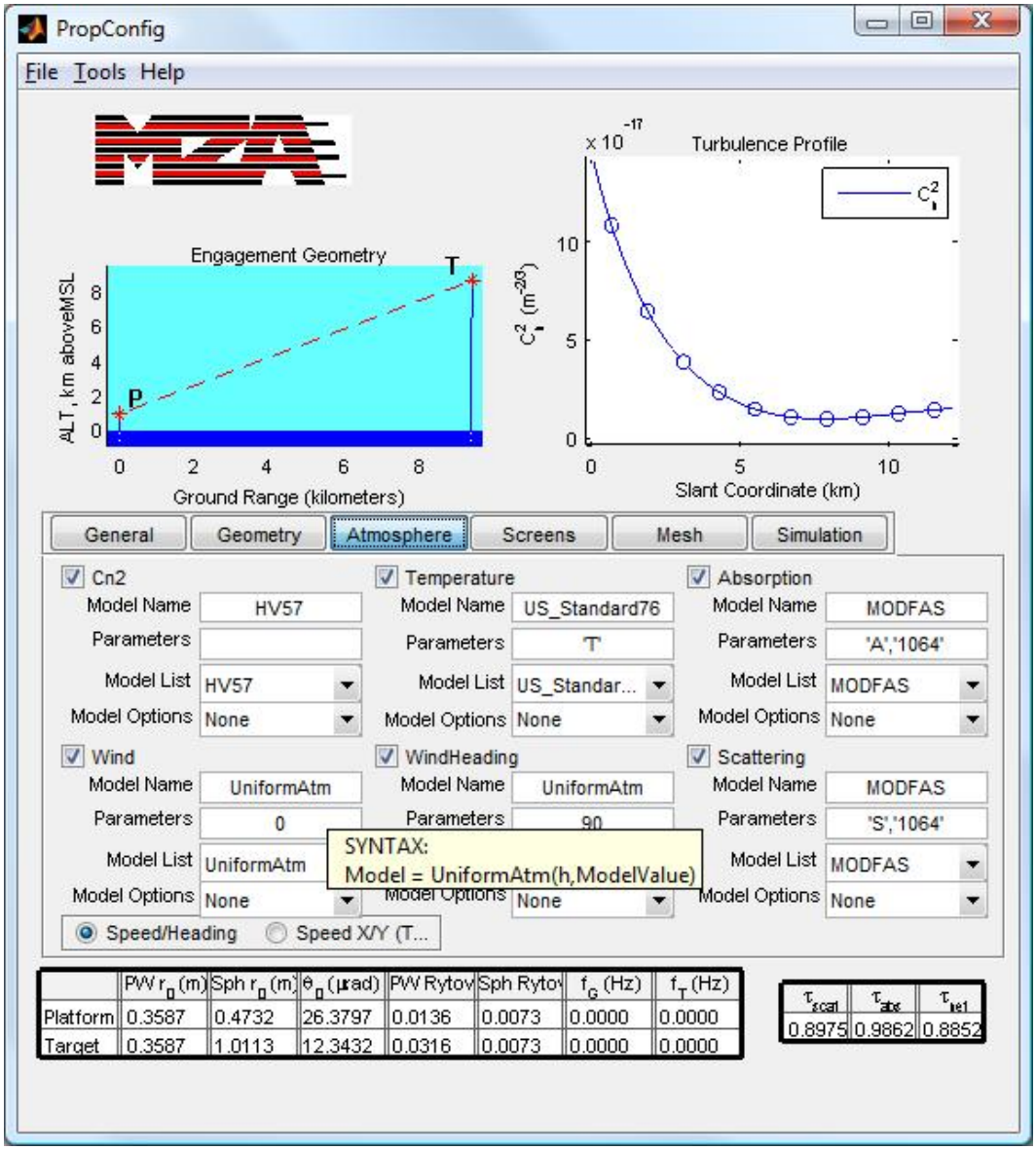
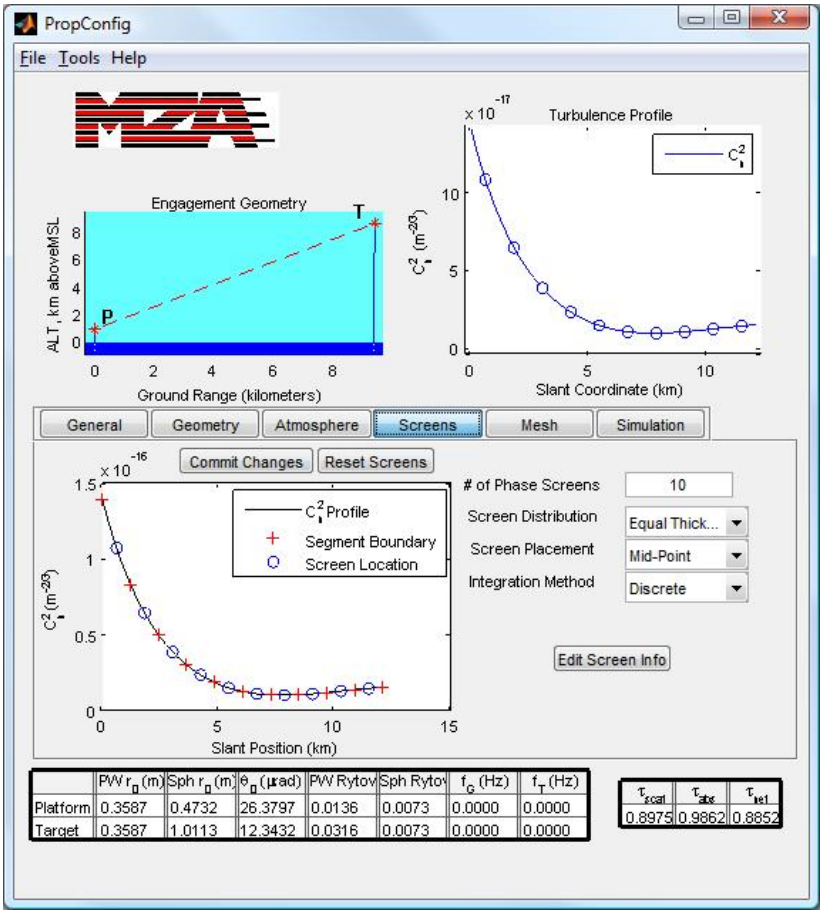
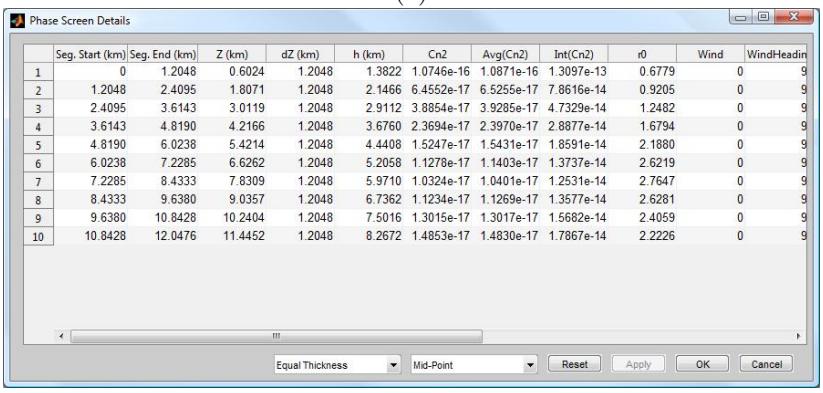


Figure 3.4: PropConfig **window open in the "Atmosphere" tab with default parameters. Tooltips on Atmosphere tab display syntax help for the model function being used.**



(a)



(b)

Figure 3.5: (a) PropConfig window open in the "Screens" tab with default parameters. Allows modification to the phase screen locations and distribution. Custom specification can be done via (b) the phase screen detail table accessible with the "Edit Screen Info" button.

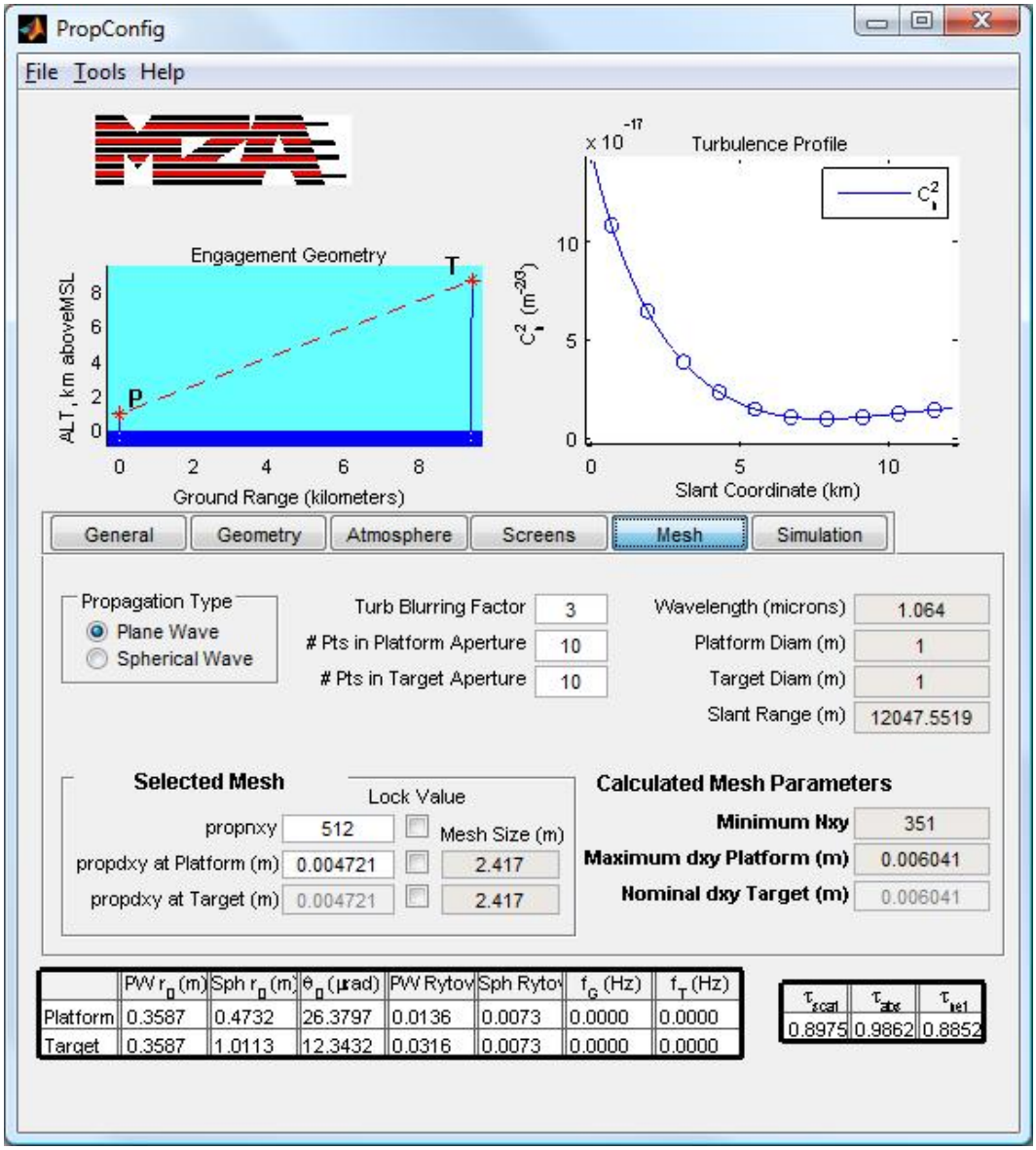


Figure 3.6: PropConfig **window open in the "Mesh" tab with default parameters. Contains parameters controlling optimal grid size and spacing for WaveTrain™ propagation mesh.**

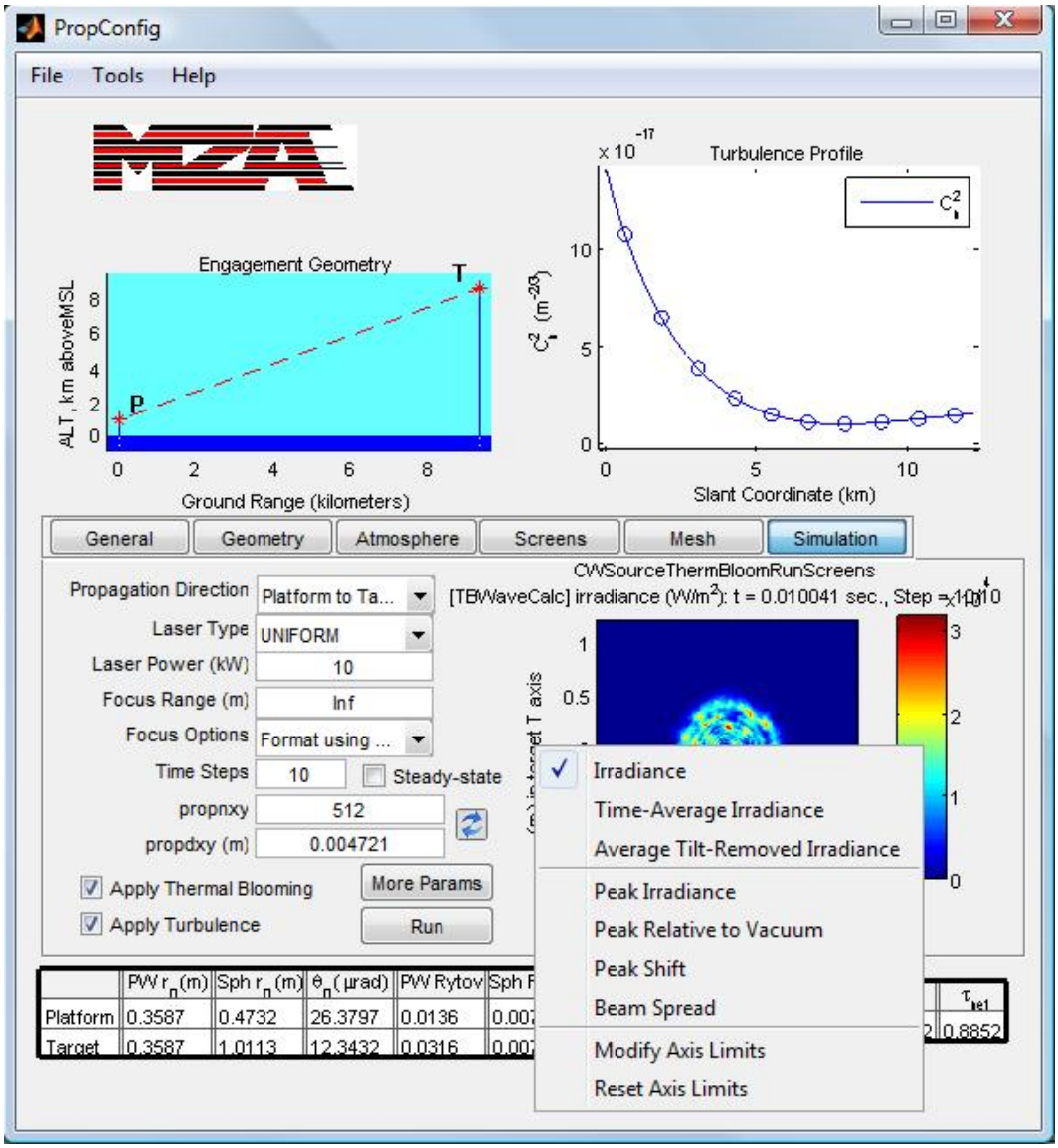
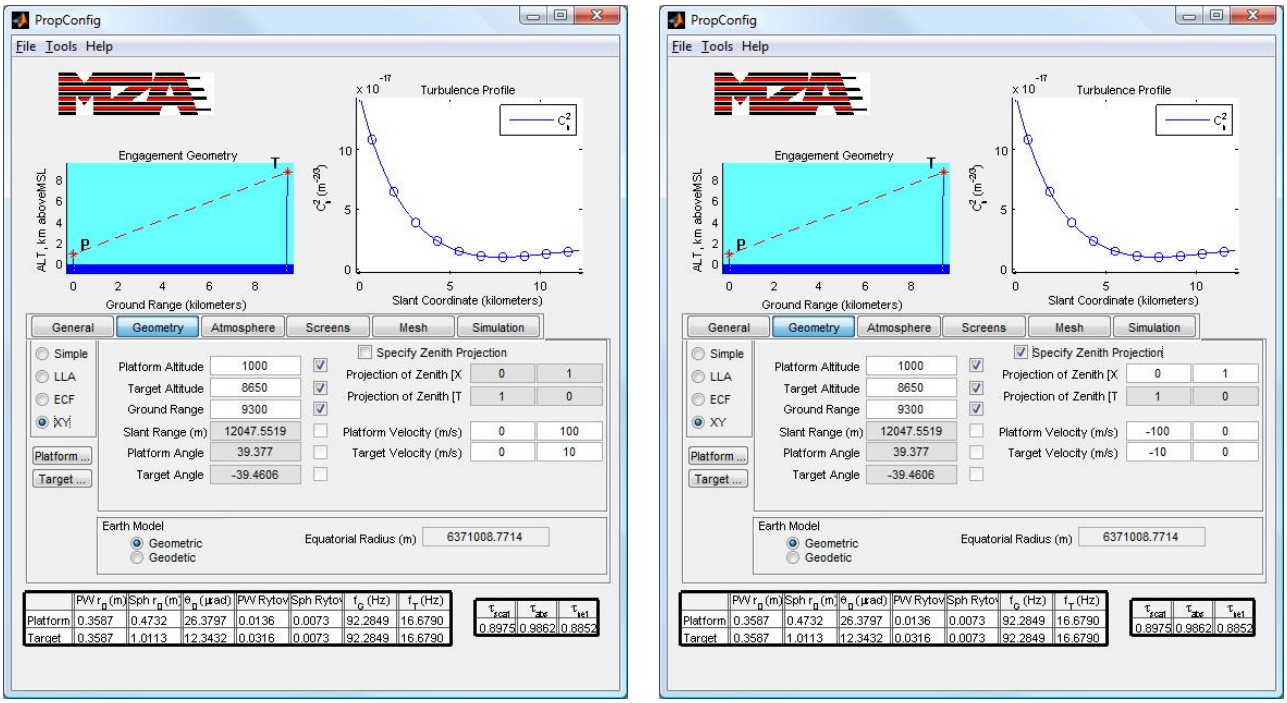


Figure 3.7: PropConfig **window open in the "Simulation" tab after running with the default atmosphere and geometry for a collimated beam. Figure shows the context menu available after running the simulation accessible via right-click on the y-axis label. The mex-system dll used for the simulation is shown on the title of the irradiance plot. Most common parameters for** TBWaveCalc **are available in the simulation tab. Other parameters can be set via** PropControl_Interface **with the "More Params" button.**

One of the options in PROPCONFIG for geometry allows the user to set up geometry using a basic XY coordinate system almost independent of any real-world 3-D geometry considerations. This capability was added for compatibility with geometry specification in TurbTool, the GUI for atmosphere and geometry setup that is part of WaveTrain™. In fact, PROPCONFIG can load a data file that was saved from TurbTool. An XY specification is converted within PROPCONFIG to a 3-D geometry specification for use in EngagementTools and ATMTools based on the "up" direction which is the projection of the zenith at the target onto the plane perpendicular to the propagation direction. Figure 3.8(a) shows the XY geometry panel after setting a platform speed of 100 m/s and a target speed of 10 m/s both heading East, via the Simple geometry specification, and using the default calculation of the zenith projection. The default behavior computes the zenith projection in the target parallel



(a) (b)

Figure 3.8: PROPCONFIG geometry setup using the XY geometry panel with (a) the default zenith projection and (b) a specified zenith projection in the Y axis.

(P) and target transverse (T) coordinate system, which could change depending on the target velocity vector, and where the cross product of the T and P unit vectors is in the direction of propagation. It should be noted that the platform is located South of the target (the default behavior for a "Simple" geometry) so that a heading of East is entirely transverse to the propagation direction. Because the target velocity is all horizontal velocity, i.e. the target has no vertical velocity, the zenith projection is entirely in the T direction. Velocities in the T direction are mapped onto the X axis and velocities in the P direction are mapped onto the Y axes using the default zenith projection. To use a different zenith projection, the user can check the box and set the desired zenith projection. The direction of X and Y is such that the cross product of X and Y is in the direction of propagation from platform to target. Figure 3.8(b) shows the XY geometry panel after setting the zenith projection to be in the Y axis. Since we require that the cross product of X and Y be in the direction of propagation, the platform and target velocities are actually in the -X direction.

3.3.0.26 Using ATMTools with WaveTrain™

ATMTools is a toolbox for atmospheric propagation modeling. Therefore, it follows that it could be used in setting up wave optics propagation simulations. This section will describe in some detail the use of output

files from PropConfig in setting up WaveTrain™ simulations. This section assumes basic familiarity with use of WaveTrain™.

Figure 3.9 shows the Baseline Adaptive Optics and Tracking (BLAT) model from the distributed WaveTrain™ Examples. Without going into too much detail, it is a model that includes atmospheric propagation with higher-order compensation using a wavefront sensor (WFS) and deformable mirror (DM) and tilt compensation using a centroid tracker with a point source at the target end of the propagation. It is the atmospheric propagation that we will be concerned with here, but will show that the PropConfig output files can also be used for setting other parameters of the simulation as well.

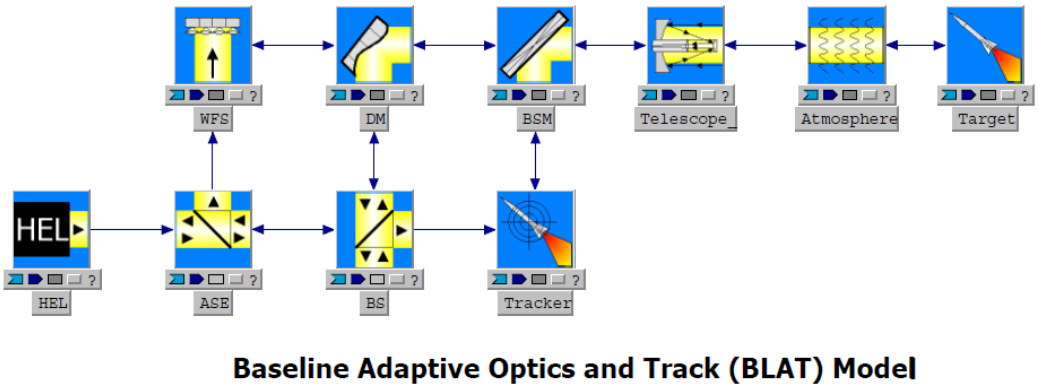


Figure 3.9: WaveTrain™ example model, Baseline Adaptive Optics and Tracking (BLAT), for illustrating use of PropConfig data files.

A run set for BLAT that does not use data files from PropConfig is shown in Figure 3.10. The setup uses the function (or constructor) **AcsAtmSpec**[2] to specify the atmospheric details. Details on the use of **AcsAtmSpec** for setting atmospheric parameters can be found at http://www.mza.com/doc/wavetrain/turbtool/index.html. It might be useful for a new user to read that documentation to become familiar with what parameters go into the atmospheric specification. This particular usage of **AcsAtmSpec** requires a profile number which is limited to values 1-3 representing Clear-1, Clear-2 and H-V 5/7 turbulence profiles. Other more general constructors for **AcsAtmSpec** are available but require the user to pass as arguments C_n^2 values, screen locations and thicknesses, and wind speed values for each phase screen. It could be quite time consuming to compute values and manually enter them into the run set window. For that reason, there are tempus™ functions that allow loading of data from MATLAB® data files.

Figure 3.11 shows a run set that uses a PropConfig data file to selectively load information for setting parameters. Run variable *CSD* is created by loading, using the tempus™ function **mliLoad**, the *computedScreenData* structure from the PropConfig data file and necessary parameters are extracted from the structure using the tempus™ function **mliGetField**. A different constructor for **AcsAtmSpec** is then used along with the extracted phase screen positions and strengths to set up the propagation. The run set also illustrates using the information in *CSD* to set the platform velocity (variable 16) and using the data file to load aperture diameter (variable 20). Similarly, one can load mesh parameters (number of pixels and pixel size) from the PropConfig data file.

The run set shown in Figure 3.12 uses the function **PropConfigSpec** to initialize the atmospheric setup.[3] The inputs to **PropConfigSpec** are as follows:

[2] WaveTrain™ and tempus™ components, functions, constructors and object methods will be shown in **boldface** and parameters, variables and objects will be shown as *variables*.

[3] Note that what is described here is not the version that has been distributed in the most recent version of WaveTrain™ (2010B); the distributed version uses **PropConfigAtmSpec**, a slightly less complicated version of **PropConfigSpec** that only does turbulence modeling. We recommend that the user use **PropConfigSpec**.

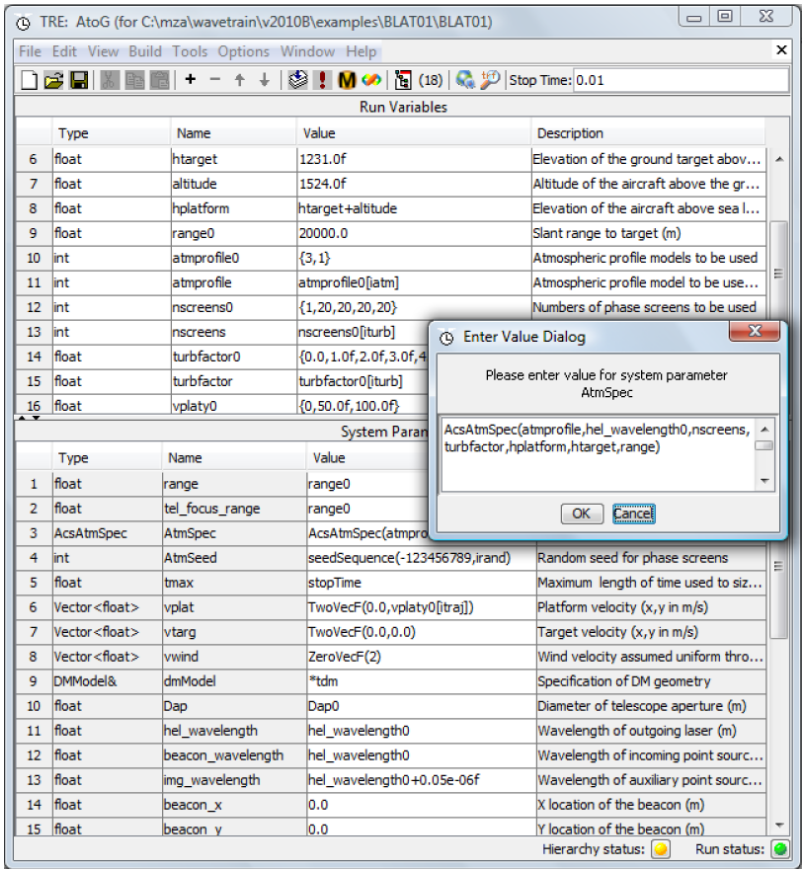


Figure 3.10: Standard run set for BLAT that does not use data files from PropConfig.

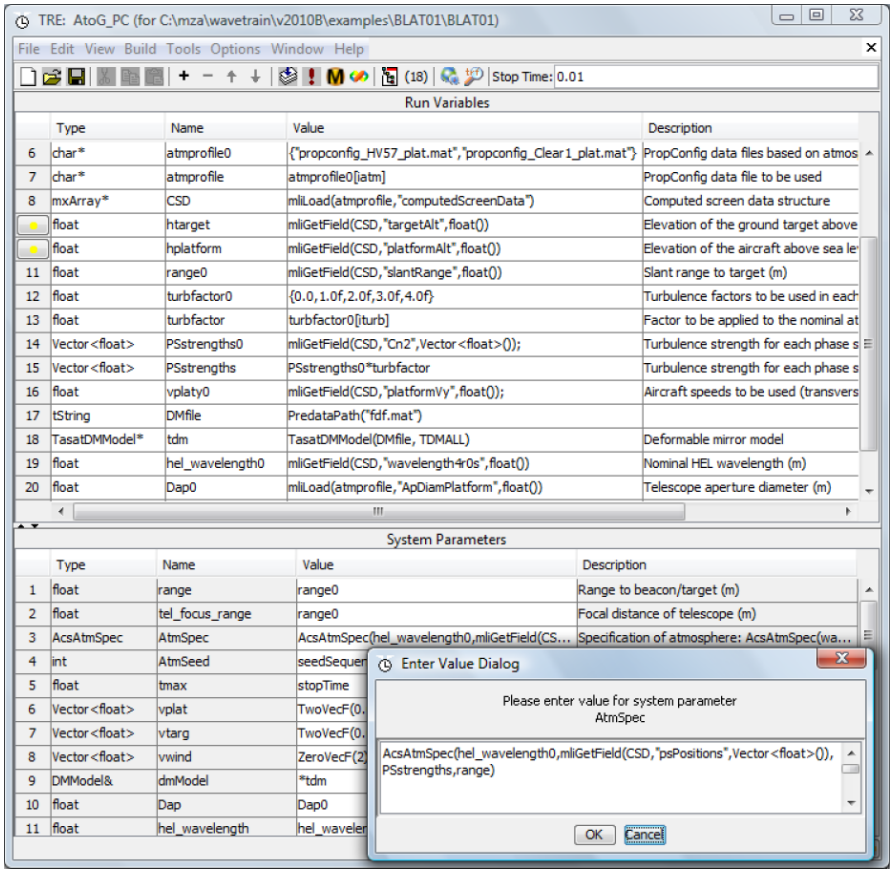


Figure 3.11: Runset for BLAT that loads selected data from a PropConfig data file.

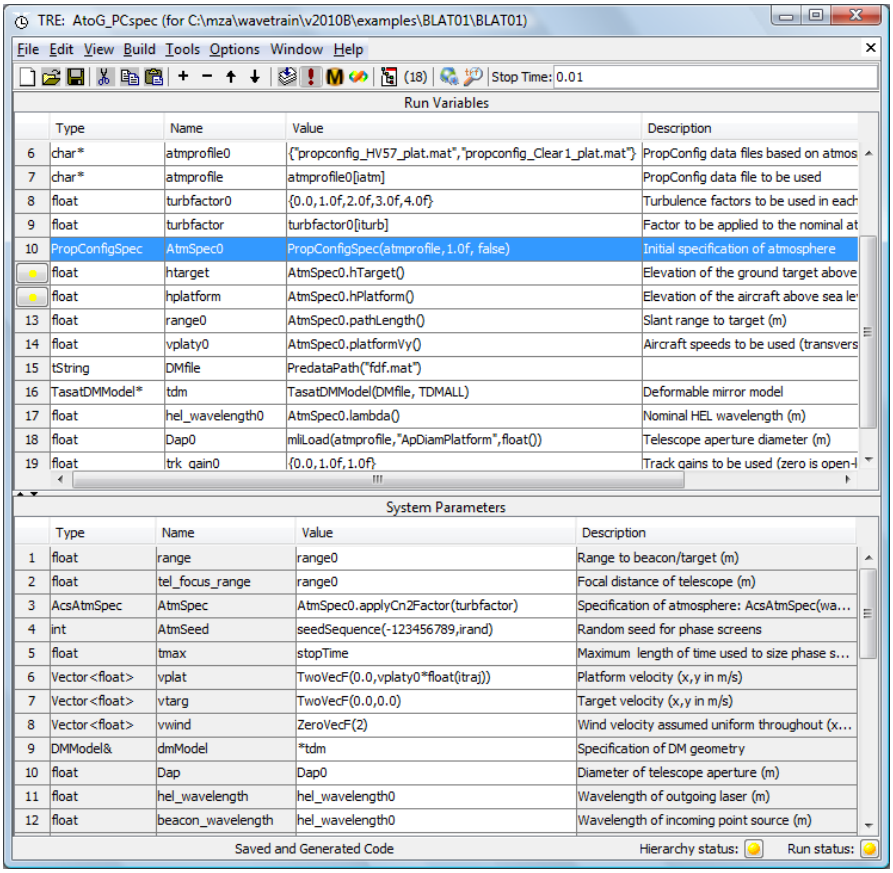


Figure 3.12: Runset for BLAT that uses the function/constructor PropConfigSpec for setting up the atmospheric parameters.

matFileName [char*]	Name of data file saved by PROPCONFIG.
cn2Factor [float]	(Optional) Multiplier on the base turbulence strength as specified in the data file. Defaults to 1.0.
useGeomForSlew [bool]	(Optional) Flag indicating whether to use slew wind in setting velocity of the phase screens. Defaults to false.
applyBlooming [bool]	(Optional) Flag to control whether thermal blooming is to be simulated. Defaults to true. This flag has no effect if the atmospheric component in the system does not include thermal blooming.
dtime [float]	(Optional) Time delta between successive updates of the thermal blooming screens. Does nothing if thermal blooming is not simulated. If a negative value is input, the default value of $4.0 \times$ *dxy* / *maxScreenVelocity* is used, where *maxScreenVelocity* is the velocity of the fastest moving screen due to both wind and slew motion. Defaults to -1.0.

If *useGeomForSlew* is set to false (the default), then **TransverseVelocity** and/or **Slew** components can be set up within the WaveTrain™ system with the following values obtained from an object created using **PropConfigSpec** (all of which are described in more detail below):

xTilt0 → *AtmSpec0*.**slewInitX**()

yTilt0 → *AtmSpec0*.**slewInitY**()

xTiltDot → *AtmSpec0*.**slewRateX**()

yTiltDot → *AtmSpec0*.**slewRateY**()

vx → *AtmSpec0*.**platformVx**() and *AtmSpec0*.**targetVx**()

vy → *AtmSpec0*.**platformVy**() and *AtmSpec0*.**targetVy**()

where *AtmSpec0* is an object created using **PropConfigSpec**. The platform and target velocities are taken directly from the PROPCONFIG data file. The initial slew components (**slewInitX** and **slewInitY**) and slew rates (**slewRateX** and **slewRateY**) are computed as

$$\mathbf{slewInitX} = (v_{p_x} - v_{t_x})/c \tag{3.1}$$

$$\mathbf{slewInitY} = (v_{p_y} - v_{t_y})/c \tag{3.2}$$

$$\mathbf{slewRateX} = (v_{t_x} - v_{p_x})/L \tag{3.3}$$

$$\mathbf{slewRateY} = (v_{t_y} - v_{p_y})/L \tag{3.4}$$

where $v_{p_{x/y}}$ and $v_{t_{x/y}}$ are the components of platform and target velocity respectively, c is the speed of light in vacuum and L is the slant range. **Slew** and/or **TransverseVelocity** components at opposite ends of the propagation path will likely need to have parameters set with a sign difference on what is computed, e.g. if *xTilt0* at the source end of the propagation is set to *AtmSpec0*.**slewInitX**() then *xTilt0* at the receiver end of the propagation may need to be set to -*AtmSpec0*.**slewInitX**(). More information about the use of modeling platform and target motion with **TransverseVelocity** and **Slew** components can be found in the WaveTrain™ User Guide. In the BLAT model, **Slew** and **TransverseVelocity** components for the platform and target are contained in the **Telescope** and **Target** components, respectively. In the run sets shown above, *useGeomForSlew* was set to false and only platform velocity in the y-axis is considered, ignoring any other slew that might have been set in PROPCONFIG.

On the other hand, if *useGeomForSlew* is set to true, then **TransverseVelocity** and **Slew** components should not be used with this atmospheric path; all input parameters for those components, if present, should be set to zero. In this case the slew wind is included with the natural wind in setting the screen velocities. This is the simplest thing to do, but it comes with a defect. It does not properly allow for the modeling of the platform and target motion during the time it takes for the light to travel the length of the path. The resulting effects are typically significant, however, only at long distances or during long simulation times.

There are some additional methods for modifying an existing **PropConfigSpec** object:

AtmSpec → *AtmSpec0*.**applyCn2Factor**(float *cn2Factor*)
Use to scale the multiplier on C_n^2 where the input *cn2Factor* is the multiplier. In the run set shown in Figure 3.12, this method is used to facilitate looping over different values of turbulence multiplier. Caution should be used however, because the value of *cn2Factor* input via this method is applied to the turbulence on top of the multiplier used to create the original **PropConfigSpec** object which is applied on top of the multiplier specified within PROPCONFIG. So if a multiplier of 2 were specified in the PROPCONFIG file, one might specify a multiplier of 0.5 in creating *AtmSpec0* so that applying additional factors via **applyCn2Factor** does not create confusion. A better rule would be to never set a turbulence multiplier when saving a PROPCONFIG data file, i.e. always save the file with a multiplier of one.

AtmSpec → *AtmSpec0*.**applyThermalBlooming**(bool *applyBlooming*)
Use to turn on/off simulation of thermal blooming where the input *applyBlooming* is a boolean value indicating whether thermal blooming should be on or not. This method gives the user a quick way to turn on or off thermal blooming in a simulation. Only applies if the atmospheric component in the WaveTrain™ system includes thermal blooming.[4]

AtmSpec → *AtmSpec0*.**applyDTime**(float *dtime*)
Use to modify the time between updates of the thermal blooming screens where the input *dtime* is the new time in seconds. Only applies if the atmospheric component in the WaveTrain™ system includes thermal blooming.

Also shown in Figure 3.12 and in the description of *useGeomForSlew* above, the variable created, *AtmSpec0*, is used to set various other parameters within the run set, e.g. *AtmSpec0*.**pathLength**() returns the path length (variable 13). Similarly, the mesh parameters are set using *AtmSpec0*.**nxy**() and *AtmSpec0*.**dxy**() (not shown). Some of the methods available for extracting data from an object created with **PropConfigSpec** and their return types are described below.

Methods for PropConfigSpec Objects

lambda() [float]	Wavelength (m).
nxy() [int]	Number of mesh points on a side.
dxy() [float]	Size of mesh points (m).
HELPower() [float]	Laser power stored in the PropConfig file (W).
HELFocus() [float]	Focus range stored in the PropConfig file (m).
hPlatform() [float]	Altitude of the platform (m).
hTarget() [float]	Altitude of the target (m).
pathLength() [float]	Slant range for the propagation (m).
targZenithXY() [Vector<float>]	Two element vector representing the projection of the zenith at the target onto the simulation XY coordinates. Useful for simulations with dispersion to set the variable *upDirection*.

[4] WaveTrain™ components for simulating thermal blooming are of limited distribution, available with specific permission from the government sponsor. Contact wavetrain-license@mza.com to request access.

platformVx() [float]	Platform velocity in the x-direction (m/s)
platformVy() [float]	Platform velocity in the y-direction (m/s)
targetVx() [float]	Target velocity in the x-direction (m/s)
targetVy() [float]	Target velocity in the y-direction (m/s)
slewRateX() [float]	Slew rate in the x-direction (rad/s).
slewRateY() [float]	Slew rate in the y-direction (rad/s).
slewInitX() [float]	Initial slew in x-direction (rad).
slewInitY() [float]	Initial slew in y-direction (rad).
nScreens() [int]	Number of phase screens.
r0(*lambda* **)** [float]	Spherical wave r_0 for the path and the input wavelength (m).
distances() [Vector<float>]	Phase screen locations along the path (m).
heights() [Vector<float>]	Phase screen altitudes along the path (m).
r0s(*lambda* **)** [Vector<float>]	Phase screen r_0s for the input wavelength (m).
rateXs() [Vector<float>]	Phase screen velocity in the x-direction (m/s).
rateYs() [Vector<float>]	Phase screen velocity in the y-direction (m/s).

3.4 Function Descriptions

Detailed technical descriptions of all the toolbox functions are given in this section. The functions are broken up into the 4 categories described in Section 1. Because the atmospheric structure described above is so prevalent in ATMTools, it will be described in detail first.

3.4.1 Atmosphere Structure

The backbone for most of the functions in ATMTools is the function ATMStruct and will be described in more detail in this section.

3.4.1.1 AtmStruct

Syntax

```
Atm = AtmStruct(Geom, Nscreens, ...
     [ModelType1, ModelName1, [P1], ..., [PN]], ...
     [ModelType2, ModelName2, [P1], ..., [PN]], ...
     [ModelType3, ...])

Atm = AtmStruct(Geom, x, dx, ...
     [ModelType1, ModelName1, [P1], ..., [PN]], ...
     [ModelType2, ModelName2, [P1], ..., [PN]], ...
     [ModelType3, ...])
```

This function will generate a MATLAB® structure with atmospheric model data for a given atmospheric path, with modeling assumptions as specified in the calling sequence. The structure may contain turbulence, absorption, scattering, and temperature model data for the laser path altitude corresponding to the input geometry. It is assumed that the specified atmospheric models are available as altitude-dependent functions (with arbitrary parameters) on the MATLAB® path, where altitude is the first input to the function. Data in these structures is appropriate for use in wave-optics simulation or propagation analysis calculations.

The model data (i.e. C_n^2 , absorption, scattering, ...) can be calculated at discrete locations along the path from the platform to the target or the data can be computed in functional form. The discrete form is useful for generating propagation paths for wave-optics simulations. The input *Geom* can be the output geometry structure from GEOMSTRUCT‡ or a comma separated (ordered) list of geometry parameters (*hp*, *ht*, *rd*).

There are two different calling sequences for the arguments preceding the model types, model names, and model parameters:

1. Specify the number of evenly spaced, equal thickness atmospheric slabs. Screens are spaced symmetrically about the midpoint of the path.

Geom [struct/list]	Geometry parameters. Can be a structure from GEOMSTRUCT‡ or a comma separated list (*hp*, *ht*, *rd*).
hp [scalar]	Altitude of platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).
NScreens [scalar]	Number of equal thickness phase screens. Screens are spaced symmetrically about the midpoint of the propagation path. For a continuous model use *NScreens* = *inf*.
MaxAlt [scalar]	(Optional) The maximum altitude over which the discrete atmosphere should be evaluated (m). Defaults to Inf. At altitudes above *MaxAlt*, the model evaluations are set to 0. *NScreens*, *x*, and *dx* are interpreted as being over the path where altitude is less than *MaxAlt*. Must be scalar.
xRange [vector]	(Optional) The normalized ranges over which the atmospheric models are to be computed. *MaxAlt* is a shortcut that will compute *xRange* for the path. The user can also manually specify the ranges. If both *MaxAlt* and *xRange* are specified, *xRange* will be set by taking the max of *xRange(1)* and the min of *xRange(2)*. Input *xRange* must be 1x2.

Example 3.4.1 (Equally spaced, equal thickness atmospheric slabs) *This example shows how to create a atmosphere structure with evenly spaced equal thickness atmospheric slabs.*

```
>> G = GeomStruct('Simple',3000,50,10000);
>> Atm = AtmStruct(G,Inf,'Cn2','HV57')
```

```
Atm =

         hp: 3000
         ht: 50
         rd: 1.0000e+04
          L: 1.0428e+04
    LatLong: [1x1 struct]
     MaxAlt: Inf
     xRange: [0 1]
    Cn2Eval: {'HV57'  'h'}
```

Produces a structure where the index structure constant, C_n^2 , is a continuous function of altitude, based on the Hufnagel-Valley (5/7) profile.

```
>> Atm = AtmStruct(121920,15000,170000,10,'Cn2','HV57',...
        'Abs','AFRLDE_Atmos','A','1064')

Atm =

         hp: 121920
         ht: 15000
         rd: 1.7000e+05
          L: 2.0237e+05
    LatLong: [1x1 struct]
     MaxAlt: Inf
     xRange: [0 1]
          z: [10x1 double]
         dz: [10x1 double]
          h: [10x1 double]
        Cn2: [10x1 double]
    Cn2Eval: {'HV57'  'h'}
        Abs: [10x1 double]
    AbsEval: {'AFRLDE_Atmos'  'h'  'A'  '1064'}
```

Produces a structure where C_n^2 and the absorption are evaluated at 10 discrete screen locations evenly spaced along the propagation path. The C_n^2 profile is based on the Hufnagel-Valley (5/7) profile and the absorption is based on the AFRL/DE Atmospheric model at a wavelength of 1064 nm.

```
>> Atm = AtmStruct(121920,15000,170000,10,'MaxAlt',50000,'Cn2','HV57',...
        'Abs','AFRLDE_Atmos','A','1064')

Atm =

         hp: 121920
         ht: 15000
         rd: 1.7000e+005
          L: 2.0237e+005
    LatLong: [1x1 struct]
     MaxAlt: 50000
     xRange: [0.6679 1]
          z: [10x1 double]
         dz: [10x1 double]
          h: [10x1 double]
        Cn2: [10x1 double]
    Cn2Eval: {'HV57'  'h'}
        Abs: [10x1 double]
```

```
AbsEval: {'AFRLDE_Atmos'  'h'  'A'  '1064'}
```

Same as the previous example except with a maximum altitude for atmosphere set at 50 km. Note by looking at ***Atm.xRange*** *that the turbulence is only modeled in the last 1/3 of the path.*

2. Specify the normalized locations and thicknesses of each atmospheric slab.

Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct‡ or a comma separated list (***hp***, ***ht***, ***rd***).
hp [scalar]	Altitude of platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).
x [vector]	Path-normalized screen location in range $0 \leq x < 1$. Passing $x = 1$ or $x = [1]$ will result in a single phase screen placed at the midpoint of the propagation path. For single screen at the target, use $x = [0.9999]$.
dx [vector]	Path-normalized screen thickness. Must be the same size as ***x*** if passed.
MaxAlt [scalar]	(Optional) The maximum altitude over which the discrete atmosphere should be evaluated (m). Defaults to Inf. At altitudes above ***MaxAlt***, the model evaluations are set to 0. ***NScreens***, ***x***, and ***dx*** are interpreted as being over the path where altitude is less than ***MaxAlt***. Must be scalar.
xRange [vector]	(Optional) The normalized ranges over which the atmospheric models are to be computed. ***MaxAlt*** is a shortcut that will compute ***xRange*** for the path. The user can also manually specify the ranges. If both ***MaxAlt*** and ***xRange*** are specified, ***xRange*** will be set by taking the max of ***xRange(1)*** and the min of ***xRange(2)***. Input ***xRange*** must be 1x2.

Example 3.4.2 (User specified slab locations and thicknesses.) *This example shows how to create an atmospheric structure with user defined screen locations and thicknesses.*

```
>> Atm = AtmStruct(121920,15000,170000,.999999,1,'Cn2','HV57', ...
                   'Abs','AFRLDE_Atmos','A','1064')

Atm =

         hp: 121920
         ht: 15000
         rd: 170000
```

```
          L: 2.0237e+005
    LatLong: [1x1 struct]
     MaxAlt: Inf
     xRange: [0 1]
          z: 2.0237e+005
         dz: 2.0237e+005
          h: 1.5000e+004
        Cn2: 6.3506e-018
    Cn2Eval: {'HV57'  'h'}
        Abs: 7.4134e-08
    AbsEval: {'AFRLDE_Atmos'  'h'  'A'  '1064'}
```

Places a single screen at the target.

```
>> Atm = AtmStruct(121920,15000,170000,[.25 .75],[.5 .5],'Cn2','HV57', ...
                   'Abs','AFRLDE_Atmos','A','1064')
Atm =
          hp: 121920
          ht: 15000
          rd: 170000
           L: 2.0237e+005
    LatLong: [1x1 struct]
     MaxAlt: Inf
     xRange: [0 1]
          z: [2x1 double]
         dz: [2x1 double]
          h: [2x1 double]
        Cn2: [2x1 double]
    Cn2Eval: {'HV57'  'h'}
        Abs: [2x1 double]
    AbsEval: {'AFRLDE_Atmos'  'h'  'A'  '1064'}
```

Places two equal thickness screen (0.5 × path-length) at 0.25 and 0.75 × path-length.

The remaining inputs are all optional, and are used to set values of model data in the output structure. There are thirteen model types currently supported: `'Cn2'` (turbulence strength), `'Abs'` (absorption coefficient), `'Scat'` (scattering coefficient), `'Temp'` (temperature), `'Wind'` (natural wind), `'WindHeading'` (natural wind heading), `'Ext'` (extinction), `'Press'` (atmospheric pressure), `'Dens'` (atmospheric density), `'Lin'` (inner scale for turbulence), `'Lout'` (outer scale for turbulence), `'RH'` (relative humidity), and `'DewPt'` (dew point). Any, all, or none of the models may be set, depending upon calling sequence. If a particular type of model is specified twice, the last specification will be used in the output structure. If an error is encountered in evaluating the model function, the output fields will not be set for the specified model type. See Section 3.4.3 for a description of the atmospheric models available in the ATMTools toolbox or type `help AtmModels` at the MATLAB® command line. To input measured data for any model type, use 'NoEval' for the *`ModelName`* and the data values, in vector form, for the input parameter. User-defined functions may also be used if the function is written such that the first argument is altitude above the Earth's surface (*`h`*) and subsequent arguments are not of type `Structure`. Model data is generated by calls to existing model functions of the form:

```
ModelName(h,p1,p2,...,pN)
```

with

ModelType [string]	Identifier for type of atmospheric model. Supported model types are 'Cn2', 'Abs', 'Scat', 'Temp', 'Wind', 'WindHeading', 'Ext', 'Press', 'Dens', 'Lin', 'Lout', 'RH', and 'DewPt', affecting the corresponding field in the Atm structure.
ModelName [string]	Function to be evaluated in setting the model values. Must correspond to a function on the MATLAB® path with calling sequence `ModelName(h,p1,p2,...,pN)`.
p1...pN [list]	Parameter values for the function *ModelName*, as required by the specified function.

The output of AtmStruct is a structure (called the atmospheric structure, *Atm*) which contains geometry information and the atmospheric model data.

Atm [struct]	Structure containing turbulence screen parameters.
Atm.hp [scalar]	Altitude of transmit/receive platform (m).
Atm.ht [scalar]	Altitude of target (m).
Atm.rd [scalar]	Downrange to target along curved earth surface (m).
Atm.L [scalar]	Slant range from platform to target (m).
Atm.LatLong [struct]	Structure of geometry information used to create/recreate an *Atm* structure.
Atm.MaxAlt [scalar]	Max Alt over which the models are evaluated (m).
Atm.xRange [vector]	Normalized slant ranges at the beginning and end of the atmosphere. Can be set by *MaxAlt* or by specifying *xRange*.
Atm.z [vector]	Phase screen distances from transmitter (m).
Atm.dz [vector]	Phase screen thicknesses (m).
Atm.h [vector]	Altitude of each phase screen above earth surface (m).
Atm.Cn2 [vector]	(Optional) *Cn2* values for the phase screens ($m^{-2/3}$).
Atm.Cn2Eval [cell]	(Optional) Turbulence model evaluation.
Atm.Abs [vector]	(Optional) Absorption coefficients for the phase screens (1/m).
Atm.AbsEval [cell]	(Optional) Absorption model evaluation.
Atm.Scat [vector]	(Optional) Scattering coefficients for the phase screens (1/m).
Atm.ScatEval [cell]	(Optional) Scattering model evaluation.

Atm.Temp [vector]	(Optional) Temperature values for the phase screens (K).
Atm.TempEval [cell]	(Optional) Temperature model evaluation.
Atm.Wind [vector]	(Optional) Natural wind speed values for the phase screens (m/s).
Atm.WindEval [cell]	(Optional) Natural wind model evaluation.
Atm.WindHeading [vector]	(Optional) Natural wind heading for the phase screens (deg from North).
Atm.WindHeadingEval [cell]	(Optional) Natural wind heading model evaluation.
Atm.VerticalWind [vector]	(Optional) Natural vertical wind.
Atm.VerticalWindEval [cell]	(Optional) Natural vertical wind model evaluation.
Atm.Ext [vector]	(Optional) Extinction coeff. values for the phase screens (1/m).
Atm.ExtEval [cell]	(Optional) Extinction model evaluation.
Atm.Press [vector]	(Optional) Atmospheric pressure values for the phase screens (Pa).
Atm.PressEval [cell]	(Optional) Pressure model evaluation.
Atm.Dens [vector]	(Optional) Atmospheric density values for the phase screens (kg/m^3).
Atm.DensEval [cell]	(Optional) Density model evaluation.
Atm.Lin [vector]	(Optional) Turbulence inner scale values for the phase screens (m).
Atm.LinEval [cell]	(Optional) Inner scale model evaluation.
Atm.Lout [vector]	(Optional) Turbulence outer scale for the phase screens (m).
Atm.LoutEval [cell]	(Optional) Outer scale model evaluation.
Atm.RH [vector]	(Optional) Relative humidity for the phase screens (%).
Atm.RHEval [cell]	(Optional) Relative humidity model evaluation.
Atm.DewPt [vector]	(Optional) Dew point for the phase screens (K).
Atm.DewPtEval [cell]	(Optional) Dew point model evaluation.

3.4.2 Atmospheric Characterization

The functions in this section are useful for computing atmospheric propagation parameters.

3.4.2.1 AnisoplanaticJitter

Syntax

[sigmaP, sigmaT] = AnisoplanaticJitter(*alpha, D, d, Atm*)

```
SCALING_CODE_API int AnisoplanaticJitter(char** errorChain,
    const double* turbMult, const int nTurbMult,
    const double* apertureDiam, const int nD, const double* pathSep,
    const int nScreens, const int nSeps, const int nGeoms,
    const double* cn2, const double* z, const double* dz,
    const double* slantRange, double* sigmaP, double* sigmaT)
```

This function computes anisoplanatic beam jitter components given the following Matlab inputs:

alpha [vector]	Turbulence strength multiplier.
D [scalar]	Tx/Rx Aperture diameter (m).
d [matrix]	Displacement of beam paths along propagation path (m). Resolved for the P axis and T axis, as calculated from PathDisp, where column 1 is the displacement in the P axis and column 2 is the displacement in the T axis, i.e., *d*=[*d*_P, *d*_T].
Atm [struct]	Atmospheric modeling structure from AtmStruct.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages - deepest error is first.
turbMult [in]	Turbulence multiplier; must be length *nTurbMult*.
nTurbMult [in]	Number of turbulence multipliers; must be 1 or *nEvals* where *nEvals* = max(*nTurbMult*, *nD*, *nGeoms*).
apertureDiam [in]	Aperture diameter (m); must be length *nD*.
nD [in]	Number of apertures; must be 1 or *nEvals* where *nEvals* = max(*nD*, *nTurbMult*, *nSeps*, *nGeoms*).
pathSep [in]	Vector separation at each point *z* along the path (m); must be 2 * *nScreens* * *nSeps*.
nScreens [in]	Number of phase screens specified for each input.
nSeps [in]	Number of separations to evaluate; must be 1 or *nEvals*.
nGeoms [in]	Number of geometries evaluated; must be 1 or *nEvals*.

cn2 [in]	Cn2 values for the phase screens; must be length *nScreens* * *nGeoms*.
z [in]	Phase screen distances from transmitter (m); must be length *nScreens* * *nGeoms*.
dz [in]	Screen thickness for each phase screen; must be length *nScreens* * *nGeoms*.
slantRange [in]	Slant range (m); must be length *nGeoms*.
sigmaP [out]	Anisoplanatic jitter in P axis (rad); Allocate to 1 * *nEvals*.
sigmaT [out]	Anisoplanatic jitter in T axis (rad); Allocate to 1 * *nEvals*.

The output is the jitter resolved in the target P and T axes, *sigmaP* and *sigmaT*, in radians. Beam displacement vector should be obtained via PATHDISP to include appropriate anisoplanatic effects (angular, aperture, chromatic, etc.)

Example 3.4.3 (AnisoplanaticJitter) *This example illustrates a call to* ANISOPLANATICJITTER *using a fixed angular separation between points along a path.*

```
>> G = GeomStruct;
>> Atm = AtmStruct;
>> d = PathDisp(G,Atm,'ANGULAR',1e-6,0);
>> [sigmaP,sigmaT] = AnisoplanaticJitter(1.0,1.0,d,Atm)
```

Anisoplanatic jitter components for turbulence multiplier of 1, 1 m Tx/Rx aperture, beam displacement *d*, *and atmospheric model in* *Atm*.

3.4.2.2 ANISOPLANATICSTREHL

Syntax

Strehl = ANISOPLANATICSTREHL(*alpha, lambda, D, d, Atm*)

```
SCALING_CODE_API int AnisoplanaticStrehl(char** errorChain,
    const double turbMult, const double lambda, const double apDiam,
    const double slantRange, const int nScreens, const double* x,
    const double* pathSep, const double* cn2, double* strehl)
```

This function calculates the Strehl ratio due to the effect of angular anisoplanatism.[13] The Matlab inputs are:

alpha [scalar]	Turbulence strength multiplier.
lambda [scalar]	Wavelength of laser (m).
D [scalar]	Aperture diameter (m).

s [Matrix]	Separation along the path as a function of z (as calculated by PathDisp) (m).
Atm [struct/list]	Atmospheric modeling parameters. Can be a discrete structure from AtmStruct or a comma separated list of (... *x*, *Cn2*, *L*).
x [vector]	Path-normalized screen locations along the propagation path.
Cn2 [vector]	Values of the refractive index structure coefficient along the path ($m^{-2/3}$).
L [scalar]	Slant range (m).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
turbMult [in]	Turbulence strength multiplier.
lambda [in]	Wavelength of laser (m).
apDiam [in]	Aperture diameter (m).
slantRange [in]	Slant range (m).
nScreens [in]	Number of data points along path.
x [in]	Path-normalized screen locations along the propagation path; must be length *nScreens*.
pathSep [in]	Separation along the path as a function of *z* (as calculated by *PathDisp*) (m); stored [p1, t1, p2, t2, ..., pN, tN] where N is the number of screens; size must be allocated to 2 * *nScreens*.
cn2 [in]	Values of the refractive index structure coefficient along the path ($m^{-2/3}$); must be length *nScreens*.
strehl [out]	Angular anisoplanatic Strehl; must be length 1.

The output is the Strehl ratio due to anisoplanatism.

Example 3.4.4 (AnisoplanaticStrehl) *This example illustrates a call to* AnisoplanaticStrehl *using a fixed angular displacement between points along a path.*

```
>> G = GeomStruct;
>> Atm = AtmStruct;
>> d = PathDisp(G,Atm,'ANGULAR',1e-6,0);
>> Strehl = AnisoplanaticStrehl(1,1.315e-6,1.5,d,Atm)
```

Anisoplanatic Strehl for turbulence multiplier of 1, 1 m Tx/Rx aperture, beam displacement *d*, *and atmospheric model in* *Atm*.

3.4.2.3 AtmosphericOTF

Syntax

[OTF, PSF, HO, HA] = AtmosphericOTF(*f, Type, Model, r0, di, D, ... lambda, alpha, Geom, Atm*)

```
SCALING_CODE_API int AtmosphericOTF(char** errorChain, const int nFx,
    const int nFy, double* f, const int typeFlag, char model,
    const double r0, const double di, const double apertureDiam,
    const double lambda, double* opticalTransFunc,
    double* pointSpreadFunc, double* otfOptics, double* otfAtmos)
```

This function computes the optical transfer function (*OTF*) and point spread function (*PSF*) for ensemble-averaged exposure through Kolmogorov turbulence [14], [15]. The Matlab inputs are:

f [matrix]	Spatial frequencies in image (cycles/m).
Type [scalar]	Switch [1/0] to calculate long/short exposure *OTF*. 1 - long exposure, 0 - short exposure.
Model [string]	Modeling method for *OTF*/*PSF*. Supported models are: 'FRIED' - Fried's theoretical propagation (default), 'GAUSSIAN' - Gaussian approximation to theory.
r0 [scalar]	(Optional) Spherical wave coherence diameter (m).
di [scalar]	(Optional) Image distance from system pupil (m).
D [scalar]	(Optional) Aperture diameter (m).
lambda [scalar]	(Optional) Mean wavelength of light in image (m).
alpha [scalar]	(Optional) Turbulence multiplier.
Geom [struct]	(Optional) Geometry structure from GeomStruct specifying platform height, target height, downrange. If *Geom* and *Atm* are input, *r0* is calculated, and *di* is assumed to be slant range given by the geometry.
Atm [struct]	(Optional) Atmospheric structure from AtmStruct. *alpha*, *Geom*, and *Atm* are specified in order to calculate an appropriate *r0* value.

If not specified, the following default values are used in modeling: *lambda* = 1 micron, *D* = 1 m, *r0* = *D*/4, *di* = 1 m, *alpha* = 1.

OTF [matrix]	Optical transfer function for imaging system in spatial frequency domain.
PSF [matrix]	Point spread function for imaging system in spatial domain.
HO [matrix]	*OTF* component due to optics only.
HA [matrix]	*OTF* component due to atmosphere only.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nFx [in]	Number of elements in the sequential dimension of *f*.
nFy [in]	Number of elements in the non-sequential dimension of *f*.
f [in]	Spatial frequencies in image (cycles/m or cycles/rad); must be *nFx* x *nFy*.
typeFlag [in]	Flag [1/0] to calculate long/short exposure OTF. 1 - long exposure, 0 - short exposure.
model [in]	Modeling method for OTF/PSF. Supported models are 'F' for Fried (default) or 'G' for Gaussian.
r0 [in]	Spherical wave coherence diameter (m).
di [in]	Image distance from system pupil (m). Set to 1 if f is in cycles/rad.
apertureDiam [in]	Aperture diameter (m); must be length 1.
lambda [in]	Mean wavelength of light in image (m); must be length 1.
opticalTransFunc [out]	Optical transfer function for imaging system in spatial frequency domain; must be allocated to length *nFx* x *nFy*.
pointSpreadFunc [out]	Point spread function for imaging system in spatial domain. If not NULL, must be allocated to length *nFx* x *nFy*.
otfOptics [out]	OTF component due to optics only; must be allocated to length *nFx* x *nFy*.
otfAtmos [out]	OTF component due to atmosphere only; must allocate to length *nFx* x *nFy*.

Example 3.4.5 (AtmosphericOTF) *Different calling sequences for computing OTF using* AtmosphericOTF.

```
>> f = linspace(0,1,100);
>> [OTF,PSF] = AtmosphericOTF(f/550e-9,1,'FRIED',0.18,1,1,550e-9)
```

Use specified r_0, no default values used

```
>> [OTF,PSF] = AtmosphericOTF(f/550e-9,0,'FRIED',1,1,550e-9,AtmStruct)
```

Calculate r_0 using default atmospheric structure from AtmStruct *and default* ***alpha*** *and uses specified* ***di***, ***lambda*** *and* ***D*** *values.*

```
>> Atm = AtmStruct;
>> [OTF,PSF] = AtmosphericOTF(f/550e-9/Atm.L,1,'GAUSSIAN',1,GeomStruct,Atm)
```

Calculate r_0 using default geometry from GeomStruct *and default atmosphere from* AtmStruct *and default* ***alpha*** *value. Calculate* ***OTF***, ***PSF*** *using default* ***lambda*** *value with* ***di*** *equal to the slant range specified by* ***G***.

3.4.2.4 BeaconD0

Syntax

d0 = BeaconD0(***alpha, lambda, Geom, Lb, Atm***)

This function computes the section size of a laser guide star (LGS) beacon (***d0***) [16] with the following inputs:

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list of (..., ***hp***, ***ht***, ***rd***) - not required if ***Atm*** is a structure from AtmStruct.
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).
Lb [scalar]	Slant range of beacon (m).
Atm [struct/string]	Atmospheric modeling parameters. Can be a structure from AtmStruct or a turbulence profile model to be used.

The output ***d0*** is the LGS beacon section size and is appropriate for computing residual phase variance for compensation from laser guide star beacon as $(D/d0)^{5/3}$. Assumes the target object is at a finite range as specified by ***Geom***. Hence propagation calculations are made using spherical wave analysis. Returns ***NaN*** if propagation path intersects earth surface.

Example 3.4.6 () *General description*

```
>> G = GeomStruct;
>> Atm = AtmStruct;
>> d0 = BeaconD0(1,550e-9,G,Atm.L/2,Atm)
```

Compute `d0` *using the engagement geometry in* `G` *at the half of the range assumed in* `Atm`. `G` *and* `Atm` *must have same values for* `hp`, `ht`, *and* `rd`.

```
>> d0 = BeaconD0(1,550e-9,G,Atm.L,'HV57')
```

Compute `d0` *using the geometry in* `G`, *the range in* `Atm` *and* `HV57` *for the turbulence model.*

```
>> d0 = BeaconD0(1,550e-9,0,12000,0,24000,'HV57')
```

Compute `d0` *for the specified geometry and range with* `HV57` *for the turbulence model.*

3.4.2.5 BeamIrradianceCovariance

Syntax

B = BeamIrradianceCovariance(*alpha, d, Atm, wvl, W0, F0*)

This function computes the irradiance covariance for a beam wave given turbulence multiplier `alpha`, point spacings `d`, atmospheric modeling data in `Atm`, and wavelength in `wvl`.[7] In this calculation, the light propagates from the target to the platform. Uses the weak turbulence assumption. Use with caution - function is numerically unstable.

`alpha` [scalar]	Turbulence strength multiplier.
`d` [vector]	Separation of the points in the aperture plane (m).
`Atm` [struct]	Atmospheric modeling structure from AtmStruct.
`wvl` [scalar]	Optical wavelength (m).
`W0` [scalar]	1/e radius of the source field (m).
`F0` [scalar]	Radius of curvature of the source field (m).

The function returns `B`, the irradiance covariance value.

Example 3.4.7 (BeamIrradianceCovariance) *The following shows a basic computation of beam irradiance covariance.*

```
>> Atm  = AtmStruct(0, 2e4, 0, 100, 'Cn2', 'HV57');
>> d = linspace(0, sqrt(Atm.L/(2*pi/wvl))*4, 10);
>> B = BeamIrradianceCovariance(1, d, Atm, 0.5e-6, 0.5, Inf);
```

Irradiance covariance for turbulence multiplier of 1, wavelength 500 nm, focused Gaussian source with beam radius 50 cm at 20 km propagating straight down, pupil-plane separations d, and H-V 5/7 turbulence model.

3.4.2.6 BeamSpread

Syntax

[RowSigma, ColSigma] = BeamSpread(*Img, RelThresh, N, DisplayPlot*)

```
SCALING_CODE_API int BeamSpread(char** errorChain,
    const double* image, const int n0, const int n1, double relThresh,
    const int nSamples, double* sigma0, double* sigma1)
```

The function BeamSpread will compute the beam spread in two axes of an input irradiance profile using a least squares fit of the irradiance to a Gaussian of the form

$$I(x) = e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2}$$

The Matlab inputs to the function are:

Img [matrix]	Average beam irradiance (arbitrary units) or a grid structure with fields .x, .y and .g.
RelThresh [scalar]	Attenuation relative to the peak for Gaussian fit.
N [scalar]	(Optional) Number of points for irradiance interpolation.
DisplayPlot [scalar]	(Optional) Option (1/0) to display plot of row and column fit. Default is 0.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
image [in]	Average beam irradiance (arbitrary units); must be size *n0* * *n1*.
n0 [in]	Number of elements in along leading dimension of image (i.e. the dimension consecutive in memory).
n1 [in]	Number of elements along second dimension of image.
relThresh [in]	Attenuation relative to the peak for Gaussian fit.
nSamples [in]	Number of points for irradiance interpolation; set to zero to just use the values at the pixels.
sigma0 [out]	Fit of beam irradiance in first image axis (pixels); allocate one element.
sigma1 [out]	Fit of beam irradiance in second image axis (pixels); allocate one element.

The output of the function is the beam spread term σ for each axis. If *Img* is input as a structure, the output beam spreads will be in the same units as *Img.x* and *Img.y*. Otherwise the beam spreads are output in units of pixels. The beam spread terms for an $M \times N$ image is computed as follows:

1. The irradiance is averaged along the rows (columns) resulting in a vector of length $M(N)$.
2. This vector is then normalized by its maximum value - keeping only the values above the user defined relative threshold. The vector length is now less than or equal to $M(N)$ depending on the relative threshold.
3. The vector is then interpolated to the user-defined number of points, if *N* is specified and is greater than zero, and fit to a Gaussian [see Eq. **??**] centered on the centroid of the vector.

3.4.2.7 BoundaryAlt

Syntax

hh = BoundaryAlt(*h, hInt, hhInt*)

```
SCALING_CODE_API int BoundaryAlt(char** errorChain, const double* h,
     const int nScreens, const int nGeoms, const double* altitudeOld,
     const double* altitudeNew, const int nOldNew, const int nLayers,
     double* hnew)
```

This function maps input altitudes in the intervals given in *hInt* into altitudes in the interval specified in *hhInt*. Intended to facilitate using atmospheric models in the earth's boundary layer where certain altitudes should be referenced to ground level, and other altitudes should be referenced to sea level. Generally, *hInt*(end) = *hhInt*(end) = boundary-layer altitude limit. However, code will accept other inputs and use them accordingly. If G represents the values specified in *hInt* and B represents the values specified in *hhInt*, the output altitudes h' are set using

$$h' = \begin{cases} B_1, & h < G_1 \\ B_i + \frac{B_{i+1}-B_i}{G_{i+1}-G_i}(h - G_i), & G_i \le h < G_{i+1} \\ h, & h \ge G_N \end{cases}$$

where $0 < i \le N$ and N is the number of layers specified. The Matlab inputs are:

h [vector]	Altitude above sea level (m).
hInt [array]	Altitude interval for mapping (m). Can be of size 1 x nLayers+1 or nScreens x nLayers+1.
hhInt [array]	Altitude interval for output (m). Can be of size 1 x nLayers+1 or nScreens x nLayers+1.

The output *hh* is a vector of mapped altitudes in meters.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.

h [in]	Altitude above sea level (m). Must be of size *nScreens* x *nGeoms*.
nScreens [in]	Number of screens. Must be the same as the number of rows in *h*.
nGeoms [in]	Number of geometries. Must be the same as the number of columns in *h*.
altitudeOld [in]	Altitude interval for output (m). Can be of size 1 x *nLayers*+1 or *nScreens* x *nLayers*+1 and the same size as *altitudeNew*.
altitudeNew [in]	Altitude interval for mapping (m). Must be of size 1 x *nLayers*+1 or *nScreens* x *nLayers*+1 and the same size as *altitudeOld*.
nOldNew [in]	Measure of rows of *altitudeNew* and *altitudeOld*. Must be 1 or *nScreens*.
nLayers [in]	Number of layers.
hnew [out]	Mapped altitudes (m). Must be the same size as h and initialized as such.

Example 3.4.8 (BoundaryAlt) *The following examples illustrate use of* BoundaryAlt *to scale altitudes.*

```
>> hh = BoundaryAlt([0 2500 6000],[0 5000],[1230 5000])
```

Map altitudes from 0 to 5000 m to a range of 1230 to 5000 m. This could be used to map altitudes above ground to altitudes above MSL, keeping boundary layer altitude fixed. Note that ***hh****(1) = 1230.*

```
>> hh = BoundaryAlt([1230 2500 6000],[1230 5000],[0 5000])
```

Map altitudes from 1230 to 5000 m to a range of 0 to 5000 m. This could be used to map altitudes above MSL to altitudes above ground, keeping boundary layer altitude fixed. Note that ***hh****(1) = 0.*

```
>> GndAlt = rand(10,1)*500+500;
>> h = EarthAlt((0.05:0.1:0.95)',1000,5000,10000);
>> hh = BoundaryAlt(h,[GndAlt repmat(5000,10,1)],[0 5000]);
>> figure; plot((0.05:0.1:0.95),[hh,h,GndAlt])
```

Scales altitudes between ground level and 5000 m to be between 0 and 5000 m. For this example, ground altitude, ***GndAlt****, is generated randomly between 500 and 1000 m. One could also use* TerrainProfiler *with DTED data to obtain ground altitude along a propagation path (see Section 2.3.2.30).*

3.4.2.8 CircFresnel

Syntax

[alphaL, c] = CircFresnel(*R0, Dp, R, Dt, lambda, Ngrid, [OutForm]*)

```
SCALING_CODE_API int CircFresnel(char** errorChain,
    const double range0, const double apertureDiam,
    const double observationRange, const double observationDiam,
    const double lambda, const int nGrid, const char outForm,
    double* alphaLr, double* alphaLi, double* c)
```

The function CircFresnel computes the complex field resulting from Fresnel diffraction of a circular aperture illuminated by an on-axis point source *R0* meters from the aperture at an observation plane *R* meters from the aperture. The Matlab inputs are:

R0 [scalar]	Phase curvature of illuminating field (m).
Dp [scalar]	Aperture diameter (m).
R [scalar]	Range from aperture to observation plane (m).
Dt [scalar]	Diameter of interest in the observation plane (m).
lambda [scalar]	Wavelength (m).
Ngrid [scalar]	Number of grid points in the observation plane.
OutForm [string]	(Optional) Output format. 'S' for slice, 'Q' for quadrant, 'F' for full image, and 'T' for full image in a structure format (.x, .y, .g). If quadrant is specified, the grid dimension is *Ngrid* x *Ngrid*, for full pattern, the grid is 2(*Ngrid*-1)x2(*Ngrid*-1). If *Ngrid* is $2^n + 1$ and the full pattern is returned, it will be properly centered for use with FFT2.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
range0 [in]	Phase curvature of illuminating field (m); must be length 1.
apertureDiam [in]	Aperture diameter (m); must be length 1.
observationRange [in]	Range from aperture to observation plane (m).
observationDiam [in]	Diameter of interest in the observation plane (m).
lambda [in]	Wavelength (m).
nGrid [in]	Number of grid points in the observation plane.

outForm [in]	Output format. 'S' for slice, 'Q' for quadrant, 'F' for full image. If quadrant is specified, the grid dimension is ***nGrid*** x ***nGrid***, for full pattern, the grid is 2(nGrid-1)x2(nGrid-1).
alphaLr [out]	Normalized real part of complex field (sqrt(W)); Allocate to N * N where N is ***nGrid*** or 2*(***nGrid*** - 1).
alphaLi [out]	Normalized imaginary part of the complex field; Aloocate to same length N * N.
`c` [out]	Evaluation points along one axis (m); Allocate to length N.

The field is given by [17]

$$\alpha_F = -iu \int_0^1 \rho J_0(v\rho) e^{\frac{i}{2}u\rho^2} \, \mathrm{d}\rho$$

where the normalized parameters, ρ, u, and v are given by

$$\rho = \frac{q}{a}, \; u = \frac{ka^2(r_0 + r)}{r_0 r}, \; v = \frac{kac}{r}$$

where q is the radial coordinate in the aperture, a is the radius of the aperture, k is the wave number ($2\pi/\lambda$), r_0 is the distance from the point source to the aperture center, and r is the distance from the aperture center to the point of observation. A solution using Lommel's equations [see Section 3.4.6.4 for a definition of the Lommel functions] is given by

$$\alpha_L = \frac{u}{2} M(u, v) - i \frac{u}{2} L(u, v)$$

However, for very large values of u the patterns are highly structured and Schwarzchild's Approximation must be used. The field is then given by

$$\alpha_{Schw}(u, v) = \frac{1}{2} e^{-i\delta} - \frac{i}{\sqrt{2}} e^{-i(\delta - \pi/4)} F(s) - \frac{1}{\sqrt{2\pi v}} \left[\frac{e^{i(\beta_+ - \pi/4)}}{1 + v/u} + \frac{e^{i(\beta_- + \pi/4)}}{1 + \sqrt{v/u}} \right]$$

where

$$\delta = \frac{v^2}{2u}, \; \beta_\pm = \frac{u}{2} \pm v, \; s = \sqrt{u/\pi}(1 - v/u)$$

and $F(s) = C(s) - iS(s)$, the Fresnel sine and cosine integrals [see Section 3.4.6.1 for information on Fresnel integrals]. The outputs of the function are

alphaL [vector/matrix]	Normalized complex field ($W^{1/2}$).
`c` [vector]	Evaluation points along one axis (m).

3.4.2.9 DistortionNumber

Syntax

```
[Nd, NdScreen, NdIpk, NdIpkScreen, NdPkShft, TransScaling] = ...
    DistortionNumber(lambda, Pwr, Focus, D, Type, S, Atm)

[Nd, NdScreen, NdIpk, NdIpkScreen, NdPkShft, TransScaling] = ...
    DistortionNumber(Laser, S, Atm)

SCALING_CODE_API int DistortionNumber(char** errorChain,
    const char laserType, const int nInputs, const double* lambda,
    const double* power, const double* focus,
    const double* apertureDiameter, const double* pathLength,
    const int nScreens, const double* screenDistances,
    const double* screenThicknesses, const double* absorption,
    const double* scattering, const double* temperature,
    const double* windSpeed, double* nd, double* ndScreen, double* ndIpk,
    double* ndIpkScreen, double* ndPkShift, double* transWeight)
```

When a high energy laser beam propagates through the atmosphere, a portion of its energy is absorbed by the atmosphere which results in changes in its density and hence in its refractive index across the beam and along the path. The change in refractive index results in phase variation across the beam. The net effect at the target in the case of atmosphere is an increase in the beam size as well as a change in the beam shape (in most cases, it looks like a crescent). This in turn results in decreased peak intensity, a shift in peak intensity location and the encircled energy in the bucket. These effects together are called thermal blooming effects and are characterized by a parameter, called distortion number (N_d). This function calculates the integrated distortion number ***Nd*** and the distortion number for each screen ***NdScreen*** [18]. The Matlab inputs are:

Laser [struct]	Structure from LaserField or a comma-separated list of (***lambda***, ***Pwr***, ***Focus***, ***D***, ***Type***, ...).
lambda [scalar]	Wavelength of laser (m)
Pwr [scalar]	Total power of laser through link (W)
Focus [scalar]	Range at which laser beam is focused (m)
D [scalar]	Starting diameter of laser (m)
Type [char]	Beam type, must be 'Gaussian' or 'Uniform'.
S [struct/list]	Structure of engagement geometry parameters as from EngagementStruct. If a structure from GeomStruct is passed in as ***S***, the engagement structure will be computed.
Atm [struct]	Atmospheric modeling parameters from AtmStruct ***Atm*** must have wind profile if calculation with wind is desired.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
laserType [in]	Must be 'G'/'g' for Gaussian or 'U'/'u' for uniform; can only have one *laserType* for all inputs.
nInputs [in]	This function can be called with more than one input parameter set, stored in memory one after the other.
lambda [in]	Wavelength of laser (m); must be length *nInputs*.
power [in]	Total power of laser through link (W); must be length *nInputs*.
focus [in]	Range at which laser beam is focused (m); must be length *nInputs*.
apertureDiameter [in]	Starting diameter of laser (m); must be length *nInputs*.
pathLength [in]	Slant range from platform to target (m); must be length *nInputs*.
nScreens [in]	The number of phase screens in atmospheric path over which to realize atmospheric characteristics; must be the same for each input set.
screenDistances [in]	Phase screen distances from transmitter (m); must be length *nScreens* * *nInputs*.
screenThicknesses [in]	Phase screen thicknesses (m); must be length *nScreens* * *nInputs*.
absorption [in]	Absorption coefficients for the phase screens (1/m); must be length *nScreens* * *nInputs*.
scattering [in]	Scattering coefficients for the phase screens (1/m); must be length *nScreens* * *nInputs*.
temperature [in]	Temperature values for the phase screens (K); must be length *nScreens* * *nInputs*.
windSpeed [in]	Magnitude of path-transverse velocity (m/s); can be computed with TransVelocity; must be length *nScreens* * *nInputs*.
nd [out]	Integrated distortion number for the entire path; must be length *nInputs*.
ndScreen [out]	Distortion number for each phase screen; must be length *nScreens* * *nInputs*.
ndIpk [out]	(NULL OK if *ndIpkScreen*, *ndPkShift*, and *transScaling* are NULL) Peak irradiance scaling parameter anchored to wave optics, used by ThermalBlooming; must be length *nInputs*.

ndIpkScreen [out]	(NULL OK if ***ndIpk***, ***ndPkShift***, and ***transScaling*** are NULL) Path weighted distortion number for each phase screen; must be ***nScreens*** * ***nInputs***.
ndPkShift [out]	(NULL OK if ***ndIpk***, ***ndIpkScreen***, and ***transScaling*** are NULL) Scaling parameter for estimating peak shift, anchored to wave optics, used by THERMALBLOOMING; must be length ***nInputs***.
transWeight [out]	(NULL OK if ***ndIpk***, ***ndIpkScreen***, and ***ndPkShift*** are NULL) Transmission weighting function used for calculation of ***ndIpk*** and ***ndIpkScreen***; must be length ***nScreens*** * ***nInputs***.

The outputs are

Nd [scalar]	Integrated distortion number for the entire path.
NdScreen [vector]	Distortion number for each phase screen.
NdIpk [scalar]	Peak irradiance scaling parameter anchored to wave optics, used by THERMALBLOOMING
NdIpkScreen [vector]	Path weighted distortion number for each phase screen.
NdPkShft [scalar]	Scaling parameter for estimating peak shift, anchored to wave optics, used by THERMALBLOOMING.
TransScaling [vector]	Transmission weighting function used for calculation of ***NdIpk*** and ***NdIpkScreen***.

and are described in detail below.

The basis for the definition of the distortion number is that the phase distortion of a laser beam, propagating in the $z-$direction, can be calculated at any given location z as

$$\phi(r) = -N_d \int_{-\infty}^{x} dx' I(x', y, z)/I_0$$

where $I(x, y, z)$ is the intensity at location z on the axis and I_0 is the unperturbed on-axis intensity and N_d, is the distortion number. When the laser power varies significantly along the path, a lengthy calculation, starting from wave propagation analysis, shows that N_d can be defined as:

$$N_d = \frac{2\pi}{\lambda} \quad \frac{n_0 - 1}{C_p \rho_0} \quad 4\sqrt{2}P \int_0^L dz \frac{\alpha_{abs}(z)}{T(z)D(z))} \left[W(z)cos(\psi) + \dot{\psi} z\right]^{-1} \times \exp\left[-\int_0^z dz' \left[\alpha_{abs}(z') + \alpha_{scat}(z')\right]\right]$$

where λ is the wavelength, C_p is the specific heat at constant pressure, ρ_0 is the air density, n_0 is the refractive index of the unperturbed atmosphere, T is the temperature in Kelvin, P is the transmitter laser power, D is

the diameter of the beam , α_{abs} is the absorption coefficient of the air, α_{scat} is the scattering coefficient of the air, $W(z)$ is the wind velocity, $\dot{\psi}$ is the slew velocity and ψ is the angle between wind vector and propagation direction. The term $(n_0 - 1)/T(z)$ is equivalent to dn/dT, the derivative of the refractive index with respect to temperature at location z.

The diameter D is defined as the distance between between $1/e^2$ intensity points for a Gaussian beam where the beam is defined as $I(x, y) = I_0 \exp\left[-\frac{x^2+y^2}{R^2}\right]$ with $R = D/(2\sqrt{2})$. If the diameter varies along the path, it should be taken inside the integral sign for accuracy and accordingly implemented in this tool box.

For converging beams, it has been suggested to incorporate $(1 - z'/F)$ where F is the focal length of the beam within the integrand but this is not yet a universally accepted definition and hence is not used in this toolbox.

A related number is the collimated distortion number (N_c) which is defined as

$$N_c = \frac{n_0 - 1}{C_p \rho_0} \quad \frac{32\sqrt{2}P}{\pi D^3} \int_0^L dz\ z\ \frac{\alpha_{abs}(z)}{T(z)} \left[W(z)cos(\psi) + \dot{\psi} z\right]^{-1} \times \exp\left[-\int_0^z dz'\ [\alpha_{abs}(z') + \alpha_{scat}(z')]\right]$$

In the above equations, all variables are as defined earlier, k is the wave number, and L is the path length. The relationship between N_c and N_d is given via the Fresnel number (N_f) as

$$N_c = \frac{N_d}{2\pi N_f}$$

where N_f is $\frac{kD^2}{8L}$ for a Gaussian beam and $\frac{kD^2}{4L}$ for a uniform beam.

In anchoring the scaling law calculation of thermal blooming Strehl ratio to wave optics for 'UNIFORM' beam profiles, a relationship between the conventional distortion parameter ***Nd*** and Strehl could not be found. A relationship between distortion parameter and Strehl that worked well for level paths overestimated Strehl for down-looking paths, where the platform was higher in altitude than the target and absorption coefficient was increasing along the path. This same relationship underestimated Strehl in up-looking paths when absorption was decreasing along the path.

Therefore, a modified distortion parameter, ***NdIpk***, is calculated using a path weighting function and a transmission weighting function.[19] The primary function of the path weighting is to account for the fact that any phase added to the beam due to thermal blooming very near the target will not propagate into a significant irradiance change in the plane of the target. In addition, it was found that the peak of the path weighting had a Fresnel number dependence. The transmission weighting function is an adjustment to the power density of the beam that seems to account for the fact that thermal blooming effects at a given point in the path add some additional broadening on the beam. This in turn decreases the effects of thermal blooming at some later point in the path. The form of the path weighting and the expression for the transmission scaling factor are given in [19].

It is the parameter ***NdIpk*** that is used to compute Strehl for a Uniform beam in the function ThermalBlooming. The output ***NdIpkScreen*** is equivalent to ***NdScreen*** and includes the path weighting and transmission scaling. Another output ***NdPkShft***, is a parameter that is used to calculate the peak shift. Equivalent parameters have not yet been anchored for a Gaussian beam and so ThermalBlooming uses the conventional distortion number, ***Nd***.

3.4.2.10 FocusStrehl

Syntax

S = FocusStrehl(*FocusRange, TargetRange, lambda, D*)

```
SCALING_CODE_API int FocusStrehl(char** errorChain,
     const double* focusRange, const double* targetRange,
     const double* lambda, const double* apertureDiam, const int nInputs,
     double* strehl)
```

This function computes the Strehl ratio (peak irradiance decrease) for a beam focused at ***FocusRange*** but incident at ***TargetRange***.

FocusRange [vector]	Range at which transmitter is focused (m).
TargetRange [scalar]	Range at which beam is incident (m).
lambda [scalar]	Transmit laser wavelength (m).
D [scalar]	Transmit aperture diameter (m).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
focusRange [in]	Range at which transmitter is focused (m); must be length ***nInputs***.
targetRange [in]	Range at which the beam is incident (m); must be length ***nInputs***.
lambda [in]	Transmit laser wavelength (m); must be length ***nInputs***.
apertureDiam [in]	Transmit aperture diameter (m); must be length ***nInputs***.
nInputs [in]	Number of elements to compute.
strehl [out]	Strehl ratio for beam at target with defocus; allocate to length ***nInputs***.

The calculation is based on a Gaussian beam model for the beam spread of a defocused beam.

$$\begin{aligned} \sigma_{defocus} &= \sqrt{\left(1-\frac{L}{f}\right)^2+\left(\frac{L}{k_0 * r_0^2}\right)^2} \\ \sigma_{focus} &= \frac{L}{k_0 * r_0^2} \\ S &= \left(\frac{\sigma_{focus}}{\sigma_{defocus}}\right)^2 \end{aligned}$$

where $\sigma_{defocus}$ is the fraction of the transmit beam dimension represented by beam dimension at the target for a defocused beam and σ_{focus} is the fraction of the transmit beam dimension represented by beam dimension at target for a focused beam, k_0 is the wavenumber and r_0 is the equivalent size parameter of the transmit aperture D, equal to $D/(2\sqrt{2})$ for a Gaussian beam. The Strehl is then given by the ratio of the focused beam area to the defocused beam area (ratio of beam dimensions squared).

3.4.2.11 GreenwoodFreq

Syntax

fG = GreenwoodFreq(*alpha, lambda, S, Atm*)

```
SCALING_CODE_API int GreenwoodFreq(char** errorChain,
    const double* lambda, const double* cn2, const double* windVelocity,
    const double* totalPathLength, const double* screenSpacing,
    const int nPaths, const int nScreensPerPath, double* freq)
```

This function computes the Greenwood frequency for propagation given turbulence assumptions, engagement parameters, and wavelength. It returns *NaN* if the propagation path intersects earth's surface. The Matlab inputs are:

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
S [struct/list]	Engagement parameters. Must be a structure from EngagementStruct or GeomStruct.
Atm [struct/string]	Atmospheric modeling parameter structure from AtmStruct. *Atm* must have wind profile/heading if calculation with wind is desired.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength of laser (m); must be length *nPaths*.
cn2 [in]	Cn2 values for the phase screens; must be length *nScreensPerPath* x *nPaths* where each screen's turbulence value is sequential.
windVelocity [in]	Line-of-sight velocity transverse to the propagation direction for platform motion, target motion, and natural wind; must be length *nScreensPerPath* x *nPaths* where each screen's velocity is sequential.
totalPathLength [in]	Slant range; must be length *nPaths*.
screenSpacing [in]	Screen thickness for each phase screen; must be length *nScreensPerPath* x *nPaths* where each screen's spacing is sequential.
nPaths [in]	Number of atmospheric paths to evaluate.
nScreensPerPath [in]	Number of phase screens in each path being evaluated.

freq [out]	Greenwood frequency (Hz); NaN if propagation path intersects earth surface; allocate to length *nPaths*.

Engagement parameters must be input as a structure from EngagementStruct as shown in the following example.

Example 3.4.9 (*S* as a structure) *This example shows how to calculate the Greenwood frequency by inputting engagement parameters as a structure from* EngagementStruct. *(alpha = 1, lambda = 1.064e-6 m, and D = 0.5 m)*

```
>> G = GeomStruct('Simple',1000,10,20000,100,10,90,180);
>> S = EngagementStruct(G);
>> Atm = AtmStruct(G,8,'Cn2','HV57','Wind','Bufton','WindHeading','UniformAtm',-90);
>> fG = GreenwoodFreq(alpha,lambda,S,Atm)

fG =

  442.3940
```

Returns fG = NaN if the propagation path intersects the surface. If Atm and S are not consistent with respect to geometry, Atm will be updated. G and S are from the previous example.

The output is a vector *fG* with the value of the Greenwood frequency given by [20]

$$f_G = \left[0.102 k_0^2 \int_0^L C_n^2(h(z)) |V(z)|^{5/3} dz\right]^{3/5}$$

where $C_n^2(h(z))$ is the refractive-index structure function coefficient at the beam altitude $h(z)$, which is a function of the position z along the path, $k_0 = 2\pi/\lambda$ where λ is the wavelength of the HEL beam, and the integrals extend from the platform to the target. $V(z)$ is the apparent atmospheric velocity transverse to the path given by TransVelocity.

3.4.2.12 HigherOrderPSD

Syntax

[Omega, f, CLow, CHi, HOVarInt, HOVar] = HigherOrderPSD(*alpha, ... lambda, D, LogFMin, LogFMax, NF, S, Atm, [NZ]*)

```
SCALING_CODE_API int HigherOrderPSD(char** errorChain,
    const double lambda, const double D, const double logFmin,
    const double logFmax, const int nF, const double* cn2,
    const double* v, const double* z, const double* dz,
    const bool srFlag, const int nScreens, const double L, double* omega,
    double* f, double cLo[1], double cHi[1], double hoVarInt[1],
    double hoVar[1])
```

This routine computes the PSD of higher order phase aberrations caused by atmospheric turbulence after removal of piston and tilt. The temporal covariance of the phase is averaged over the aperture and Fourier transformed to give a single PSD. The function returns a warning if Simpson's Rule integration does not include at least 85% of the theoretical variance. The Matlab inputs are:

alpha [scalar]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
D [scalar]	Aperture Diameter (m).
LogFMin [scalar]	log10 of the minimum frequency for PSD calculation.
LogFMax [scalar]	log10 of the maximum frequency for PSD calculation.
NF [scalar]	Number of frequencies per decade (logarithmically spaced) for PSD calculation.
S [struct]	Engagement parameters. Can be a structure from EngagementStruct or GeomStruct. If a structure from GeomStruct is passed in as *S*, the engagement structure will be computed.
Atm [struct]	Atmospheric modeling parameter structure from AtmStruct. Can be a continuous or a discrete model.
NZ [scalar]	Number of evaluation points for the numerical integral (using Simpson's rule integration) with range. If *Atm* is a structure defining a discrete atmospheric model, *NZ* is ignored.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength (m); must be length 1.
D [in]	Aperture diameter (m); must be length 1.
logFmin [in]	log10 of the minimum frequency for PSD calculation.
logFmax [in]	log10 of the maximum frequency for PSD calculation.
nF [in]	Number of frequencies (logarithmically spaced) for PSD calculation.
cn2 [in]	Cn2 values for the phase screens; must be length *nScreens*.
v [in]	Transverse velocity values for the phase screens (m/s); must be length *nScreens*.
z [in]	Phase screen distances from transmitter (m); must be length *nScreens*.
dz [in]	Screen thickness for each phase screen; must be length *nScreens*.
srFlag [in]	Flag indicating use of Simpson's rule integration. If TRUE, Simpson's rule is used.

nScreens [in]	Number of phase screens specified - must be greater than 3 and even if *srFlag* is TRUE.
L [in]	Slant range (m); must be length 1.
omega [out]	PSD of higher order aberrations at the frequencies in the input vector *f* (rad^2/Hz); length *nF*.
f [out]	Frequencies for which the PSD is calculated (Hz); length *nF*.
cLo [out]	Coefficient for low frequency limit (rad^2/Hz).
cHi [out]	Coefficient for high frequency limit (rad^2/Hz).
hoVarInt [out]	Higher order phase variance as integrated from the computed PSD (rad^2).
hoVar [out]	Theoretical higher order variance (rad^2).

The outputs are:

Omega [vector]	PSD of higher order aberrations at the frequencies in the input vector *f*. The PSD is obtained by averaging the pointwise PSDs of the phase AFTER TILT REMOVAL. The units are rad^2/Hz.
f [vector]	Frequencies for which the PSD is calculated.
CLow [scalar]	Coefficient for low frequency limit. In the limit of low frequency, the PSD goes to the constant *CLow*.
CHi [scalar]	Coefficient for high frequency limit. In the limit of high frequency, the PSD goes as $CHi \times f^{-8/3}$.
HOVarInt [scalar]	Higher order phase variance as integrated from the computed PSD.
HOVar [scalar]	Theoretical higher order variance.

The PSD calculated in this routine is one half of a two-sided PSD. The integral under the PSD as computed must be doubled to account for power at negative frequencies. There is a convenient way to calculate the total power when the PSD is generated for a vector of logarithmically-spaced frequencies.

Example 3.4.10 (PSD of higher order phase aberrations) *This example shows how to compute the PSD of higher order phase aberrations caused by atmospheric turbulence after removal of piston and tilt with a turbulence multiplier of one, wavelength of 1.064 μm, and aperture diameter of 0.5 m.*

```
>> G = GeomStruct('Simple',1000,10,20000,100,10,0,180);
>> S = EngagementStruct(G);
>> Atm = AtmStruct(G,8,'Cn2','Clear1Night','Wind','Bufton');
>> [Omega,f,CLow,CHi,HOVarInt,HOVar] = HigherOrderPSD(1,1.064e-6,...
```

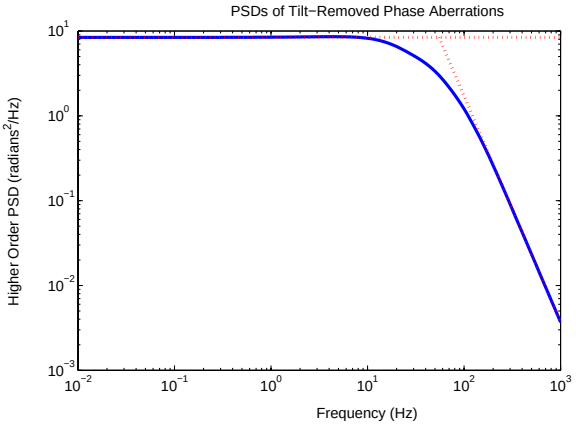


Figure 3.13: PSD from HigherOrderPSD.

```
                                                    0.5,-2,3,30,S,Atm)

CLow =

    8.3994


CHi =

  3.6970e+005


HOVarInt =

  984.5804


HOVar =

  984.0273
```

The outputs `Omega` *and* `f` *have been plotted in Figure 3.13.*

`HigherOrderPSD(...)` with no outputs will generate a loglog plot of the PSD as in Figure 3.13

3.4.2.13 IrradianceCovariancePathWeight

Syntax

W = IrradianceCovariancePathWeight(*xi, d, wvl, L, sourceStr, [LAMBDA], [THETA]*)

```
SCALING_CODE_API int IrradianceCovariancePathWeight(char** errorChain,
    const int nx, double* x, const double s, const double wvl,
    const double slantRange, const double* lambda, const double* theta,
    double* pathWeight)
```

This function computes a weighting of Cn2 (turbulence strength) over the propagation path for the irradiance covariance. Handles plane, spherical, and Gaussian beam cases (can be numerically unstable in beam wave cases). Uses the weak turbulence assumption.[7] The inputs are

xi [vector]	Normalized location along the propagation path.
d [scalar]	Magnitude of the separation of points in the observation plane.
wvl [scalar]	Optical wavelength (m).
L [scalar]	Propagation distance (m).
sourceStr [string]	Type of optical field source. Either 'plane', 'spherical', or 'beam'.
LAMBDA [scalar]	(Optional) Gaussian beam target plane diffraction parameter. Defaults to 0 (infinite beam).
THETA [scalar]	(Optional) Gaussian beam target plane refraction parameter. Defaults to 0 (focused).

The output *W* is the C_n^2 weighting along the path.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements in vector *x*.
x [in]	Normalized location along the propagation path.
s [in]	Magnitude of the separation of points in the observation plane.
wvl [in]	Optical wavelength [m].
slantRange [in]	Propagation distance [m].
sourceStr [in]	Specifies the type of optical field source, either 'plane' or 'spherical'.
pathWeight [out]	Path weighting evaluated along the path.

Example 3.4.11 (IrradianceCovariancePathWeight) *Example computation of the C_n^2 path weighting using* IrradianceCovariancePathWeight.

```
>> Atm  = AtmStruct(0, 2e4, 0, 100, 'Cn2', 'HV57');
>> d = 2e-2;  % points separated by 2 cm at the pupil
>> W = IrradianceCovariancePathWeight(Atm.z/Atm.L, d, 0.5e-6, Atm.L, 'spherical');
```

Path weighting function for irradiance covariance with wavelength of 500 nm, point source at 20 km propagating straight down, pupil-plane separation 2 cm, and atmospheric model Atm.

3.4.2.14 NollMatrix

Syntax

ZernikeCov = NollMatrix(*N*)

```
SCALING_CODE_API int NollMatrix(char** errorChain, const int n,
    double* zernikeCovMat)
```

This function computes the covariance matrix of Zernike polynomials with $N \geq 2$ for propagation through Kolmogorov turbulence (no anisoplanatism) given the maximum Zernike order under Noll's ordering scheme [21]. Diagonal elements are the modal variance of each atmospheric Zernike mode. The covariance matrix, Γ_a, is given by

$$\Gamma_a = \overline{\mathbf{a}\mathbf{a}^T}$$

where **a** is the column vector of expansion coefficients. The elements of the covariance matrix are given by [21, 22]

$$\begin{aligned}\overline{a_i a_j} &= 0.0072\left(\frac{D}{r_0}\right)^{5/3}(-1)^{(n_i+n_j-2m_i)/2}[(n_i+1)(n_j+1)]^{1/2}\pi^{8/3}\delta_{m_i m_j} \\ &\times \frac{\Gamma(14/3)\Gamma[(n_i+n_j-5/3)/2]}{\Gamma[(n_i-n_j+17/3)/2]\Gamma[(n_j-n_i+17/3)/2]\Gamma[(n_i+n_j+23/3)/2]}\end{aligned}$$

for $i-j =$ even, and $\overline{a_i a_j} = 0$ for $i-j =$ odd. m_i and n_i refer to the azimuthal and radial orders associated with the i^{th} Zernike polynomial, respectively. The above equation can be used to obtain expressions for the aperture averaged mean square phase error after removal of the first N Zernike modes.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Maximum Zernike order (Noll's ordering).
zernikeCovMat [out]	Covariance matrix for Zernike polynomials 2:N normalized to $(D/r_0)^{5/3}$; size must be n-1 x n-1; this is equal to its transpose.

3.4.2.15 OpenLoopJitter

Syntax

sigma = OpenLoopJitter(*r0, D, lambda*)

```
SCALING_CODE_API int OpenLoopJitter(char** errorChain,
    const double* r0, const double* apDiam, const double* lambda,
    const int nElements, double* sigmaJitter)
```

This function computes the jitter of a laser propagated through atmospheric turbulence. The Matlab inputs are:

r0 [vector]	Atmospheric phase coherence length (Fried's parameter) (m).
D [vector]	Diameter of transmit aperture (m).
lambda [vector]	Laser wavelength of propagating beam (m).

The output *sigma* is the standard deviation ($1 \times \sigma$) for a Gaussian probability density of atmospheric tilt, given by the following equation [23].

$$\sigma = \sqrt{B_Z \frac{4}{\pi^2} \left(\frac{D}{r_0}\right)^{5/3} \left(\frac{\lambda}{D}\right)^2}$$

where B_Z is the covariance matrix of a Zernike polynomial of N = 2 calculated using NollMatrix.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
r0 [in]	Atmospheric phase coherence length (Fried's parameter) (m).
apDiam [in]	Diameter of transmit aperture (m).
lambda [in]	Laser wavelength of propagating beam (m).
nElements [in]	The length of each input and output array.
sigmaJitter [out]	Standard deviation of beam jitter (rad).

3.4.2.16 OpenLoopStrehl

Syntax

[Strehl, StrehlTR] = OpenLoopStrehl(*r0, D, [Dobs], [method]*)

```
SCALING_CODE_API int OpenLoopStrehl(char** errorChain,
    const double* r0, const double* apDiam, const double* obsDiam,
    const int nInputs, const int method, double* strehl,
    double* strehlTiltRemoved)
```

This function computes the Strehl ratio of a laser propagated through a turbulent medium [14]. The Matlab inputs are:

r0 [vector]	Atmospheric phase coherence length (Fried's parameter) (m).
D [vector]	Diameter of transmit aperture (m).

Dobs [vector]	(Optional) Diameter of transmitter central obscuration (m).
method [scalar]	(Optional) Method for computing *Strehl*, 0 for LUT (default) or 1 for analysis. Must specify *Dobs* (can be []) to specify *method*.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
r0 [in]	Atmospheric phase coherence length (Fried's parameter) (m); must be length *nInputs*.
apDiam [in]	Diameter of transmit aperture (m); must be length *nInputs*.
obsDiam [in]	Diameter of transmitter central obscuration (m); must be length *nInputs*.
nInputs [in]	Number of strel ratios to compute.
method [in]	Method for computing Strehl, 0 for lookup table or 1 for analysis.
strehl [out]	(NULL OK) Strehl ratio for tilt-included beam.
strehlTiltRemoved [out]	(NULL OK) Strehl ratio for tilt-removed beam (higher-order only).

The outputs are the Strehl ratio for tilt-included, *Strehl*, and the Strehl ratio for a tilt-removed beam, *StrehlTR* (higher-order only). The optical transfer function (OTF) is given by [24]

$$\mathcal{H}(\vec{f}) = \mathcal{H}_o(\vec{f})\mathcal{H}_{LE}(\vec{f})$$

where

$$\begin{aligned} \mathcal{H}_o(\vec{f}) &= \frac{W(\vec{f}\lambda d_i) \star W(\vec{f}\lambda d_i)}{W(0) \star W(0)} \\ \mathcal{H}_{LE}(\vec{f}) &= \exp\left\{ -\frac{1}{2} 6.88 \left(\frac{\bar{\lambda} d_i |\vec{f}|}{r_o} \right)^{5/3} \right\} \end{aligned}$$

where $W(\vec{f}\lambda d_i)$ is the Fourier transform of the pupil function (ones inside pupil, zeros elsewhere) and where $\star$ represents correlation defined by

$$f(\vec{x}) \star g(\vec{x}) = \int d\vec{x}' f(\vec{x}' - \vec{x}) g^*(\vec{x}')$$

The Strehl ratio is then given by [14]

$$SR = \frac{s(\vec{x} = 0)}{s_{dl}(\vec{x} = 0)}$$

where $s(\vec{x})$ is PSF given as the inverse Fourier transform of the OTF and $s_{dl}(\vec{x})$ is the diffraction-limited PSF. To calculate the tilt-removed Strehl, ***StrehlTR***, replace the long-exposure OTF with a short-exposure OTF given by

$$\mathcal{H}_{SE}(\vec{f}) = \exp\left\{-\frac{1}{2}6.88\left(\frac{|\bar{\lambda}d_i\vec{f}|}{r_o}\right)^{5/3}\left[1-\left(\frac{|\bar{\lambda}d_i\vec{f}|}{D}\right)^{1/3}\right]\right\}$$

3.4.2.17 PLANED0

Syntax

dO = PLANED0(*alpha, lambda, Geom, Lb, Atm*)

This function computes the section size of a laser guide star beacon (*dO*) given turbulence assumptions, propagation geometry, and wavelength [16]. Output is appropriate for computing residual phase variance for compensation from laser guide star beacon as $(D/dO)^{5/3}$. Assumes the target object is at a long range compared to the beacon and hence plane wave analysis is appropriate. However, integration of C_n^2 is done over the propagation path to target specified by *Geom*. Returns *NaN* if propagation path intersects earth surface. The inputs are:

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
Geom [struct/list]	Geometry parameters. Can be a structure from GEOMSTRUCT‡ or a comma separated list of (..., *hp*, *ht*, *rd*) - not required if *Atm* is a structure from ATMSTRUCT.
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).
Lb [scalar]	Slant range of beacon (m).
Atm [struct/string]	Atmospheric modeling parameters. Can be a structure from ATMSTRUCT or a turbulence profile model to be used.

3.4.2.18 PLANEIRRADIANCECOVARIANCE

Syntax

D = PLANEIRRADIANCECOVARIANCE(*alpha, d, Atm, wvl*)

This function computes the irradiance covariance for a plane wave given turbulence multiplier alpha, point spacings d, atmospheric modeling data in Atm, and wavelength in wvl [7]. In this calculation, the light propagates from the target to the platform, which is consistent with calculations of PLANERYTOV and PLANER0. Uses the weak turbulence assumption. The inputs are

alpha [scalar]	Turbulence strength multiplier.
d [vector]	Separation of the points in the aperture plane (m).
Atm [struct]	Atmospheric modeling structure from AtmStruct.
wvl [scalar]	Optical wavelength (m).

The output is

B [vector]	Irradiance covariance value.

Example 3.4.12 (PlaneIrradianceCovariance) *Illustrate use of* PlaneIrradianceCovariance.

```
>> G = SimpleGeom(0, 2e4, 0);
>> Atm  = AtmStruct(G, 100, 'Cn2', 'HV57');
>> d = linspace(0, 0.5, 30);
>> B = PlaneIrradianceCovariance(1, d, Atm, 0.5e-6);
```

Irradiance covariance value for turbulence multiplier of 1, wavelength 0f 500 nm, planar source at 20 km propagating straight down, pupil-plane separations `d`*, and atmospheric model* `Atm`*.*

3.4.2.19 PlaneR0

Syntax

[r0, r0Screen] = PlaneR0(*alpha, lambda, Geom, Atm*)

This function computes the plane wave coherence diameter (Fried parameter) given turbulence assumptions, propagation geometry, and wavelength [25]. It also calculates the screen *r0* if the input *Atm* is a discrete structure from AtmStruct. It returns *NaN* if the propagation path intersects earth's surface (determined by IsGoodGeom). The inputs are:

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list of (..., *hp*, *ht*, *rd*) - not required if *Atm* is a structure from AtmStruct.
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).

Atm [struct/string]	Atmospheric modeling parameters. Can be a structure from AtmStruct or a turbulence profile model to be used.

The function returns:

r0 [vector]	Plane wave coherence diameter (m).
r0Screen [matrix]	Plane wave coherence diameter at each phase screen.

The output, *r0*, is the plane wave coherence diameter in meters and is given by

$$r_0 = \left[0.423 k_0^2 \int_0^L C_n^2(h(z))dz\right]^{-3/5}$$

where $C_n^2(h(z))$ is the refractive-index structure function coefficient at the beam altitude $h(z)$, which is a function of the position z along the path, $k_0 = 2\pi/\lambda$ where λ is the wavelength of the laser, L is the slant range, and the integral extends from the platform to the target. The output *r0Screen* is simply the integrand of the above equation giving an r0 for each screen. *r0Screen* is also calculated in the function ScreenR0 [see Section 3.4.2.25]. The calling sequence for PlaneR0 is the same as that of SphericalR0. Several examples of the available calling sequences are in Section 3.4.2.30 and will not be repeated here.

3.4.2.20 PlaneRytov

Syntax

Rytov = PlaneRytov(*alpha, lambda, Geom, Atm*)

This function computes the plane wave *Rytov* number given turbulence assumptions, propagation geometry, and wavelength [26] as follows:

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list of (..., *hp*, *ht*, *rd*) - not required if *Atm* is a structure from AtmStruct.
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).

Atm [struct/string]	Atmospheric modeling parameters. Can be a structure from AtmStruct or a turbulence profile model to be used.

The output *Rytov* is the plane wave Rytov number given by

$$\sigma_R^2 = 0.563 k_0^{7/6} \int_0^L C_n^2(h(z)) z^{5/6} dz$$

where $C_n^2(h(z))$ is the refractive-index structure function coefficient at the beam altitude $h(z)$, which is a function of the position z along the path, $k_0 = 2\pi/\lambda$ where λ is the wavelength of the laser, L is the slant range, and the integral extends from the platform to the target. The function returns *NaN* if the propagation path intersects the earth's surface. The calling sequence for PlaneRytov is the same as that of SphericalRytov. Several examples of the available calling sequences are in Section 3.4.2.31 and will not be repeated here.

3.4.2.21 PlaneWaveStrFcn

Syntax

D = PlaneWaveStrFcn(*alpha, d, Atm, wvl*)

This function computes the wave structure function for a plane wave in units of $(rad)^2$ given turbulence multiplier *alpha*, point spacings *d*, atmospheric modeling data in *Atm*, and wavelength in *wvl*. The inputs are:

alpha [scalar]	Turbulence strength multiplier.
d [vector]	Separation of the points in the aperture plane (m).
Atm [struct]	Atmospheric modeling structure from AtmStruct.
wvl [scalar]	Optical wavelength (m).

Since the calculation is for plane wave, the light propagation direction does not matter. The output is:

D [vector]	Structure function value (rad^2).

Example 3.4.13 (PlaneWaveStrFcn) *This example illustrates use of* PlaneWaveStrFcn.

```
>> Atm  = AtmStruct(0, 2e4, 0, 100, 'Cn2', 'HV57');
>> d = linspace(0, 0.5, 30);
>> D = PlaneWaveStrFcn(1, d, Atm, 0.5e-6);
```

Wave structure function value for turbulence multiplier of 1, wavelength 0f 500 nm, point source at 20 km propagating straight down, separation vector d, and atmospheric model Atm.

3.4.2.22 PropParams

Syntax

p = PropParams(*[alpha], [lambda], [D], [Power], G, Atm*)

This function returns an output structure containing values for atmospheric propagation parameters computed with the following inputs:

alpha [vector]	(Optional) Multiplier on turbulence strength only.
lambda [vector]	(Optional) Wavelength for propagation (m).
D [vector]	(Optional) Transmit aperture diameter (m).
Power [vector]	(Optional) Laser power (W).
G [struct]	Geometry structure from GeomStruct.
Atm [struct]	Atmospheric modeling structure from AtmStruct.

If any optional inputs are to be specified, all previous optional inputs must be specified also. For example to specify *lambda* = 1.06 micron, one must also specify *alpha*.

```
p = PropParams(1,1.06e-6,G,Atm)
```

If none of the optional inputs are specified, the defaults are *alpha* = 1, *lambda* = 1 micron, *D* = 1 m, *Power* = 1 MW. The output structure is:

p [struct]	Structure of propagation parameters.
p.power [scalar]	Power for parameter calculations (W).
p.wavelength [scalar]	Wavelength for parameter calculations (m).
p.D [scalar]	Aperture diameter for parameter calculations (m).
p.Trans [scalar]	Atmospheric transmission given input abs/scat.
p.Rytov [scalar]	Spherical wave Rytov number [see SphericalRytov].
p.Rytov_PW [scalar]	Plane wave Rytov number [see PlaneRytov].
p.r0 [scalar]	Spherical wave coherence diameter [see SphericalR0] (m).
p.r0_PW [scalar]	Plane wave coherence diameter [see PlaneR0] (m).
p.theta0 [scalar]	Isoplanatic angle [see SphericalTheta0] (rad).
p.fG [scalar]	Greenwood frequency [see GreenwoodFreq] (Hz).
p.fT [scalar]	Tyler frequency [see TylerFreq] (Hz).
p.Nd [scalar]	Thermal blooming distortion number [see DistortionNumber].

Any or all of the inputs values may be arrays, in which case output *p* is a struct array. However, if any of the inputs are arrays, all inputs must be either scalar or the same length.

3.4.2.23 PupilProp

Syntax

[irr, x, y] = PupilProp(*D, lambda, L, F, ObsRatio, N, ... ScreenCn2, NormScreenPos, NormScreenDepth, RandStart, NumRand*)

[irr, x, y] = PupilProp(*D, lambda, L, F, ObsRatio, N, ... Atm, RandStart, NumRand*)

This function computes the irradiance distribution after propagation from a uniformly illuminated annular aperture. Allows for propagation through Kolmogorov phase screens of arbitrary strength distributed arbitrarily over the propagation path. The inputs are:

D [scalar]	Aperture diameter (m).
lambda [scalar]	Wavelength (m).
L [scalar]	Path length (m).
F [scalar]	Focus range of pupil optics (m).
ObsRatio [scalar]	Obscuration ratio of annular aperture.
N [scalar]	Size of propagation grid.
Atm [struct]	Structure from AtmStruct or a comma-separated list of (*ScreenCn2*, *NormScreenPos*, *NormScreenDepth*).
ScreenCn2 [vector]	Strength of turbulence on each phase screen ($m^{-2/3}$).
NormScreenPos [vector]	Position of phase screens normalized to path length.
NormScreenDepth [vector]	Thickness of phase screens normalized to path length.
RandStart [scalar]	Index of for initial phase screen random seed.
NumRand [scalar]	Number of random draws for turbulence phase screens.

If screens are not specified, vacuum propagation is considered. When turbulence is specified, PupilProp returns the average irradiance over random phase screen realizations.

3.4.2.24 RefractAlt

Syntax

[hRef, dh] = RefractAlt(*x, hp, ht, rd, lambda, EarthRadius*)

```
SCALING_CODE_API int RefractAlt(char** errorChain, const int nInputs,
    const double* x, const int nx, const double* hp, const double* ht,
    const double* rd, const double* re, const int nLambda,
    const double* lambda, double* hRef, double* dh)
```

RefractAlt calculates the new altitudes, *hRef*, due to refraction of light along the path, and the difference between these altitudes and the original refraction-free altitudes, *dh* based on [13]. The Matlab inputs are

x [vector]	Normalized path position from aircraft to target or number of segments along the path. Use 0.9999 to calculate altitude at the target.
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list of (..., *hp*, *ht*, *rd*, ...).
hp [vector]	Altitude of transmit/receive platform (m).
ht [vector]	Altitude of target (m).
rd [vector]	Downrange of target along curved earth surface (m).
lambda [vector]	Wavelength of laser (m).
EarthRadius [scalar]	(Optional) Spherical Earth radius (m). Default is from PhysicalConst('remean').

The function issues a warning if any of the new altitudes intersect earth's surface.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nInputs [in]	This function allows multiple inputs/outputs, one right after the other in memory.
x [in]	Normalized path position from aircraft to target at which to calculate height after refraction; must be length *nx* * *nInputs*.
nx [in]	Number of path positions; must be the same for each input.
hp [in]	Altitude of transmit/receive platform (m); must be length *nInputs*.

ht [in]	Altitude of target (m); must be length *nInputs*.
rd [in]	Downrange of target along curved earth surface (m); must be length *nInputs*.
re [in]	Spherical Earth radius (m); must be length *nInputs*.
nLambda [in]	Number of wavelengths; must be 1 or *nInputs* if *nInputs* ¿ 1.
lambda [in]	Wavelength of laser (m); must be *nLambda*.
hRef [out]	Altitudes due to refraction of light along the path (m); will be *nx* * *nInputs* where *nx* is the leading dimension.
dh [out]	(NULL OK) Difference between altitudes due to diffraction and the original diffraction-free altitudes (m); will be *nx* * max(*nInputs*,*nLambda*) where *nx* is the leading dimension.

3.4.2.25 ScreenR0

Syntax

r0s = ScreenR0(*alpha, lambda, Geom, Atm, [Nscreens], [intFlag]*)

```
SCALING_CODE_API int ScreenR0(char** errorChain, const double* lambda,
    const double* cn2, const double* screenSpacing, const int nPaths,
    const int nScreensPerPath, double* r0s)
```

This function computes the coherence diameter (Fried parameter) for each phase screen given turbulence assumptions, propagation geometry, and wavelength. Returns *NaN* if propagation path intersects earth surface. The Matlab inputs are:

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list of (..., *hp*, *ht*, *rd*) - not required if *Atm* is a structure from AtmStruct.
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).
Atm [struct/string]	Atmospheric modeling parameters. May be a structure from AtmStruct or turbulence profile to be used.

Nscreens [scalar]	(Optional) Number of phase screens. Required if *Atm* is a turbulence profile string or *Atm* is not a discrete structure.
intFlag [logical]	(Optional) Flag indicating the integral type. If *intFlag*=false (default) calculation will use *Atm.Cn2.*∗*Atm.dz.* If *intFlag*=true, calculation will integrate *Atm.Cn2Eval* over the phase screen segments.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength of laser (m); must be length *nPaths*.
cn2 [in]	Cn2 values for the phase screens; must be length *nScreensPerPath* x *nPaths* where each screen's turbulence value is sequential.
screenSpacing [in]	Screen thickness for each phase screen (m); must be length *nScreensPerPath* x *nPaths* where each screen's spacing is sequential.
nPaths [in]	Number of atmospheric paths to evaluate.
nScreensPerPath [in]	Number of phase screens in each path being evaluated.
r0s [out]	Coherence diameter for each phase screen (m); must be length *nScreensPerPath* x *nPaths* where each path's *r0* values are sequential.

The output, *r0s*, is the plane wave coherence diameter for each phase screen, given by

$$r_0(z) = \left[0.423 k_0^2 C_n^2(h(z))\Delta z\right]^{-3/5}$$

The plane wave coherence diameter from PLANER0 can be calculated as

$$r_0 = \left[\Sigma r_0(z_i)^{-5/3}\right]^{-3/5}$$

3.4.2.26 SPHERICALIRRADIANCECOVARIANCE

Syntax

D = SPHERICALIRRADIANCECOVARIANCE(*alpha, d, Atm, wvl*)

This function computes the irradiance covariance for a spherical wave given turbulence multiplier, point spacings, atmospheric modeling data, and wavelength. In this calculation, the light propagates from the target to the platform, which is consistent with calculations of SphericalRytov and SphericalR0. Uses the weak turbulence assumption.[7] The inputs are

alpha [scalar]	Turbulence strength multiplier.
d [vector]	Separation of the points in the aperture plane (m).
Atm [struct]	Atmospheric modeling structure from AtmStruct.
wvl [scalar]	Optical wavelength (m).

The output is

B [vector]	Irradiance covariance value.

Example 3.4.14 (SphericalIrradianceCovariance) *Example illustrating use of* SphericalIrradianceCovariance.

```
>> G = SimpleGeom(0, 2e4, 0);
>> Atm  = AtmStruct(G, 100, 'Cn2', 'HV57');
>> d = linspace(0, 0.5, 30);
>> B = SphericalIrradianceCovariance(1, d, Atm, 0.5e-6);
```

Irradiance covariance value for turbulence multiplier of 1, wavelength 0f 500 nm, collimated Gaussian source with beam radius 5 cm at 20 km propagating straight down, pupil-plane separations `d`*, and atmospheric model* `Atm`*.*

3.4.2.27 SphericalIrradianceVariance

Syntax

sigI2 = SphericalIrradianceVariance(*alpha, lambda, Geom, Atm*)

This function computes the spherical wave normalized irradiance variance given turbulence assumptions, propagation geometry, and wavelength. Returns *NaN* if propagation path intersects earth surface. Propagation is from platform to the target and this function returns the target plane variance. Accounts for strong turbulence conditions.[7] The inputs are

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma-separated list of (...,hp,ht,rd) - not required if Atm is a structure from AtmStruct.

hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).
Atm [struct/string]	Atmospheric modeling parameters. Can be a structure from AtmStruct or a turbulence profile model to be used.

The output is the normalized spherical wave irradiance variance.

3.4.2.28 SphericalLogIrrCovariance

Syntax

[BlnX, BlnY] = SphericalLogIrrCovariance(*alpha, d, Atm, wvl, [type]*)

This function computes the large and small scale log-irradiance covariance for a spherical wave under strong fluctuation theory. The inputs are

alpha [scalar]	Turbulence strength multiplier.
d [vector]	Separation of the points in the irradiance profile (m).
Atm [struct]	Atmospheric modeling structure from AtmStruct.
wvl [scalar]	Optical wavelength (m).
type [string]	(Optional) Scale of covariance to output if the number of outputs is one. Ignored for two outputs. Specify 'large' (default), 'small', or 'both' for combined.

The outputs are

BlnX [vector]	Large scale log-irradiance covariance value.
BlnY [vector]	Small scale log-irradiance covariance value.

The log irradiance values are given by [7]

$$B_{\mathrm{lnX,Y}}(\rho) = 8\pi^2 k^2 L \int_0^1 \int_0^\infty \kappa \Phi_n(\kappa) G_{\mathrm{X,Y}}(\kappa) J_o(\kappa\xi\rho) \left\{ 1 - \cos\left[\frac{L\kappa^2}{k} \xi(1-\xi) \right] \right\} \mathrm{d}\kappa \mathrm{d}\xi$$

where $B_{\ln X}$ is the covariance attributed to the large-scale eddies, $B_{\ln U}$ is the covariance attributed to the small-scale eddies, $k = 2\pi/\lambda$, L is the slant range, $\Phi_n(\kappa)$ is the power spectral density for the index variations, κ is spatial frequency, J_o is the Bessel function of the first kind (order 0), and $\xi = z/L$ is the normalize path position. The filter functions $G_{X,Y}(\kappa)$ are given by [27, 28]

TODO

For Kolmogorov turbulence, this can be written as

$$B_{\mathrm{lnX,Y}}(\rho) = 0.033(4\pi^2)k^{7/6}L^{11/6}\int_0^1 C_n^2(\xi)W_{\mathrm{X,Y}}(\xi,\rho)\mathrm{d}\xi$$

where $W_{X,Y}(\xi,\rho)$ is the path weighting function as computed by SphericalLogIrrPathWeight. Then the irradiance covariances are given by

$$\begin{aligned} B_X &= (\exp(B_{\ln X}) - 1) \\ B_Y &= (\exp(B_{\ln Y}) - 1) \\ B_I &= (\exp(B_{\ln X} + B_{\ln Y}) - 1) \end{aligned}$$

where B_I is the full irradiance covariance.

Example 3.4.15 (SphericalLogIrrCovariance) *General description*

```
>> G = SimpleGeom(0, 2e4, 0);
>> NZ = 100;
>> Atm  = AtmStruct(G, NZ, 'Cn2', 'HV57');
>> wvl = 0.5e-6;
>> d = 0.02;  % points separated by 2 cm
>> [BlnX, BlnY] = SphericalLogIrrCovariance(1, d, Atm, wvl);
```

Compute large scale (`BlnX`) and small scale (`BlnY`) log irradiance covariance.

3.4.2.29 SphericalLogIrrPathWeight

Syntax

[Wx, Wy] = SphericalLogIrrPathWeight(*xi, dhat, Rytov, [type]*)

This function computes weighting of C_n^2 (turbulence strength) over propagation path for the log-irradiance covariance for a spherical wave. Can return the large scale (*Wx*), small scale (*Wy*), both, or *Wx* + *Wy* (single output with *type* = 'both'). This function uses the integral formulation from Andrews and Phillips and can be computationally intensive. The inputs are

xi [vector]	Normalized location along the propagation path.
dhat [scalar]	Normalized (by $\sqrt{L/k}$) separation in the observation plane.
Rytov [scalar]	Spherical wave Rytov number from SphericalRytov.
type [string]	(Optional) Scale of covariance to output if the number of outputs is one. Either 'large' (default if number of outputs is 1), 'small', or 'both' (default if number of outputs is 2).

The outputs are

Wx [vector]	Large scale path weighting (or *Wx* + *Wy* if type is 'both') evaluated along the path.
Wy [vector]	Small scale path weighting evaluated along the path.

The path weighting functions are given by [7]

TODO

Example 3.4.16 (SphericalLogIrrPathWeight) *Compute combined large and small scale covariance path weighting for a simple geometry.*

```
>> G = SimpleGeom(0, 2e4, 0);
>> NZ = 100;
>> Atm  = AtmStruct(G, NZ, 'Cn2', 'HV57');
>> wvl = 0.5e-6;
>> dhat = 0.02/sqrt(Atm.L*wvl/(2*pi));  % points separated by 2 cm
>> x = Atm.z / Atm.L;
>> Rytov = SphericalRytov(1,wvl,Atm);
>> W = SphericalLogIrrPathWeight(x,dhat,Ryt,'both');
```

Path weighting functions for irradiance covariance with wavelength of 500 nm, point source at 20 km propagating straight down, pupil-plane separation 2 cm, and atmospheric model from ***Atm****.*

3.4.2.30 SphericalR0

Syntax

r0 = SphericalR0(*alpha, lambda, Geom, Atm*)

```
SCALING_CODE_API int SphericalR0(char** errorChain,
     const double* lambda, const double* cn2,
     const double* totalPathLength, const double* screenDistances,
     const double* screenSpacing, const int nPaths,
     const int nScreensPerPath, double* r0)
```

This function computes the spherical wave coherence diameter (Fried parameter) given turbulence assumptions, propagation geometry, and wavelength. It returns *NaN* if the propagation path intersects earth's surface (determined by IsGoodGeom). The Matlab inputs are:

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list (. . . ,*hp* ,*ht* ,*rd*) - not required if *Atm* is a struct from AtmStruct.
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).
Atm [struct/string]	Atmospheric modeling parameters. Can be a structure from AtmStruct or a turbulence profile model to be used.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength of laser (m); must be length *nPaths*.
cn2 [in]	Cn2 values for the phase screens; must be length *nScreensPerPath* x *nPaths* where each screen's turbulence value is sequential.
totalPathLength [in]	Slant range; must be length *nPaths*.
screenDistances [in]	Phase screen distances from transmitter (m); must be length *nScreensPerPath* x *nPaths* where each screen's spacing is sequential.
screenSpacing [in]	Screen thickness for each phase screen; must be length *nScreensPerPath* x *nPaths* where each screen's spacing is sequential.
nPaths [in]	Number of atmospheric paths to evaluate.
nScreensPerPath [in]	Number of phase screens in each path being evaluated.
r0 [out]	Spherical wave coherence diameter (m); should be allocated to length *nPaths*.

There are two calling sequences based on the type of *Atm*:

1. *Atm* as a structure from AtmStruct

Example 3.4.17 (*Atm* as a structure) *This example shows how to calculate the Fried parameter by inputting a structure from* AtmStruct*.*

```
>> r0 = SphericalR0(1.0,1.0e-6,AtmStruct)

r0 =

    0.3655
```

Compute r_0 at 1 micron using the turbulence model in the input *Atm* *struct.* *Geom* *is not needed with this calling sequence.*

```
>> r0 = SphericalR0(2.0,1.0e-6,GeomStruct,AtmStruct)
```

Compute r_0 scaling the input turbulence model by a factor of 2 in addition to any turbulence multiplier already in the *Atm* *structure. If* *Geom* *and geometry in* *Atm* *are not consistent,* *Atm* *will be updated.*

2. *Atm* as a string representing the turbulence profile model.

Example 3.4.18 (*Atm* as a string) *This example shows how to calculate the Fried parameter by inputting a string for the turbulence profile model.*

```
>> r0 = SphericalR0(2.0,1.0e-6,GeomStruct,{'HufnagelValley',15,1e-15})
```

Compute r_0 scaling the input turbulence model by a factor of 2. To pass additional parameters to the model, pass the model name and parameters in a cell array.

```
>> r0 = SphericalR0(1,1.315e-6,1000,10,20000,'HV57')

r0 =

    0.1538
```

Geom input as a comma-separated list with a continuous Atm model.

```
>> r0 = SphericalR0([1 2],1.064e-6,G,'HV57')

r0 =

    0.1193    0.0787
```

Input alpha as a vector. G is the geometry structure from Example 3.4.9.

The output, *r0*, is a vector the length of *alpha* with the value of the spherical wave coherence diameter in meters given by [29]

$$r_0 = \left[0.423k_0^2 \int_0^L C_n^2(h(z))(1 - z/L)^{5/3} dz\right]^{-3/5}$$

where $C_n^2(h(z))$ is the refractive-index structure function coefficient at the beam altitude $h(z)$, which is a function of the position z along the path, $k_0 = 2\pi/\lambda$ where λ is the wavelength of the laser, L is the slant range, and the integral extends from the platform to the target.

3.4.2.31 SphericalRytov

Syntax

Rytov = SphericalRytov(*alpha, lambda, Geom, Atm*)

```
SCALING_CODE_API int SphericalRytov(char** errorChain,
    const double* lambda, const double* cn2,
    const double* totalPathLength, const double* screenDistances,
    const double* screenSpacing, const int nPaths,
    const int nScreensPerPath, double* rytov)
```

This function computes the spherical wave Rytov number (σ_χ^2) given turbulence assumptions, propagation geometry, and wavelength. It returns *NaN* if the propagation path intersects earth's surface (determined by IsGoodGeom). The Matlab inputs are:

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).

Geom [struct/list]	Geometry parameters. Can be a structure from GEOMSTRUCT) or a comma separated list (. . . ,*hp* ,*ht* ,*rd*) - not required if *Atm* is a struct from ATMSTRUCT.
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).
Atm [struct/string]	Atmospheric modeling parameters. Can be a structure from ATMSTRUCT or a turbulence profile model to be used.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength of laser (m); must be length *nPaths* .
cn2 [in]	Cn2 values for the phase screens; must be length *nScreensPerPath* x *nPaths* where each screen's turbulence value is sequential.
totalPathLength [in]	Slant range; must be length *nPaths* .
screenDistances [in]	Phase screen distances from transmitter (m); must be length *nScreensPerPath* x *nPaths* where each screen's spacing is sequential.
screenSpacing [in]	Screen thickness for each phase screen; must be length *nScreensPerPath* x *nPaths* where each screen's spacing is sequential.
nPaths [in]	Number of atmospheric paths to evaluate.
nScreensPerPath [in]	Number of phase screens in each path being evaluated.
rytov [out]	Spherical wave rytov number; should be allocated to length *nPaths* .

There are two calling sequences based on the type of *Atm*:

1. *Atm* as a structure from ATMSTRUCT

Example 3.4.19 (*Atm* as a structure) *This example shows how to calculate the Rytov number by inputting a structure from* ATMSTRUCT.

```
>> Rytov = SphericalRytov(1.0,1.0e-6,AtmStruct)
```

Compute Rytov at 1 micron using the turbulence model in the input *Atm* *struct.* *Geom* *is not needed with this calling sequence.*

```
>> Rytov = SphericalRytov(2.0,1.0e-6,GeomStruct,AtmStruct)
```

Compute Rytov scaling the input turbulence model by a factor of 2 in addition to any turbulence multiplier already in the ***Atm*** *struct. If* ***Geom*** *and geometry in* ***Atm*** *are not consistent,* ***Atm*** *will be updated.*

2. ***Atm*** as a string representing the turbulence profile model.

Example 3.4.20 (*Atm* as a string) *This example shows how to calculate the Rytov number by inputting a string for the turbulence profile model.*

```
>> Rytov = SphericalRytov(2.0,1.0e-6,GeomStruct, ...
                {'HufnagelValley',15,1e-15})
```

Compute Rytov scaling the input turbulence model by a factor of 2. To pass additional parameters to the model, pass the model name and parameters in a cell array.

```
>> Rytov = SphericalRytov(1,1.315e-6,1000,10,20000,'HV57')

Rytov =

    0.6247
```

Geom *input as a comma separated list.*

```
>> Rytov = SphericalRytov([1 2],1.064e-6,G,'HV57')

Rytov =

    0.7998    1.5996
```

G *is the geometry structure from Example 3.4.9 and* ***alpha*** *is a vector.*

The output, ***Rytov***, is a vector the length of ***alpha*** with the value of the spherical wave Rytov number given by [26]

$$\sigma_\chi^2 = 0.563 k_0^{7/6} \int_0^L C_n^2(h(z)) z^{5/6} (1 - z/L)^{5/6} dz$$

where $C_n^2(h(z))$ is the refractive-index structure function coefficient at the beam altitude $h(z)$, which is a function of the position z along the path, $k_0 = 2\pi/\lambda$ where λ is the wavelength of the laser, L is the slant range, and the integral extends from the platform to the target.

3.4.2.32 SphericalTheta0

Syntax

Theta0 = SphericalTheta0(*alpha, lambda, Geom, Atm*)

```
SCALING_CODE_API int SphericalTheta0(char** errorChain,
    const double* lambda, const double* cn2,
    const double* totalPathLength, const double* screenDistances,
    const double* screenSpacing, const int nPaths,
    const int nScreensPerPath, double* theta0)
```

This function computes the spherical wave isoplanatic angle (θ_0) given turbulence assumptions, propagation geometry, and wavelength. It returns *NaN* if the propagation path intersects earth's surface (determined by IsGoodGeom). The Matlab inputs are

alpha [vector]	Multiplier on turbulence model.
lambda [scalar]	Wavelength of laser (m).
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct) or a comma separated list (...,*hp*,*ht*,*rd*) - not required if *Atm* is a struct from AtmStruct.
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).
Atm [struct/string]	Atmospheric modeling parameters. Can be a structure from AtmStruct or a turbulence profile model to be used.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength of laser (m); must be length *nPaths*.
cn2 [in]	Cn2 values for the phase screens; must be length *nScreensPerPath* x *nPaths* where each screen's turbulence value is sequential.
totalPathLength [in]	Slant range; must be length *nPaths*.
screenDistances [in]	Phase screen distances from transmitter (m); must be length *nScreensPerPath* x *nPaths* where each screen's spacing is sequential.
screenSpacing [in]	Screen thickness for each phase screen; must be length *nScreensPerPath* x *nPaths* where each screen's spacing is sequential.
nPaths [in]	Number of atmospheric paths to evaluate.
nScreensPerPath [in]	Number of phase screens in each path being evaluated.
theta0 [out]	Spherical wave isoplanatic angle (rad); should be allocated to length *nPaths*.

There are two calling sequences based on the type of *Atm*:

1. *Atm* as a structure from AtmStruct

Example 3.4.21 (*Atm* as a structure) *This example shows how to calculate the isoplanatic angle by inputting a structure from* AtmStruct.

```
>> theta0 = SphericalTheta0(1.0,1.0e-6,AtmStruct)

theta0 =

   3.1544e-07
```

Compute θ_0 at 1 micron using the turbulence model in the input *Atm* *struct.* *Geom* *is not needed with this calling sequence.*

```
>> theta0 = SphericalTheta0(2.0,1.0e-6,GeomStruct,AtmStruct)
```

Compute θ_0 scaling the input turbulence model by a factor of 2 in addition to any turbulence multiplier already in the *Atm* *struct. If* *Geom* *and geometry in* *Atm* *are not consistent,* *Atm* *will be updated.*

2. *Atm* as a string representing the turbulence profile model.

Example 3.4.22 (*Atm* as a string) *This example shows how to calculate the isoplanatic angle by inputting a string for the turbulence profile model.*

```
>> theta0 = SphericalTheta0(2.0,1.0e-6,GeomStruct, ...
                {'HufnagelValley',15,1e-15})
```

Compute θ_0 scaling the input turbulence model by a factor of 2. To pass additional parameters to the model, pass the model name and parameters in a cell array.

```
>> Theta0 = SphericalTheta0(1,1.315e-6,1000,10,20000,'HV57')

Theta0 =

  5.3870e-007
```

Geom *input as a comma separated list with continuous turbulence model.*

```
>> Theta0 = SphericalTheta0([1 2],1.064e-6,G,'HV57')

Theta0 =

  1.0e-006 *

    0.4178    0.2756
```

G *is the geometry structure from Example 3.4.9 and* *alpha* *is a vector.*

The output, *Theta0*, is a vector the length of *alpha* with the values of the spherical wave isoplanatic angle in radians given by [30]

$$\theta_0 = \left[2.91k_0^2 \int_0^L C_n^2(h(z))z^{5/3}dz\right]^{-3/5}$$

where $C_n^2(h(z))$ is the refractive-index structure function coefficient at the beam altitude $h(z)$, which is a function of the position z along the path, $k_0 = 2\pi/\lambda$ where λ is the wavelength of the laser, and the integral extends from the platform to the target.

3.4.2.33 SphericalWaveStrFcn

Syntax

D = SphericalWaveStrFcn(*alpha, d, Atm, wvl*)

This function computes the wave structure function for a spherical wave in units of rad^2 given turbulence multiplier *alpha*, point spacings *d*, atmospheric modeling data in *Atm*, and wavelength in *wvl*. The inputs are

alpha [scalar]	Turbulence strength multiplier.
d [vector]	Separation of the points in the aperture plane (m).
Atm [struct]	Atmospheric modeling structure from AtmStruct.
wvl [scalar]	Optical wavelength (m).

In this calculation, the light propagates from the target to the platform.

D [vector]	Structure function value (rad^2).

Example 3.4.23 (Use of *SphericalWaveStrFcn*) *The following example shows a calculation of the spherical wave structure function.*

```
>> Atm  = AtmStruct(0, 2e4, 0, 100, 'Cn2', 'HV57');
>> d = linspace(0, 0.5, 30);
>> D = SphericalWaveStrFcn(1, d, Atm, 0.5e-6);
```

Wave structure function value for turbulence multiplier of 1, wavelength 0f 500 nm, point source at 20 km propagating straight down, separation vector d, and atmospheric model Atm.

3.4.2.34 StrehlEquivFocus

Syntax

[FocusRangePast, FocusRangePrior] = StrehlEquivFocus(*S, ... TargetRange, lambda, D*)

```
SCALING_CODE_API int StrehlEquivFocus(char** errorChain,
    const double* strehlRatio, const int nStrehlRatio,
    const double* targetRange, const double* wavelength,
    const double* apDiam, double* focusRangePast,
    double* focusRangePrior)
```

This function computes a focusing range for a laser beam that gives an equivalent Strehl ratio for propagation through vacuum as that specified by the input S. Since this can be accomplished by focusing past the target or by focusing prior to the target, both solutions are given. The Matlab inputs are:

S [vector]	Strehl ratio for beam at target with defocus.
TargetRange [scalar]	Range at which beam is incident (m).
lambda [scalar]	Transmit laser wavelength (m).
D [scalar]	Transmit aperture diameter (m).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
strehlRatio [in]	Strehl ratio for beam at target with defocus; must be length *nStrehlRatio*.
nStrehlRatio [in]	Length of the *strehlRatio* array and also length of the output array(s).
targetRange [in]	Range at which beam is incident (m); must be of length *nStrehlRatio*.
wavelength [in]	Transmit laser wavelength (m); must be of length *nStrehlRatio*.
apDiam [in]	Transmit aperture diameter (m); must be of length *nStrehlRatio*.
focusRangePast [out]	(NULL OK) Range at which transmitter is focused past the target to give an equivalent Strehl (m); can be length *nStrehlRatio* or NULL to indicate that this value is not needed.
focusRangePrior [out]	(NULL OK) Range at which transmitter is focused prior to the target to give an equivalent Strehl (m); can be length *nStrehlRatio* or NULL to indicate that this value is not needed.

The Matlab outputs are:

FocusRangePast [vector]	Range at which transmitter is focused past the target to give an equivalent Strehl (m).

FocusRangePrior [vector]	Range at which transmitter is focused prior to the target to give an equivalent Strehl (m).

If the input Strehl is very small, FocusRangePast may give a negative focus range (indicating a divergent beam). This function is the inverse of FocusStrehl.

3.4.2.35 TBWaveCalc

Syntax

```
[Irr, x, y, t, Ipk, Irel, Shft, Sig, W0, Field] = ...
    TBWaveCalc(lambda, Pwr, Focus, D, Type, S, Atm, [PC])

[Irr, x, y, t, Ipk, Irel, Shft, Sig, W0, Field] = ...
    TBWaveCalc(Laser, S, Atm, [PC])

[Irr, x, y, t, Ipk, Irel, Shft, Sig, W0, Field] = ...
    TBWaveCalc([Type], PropConfigFile, [PC])

TB = TBWaveCalc(...)
```

This function computes thermal blooming[5] and turbulence combined effects on wave-optics propagation [31]. It can also be used to model the effects of thermal blooming or turbulence, separately. It models propagation of a laser source with arbitrary wavelength, power, focus distance, and size/type given an atmospheric specification structure (*Atm*) from AtmStruct, the engagement parameter velocity decomposition, and optionally, a propagation control structure (*PC*). If TBWaveCalc is called with a single output and no inputs, the default propagation control structure is returned, i.e. `PC = TBWaveCalc`. The fields *PC.dt* and *PC.propDxy*, if left empty, will be computed using the default method. The inputs are

Laser [struct]	Structure from LaserField or a comma-separated list of (*lambda*, *Pwr*, *Focus*, *D*, *Type*, ...).
lambda [scalar]	Wavelength of laser (m)
Pwr [scalar]	Total power of laser through link (W)
Focus [scalar]	Range at which laser beam is focused (m)
D [scalar]	Starting diameter of laser (m)
Type [char/struct]	Beam type, must be 'GAUSSIAN', 'UNIFORM' or a structure containing a near-field pattern. If 'GAUSSIAN', $I(r) = I_0 \exp[-(r/\sigma)^2]$, where $\sigma = D/2\sqrt{2}$. If 'UNIFORM', $I(r)$ is constant over full aperture diameter D. Near-field structure must have fields .x, .y, and .g and should not include any focus phase.
S [struct]	Engagement parameters. Can be a structure from EngagementStruct or GeomStruct. If a structure from GeomStruct is passed in as *S*, the engagement structure will be computed.

[5] The mex-system dll's for propagation with thermal blooming are of limited distribution to government entities and government contractors working on government programs. Request access by contacting atmtools@mza.com.

Atm [struct]	Atmospheric modeling parameters. Must be a discrete model from AtmStruct Propagation path length is set from *Atm.L*. *Atm* must have wind profile if calculation with wind is desired.
PropConfigFile [char]	Name of a data file saved by PropConfig. Uses wavelength, power, focus range, aperture diameter and geometry and atmosphere.
PC [struct]	(Optional) Propagation control parameters for wave-optics modeling.
PC.propNxy [scalar]	Propagation mesh dimension (defaults to 256 by 256).
PC.propDxy [scalar]	Propagation mesh resolution in *x* and *y* defaults to dxy to aperture function or focus term (m).
PC.ApplyBlooming [scalar]	Flag (1/0) indicating whether thermal blooming phase will be applied in propagation (defaults to 1).
PC.RandIndex [scalar]	Index for phase screen random seed (defaults to 0).
PC.dt [scalar]	Delta time between propagation steps in simulation. Defaults to value giving max step motion equal to 1/10 beam diameter (s).
PC.TimeSteps [scalar/string]	Number of *dt* time steps in simulation (defaults to 30). If identifier 'STEADY-STATE' is provided, simulation will run until peak irradiance convergence or 50 time steps, whichever comes first.
PC.ConvergeThresh [scalar]	Convergence threshold for 'STEADY-STATE.' Defaults to 0.05.
PC.ReportSteps [string]	Identifier of which steps to report from simulation. 'ALL' - report metrics at each time step (default), 'AVG' - report metrics from average of time steps, 'LAST' - report metrics from the last time step.
PC.ReportBegin [scalar]	Time step at which report begins (defaults to 1).
PC.DisplaySteps [string]	Identifier determining whether propagation time steps are displayed on screen (defaults to 'NO') 'YES' - display irradiance at each step, 'NO' - don't display irradiance at each step.
PC.DLLName [string]	Runset name for compiled DLL to use. Default uses CWSourceThermBloomRunScreens in ATMTools.
PC.tempusBin [string]	Full path to tempus/bin directory. Default uses tempusSupport/bin as distributed with ATMTools. Specify as empty (`''`) to use tempus/bin from the system path.
PC.callbackFcn [function_handle]	Callback function that is checked at each iteration of the loop. If the value returned from the callback function is false, execution will stop. Uses progressbar by default. Set to `@(v)(true)` to suppress.

The input *Atm* must have fields *Abs*, *Scat*, and *Temp* in order to model thermal blooming. If *Atm.Cn2* is specified, wave-optics propagation will also include turbulence phase screens as specified by *Atm.z*. Input *Atm* may also include models for inner and outer scale given the C_n^2 model employed. The outputs are

Irr [matrix]	Irradiance after propagation (W/m^2).
x [vector]	X locations of irradiance values (m).
y [vector]	Y locations of irradiance values (m).
t [vector]	Time at which peak irradiance values are reported (every time step) (s).
Ipk [vector]	Peak irradiance values at every time step (W/m^2).
Irel [vector]	Peak irradiance relative to vacuum–includes atmospheric attenuation–Strehl=*Irel*/Transmission(*Atm*).
Shft [vector]	Shift of peak irradiance from optical axis [X, Y] (m).
Sig [vector]	Sigma value for Gaussian fit to target irradiance [X, Y] (m). Irradiance is fit to a Gaussian form: $\exp[-0.5((X/\sigma_X)^2 + (Y/\sigma_Y)^2)]$.
WO [struct]	Configuration and inputs used for wave-optics modeling.
WO.PC [struct]	Input propagation control structure.
WO.dllName [string]	Name of DLL used to perform wave-optics calculations.
WO.P	Parameter list used by DLL wave-optics model.
WO.Laser [struct]	Structure used to simulate the laser.
WO.Time [scalar]	Runtime for wave-optics model (s).
Field [matrix]	Complex field value after propagation, i.e., *Irr*=*Field*.*conj(*Field*).
TB [struct]	A structure whose fields contain the above data. If only a single output is requested, the output will be a structure.

TBWaveCalc uses tempus™ mex-system propagation model and accompanying TMX toolbox (see `HELP tmx.tbx`). The following are available `mex` systems for use in TBWaveCalc:

- CWSourceThermBloomRunScreens - Open-loop, plane wave propagation including turbulence and thermal blooming
- CWSourceTurbRunScreens - Open-loop, plane wave propagation including turbulence only
- CWSourceTurbSWPRunScreens - Open-loop, spherical wave propagation including turbulence only.

To use a dll that is not the default, you need to specify the DLL name in the propagation control structure input to TBWaveCalc, i.e.

```
>> PC.DLLName = 'CWSourceTurbSWPRunScreens';
```

If using CWSourceTurbSWPRunScreens, *PC.propDxy* must be specified as a two element vector with the platform pixel size and the target pixel size, i.e.

```
>> PC.propDxy = [0.003 0.03];
```

Example 3.4.24 (TBWaveCalc) *These examples show general use of* TBWaveCalc.

```
>> [Irr,x,y,t,Irel,Shft,Sig,W0,Field] = TBWaveCalc(1.064e-6,...
           10e3,inf,0.5,'UNIFORM',GeomStruct,AtmStruct)
```

Simulate propagation for a uniform beam with default propagation parameters.

```
>> x = ((1:512)-257)*0.005; y = x;
>> Laser = LaserField(x,y,'GAUSSIAN',25e3,1.0e-6,50e3,0.25,0.5);
>> [Irr,x,y,t,Irel,Shft,Sig,W0,Field] = ...
           TBWaveCalc(Laser,GeomStruct,AtmStruct);
```

Set up the laser field structure for a clipped Gaussian beam and simulate the propagation using the input geometry and atmosphere.

Example 3.4.25 (TBWaveCalc with PropConfig data) *These examples illustrate use of data files saved by* PropConfig *with* TBWaveCalc.

```
>> [Irr,x,y,t,Ipk,Irel,Shft,Sig,W0,Field] = TBWaveCalc(PropConfigFile);
```

Gets wavelength, power, focus range and aperture diameter from the PropConfig *data file and simulates a 'UNIFORM' laser with the geometry and atmosphere as setup in* PropConfig.

```
>> [Irr,x,y,t,Ipk,Irel,Shft,Sig,W0,Field] = ...
           TBWaveCalc(NF,PropConfigFile);
```

Uses wavelength, power, focus range and aperture diameter from the PropConfig *data file to scale the input near-field pattern and simulates the propagation with the geometry and atmosphere as setup in* PropConfig.

3.4.2.36 ThermalBlooming

Syntax

[TB, TBPhase] = ThermalBlooming(*Method, lambda, Pwr, Focus, D, ... Type, S, Atm, [P1, P2, ...PN]*)

[TB, TBPhase] = ThermalBlooming(*Method, Laser, S, Atm, [P1, P2, ...PN]*)

```
SCALING_CODE_API int ThermalBlooming(char** errorChain,
    char laserType, const int nInputs, const double* lambda,
    const double* power, const double* focus,
    const double* apertureDiameter, const double* pathLength,
    const int nScreens, const double* screenDistances,
    const double* screenThicknesses, const double* absorption,
    const double* scattering, const double* temperature,
    const double* windSpeed, double* nd, double* ndIpk, double* nc,
    double* ndPkShft, double* pkShftP, double* sprdP, double* sprdT,
    double* strehlA, double* strehlB, double* ndPhase,
    double* ndIpkPhase, double* ncPhase, double* ndPkShftPhase,
    double* pkShftPPhase, double* sprdPPhase, double* sprdTPhase,
    double* strehlAPhase, double* strehlBPhase)
```

This function computes the effect of thermal blooming using the specified analysis method given by input *Method* for the propagation scenario described by the inputs. The Matlab inputs are:

Method [string]	Identifier for analysis methodology (e.g., 'SCALING').
Laser [struct]	Structure from LaserField or a comma separated list of (... *lambda*, *Pwr*, *Focus*, *D*, *Type*, ...).
lambda [scalar]	Wavelength of laser (m)
Pwr [scalar]	Total power of laser through link (W)
Focus [scalar]	Range at which laser beam is focused (m)
D [scalar]	Starting diameter of laser (m)
Type [char]	Beam type, must be 'Gaussian' or 'Uniform'.
S [struct]	Structure of engagement geometry parameters as from EngagementStruct. If a structure from GeomStruct is passed in as *S*, the engagement structure will be computed.
Atm [struct]	Atmospheric modeling parameters from AtmStruct. *Atm* must have wind profile if calculation with wind is desired.
P1...PN [list]	Additional parameters required for specified analysis method.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
laserType [in]	Must be 'G'/'g' for Gaussian or 'U'/'u' for uniform; can only have one *laserType* for all inputs.
nInputs [in]	This function can be called with more than one input parameter set, stored in memory one after the other.
lambda [in]	Wavelength of laser (m); must be length *nInputs*.
power [in]	Total power of laser through link (W); must be length *nInputs*.
focus [in]	Range at which laser beam is focused (m); must be length *nInputs*.
apertureDiameter [in]	Starting diameter of laser (m); must be length *nInputs*.
pathLength [in]	Slant range from platform to target (m); must be length *nInputs*.
nScreens [in]	The number of phase screens in atmospheric path over which to realize atmospheric characteristics; must be the same for each input set.
screenDistances [in]	Phase screen distances from transmitter (m); must be length *nScreens* * *nInputs*.
screenThicknesses [in]	Phase screen thicknesses (m); must be length *nScreens* * *nInputs*.
absorption [in]	Absorption coefficients for the phase screens (1/m); must be length *nScreens* * *nInputs*.
scattering [in]	Scattering coefficients for the phase screens (1/m); must be length *nScreens* * *nInputs*.
temperature [in]	Temperature values for the phase screens (K); must be length *nScreens* * *nInputs*.
windSpeed [in]	Magnitude of path-transverse velocity (m/s); can be computed with TransVelocity; must be length *nScreens* * *nInputs*.
nd [out]	Distortion number for focused laser beam in propagation path; must be length *nInputs*.
ndIpk [out]	Scaling parameter for estimation of peak irradiance; must be length *nInputs*.
nc [out]	Distortion number for collimated laser beam in propagation path; must be length *nInputs*.
ndPkShft [out]	Scaling parameter for estimation of peak shift; must be length *nInputs*.

pkShftP [out]	Angular shift of peak irradiance (rad); must be length *nInputs*.
sprdP [out]	Beam spread in P axis relative to diffraction; must be length *nInputs*.
sprdT [out]	Beam spread in T axis relative to diffraction; must be length *nInputs*.
strehlA [out]	Relative peak irradiance computed via method A above (based on ndIpk); must be length *nInputs*.
strehlB [out]	Relative peak irradiance computed via method B above (based on ndIpk); must be length *nInputs*.
ndPhase [out]	(NULL OK if all phase outputs are NULL) Distortion number for focused laser beam in propagation path; must be length *nInputs*.
ndIpkPhase [out]	(NULL OK if all phase outputs are NULL) Scaling parameter for estimation of peak irradiance; must be length *nInputs*.
ncPhase [out]	(NULL OK if all phase outputs are NULL) Distortion number for collimated laser beam in propagation path; must be length *nInputs*.
ndPkShftPhase [out]	(NULL OK if all phase outputs are NULL) Scaling parameter for estimation of peak shift; must be length *nInputs*.
pkShftPPhase [out]	(NULL OK if all phase outputs are NULL) Angular shift of peak irradiance (rad); must be length *nInputs*.
sprdPPhase [out]	(NULL OK if all phase outputs are NULL) Beam spread in P axis relative to diffraction; must be length *nInputs*.
sprdTPhase [out]	(NULL OK if all phase outputs are NULL) Beam spread in T axis relative to diffraction; must be length *nInputs*.
strehlAPhase [out]	(NULL OK if all phase outputs are NULL).
strehlBPhase [out]	(NULL OK if all phase outputs are NULL).

Currently supported analysis methods are:

'SCALING' Computes angular peak shift, beam spread (relative to diffraction), and Strehl ratio using the following empirical scaling laws based on wave-optics modeling of thermal blooming effects for input *Type*.[31] See Section 3.4.2.9 for a description of the different distortion parameters.

'Uniform' In the case of an 'UNIFORM' beam, the scaling parameter for calculation of Strehl is *NdIpk* (from DistortionNumber). *PkShiftP* is calculated based on the parameter *NdPkShft* (also

from DISTORTIONNUMBER). The expressions for Strehl are given by [see [19]]

$$\begin{aligned}
\theta_P &= 2.16 \times 10^{-4}\sqrt{N_{PkShft}\frac{\lambda}{D}} + 1.75 \times 10^{-2}\, N_{PkShft}\frac{\lambda}{D} \\
\sigma_P &= 1 + 8.95 \times 10^{-4} N_{Ipk} \\
\sigma_T &= 1 + 3.50 \times 10^{-2} N_{Ipk} \\
S_A &= (1 + 3.84 \times 10^{-2} N_{Ipk})^{-1} \\
S_B &= (\sigma_P\, \sigma_T)^{-1}
\end{aligned}$$

'Gaussian' For 'GAUSSIAN,' all calculations use the conventional distortion parameter ***Nd***, and so ***TB.NdIpk*** and ***TB.NdPkShft*** will be returned as []. The expressions for the outputs are given by [see [32]]

$$\begin{aligned}
S_A &= (1 + 1.61 \times 10^{-2} N_d + 2.70 \times 10^{-3} N_d^2 + -1.31 \times 10^{-6} N_d^3)^{-1} \\
\theta_P &= (N_d/10) 0.85 \times 10^{-6} \\
\sigma_P &= (1 + 0.0016 N_d^{3/2}) \\
\sigma_T &= (1 + 0.0111 N_d^{3/2}) \\
S_B &= (\sigma_P\, \sigma_T)^{-1}
\end{aligned}$$

TB.StrehlA is based on a fit of a calculated distortion parameter to the amount of increase in beam area from the diffraction-limited case. ***TB.StrehlB*** is based on the amount of increase in the beam dimensions (***SprdP*** and ***SprdT***). Method A and Method B will give approximately equal Strehl ratio values. A single thermal blooming Strehl can be obtained by

$$S = \sqrt{(S_A)(S_B)}$$

The thermal blooming Strehl ratio reported by the 'SCALING' method is the peak irradiance relative to a beam propagated through vacuum, but experiencing the same attenuation (absorption/scattering) as through the blooming volume. Multiply by atmospheric transmission to compute the peak irradiance reduction relative to the vacuum peak irradiance. The output when 'SCALING' method is used is:

Returns

TB [struct]	Metrics associated with thermal blooming analysis with different fields depending on ***Method***.
TB.FocusRange [scalar]	Focus range used for computing effects of thermal blooming.
TB.Nd [scalar]	Distortion number for focused laser beam in propagation path.
TB.NdIpk [scalar]	Scaling parameter for estimation of peak irradiance.
TB.Nc [scalar]	Distortion number for collimated laser beam in propagation path.
TB.NdPkShft [scalar]	Scaling parameter for estimation of peak shift.
TB.PkShiftP [scalar]	Angular shift of peak irradiance (rad).
TB.SprdP [scalar]	Beam spread in P axis relative to diffraction.
TB.SprdT [scalar]	Beam spread in T axis relative to diffraction.
TB.StrehlA [scalar]	Relative peak irradiance computed via method A above (based on ***NdIpk***).

TB.StrehlB [scalar]	Relative peak irradiance computed via method B above (based on *NdIpk*).
TBPhase [struct]	Beam metrics associated with thermal blooming analysis on the phase of the incident irradiance with the same fields as *TB*.

The output structure *TBPhase* has the same form as TB, but models the effect of thermal blooming on the phase and would be applicable to cases where the incident irradiance is propagated to a focus, e.g. in a relay uplink. The distortion number used is *Nd*, path and transmission weighting are not used in this case. The relationships used are:

$$\begin{aligned} \theta_P &= 0.040704 N_d \frac{\lambda}{D(1-L/F)} \\ S_A &= (1 + 4.3833 \times 10^{-4} N_{dIpk}^2)^{-1} \\ \sigma_P &= 1 + 5.274 \times 10^{-7} N_{dIpk}^3 \\ \sigma_T &= 1 + 2.840 \times 10^{-4} N_{dIpk}^2 \\ S_B &= (\sigma_P \, \sigma_T)^{-1} \end{aligned}$$

'WAVEOPTICS' Computes thermal blooming beam metrics via a MEX system wave-optics simulation of steady-state thermal blooming [31]. Uses the basic function TBWaveCalc for calculation [see Section 3.4.2.35]. Propagation parameters can be controlled through a *PropControl* input as in TBWaveCalc. If propagation control is not exercised, the simulation parameters will be automatically computed given other inputs. The output of ThermalBlooming when 'WAVEOPTICS' method is used is:

TB [struct]	Beam metrics associated with thermal blooming analysis with different fields depending on *Method*.
TB.Irr [matrix]	Irradiance after propagation (W/m^2).
TB.x [vector]	X locations of irradiance values (m).
TB.y [vector]	Y locations of irradiance values (m).
TB.t [vector]	Time at which irradiance values are reported (s).
TB.Ipk [vector]	Peak irradiance values (W/m^2).
TB.Irel [vector]	Peak irradiance relative to vacuum–includes atmospheric attenuation.
TB.Shft [vector]	Shift of peak irradiance from optical axis [X,Y] (m).
TB.Sig [vector]	Sigma value for Gaussian fit to target irradiance [X,Y] (m). Irradiance is fit to a Gaussian form: $\exp[-0.5((X/\sigma_X)^2 + (Y/\sigma_Y)^2)]$.
TB.WO [struct]	Configuration and inputs used for wave-optics modeling.
WO.PC [struct]	Input propagation control structure.

WO.dllName [string]	Name of DLL used to perform wave-optics calculations.
WO.P	Parameter list used by DLL wave-optics model.
WO.Laser [struct]	Structure used to simulate laser.
WO.Time [scalar]	Runtime for wave-optics model (s).

3.4.2.37 TiltAngCovScale

Syntax

[thetaCovX, thetaCovY, tvar] = TiltAngCovScale(*lambda, D, G, ... Atm, [DispCov]*)

```
SCALING_CODE_API int TiltAngCovScale(char** errorChain,
    const double lambda, const double D, const double* posPlatEcf,
    const double* posTargEcf, const double* velPlatEcf,
    const double* velTargEcf, const double* re, const int nRE,
    const int nScreens, const double* z, const double* dz,
    const double* cn2, const double* windHeading,
    const double* horzWindSpeed, const double* vertWindSpeed,
    double* thetaCovX, double* thetaCovY, double* tvar, double* aOut,
    double* tcovXOut, double* tcovYOut)
```

This function computes the covariance scale lengths for modeling Kolmogorov angular tilt anisoplanatism as a Markov random process in 2 dimensions. The Matlab inputs are:

lambda [scalar]	Wavelength for propagation (m).
D [scalar]	Diameter of receive/transmit aperture (m).
G [struct]	Geometry structure from GeomStruct.
Atm [struct]	Atmospheric modeling structure from AtmStruct.
DispCov [scalar]	(Optional) Flag (1/0) for covariance and fit plot.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength (m).
D [in]	Aperture diameter (m).

posPlatEcf [in]	ECF position vector(s) for platform (m); must be length 3. This function currently only works for a single geometry.
posTargEcf [in]	ECF position vector(s) for target (m); must be length 3.
velPlatEcf [in]	ECF velocity vector(s) for platform (m); must be length 3.
velTargEcf [in]	ECF velocity vector(s) for target (m); must be length 3.
re [in]	Specify the earth radius belonging to each geometry; must be length *nRE*.
nRE [in]	Specify 1 for spherical earth with radius *re* or 2 for oblate earth; must be 1 or 2.
nScreens [in]	Number of phase screens specified for each geometry.
z [in]	Screen distances from transmitter (m); must be length *nScreens*.
dz [in]	Screen thickness for each phase screen; must be length *nScreens*.
cn2 [in]	Cn2 values for each phase screen; must be length *nScreens*.
windHeading [in]	Wind heading for each phase screen (deg); must be length *nScreens*.
horzWindSpeed [in]	Horizontal wind speed for each phase screen (m/s); must be length *nScreens*.
vertWindSpeed [in]	Vertical wind speed for each phase screen (m/s); must be length *nScreens*.
thetaCovX [out]	Covariance scale for X-tilt with X ang offset (rad).
thetaCovY [out]	Covariance scale for Y-tilt with X ang offset (rad).
tvar [out]	Variance of tilt fluctuations (rad^2).
aOut [out]	Angular separation vector for info display (deg); if not NULL, must be allocated to length 20.
tcovXOut [out]	Tilt covariance in X for info/display (rad); if not NULL, must be allocated to length 20.
tcovYOut [out]	Tilt covariance in Y for info/display (rad); if not NULL, must be allocated to length 20.

The Matlab outputs are:

thetaCovX [scalar]	Covariance scale for X-tilt with X ang offset (rad).
thetaCovY [scalar]	Covariance scale for Y-tilt with X ang offset (rad).
tvar [scalar]	Variance of tilt fluctuations (rad^2).

Returns angular scale lengths and tilt variance quantities for modeling tilt angular covariance as:

$$C_X(a,0) = \sigma^2 \exp(-a/\theta_X) \quad (3.5)$$
$$C_Y(a,0) = \sigma^2 \exp(-a/\theta_Y) \quad (3.6)$$

where σ^2 is the tilt variance, *tvar*, and θ_X and θ_Y are the covariance scales for X-tilt (*thetaCovX*) and Y-tilt (*thetaCovY*), respectively.

3.4.2.38 TiltCovariance

Syntax

TCov = TiltCovariance(*alpha, D, d, Atm, [C1], [C2]*)

```
SCALING_CODE_API int TiltCovariance(char** errorChain,
    const double* apertureDiam, const int nDiam, const double* pathSeps,
    const int nScreens, const int nSeps, const int nGeoms,
    const double* cn2, const double* z, const double* dz,
    const double* slantRange, const bool fullCov, const char c1,
    const char c2, double* tiltCov)
```

TiltCovariance computes the angular-equivalent Zernike tilt covariance. The Matlab inputs are:

s [matrix]	Separation of beam paths along propagation path (m) Resolved for the P axis and T axis, as calculated from PathDisp. Column 1 is separation in P axis, col 2 is separation in T axis.
D [scalar]	Tx/Rx Aperture diameter (m).
C1 [string]	Designation of tilt component 1 ('P' or 'T').
C2 [string]	Designation of tilt component 2 ('P' or 'T').
Atm [struct]	Atmospheric modeling structure from AtmStruct.

The output is *TCov*, the tilt covariance in rad^2.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
apertureDiam [in]	Aperture diameter (m); must be length *nDiam*.
nDiam [in]	Number of apertures; must be 1 or *nEvals* where *nEvals* = max(*nDiam*, *nSeps*, *nGeoms*).
pathSeps [in]	Vector separation at each point z along the path (m); must be 2 * *nScreens* * *nSeps*.
nScreens [in]	Number of phase screens specified for each input.
nSeps [in]	Number of separations to evaluate; must be 1 or *nEvals*.
nGeoms [in]	Number of geometries evaluated; must be 1 or *nEvals*.
cn2 [in]	Cn2 values for the phase screens; must be length *nScreens* * *nGeoms*.
z [in]	Phase screen distances from transmitter (m); must be length *nScreens* * *nGeoms*.
dz [in]	Screen thickness for each phase screen; must be length *nScreens* * *nGeoms*.
slantRange [in]	Slant range (m); must be length *nGeoms*.
fullCov [in]	TRUE for full covariance matrix, FALSE otherwise.
c1 [in]	Designation of tilt component 1 ('P' or 'T'); ignored if *fullCov* = true.
c2 [in]	Designation of tilt component 2 ('P' or 'T'); ignored if *fullCov* = true.
tiltCov [out]	Tilt covariance (rad^2); Allocate to 1 * *nEvals* OR 4 * *nEvals* (for full covariance matrix).

3.4.2.39 TiltCovariancePathWeight

Syntax

W = TiltCovariancePathWeight(*x, sigma, C1, C2*)

```
SCALING_CODE_API int TiltCovariancePathWeight(char** errorChain,
     const double* x, const int nScreens, const int nScreenEvals,
     const double* sigmaIn, const int nSeps, const char c1, const char c2,
     double* pathWeight)
```

This function computes the weighting of C_n^2 (turbulence strength) over propagation path for Zernike-tilt covariance for the specified positions *x* normalized to the path length (0-1), and the accompanying aperture diameter normalized beam separation vector sigma. The Matlab inputs are:

x [vector]	Path position normalized to path length.
sigma [vector]	Separation of beam paths at each normalized point *x* along propagation path, normalized to the Tx/Rx aperture diameter.
C1 [string]	Designation of tilt component 1 ('P' or 'T').
C2 [string]	Designation of tilt component 2 ('P' or 'T').

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Normalized phase screen distances from transmitter (m); must be length *nScreens* * *nScreenEvals*.
nScreens [in]	Number of phase screens specified for each input.
nScreenEvals [in]	Number different normalized position vectors; must be 1 or *nSeps*.
sigmaIn [in]	Normalized (by apDiam) separation at each point x; must be 2 * *nScreens* * *nSeps*.
nSeps [in]	Number of sepoerations evaluated.
c1 [in]	Designation of tilt component 1 ('P' or 'T').
c2 [in]	Designation of tilt component 2 ('P' or 'T').
pathWeight [out]	Path weighting evaluated at normalized positions x; Allocate to *nScreens* * *nSeps*.

Path weighting is computed for covariance of *C1* component with *C2* component. The output is:

W [vector]	Path weighting evaluated at normalized positions *x*.

3.4.2.40 TiltTempCovScale

Syntax

[TauX, TauY, tvar] = TiltTempCovScale(*lambda, D, G, Atm, [DispCov]*)

```
SCALING_CODE_API int TiltTempCovScale(char** errorChain,
    const double lambda, const double D, const double* posPlatEcf,
    const double* posTargEcf, const double* velPlatEcf,
    const double* velTargEcf, const double* re, const int nRE,
    const int nScreens, const double* z, const double* dz,
    const double* cn2, const double* windHeading,
    const double* horzWindSpeed, const double* vertWindSpeed,
    double* thetaCovX, double* thetaCovY, double* tvar, double* aOut,
    double* tcovXOut, double* tcovYOut)
```

This function computes the temporal covariance scale lengths for modeling Kolmogorov tilt fluctuations as a Markov random process in time. The Matlab inputs are:

lambda [scalar]	Wavelength for propagation (m).
D [scalar]	Diameter of receive/transmit aperture (m).
G [struct]	Geometry structure from GeomStruct.
Atm [struct]	Atmospheric modeling structure from AtmStruct.
DispCov [scalar]	(Optional) Flag (1/0) for covariance and fit plot.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength (m).
D [in]	Aperture diameter (m).
posPlatEcf [in]	ECF position vector(s) for platform (m); must be length 3. This function currently only works for a single geometry.
posTargEcf [in]	ECF position vector(s) for target (m); must be length 3.
velPlatEcf [in]	ECF velocity vector(s) for platform (m); must be length 3.
velTargEcf [in]	ECF velocity vector(s) for target (m); must be length 3.
re [in]	Specify the earth radius belonging to each geometry; must be length *nRE*.

nRE [in]	Specify 1 for spherical earth with radius *re* or 2 for oblate earth; must be 1 or 2.
nScreens [in]	Number of phase screens specified for each geometry.
z [in]	Screen distances from transmitter (m); must be length *nScreens*.
dz [in]	Screen thickness for each phase screen; must be length *nScreens*.
cn2 [in]	Cn2 values for each phase screen; must be length *nScreens*.
windHeading [in]	Wind heading for each phase screen (deg); must be length *nScreens*.
horzWindSpeed [in]	Horizontal wind speed for each phase screen (m/s); must be length *nScreens*.
vertWindSpeed [in]	Vertical wind speed for each phase screen (m/s); must be length *nScreens*.
TauX [out]	Temporal covariance scale for X-tilt (rad).
TauY [out]	Temporal covariance scale for Y-tilt (rad).
tvar [out]	Variance of tilt fluctuations (rad^2).
aOut [out]	Time delay vector for info display (s); if not NULL, must be allocated to length 20.
tcovXOut [out]	Tilt covariance in X for info/display (rad); if not NULL, must be allocated to length 20.
tcovYOut [out]	Tilt covariance in Y for info/display (rad); if not NULL, must be allocated to length 20.

The Matlab outputs are

TauX [scalar]	Temporal covariance scale for X-tilt (s).
TauY [scalar]	Temporal covariance scale for Y-tilt (s).
tvar [scalar]	Variance of tilt fluctuations (rad^2).

Returns temporal scale lengths and tilt variance quantities for modeling tilt temporal covariance as:

$$C_X(t,0) = \sigma^2 \exp(-t/\tau_x) \quad (3.7)$$
$$C_Y(t,0) = \sigma^2 \exp(-t/\tau_y) \quad (3.8)$$

where σ^2 is the tilt variance, *tvar*, and τ_x and τ_y are the temporal covariance scales for X-tilt (*TauX*) and Y-tilt (*TauY*), respectively.

3.4.2.41 TiltVariance

Syntax

```
[T02, ratio] = TiltVariance([PropDir], alpha, TxD, [RxD], Atm, ...
    [fmin], [L0], [TOL])
```

This function calculates tilt variance in a limited band and the *ratio* of the band-limited tilt variance to the full band tilt variance. The inputs are

PropDir [string]	(Optional) Propagation direction. 'BACKWARD' (default), 'FORWARD', 'BACKWARD-AP', or 'FORWARD-AP'.
alpha [scalar]	Turbulence multiplier.
TxD [scalar]	Transmit aperture diameter (m).
RxD [scalar]	(Optional) Receive aperture diameter (m). Defaults to zero (focused beam).
Atm [struct]	Discrete atmospheric structure from AtmStruct with turbulence profile.
fmin [scalar]	(Optional) Minimum spatial frequency present (1/m). Defaults to zero.
L0 [scalar]	(Optional) Outer scale of turbulence. Defaults to inf unless contained in *Atm*.
TOL [scalar]	(Optional) Relative tolerance for quad. Default is 1e-6.

The input *PropDir* sets the direction of propagation and indicates at which point tilt is to be measured. 'BACKWARD' and 'FORWARD' calculate tilt on a focal plane. 'BACKWARD-AP' and 'FORWARD-AP' calculate tilt in an aperture plane, for example in relay mirror uplink modeling.

The full tilt variance for the von Kármen spectral model is given by [23]

$$T_0^2 = 0.2073\, k_0^2 \int dz\, C_n^2(h(z)) \int d\vec{\kappa}\, (\kappa^2 + \kappa_0^2)^{-11/6} \left(\frac{8\lambda}{\pi D_p}\right)^2 \left[\frac{J_2(\kappa D(z)/2)}{\kappa D(z)/2}\right]^2$$

where k_0 is the wave number $(2\pi/\lambda)$, $C_n^2(h(z))$ is the refractive index structure coefficient which varies with altitude along the path, and κ_0 is $2\pi/L_0$ where L_0 is the outer scale. To allow for beam formatting, the diameter in the Bessel function and it's denominator varies with position along the path. The diameter outside the Bessel function is a diffraction term and has been set to the transmit aperture diameter, D_p. Over the full frequency band, this yields open loop tilt variance given by

$$\begin{aligned} T_0^2 \;=\; & \frac{6.08}{D_p^2} \int dz\, C_n^2(h(z)) D(z)^{5/3} \left\{ {}_2F_3\left[\frac{7}{3}, \frac{11}{6}; \frac{5}{6}, \frac{29}{6}, \frac{17}{6}; \left(\frac{\pi D(z)}{L_0}\right)^2\right] \right. \\ & \left. -1.42 \frac{D(z)}{L_0}^{1/3} {}_2F_3\left[\frac{5}{2}, 2; \frac{7}{6}, 5, 3; \left(\frac{\pi D(z)}{L_0}\right)^2\right] \right\} \end{aligned}$$

To derive tilt variance in a limited band, make the following substitutions

$$\begin{aligned} d\vec{\kappa} &= \kappa\, d\kappa\, d\theta \\ \kappa &= \frac{2\pi}{D(z)} x \end{aligned}$$

Then the tilt in a limited band is given by

$$\hat{T}_0^2 = \frac{1.58}{D_p^2} \int dz\, C_n^2(h(z))\, D(z)^{5/3} \int_{D(z)f_{min}}^{\infty} \frac{dx}{x} J_2^2(\pi x) \left[x^2 + \left(\frac{D(z)}{L_0} \right)^2 \right]^{-11/6}$$

In wave optics simulations, phase screens generated using an FFT method generally exhibit a deficiency in low spatial frequencies content. With closed-loop tilt compensation, this is not generally a concern as most low frequency components are removed with high enough bandwidth. However, in open loop (no compensation), there can be a significant difference between tilt seen from simulation and expected tilt from theory. The minimum frequency, f_{min}, for a wave optics simulation is the reciprocal of the phase screen dimension. Example 3.4.26 illustrates use of tilt variance to interpret results from wave optics simulation.

Example 3.4.26 *Example calculation of tilt variance in a limited frequency band.*

```
>> Atm = AtmStruct(3000,10,5000,10,'Cn2','HV57');
>> [T02,ratio] = TiltVariance(1,0.5,Atm,1/(512*0.003))

T02 =

  4.6699e-013


ratio =

    0.3594
```

Tilt variance expected from wave optics for a transmit aperture of 50 cm, focused on target with phase screens of about 1.5 m across. The single-axis jitter expected from the simulation is $\sqrt{T02/2} = 0.48$ μrad, which is 60% ($\sqrt{ratio}$) of that expected from theory.

3.4.2.42 TRANSMISSION

Syntax

[ExtTrans, AbsTrans, ScatTrans, TotTrans] = TRANSMISSION(*Atm*)

```
SCALING_CODE_API int TRANSMISSION(char** errorChain,
     const double* attenCoeff, const double* propLenDiff,
     const int nInputPaths, const int nCoeffsPerPath,
     double* transmission)
```

This function computes atmospheric transmission given an input atmospheric model structure from ATMSTRUCT. It returns the transmission given extinction model, absorption model, and scattering model separately as fields in *Atm*. The Matlab outputs are:

ExtTrans [scalar]	Transmission for specified extinction model, set equal to *TotTrans* if Ext model is not specified.
AbsTrans [scalar]	Transmission for absorption losses only.
ScatTrans [scalar]	Transmission for scattering losses only.
TotTrans [scalar]	Total transmission including losses due to specified absorption and scattering.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages - deepest error is first.
attenCoeff [in]	Attenuation coefficient for absorption, scattering, or extinction (1/m).
propLenDiff [in]	Discrete distances over which to apply each attenuation factor; often this is the phase screen thickness (m).
nInputPaths [in]	Multiple paths can be input to compute transmission over; sequential elements belong to the same path.
nCoeffsPerPath [in]	Number of coefficients and path lengths to integrate over to produce path transmission.
transmission [out]	Transmission for each input path; can be multiplied by laser intensity at aperture to get intensity at target.

Transmission T is calculated as

$$T = e^{-\sum \alpha(z)dz}$$

If both absorption and scattering models are specified, and if the transmission calculated from the extinction model does not equal the transmission calculated from the sum of the absorption model and scattering model, then the function will issue a warning. If an extinction model is not specified, then *ExtTrans* will be set equal to *TotTrans*.

3.4.2.43 TurbConst

Syntax

const = TurbConst(*approx*)

```
SCALING_CODE_API int TurbConst(char** errorChain,
    const double* approx, const int nApprox, double* constValues)
```

This function returns the exact value of constants that appear frequently in turbulence calculations given the approximate value often written in equations. Below is a list of the approximate values and the equations used to calculate the exact values.

Approx. Value	=	Equation for Exact Value	Reference
0.033	=	$5\sqrt{3}\,\frac{\Gamma\left(\frac{2}{3}\right)}{36\pi^2}$	[23]
6.88	=	$2\left(24\frac{\Gamma(1.2)}{5}\right)^{5/6}$	[29]
2.91	=	$2\sqrt{\pi}\,\frac{\Gamma\left(\frac{1}{6}\right)}{5\Gamma\left(\frac{2}{3}\right)}$	[33]
0.563	=	$\frac{\sqrt{2\pi}\left(3-\sqrt{3}\right)}{12\cdot 2^{1/3}}\,\Gamma\left(\frac{1}{3}\right)$	[33]
1.23	=	$\sqrt{2\pi}(3-\sqrt{3})/12/2^{1/3}\Gamma\left(\frac{1}{3}\right)24/11$	
0.023	=	$\frac{25\Gamma\left(\frac{5}{6}\right)}{36\pi^{8/3}\Gamma\left(\frac{1}{6}\right)}\left[\frac{24}{5}\Gamma(1.2)\right]^{5/6}$	[33]

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
approx [in]	Approximate values of turbulence constants; constants supported are 6.88, 0.033, 2.91, 0.563, 1.23, and 0.023.
nApprox [in]	Length of input and output vectors.
constValues [out]	More precise values of turbulence constants.

3.4.2.44 TylerFreq

Syntax

fT = TylerFreq(*alpha, lambda, D, S, Atm*)

```
SCALING_CODE_API int TylerFreq(char** errorChain,
     const double* lambda, const double* d, const double* cn2,
     const double* windVelocity, const double* totalPathLength,
     const double* screenSpacing, const int nPaths,
     const int nScreensPerPath, double* freq)
```

This function computes the Tyler frequency for propagation given turbulence assumptions, engagement parameters, and wavelength. It returns *NaN* if the propagation path intersects earth's surface. The Matlab inputs are:

alpha [vector]	Multiplier on turbulence model.

lambda [scalar]	Wavelength of laser (m).
D [scalar]	Aperture diameter (m).
S [struct/list]	Engagement parameters. Must be a structure from EngagementStruct or GeomStruct.
Atm [struct]	Atmospheric modeling parameter structure from AtmStruct. *Atm* must have wind profile/heading if calculation with wind is desired.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength of laser (m); must be length *nPaths*.
d [in]	Aperture diameter (m); must be length *nPaths*.
cn2 [in]	Cn2 values for the phase screens; must be length *nScreensPerPath* x *nPaths* where each screen's turbulence value is sequential.
windVelocity [in]	Line-of-sight velocity transverse to the propagation direction for platform motion, target motion, and natural wind; must be length *nScreensPerPath* x *nPaths* where each screen's velocity is sequential.
totalPathLength [in]	Slant range; must be length *nPaths*.
screenSpacing [in]	Screen thickness for each phase screen; must be length *nScreensPerPath* x *nPaths* where each screen's spacing is sequential.
nPaths [in]	Number of atmospheric paths to evaluate.
nScreensPerPath [in]	Number of phase screens in each path being evaluated.
freq [out]	Tyler frequency for target motion only (Hz); should be allocated to length *nPaths*.

Engagement parameters are input as a structure from EngagementStruct as shown in the example below.

Example 3.4.27 (*S* as a structure) *This example shows how to calculate the Tyler frequency by inputting engagement parameters as a structure from* EngagementStruct. *(S is the engagement structure from Example 3.4.9, alpha = 1, lambda = 1.064e-6 m, and D = 0.5 m)*

```
>> Atm = AtmStruct(G,8,'Cn2','HV57','Wind','Bufton','WindHeading','UniformAtm',-90);
>> fT = TylerFreq(alpha,lambda,D,S,Atm)
```

```
fT =

   61.4167
```

Returns `fT` *= NaN if the propagation intersects the surface. If* `Atm` *and* `S` *are not consistent with respect to geometry,* `Atm` *will be updated.*

The output, `fT`, is a vector with length of `alpha` with the value of the Tyler frequency in hertz for target motion only given by [34]

$$f_T = 0.368 D^{-1/6} \lambda^{-1} \left[\int_0^L C_n^2(h(z)) |V(z)|^2 dz \right]^{1/2}$$

where $C_n^2(h(z))$ is the refractive-index structure function coefficient at the beam altitude $h(z)$, which is a function of the position z along the path, λ is the wavelength of the HEL beam, and the integrals extend from the platform to the target. $V(z)$ is the apparent atmospheric velocity transverse to the path given by TransVelocity.

3.4.2.45 WaveStrFcnPathWeight

Syntax

`W = WaveStrFcnPathWeight(s, l0, L0)`

This function computes weighting of C_n^2 (turbulence strength) over propagation path for the wave structure function. The inputs are

`s` [vector]	Magnitude of the separation of rays along propagation path (m). For plane waves, `s` is constant along the path. For spherical waves, `s` converges along straight lines from its maximum value at the platform pupil to a point in the target plane.
`l0` [vector]	Inner scale value along the path (m).
`L0` [vector]	Outer scale value along the path (m).

The output is

`W` [vector]	Path weighting evaluated along the path.

3.4.2.46 WeightFunctionRef

Syntax

WF = WeightFunctionRef(*z, L, rho, phi, delta*)

```
SCALING_CODE_API int WeightFunctionRef(char** errorChain,
    const double l, const int nInputs, const double* z,
    const double* alpha, const double* phi, const double* delta,
    double* wf)
```

This function calculates the weighting term of C_n^2 for the calculation of the anisoplanatic Strehl ratio. The Matlab inputs are:

z [vector]	Distances from transmitter (m), usually distances of the phase screens from the transmitter.
L [scalar]	Slant range of optical path (m), as from platform to target.
rho [scalar]	Radial polar coordinate in the aperture (m).
phi [scalar]	Angular polar coordinate in the aperture (rad).
delta [scalar]	Separation distance along the optical path (m).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
l [in]	Slant range of optical path (m), as from platform to target.
nInputs [in]	Number of points over which to compute the weighting function.
z [in]	Distances from transmitter (m), usually distances of the phase screens from the transmitter; must be length *nInputs*.
alpha [in]	Radial polar coordinate in the aperture (m); must be length *nInputs*.
phi [in]	Angular polar coordinate in the aperture (rad); must be length *nInputs*.
delta [in]	Separation distance along the optical path (m); must be length *nInputs*.
wf [out]	Weighting function values; must be length *nInputs*.

The output weighting function, *WF*, is given by [13]

$$\begin{aligned} WF &= \rho(1-z/L)^{5/3} + (\delta)^{5/3} \\ &\quad -0.5(\rho(1-z/L)^2 + 2\rho(1-z/L)(\delta)\cos(\phi) + (\delta)^2)^{5/6} \\ &\quad -0.5(\rho(1-z/L)^2 - 2\rho(1-z/L)(\delta)\cos(\phi) + (\delta)^2)^{5/6} \end{aligned}$$

where ρ and ϕ are the radial and angular polar coordinates, respectively, δ is the separation along the path, z is a vector of distances along the path from zero to L, and L is the total slant range.

3.4.2.47 WeightMat

Syntax

[W] = WeightMat(*dx, [x], [thick]*)

This function computes the weighting matrix (*W*) associated with the discrete profile such that $P = KWC_n^2$ where P is a vector with elements $r_0^{-5/3}$ (Fried Parameter), $\theta_0^{-5/3}$ (isoplanatic angle), σ_χ^2 (Rytov), and M_0 (integrated C_n^2). K is a diagonal matrix with the appropriate scaling constants and C_n^2 is a vector of C_n^2 values. The inputs are

dx [vector/struct]	If vector, NORMALIZED screen thickness values. If structure, turbulence profile information with fields L, z, and dz (as from AtmStruct).
x [vector]	(Optional) NORMALIZED screen locations. If not passed, it will be computed assuming dx spans the path and with screens centered in each slab.
thick [scalar]	(Optional) 1 for thick slab and 0 (default) for thin slab model.

The output *W* is a matrix of dimension 4 by length *dx* (number of screens) where the first row is the weighting vector for r_0, the second row is the weighting vector for θ_0, the third row is the weighting vector for σ_χ , and the fourth row is the weighting vector for M_0. For a thin slab these are given by

$$\begin{aligned} W_r &= (1-x)^{5/3}dx \\ W_t &= x^{5/3}dx \\ W_x &= [x(1-x)]^{5/6}dx \\ W_m &= dx \end{aligned}$$

For a thick slab the weighting vectors are calculated by integrating the above expressions from $x - dx/2$ to $x + dx/2$.

3.4.2.48 WindDirCos

Syntax

[w_TP w_TT] = WindDirCos(*[x], G, Atm, [OutType]*)

```
SCALING_CODE_API int WindDirCos(char** errorChain,
    const double* windHeading, const double* horzWindSpeed,
    const double* vertWindSpeed, const double* screenPositions,
    const double* pathLength, const double* posPlatEcf,
    const double* posTargEcf, const double* velTargEcf,
    const double* earthRadius, const int nEarthRadii, const int nScreens,
    const int nGeoms, double* windTargetParallel,
    double* windTargetTransverse)
```

This function returns the direction cosines for natural wind transverse to the direction of propagation. The Matlab inputs are:

x [vector]	(Optional) Position along path, normalized to slant range (0-1). Not required if *Atm* is a discrete structure from AtmStruct.
G [struct]	Geometry structure from GeomStruct - can be an array of structures.
Atm [struct]	Atmospheric structure from AtmStruct - can be an array of structures.
OutType [string]	(Optional) 'T' for T-axis direction cosine and 'P' for P-axis direction cosine - only used if a single output is desired.

The outputs are the wind direction cosines of the phase screens specified by *x* as follows:

w_TP [vector]	Angle cosine of wind, in target "parallel" direction.
w_TT [vector]	Angle cosine of wind, in target "transverse" direction.

The input structures, *G* and *Atm*, can be a single structure or an array of structures. If both are arrays, they must be the same length. If *Atm* is an array and *G* is scalar, the function assumes that all of *Atm* have the same geometry as *G*. If *G* is an array and *Atm* is scalar, ChangeAtm will be used to create an array of structures with geometries of *G*.

Given the ECF wind vector $\mathbf{W}$ and the ECF vectors for position ($\mathbf{R}$) and velocity ($\mathbf{V}$) of both platform (P) and target (T), the wind direction cosines are computed as [1]:

$$\texttt{w\_TP} = \frac{\mathbf{W} \bullet \hat{\rho}_{\mathbf{P}}}{|\mathbf{W}|}$$

$$\texttt{w\_TT} = \frac{\mathbf{W} \bullet \hat{\rho}_{\mathbf{T}}}{|\mathbf{W}|}$$

where

$$\hat{\rho}_P = \frac{1}{\mathtt{V\_TTP}}\left[\mathbf{V}_T - \frac{[(\mathbf{R}_T - \mathbf{R}_P) \bullet \mathbf{V}_T]\,(\mathbf{R}_T - \mathbf{R}_P)}{\mathtt{L}^2}\right],$$

$$\hat{\rho}_T = \frac{\hat{\rho}_P \times (\mathbf{R}_T - \mathbf{R}_P)}{\mathtt{L}},$$

and $\mathtt{L} = |\mathbf{R}_T - \mathbf{R}_P|$ is the slant range from platform to target.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
windHeading [in]	Natural wind heading for the phase screens (deg from North); must be *nScreens* x *nGeoms* with the value for each screen at consecutive addresses.
horzWindSpeed [in]	Natural wind speed values for the phase screens (m/s); must be *nScreens* x *nGeoms*.
vertWindSpeed [in]	Natural wind speed values orthogonal to North-South plane and in "up" direction (m/s); must be *nScreens* x *nGeoms*.
screenPositions [in]	Phase screen distances from transmitter (m); must be *nScreens* x *nGeoms*.
pathLength [in]	Slant range from platform to target (m); must be length *nGeoms*.
posPlatEcf [in]	ECF position vector(s) for platform (m); must be length 3 x *nGeoms* with each coordinate belonging to one geometry specification consecutive.
posTargEcf [in]	ECF position vector(s) for target (m); must be length 3 x *nGeoms*.
velTargEcf [in]	ECF velocity vector(s) for target (m); must be length 3 x *nGeoms*.
earthRadius [in]	Specify the earth radius belonging to each geometry; must be *nEarthRadii* x *nGeoms* with each radius belonging to one geomeyry specification consecutive.
nEarthRadii [in]	Specify 1 for spherical earth with that radius or 2 for oblate earth; must be 1 or 2.
nScreens [in]	Number of phase screens specified for each geometry.
nGeoms [in]	Number of geometries over which to compute wind direction cosines.
windTargetParallel [out]	Angle cosine of wind, in target "parallel" direction; will be length *nScreens* x *nGeoms* with the value for each screen at consecutive addresses.
windTargetTransverse [out]	Angle cosine of wind, in target "transverse" direction; will be length nScreens x *nGeoms*.

3.4.2.49 wtBLS900

Syntax

wt = wtBLS900(*x, dx*)

This function returns a path weighting function of C_n^2 for the BLS-900 scintillometer. The weighting function would be used to compute a path-weighted value of C_n^2 as follows:

```
Cn2Wt = sum(Cn2.*wt)./sum(wt).
```

The inputs are

x [matrix]	Normalized bin center locations.
dx [matrix]	Normalized bin widths.

The output is

wt [matrix]	BLS-900 path weighting function.

3.4.2.50 ZernikeCoeffCov

Syntax

ZCov = ZernikeCoeffCov(*alpha, D, s, i, j, Atm, [Norm]*)

This function computes Zernike coefficient covariance for mode i with mode j in units of m^2 given the following inputs:

alpha [vector]	Turbulence strength multiplier.
D [scalar]	Tx/Rx Aperture diameter (m).
s [matrix]	Separation of beam paths along propagation path (m). Resolved for the P axis and T axis, as calculated from PathDisp. Column 1 is separation in P axis, col 2 is separation in T axis, i.e., d=[d_P, d_T].
i..j [scalars]	Mode numbers of the Zernike coefficients (Noll's ordering).
Atm [struct]	Atmospheric modeling structure from AtmStruct.
Norm [string]	(Optional) Flag to indicate whether covariance should be normalized. If omitted, output is not normalized

The optional input *Norm* determines whether covariance is normalized and output in appropriate units. The following options are available:

- 'N': normalized to $(D/r_0)^{5/3}$ for spherical-wave r0, phase rad^2
- 'A': converted to angular rad^2, natural for i,j = 2,3.

The output is

ZCov [vector/matrix]	Zernike Coefficient covariance (rad^2) or normalized to $(D/\mathrm{r0})^{5/3}$.

3.4.2.51 ZernikeCoefficient

Syntax

c_out = ZernikeCoefficient(*p, c, r, t*)

This function computes Zernike coefficients for given Zernike modes when provided with the phase screen and radial coordinates. The inputs are

p [matrix]	Phase screen.
c [vector]	Zernike modes for which coefficients are to be computed.
r [matrix]	Radius values.
t [matrix]	Theta values.

The output, *c_out*, is

c_out [vector]	Zernike coefficients corresponding to each input mode.

3.4.2.52 ZernikeCoord

Syntax

[rho, theta, Mask] = ZernikeCoord(*M, [N], [CenRowCol], [rad]*)

This function returns radial and azimuthal coordinate system for use with Zernike polynomial functions. Also returns a mask to be applied to the coordinate system for the aperture as well as the rectangular coordinates. The X coordinate is down the rows, which the Y coordinate is across the columns. The inputs are

M [scalar]	Number of rows in output coordinate system.
N [scalar]	(Optional) Number of columns in output coordinate system. Defaults to *M* if not specified.
CenRowCol [vector]	(Optional) [row col] of center of coordinate system. Defaults to floor(*N*/2)+1.
rad [scalar]	(Optional) Radius output coordinate system (pixels). Defaults to largest circle possible in grid space.

The outputs are

rho [matrix]	Radius values.
theta [matrix]	Theta values.
Mask [matrix]	1/0 of values within circular aperture.
xx [matrix]	Grid of x-coordinate values.
yy [matrix]	Grid of y-coordinate values.

3.4.2.53 ZernikeCovariancePathWeight

Syntax

W = ZernikeCovariancePathWeight(*x, sigma, i, j, ni, nj, mi, mj*)

```
SCALING_CODE_API int ZernikeCovariancePathWeight(char** errorChain,
     const int nX, const double* x, const double* pathSepP,
     const double* pathSepT, const int i, const int j, const int ni,
     const int nj, const int mi, const int mj, double* w)
```

This function computes weighting of C_n^2 (turbulence strength) over propagation path for covariance of Zernike coefficients for the specified positions *x* normalized to the path length (0-1), and the accompanying aperture diameter normalized beam separation vector *sigma*. Path weighting is computed for the covariance of the ith and jth Zernike coefficients.

x [vector]	Path position normalized to the path length.
sigma [vector]	Separation of beam paths at each normalized point *x* along propagation path, normalized to the Tx/Rx aperture diameter.
i..j [scalars]	Noll orders of the Zernike coefficients.
ni..nj [scalars]	Radial orders corresponding to ith and jth Noll orders.
mi..mj [scalars]	Azimuthal orders corresponding to ith and jth Noll orders.

The output is the path weighting used by ZernikeCoeffCov

W [vector]	Path weighting evaluated at normalized positions *x*.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nX [in]	Length of input and output vectors.
x [in]	Path position normalized to the path length, vector length of *nX*.
pathSepP [in]	Separation of beam paths at each normalized point x along propagation path in the target P-axis, normalized to the Tx / Rx aperture diameter.
pathSepT [in]	Separation of beam paths at each normalized point x along propagation path in the target T-axis, normalized to the Tx / Rx aperture diameter.
i [in]	Noll orders of the Zernike coefficients.
j [in]	Noll orders of the Zernike coefficients.
ni [in]	Radial orders corresponding to ith and jth Noll orders.
nj [in]	Radial orders corresponding to ith and jth Noll orders.
mi [in]	Azimuthal orders corresponding to ith and jth Noll orders.

mj [in]	Azimuthal orders corresponding to ith and jth Noll orders.
w [out]	Path weighting evaluated at normalized positions *x*.

3.4.2.54 ZernikePolynomial

Syntax

zp_out = ZernikePolynomial(*noll_order, rho, theta*)

This function returns the value of the Zernike polynomial of order *noll_order* at normalized radius *rho* and angle *theta*, where $0 \leq$ *rho* ≤ 1 and $0 \leq$ *theta* $\leq 2\pi$. *rho* and *theta* may be scalar or matrix quantities. The inputs are

noll_order [scalar]	Noll order for required Zernike polynomial.
rho [matrix]	Radius values.
theta [matrix]	Theta values.

The output, *zp_out*, is

zp_out [vector]	Zernike polynomial values.

3.4.2.55 ZernikeReconstruct

Syntax

img_out = ZernikeReconstruct(*orders, coeffs, r, t*)

This function produces phase screens given Zernike orders to be used and their corresponding magnitudes (coefficients). The inputs are

orders [vector]	Zernike *orders* for which reconstruction is to be computed.
coeffs [vector]	Coefficients for magnitude of Zernike orders.
r [matrix]	Radius values.
t [matrix]	Theta values.

The output, *img_out*, is

img_out [matrix]	Reconstructed image.

3.4.2.56 ZTiltPSD

Syntax

[XPSD, YPSD, F, Xvar, Yvar, Tvar] = ZTiltPSD(*alpha, D, LogFMin, ... LogFMax, NF, S, Atm, [NZ]*)

```
SCALING_CODE_API int ZTiltPSD(char** errorChain,
    const double apertureDiam, const double logFmin,
    const double logFmax, const int nF, const double* cn2,
    const double* v, const double* vx, const double* vy, const double* z,
    const double* dz, const bool srFlag, const int nScreens,
    const double slantRange, double* xPSD, double* yPSD, double* f,
    double xVar[1], double yVar[1], double tVar[1])
```

This routine computes the PSD of Zernike tilt phase aberrations caused by atmospheric turbulence. The temporal covariance of the phase is averaged over the aperture and Fourier transformed to give a single PSD for each direction. The function returns a warning if Simpson's Rule integration does not include at least 85% of the theoretical variance. The Matlab inputs are:

alpha [scalar]	Multiplier on turbulence model.
D [scalar]	Aperture Diameter (m).
LogFMin [scalar]	log10 of the min frequency for PSD calculation.
LogFMax [scalar]	log10 of the max frequency for PSD calculation.
NF [scalar]	Number of frequencies per decade (logarithmically spaced) for PSD calculation.
S [struct]	Engagement parameters. Can be a structure from EngagementStruct or GeomStruct. If a structure from GeomStruct is passed in as *S*, the engagement structure will be computed.
Atm [struct]	Atmospheric modeling parameter structure from AtmStruct. Can be a continuous or a discrete model.
NZ [scalar]	Number of evaluation points for the numerical integral (using Simpson's rule integration) with range. If *Atm* is a structure defining a discrete atmospheric model, *NZ* is ignored.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
apertureDiam [in]	Aperture diameter (m); must be length 1.
logFmin [in]	log10 of the minimum frequency for PSD calculation.
logFmax [in]	log10 of the maximum frequency for PSD calculation.
nF [in]	Number of frequencies (logarithmically spaced) for PSD calculation.
cn2 [in]	Cn2 values for the phase screens; must be length *nScreens*.
v [in]	Magnitude of transverse velocity values for the phase screens (m/s); must be length *nScreens*.
vx [in]	X-transverse velocity values for the phase screens (m/s); must be length *nScreens*.
vy [in]	Y-transverse velocity values for the phase screens (m/s); must be length *nScreens*.
z [in]	Phase screen distances from transmitter (m); must be length *nScreens*.
dz [in]	Screen thickness for each phase screen; must be length *nScreens*.
srFlag [in]	Flag indicating use of Simpson's rule integration. If TRUE, Simpson's rule is used.
nScreens [in]	Number of phase screens specified - must be greater than 3 and even if *srFlag* is TRUE.
slantRange [in]	Slant range (m); must be length 1.
xPSD [out]	x-PSD of Zernike tilt aberrations at the frequencies in the input vector f (rad^2/Hz); length *nF*.
yPSD [out]	y-PSD of Zernike tilt aberrations at the frequencies in the input vector f (rad^2/Hz); length *nF*.
f [out]	Frequencies for which the PSD is calculated (Hz); length *nF*.
xVar [out]	X-Tilt variance as integrated from the computed PSD (rad^2).
yVar out]	Y-Tilt variance as integrated from the computed PSD (rad^2).
tVar [out]	Theoretical higher order variance (rad^2).

The PSD calculated in this routine is one half of a two-sided PSD. The integral under the PSD as computed must be doubled to account for power at negative frequencies. There is a convenient way to calculate the total power when the PSD is generated for a vector of logarithmically spaced frequencies. The outputs are

XPSD [vector]	PSD of X-Tilt aberrations at the frequencies in the input vector f. The PSD is obtained by averaging the pointwise PSDs of the phase Tilt. The units are rad^2/Hz.
YPSD [vector]	PSD of Y-Tilt aberrations at the frequencies in the input vector f. The PSD is obtained by averaging the pointwise PSDs of the phase Tilt. The units are rad^2/Hz.
F [vector]	Frequencies for which the PSD is calculated (1/s).
Xvar [scalar]	X-Tilt variance as integrated from the computed PSD rad^2.
Yvar [scalar]	Y-Tilt variance as integrated from the computed PSD rad^2.
Tvar [scalar]	Theoretical tilt variance rad^2. If the spacing of the base 10 logarithm is ***dlogf***, then the total variance is given by $\text{TVar} = 2\text{dlogf}\ln(10) \times \sum(f \times \omega)$.

Example 3.4.28 (PSD of Zernike tilt phase aberrations) *This example shows how to compute the PSD of Zernike tilt phase aberrations caused by atmospheric turbulence with a turbulence multiplier of one and an aperture diameter of 0.5 m.*

```
>> Atm = AtmStruct(S.G,8,'Cn2','HV57','Wind','Bufton');
>> [XPSD,YPSD,F,Xvar,Yvar,Tvar] = ZTiltPSD(1,0.5,-2,3,30,S,Atm)

Xvar =

  8.1864e-012


Yvar =

  8.4713e-012


Tvar =

  9.0305e-012
```

The outputs ***XPSD***, ***YPSD***, *and* ***F*** *have been plotted in Figure 3.14.* ***S*** *is the engagement structure from Example 3.4.9.*

`ZTiltPSD(...)` with no outputs generates a loglog plot of the PSD as in Figure 3.14.

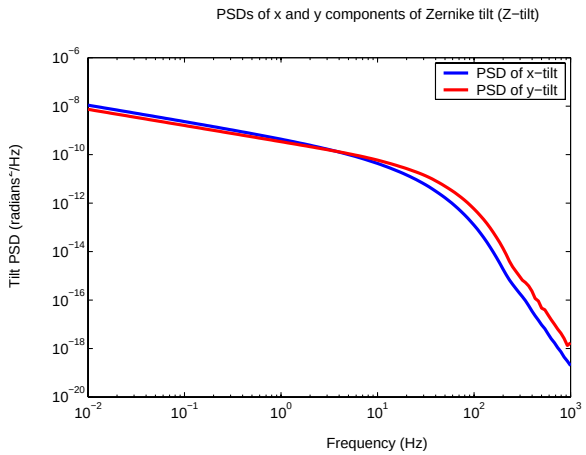


Figure 3.14: PSD from ZTILTPSD.

3.4.3 Atmospheric Model Functions

This section contains functions for use in atmospheric modeling. There are functions which generate profiles of the refractive index structure constant, C_n^2 , and absorption and scattering coefficients, temperature, density and other parameters.

3.4.3.1 AFGLAMOS

Syntax

Cn2 = AFGLAMOS(*h*)

```
SCALING_CODE_API int AFGLAMOS(char** errorChain, const double* h,
    const int nH, double* cn2)
```

This function returns the value of the AFGL AMOS night turbulence profile given altitude *h* above sea level in meters [35]. The model is only valid above 3052 m and returns *NaN* below this altitude. C_n^2 is given by

$$\log_{10}(C_n^2) = \begin{cases} -12.412 - 0.4713h - 0.0906h^2 & 3.052 \le h \le 5.2 \\ -17.1273 - 0.0301h - 0.001h^2 + 0.5061 \exp\left(-0.5\left(\frac{h-15.0866}{3.2977}\right)^2\right) & h > 5.2 \end{cases}$$

where h is in kilometers. The only input is *h* and the output *Cn2* is a vector the size of *h* with the values of C_n^2 with units of $\mathrm{m}^{-2/3}$. Figure 3.15 shows C_n^2 as a function of altitude as calculated by AFGL AMOS.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

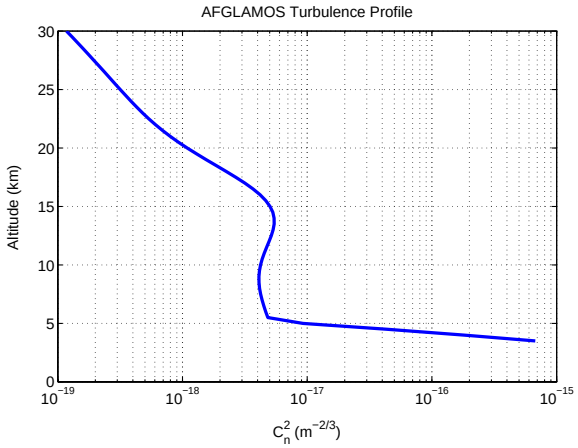


Figure 3.15: Turbulence profile calculated using AFGL AMOS night model.

3.4.3.2 AFRLDE_ATMOS

Syntax

Model = AFRLDE_ATMOS(*h, [ModelType], [lambda]*)

```
SCALING_CODE_API int AFRLDE_ATMOS(char** errorChain, const double* h,
    const int nH, const char* modelType, const char* lambda,
    double* modelValue)
```

This function returns the value of the AFRL/DE model for absorption and scattering coefficients, and temperature as a function of altitude *h* in meters for a given wavelength. Models were developed for 0 km to 11 km altitude, but can be extended to higher altitudes based on an exponential fit. The wavelengths supported are 1.064 μm and 1.31521 μm. The coefficients are modeled as [36].

$$
\begin{aligned}
&\text{For } 1.31521\ \mu m \\
&\qquad \alpha_{mol}(h) &&= 2.2 \times 10^{-5} \exp(-h/1396.3) \\
&\qquad \alpha_{aero}(h) &&= 1.0 \times 10^{-5} \exp(-h/3000) \\
&\qquad \alpha_{scat}(h) &&= 5.0 \times 10^{-5} \exp(-h/3000) \\
&\text{For } 1.064\ \mu m \\
&\qquad \alpha_{mol}(h) &&= 3.627 \times 10^{-9} \exp(-h/2862.9) \\
&\qquad \alpha_{aero}(h) &&= 1.1 \times 10^{-5} \exp(-h/3000) \\
&\qquad \alpha_{scat}(h) &&= 5.5 \times 10^{-5} \exp(-h/3000)
\end{aligned}
$$

The Matlab inputs are:

h [vector]	Altitude of laser along propagation path (m).
ModelType [string]	(Optional, defaults to 'A') Identifier for the type of model desired. Supports 'A' (absorption), 'S' (scattering), 'E' (total extinction), and 'T' (temperature) model options.
lambda [string]	(Optional, defaults to '1315') Identifier for desired laser wavelength. Supports '1064' for YAG, '1315' for COIL.

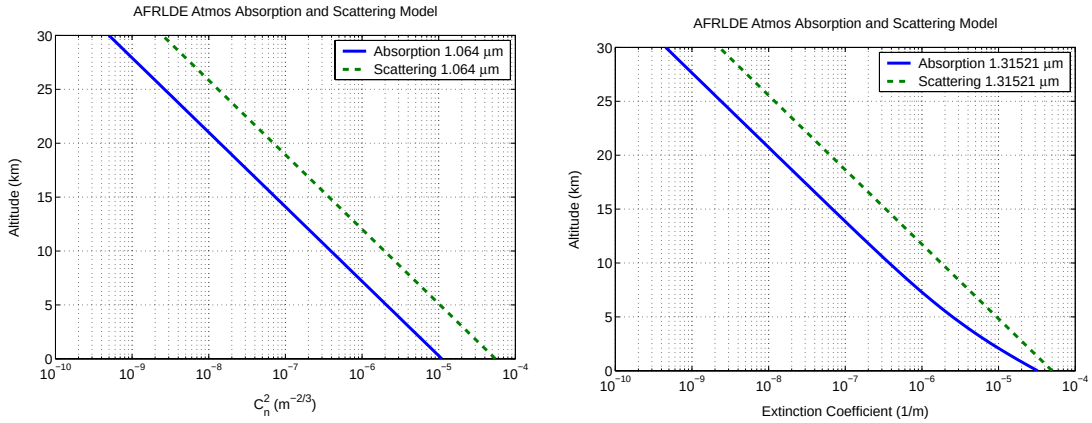


Figure 3.16: Absorption and scattering coefficients with altitude from AFRLDE_Atmos for two different wavelengths, (a) 1.064 μ m and (b) 1.31521 μ m.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
modelType [in]	(NULL OK) Identifier for the type of model desired; supports "a" (absorption), "s"(scattering), "e" (total extinction), "t" temperature; default is "a."
lambda [in]	(NULL OK) Identifier for desired laser wavelength; supports "1064" for YAG, "1315" for COIL (1.31521 mocrons); defaults to "1315."
modelValue [out]	Atmospheric model value for desired modelType (i.e. absorption, scattering, or extinction) (1/m); must be length *nH*.

The function uses the logarithmic fit above to calculate absorption and scattering and uses US_Standard76 to calculate temperature. The output, *Model*, is a vector the size of *h* with the absorption, scattering or extinction coefficients (m^{-1}) or temperature (K) at each point. Figure 3.16 shows the absorption and scattering coefficients as calculated by AFRLDE_Atmos for two different wavelengths, 1.064 and 1.31521 μm.

3.4.3.3 AIRS

Syntax

```
mData = AIRS(h, x, G, ModelType, fname, [KH_KM_ratio], [Lo], ...
    [mergeMode], [p1], ..., [pN])
```

This function returns model data interpolated from Atmospheric InfraRed Sounder (AIRS) measurements. The inputs are

h [vector]	Altitude above sea level (m).
x [vector]	Normalized screen locations.
G [struct]	Geometry structure as from GeomStruct.
ModelType [string]	Desired model for output. Can be any one of the following (case insensitive): Cn2 - 'C' or 'Cn2', Temp - 'Temp', 'Temperature', or 'T', Press - 'Press', 'Pressure', or 'P', Temperature Lapse Rate - 'dtdz', 'TempLapseRate', 'TemperatureLapseRate', Cloud Height - 'CloudHeight'.
fname [char]	File name for the AIRS data.
KH_KM_ratio [scalar]	(Optional) Diffusivity ratio. Only used for Cn2 calculation. Defaults to 0.01.
Lo [scalar]	(Optional) Outer scale (m). Only used for Cn2 calculation. Defaults to 150.
mergeModel [string]	Model to use for low altitude values where *NaN* is returned from the AIRS data. Will be scaled to return the same value at the lowest altitude which is not *NaN*.
p1,...,pN [arb]	(Optional) Parameters required for *mergeModel*.

Data from AIRS can be downloaded from http://mirador.gsfc.nasa.gov/ using AIRX2SUP as the keyword. Then use processAIRShdf to process the downloaded data to create a lookup table for use with this function. The output is

mData [vector]	Model values at specified altitudes.

3.4.3.4 ApparentGravity

Syntax

AppGrav = ApparentGravity(*h, [Lat]*)

```
SCALING_CODE_API int ApparentGravity(char** errorChain,
    const double* h, const int nH, const double* Lat, const int nLat,
    double* appGrav)
```

This function returns the value of the apparent gravity profile given altitude above sea level in meters and optionally the latitude, ***Lat*** in degrees from -90 to 90. This function will generate the slight changes in gravity with altitude. It is valid from the surface to 86 km above mean sea level.

$$\texttt{AppGrav} = g_0 \left(\frac{r_e}{r_e + h} \right)^2$$

where r_e is the radius of the earth and g_0 is the gravitational constant as a function of latitude θ given by

$$g_0 = 9.780356(1 + 0.0052885 \sin(\theta)^2 - 0.0000059 \sin(2\theta)^2).$$

AppGrav is in units of m s^{-2} [37].

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitudes above ground level (m).
nH [in]	Number of elements in vector ***h***, must be 1 or ***nEvals*** where ***nEvals*** = max(***nH***, ***nLat***).
Lat [in]	(NULL OK) Latitude position in degrees (-90 to 90).
nLat [in]	Number of elements in ***Lat***, must be 1 or ***nEvals***.
appGrav [out]	Apparent gravity (m/s^2) ; Must be length ***nEvals***.

3.4.3.5 AtmModels

Syntax

```
AtmModels
```

AtmModels will display a list of all currently available models for atmospheric propagation. The following is the result of typing `help AtmModels` or `AtmModels` at the MATLAB® command prompt. Currently supported models:

```
Currently supported models:
 ~ Cn2:
    - AFGL AMOS                    Cn2 = AFGLAMOS(h)
    - Clear1Night                  Cn2 = Clear1Night(h)
    - Clear2Night                  Cn2 = Clear2Night(h)
    - DLRmodHV57                   Cn2 = DLRmodHV57(h,meanRefract)
    - dynamicPiecewiseMaritimeAIRS Cn2 = dynamicPiecewiseMaritimeAIRS( ...
                                         h,[Cn2_0],[locale],[ht],[Cn2_ht]);
    - GreenwoodCn2                 Cn2 = GreenwoodCn2(h)
    - Hufnagel-Valley              Cn2 = HufnagelValley(h,V,A)
    - Hufnagel-Valley (5/7)        Cn2 = HV57(h)
    - LEEDRAtmos*                  Cn2 = LEEDRAtmos(h,'C')
    - Maui 3                       Cn2 = Maui3Cn2(h)
    - Modified Hufnagel-Valley     Cn2 = ModHufnagelValley(h)
```

```
    - Random Cn2                Cn2 = RandCn2(h,x,dx,'model',Params)
    - SLC Day                   Cn2 = SLCDay(h)
    - SLC Night                 Cn2 = SLCNight(h)
    - Uniform Cn2               Cn2 = UniformCn2(h,C0)
    - WMSR Cn2                  Cn2 = WSMRCn2(h,ModelType,GroundAlt)
    - NSLOT                     Cn2 = NSLOT_v2(zref,u,tair,tsea, ...
                                            rh,pr,zu,zt,zq,zp,lambda)

~ InnerScale:
    - InnerScale                Lin = InnerScale(h)

~ OuterScale:
    - OuterScale                Lout = OuterScale(h,m)

~ Wind:
    - Bufton                    w = Bufton(h)
    - Clear-2 Wind              w = Clear2Wind(h)
    - LEEDRAtmos*               w = LEEDRAtmos(h,'W')
    - SOR Wind                  w = SORWind(h,season)
    - WSMR Wind                 w = WSMRWind(h)

~ WindHeading:
    - LEEDRAtmos*               wh = LEEDRAtmos(h,'WH')
    - WSMR Wind Heading         wh = WSMRWindHeading(h)

~ Scattering:
    - AFRLDE_Atmos              scat = AFRLDE_Atmos(h,'S',lambda)
    - GADSAtmos*                scat = GADSAtmos(h,'S',lambda,Site)
    - LEEDRAtmos*               scat = LEEDRAtmos(h,'S',lambda)
    - MODFAS                    scat = MODFAS(h,'S',lambda)
    - OceanAtmos*               scat = OceanAtmos(h,'S',lambda,[Lat Lon])

~ Absorption:
    - AFRLDE_Atmos              abs = AFRLDE_Atmos(h,'A',lambda)
    - GADSAtmos*                abs = GADSAtmos(h,'A',lambda,Site)
    - LEEDRAtmos*               abs = LEEDRAtmos(h,'A',lambda)
    - MODFAS                    abs = MODFAS(h,'A',lambda)
    - OceanAtmos*               abs = OceanAtmos(h,'A',lambda,[Lat Lon])

~ Extinction:
    - AFRLDE_Atmos              ext = AFRLDE_Atmos(h,'E',lambda)
    - GADSAtmos*                ext = GADSAtmos(h,'E',lambda,Site)
    - LEEDRAtmos*               ext = LEEDRAtmos(h,'E',lambda)
    - MODFAS                    ext = MODFAS(h,'E',lambda)
    - OceanAtmos*               ext = OceanAtmos(h,'E',lambda,[Lat Lon])

~ Temperature:
    - GADSAtmos*                T = GADSAtmos(h,'T',lambda,Site)
    - LEEDRAtmos*               T = LEEDRAtmos(h,'T')
    - OceanAtmos*               T = OceanAtmos(h,'T',lambda,[Lat Lon])
    - US_Standard76             T = US_Standard76(h,'TEMPERATURE')

~ Pressure:
    - GADSAtmos*                P = GADSAtmos(h,'P',lambda,Site)
```

```
    - LEEDRAtmos*                P = LEEDRAtmos(h,'P')
    - OceanAtmos*                P = OceanAtmos(h,'P',lambda,[Lat Lon])
    - US_Standard76              P = US_Standard76(h,'PRESSURE')

~ Relative Humidity:
    - GADSAtmos*                 RH = GADSAtmos(h,'R',lambda,Site)
    - LEEDRAtmos*                RH = LEEDRAtmos(h,'R')
    - OceanAtmos*                RH = OceanAtmos(h,'R',lambda,[Lat Lon])

~ Dew Point:
    - GADSAtmos*                 Dpt = GADSAtmos(h,'D',lambda,Site)
    - LEEDRAtmos*                Dpt = LEEDRAtmos(h,'D')
    - OceanAtmos*                Dpt = OceanAtmos(h,'D',lambda,[Lat Lon])

~ Density:
    - US_Standard76              D = US_Standard76(h,'DENSITY')

~ General:
    - LogScaling                 Model = LogScaling(h,base,A,B)
                                         [A*base^(-h/B)]
    - LUTAtm                     Model = LUTAtm(h,var,fileName)
    - UniformAtm                 Model = UniformAtm(h, ModelValue)
```

* Indicates model functions which make use of additional data, available only by request.

3.4.3.6 AVERAGEATM

Syntax

Model = AVERAGEATM(*h, dx, G, BaseModel, [p1], ..., [pN]*)

This function computes the average value of the input model over the specified phase screen slabs. Can be used with any model type. The inputs are

h [vector]	Altitude (m).
dx [vector]	Normalized screen thicknesses.
G [struct/list]	Geometry structure as from GEOMSTRUCT or comma-separated list of (..., *hp*, *ht*, *rd*, [*re*], ...).
hp [scalar]	Platform altitude (m).
ht [scalar]	Target altitude (m).
rd [scalar]	Ground range (m).
re [scalar]	(Optional) Earth radius (m).
BaseModel [string]	Atmospheric modeling function on path or evaluation.
p1,...,pN [arb]	(Optional) Parameters required for *BaseModel*.

The output is

Model [vector]	Average value of the input base model over the specified slabs.

Example 3.4.29 has some examples of how to use AverageAtm. For examples of using AverageAtm with BoundaryAtm and TerrainAtm, refer to the example script NestingModelFunctions in the ATMTools/Examples directory.

Example 3.4.29 (AverageAtm) *This example illustrates use of* AverageAtm.

```
>> Atm = AtmStruct;
>> Model = AverageAtm(Atm.h,Atm.dz/Atm.L,G,'MODFAS','A','1315')
```

Computes average absorption coefficient over the slabs specified in *Atm*

```
>> Model = AverageAtm(Atm.h,Atm.dz/Atm.L,G,Atm.Cn2Eval)
```

Computes average Cn2 for the model used in *Atm*

```
>> Atm = AtmStruct(GeomStruct,10,'Cn2','AverageAtm','dx','G','HV57')
>> Atm = AtmStruct(GeomStruct,10,'Cn2','AverageAtm','HV57')
```

Using AverageAtm *as a model function to return model average in* *Atm.Cn2*. *When using* AverageAtm *as a model function input to* AtmStruct, *dx and G need not be specified explicitly, but if they are they must be input as a string which will be evaluated within* AtmStruct *to ensure the correct values.*

3.4.3.7 BoundaryAtm

Syntax

[Model, hh] = BoundaryAtm(*h, hInt, hhInt, BaseModel, [p1], ..., [pN]*)

This function returns model data (C_n^2 , extinction coefficients, temperature, etc.) with a linear scaling of the input altitude for different intervals. The inputs are

h [vector]	Altitude above sea level (m).
hInt [array]	Altitude interval for mapping (m). Can be of size 1 × 2 or NScreens × 2.
hhInt [array]	Altitude interval for output (m). Can be of size 1 × 2 or NScreens × 2.
BaseModel [string]	Atmospheric modeling function on path.
p1,...,pN [arb]	(Optional) Parameters required for *BaseModel*.

The function maps input altitudes in the interval [*hInt*(1),*hInt*(2)] into altitudes in the interval [*hhInt*(1) *hhInt*(2)] before calling the atmospheric model '*BaseModel*' with altitude *hh* given parameters p1,...,pN. This function is intended to facilitate using atmospheric models in the earth's boundary layer where certain altitudes should be referenced to ground level, and other altitudes should be referenced to sea level. Generally, *hInt*(2) = *hhInt*(2) = boundary-layer altitude limit. However, the code will accept other inputs and use them accordingly. Refer to Section 3.4.2.7 and help for BoundaryAlt for details on the altitude mapping. The outputs are

Model [vector]	Evaluated atmospheric model for new altitudes.
hh [vector]	Mapped altitudes used for evaluating model.

Example 3.4.30 (Use of BoundaryAtm) *This example illustrates different ways to use* BoundaryAtm

```
>> [Model,hh] = BoundaryAtm(0,[0 5000],[1230 5000],'Clear1Night')
```

Model = 1.5059e-015, hh = 1230
0 m altitude maps to 1230 m for input to Clear1Night

```
>> [Model,hh] = BoundaryAtm(0,[1230 5000],[0 5000],'HV57')
```

Model = 1.7270e-014, hh = 0
1230 m altitude maps to 0 m for input to HV57

```
>> [Model,hh] = BoundaryAtm(1230,[1230 5000],[0 5000],'MODFAS','A','1064')
```

Model = 8.8903e-006, hh = 0
1230 m altitude maps to 0 m for input to MODFAS with parameters 'A' (absorption) and '1064' (nm wavelength)

```
>> Atm = AtmStruct(1230,5000,6000,10,'Cn2','HV57');
>> [Model,hh] = BoundaryAtm(1230,[1230 5000],[0 5000],Atm.Cn2Eval)
```

Evaluate the turbulence model in Atm with original altitudes from 1230 to 5000 mapped to the range 0 to 5000 m.

```
>> x = linspace(0,1,10);
>> [hTerr,GndAlt] = TerrainProfiler(x,G);
>> model = BoundaryAtm(EarthAlt(x,G),[GndAlt repmat(5000,10,1)],[0 5000],'HV57')
```

Scales altitudes between ground level and 5 km to be between 0 and 5 km for input to HV57 *(requires DTED data - see* bilLoad*).*

3.4.3.8 Bufton

Syntax

v = Bufton(*h, [vg], [hpk], [hscale], [vt]*)

```
SCALING_CODE_API int Bufton(char** errorChain, const double* h,
     const int nH, const double* vg, const double* hpk,
     const double* hScale, const double* vt, double* windSpeed)
```

This function returns the value of the Bufton wind model as a function of altitude. The Matlab inputs are:

h [vector]	Altitude above sea level (m).
vg [scalar]	(Optional) Ground wind speed - defaults to 5 (m/s).
hpk [scalar]	(Optional) Altitude of the peak - defaults to 9400 (m).
hscale [scalar]	(Optional) Scale height - defaults to 4800 (m).
vt [scalar]	(Optional) Tropopause wind speed - defaults to 30 (m/s).

This model was originally used for wind profiles from the Maui Space Surveillance Site on Haleakala and as such, the altitude corresponding to z = 0 is 3048 meters above MSL when using the default values for ***vg***, ***hpk***, ***hscale***, and ***vt***.[20] The following equation is used to calculate wind speed [7].

$$v = v_g + v_t \times \exp\left[-\left(\frac{h - h_{pk}}{h_{scale}}\right)^2\right]$$

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitudes above ground level (m).
nH [in]	Number of elements in vector ***h***.
vg [in]	(NULL OK) Ground wind speed - defaults to 5 (m/s); assumed to be scalar.
hpk [in]	(NULL OK) Altitude of the peak (altitude of the tropopause) - defaults to 9400 (m); assumed to be scalar.
hScale [in]	(NULL OK) Scale height (thickness of tropopause) - defaults to 4800 (m); assumed to be scalar.
vt [in]	(NULL OK) Tropopause wind speed - defaults to 30 (m/s); assumed to be scalar.
windSpeed [out]	Values of Bufton wind model (m/s); Must be length ***nH***.

The only input is ***h*** and the output is a vector ***v*** the size of ***h*** with the value of wind velocity in m/s. Figure 3.17 is a plot of wind velocity with altitude as calculated using the Bufton wind model.

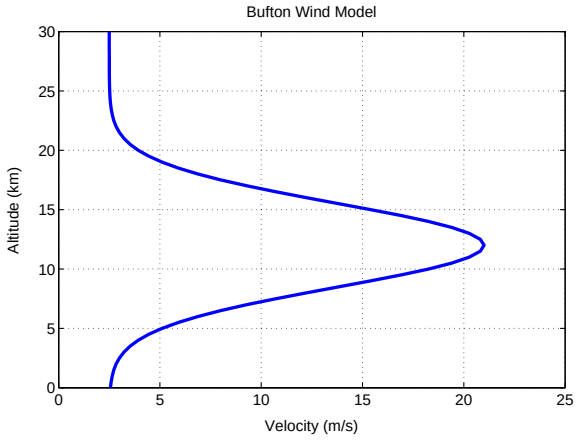


Figure 3.17: Wind profile calculated using the Bufton wind model.

3.4.3.9 CalcHRange

Syntax

HRange = CalcHRange(*h*)

```
SCALING_CODE_API int CalcHRange(char** errorChain, const int nH,
    const double* h, double* hRange)
```

This function is used by functions MODTRAN and FASCODE to calculate the parameter *HRANGE* as a function of altitude for use in the FORTRAN executables MODTRAN and FASCODE when parameter *ITYPE* = 1 and *HRANGE* is not specified. An *ITYPE* of 1 corresponds to horizontal (constant-pressure) path and *HRANGE* specifies the horizontal range for that path. For performing analysis of extinction at a large range of altitudes, one may want to vary *HRANGE* with height because at higher altitudes transmission is high due to low extinction and so a larger *HRANGE* would be needed in order to see a significant enough difference to get nonzero extinction coefficients. Table ?? gives the *HRANGE* value for different ranges of altitudes.

Altitude range (km)	*HRANGE* (km)
$h \leq 5$	10
$5 < h \leq 10$	30
$10 < h \leq 20$	100
$20 < h \leq 30$	300
$h > 30$	500

Table 3.1: *HRANGE* for different altitudes.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nH [in]	Length of input and output vectors.
h [in]	Altitudes at which MODTRAN and FASCODE calculate extinction (m); vector of length *nH*.

hRange [out]	Horizontal range (size of *h*) for MODTRAN/FASCODE (m); length *nH*.

3.4.3.10 ChangeAtm

Syntax

```
Atm = ChangeAtm(Atm, [PName1, NewValue1, [P1], ..., [PN]], ...
     [PName2, NewValue2, [P1], ..., [PN]], ...
     [PName3, ...] )
```

This function is used to change parameters of an existing atmospheric structure from AtmStruct. The inputs are as follows:

Atm [struct]	Atmospheric modeling parameters structure from AtmStruct.
PNames [string]	Name of parameters to be altered. Can be GEOM, SCREENS, wavelength, MaxAlt, alpha with model name, or any of the allowed atmospheric models (see AtmStruct for details).
NewValues	New values for the parameters or model function names for the atmospheric models.
P1...PN	Function arguments required if the new parameter is a function.

Any of the allowed atmospheric models from AtmStruct can be changed, in addition to geometry, number of screens, maximum altitude, turbulence multiplier alpha when a C_n^2 model is present, and wavelength in extinction models. For a change in wavelength to work properly, a wavelength string must be the third input to the model function. Additionally, multipliers of other profiles can be specified with the input 'alpha[model]' for the parameter name. To change turbulence multiplier, specify either 'alphaCn2' or 'alpha.'

Example 3.4.31 (Changing an atmospheric structure) *This example shows how to change geometry, screens or any of the atmospheric models in an atmospheric structure.*

```
>> Atm = AtmStruct(1000,10,20000,8,'Cn2','HV57');
>> AtmNew = ChangeAtm(Atm,'Screens',20)

AtmNew =

        hp: 1000
        ht: 10
        rd: 20000
         L: 2.0026e+004
    LatLong: [1x1 struct]
     MaxAlt: Inf
     xRange: [0 1]
          z: [20x1 double]
         dz: [20x1 double]
```

```
          h: [20x1 double]
         Cn2: [20x1 double]
    Cn2Eval: {'HV57'  'h'}
```

Change the number of evenly-spaced, equal-thickness phase screens from 8 to 20.

```
>> AtmNew = ChangeAtm(Atm,'Screens',[.25 .75],[.5 .5])

AtmNew =

         hp: 1000
         ht: 10
         rd: 20000
          L: 2.0026e+004
    LatLong: [1x1 struct]
     MaxAlt: Inf
     xRange: [0 1]
          z: [2x1 double]
         dz: [2x1 double]
          h: [2x1 double]
         Cn2: [2x1 double]
    Cn2Eval: {'HV57'  'h'}
```

Specify the normalized locations and thicknesses of each atmospheric slab.

```
>> AtmNew = ChangeAtm(Atm,'GEOM',3000,10,2000,'alpha',2)

AtmNew =

         hp: 3000
         ht: 10
         rd: 2000
          L: 3.5975e+003
    LatLong: [1x1 struct]
     MaxAlt: Inf
     xRange: [0 1]
          z: [8x1 double]
         dz: [8x1 double]
          h: [8x1 double]
         Cn2: [8x1 double]
    Cn2Eval: {[2]  'HV57'  'h'}
```

Change geometry (hp, ht, rd) and turbulence multiplier, ***alpha****. Note that slant range (L) has been recomputed.*

```
>> AtmNew = ChangeAtm(Atm,'alpha',[1 2 3 4]);
```

Create a 1x4 array of structures with increasing turbulence multiplier.

```
>> Atm = ChangeAtm(Atm,'Abs','MODFAS','A','1315','RURAL',...
    'Scat','MODFAS','S','1315','RURAL')
```

```
Atm =

          hp: 1000
          ht: 10
          rd: 2.0000e+04
           L: 2.0026e+04
     LatLong: [1x1 struct]
      MaxAlt: Inf
      xRange: [0 1]
           z: [8x1 double]
          dz: [8x1 double]
           h: [8x1 double]
         Cn2: [8x1 double]
     Cn2Eval: {'HV57'  'h'}
         Abs: [8x1 double]
     AbsEval: {'MODFAS'  'h'  'A'  '1315'  'RURAL'}
        Scat: [8x1 double]
    ScatEval: {'MODFAS'  'h'  'S'  '1315'  'RURAL'}
```

Added absorption and scattering to ***Atm****.*

```
>> AtmNew = ChangeAtm(Atm,'Wavelength',1e-6)

AtmNew =

          hp: 1000
          ht: 10
          rd: 2.0000e+04
           L: 2.0026e+04
     LatLong: [1x1 struct]
      MaxAlt: Inf
      xRange: [0 1]
           z: [8x1 double]
          dz: [8x1 double]
           h: [8x1 double]
         Abs: [8x1 double]
     AbsEval: {'MODFAS'  'h'  'A'  [1.0000e-06]  'RURAL'}
         Cn2: [8x1 double]
     Cn2Eval: {'HV57'  'h'}
        Scat: [8x1 double]
    ScatEval: {'MODFAS'  'h'  'S'  [1.0000e-06]  'RURAL'}
```

Changed wavelength to 1 μm.

```
>> Atm = ChangeAtm(Atm,'alphaWind',[1 4 10]);
```

Creates an array of structures with different multipliers on wind speed.

```
>> Atm = ChangeAtm(Atm,'Abs','BoundaryAtm',[1500 5000],[0 5000],Atm.AbsEval)
```

Change the absorption model boundaries

The output is the input ***Atm*** structure with the fields in ***PNames*** having the values in ***NewValues*** and with any dependent fields having been recalculated.

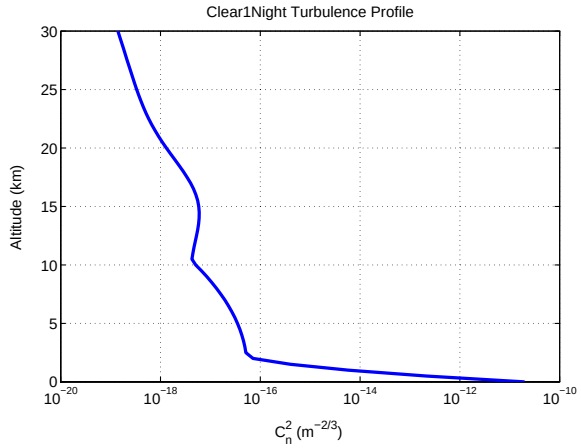


Figure 3.18: Turbulence profile calculated using CLEAR-1 Night.

3.4.3.11 CLEAR1NIGHT

Syntax

Cn2 = CLEAR1NIGHT(*h*)

```
SCALING_CODE_API int CLEAR1NIGHT(char** errorChain, const double* h,
    const int nH, double* cn2)
```

This function returns the value of the CLEAR-1 night turbulence profile given altitude above sea level in meters [35]. CLEAR-1 night is only defined from 1230 m to 30 km altitude, however, this function will return values for altitudes lower than 1230 m and higher than 30 km. The values below 1230 m are an extrapolation of the model form between 1230 m and 2130 m. The values above 30 km are an extrapolation of the model form between 10.34 km and 30 km. The only input is *h* in meters. The following are the equations used to calculate C_n^2 :

$$\log_{10}(C_n^2) = \begin{cases} -10.7025 - 4.3507h + .8141h^2 & h \leq 2.13 \\ -16.2897 + 0.0335h - 0.0134h^2 & 2.13 < h \leq 10.34 \\ -17.0577 - 0.0449h - 0.0005h^2 + 0.6181 \exp\left(-0.5\left(\frac{h-15.5617}{3.4666}\right)^2\right) & h > 10.34 \end{cases}$$

where h is in kilometers. The output is a vector the size of *h* with the values of C_n^2 with units of $\mathrm{m}^{-2/3}$. Figure 3.18 shows the value of C_n^2 as calculated using the CLEAR-1 Night model.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

3.4.3.12 Clear2Night

Syntax

Cn2 = Clear2Night(*h*)

```
SCALING_CODE_API int Clear2Night(char** errorChain, const double* h,
    const int nH, double* cn2)
```

This function returns the Clear-2 Night turbulence profile. Profile is based on data taken above White Sands Missile Range. Data-based profile function is defined for 1230 m $\leq h \leq$ 30 km. Ground level of model is $h =$ 1216 m, and for numerical convenience we define *Cn2* below 1230 m to be equal to the model value at 1230 m. For $h >$ 30 km, returned values are an extrapolation of the data-based function. The Matlab input is:

h [vector]	Altitude above sea level (m).

The Matlab output is:

Cn2 [vector]	Index of refraction structure coefficient ($m^{-2/3}$).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

Figure 3.19 is a plot of turbulence with altitude.

3.4.3.13 Clear2Wind

Syntax

v = Clear2Wind(*h*)

```
SCALING_CODE_API int Clear2Wind(char** errorChain, const double* h,
    const int nH, double* wind)
```

This function returns the value of Clear-2 wind model. The Matlab input is

h [vector]	Altitude above sea level (m).

The Matlab output is

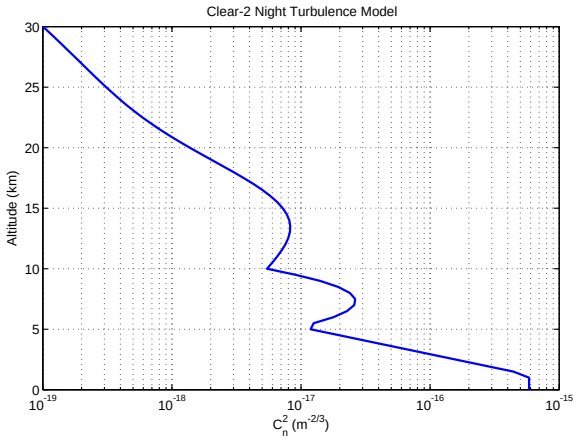


Figure 3.19: Turbulence profile calculated using the Clear-2 night model.

v [vector]	Value of wind model (m/s).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
wind [out]	Value of wind model (m/s); must be length *nH*.

Figure 3.20 is a plot of wind velocity with altitude.

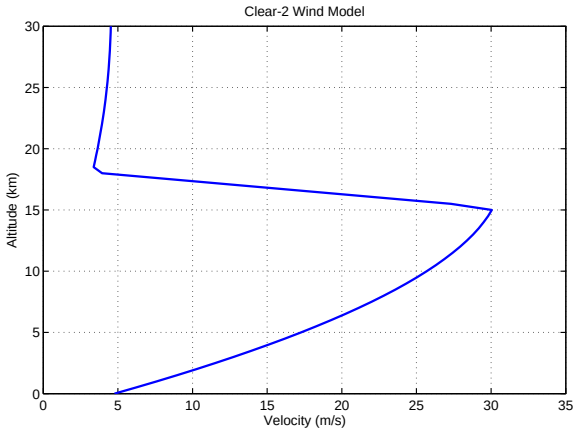


Figure 3.20: Wind profile calculated using the Clear-2 wind model.

3.4.3.14 DEWPT2RH

Syntax

MZA Associates Corporation

[RH, es] = DEWPT2RH(*dewPt, Temp*)

The Matlab outputs are:

RH [vector]	Relative humidity (%).
es [vector]	Saturation vapor pressure (Pa).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
dewPt [in]	The dew point temperature (in Kelvin); must be of length *n*.
Temp [in]	The air temperature (in Kelvin); must be of length *n*.
n [in]	Length of input and output vectors.
RH [out]	The relative humidity (in %); must be of length n.
es [out]	Saturation vapor pressure (in Pascals); must be of length *n*.

3.4.3.15 DLRmodHV57

Syntax

Cn2 = DLRmodHV57(*h*)

```
SCALING_CODE_API int DLRmodHV57(char** errorChain, const double* h,
    const double* meanRefract, const int n, double* Cn2)
```

This function returns the value of the DLR-modified Hufnagel-Valley (5/7) turbulence profile given altitude above sea level in meters.[38] The Matlab inputs are:

h [vector]	Altitude above sea level (m).
meanRefract [vector]	(Optional) Mean refractivity. A vector of the same size as *h* representing the mean refractivity with altitude. Defaults to the exponential model described in the reference below.

The Matlab output is:

Cn2 [vector]	Index of refraction structure coefficient ($m^{-2/3}$).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); must be length *n*.
meanRefract [in]	Mean refractivity with altitude, defaults to the exponential model described in the reference; must be length *n*.
n [in]	Length of input and output vectors.
Cn2 [out]	The index of refraction structure coefficient ($m^{-2/3}$); allocate to length *n*.

3.4.3.16 dynamicPiecewiseMaritimeAIRS

Syntax

Cn2 = dynamicPiecewiseMaritimeAIRS(*h, [Cn2_0], [locale], [ht], [Cn2_ht]*)

Computes C_n^2 using AFIT's dynamic piecewise model based on AIRS maritime observations [39]. The inputs are:

h [vector]	Altitude above sea level (m).
Cn2_0 [scalar]	(Optional) *Cn2* at ground level ($m^{-2/3}$) - defaults to 1.1154e-16 for 'Polar' and 1.7783e-18 for 'Tropic'.
locale [string]	(Optional) Location - either 'Polar' or 'Tropic' - defaults to 'Tropic'.
ht [scalar]	(Optional) Altitude of the tropopause (m) - defaults to 9750 for 'Polar' and 17060 for 'Tropic'.
Cn2_ht [scalar]	(Optional) Value of *Cn2* at the tropopause ($m^{-2/3}$) - defaults to 1.1154e-16.

The output is:

Cn2 [vector]	Index of refraction structure coefficient ($m^{-2/3}$).

3.4.3.17 DynamicViscosity

Syntax

DynViscosity = DynamicViscosity(*T*)

```
SCALING_CODE_API int DynamicViscosity(char** errorChain,
    const double* T, const int nT, double* DynViscosity)
```

This function returns the dynamic viscosity as a function of temperature. The input, *T*, is the temperature in Kelvin. The output is the dynamic viscosity in units of N s/m^2 given by

$$\texttt{DynViscosity} = \frac{\beta\, T^{1.5}}{S+T}$$

where $\beta = 1.458 \times 10^{-6}$ Ns/m^2 and S is Sutherland's constant, $S = 110.4$ K [37].

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
T [in]	Temperature (in Kelvin); must be length *nT*.
nT [in]	Number of elements in *T*.
DynViscosity [out]	Value for the dynamic viscosity (units are (Nsm^{-2})); allocate to length *nT*.

3.4.3.18 EdlenOwens67

Syntax

m = EdlenOwens67(*lambda*)

```
SCALING_CODE_API int EdlenOwens67(char** errorChain, double* lambda,
    const int nl, double* m)
```

This function computes the refractive index modulus, *m*, for a given wavelength, *lambda*, based on work published by Edlen and Owens [13, 40, 41, 42].

$$m = 1 \times 10^{-8}\left(8338.57 + \frac{2405004}{130-\lambda^{-2}} + \frac{15990}{38.9-\lambda^{-2}}\right)$$

where λ is in microns.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lambda [in]	Wavelength (m); must be length *nl*.

nl [in]	Number of wavelengths to evaluate.
m [out]	Refractive index modulus; allocate to *nl*.

3.4.3.19 EqualCn2Screens

Syntax

[x, dx] = EqualCn2Screens(*[n], Atm, [Nscreens], [Geom]*)

```
SCALING_CODE_API int EqualCn2Screens(char** errorChain,
    const int nInputScreens, const double* screenCn2,
    const double* screenThickness, const int nOutputScreens,
    double* normScreenPosition, double* normScreenThickness)
```

This function calculates path-normalized phase screen positions and thicknesses for equal strength phase screens. The Matlab inputs are:

n [scalar]	(Optional) Number of screens for discretizing the calculation. Defaults to Nscreens and is increased as needed.
Atm [struct/string]	Atmospheric modeling parameters. Can be a structure from AtmStruct or a turbulence profile model to be used.
Nscreens [scalar]	(Optional) Number of phase screens to calculate. If *Atm* is a discrete structure from AtmStruct and *Nscreens* is not specified, *Nscreens* will be the number of screens in *Atm*.
Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list of (*hp*, *ht*, *rd*, ...) - not required if *Atm* is a structure from AtmStruct
hp [scalar]	Altitude of transmit/receive platform (m).
ht [scalar]	Altitude of target (m).
rd [scalar]	Downrange of target along spherical earth surface (m).

This function first calculates the plane wave coherence length, $r_0^{-5/3}$, and divides by *Nscreens* so that each segment will have the same strength. The phase screen position will be at the center of the segment. This function discretizes the calculation by breaking the path into *n* segments and integrating C_n^2 to reach the desired strength per phase screen. The Matlab outputs are:

x [vector]	Path-normalized phase screen positions.
dx [vector]	Path-normalized phase screen thicknesses.

If *Atm.MaxAlt* is not infinite, output screen positions are over the path where altitude is less than *Atm.MaxAlt*, as assumed by AtmStruct.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nInputScreens [in]	Number of elements in *screenCn2* and *screenThickness*.
screenCn2 [in]	Cn2 values at consecutive screen positions along path ($m^{-2/3}$).
screenThickness [in]	Size of each consecutive screen along path.
nOutputScreens [in]	Number of desired equal strength screens.
normScreenPosition [out]	Normalized position of each output screen of equal strength.
normScreenThickness [out]	Normalized thickness of each output screen of equal strength.

3.4.3.20 FASCODE

Syntax

ExtCoeff = FASCODE(*Atm, FileName, [FieldName1, FieldValue1], ... [FieldName2, FieldValue2], ...*)

This function runs the FORTRAN executable FASCODE and returns a structure that has the extinction coefficient and the transmission. FASCODE is not distributed with ATMTools. The user must obtain a version of FASCODE and install in ATMTools/OTT/FASCODE. The inputs to this function are as follows:

Atm [struct]	Atmospheric modeling parameters for a discrete atmosphere. Can be a structure from AtmStruct with fields hp, ht, L, h, z.
FileName [string]	(Optional, defaults to 'FAS_NORM2') The file name to be used in FASCODE. Users can input one of their files or use one from the library (Input_library) directory. Must be passed in (can be empty) if using optional arguments that follow.

FieldName [string]	Name of parameter to be modified from the value in *FileName*. Available parameters are: 'LAMBDA'–Wavelength (m), 'MODEL'–Atmospheric model (1-6), 'IHAZE'–Aerosols model to be used (1-10), 'IVULCN'–Extinction type for stratospheric aerosols (1-8), 'ICLD'–inclusion of clouds (1-20), 'IAERSL' –Aerosols (0 or 1), 'ITYPE'–Type of path (1, 2 (default)), 'HRANGE'–Horizontal path length, ITYPE=1 (m), 'VIS'–Surface meteorological range (km), 'FAS_EXE'–Filename of the FASCODE executable file. Must be in the ATMToolsPath/OTT/FASCODE folder. Defaults to FASCD3P.EXE, 'LNFL_EXE'–Filename of the FASCODE executable file. Must be in the ATMToolsPath/OTT/HITRAN folder. Defaults to LNFL.exe
FieldValue [scalar]	Value of specified *FieldName* with units depending on *FieldName* as above.

The output is a single structure as follows:

ExtCoeff [struct]	Structure containing the extinction coefficients and information about the run parameters.
ExtCoeff.ver [string]	Version of FASCODE run.
ExtCoeff.Name [struct]	The name of each parameter included in the structure.
ExtCoeff.Value [struct]	In which the values of the parameters are stored.
ExtCoeff.Units [struct]	The units of this parameter.

FieldNames are FASCODE parameters. Refer to FASCODE Users Manual. If the user specifies any of these to be changed, the output substructures will have a field corresponding to that parameter. The substructures also have fields *Heights* (input altitudes from *Atm.h*), *SubTran* (transmission) and *SubAlpha* (extinction coefficient). The extinction coefficient and transmission can be due to total extinction, molecular extinction or molecular absorption and depends on the input parameters as shown in Table ??. An asterisk indicates that any value can be used.

Extinction Type	*IHAZE*	*IAERSL*
Total	*	*
Molecular Extinction	0	*
Molecular Absorption	0	0

Table 3.2: FASCODE parameters for extracting different extinction coefficients.

3.4.3.21 GADSAtmos

Syntax

```
out = GADSAtmos(h, model, lambda, site, ...
    [BLalt], [season], [TOD], [percentile])
```

This function generates absorption and scattering coefficients as well as temperature and pressure using the Global Aerosol Data Set (GADS) and LEEDR/ExPERT meteorological data. The user must have a MATLAB® version (m-code or p-code) on the MATLAB® path to use this function. It will generate the proper structure for use in LEEDRAtmos based on the provided inputs. The inputs are:

h [vector]	Altitude AGL (m).
model [string/cell]	'A' for absorption, 'S' for scattering, 'T' for temperature, 'P' for pressure, 'R' for Relative humidity, 'D' for dew-point, 'AA' for aerosol absorption, 'AS' for aerosol scattering, 'MA' for molecular absorption, and 'MS' for molecular scattering. Can be a cell array of models in which case the output can be a single matrix with the appropriate data or each *model* can be output into separate variables, see the examples below. If passed in as [], output will be a structure with all *model* data.
lambda [string]	Identifier for desired laser wavelength. See Table 3.3.
site [scalar]	Site identifier. There are more than 500 sites in the database. Type LEEDRSites for a listing.
BLalt [scalar]	(Optional) Altitude of the boundary layer, defaults to 1524 for daily avg or *season*/time dependent values (m).
season [scalar]	(Optional) Index to *season*: 1 - summer (default), 2 - winter.
TOD [scalar]	(Optional) Index to time of day. See Table 3.4.
percentile [scalar]	(Optional) Index to atmosphere *percentile*. See Table 3.5.

Index	String	Value (μm)	Index	String	Value (μm)
1	'355'	0.355	2	'380'	0.380
3	'400'	0.400	4	'532'	0.532
5	'525'	0.525	6	'550'	0.550
7	'620'	0.620	8	'650'	0.650
9	'670'	0.670	10	'680'	0.680
11	'690'	0.690	12	'1000'	1.000
13	'1030'	1.030	14	'1045'	1.045
15	'1047'	1.047	16	'1054'	1.054
17	'1064'	1.0642	18	'1315'	1.31525
19	'1586'	1.586	20	'1625'	1.625
21	'2141'	2.141	22	'3800'	3.800
23	'4636'	4.636	24	'9300'	9.300
25	'10600'	10.600			

Table 3.3: Wavelength index and corresponding values for input to GADSAtmos.

The output is:

Index	Time
1	00:00 to 03:00
2	03:00 to 06:00
3	06:00 to 09:00
4	09:00 to 12:00
5	12:00 to 15:00
6	15:00 to 18:00
7	18:00 to 21:00
8	21:00 to 00:00
9	Daily average (default)

Table 3.4: Time of day (*TOD*) index and corresponding values for input to GADSAtmos.

Index	1	2	3	4	5 (default)	6	7	8	9
Percentile	1%	5%	10%	20%	50%	80%	90%	95%	99%

Table 3.5: Relative humidity (*percentile*) index and corresponding values for input to GADSAtmos.

out [vector]	Absorption or scattering coefficient (1/m), temperature (K), pressure (Pa), rel humidity (%), or dewpoint (K).

Example 3.4.32 (GADSAtmos) *Examples using* GADSAtmos *to compute atmospheric model data.*

```
>> h = linspace(10,1000,10)';
>> Abs = GADSAtmos(h,'A','1315',1,1524,1,9,5)
```

Compute absorption for the specified inputs - must have p-code or m-code version of LEEDR on the path.

```
>> [Abs Temp] = GADSAtmos(h,{'A' 'T'},'1315',1,1524,1,9,5)
%    - Compute absorption and temperature for the specified inputs
```

Compute absorption and temperature for the specified inputs.

```
>> [MAbs MScat] = GADSAtmos(h,{'MA' 'MS'},'1315',1,1524,1,9,5)
```

Return molecular absorption and scattering from the computed LEEDR profile.

```
>> data = GADSAtmos(h,{'A' 'T'},'1315',1,1524,1,9,5)
```

Returns an Nx2 matrix with absorption in column 1 and temperature in column 2.

```
>> data = GADSAtmos(h,[],'1315',1,1524,1,9,5)
```

Returns a structure with all of the models (including a breakout of aerosol and molecular absorption and scattering).

```
>> Atm = AtmStruct(G,100,'Abs','GADSAtmos','A','1315',1,...)
```

Generate an ***Atm*** *structure using the ExPERT data.*

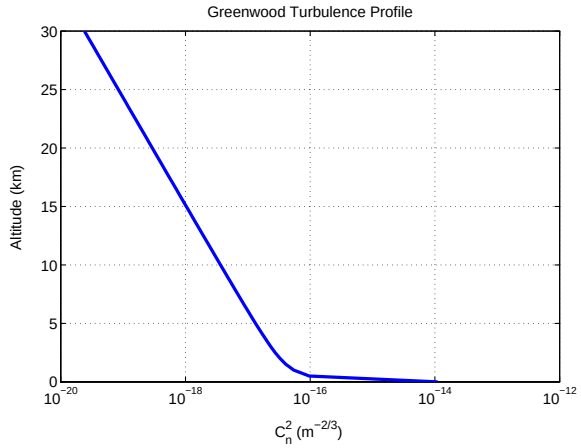


Figure 3.21: Turbulence profile calculated using Greenwood model.

3.4.3.22 GreenwoodCn2

Syntax

Cn2 = GreenwoodCn2(*h*)

```
SCALING_CODE_API int GreenwoodCn2(char** errorChain, const double* h,
    const int nH, double* cn2)
```

This function returns the value of the Greenwood turbulence profile given altitude above sea level *h* in meters [14]. The equation used to calculate C_n^2 is

$$C_n^2 = [2.2 \times 10^{-13}(h+10)^{-1.3} + 4.3 \times 10^{-17}] \exp\left(-\frac{h}{4000}\right).$$

The output *Cn2* is a vector the size of *h* with the values of C_n^2 , the index of refraction structure coefficient, with units of $\mathrm{m}^{-2/3}$. Figure 3.21 shows C_n^2 as a function of altitude as calculated by the Greenwood turbulence model.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

3.4.3.23 HufnagelValley

Syntax

Cn2 = HufnagelValley(*h*, *[v]*, *[A]*)

Cn2 = HufnagelValley(*h*, *lambda*, *r0*, *theta0*)

```
SCALING_CODE_API int HufnagelValley(char** errorChain,
    const double* h, const int nH, const double* highAltWindSpeed,
    const double* groundTurb, double* cn2)
```

This function returns the value of the Hufnagel Valley turbulence profile given altitude above sea level in meters [35]. The Matlab inputs are:

h [vector]	Altitude above sea level (m).
v [scalar]	(Optional) High altitude wind speed (m/s).
A [scalar]	(Optional) Strength of turbulence near the ground ($m^{-2/3}$).
lambda [scalar]	Laser wavelength (m). Defaults to 500 nm. Can pass empty ([]) to use default value and specify *r0* and *theta0*.
r0 [scalar]	Fried parameter (m). Defaults to 0.05 m. Can pass empty ([]) to use default value and specify *theta0*.
theta0 [scalar]	Isoplanatic angle (rad). Defaults to 7e-6 rad. Can pass empty ([]) to use default value.

If *v* and *A* are not passed, the function defaults to HV (5/7) by setting $v = 21$ m/s and $A = 1.7\times10^{-14}\mathrm{m}^{-2/3}$. If all of *lambda*, *r0*, and *theta0* are provided, values for *v* and *A* are computed using [43].

$$\begin{aligned} v &= 27\sqrt{75\theta_0^{-5/3}\lambda^2 - 0.14} \\ A &= 1.29\times10^{-12}r_0^{-5/3}\lambda^2 - 1.61\times10^{-13}\theta_0^{-5/3}\lambda^2 - 3.89\times10^{-15} \end{aligned}$$

Once values for *v* and *A* are determined, the equation below is used to calculate C_n^2 .

$$C_n^2 = 5.94\times10^{-53}\left(\frac{v}{27}\right)^2 h^{10}\exp\left(\frac{-h}{1000}\right) + 2.7\times10^{-16}\exp\left(\frac{-h}{1500}\right) + A\exp\left(\frac{-h}{100}\right)$$

The output *Cn2* is a vector the size of *h* with the values of C_n^2 with units of $m^{-2/3}$. Figure 3.22 shows C_n^2 as a function of altitude as calculated by Hufnagel Valley model for $A = 2\times10^{-14}\,m^{-2/3}$ and for different values of *v*.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.

highAltWindSpeed [in]	(NULL OK) High altitude wind speed (m/s); defaults to 21 m/s; scalar.
groundTurb [in]	(NULL OK) Strength of turbulence near the ground ($m^{-2/3}$); defaults to 1.7e-14 $m^{-2/3}$; scalar.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

3.4.3.24 HV57

Syntax

Cn2 = HV57(*h*)

```
SCALING_CODE_API int HV57(char** errorChain, const double* h,
    const int nH, double* cn2)
```

This function returns the value of the Hufnagel-Valley (5/7) turbulence profile given altitude above sea level *h* in meters [7]. This is the same as HUFNAGEL VALLEY with $v = 21$ m/s and $A = 1.7 \times 10^{-14}\,\mathrm{m}^{-2/3}$. The only input is *h* and the output *Cn2* is a vector the size of *h* with the values of C_n^2 with units of $\mathrm{m}^{-2/3}$. The equation used to calculate C_n^2 is as follows:

$$C_n^2 = 5.94 \times 10^{-53} \left(\frac{v}{27}\right)^2 h^{10} \exp\left(\frac{-h}{1000}\right) + 2.7 \times 10^{-16} \exp\left(\frac{-h}{1500}\right) + A \exp\left(\frac{-h}{100}\right)$$

Figure 3.23 shows the values of C_n^2 as calculated using HV 5/7 model.
The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

3.4.3.25 INNERSCALE

Syntax

Lout = INNERSCALE(*h*)

```
SCALING_CODE_API int INNERSCALE(char** errorChain, const int nH,
    const double* h, double* lIn)
```

This function computes the inner scale for turbulence using a physical model. The only Matlab input is:

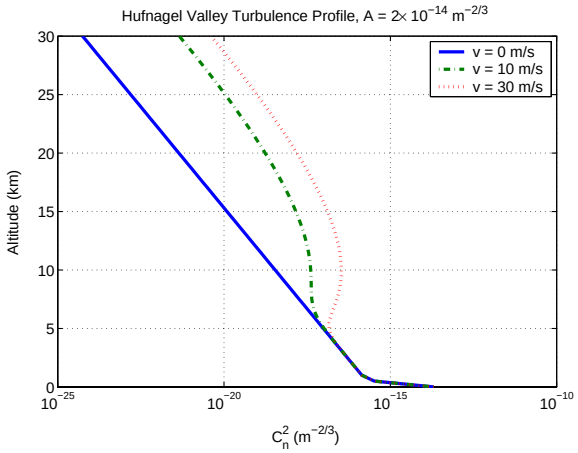


Figure 3.22: Turbulence profile calculated using Hufnagel Valley model for $A = 2\times10^{-14}\,\mathrm{m}^{-2/3}$ and different values of v.

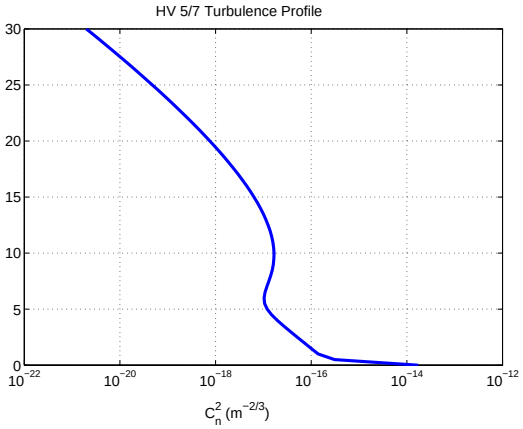


Figure 3.23: Turbulence profile calculated using HV 5/7 model.

h [vector]	Altitude above sea level (m).

The Matlab output is:

Lin [vector]	Inner scale for turbulence (m).

where *Lin* is computed using

$$L_{in} = 3.5 \times 10^{-3} \left(1 + e^{h/8000}\right). \tag{3.9}$$

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nH [in]	Length of input and output vectors.
h [in]	Altitude above sea level (m); vector of length *nH*.
lIn [out]	Inner scale for turbulence (m); length *nH*.

3.4.3.26 LEEDRAtm

Syntax

ModelData = LEEDRAtm(*G, NScreens, [lambda]*)

This function generates an *Atm* structure using all LEEDR data. LEEDR data must be loaded and available in the MATLAB® application data - see loadLEEDRAtmos. The inputs are:

Geom [struct/list]	Geometry parameters. Can be a structure from GeomStruct or a comma separated list of (*hp*, *ht*, *rd*, ...).
hp [scalar]	Platform altitude (m AGL).
ht [scalar]	Target altitude (m AGL).
rd [scalar]	Downrange along spherical earth surface (m).

NScreens [scalar]	(Optional) Number of equal thickness phase screens. Screens are spaced symmetrically about the midpoint of the propagation path. For a continuous model use *NScreens*=Inf.
x [vector]	(Optional) Path-normalized screen location in range $0 \leq x < 1$. Passing *x*=1, will result in a single phase screen placed at the midpoint of the propagation path. For a single screen at the target, use *x*=0.9999.
dx [vector]	(Optional) Path-normalized screen thickness. Must be the same size as *x* if passed.
lambda [scalar]	(Optional) Desired wavelength for computation of absorption, scattering, and extinction (m). If this wavelength does not match the wavelength in the precomputed data, LEEDR will be re-run. If not input, will use the first wavelength in the LEEDR data.

Note that if *x* is a scalar value equal to 1, it will be interpreted as *NScreens* = 1 and any value passed for *dx* could be interpreted as wavelength. For a single screen at the target, use *x* = 0.9999 and *dx* = 1.

The output is:

Atm [struct]	Structure containing atmospheric screen parameters.

Example 3.4.33 (LEEDRAtm) *Generate* *Atm* *structures using* LEEDRAtm.

```
>> loadLEEDRAtmos('LEEDR_RuralMidLatSummer.h5')
>> Atm = LEEDRAtm(GeomStruct,10,1e-6)
```

Generate *Atm* *using LEEDR data with extinction at 1e-6 m wavelength.*

```
>> Atm = LEEDRAtm(GeomStruct,10)
```

Generate *Atm* *with extinction at the wavelength of the LEEDR data.*

3.4.3.27 LEEDRAtmos

Syntax

ModelData = LEEDRAtmos(*h, modelName, [lambda]*)

```
SCALING_CODE_API int LEEDRAtmos(char** errorChain,
    const char* lutFileName, const char* modelName, const int nH,
    const double* h, const double lambda, double* modelData)
```

This function generates model data at specified altitudes, *h*, based on pre-computed LEEDR atmospheric data. The Matlab inputs are:

h [vector]	Altitude above sea level along the path (m).
ModelName [string]	Desired LEEDR model.
lambda [scalar]	(Optional) Desired wavelength for computation of absorption, scattering, and extinction (m). If this wavelength does not match the wavelength in the precomputed data, LEEDR will be re-run. This input is only required for model types 'A', 'S', and 'E'.

The following is a list of available models and the options

- Temperature - 'currentTemperature', 'temperature', or 'T',
- Relative Humidity - 'currentRH', 'relativehumidity', 'RH' or 'R',
- Pressure - 'currentPressure', 'pressure', 'PR', or 'P',
- Dew Point - 'currentDewpoint', 'dewpoint', 'DP', or 'D',
- Wind Speed - 'naturalWind', 'windSpeed' or 'W',
- Wind Heading - 'windDir', 'windHeading' or 'WH',
- C_n^2 - 'Turb' or 'C',
- Absorption - 'TotalAbs' or 'A',
- Molecular Absorption - 'MolecAbs' or 'MA',
- Aerosol Absorption - 'AeroAbs' or 'AA',
- Cloud Absorption - 'CloudAbs' or 'CA',
- Rain Absorption - 'RainAbs' or 'RA',
- Scattering - 'TotalScat' or 'S',
- Molecular Scattering - 'MolecScat' or 'MS',
- Aerosol Scattering - 'AeroScat' or 'AS',
- Cloud Scattering - 'CloudScat' or 'CS',
- Rain Scattering - 'RainScat' or 'RS',
- Extinction - 'TotalExt' or 'E'.

If the user has not loaded LEEDR data prior to calling LEEDRAtmos, the function will call buildLEEDRAtmos in an attempt to generate data for the default set of LEEDR inputs, which requires p-code or m-code version of LEEDR on the Matlab® path. Also, p-code or m-code version of LEEDR is required in order to compute extinction coefficients at wavelengths other than that specified in the LEEDR data. For information on using LEEDR with ATMTools and getting pre-computed LEEDR data into Matlab®, see Section 3.4.4.

The Matlab output is:

ModelData [vector]	Model values at specified altitudes.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutFileName [in]	Name of the file containing LEEDR lookup table data; file must be in the directory specified by SetLEEDR-LookupPath or must contain the full path to the file; pass in a single, null-terminating string.
modelName [in]	Desired LEEDR model; can be any one of the following (case insensitive): Cn2 - "C", "Cn2", or "Turb"; Abs - "A", "Abs", "Absorption", "TotalAbs", or "TotalAbsorption"; AeroAbs - "AA", "AAbs", "AeroAbs", or "AerosolAbsorption"; MolecAbs - "MA", "MAbs", "MolecAbs", or "MolecularAbsorption"; CloudAbs - "CA", "CAbs", "CldAbs", "CloudAbs", or "CloudAbsorption"; RainAbs - "RA", "RAbs", "RainAbs", or "RainAbsorption"; Scat - "S", "Scat", "Scattering", "TotalSca", "TotalScat", or "TotalScattering"; MolecScat - "MS", "MScat", "MolecSca", "MolecScat", or "MolecularScattering"; AeroScat - "AS", "AScat", "AeroSca", or "AerosolScattering"; CloudScat - "CS", "CScat", "CldSca", "CldScat", "CloudSca", "CloudScat", or "CloudScattering"; RainScat - "RS", "RScat", "RainSca", "RainScat", or "RainScattering"; Ext - "E", "Ext", "Extinction", "TotalExt", or "TotalExtinction"; Temp - "T", "Temp", or "Temperature"; Wind - "W", "NaturalWind", or "WindSpeed"; WindHeading - "WH", "WindDir", or "WindHeading"; Press - "P", "Press", or "Pressure"; RH - "R", "RH", or "RelativeHumidity"; DewPoint - "D", "DP", or "DewPoint";
nH [in]	Length of input *h* and output modelData.
h [in]	Altitude above sea level (m); vector of length *nH*.
lambda [in]	Desired wavelength for computation of absorption, scattering, and extinction (m); ignored if model is not related to transmission; if wavelength is required but LUT does not have the requeted wavelength, an error is returned.
modelData [out]	Model values at specified altitudes; must be length *nH*.

Example 3.4.34 (LEEDRAtmos) *This example shows how to create an Atm structure that references LEEDR atmospheric data.*

```
>> loadLEEDRAtmos('LEEDR_RuralMidLatSummer.h5');
>> Atm = AtmStruct(GeomStruct,10,'Cn2','LEEDRAtmos','C', ...
     'Ext','LEEDRAtmos','E',1e-6)
```

Generate an `Atm` *structure using* `LEEDRAtmos` *for turbulence and extinction profile. The file in the call to loadLEEDRAtmos is distributed with ATMTools as an example data file and can be found in the directory returned from* `LEEDRLookupPath`.

3.4.3.28 LEEDRWxCube

Syntax

ModelData = LEEDRWxCube(*h, modelName, [lambda], x, G, [DTEDdir], ... [lutLocation], [lutDateTime]*)

ModelData = LEEDRWxCube(*h, modelName, [lambda], lats, lons, ... [DTEDdir], [lutLocation], [lutDateTime]*)

```
SCALING_CODE_API int LEEDRWxCube(char** errorChain,
     const char* lutLocation, const char* lutDateTime,
     const char* modelName, const int nLocs, const double* alts,
     const double* lats, const double* lons, const double lambda,
     bool useElevation, double* modelData)
```

This function generates model data at specified altitudes (AGL), *h*, based on precomputed LEEDR Wx Cube data generated using the `buildLEEDRWxCube` function. To convert between MSL (used by SHaRE and ATMTools) and AGL, use `TerrainAtm` (requires DTED data from http://earthexplorer.usgs.gov/ - see `bilLoad`). If the weather cube was generated using `buildLEEDRWxCube`, there is an option to also generate the terrain elevation data for the cube (provided the DTED data is available). If *DTEDdir* is true or if *DTEDdir* specifies a directory with the necessary DTED data, terrain elevation data will be computed if it doesn't already exist and elevation.h5 file will be saved to the *lutLocation* directory. If the elevation.h5 file exists in the *lutLocation* directory, it will be used to convert MSL to AGL within the function. To force AGL computations, set the input *DTEDdir* to false.

The inputs are:

h [vector]	Altitude above ground level along the path (m). LEEDR weather cube data is all AGL. To convert from MSL to AGL use `TerrainAtm` (see example below).
ModelName [string]	Desired LEEDR model. Can be any one of the following (case insensitive): Cn2 - 'C' or 'Turb'; Abs - 'A' or 'Absorption'; Scat - 'S' or 'Scattering'; Ext - 'E' or 'Extinction'; Temp - 'T' or 'Temperature'; Wind - 'W', 'NaturalWind', or 'WindSpeed'; WindHeading - 'WH' or 'WindDir'; Press - 'P' or 'Pressure'; RH - 'R' or 'RelativeHumidity'; Albedo - 'Alb' or 'Albedo'; Density - 'D', 'AirDensity', or 'Density'; GroundAlt - 'G', 'GA', or 'GroundAlt'.

lambda [scalar]	(Optional) Desired wavelength for computation of absorption, scattering, and extinction (m). This input is only required for model types 'A', 'S', and 'E', can be omitted otherwise.
x [scalar/vector]	Number of screens or norm screen locations.
G [struct]	Geometry structure as from GeomStruct.
lats [vector]	Latitudes of screen locations (deg).
lons [vector]	Longitudes of screen locations (deg).
DTEDdir [bool/char]	(Optional) Directory holding the downloaded DTED data - data can be downloaded from http://earthexplorer.usgs.gov/. Set to false to prevent generation and/or use of elevation data. If true, uses elevation data to convert from MSL to AGL altitudes and will generate elevation data if not there and DTED data is in the default DTED directory - (`EngagementToolsPath(-1)/DTED`). Defaults to true.
lutLocation [string]	(Optional) Location identifier for the weather cube. Defaults to last used value.
lutDateTime [string]	(Optional) Date/time/cycle identifier for weather cube in the format of YYYYMMDDHHHHCCC such as '201501010000006' for 1 Jan 2015, 00:00, 6 hour forecast. Defaults to the last used value.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutLocation [in]	Location identifier for weather cube, pass in a single, null-terminating string.
lutDateTime [in]	Date/time/cycle identifier for weather cube (YYYYMMDDHHHHCCC), pass in a single, null-terminating string such as "201501010000006".
modelName [in]	Desired LEEDR model; can be any one of the following (case insensitive): Cn2 - "C", "Cn2", or "Turb"; Abs - "A", "Abs", "Absorption"; Scat - "S", "Scat", "Scattering"; Ext - "E", "Ext", "Extinction"; Temp - "T", "Temp", or "Temperature"; Wind - "W", "NaturalWind", or "WindSpeed"; WindHeading - "WH", "WindDir", or "WindHeading"; Press - "P", "Press", "Pressure"; RH - "R", "RH", or "RelativeHumidity"; Albedo = "Alb" or "Albedo"; Density - "Dens", "AirDensity", or "Density"; GroundAlt - "G", "GA", or "GroundAlt."

nLocs [in]	Number of elements to evaluate.
lats [in]	Latitudes to evaluate; must be length *nLocs*.
lons [in]	Longitudes to evaluate; must be length *nLocs*.
alts [in]	Altitude above sea level (m); must be length *nLocs*.
lambda [in]	Desired wavelength for computation of absorption, scattering, and extinction (m); ignored if model is not related to transmission; if wavelength is required but LUT does not have the required wavelength, an error is returned.
modelData [out]	Model values at specified altitudes; allocate to *nLocs*.

Example 3.4.35 (LEEDRWxCube) *This example illustrates various calls to* LEEDRWxCube *and creating an Atm structure using* LEEDRWxCube.

```
>> G = GeomStruct('LLA',[38.9 -76.8 50],[39.02 -76.96 193]);
>> Atm = AtmStruct(G,10, ...
          'Cn2','LEEDRWxCube','C','x','G',false,'NCR','201501010000006', ...
          'Ext','LEEDRWxCube','E',1.06e-6,'x','G',false,'NCR','201501010000006')
```

Generate an Atm structure using LEEDR data for turbulence and extinction (should use 'x' and 'G' so if geometry changes, the screen positions will also get updated). This assumes ground level of 0 meters and no terrain effects.

```
>> LLA = ecf2LLA(ScreenECF(10,G));
>> Cn2 = LEEDRWxCube(LLA(:,3),'C',LLA(:,1),LLA(:,2),'C:/MyDTEDdir','NCR','201501010000006')
```

Uses elevation.h5 data, if present, OR generates from DTED data in C:/MyDTEDdir if all necessary data has been downloaded. If elevation.h5 does not exists and DTED data is not available, assumes 0 ground altitude.

```
>> ext = LEEDRWxCube(LLA(:,3),'E',1.06e-6,LLA(:,1),LLA(:,2), ...
          true,'NCR','201501010000006')
```

Uses elevation.h5 data, if present, OR attempts to generate from default DTED directory.

```
>> ext = TerrainAtm(LLA(:,3),grndAlt,'LEEDRWxCube','E',1.06e-6, ...
          LLA(:,1),LLA(:,2),true,'NCR','201501010000006')
```

Compute extinction for fixed ground level grndAlt. TerrainAtm *will pass AGL altitudes to* LEEDRWxCube *and will update the call to* LEEDRWxCube *to force AGL calculations.*

```
>> Atm = AtmStruct(G,100, ...
          'Cn2','TerrainAtm','LEEDRWxCube','C','x','G',true,'NCR','201501010000006', ...
          'Ext','TerrainAtm','LEEDRWxCube','E',1.06e-6,'x','G',true,'NCR','201501010000006')
```

Atm structure with turbulence and extinction using TerrainAtm with elevation data to adjust altitude.

3.4.3.29 LEEDRWXCUBEATM

Syntax

ModelData = LEEDRWXCUBEATM(*G, NScreens, lambda, lutLocation, lutDateTime*)

This function generates an *Atm* structure using all LEEDR Wx cube data. LEEDR Wx cube data will be loaded if not already (see LOADLEEDRWXCUBE). The inputs are

Geom [struct]	Geometry structure from GEOMSTRUCT.
NScreens [scalar]	(Optional) Number of equal thickness phase screens. Screens are spaced symmetrically about the midpoint of the propagation path. For a continuous model use *NScreens*=Inf.
x [vector]	(Optional) Path-normalized screen location in range $0 \leq x < 1$. Passing *x*=1 will result in a single phase screen placed at the midpoint of the propagation path. For single screen at the target, use *x*=0.9999.
dx [vector]	(Optional) Path-normalized screen thickness. Must be the same size as *x* if passed.
lambda [scalar]	Desired wavelength for computation of absorption, scattering, and extinction (m). If this wavelength does not match the wavelength in the precomputed data, an error will occur.
DTEDdir [bool/char]	(Optional) Directory holding the downloaded DTED data - data can be downloaded from http://earthexplorer.usgs.gov/. Set to false to prevent generation and/or use of elevation data. If true, uses elevation data to convert from MSL to AGL altitudes and will generate elevation data if not there and DTED data is in the default DTED directory - (EngagementToolsPath(-1)/DTED). Defaults to true.
lutLocation [char]	Location identifier for weather cube.
lutDateTime [char]	Date/time/cycle identifier for weather cube (YYYYMMDDHHHHCCC).

The output is an *Atm* structure, as from ATMSTRUCT using LEEDRWXCUBE for all atmospheric models.

Example 3.4.36 (LEEDRWxCubeAtm) *This example illustrates creating an* *Atm* *structure using* LEEDRWXCUBEATM.

```
>> G = GeomStruct('LLA',[39.75 -83.5 330],[39.79 -83.5 5000]);
>> Atm = LEEDRWxCubeAtm(G,10,1e-6,true,'N40.00_N39.50_W83.00_W84.50', ...
      '201802270000006')
```

Generate *Atm* *using LEEDR Wx Cube data with abs/scat at 1e-6 microns. The specified data is distributed with ATMTools and can be found in the directory returned from* LOOKUPPATH('LEEDRWXCUBE'). *In this example, the geometry specified is within the bounds of the LUT.*

3.4.3.30 LogScaling

Syntax

Model = LogScaling(*h, base, A, B*)

```
SCALING_CODE_API int LogScaling(char** errorChain, const double* h,
     const int nH, const double base, const double baseValue,
     const double scaleHeight, double* model)
```

This function is to be used for generic altitude-dependent, logarithmically-descreasing atmospheric models with surface value *A* and scale height *B*. The Matlab inputs are:

h [vector]	Altitude above sea level (arbitrary length units).
base [scale]	Base for logarithmically-decreasing model.
A [scalar]	Value of model at *h*=0 (arbitrary units).
B [scalar]	Scale height of log model (same units as *h*).

The output *Model* is a vector the size of *h* with the value of the function $Model = A \times \text{base}^{-(h/B)}$.
The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level.
nH [in]	The length of the array *h*.
base [in]	The exponential base for logarithmically-decreasing model.
baseValue [in]	Value of model at h=0.
scaleHeight [in]	The scale height of the log model (same units as *h*).
model [out]	Value for each input altitude *h*.

3.4.3.31 LUTAtm

Syntax

ModelData = LUTAtm(*h, varname, fileName*)

This function is a general LUT function for extrapolating atmospheric data. The inputs are

h [vector]	Altitude (m).
varname [string]	Name of vector in *fileName* to interpolate into.
fileName [string]	Name of MATLAB® data file containing the data for interpolation.

The specified MATLAB® data file is a user created file and must contain a vector Altitude and a vector for the desired data. If this function and the output data are to be used within an Atm structure, the units of the saved data must be as described in ATMSTRUCT. Additionally, the variable name for the data to be interpolated should not coincide with the keyword in ATMSTRUCT for the model type. For example, data for a turbulence profile should not be named *Cn2* but should be named something like *turb* instead.

The output is

ModelData [vector]	Model data, the result of the interpolation.

Example 3.4.37 (LUTAtm) *This example illustrates use of* LUTATM.

```
>> Altitude = linspace(0,10e3,500)';
>> totalAbsorption = MODFAS(Altitude,'A',1e-6,'MidLatSum');
>> save('MyExtinctionData.mat','Altitude','totalAbsorption')
```

Create an example LUT for use with LUTATM.

```
>> Model = LUTAtm(h,`totalAbsorption',`MyExtinctionData.mat')
```

Interpolates vectors Altitude and totalAbsorption found in the specified file.

```
>> Atm = AtmStruct(10,5000,10000,20,'Cn2','LUTAtm','turb','MyTurbulenceData.mat')
```

Create an Atm structure that uses LUTATM *as an atmospheric model for turbulence as interpolated from MyTurbulenceData.mat.*

3.4.3.32 MAUI3CN2

Syntax

Cn2 = MAUI3CN2(*h*)

```
SCALING_CODE_API int Maui3Cn2(char** errorChain, const double* h,
     const int nH, double* cn2)
```

This function returns the Maui 3 Turbulence profile as defined in standard literature [44]. This is the most recent (vintage Feb 2003) model for propagation above Maui's Mt. Haleakala. It is considered more correct than the old AFGL AMOS model for matching median propagation quantities. Maui 3 has been designated the official Maui-site model for the Active Track program. Data-based profile values are defined for $h > 3050$ m. Ground level is $h = 3038$ m, and for numerical convenience we define C_n^2 below 3050m to be equal to C_n^2 (3050 m). The Matlab input is:

h [vector]	Altitude above sea level (m).

The API function returns system error codes and the arguments are:

errorChain in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

The Matlab output is:

Cn2 [vector]	Index of refraction structure coefficient ($m^{-2/3}$).

Figure 3.24 is a plot of turbulence with altitude.

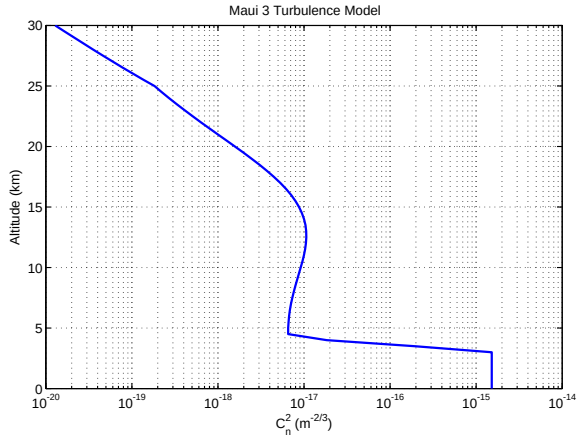


Figure 3.24: Turbulence profile calculated using the Maui 3 model.

3.4.3.33 MIEV0

Syntax

```
[QEXT, QSCA, GQSC, SFORW, SBACK, S1, S2, TFORW, TBACK, PMOM, SPIKE] = ...
    MIEV0(XX, CREFIN, [PERFCT], [ANYANG], [XMU], [MIMCUT], [NMOM], ...
    [IPOLZN] )
```

This function computes Mie scattering and extinction efficiencies; asymmetry factor; forward- and backscatter amplitude; scattering amplitudes vs. scattering angle for incident polarization parallel and perpendicular to the plane of scattering; coefficients in the Legendre polynomial expansions of either the unpolarized phase function or the polarized phase matrix; some quantities needed in polarized radiative transfer; and information about whether or not a resonance has been hit. This MATLAB® code is a transcription of the original FORTRAN code by Wiscombe [45, 46]. Where possible, vectorization has been implemented. MOST input and internal variables and names have been preserved. NOTE : *S1*, *S2*, *SFORW*, *SBACK*, *TFORW*, AND *TBACK* are calculated internally for negative imaginary refractive index; for positive imaginary index, their complex conjugates are taken before they are returned, to correspond to customary usage in some parts of physics (in particular, in papers on cam approximations to Mie theory).

The inputs are:

XX [scalar]	Mie size parameter (2 ∗ pi ∗ radius / wavelength).
CREFIN [scalar]	Complex refractive index (imag part can be + or -, but internally a negative imaginary index is assumed). If imag part is - , scattering amplitudes as in Van de Hulst [47] are returned; if imag part is + , complex conjugates of those scattering amplitudes are returned (the latter is the convention in physics). ∗∗ % NOTE ∗∗ In the 'PERFECT' case, scattering amplitudes in the Van de Hulst convention will automatically be returned unless Im(*CREFIN*) is positive; otherwise, *CREFIN* plays no role.
PERFCT [scalar]	(Optional) If TRUE, assume refractive index is infinite and use special case formulas for Mie coefficients 'a' and 'b' (see [48]). This is sometimes called the "totally reflecting", sometimes the "perfectly conducting" case. (See *CREFIN* for additional information). Default is FALSE.
ANYANG [scalar]	(Optional) If TRUE, any angles whatsoever may be input through *XMU*. If FALSE, the angles are monotone increasing and mirror symmetric about 90 degrees (this option is advantageous because the scattering amplitudes *S1* , *S2* for the angles between 90 and 180 degrees are evaluable from symmetry relations, and hence are obtained with little added computational cost.) Default is TRUE.
XMU [vector/scalar]	If *ANYANG* is TRUE, cosines of angles at which *S1* , *S2* are to be evaluated. If *ANYANG* is FALSE, then *XMU* can be the number of angles to use between 0 and 180 degrees inclusive (set to zero to skip computation of *S1* and *S2*) or the angles at which to compute but must be mirror symmetric about 90 degrees. If *ANYANG* is false and *XMU*<=1, *XMU* will be interpreted as the angle cosine, *ANYANG* will be reset to true and *S1* and *S2* will be computed at the single angle acos(*XMU*).
MIMCUT [scalar]	(Optional) Positive value below which imaginary refractive index is regarded as zero (computation proceeds faster for 0 imaginary index). Default is 0.
NMOM [scalar]	Highest Legendre moment *PMOM* to calculate, numbering from zero (*NMOM*=0 prevents calculation of *PMOM*).

IPOLZN [scalar]	Specifies which Legendre moments to compute. If POSITIVE, compute the Legendre moments *PMOM* for the Mueller matrix elements determined by the digits of *IPOLZN*, with 1 referring to M1, 2 to M2, 3 to S21, and 4 to D21. E.g., if *IPOLZN*=14 then only moments for M1 and D21 will be returned, if *IPOLZN*=1234 all four moments will be returned. If 0, compute Legendre moments *PMOM* for the unpolarized unnormalized phase function. If NEGATIVE, compute Legendre moments *PMOM* for the Sekera phase quantities determined by the digits of ABS(*IPOLZN*), with 1 referring to R1, 2 to R2, 3 to R3, and 4 to R4. E.g., if *IPOLZN*=-14 then only moments for R1 and R4 will be returned, if *IPOLZN*=-1234 all four moments will be returned.

The outputs are:

QEXT [scalar]	Extinction efficiency factor.
QSCA [scalar]	Scattering efficiency factor.
GQSC [scalar]	Asymmetry factor times scattering efficiency (allows calculation of radiation pressure efficiency factor QPR=*QEXT* - *GQSC*).
SFORW [scalar]	Complex forward-scattering amplitude *S1* at 0 degrees, *S2*(0)=*S1*(0).
SBACK [scalar]	Complex backscattering amplitude *S1* at 180 degrees, *S2*(180)=-*S1*(180).
S1 [vector]	Complex Mie scattering amplitude for s-pol (perpendicular to the scattering plane) at angles specified.
S2 [vector]	Complex Mie scattering amplitude for p-pol (parallel to the scattering plane) at angles specified.
TFORW [vector]	Complex values of: *TFORW*(1)=(*S2* - (MU)∗*S1*) / (1 - MU∧2) and *TFORW*(2)=(*S1* - (MU)∗*S2*) / (1 - MU∧2) at angle theta=0 (MU=COS(theta)=1), where the expressions on the right-hand side are indeterminate. (These quantities are required for doing polarized radiative transfer.).
TBACK [vector]	Complex values of: *TBACK*(1)=(*S2* - (MU)∗*S1*) / (1 - MU∧2) and *TBACK*(2)=(*S1* - (MU)∗*S2*) / (1 - MU∧2) at angle theta=180 (MU=COS(theta)=-1).
PMOM [matrix]	Moments M=0 to *NMOM* of unnormalized NP-th phase quantity PQ moments with M>2∗NTRM are zero, where NTRM=number of terms in Mie series.

SPIKE [scalar] — Magnitude of the smallest denominator of either Mie coefficient (a-sub-n or b-sub-n), taken over all terms in the Mie series past N=size parameter *XX*. Values of *SPIKE* below about 0.3 signify a ripple spike, since these spikes are produced by abnormally small denominators in the Mie coefficients (normal denominators are of order unity or higher). Defaults to 1.0 when not on a spike. Does not identify all resonances.

3.4.3.34 MODFAS

Syntax

Model = MODFAS(*h, [ModelType], [lambda], [AtmModel], [IHAZE], [param]*)

```
SCALING_CODE_API int MODFAS(char** errorChain, const double* h,
     const int nH, const char* modelType, const char* lambda,
     const char* atmModel, const char* iHaze, const double param,
     double* modelValue)
```

This function returns model data for absorption, scattering, or total extinction coefficients as a function of altitude *h* for a given wavelength and atmospheric condition determined by parameters used in MODTRAN and FASCODE. This function uses a look-up table from data obtained by running the ATMTools functions MODTRAN and FASCODE. Models are valid from 0 km to 50 km altitude. Supported wavelengths are '532', '1000', '1030', '1060', '1064', '1080', '1300', '1315', '1550', '1620', '2700', and '3800'. The Matlab inputs are:

h [vector] — Altitude above sea level (m).

ModelType [string] — (Optional, defaults to 'A') Identifier for the type of model desired. Supports 'A' (absorption), 'S' (scattering), 'E' (total extinction) model options, 'AA' (aerosol absorption), 'MA' (molecular absorption), 'AS' (aerosol scattering), and 'MS' (molecular scattering).

lambda [string] — (Optional) Identifier for desired laser wavelength. Defaults to '1315'

AtmModel [string] — (Optional) Parameter from MODTRAN/FASCODE that identifies atmospheric conditions. Takes values as follows (numbers may also be input as strings): 'Trop' (1)–Tropical Atmosphere, 'MidLatSum' (2)–Midlatitude Summer, 'MidLatWint' (3)–Midlatitude Winter, 'SubArcSum' (4)–Subarctic Summer, 'SubArcWint' (5)–Subarctic Winter, 'USStd' (6)–1976 U.S. Standard (default), 'Rural'–(*AtmModel* 6 ('USStd') and *IHAZE* 1 ('Clear')) Based on AFRL/DE model, 'RuralHazy'–(*AtmModel* 6 ('USStd') and *IHAZE* 2 ('Hazy')) Based on AFRL/DE model, 'Maritime'–(*AtmModel* 6 ('USStd') and *IHAZE* 4 ('Maritime')) Based on Boeing Western Pacific model.

IHAZE [string]	(Optional) Parameter from MODTRAN/FASCODE that identifies desired condition for aerosols model. Supported values are as follows (numbers may also be input as strings): 'Clear' (1)–RURAL Extinction, VIS=23 km (Clear) (default), 'Hazy' (2)–RURAL Extinction, VIS=5 km (Hazy), 'NavyMaritime' (3)–NAVY MARITIME Extinction, 'Maritime' (4)–MARITIME Extinction, VIS=23 km, 'Urban' (5)–URBAN Extinction, VIS=5, 'Desert' (10)–DESERT Extinction km.
param [scalar]	(Optional) Additional input parameter option based on the input *IHAZE* value. Wind – Wind speed. Only used with Navy Maritime *IHAZE* = 3 or Desert *IHAZE* = 10. Use 0 or leave empty to use the default wind speeds in MODTRAN and FASCODE based on *AtmModel*. Other options are 5 m/s or 10 m/s, VIS – Visibility. Only used with Urban *IHAZE* = 5. Options are 5 or 23 km. Default is 5 km.

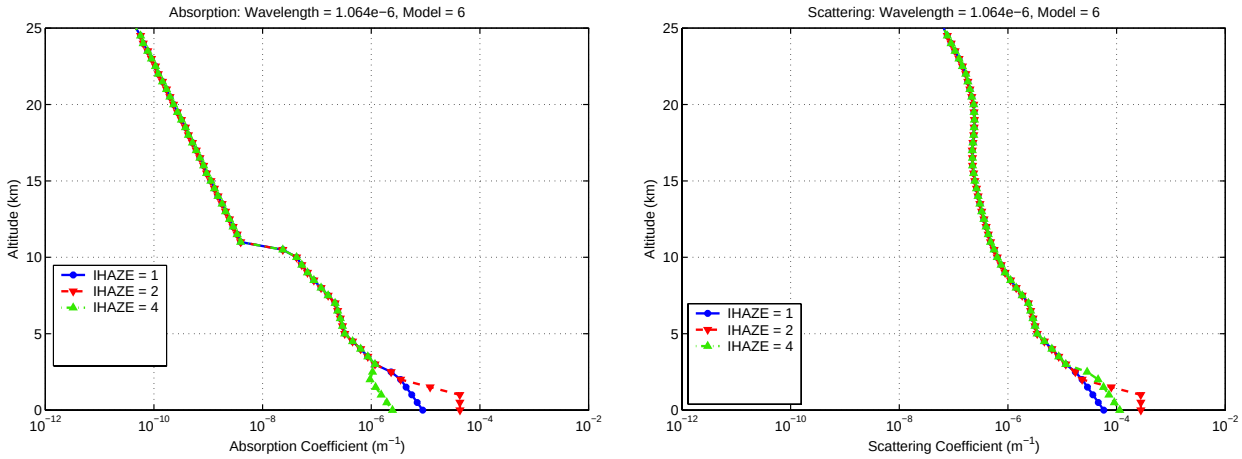
The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
modelType [in]	(NULL OK) Identifier for the type of model desired; supports "a" (absorption), "s"(scattering), "e" (total extinction), "aa" (aerosol absorption), "ma" (molecular absorption), "as" (aerosol scattering), and "ms" (molecular scattering); must be lowercase; default is "a".
lambda [in]	(NULL OK) Identifier for desired laser wavelength; supports "1064" for YAG, "1315" for COIL, "532", "1000", "1030", "1060", "1080", "1300", "1550", "1620", "2700", and "3800"; defaults to "1315".

atmModel [in]	(NULL OK) Parameter from MODTRAN/FASCODE that identifies atmospheric conditions; takes values as follows: "trop" for tropical atmosphere, "midlatsum" for Midlatitude Summer, "midlatwint" for Midlatitude Winter, "subarcsum" for Subarctic Summer, "subarcwint' for Subarctic Winter, "usstd" for 1976 U.S. Standard, "rural" overrides *atmModel* and *iHaze* as "usstd" and "clear" based on AFRL/DE model, "ruralhazy" overrides *atmModel* and *iHaze* as "usstd" and "hazy" based on AFRL/DE model, and "maritime" overrides *atmModel* and *iHaze* as "usstd" and "maritime" based on Boeing Western Pacific model; default is "usstd"; must be all lowercase.
iHaze [in]	(NULL OK) Parameter from MODTRAN/FASCODE that identifies desired condition for aerosols model; supported values include: "clear" for rural extinction and 23 km visibility, "hazy" for rural extinction and 5 km visibility, "navymaritime" for navy maritime extinction, "maritime" for maritime extinction and 23 km visibility, "urban" for urban extinction and 5 km visibility; default is "clear"; must be all lowercase.
param [in]	An additional input parameter option based on the input *iHaze* value; is wind speed when used with *iHaze*="navymaritime", must use either 0, 5, or 10 (m/s); is visibility when used with *iHaze*="urban", in this case options are 5 or 23 km; for other *iHaze* conditions this value will be ignored.
modelValue [out]	Atmospheric model value for desired *modelType* (i.e. absorption, scattering, or extinction) (1/m); must be length *nH*.

The output is a vector *Model* the size of *h* of the desired extinction coefficients as a function of altitude in units of 1/m. Look-up table data is stored in 'ModFasData.mat' and is obtained as follows (see Section 3.4.3.36 or 3.4.3.20 for explanation of parameters):

- Aerosols absorption and scattering are obtained by running `MODTRAN` with the desired `lambda`, `MODEL`, and `IHAZE`. For `IHAZE` = 3, `MODTRAN` was run with `ICSTL` (air mass character) = 5 and wind speeds of 0 (for `MODTRAN` default), 5, and 10 m/s.
- Running `FASCODE` with the desired `lambda`, `MODEL`, and `IHAZE` will yield total extinction. For `IHAZE` = 3, `FASCODE` was run with `ICSTL` (air mass character) = 5 and wind speeds of 0 (for `FASCODE` default), 5, and 10 m/s.
- Setting `IHAZE` = 0 will yield molecular extinction from `FASCODE`.
- Running `FASCODE` with `IHAZE` = 0 and `IAERSL` = 0 will return molecular absorption.
- Molecular scattering is obtained by subtracting molecular absorption from molecular extinction.
- Total absorption is obtained by adding aerosols absorption and molecular absorption.
- Similarly, total scattering is obtained by adding aerosols scattering and molecular scattering.



(a) (b)

Figure 3.25: (a) Absorption and (b) scattering coefficients with altitude for 1.064 μm from MODFAS.

Figures 3.25 and 3.26 show the absorption and scattering coefficients for 1.064 μm and 1.31521 μm, respectively, for one model, *MODEL* = 6 (1976 US Standard), and three different *IHAZE* values. Results for other models are similar.

The following is an example:

```
>> h = [0:1000:5000];
>> Scat = MODFAS(h,'S','1064','MidLatSum','Hazy')

Scat =

  1.0e-003 *

    0.2926    0.2813    0.0232    0.0116    0.0064    0.0034
```

Returns a vector of the size of h for the scattering coefficients as a function of h for 1.064 μm Midlatitude Summer Hazy atmosphere. The same result could be obtained by typing `Scat = MODFAS(h,'S','1064',2,2)`.

3.4.3.35 MODHUFNAGELVALLEY

Syntax

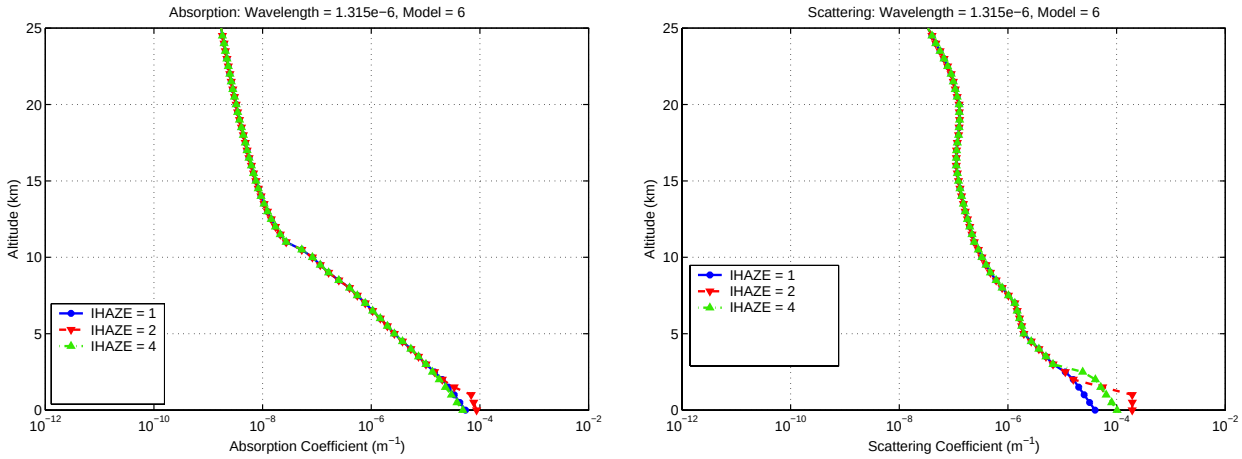
Cn2 = MODHUFNAGELVALLEY(*h*)

```
SCALING_CODE_API int MODHUFNAGELVALLEY(char** errorChain,
    const double* h, const int nH, double* cn2)
```

This function returns the value of the Modified Hufnagel-Valley turbulence profile given altitude above sea level *h* in meters [14]. The only Matlab input is *h* and the output *Cn2* is a vector the size of *h* with the values of C_n^2 with units of $\mathrm{m}^{-2/3}$. The equation used to calculate C_n^2 is as follows:

$$C_n^2 = 8.16 \times 10^{-54} h^{10} \exp\left(\frac{-h}{1000}\right) + 3.02 \times 10^{-17} \exp\left(\frac{-h}{1500}\right) + 1.9 \times 10^{-15} \exp\left(\frac{-h}{100}\right)$$

The API function returns system error codes and the arguments are:



(a) (b)

Figure 3.26: (a) Absorption and (b) scattering coefficients with altitude for 1.31521 μm from `MODFAS`.

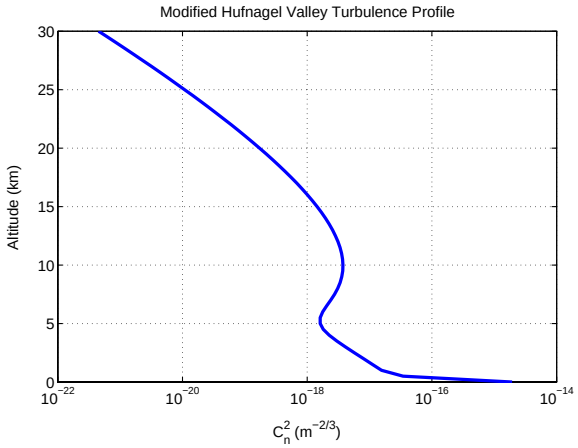


Figure 3.27: Turbulence profile calculated using the Modified Hufnagel Valley model.

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

Figure 3.27 shows the values of C_n^2 as calculated using Modified Hufnagel Valley model.

3.4.3.36 `MODTRAN`

Syntax

```
ExtCoeff = MODTRAN(Atm, [FileName], [FieldName1, FieldValue1], ...
    [FieldName2, FieldValue2], ...)
```


This function runs the `FORTRAN` executable `MODTRAN` and returns a structure that has the extinction coefficient and the transmission. MODTRAN is not distributed with ATMTools. The user must obtain MODTRAN

Atm [struct]	Atmospheric modeling parameters for a discrete atmosphere. Can be a structure from AtmStruct with the fields hp, ht, L, h, z.
FileName [string]	(Optional, defaults to 'MOD_NORM') The file name to be used in MODTRAN. Users can input one of their files or use one from the library (Input_library) directory. Must be passed in (can be empty) if using optional arguments that follow.
FieldName [string]	Name of parameter to be modified from the value in *FileName*. Available parameters are: 'LAMBDA'–Wavelength (m), 'MODEL'–Atmospheric model (1-6), 'IHAZE'–Aerosols model to be used (1-10), 'IVULCN'–Extinction type for stratospheric aerosols (1-8), 'ICLD'–inclusion of clouds (1-20), 'ITYPE'–Type of path (1, 2 (default)), 'HRANGE'–Horizontal path length, *ITYPE*=1 (m), 'ICSTL'–Air mass character, *IHAZE*=3 (1-10), 'WSS'–Current wind speed, *IHAZE*=3 or 10, 'WHH'–24-hr average wind speed, *IHAZE*=3 or 10, 'VIS'–Surface meteorological range (km), 'MOD_EXE'–Filename of the MODTRAN executable file. Must be in the ATMToolsPath/OTT/MODTRAN folder. Defaults to MODTRAN.EXE.
FieldValue [scalar]	(Required if specifying *FieldName*) Value of specified *FieldName* with units depending on *FieldName*.

The output is

ExtCoeff [struct]	If Lambda is one of the inputs *FieldName*, then *ExtCoeff* is a structure containing the extinction coefficients and information about the run parameters. Otherwise, *ExtCoeff* is a 3 dimensional tensor where the columns of each matrix *ExtCoeff*(:, :, i) represents a component of extinction as a function of wave number. The columns are: 1 - Total Extinction, 2 - Total Molecular, 3 - Molecular Absorption, 4 - Molecular Scattering, 5 - Total Aerosols, 6 - Aerosols Absorption, 7 - Aerosols Scattering.
ExtCoeff.ver [string]	Version of MODTRAN run.
ExtCoeff.Name [struct]	The name of each parameter included in the structure.
ExtCoeff.Value [struct]	In which the values of the parameters are stored.
ExtCoeff.Units [struct]	The units of this parameter.

FieldNames are MODTRAN parameters. If the user specifies any of these to be changed, the output substructures will have a field corresponding to that parameter. The substructures also have vector fields for altitudes,

total extinction, molecular extinction, aerosols extinction, molecular and aerosols absorption and molecular and aerosols scattering.

3.4.3.37 NSLOT_v2

Syntax

[cnsq, ustar, tstar, qstar, l, data_flag] = NSLOT_v2(*zref, u, tair, ... tsea, rh, pr, zu, zt, zq, zp, lambda*)

This function is a bulk model for computing surface-layer scaling parameters, and the refractive index structure parameter from mean values of meteorological quantities measured near the ocean surface.[49] The inputs are

zref [vector]	Reference height above surface to compute *cnsq* (m). Returns -99 if *zref* < 1m or *zref* > 50m.
u [vector]	Wind speed (m/s). Returns -99 if *u* < 0.1m/s or *u* > 30m/s.
tair [vector]	Air temperature (deg C). Returns -99 if *tair* < -60C or *tair* > 60C.
tsea [vector]	Sea surface temperature (deg C). Returns -99 if *tsea* < -2C or *tsea* > 40C.
rh [vector]	Air relative humidity (%). Returns -99 if *rh* < 0% or *rh* > 100%.
pr [vector]	Atmospheric pressure (mb) (default is 1013.25). Resets to 1013.25 if *pr* < 940mb or *pr* > 1050mb.
zu [vector]	Wind speed measurement height (m). Returns -99 if *zu* < 1m or *zu* > 50m.
zt [vector]	Air temperature measurement height (m). Returns -99 if *zt* < 1m or *zt* > 50m.
zq [vector]	Relative humidity measurement height (m). Returns -99 if *zq* < 1m or *zq* > 50m.
zp [vector]	Atmospheric pressure measurement height (m) (default is 10). Resets to 10m if *zp* < 1m or *zp* > 50m.
lambda [scalar]	Optical wavelength for computing *cnsq* (microns). Must be in the range 0.36 to 4.0 or 7.8 to 19.0 microns, returns -99 if *lambda* outside this range.

The wind speed, air temperature and air humidity must be measured between 1 and 50 m above the ocean surface. The model will still run if either the pressure or pressure measurement height inputs are bad (*data_flag* <= 3). In such cases the pressure is defaulted to 1013.25 mb and the pressure measurement height is defaulted to 10 m. The model will not run if any other inputs besides the pressure and/or pressure measurement height are bad (*data_flag* > 3). All output parameters (except *data_flag*) are set to -99 if any input data besides

pressure or pressure height are bad. All output parameters (except *data_flag*) are set to -98 if the bulk model (bulk_params subroutine) does not converge.

The outputs are

cnsq [vector]	Refractive index structure parameter at *zref* ($m^{-2/3}$).
ustar [vector]	Bulk wind speed scaling parameter (m/s).
tstar [vector]	Bulk potential temperature scaling parameter (K).
qstar [vector]	Bulk specific humidity scaling parameter (g/g).
l [vector]	Obukhov length (m).
data_flag [vector]	0 if all input data are good, 1 if input pressure is bad, 2 if input pressure height is bad, 3 if input pressure and height are both bad, 4 or more if any input data other than pressure/height are bad.

3.4.3.38 OceanAtmos

Syntax

out = OceanAtmos(*h, model, lambda, Location, [season], [ANAMflag]*)

This function generates absorption and scattering coefficients as well as temperature and pressure using the LEEDR oceanic database and the Advanced Navy Aerosol Model (ANAM) or the Navy Aerosol Model (NAM). The user must have a MATLAB® version (m-code or p-code) on the MATLAB® path to use this function. It will generate the proper structure for use in LEEDRAtmos based on the provided inputs. The inputs are

h [vector]	Altitude (m).
model [string/cell]	'A' for absorption, 'S' for scattering, 'T' for temperature, 'P' for pressure, 'R' for Relative humidity, 'D' for dewpoint, 'AA' for aerosol absorption, 'AS' for aerosol scattering, 'MA' for molecular absorption, and 'MS' for molecular scattering. Can be a cell array of models in which case the output can be a single matrix with the appropriate data or each *model* can be output into separate variables, see the examples below. If passed in as [], output will be a structure with all *model* data.
lambda [string]	Identifier for desired laser wavelength. See Table 3.3
location [vector]	[Lat Lon] of ocean site. If not a valid site, a warning is issued and standard meteorological parameters are used.
season [scalar]	(Optional) Index to *season*: 1 - summer (default), 2 - winter.

ANAMflag [logical]	(Optional) If true, ANAM *model* is used, otherwise, NAM is used. The default is true.

The output is

out [vector]	Absorption or scattering coefficient (1/m), temperature (K), pressure (Pa), rel humidity (%), or dewpoint (K).

Example 3.4.38 (OceanAtmos) *Examples using* OceanAtmos *to compute atmospheric model data.*

```
>> h = linspace(10,1000,10)';
>> Abs = OceanAtmos(h,'A','1315',[0 0],1)
```

Compute absorption for the specified inputs - must have p-code or m-code version of LEEDR on the path.

```
>> data = OceanAtmos(h,{'A' 'T'},'1315',[0 0],1)
```

Returns an Nx2 matrix with absorption in column 1 and temperature in column 2.

```
>> data = OceanAtmos(h,[],'1315',[0 0],1)
```

Returns a structure with all of the models (including a breakout of aerosol and molecular absorption and scattering).

```
>> Atm = AtmStruct(GeomStruct,100,'Abs','OceanAtmos','A','1315',[0 0],...)
```

Generate an Atm *structure using the Marine data.*

3.4.3.39 OuterScale

Syntax

Lout = OuterScale(*h, [mVal]*)

Lout = OuterScale(*h, [mType]*)

```
SCALING_CODE_API int OuterScale(char** errorChain,
     const char modelType, const int nH, const double* h, const double* m,
     double* Lout)
```

This function computes an outer scale proportional to altitude above sea level. The Matlab inputs are:

h [vector]	Altitude above sea level (m).
mVal [scalar/vector]	(Optional) Multiplier on altitude.
mType [char]	(Optional) Model for outer scale. Currently only supports 'Physical' specifies physical model of outer scale.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
modelType [in]	Can be 'p'/'P' to specify the physical model of outer scale; anything else just multiplies *m* and *h*.
nH [in]	The number of elements in the array *h*.
h [in]	Altitude above sea level (in meters); must be length *nH*.
m [in]	(NULL OK) The multiplier on altitude (if NULL, then 1); if input must be length *nH*.
Lout [out]	The outer scale for turbulence (in meters); allocate to length *nH*.

The multiplier, *mVal*, can be a vector with the same length of *h*. This could be useful since *h* is altitude above sea level and the outer scale may be more related to the altitude above ground. If only altitude is input, the function will compute outer scale using the physical model. The output is

Lout [vector]	Outer scale for turbulence (m).

If *mType* is specified as 'Physical' outer scale is computed using the equation below.

$$L_{out} = \begin{cases} 0.4h & h \leq 270 \\ 0.4 \times 270 e^{(-h/15000)} & h > 270 \end{cases} \tag{3.10}$$

3.4.3.40 processAIRShdf

Syntax

AIRSdata = processAIRShdf(*[startTime], [endTime], [bbox], HDFname, procStruct*)

This function can download AIRS data, save to HDF file and optionally process and save to a MAT file. Data can be downloaded from http://mirador.gsfc.nasa.gov/. Using this function to download data requires functions that are not distributed with ATMTools. The inputs are

startTime [varied]	(Optional) Start time for data to download in any MATLAB® date format (in UTC).
endTime [varied]	(Optional) End time for data to download (in UTC).
bbox [vector]	(Optional) Bounding box for data to download [west *lon*, south *lat*, east *lon*, north *lat*].
HDFname [string]	File name for HDF data file of AIRS data (without extension). If *startTime*, *endTime*, and *bbox* are not input, assumes *HDFname* exists and contains data that has already been downloaded and needs to be processed.
procStruct [struct]	(Optional) Structure with additional information for the processing. May have the following fields: MATname [string] - File name for saving processed AIRS data, Lo [scalar] - (Optional) Assumed outer scale, defaults to 150 m, KH_KM [scalar] - (Optional) Assumed mixing ratio, defaults to 0.01.

The output is a structure of data, some of which is taken directly from the AIRS data file and some is computed. The processed data is essential a lookup table that can be used with AIRS to compute model data as a function of altitude. The output is described below.

AIRSdata [struct]	Structure of data obtained from the AIRS data file.
AIRSdata.lat [array]	Latitude for each observation point (deg).
AIRSdata.lon [array]	Longitude for each observation point (deg).
AIRSdata.alt [vector]	Altitudes of midpoints of pressure layers (m).
AIRSdata.F [cell]	Delauney triangulation for each mid-point layer.
AIRSdata.alt_agl [array]	Altitude at mid-point of each pressure layer (m).
AIRSdata.Temp [array]	Temperature at the mid-point altitudes (K).
AIRSdata.PotentialTemp [array]	Potential temperature at the mid-point altitudes (K).
AIRSdata.Cn2 [array]	C_n^2 values for each *lat*/*lon* ($m^{-2/3}$).
AIRSdata.Press_midpt [vector]	Pressure at mid-point altitudes (mbar).
AIRSdata.dTdz [array]	Temperature gradient (K/m).
AIRSdata.time [array]	Time for observation (sec since Jan 1, 1993).
AIRSdata.JulianDate [array]	Julian data for each observation.

AIRSdata.sat_lat [array]	Latitude of the satellite (deg).
AIRSdata.sat_lon [array]	Longitude of the satellite (deg).
AIRSdata.sat_alt [array]	Altitude of the satellite (m).
AIRSdata.airs_lat [array]	Ground latitude of the observations (deg).
AIRSdata.airs_lon [array]	Ground longitude of the observations (deg).
AIRSdata.airs_height [array]	Altitude of each observation (m).
AIRSdata.Press [array]	Measured pressure for each layer (mbar).
AIRSdata.Cn2Processing [struct]	Structure of information used for *Cn2* processing.
AIRSdata.CloudHeight [array]	Cloud top heights for each *lat*/*lon* (m).
AIRSdata.CloudLat [array]	Latitude at cloud top heights (deg).
AIRSdata.CloudLon [array]	Longitude at cloud top heights (deg).
AIRSdata.CloudInterpIDX [vector]	Indicates where *AIRSdata.CloudHeight* has been interpolated.
AIRSdata.CloudInterpolant [scatteredInterpolant]	Scattered interpolant for interpolating cloud height as a function of *lat* and *lon*.
AIRSdata.hdf [char]	Name of HDF file.

3.4.3.41 PathDisp

Syntax

s = PathDisp(*G, Atm, DispType, p1, p2, ...*)

```
SCALING_CODE_API int PathDisp(char** errorChain, const char* dispType,
     const int nInputs, const double* posPlatEcf,
     const double* posTargEcf, const double* velPlatEcf,
     const double* velTargEcf, const double* re, const double* hp,
     const double* ht, const double* rd, const double* l,
     const int nScreens, const double* z, const double* windHeading,
     const double* horzWindSpeed, const double* vertWindSpeed,
     const double* p1, const double* p2, const double* p3,
     const double* p4, double* disp)
```

PathDisp computes the displacement of a beam along a propagation path based upon the input displacement type *DispType* and accompanying parameters. The Matlab inputs are

G [struct]	Platform/target geometry structure from GeomStruct
Atm [struct]	Atmospheric modeling structure from AtmStruct†. Must be a discrete atmospheric model.
DispType [string]	Identifier for type of beam displacement to model.
p1...pN [list]	(Optional) Input parameters to accompany *DispType* identifier.

The input parameter list depends on the input *DispType*.

'ANGULAR' Constant angular offset projected toward target

theta [scalar]	Angular separation of beams (rad).
phi [scalar]	Orientation of separation at target measured counter-clockwise from P axis (rad).

'APERTURE' Apertures with fixed offset directed to same target point

d [scalar]	Separation of aperture centers (m).
beta [scalar]	Orientation of separation of aperture centers measured counter-clockwise from P axis (rad).

'APERTURE-ANGULAR' Aperture separation and angular offset together

d [scalar]	Separation of aperture centers (m).
beta [scalar]	Orientation of separation of aperture centers measured counter-clockwise from P axis (rad).
theta [scalar]	Angular separation of beams (rad).
phi [scalar]	Orientation of separation at target measured counter-clockwise from P axis (rad).

'CHROMATIC' Beams of different wavelength directed to same target point

lambda1 [scalar]	Wavelength of beam 1 (m).
lambda2 [scalar]	Wavelength of beam 2 (m).

'CHROMATIC-ANGULAR' Different wavelength with angular offset

lambda1 [scalar]	Wavelength of beam 1 (m).
lambda2 [scalar]	Wavelength of beam 2 (m).
theta [scalar]	Angular separation of beams (rad).
phi [scalar]	Orientation of separation at target measured counter-clockwise from P axis (rad).

'TIME-OF-FLIGHT' Round-trip propagation time multiplied by beam clearing velocity. This option requires no additional parameters.

'TIME-DELAY' Compensation for anisoplanatic delay for a given delay.

tau [scalar]	The delay time at the target from position A to B.

If a vector is used, then it is assumed that the length of *tau* equals the length of *G*. Or, if a mismatch occurs then only *G*(1) will be used with the vector *tau*.

The Matlab output is

s [matrix]	Vector displacement along the propagation path. Number of rows in *s* equals number of screen positions in *Atm*. Column 1 is displacement in P axis, column 2 is displacement in T axis, i.e., *s*=[*s*_P, *s*_T].

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
dispType [in]	Identifier for type of beam displacement to model; current valid identifiers are: "ANGULAR" - Constant angular offset projected toward target, "APERTURE" - Apertures with fixed offset directed to same target point, "APERTURE-ANGULAR" - Aperture separation and angular offset together, "CHROMATIC" - Beams of different wavelength directed to same target point, offset due to atmospheric dispersion, "CHROMATIC-ANGULAR" - Dispersion combined with angular offset, "TIME-OF-FLIGHT" - Round-trip propagation time multiplied by beam clearing velocity, "TIME-DELAY" - Compensation for anisoplanatic delay for a given delay tau; *dispType* is an all-capitols null-terminated string, and there can only be one selected *dispType* (i.e. not *nInputs dispType*s).

nInputs [in]	The number of inputs into this function, repeated one after the other in memory; also corresponds to the number of outputs.
posPlatEcf [in]	ECF position vector(s) for platform (m); must be length 3 x *nInputs*.
posTargEcf [in]	ECF position vector(s) for target (m); must be length 3 x *nInputs*.
velPlatEcf [in]	ECF velocity vector(s) for platform (m); must be length 3 x *nInputs*.
velTargEcf [in]	ECF velocity vector(s) for target (m); must be length 3 x *nInputs*.
re [in]	Spherical Earth radius (m); must be length *nInputs*.
hp [in]	Altitude of transmit/receive platform (m); must be length *nInputs*.
ht [in]	Altitude of target (m); must be length *nInputs*.
rd [in]	Downrange of target along curved earth surface (m); must be length *nInputs*.
l [in]	Slant range from platform to target (m); must be length *nInputs*.
nScreens [in]	Number of phase screens specified for each input.
z [in]	Phase screen distances from transmitter (m); must be length *nScreens* x *nInputs*.
windHeading [in]	(NULL OK unless *dispType*="TIME-OF-FLIGHT" or "TIME-DELAY") Natural wind heading for the phase screens (degrees from North); must be *nScreens* x *nInputs*.
horzWindSpeed [in]	(NULL OK unless *dispType*="TIME-OF-FLIGHT" or "TIME-DELAY") Natural wind speed values for the phase screens (m/s); must be *nScreens* x *nInputs*.
vertWindSpeed [in]	(NULL OK unless *dispType*="TIME-OF-FLIGHT" or "TIME-DELAY") Natural wind speed values orthogonal to North-South plane and in "up" direction (m/s); must be *nScreens* x *nInputs*.
p1 [in]	(NULL OK for *dispType*="TIME-OF-FLIGHT"); if input must be length *nInputs*; for "ANGULAR", the angular separation of beams (rad); for "APERTURE", separation of aperture centers (m); for "APERTURE-ANGULAR", separation of aperture centers (m); for "CHROMATIC", wavelength of beam 1 (m); for "CHROMATIC-ANGULAR", wavelength of beam 1 (m); for "TIME-OF-FLIGHT", ignored; for "TIME-DELAY", the delay time at the target from position (a) to (b).

p2 [in]	(NULL OK for *dispType* = "TIME-OF-FLIGHT" and "TIME-DELAY"); if input must be length *nInputs*; for "ANGULAR", orientation of separation at target measured counter-clockwise from P axis (rad); for "APERTURE", orientation of separation of aperture centers measured counter-clockwise from P axis (rad); for "APERTURE-ANGULAR", orientation of separation of aperture centers measured counter-clockwise from P axis (rad); for "CHROMATIC", wavelength of beam 2 (m); for "CHROMATIC-ANGULAR", wavelength of beam 2 (m); for "TIME-OF-FLIGHT", ignored; for "TIME-DELAY", ignored.
p3 [in]	(NULL OK unless *dispType* = "APERTURE-ANGULAR" or "CHROMATIC-ANGULAR"); if input must be length *nInputs*; for "ANGULAR", ignored; for "APERTURE", ignored; for "APERTURE-ANGULAR", orientation of separation at target measured counter-clockwise from P axis (rad); for "CHROMATIC", ignored; for "CHROMATIC-ANGULAR", Angular separation of beams (rad); for "TIME-OF-FLIGHT", ignored; for "TIME-DELAY", ignored.
p4 [in]	(NULL OK unless *dispType* = "APERTURE-ANGULAR" or "CHROMATIC-ANGULAR"); if input must be length *nInputs*; for "ANGULAR", ignored; for "APERTURE", ignored; for "APERTURE-ANGULAR", orientation of separation of aperture centers measured counter-clockwise from P axis (rad); for "CHROMATIC", ignored; for "CHROMATIC-ANGULAR", orientation of separation at target measured counter-clockwise from P axis (rad); for "TIME-OF-FLIGHT", ignored, for "TIME-DELAY", ignored.
disp [out]	Vector displacement along the propagation path in target parallel/transverse coordinates (m); stored [p1, t1, p2, t2, ..., pN, tN] where N is the number of screens; repeats for *nInputs*; size must be allocated to 2 x *nScreens* x *nInputs*.

3.4.3.42 RandCn2

Syntax

Cn2 = RandCn2(*Atm, [N_Prof], [sigma], [thresh], [fixParam]*)

This function returns a random C_n^2 profile based on the input model, *baseModel*, with the values specified by *fixParam* constrained to be equal to the computed values from the *baseModel* profile. The inputs are

Atm [structure]	Atmosphere structure from ATMSTRUCT or a comma-separated list of (*h*, *x*, *dx*, *L*, *baseModel*, ...).
h [vector]	Altitude above sea level (m).
x [vector]	Normalized screen locations along the propagation path.
dx [vector]	Normalized screen thicknesses along the propagation path.
L [scalar]	Slant range of propagation (m).
baseModel [string/fcnHandle]	Base turbulence profile model. If *baseModel* has more than a single input argument, must be passed in as 'ModelName(*h*, P1, ...)'. Note that this may not work for all *Cn2* models in ATMTools depending on the additional arguments required by the model.
Nprof [scalar]	(Optional) Number of random profiles to generate, defaults to 1.
sigma [scalar]	(Optional) Standard deviation of random vector - related to the standard deviation of the *Cn2* profiles generated (by default this function will use the largest integer value that returns *Nprof* "Good" profiles). Cannot be passed in unless *Nprof* is also passed. Pass as empty ([]) to use default setting.
thresh [scalar]	(Optional) Threshold for the normalized *Cn2* values. Values of normalized *Cn2* less than threshold are not varied. Default is 0.01. Cannot be passed unless *Nprof* and *sigma* are passed.
fixParam [string]	String identifiers for *parameters* to be fixed across random profiles.

The outputs are

Cn2 [struct/vect]	An array of structures where each entry is a random profile generated using the profile in the first entry, or a vector of profiles *Cn2* values if called by ATMSTRUCT ($m^{-2/3}$).
P [Matrix]	Parameter matrix 4 by (*Nprof*+1) with the values of [r0 theta0 Rytov M0] for each of the profiles, where r0 is the Fried parameter (m) at 1 micron, theta0 is the isoplanatic angle (rad) at 1 micron, Rytov is the Rytov number at 1 micron, and M0 is the path average *Cn2* ($m^{-2/3}$). If input is a structure this is added as a field in vector form (length of 4).

Example 3.4.39 (RandCn2) *The example illustrates several calls to* RandCn2.

```
>> Atm = AtmStruct(GeomStruct,10);
>> [Cn2, P] = RandCn2(Atm.h,Atm.z/Atm.L,Atm.dz/Atm.L,Atm.L, ...
           'Clear1Night',100,5,0.01)
```

Compute 100 random profiles based on Clear1Night for the phase screen locations in ***Atm***.

```
>> Atm = AtmStruct(GeomStruct,100,'Cn2','RandCn2','x','dx','L', ...
    'Clear1Night',1,[],0.01,'r0','theta0','rytov')
```

Create an ***Atm*** *structure with a random C_n^2 profile based on Clear-1 Night with fixed values for r_0, θ_0, and Rytov.*

```
>> Cn2 = RandCn2(Atm.h,Atm.z/Atm.L,Atm.dz/Atm.L,Atm.L, ...
           'UniformCn2(h,''r0'',.05,1e-6,L,x,dx)','r0')
```

Return a single random C_n^2 profile based on a uniform C_n^2 profile with a specified r_0.

```
>> Atm = AtmStruct(GeomStruct,100,'Cn2','RandCn2','x','dx','L', ...
    'UniformCn2(h,''r0'',.05,1e-6,L,x,dx)', ...
    1,[],0.01,'r0','theta0','rytov')
```

Create an Atm structure with a random C_n^2 profile based on a uniform C_n^2 profile with a specified r_0 having the same r_0, θ_0, and Rytov.

3.4.3.43 RefIndex

Syntax

[n, dn] = RefIndex(*h, lambda*)

```
SCALING_CODE_API int RefIndex(char** errorChain, const double* h,
     const int nH, double lambda, double* n, double* dn)
```

Given an altitude vector, ***h***, this function computes the refractive index, ***n***, and the derivative of the refractive index, ***dn***, at the specified altitudes for the given wavelength, ***lambda***.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages, deepest error is first.
h [in]	Altitude above sea level (m); vector of length ***nH***.
nH [in]	Length of input and output vectors.
lambda [in]	Wavelength of laser (m).
n [out]	Refractive index; length ***nH***.
dn [out]	Derivative of the refractive index with respect to ***h*** (m^{-1}); length ***nH***.

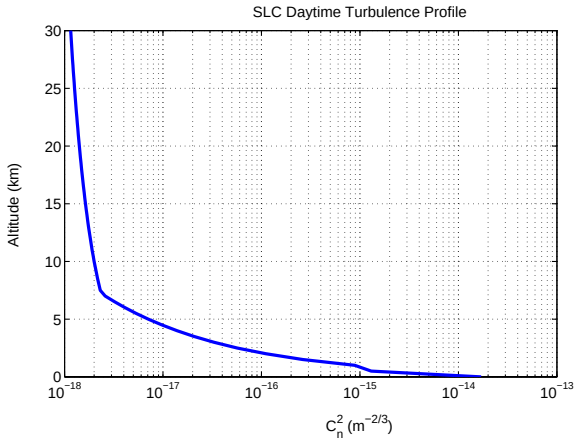


Figure 3.28: Turbulence profile calculated using the SLC daytime model.

3.4.3.44 SLCDay

Syntax

Cn2 = SLCDay(*h*)

```
SCALING_CODE_API int SLCDay(char** errorChain, const double* h,
    const int nH, double* cn2)
```

This function returns the value of the SLC daytime turbulence profile given altitude above sea level *h* in meters [35]. The only input is *h* and the output *Cn2* is a vector the size of *h* with the values of C_n^2 with units of $\mathrm{m}^{-2/3}$. The equation used to calculate C_n^2 is as follows:

$$C_n^2 = \begin{cases} 1.7 \times 10^{-14} & h < 18.5 \\ 3.13 \times 10^{-13} h^{-1.05} & 18.5 \leq h < 240 \\ 1.3 \times 10^{-15} & 240 \leq h < 880 \\ 8.87 \times 10^{-7} h^{-3} & 880 \leq h < 7200 \\ 2 \times 10^{-16} h^{-0.5} & h \geq 7200 \end{cases}$$

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

Figure 3.28 shows the values of C_n^2 as calculated using the SLC daytime model.

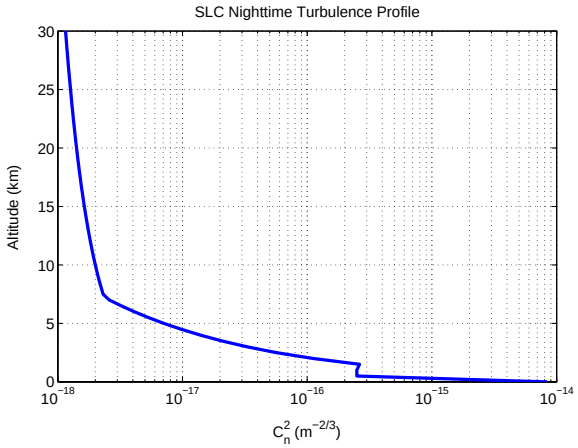


Figure 3.29: Turbulence profile calculated using the SLC nighttime model.

3.4.3.45 SLCNight

Syntax

Cn2 = SLCNight(*h*)

```
SCALING_CODE_API int SLCNight(char** errorChain, const double* h,
    const int nH, double* cn2)
```

This function returns the value of the SLC nighttime turbulence profile given altitude above sea level *h* in meters [35]. The only input is *h* and the output *Cn2* is a vector the size of *h* with the values of C_n^2 with units of $\mathrm{m}^{-2/3}$. The equation used to calculate C_n^2 is as follows:

$$C_n^2 = \begin{cases} 8.4 \times 10^{-15} & h < 18.5 \\ 2.87 \times 10^{-12} h^{-2} & 18.5 \leq h < 110 \\ 2.5 \times 10^{-16} & 110 \leq h < 1500 \\ 8.87 \times 10^{-7} h^{-3} & 1500 \leq h < 7200 \\ 2 \times 10^{-16} h^{-0.5} & h \geq 7200 \end{cases}$$

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

Figure 3.29 shows the values of C_n^2 as calculated using the SLC nighttime model.

3.4.3.46 SOR2MIATM

Syntax

[Atm, G] = SOR2MIATM(*Direction, ModelType, Percentile, Distribution*)

This function returns an atmospheric modeling (*Atm*) structure containing C_n^2 modeling data with 10 path segments for specified conditions. The inputs are

Direction [string]	Direction of laser propagation 'SOR2mi' or '2miSOR' - defaults to 'SOR2mi'.
ModelType [string]	Modeling data for distribution 'ALL' ('A'), 'DAY' ('D'), 'NIGHT' ('N'), 'TRANSITION' ('T') - defaults to 'ALL'.
Percentile [string]	Percentile of C_n^2 distribution: '10%', '50%', '90%' - defaults to '50%'.
Distribution [string]	Distribution assumption: 'LOGNORMAL' or 'GAMMA' - defaults to 'LOGNORMAL'.

The output *Atm* structure will have *Atm.Cn2* values for the specified conditions, and *Atm.Cn2Eval* = 'NoEval'. The output is based upon analysis of differential-tilt turbulence profiler measurements made Sep 2004 at Starfire Optical Range, Kirtland AFB, NM for propagation from SOR to the "2 mile" test site. Also returns the ECF geometry structure for specified path as an optional second output.

Atm [struct]	Atmospheric modeling structure containing Cn2 model.
G [struct]	Geometry structure for specified path.

3.4.3.47 SORWIND

Syntax

v = SORWIND(*h, [season]*)

This function returns the wind profile for Starfire Optical Range (SOR) as a function of altitude and time of year. The inputs are

h [vector]	Altitude above mean sea level (m).
season [string]	(Optional) Season. One of 'spring' (1), 'summer' (2), 'fall' (3), or 'winter' (4). Defaults to 'spring'.

The outputs, *v*, is the wind speed in m/s. Figure 3.30 contains a plot of the wind speed as a function of altitude for different seasons.

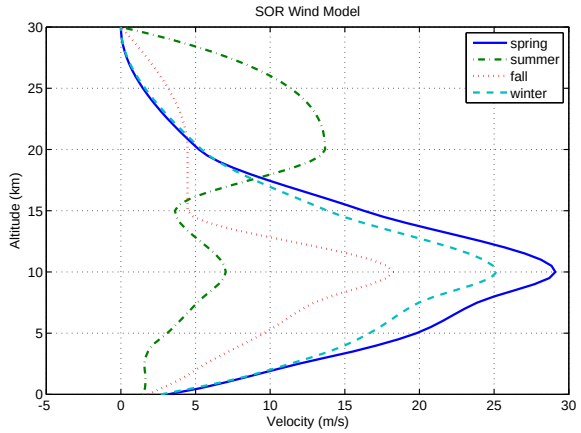


Figure 3.30: SOR wind profile for different seasons

3.4.3.48 SpeedOfSound

Syntax

SoundSpeed = SpeedOfSound(*T*)

[vsPlat vsTarg] = SpeedOfSound(*G, [Atm]*)

SoundSpeed = SpeedOfSound(*Atm*)

```
SCALING_CODE_API int SpeedOfSound(char** errorChain,
    const double* temperature, const int nTemperature,
    double* speedOfSound)
```

This function returns the speed of sound as a function of temperature. The Matlab inputs are:

T [vector]	Temperature (K).
G [struct]	A geometry structure as from GeomStruct to compute the speed of sound at the platform and target altitudes. If *Atm* is also passed in, then the speed of sound is evaluated using the temperature model in the input *Atm*. Otherwise, the temperature model will be from US_Standard76.
Atm [struct]	An atmosphere structure as from AtmStruct. If input with a *G* structure, the temperature model in *Atm* will be used to compute the speed of sound at the platform and target altitudes. If only an *Atm* structure is input, speed of sound will be computed at the phase screen altitudes. If no temperature model is present, US_Standard76 will be used.

Note that the different syntaxes produce different results, as shown in . To return the speed of sound at specific temperature values, input an array of these values directly. Input a geometry structure to return the speed of

sound at the platform and target altitudes using US_Standard76 for temperature or the temperature model in the optional *Atm* structure. If the only input is an *Atm* structure, the speed of sound will be computed at the temperature of the phase screens.

The Matlab outputs are:

SoundSpeed [vector]	Speed of sound (m/s). If an *Atm* struct is entered without a *G*, then this will be evaluated at the points along *Atm.h*.
vsPlat [vector]	Speed of sound at the platform location (if a *G* structure is input as the first argument).
vsTarg [vector]	(Optional) Speed of sound at the target location (if a *G* structure is input as the first argument).

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
temperature [in]	Temperature (K); must be length *nTemperature*.
nTemperature [in]	Length of input and output arrays.
speedOfSound [out]	Speed of sound (m/s); must be length *nTemperature*.

Example 3.4.40 (SpeedofSound) *This example illustrates the different computations done in* SpeedofSound.

```
>> v = SpeedofSound(300);
```

Return speed of sound at specified temperature.

```
>> [vP vT] = SpeedofSound(GeomStruct);
```

Return speed of sound at platform and target altitudes assuming US_Standard76 *for temperature model.*

```
>> v = SpeedofSound(AtmStruct)
```

Compute speed of sound as a function of the temperature at the phase screen altitudes.

The speed of sound in m/s is given by

$$\texttt{SoundSpeed} = \sqrt{\frac{c_p}{c_v}\frac{R}{M_0}T}$$

where c_p is the specific heat of air at constant pressure, c_v is the specific heat of air at constant volume, and the ratio $c_p/c_v = 1.40$. R is the universal gas constant = 8.31432 J/mol·K, and M_0 is the mean molecular weight of atmospheric gases, $M_0 = 28.9644$ g/mol [37].

3.4.3.49 TerrainAtm

Syntax

[Model hh] = TerrainAtm(*h, x, G, BaseModel, [p1], ..., [pN], [dataDir]*)

[Model hh] = TerrainAtm(*h, hGnd, BaseModel, [p1], ..., [pN]*)

This function returns the value of an atmospheric model evaluated for screen altitude above ground level. The inputs are

h [vector]	Altitude above sea level (m).
x [vector]	Normalized screen locations (m).
G [struct]	Geometry structure as from GeomStruct.
hGnd [vector]	Altitude of ground (m).
BaseModel [string]	Atmospheric modeling function on path.
p1,...,pN [arb]	(Optional) Parameters required for evaluating *BaseModel*.
dataDir [string]	(Optional) Full directory location of DTED data files. Defaults to EngagementTools/DTED.

The outputs are

Model [vector]	Evaluated atmospheric model for new altitudes.
hh [vector]	Altitudes above terrain used for evaluating model (m).

Example 3.4.41 has some examples of how to use TerrainAtm. For examples of using TerrainAtm with BoundaryAtm and AverageAtm, refer to the example script NestingModelFunctions in the ATMTools/Examples directory.

Example 3.4.41 (TerrainAtm) *This example illustrates use of* TerrainAtm.

```
>> Model = TerrainAtm(h,1000,'HV57')
```

Compute H-V 5/7 turbulence profile for altitudes above a fixed ground level of 1 km.

```
>> Model = TerrainAtm(h,x,G,'HV57');
```

Compute H-V 5/7 for altitudes above terrain for locations in input G using digital terrain elevation data (DTED) to determine terrain altitude along the path.

```
>> Atm = AtmStruct(G,100,'Cn2','TerrainAtm',1000,'HV57');
```

Create an Atm structure with H-V 5/7 for altitudes above a fixed ground level of 1 km.

```
>> Atm = AtmStruct(G,100,'Cn2','TerrainAtm','x','G','HV57','C:/DTED/MyLoc');
>> Atm = AtmStruct(G,100,'Cn2','TerrainAtm','HV57','C:/DTED/MyLoc')
```

Create an Atm structure with H-V 5/7 for altitudes above ground for locations in input G using digital terrain elevation data (DTED) to determine terrain altitude along the path. When using TerrainAtm *as a model function input to* AtmStruct*, x and G need not be specified explicitly, but if they are they must be input as a string which will be evaluated within* AtmStruct *to ensure the correct values.*

3.4.3.50 UniformAtm

Syntax

Model = UniformAtm(*h, ModelValue*)

```
SCALING_CODE_API int UniformAtm(char** errorChain, const int n,
    const double modelValue, double* model)
```

This function returns the value of a uniform (constant) model over a propagation path given input *ModelValue* for an arbitrary atmospheric characteristic. UniformAtm is a function of altitude *h*, and returns a vector *Model* of size(*h*). However, the value returned is constant with respect to altitude variation. May be used to represent any atmospheric characteristic that does not vary over a propagation path.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Length of output vector.
modelValue [in]	Constant value of model over propagation path.
model [out]	Model value for each input altitude *h*.

3.4.3.51 UniformCn2

Syntax

Cn2 = UniformCn2(*h, C0*)

Cn2 = UniformCn2(*h, PARAM, ParamValue, lambda, G, x, dx, [xRange]*)

This function returns the value of a uniform (constant) C_n^2 value over a propagation path given an input value *C0*, or given a specified atmospheric parameter, its value, and the wavelength, range, and (if applicable) the set of discrete turbulence phase screens for which the parameter value was calculated. UniformCn2 is a function of altitude *h* above sea level in meters, and returns a *Cn2* vector the size of *h*, but the *Cn2* value returned is constant with respect to altitude variation. The inputs are as follows:

h [vector]	Altitude above earth's surface (m).
C0 [scalar]	Constant *Cn2* value for propagation path ($m^{-2/3}$).
PARAM [string]	Identifier for atmospheric propagation parameter to be specified to determine value of uniform turbulence. Can be set to 'r0', 'rytov', 'theta0'.
ParamValue [scalar]	Numerical value of the atmospheric propagation parameter specified for uniform turbulence.
lambda [scalar]	Wavelength at which specified propagation parameter value was computed (m).

G [struct/scalar]	Geometry structure from GeomStruct or propagation path length for which specified propagation parameter value was computed (m).
x [vector]	For discrete turbulence model, the normalized location of phase screens at which turbulence is introduced along propagation path (m).
dx [vector]	For discrete turbulence model, the normalized thickness of phase screens for turbulence introduced along propagation path (m).
xRange [vector]	(Optional) Normalized range over which input x and dx are normalized. Defaults to [0 1].

The following are the equations used for calculating C_n^2 if coherence length (r_0), Rytov number (σ_χ^2), or isoplanatic angle (θ_0) are specified.

$$
\begin{aligned}
C_n^2(r_0) &= \frac{r_0^{-5/3}}{(2.91/6.88)\, k_0^2 L \int_0^1 (1-x)^{5/3} dx} \\
C_n^2(\sigma_\chi^2) &= \frac{\sigma_\chi^2}{0.563\, k_0^{7/6} L^{11/6} \int_0^1 [x(1-x)]^{5/6} dx} \\
C_n^2(\theta_0) &= \frac{\theta_0^{-5/3}}{2.91\, k_0^2 L^{8/3} \int_0^1 x^{5/3} dx}
\end{aligned}
$$

where $k_0 = 2\pi/\lambda$ and x is the location of phase screens normalized to the propagation path length. Example 3.4.42 shows how to calculate C_n^2 using three different calling sequences.

Example 3.4.42 (Calculation of C_n^2 using UniformCn2) *This example shows how to calculate C_n^2 using* UniformCn2.

```
>> Cn2 = UniformCn2(h,1e-15)
```

Returns a vector the length of h with constant $C_n^2 = 1 \times 10^{-15}\ m^{-2/3}$.

```
>> Cn2 = UniformCn2(h,'r0',0.05,1.064e-6,20000)
```

Returns a vector the length of h with constant C_n^2 value which gives r_0 of 0.05 for wavelength of 1.064 μm and 20 km propagation distance using a continuous model.

```
>> Cn2 = UniformCn2(h,'r0',0.10,1e-6,50000,[0.1,0.3,0.5,0.7,0.9],...
                    [.1,.1,.1,.1,.1])
UniformCn2: Using discrete model with L = 50000 m.
Warning: UniformCn2: sum(dz) = 25000 m, L = 50000 m.
         Turbulence does not fill propagation volume.
```

Returns constant C_n^2 value giving r_0 of 0.10 m at 1 micron wavelength for 50 km propagation path using discrete 5-bin model for which turbulence does not fill the entire propagation volume.

```
>> Atm = AtmStruct(100000,50,10000,20,'MaxAlt',30000,'Cn2', ...
          'UniformCn2','r0',0.1,1e-6,'L','x','dx','xRange');
```

Setup an Atm structure using UniformCn2 *for turbulence model. Input of* ***xRange*** *is optional, it will be added if slant range and* ***x*** *and* ***dx*** *are specified as strings and* ***xRange*** *is not [0 1]. For a continuous Atm structure, must specify '*G*' instead of '*L*'.*

3.4.3.52 US_Standard76

Syntax

[Model1, Model2, Model3] = US_Standard76(*h, ModelType1, ... [ModelType2], [ModelType3]*)

```
SCALING_CODE_API int US_Standard76(char** errorChain, const double* h,
    const int nH, double* temperature, double* pressure,
    double* density)
```

This function calculates Temperature, Pressure and/or Density as a function of altitude in meters. The Matlab inputs are:

h [vector]	Altitude above sea level (m)
ModelType [string]	The desired model type. Available types are 'TEMPERATURE', 'PRESSURE', or 'DENSITY'
ModelType2 [String]	(Optional) 2nd desired model type
ModelType3 [String]	(Optional) 3rd desired model type

The function returns vectors *Model* the size of *h* with the value of temperature (K), pressure (Pa), or density (kg/m^3), depending on *ModelType*, at the given altitudes. Data computed using the 1976 US Standard model [37] is loaded from the file 'USS_UPPER.mat'. The function can return up to three output vectors if more than one model type is specified. If only one output is specified with more than one model type, the output will be a matrix containing all the data.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
temperature [out]	(NULL OK) Temperature (k); if NULL temperature will not be evaluated; otherwise, expected length is *nH*.
pressure [out]	(NULL OK) Pressure (Pa); if NULL pressure will not be evaluated; otherwise, expected length is *nH*.
density [out]	(NULL OK) Density (kg/m^3); if NULL density will not be evaluated; otherwise, expected length is *nH*.

3.4.3.53 VTP2VLLA

Syntax

Vlla = VTP2VLLA(*h, x, G, Vtp, [Vmag], [upDir], [outType], coordBase*)

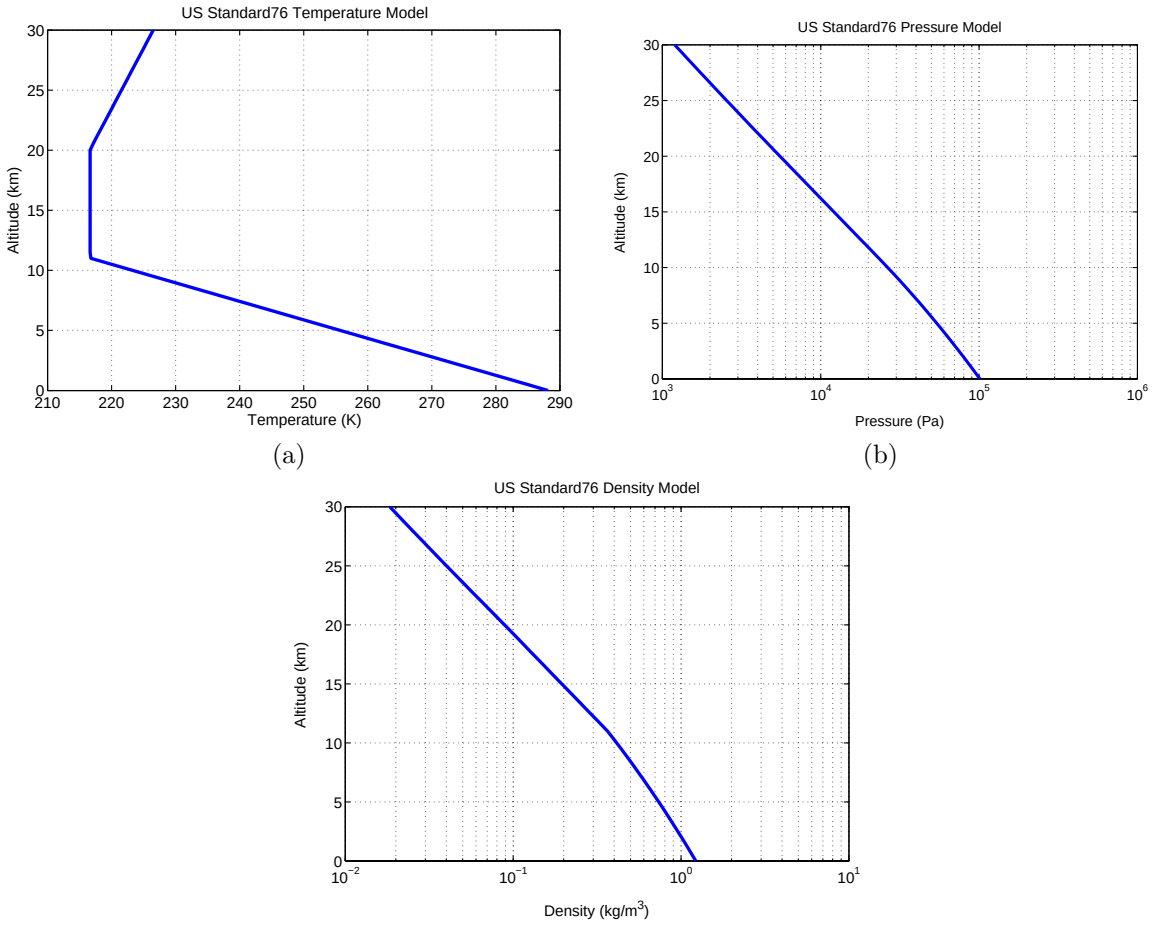


Figure 3.31: (a) Temperature, (b) Pressure, and (c) Density calculated using the 1976 US Standard model.

This function converts velocity in the target TP coordinate frame defined in a plane perpendicular to the propagation to an LLA velocity based on the platform and target locations. Can specify an alternate vector for projection of the zenith onto the plane perpendicular to the propagation, ***upDir***, for an arbitrary rotation of the TP coordinate frame. One can also specify that the platform be used as the basis for the TP coordinate frame. This option should be used if the user wants to analyze the reverse geometry but does not want to change the relationship between TP/XY and the propagation direction.

h [vector]	Altitude above mean sea level (m).
x [vector]	Normalized screen locations.
G [struct]	Geometry structure from GeomStruct.
Vtp [vector]	Wind velocity in TP coordinates (m/s). The size is either 1×2 for a constant TP velocity along the path or NScreens $\times$ 2 to vary TP velocity along the path.
Vmag [scalar]	(Optional) Magnitude for the velocity. If not specified, magnitude will be equal to that of ***Vtp***.
upDir [vector]	(Optional) Projection of the zenith onto the plane perpendicular to the propagation direction. Defaults to zenith projection in the target TP coordinates.
outType [string]	(Optional) Indicates the output. Either 'horizontal' for only horizontal wind speed, 'heading' for wind heading, or 'vertical' for vertical wind speed. If not input, the output is an LLA velocity vector for wind.
coordBase [string]	(Optional) Specify 'P' to use the platform as the base for the TP coordinates. Defaults to target-based coordinates.

Input of ***h*** and ***x*** facilitates computation of natural wind along the path. To compute velocity for platform or target use ***h*** equal to ***hp*** or ***ht*** and ***x*** equal to 0 or 1, respectively. To compute the LLA velocity, the decomposition ***Vtp*** is used with ***upDir*** to compute the in-plane horizontal and vertical velocity components. The total speed, ***Vmag***, is used to compute the out-of-plane components for the horizontal and vertical velocity. If not input, it is assumed that the vector sum of the out-of-plane components is zero. The in-place and out-of-plane components of horizontal velocity determine the heading.

The output is

Wlla [array]	LLA velocity vector or speed/heading based on ***outType***.

Example 3.4.43 (VTP2VLLA) *Illustrates use of VTP2VLLA to determine velocity from a decomposition of velocity.*

```
>> S = EngagementStruct(GeomStruct('Simple',1000,10000,100000, ...
       100,10,90,0));
>> VTLLA = VTP2VLLA(S.ht,1,S.G,[0 S.V_TTP],S.vt);
```

Returns the LLA velocity vector for the target.

```
>> VTLLA = VTP2VLLA(S.ht,1,S.G,[100 20],[],[1 0]);
```

Determine the LLA velocity vector for the target when it is known that the target has an apparent "up" velocity of 100 m/s and an apparent velocity to the right of 20 m/s. An apparent velocity to the left would be specified as -20.

```
>> Atm = AtmStruct(S.G,10);
>> Wlla = VTP2VLLA(Atm.h,Atm.z/Atm.L,S.G,[5 0]);
```

Returns an LLA velocity vector for natural wind along the path

```
>> Atm = AtmStruct(S.G,10, ...
        'Wind','VTP2VLLA','x','G',[5 0],5,[],'horizontal', ...
        'WindHeading','VTP2VLLA','x','G',[5 0],5,[],'heading', ...
        'VerticalWind','VTP2VLLA','x','G',[5 0],5,[],'vertical')
```

Computes an Atm structure with a 5 m/s wind perpendicular to the propagation direction and perpendicular to the target motion.

3.4.3.54 WSMRatm

Syntax

[Atm, G] = WSMRatm(*Direction, DateRange, TimeRange, OutputType, [Atm0]*)

This function returns an atmospheric modeling (*Atm*) structure containing C_n^2 modeling data with 10 path segments for specified conditions. The output is based upon differential-tilt turbulence profiler measurements made from May to August at White Sands Missile Range (WSMR) for propagation between North Oscura Peak (NOP) and the Beck test site. *Atm* structure will have *Atm.Cn2* values for the specified conditions, and *Atm.Cn2Eval* = 'NoEval'. Also returns the ECF geometry structure for the specified path as an optional second output. A range of dates to be used and range of hours on those dates can be specified. The nature of the output data for C_n^2 is determined by input *OutputType* (see options below).

Direction [string]	Direction of laser propagation 'NOPBeck' or 'BeckNOP,' defaults to 'NOPBeck'.
DateRange [cell]	Range of observation dates to consider from database - Use MATLAB® date strings, e.g., '01-Jan-2006.' To specify a date interval use {'01-May-2006', '31-May-2006'}. Defaults to {'01-Jan-2006', '31-Dec-2006'}.
HourRange [vector]	Hour of day interval to consider from database. Use vector interval, e.g., [0 6]. Defaults to [0 24].

OutputType [scalar/string]	Type of output atmospheric modeling desired for the specified interval. Input scaling value provides a margin of Cn2 distribution, i.e., *OutputType*=0.5 gives 50%-tile. String options include: • 'ALL' - All observations • 'MIN' - Minimum Cn2 for each range bin • 'MAX' - Maximum Cn2 for each range bin • 'MEAN' - Mean of Cn2 for each range bin • 'BEST' - Profile with lowest integrated Cn2 • 'WORST' - Profile with highest integrated Cn2.
Atm0 [structure]	(Optional) Baseline atmospheric modeling structure for which only the Cn2 model will be changed.

The outputs are

Atm [struct]	Atmospheric modeling structure containing Cn2 model.
G [struct]	Geometry structure for specified path.

3.4.3.55 WSMRCn2

Syntax

Cn2 = WSMRCn2(*h, ModelType, [GroundAlt]*)

```
SCALING_CODE_API int WSMRCn2(char** errorChain, const double* h,
     const int nH, const char* modelType, const double* groundAlt,
     double* cn2)
```

This function returns the value of the White Sands Missile Range (WSMR) turbulence profile given altitude above ground level in meters. The inputs are

h [vector]	Altitude above ground/sea level (m).
ModelType [string]	Model descriptor for turbulence condition. Default is '50%Day' if not specified. Valid inputs are: '10%Best', '50%Day', '50%Night', '80%WorstSpring', '90%Worst-Day', 'Night', and 'Average'.
GroundAlt [scalar]	(Optional) Ground altitude above sea level if input altitude is meters above sea level. Values less than *GroundAlt* will be set to *GroundAlt* for evaluation of the model.

If input altitude is specified as meters above sea level, the ground altitude can optionally be specified. The model identified as '80%WorstSpring' is the 80th percentile worst C_n^2 for springtime measurements collected between 1000-1400 local time at WSMR.

The output is

Cn2 [vector]	Index of refraction structure coefficient ($m^{-2/3}$).

Figure 3.32 shows a plot of the WMSR turbulence profiles under different conditions.

The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
h [in]	Altitude above ground/sea level (m); vector of length *nH*.
nH [in]	Length of input and output vectors.
modelType [in]	(NULL OK) Model descriptor for turbulence condition. Default is '50%Day' if not specified. Valid inputs are: '10%Best', '50%Day', '50%Night', '80%WorstSpring', '90%WorstDay', 'Night', and 'Average'.
groundAlt [in]	(NULL OK) Ground altitude (m). Set to 0 if passed in as NULL.
cn2 [out]	Index of refraction structure coefficient ($m^{-2/3}$); must be allocated to length *nH*.

3.4.3.56 WSMRWIND

Syntax

v = WSMRWIND(*h*)

This function returns the value of the White Sands Missile Range (WSMR) wind profile. The input is

h [vector]	Altitude above mean sea level (m).

The output is

v [vector]	Wind speed (m/s).

Figure 3.33 is a plot of wind velocity with altitude for the WSMR profile.

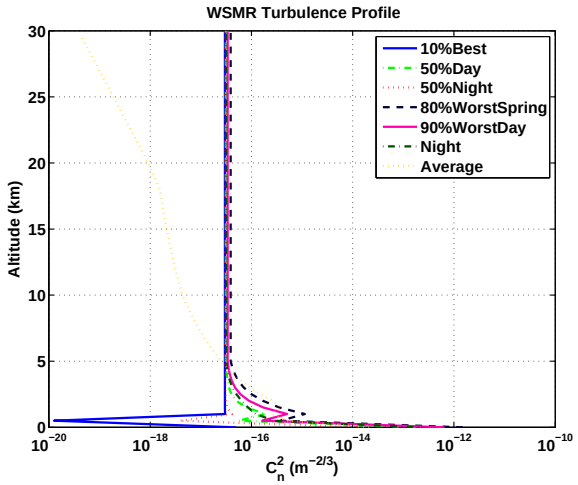


Figure 3.32: Turbulence profile as a function of altitude calculated using `WSMRCn2` for different conditions.

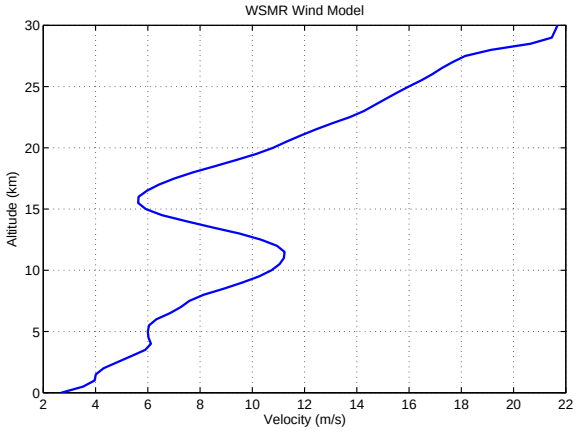


Figure 3.33: Wind profile as a function of altitude calculated using `WSMRWind`.

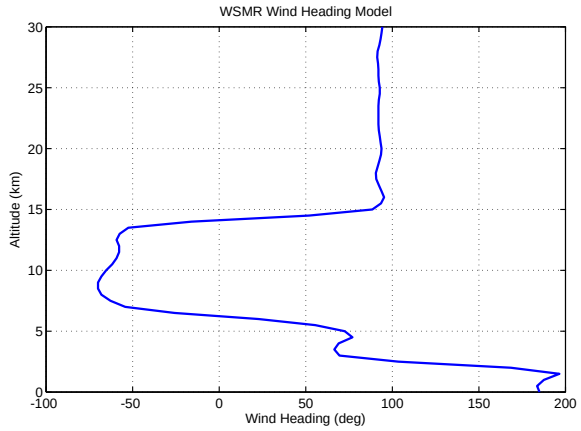


Figure 3.34: Wind heading profile as a function of altitude calculated using WSMRWINDHEADING.

3.4.3.57 WSMRWINDHEADING

Syntax

wh = WSMRWINDHEADING(h)

This function returns the value of the White Sands Missile Range (WSMR) wind heading profile. The input is

h [vector]	Altitude above mean sea level (m).

The output is

wh [vector]	Wind heading (deg).

Figure 3.34 is a plot of wind heading with altitude for the WSMR profile.

3.4.4 LEEDR Atmospheric Model Functions

This section contains descriptions of functions distributed with ATMTools for use in atmospheric modeling with the Laser Environmental Effects Definition and Reference (LEEDR)[6]. Many of these functions require the user to have an m-code or p-code version of LEEDR on the MATLAB® path or at least to have pre-computed atmospheric data from LEEDR. The functions are compatible with LEEDR versions 3.3 and 4.0. Section 3.4.4.1 gives the user an overview of using LEEDR in ATMTools with the following sections providing more detail on the functions available. In addition to the those described below, the functions LEEDRATMOS, GADSATMOS, and OCEANATMOS described in Section 3.4.3 also make use of LEEDR.

3.4.4.1 Using LEEDR in ATMTools

There are several options for getting LEEDR data into MATLAB®. From the LEEDR GUI after computing the atmospheric profile, the data can be saved to a MATLAB® data file. The function LOADLEEDRATMOS can then be used to load the data from the file and put it into the MATLAB® application data (or app data). In a sense,

[6]For information on LEEDR see http://www.afit.edu/CDE/page.cfm?page=329&tabname=Tab5A or contact AFIT CDE, 2950 Hobson Way, Wright-Patterson AFB, Ohio, 937.255.3636 x4506 or CDE@afit.edu

setting app data is like creating a global variable, but this method is a little more robust because the data can't be cleared by issuing the command `clear all`. If using an m-code or p-code version of LEEDR 3.3 directly from MATLAB®, the data is automatically put into the MATLAB® app data after the calculation. When running LEEDR 4.0 from MATLAB®, the user must manually send the data to MATLAB® from the GUI and then pass the resulting variable into loadLEEDRAtmos to add the data to the app data. To see the LEEDR output structure, issue the command

```
>> LEEDROut = loadLEEDRAtmos;
```

The output *LEEDROut* is a LEEDR LUT object from LEEDRLUT. Another option for getting LEEDR data is to use the functions setLEEDRInputs and buildLEEDRAtmos to compute the LEEDR data outside of the GUI. These functions access the LEEDR calculations and require m-code or p-code version of LEEDR on the MATLAB® path. The MATLAB® command `LEEDRLUT.getInputNames` will return a cell array of inputs that can be changed using setLEEDRInputs. After running buildLEEDRAtmos, the LEEDR output data will be put into the MATLAB® app data. There is an Example script distributed with ATMTools, Examples/LEEDRAtmosphere.m, as well as sample LEEDR atmospheric data in the same directory.

Once LEEDR output data is in the MATLAB® app data, one can use saveLEEDRAtmos to save the data to a MATLAB® data file, mergeLEEDRAtmos to merge the data with other pre-computed LEEDR data at a different wavelength, and LEEDRAtmos to access and interpolate the data and create an Atm structure. Below is an example of a call to AtmStruct that uses LEEDRAtmos.

Example 3.4.44 (Atm with LEEDR) *This example illustrates use of* LEEDRAtmos *for creating an atmospheric structure.*

```
>> loadLEEDRAtmos('myLEEDRdata.mat')
>> Atm = AtmStruct(G,10,'Cn2','LEEDRAtmos','Turb', ...
    'Abs','LEEDRAtmos','A','1064', ...
    'Scat','LEEDRAtmos','S','1064', ...
    'Temperature','LEEDRAtmos','T', ...
    'Wind','LEEDRAtmos','W','WindHeading','LEEDRAtmos','WH')
```

This will create an Atm structure using the LEEDR data that is stored in the app data from the previous call to loadLEEDRAtmos. *If the LEEDR data does not contain calculations for 1.064 micron wavelength, data for that wavelength will be computed if LEEDR is on the* MATLAB® *path or absorption and scattering will not be set.*

Compatibility Considerations For scripts accessing the LEEDR 3.3 data through ATMTools, changes will need to be made to work with this release of ATMTools. If *m* is the output from loadLEEDRAtmos, the following table shows a few of the replacements that should be made.

Replace (LEEDR 3.3):	With (LEEDRLUT Method):
`m.Inputs.boundaryLayerHeight`	`m.getBoundaryLayerAltitude()`
`m.siteAltitude`	`m.getSiteAltitude()`
`m.Inputs.Wavelength`	`m.getWavelength()`
`m.altVector`	`m.getAltitudes()`
`m = getappdata(0,'LEEDRAtmosphere')`	`m = loadLEEDRAtmos`
`rmappdata(0,'LEEDRAtmosphere')`	`loadLEEDRAtmos([])`

Additionally, we have provided a method for computing LEEDR output using LEEDR 4.0 and inputs from LEEDR 3.3.

Example 3.4.45 (Updating LEEDR data) *This example illustrates how one would recompute older LEEDR data using a newer version of LEEDR.*

```
>> m_3p3 = loadLEEDRAtmos('LEEDR_3p3_Data.mat');
>> m_4p0 = buildLEEDRAtmos(updateLEEDRInputs(m_3p3.Inputs),'Altitudes',m_3p3.getAltitudes())
```

Load LEEDR data created with LEEDR 3.3, update the inputs to LEEDR 4.0 format, update the altitudes, then build the LEEDR atmosphere with LEEDR 4.0 on the MATLAB® *path. Altitudes must be included separately since altitudes are part of the outputs in LEEDR 3.3. Otherwise, altitudes used will be the default altitudes for LEEDR 4.0.*

```
>> m_4p0 = updateLEEDRAtmos(m3p3);
```

The function updateLEEDRAtmos *was added for Release 2015a to facilitate updating existing LEEDR data. This method will maintain the same altitudes.*

```
>> figure; plot(m_3p3.getTotalAbsorption(),m_3p3.getAltitudes(),'-bo', ...
    m_4p0.getTotalAbsorption(),m_4p0.getAltitudes(),'-gs')
```

Obtain a plot comparing absorption with altitude for the different LEEDR versions. Because both m_3p3 and m_4p0 are LEEDRLUT objects, the commands to return the absorption and altitude are the same, independent of LEEDR version.

3.4.4.2 setLEEDRpath

Syntax

setLEEDRpath(*force*)

This function sets the MATLAB® path to include LEEDR directories for use of LEEDR in script mode. The user must have an m-code or p-code version of LEEDR installed and have the main directory containing the leedr.m or leedr.p file on the MATLAB® path. This function will then locate and add all other necessary directories to the path. The optional input, *force*, can be passed as true to override any persistent or global variable settings. The default value for *force* is false. Set this to true to force the addition of LEEDR to your path - sometimes the global variable for LEEDR gets set to true and LEEDR is removed from the path.

3.4.4.3 buildLEEDRAtmos

Syntax

LEEDROut = buildLEEDRAtmos(*[inpt], [PName1], [PValue1], ...*)

This function generates LEEDR model data for use with the function LEEDRAtmos. Requires p-code or m-code version of LEEDR available from the Air Force Institute of Technology. Creates an entry in the workspace application data for use by LEEDRAtmos - use `m = loadLEEDRAtmos` with no inputs to obtain the data generated and `loadLEEDRAtmos([])` to clear the data. The inputs are

inpt [various]	(Optional) Can be output generated from the LEEDR GUI or the output resulting from a LEEDR run. If not input code will use results from the MATLAB® application data, if available, or the default LEEDR inputs.

PName [string]	(Optional) Name of parameter to be updated. Use `LEEDRLUT.getInputNames()` to obtain a list of parameters that can be changed. Parameter names are case sensitive.
PValue [varies]	(Optional) Value for parameter associated with *PName*. There must be a parameter value for each parameter name specified.

The output, *LEEDROut*, is only returned if requested and is a LEEDRLUT object containing LEEDR computed atmospheric model data.

Example 3.4.46 (buildLEEDRAtmos) *This example shows how to call* BUILDLEEDRATMOS.

```
>> output = buildLEEDRAtmos('Latitude',39.83,'Longitude',-84.05)
```

Compute LEEDR atmospheric data using the current data or default settings if no current data exists with a change to the location.

```
>> output = buildLEEDRAtmos(output.Inputs,'Season',2)
```

Use the Inputs structure from ***output*** *with a change to the season.*

3.4.4.4 BUILDLEEDRWXCUBE

Syntax

[wxCube, location, nomadsFile] = BUILDLEEDRWXCUBE(*Wavelength, LatLong, grbFile, [PName1], [PValue1], ...*)

[wxCube, location, nomadsFile] = BUILDLEEDRWXCUBE(*Wavelength, G, [PName1], [PValue1], ...*)

This function generates LEEDR weather cube data for use with the function LEEDRWXCUBE. This requires p-code or m-code version of LEEDR available from the Air Force Institute of Technology. After generating the data, the function will write mesh and h5 files to the directory returned by LEEDRWXCUBELOOKUPPATH. The inputs are

Wavelength [vector]	vector of wavelengths (m).
G [struct]	Geometry information, including dates/times (in UT) in TALO.
LatLong [vector]	Boundaries of the generated cube in format [North Lat, South Lat, East Lon, West Lon].
grbFile [char]	Full path to pre-downloaded NOMADS data file. Data can be downloaded from nomads.ncdc.noaa.gov/data/gfs4/. Filename must be formatted as follows: gfs_4_YYYYMMDD_HH00_CCC (NOAA downloads) OR YYYYMMDDHH00CCC (LEEDR downloads).

PValue [varies]	(Optional) Value for parameter associated with *PName*. There must be a parameter value for each parameter name specified.
'useCorrK'	true/false - Flag for setting whether or not to use correlated-K for the molecular absorption which can be much faster when generating multiple wavelengths and is preferred if band absorption is desired - default is true.
'Terrain'	terrainValue - Description of the site terrain (case insensitive) or surface [albedo, evap const, surface roughness (m)]. See MzaSurfCn2.
'Cn2'	cn2Model - identifier to indicate the LEEDR Cn2 model. See Turbulence.list for a list of available LEEDR Cn2 models. If set to 'Calculations - Tatarski', ground level values will be computed using MZAsurfCn2 for daytime hours unless the surfCn2Replace parameter is set to false.
'surfCn2'	surfCn2Replace - if set to false, the surface value is not replaced using the MZASurfCn2 model. Pass in a value (or matrix of values with size(wxCube_grid,1)) to use for ground level Cn2. Grid size determined by numel(floor(LatLong(2)*2)/2:.5:ceil(LatLong(1)*2)/2) × numel(floor(LatLong(4)*2)/2:.5:ceil(LatLong(3)*2)/2). Default true if cn2Model is 'Calculations - Tatarski'.
'useCn2NN'	true/false/char - Flag for setting whether or not to use the machine learning neural network to adjust ground level Cn2. Can also pass in a file name containing the neural network data to be used. Defaults to true - using the NN data from Cn2WeatherNN_30N_2L_11I.mat unless useMZAsurfCn2 is false.
'MinSolarIrr'	Minimum value for ground level solar irradiance. Use to possibly override the NaN return from surface model when solar irradiance is low. May still get NaN under low wind conditions. Ignored if *surfCn2* is false. NOTE: The model does not support Cn2 estimation under these conditions.
'MaxAlt'	maxAlt - altitude above which the atmosphere is considered a vacuum (m). Ignored if LatLong is passed in. Use this altitude to limit the extent of the cube lats and lons when passing in a GEOM structure. The cube will still be generated with the default altitudes (1001 divisions from 0 to 100km). Be sure to use the same *MaxAlt* in the ATM structure to prevent the ATM screens from going outside the limits of the cube.
'nLayers'	Number of layers (altitudes) between 100 m and *MaxAlt* to use when generating the weather cube - defaults to 1000. Actual numbers of layers will be *nLayers* + 8 (surface layers).

'DTEDdir'	Directory holding the downloaded DTED data - data can be downloaded from http://earthexplorer.usgs.gov/. Set to false to prevent generation of DTED data.
'writeMesh'	true/false - Flag for setting whether or not mesh files are generated. This is the original binary storage format for LEEDR generated weather cubes. Default is false.

The outputs are

wxCube [struct]	Weather cube data.
location [char]	Folder name with the location information from Lat-Long.
nomadsFile [char]	Folder name for date/time in format YYYYMMDDHHHCCC where HHHH is the hour cycle (ZULU) and CCC is the forecast hour.

3.4.4.5 CFLOSSites

Syntax

data = CFLOSSites(*Cond, [NumSites]*)

This function loads the data descriptions for the CFLOS sites. Data is displayed if no outputs are assigned. The inputs are

Cond [string/vector]	Condition to test against. If a string, returns all sites with string in any of the site names. If not a string, assumes the input is site number if column vector or [Latitude Longitude] and will return the closest site to that position.
NumSite [scalar]	Number of sites to return if passing in [Lat Long]. Defaults to 5.

The outputs are

data [cell]	Cell array of CFLOS site *data* of size # Sites x 4 with: Col 1 - Site Index, Col 2 - Site Name, Col 3 - Latitude of the site (deg), Col 4 - Longitude of the site (deg).

3.4.4.6 DOWNLOADGRBDATA

Syntax

grbFile = DOWNLOADGRBDATA(*obsDay, obsForecast, [obsCycle], [gfsDataPath]*)

This function downloads the relevant Grib file from https://nomads.ncdc.noaa.gov. If standard data does not exist for requested times, will try to get re-analysis data. The arguments are

obsDay [datetime]	Day of observation. Can be a Matlab datetime object of a date vector. If not a datetime object, GMT is assumed.
obsForecast [scalar]	Forecast hour. Valid forecasts are every 3 hours starting at 0 up to 240 hours, then every 12 hours up to 384 hours. For re-analysis data, forecasts are 0, 3, and 6 hours only.
obsCycle [scalar]	(Optional) Cycle of the GFS observation. Valid cycles are 0, 6, 12, and 18 hours - will select closest valid cycle. If not passed in or passed in as empty, the *obsCycle* will be *obsDay.Hour*.
gfsDataPath [char]	(Optional) Path to folder for storing GFS data. Defaults to current working directory.

The output is a cell array of filenames for the Grib files downloaded.

3.4.4.7 LEEDRLUT

Syntax

LEEDROut = LEEDRLUT(*inpts*)

This function creates a LEEDR LUT object from the inputs for use with LEEDR in ATMTools. This function does not set or modify LEEDR profiles in the Matlab app data. For that use LOADLEEDRATMOS and/or BUILDLEEDRATMOS.The optional input is

inpts [various]	LEEDR input data. Can be a LEEDR input structure or object or pre-computed LEEDR data structure or object. If not input, returns LEEDR data from the MATLAB® application data if it exists or a LEEDR object with the default input.

The output is

LEEDROut [LEEDRLUT]	LEEDR LUT object.
LEEDROut.Inputs [struct/obj]	LEEDR inputs structure or object.
LEEDROut.Outputs [struct/obj]	LEEDR outputs structure or object.
LEEDROut.time [cell]	Time-stamp for the LEEDR run.
LEEDROut.name [cell]	User-specified name(s) for the LEEDR run.
LEEDROut.LEEDRver [char]	LEEDR version information for the data.
LEEDROut.hasRun [logical]	Flag indicating whether *LEEDROut* contains output data.

Example 3.4.47 (LEEDRLUT Tips) *The following examples illustrate useful commands for use with a LEEDRLUT object.*

```
>> loadLEEDRAtmos('LEEDR_RuralMidLatSummer.h5');
>> m = LEEDRLUT()
```

Return LEEDR LUT data from the MATLAB® *application data if loaded or LEEDR LUT data populated with default inputs.*

```
>> m.addWavelength(1.03e-6)
```

Compute LEEDR data at a new wavelength and merge with data in `m`*. Must have p-code or m-code version of LEEDR on the path.*

```
>> loadLEEDRAtmos(m)
```

Add LEEDR data to or merge with existing data in the Matlab app data.

```
>> LEEDRLUT.getInputNames()
```

Returns a cell array of parameter names that can be used with setLEEDRInputs *and* buildLEEDRAtmos*.*

```
>> S = struct(m)
```

Create a structure of most commonly accessed LEEDR data. `S` *always has the same format, regardless of LEEDR version.*

```
>> lds = m.export()
```

Export the LEEDR LUT object to the original format of the LEEDR data.

```
>> allmethods = methods('LEEDRLUT');
>> getMethods = allmethods(strncmp(allmethods,'get',3))
```

Returns in `getMethods` *all methods that return data.*

3.4.4.8 LEEDRSites

Syntax

data = LEEDRSites(*Cond, [NumSites]*)

This function loads the data descriptions for the LEEDR sites. Requires m-code or p-code version of LEEDR on the MATLAB® path. Data is displayed in the MATLAB® command window if no outputs are assigned. The inputs are

Cond [string/vector]	Condition to test against. If a string, returns all sites with string in any of the site names. If not a string, assumes the input is site number if column vector or [Latitude Longitude] and will return the closest sites to that position.
NumSite [scalar]	Number of sites to return if passing in [Lat Long]. Defaults to 5.

The output is

data [cell]	Cell array of LEEDR site *data* of size # Sites x 8 with: Col 1 - Site Index, Col 2 - Site Name, Col 3 - Site Name, Col 4 - Latitude of the site (deg), Col 5 - Longitude of the site (deg), Col 6 - Atmosphere type of the site, Col 7 - Site Name, Col 8 - Altitude of the site (m).

3.4.4.9 LEEDRWXCUBELUT

Syntax

wxCube = LEEDRWXCUBELUT(*lambda, LatLong, grbFile, [PName1], [PValue1], ...*)

wxCube = LEEDRWXCUBELUT(*lambda, G, grbFile, [PName1], [PValue1], ...*)

wxCube = LEEDRWXCUBELUT(*loc, dt*)

wxCube = LEEDRWXCUBELUT(*wxCube*)

This function generates LEEDR weather cube object data for use with the function LEEDRWxCube. Requires p-code or m-code version of LEEDR available from the Air Force Institute of Technology. Creates an entry in the workspace application data for use by LEEDRWxCube - use `wxCube = loadLEEDRWxCube` with no inputs to obtain the data generated. Will write mesh and h5 files to LEEDRWXCUBE lookup path.

The inputs are

Wavelength [vector]	Vector of wavelengths (m).
G [struct]	Geometry information, including dates/times (in UT) in TALO.
LatLong [vector]	Boundaries of the generated cube in format [North Lat, South Lat, East Lon, West Lon].
grbFile [char]	Full path to pre-downloaded NOMADS data file. Data can be downloaded from nomads.ncdc.noaa.gov/data/gfs4/. Filename must be formatted as either gfs_4_YYYYMMDD_HH00_CCC (NOAA downloads) OR YYYYMMDDHH00CCC (LEEDR downloads).
PValue [varies]	(Optional) Value for parameter associated with PName. There must be a parameter value for each parameter name specified.

Optional parameters can be specified when generating weather cube data. There must be a parameter value for each parameter name specified. The following is a list of the available options for *PName* and a description of the *PValue*.

- 'useCorrK' = true/false - Flag for setting whether or not to use correlated-K for the molecular absorption which can be much faster when generating multiple wavelengths and is preferred if band absorption is desired - default is true.
- 'Terrain' = *terrainValue* - Description of the site terrain (case insensitive) or surface [albedo, evap const, surface roughness (m)]. See MzaSurfCn2.
- 'Cn2' = *cn2Model* - Identifier to indicate the LEEDR Cn2 model. See Turbulence.list in LEEDR for a list of available Cn2 models. If set to 'Calculations - Tatarski', ground level values will be computed using MZAsurfCn2 for daytime hours unless the *surfCn2Replace* parameter is set to false.
- 'surfCn2' = *surfCn2Replace* - if set to false, the surface value is not replaced using the MZASurfCn2 model. Default true if *cn2Model* is 'Calculations - Tatarski'.
- 'useCn2NN' = true/false/char - Flag for setting whether or not to use the machine learning neural network to adjust ground level Cn2. Can also pass in a file name containing the neural network data to be used. Defaults to true - using the NN data from Cn2WeatherNN_30N_2L_11I.mat unless useMZAsurfCn2 is false.
- 'MinSolarIrr' = Minimum value for ground level solar irradiance. Use to possibly override the NaN return from surface model when solar irradiance is low. May still get NaN under low wind conditions. Ignored if *surfCn2Replace* is false. NOTE: The model does not support Cn2 estimation under these conditions.
- 'MaxAlt' = maxAlt - altitude above which the atmosphere is considered a vacuum (m). Ignored if *LatLong* is passed in. Use this altitude to limit the extent of the cube lats and lons when passing in a GEOM structure. The cube will still be generated with the default altitudes (1001 divisions from 0 to 100km). Be

sure to use the same ***maxAlt*** in the ATM structure to prevent the ATM screens from going outside the limits of the cube.

- 'nLayers' = Number of layers (altitudes) between 100 m and ***maxAlt*** to use when generating the weather cube - defaults to 1000. Actual numbers of layers will be ***nLayers*** + 8 (surface layers)
- 'DTEDdir' = Directory holding the downloaded DTED data - data can be downloaded from http://earthexplorer.usgs.gov/. Set to false to prevent generation of DTED data.
- 'writeMesh' = true/false - Flag for setting whether or not mesh files are generated. This is the original binary storage format for LEEDR generated weather cubes. Default is false.

The output is

wxCube [LEEDRWXCUBEOBJECT]	Weather cube data.
location [char]	Identifier for the weather cube location.
gfs [char]	Date/Time identifier for the GFS data used.
lats [vector]	Latitude points in the mesh (deg). Length nPts=nLats∗nLons.
lons [vector]	Longitude points in the mesh (deg). Length nPts.
alts [vector]	Altitudes at which data is computed (m). Length nAlts.
wl [vector]	Wavelengths for which extinction, absorption, and scattering are computed (m). Length nWvls.
abs [array]	Absorption coefficient (km^{-1}). Size nPts × nWvls × nAlts.
airDensity [array]	Density (kg/m^3). Size nPts × nAlts.
albedo [array]	Albedo (scattering/extinction). Size nPts × nWvls × nAlts.
cn2 [array]	Index structure constant ($m^{-2/3}$). Size nPts × nAlts.
ext [array]	Extinction coefficient (km^{-1}). Size nPts × nWvls × nAlts.
pressure [array]	Pressure (mbar). Size nPts × nAlts.
rh [array]	Relative humidity (%). Size nPts × nAlts.
scatter [array]	Scattering coefficient (km^{-1}). Size nPts × nWvls × nAlts.
temperature [array]	Temperature (K). Size nPts × nAlts.
vv [array]	Vertical velocity (Pa/s). Size nPts × nAlts.
windDir [array]	Wind direction (deg). Size nPts × nAlts.
windSpeed [array]	Wind speed (m/s). Size nPts × nAlts.
verticalWind [array]	Vertical wind speed (m/s). Size nPts × nAlts.

wx [array]	Clouds/rain/fog status. Size nPts × 6 × nAlts where the second dimension contains fraction of cloud coverage and altitudes for 3 layers.
surface [struct]	Surface data, if generated, for each model.
DTED [struct]	Structure containing elevation data for the mesh. Empty ([]) if not available.

3.4.4.10 LOADLEEDRATMOS

Syntax

LEEDROut = LOADLEEDRATMOS(*inputs*)

This function loads pre-computed LEEDR atmospheric data from a file and places the data into the desktop application data for use with LEEDRATMOS. It can also take in a LEEDR data structure and place that data into the desktop application data. The input is

inputs [string/struct]	Filename from which to load data or a LEEDR outputs structure.

If no inputs are passed in, the function will return the application data, if any exists. If the input is empty ([]), the function will clear the current application data.

The output for this function, *LEEDROut*, is only returned if requested and is a LEEDR LUT data object containing computed atmospheric model data. The input data will replace existing LEEDR data in the application data - use MERGELEEDRATMOS to combine the input data with existing data.

3.4.4.11 LOADLEEDRWXCUBE

Syntax

wxCube = LOADLEEDRWXCUBE(*location, time*)

This function returns the raw data and associated location for a LEEDR Wx Cube. The inputs are

location [char]	Identifier for the physical location of the data.
time [char]	Identifier for the time of the GFS data.

The optional output is *wxCube* which is weather cube data object from LEEDRWXCUBELUT.

Example 3.4.48 (loadLEEDRWxCube) *Load LEEDR Wx cube data.*

```
>> wxCube = loadLEEDRWxCube('N40.00_N39.50_W83.00_W84.50','201802270000006')
```

3.4.4.12 meanRefract

Syntax

N = meanRefract(*wavelength, press, temp, dp*)

This function returns the value of the mean refractivity (n - 1) using the LEEDR function CiddorOld. The inputs are

wavelength [scalar]	Wavelength (m).
press [vector]	Pressure (Pa). Note, the output of LEEDR has pressure in millibars. Multiply by 100 to get Pa.
temp [vector]	Temperature (K).
dp [vector]	Dew point (K).

The output is

N [vector]	Mean refractivity.

3.4.4.13 mergeLEEDRAtmos

Syntax

LEEDROut = mergeLEEDRAtmos(*LEEDRIn*)

This function combines a LEEDR data structure with the LEEDRAtmosphere data in the workspace application data and places the result in the application data for the workspace. It will only combine the structures if the fields in "Inputs" have identical values other than the Wavelength. If the input data does not match the current LEEDR data, current LEEDR data will be replaced. The inputs are

inputs [various]	Output structure from LEEDR, or a list of filenames with stored LEEDR, or a combination of structures and filenames. NOTE: Currently ALL .Inputs structures other than the Wavelength field must be equal.

The output is the LEEDR data structure that is the result of the merge.

Example 3.4.49 (mergeLEEDRAtmos) *Examples of merging LEEDR atmospheric data.*

```
>> output = mergeLEEDRAtmos(leedrOutput)
```

Merge output from LEEDR with the current Matlab application data. Output will be the merged LEEDR data if only wavelength is different. Merged LEEDR data will be added to the app data.

```
>> output = mergeLEEDRAtmos('file1.mat','file2.mat')
>> output = mergeLEEDRAtmos({'file1.mat' 'file2.mat'})
```

Combine the LEEDR data from file1.mat and file2.mat into a single LEEDR data structure. This will also combine with existing data if it exists and the only difference is wavelength.

3.4.4.14 PCFLOS

Syntax

[val data] = PCFLOS(*Geom, [season], [TOD], [siteID]*)

This function returns the probability of cloud-free line of sight given geometry, season, time of day and specified location. The inputs are

Geom [struct/list]	Geometry parameters. Can be a structure from GEOMSTRUCT or a comma separated list of (*hp*, *ht*, *rd*, ...).
hp [scalar]	Platform altitude (m).
ht [scalar]	Target altitude (m).
rd [scalar]	Downrange along spherical earth surface (m).
season [scalar]	(Optional) Index to *season*: 1 - summer (default), 2 - winter.
TOD [scalar]	(Optional) Index to time of day: 1 - 00:00 to 03:00, 2 - 03:00 to 06:00, 3 - 06:00 to 09:00, 4 - 09:00 to 12:00, 5 - 12:00 to 15:00, 6 - 15:00 to 18:00, 7 - 18:00 to 21:00, 8 - 21:00 to 00:00, 9 - Daily average (default).
siteID [scalar]	(Optional) Not required when input geometry is a structure (will override position in input geometry if passed in with a geometry structure). Index to the site for CFLOS computation - see CFLOSSITES. If not passed in and input geometry is a list, will use the site closest to [0 0].

The outputs are

val [scalar]	Probability of cloud free line of sight for the input conditions.
data [struct]	Structure containing computations for CFLOS for various combinations of platform altitude, target altitude, view angle, and time of day.

3.4.4.15 saveLEEDRAtmos

Syntax

saveLEEDRAtmos(*fname*)

This function saves pre-computed LEEDR atmospheric data from the desktop application data to a MATLAB® data file, *fname*. The input is

fname [string]	Filename to which data will be saved. Supported file types are .mat, .h5, and .lds

If the data is LEEDR 4.0 object data, it will be saved as fname.lds unless *fname* contains an extension.

3.4.4.16 setLEEDRboundaryLayerAlt

Syntax

boundaryLayerAlt = setLEEDRboundaryLayerAlt(*timeIDX, season*)

boundaryLayerAlt = setLEEDRboundaryLayerAlt(*inputs*)

This function returns the LEEDR default boundary layer altitude based on the LEEDR Inputs structure. The inputs are

timeIDX [scalar/string]	Time of day index (or corresponding string).
season [scalar/string]	Season index - 1 for summer, 2 for winter (or corresponding string).
inputs [struct]	Input structure generated from the LEEDR GUI.

If the inputs are *timeIDX* and *season*, it is assumed the metType is 'ExPERT'.

The output is

boundaryLayerAlt [scalar]	LEEDR default boundary layer.

3.4.4.17 setLEEDRInputs

Syntax

LEEDRIn = setLEEDRInputs(*[inpt], [PName1], [PValue1], ...*)

This function generates LEEDR model data input structure for use with the function buildLEEDRAtmos. Requires p-code or m-code version of LEEDR available from the Air Force Institute of Technology. The inputs are

inpt [various]	(Optional) Can be output generated from the LEEDR GUI or the output resulting from a LEEDR run. If not input code will use results from the MATLAB® application data, if available, or the default LEEDR inputs.
PName [string]	(Optional) Name of parameter to be updated. Use `LEEDRLUT.getInputNames()` to obtain a list of parameters that can be changed. Parameter names are case sensitive.
PValue [varies]	(Optional) Value for parameter associated with *PName*. There must be a parameter value for each parameter name specified.

The output is a modified LEEDRLUT object based on the inputs with uncomputed outputs.

Example 3.4.50 (setLEEDRInputs) *This example shows how to call setLEEDRInputs.*

```
>> inputs = setLEEDRInputs('Latitude',39.83,'Longitude',-84.05)
```

Return input structure based on current application data if available or default inputs with a change to the location.

3.4.4.18 updateLEEDRAtmos

Syntax

inputs = updateLEEDRAtmos(*inOld*)

Use this function to update existing input data for LEEDR 3 to use with LEEDR 4. The input is

inOld [struct]	LEEDR 3.3 or older inputs structure.

The function will take the input from *inOld* convert the inputs for use with LEEDR 4 and then run LEEDR 4 to obtain the outputs. The user must have LEEDR 4 on the MATLAB® path. The output is

LEEDROut [LEEDRDLUT]	LEEDR LUT data object.

3.4.4.19 updateLEEDRInputs

Syntax

inputs = updateLEEDRInputs(*inOld*)

Use this function to update existing input data for LEEDR 3.3 to use with LEEDR 4.0.

inOld [struct]	LEEDR 3.3 or older inputs structure.

The output is

inputs [obj/struct]	LEEDR 4.0 input object, if LEEDR 4.0 is on the path, or LEEDR 4.0 inputs structure.

3.4.5 Functions to Support Turbulence Prediction Modeling

3.4.5.1 MZAneuralNetCn2Scaling

Syntax

SF = MZAneuralNetCn2Scaling(*net, inputs, [method], [dt]*)

This function applies neural network to the inputs and return the resulting scale factor. The inputs are

net [string/struct]	Filename with trained neural network information OR structure with fields 'deepnet' and 'classes,' as described below.
deepnet [network]	Trained neural network.
classes [vector]	Output classes.
inputs [vector]	Neural network input variables, , must match number of inputs required by *deepnet*.

method [string]	(Optional) Can be 'ML' for maximum likelihood (default), 'WS' for weighted average of scale factor (not recommended), or 'WC' for the weighted average of the class - default for RODN neural network.
dt [bool]	Boolean indicating day (true) or night (false). If omitted, daytime is assumed.

The function returns *SF* the resulting scale factor.

3.4.5.2 MZAsurfCn2

Syntax

```
[cn2, Wg, Lin] = MZAsurfCn2(DATE, lat, lon, alt, press, temp, wind,
windAlt, ...
        param, value, ...)
```

This function applies bulk modeling methods to estimate the ground level Cn2 value [50, 51, 52]. The inputs are

DATE [datetime]	Date of observation. Can be a MATLAB® datetime object of a date vector. If not a datetime object, GMT is assumed. Ignored if ground level solar irradiance is given.
lat [scalar]	(Optional) Latitude of observation point (deg). Ignored if ground level solar irradiance is given.
lon [scalar]	(Optional) Longitude of observation point (deg). Ignored if ground level solar irradiance is given.
alt [scalar]	Altitude for C_n^2 estimate (m AGL).
press [scalar]	Observed station pressure (Pa).
temp [scalar]	Observed temperature (K).
wind [scalar]	Observed wind speed (m/s).
windAlt [scalar]	Altitude at which the wind speed is measured (m AGL).
param [char]	Parameter name. Described below.
value [various]	Parameter value. Described below.

The inputs include possible param/value pairs as described below.

- 'CloudCondition' - Observed cloud conditions in a 3x2 matrix. Column 1 holds the fractional cloud coverages at each layer, column 2 holds the cloud layer altitude in ft AGL. Defaults to no clouds.
- 'SolarIrradiance' - Ground level solar irradiance (W/m^2)
- 'MinSolarIrr' - Minimum value for ground level solar irradiance. Use to possibly override the NaN return when solar irradiance is low. May still get NaN under low wind conditions. NOTE: The model does not support C_n^2 estimation under these conditions.

Description	albedo	evap const
'open ocean'	0.06	1.0
'conifer forest'	0.11	1.0
'deciduous trees'	0.165	1.0
'green grass'	0.25	1.0
'grass soil'	0.21	0.6
'bare soil'	0.17	0.2
'desert sand'	0.4	0.0
'fresh asphalt'	0.04	0.0
'worn asphalt'	0.1	0.0
'old concrete'	0.27	0.0
'new concrete'	0.55	0.0
'ocean ice'	0.6	0.5
'fresh snow'	0.06	1.0

Table 3.6: Possible terrain descriptions and corresponding albedo and evaporative constant values.

- 'Terrain' - Description of the site terrain (case insensitive) or surface [albedo, evap const]. Defaults to [0.17,0.5].

The outputs are

cn2 [scalar]	Estimate of ground level Cn2 ($m^{-2/3}$).
Wg [scalar]	Solar irradiance at the ground (W/m^2).
Lin [scalar]	Estimate of inner scale (m).

Example 3.4.51 (MZAsurfCn2) *Compute ground C_n^2 using various options in* MZAsurfCn2.

```
>> [cn2,Wg,Lin] = MZAsurfCn2([2016 7 29 12 58 0],38.818,-76.867,2, ...
                  1.0008e5,297.15,3.6011,10,'CloudCondition', ...
                  [0 0; 0.75 1000; 0.75 1400],'Terrain','green grass')
```

C_n^2, ground level solar irradiance, and inner scale given time, location, met data, cloud condition and terrain. Use date and location to compute ground solar irradiance.

```
>> [cn2,~,Lin] = MZAsurfCn2([],[],[],2,1.0008e5, ...
                  297.15,3.6011,10,'SolarIrradiance',395.9468, ...
                  'Terrain',[0.25,1.0])
```

C_n^2 and inner scale given met data, ground level solar irradiance, and terrain albedo and evaporative constant. Date and location are not needed since ground solar irradiance is passed in.

3.4.5.3 readMetData

Syntax

metData = readMetData(*fn, fmt, rowIndex*)

This function reads meteorological data from file in one of three different formats. The supported formats are

- fmt = 1: A CSV file provided by the 14th weather squadron with historical METAR data
- fmt = 2: (Most common) A text file with METAR data downloaded from https://climate.af.mil/WebObs/index.jsp (requires CAC) OR one-minute observation data

The inputs are

fn [string]	Filename to be read.
fmt [scalar]	Identifier for the data format. Described above.
rowIndex [varies]	Identifier for the data to extract. If a scalar it is the row number (not counting any header rows), if a date (datetime or date vector), will return data from the row with the closest date/time match. If *rowIndex* is a date vector, it is assumed to be in UTC time.

The output is a structure as described below.

metData [struct]	Structure containing meteorological data read from the input file.
DATE [datetime]	Date of observation time (Matlab datetime object).
site [string]	Site code.
lat [scalar]	Site latitude (deg decimal).
lon [scalar]	Site longitude (deg decimal).
alt [scalar]	Site altitude (m MSL).
temp [scalar]	Temperature (K).
dp [scalar]	Dewpoint (K).
press [scalar]	Pressure (Pa).
rh [scalar]	Relative humidity (%).
windSpeed [scalar]	Wind speed (m/s).
windDir [scalar]	Wind direction (deg).
vis [scalar]	Visibility (m).
cloudCond [matrix]	Observed cloud conditions in a 3x2 matrix. Column 1 holds the fractional cloud coverages at each layer, column 2 holds the cloud layer altitude in ft AGL.
surfCn2 [scalar]	Surface C_n^2 - NaN if fmt=1 or 2 ($m^{-2/3}$).
METAR [string]	Raw string data from the METAR file.

3.4.6 Specialized Mathematical Functions

3.4.6.1 FresnelInt

Syntax

[C, S] = FresnelInt(*x, [est]*)

```
MZA_SCI_COMP_API int FresnelInt(char** errorChain, const double* z,
    int nZ, bool est, double* C, double* S)

SCALING_CODE_API int FresnelInt(char** errorChain, const double* z,
    const int nZ, const bool est, double* fresnelCos,
    double* fresnelSin)
```

This function computes the value of the Fresnel sine and cosine integrals

$$C(x) - iS(x) = \int_0^x dt e^{-i\pi t^2/2}.$$

The Matlab inputs are:

x [vector]	Upper limit on integration.
est [scalar]	(Optional) Flag for computation method. TRUE for approximation and FALSE for quadl integration.

The function will either approximate the solution or use MATLAB® function `quadl` to compute, depending on the flag *est*. If *est* is TRUE, which is the default, the integrals will be approximated as [53]

$$\begin{aligned} C(x) &= \frac{1}{2} + f(x)\sin(\frac{\pi}{2}x^2) - g(x)\cos(\frac{\pi}{2}x^2) \\ S(x) &= \frac{1}{2} - f(x)\cos(\frac{\pi}{2}x^2) - g(x)\sin(\frac{\pi}{2}x^2) \end{aligned}$$

where

$$\begin{aligned} f(x) &= \sum_{n=0}^{11} f_n x^{-2n-1} \\ g(x) &= \sum_{n=0}^{11} g_n x^{-2n-1} \end{aligned}$$

and the values f_n and g_n can be found in the cited reference (or within the function). Note that both *C*(*x*) and *S*(*x*) have odd symmetry, therefore, one need only compute for positive values of *x*.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
z [in]	Upper limit on integration; length *nZ*.
nZ [in]	Number of elements in *z*.
est [in]	TRUE for estimation method.

fresnelCos [out]	Fresnel cosine integral; length *nV*.
fresnelSin [out]	Fresnel sine integral; length *nV*.

3.4.6.2 HYPER

Syntax

F = HYPER(*A, B, x, [N]*)

```
SCALING_CODE_API int HYPER(char** errorChain, const double* a,
     const int na, const double* b, const int nb, const double* zr,
     const double* zi, const int nz, const int* numTerms,
     double* funcReal, double* funcImag)
```

This function returns the complex hypergeometric ${}_aF_b(A;B;x)$ function using a series approximation with N terms where a is the length of the vector *A* and b is the length of the vector *B*. The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
a [in]	First argument to hypergeometric function
na [in]	Number of elements in *a*
b [in]	Second argument to hypergeometric function
nb [in]	Number of elements in *b*
zr [in]	(NULL OK) Real part of the evaluation points; must be length *nz*.
zi [in]	(NULL OK) Imaginary part of the evaluation points; must be length *nz*.
nz [in]	Number of evaluation points
numTerms [in]	(NULL OK) Number of terms in expansion. Defaults to 1000.
funcReal [out]	Real part of complex hypergeometric function at the evaluation points; allocate to length *nz*.
funcImag [out]	Imag part of complex hypergeometric function at the evaluation points; allocate to length *nz*.

3.4.6.3 HYPER2F3AS

Syntax

Fas = HYPER2F3AS(*B, D, z*)

```
SCALING_CODE_API int HYPER2F3AS(char** errorChain, const double* b,
    const double* d, const double* z, const int nz, double* fas)
```

This function returns an asymptotic approximation of ${}_2F_3(B, D, -z)$, for $z \gg 1$ [54]. This routine calculates the asymptotic expression for the generalized hypergeometric function for the case for which there are two numerator parameters and three denominator parameters. The asymptotic expression approximates ${}_2F_3(B, D, -z)$ when *z* is large and positive.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
b [in]	First argument to hypergeomtric function.
d [in]	Second argument to hypergeomtric function.
z [in]	Evaluation points.
nz [in]	Number of evaluation points.
fas [out]	Complex hypergeometric function evaluated at *z*; must be size *nz*.

3.4.6.4 LOMMEL

Syntax

[L M] = LOMMEL(*u, v, [tol]*)

```
MZA_SCI_COMP_API int LOMMEL(char** errorChain, double u,
    const double* v, int nV, const double* TOL, double* L, double* M)

SCALING_CODE_API int LOMMEL(char** errorChain, const double u,
    const double* v, const int nV, const double* TOL, double* L,
    double* M)
```

This function computes values of the Lommel functions *L* and *M* using U and V Lommel functions to within an absolute tolerance of *tol*. The Matlab inputs are:

u [scalar]	First argument - typically Fresnel number.
v [vector]	Second argument - typically scaled radius aperture coordinate.

`tol` [scalar]	(Optional) Tolerance or number of terms used in the series expansion. Defaults to 3.

The Lommel functions, `L` and `M`, are given by [17]

$$\begin{aligned}\frac{u}{2}L(u,v) &= \sin\frac{v^2}{2u} + V_0(u,v)\sin\frac{u}{2} - V_1(u,v)\cos\frac{u}{2} \\ &= U_1(u,v)\cos\frac{u}{2} + U_2(u,v)\sin\frac{u}{2} \\ \frac{u}{2}M(u,v) &= \cos\frac{v^2}{2u} - V_0(u,v)\cos\frac{u}{2} - V_1(u,v)\sin\frac{u}{2} \\ &= U_1(u,v)\sin\frac{u}{2} - U_2(u,v)\cos\frac{u}{2}\end{aligned}$$

where

$$\begin{aligned}V_0(u,v) &= J_0(v) - \left(\frac{v}{u}\right)^2 J_2(v) + \left(\frac{v}{u}\right)^4 J_4(v) + -\ldots, \\ V_1(u,v) &= \left(\frac{v}{u}\right) J_1(v) - \left(\frac{v}{u}\right)^3 J_3(v) + \left(\frac{v}{u}\right)^5 J_5(v) + -\ldots, \\ U_1(u,v) &= \left(\frac{u}{v}\right) J_1(v) - \left(\frac{u}{v}\right)^3 J_3(v) + \left(\frac{u}{v}\right)^5 J_5(v) + -\ldots, \\ U_2(u,v) &= \left(\frac{u}{v}\right)^2 J_2(v) - \left(\frac{v}{u}\right)^4 J_4(v) + \left(\frac{u}{v}\right)^6 J_6(v) + -\ldots\end{aligned}$$

and where $J_n(v)$ are Bessel functions of the first kind and order n. If `tol` is an integer, `tol` terms will be used in the series expansion. Otherwise, terms will be added until the difference is less than `tol`. When $v < u$, the Lommel functions are evaluated in terms of V_0 and V_1 and in terms of U_1 and U_2 when $v > u$. When $v = u$, these expressions become

$$\begin{aligned}V_0(u,u) &= \frac{1}{2}[J_0(u) + \cos u],\ V_1(u,u) = \frac{1}{2}\sin u, \\ U_1(u,u) &= \frac{1}{2}\sin u,\ U_2(u,u) = \frac{1}{2}[J_0(u) - \cos u]\end{aligned}$$

The API function returns system error codes and the arguments are:

`errorChain` [in,out]	Holds error and warning messages- deepest error is first.
`u` [in]	First argument - typically Fresnel number.
`v` [in]	Second argument - typically scaled radius aperture co-ordinate; length `nV`.
`nV` [in]	Number of elements in seconds argument.
`TOL` [in]	(NULL OK) Absolute tolerance OR if integer, number of terms in expansion - defaults to 3.
`L` [out]	Lommel function `L`; length `nV`.
`M` [out]	Lommel function `M`; length `nV`.

3.4.6.5 SGAMMA

Syntax

out = SGAMMA(*a*, *b*)

```
SCALING_CODE_API double SGAMMA(const double* a, const int p,
    const double* b, const int q)
```

This function returns a ratio of products of gamma functions given

a [vector]	Vector of numerator gamma function arguments.
b [vector]	Vector of denominator gamma function arguments.

If *a* is a vector of length p and *b* is a vector of length q, then the resulting ratio is given by

$$\Gamma(a,b) = \frac{\Gamma(a_1)\Gamma(a_2)...\Gamma(a_p)}{\Gamma(b_1)\Gamma(b_2)...\Gamma(b_q)}.$$

where x_i represents the ith element of the vector x.

The API function returns system error codes and the arguments are:

a [in]	Numerator gamma function arguments.
p [in]	Number of elements in *a*.
b [in]	Denominator gamma function arguments.
q [in]	Number of elements in *b*.

3.4.6.6 ZERNIKEORDER

Syntax

[n, m] = ZERNIKEORDER(*zi*)

```
SCALING_CODE_API int ZERNIKEORDER(char** errorChain,
    const double* zernikeIndex, const int nZernikeIndices,
    double* radialOrder, double* azimuthalOrder)
```

This function computes the radial and azimuthal order of Zernike polynomial using Noll's ordering scheme [21]. The input is *zi* the index of the Zernike polynomial. The outputs are *n* and *m* which are the radial and azimuthal orders of the Zernike polynomial. Both *n* and *m* will be nonnegative integers satisfying $m \leq n$ and $n - m =$ even.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
zernikeIndex [in]	Index of Zernike polynomial.
nZernikeIndices [in]	Number of elements in *zernikeIndex* and outputs.
radialOrder [out]	Radial order of Zernike polynomial; must be size *nZernikeIndices*.
azimuthalOrder [out]	Azimuthal order of Zernike polynomial; must be size *nZernikeIndices*.

3.4.7 API Support Functions

3.4.7.1 ATMLookupPath

Syntax

p = ATMLookupPath(*[LUTpath]*)

This function is used to obtain and, optionally, set the path to lookup table files for ATMTools. The input *LUTpath* should only be passed to change the lookup path for ATMTools and must contain the full path to the desired directory. It is not recommended that one change the lookup path for ATMTools as several functions in ATMTools rely on data in that directory. The function will return the current setting for the full path to the ATMTools data directory. This can also be obtained using LookupPath('ATM').

3.4.7.2 LEEDRLookupPath

Syntax

p = LEEDRLookupPath(*[LUTpath]*)

This function is used to obtain and, optionally, set the path to lookup table files for LEEDR profile data. The input *LUTpath* should only be passed to change the lookup path for LEEDR profile data and must contain the full path to the desired directory. The function will return the current setting for the full path to the LEEDR data directory. This can also be obtained using LookupPath('LEEDR').

3.4.7.3 LEEDRWxCubeLookupPath

Syntax

p = LEEDRWxCubeLookupPath(*[LUTpath]*)

This function is used to obtain and, optionally, set the path to lookup table files for LEEDR weather cubes. The input *LUTpath* should only be passed to change the lookup path for LEEDR Wx cubes and must contain the full path to the desired directory. The function will return the current setting for the full path to the LEEDR weather cube data directory. This can also be obtained using LookupPath('LEEDRWxCube').

3.4.7.4 LookupPath

Syntax

p = LookupPath(*LUTtype, [LUTpath]*)

This function returns the path to lookup table data files for the specified lookup table type. The possible values for *LUTtype* are 'AERO', 'AIRS', 'ATM', 'LEEDR', 'LEEDRWxCube', or 'SHARE'. *LUTtype* defaults to 'ATM' if not specified. Can pass in *LUTpath* to set the full path to directory containing the desired lookup table data. Only pass in LUTpath to change the lookup path for the LUT type.

3.4.7.5 saveMODFASAsH5

Syntax

saveMODFASAsH5(*[fileName], [S]*)

Save MODFAS data to an h5 file for use with the Scaling Code API. With no inputs, loads ModFasData.mat and saves as ModFasData.h5. Can optionally pass in a structure, S, which is the result of S=load(fileName). File is saved to the default lookup path, as returned by LookupPath('ATM').

3.4.8 General Utility Functions and User Interfaces

3.4.8.1 addModelType

Syntax

AllModels = addModelType(*[modelType]*)

Use this function to add a model type to the standard models recognized by AtmStruct and ChangeAtm. Use with no inputs to get a cell array of models available.

modelType [string/cell/list]	(Optional) String, cell array of strings or comma-separated list of model types to add.

The function addModelType works by setting the global variable *Standard_Models*. When MATLAB® is restarted or when global variables are cleared (as when command `clear all` is issued), all added model types will also be cleared. Non-standard model types can be added to an Atm structure without using addModelType, but they cannot use model functions to compute data and they will not be reset if the geometry or screens changes. The output is

AllModels [cell]	Cell array of model types recognized in ATMTools.

Example 3.4.52 (addModelType) *This examples illustrates adding a model type*

```
>> Atm = AtmStruct(5000,10,10000,100,'ScatDens',ones(100,1));
>> Atm2 = ChangeAtm(Atm,'screens',10)

Atm2 =

          hp: 5000
          ht: 10
          rd: 1.0000e+004
           L: 1.1179e+004
     LatLong: [1x1 struct]
      MaxAlt: Inf
      xRange: [0 1]
           z: [10x1 double]
          dz: [10x1 double]
           h: [10x1 double]
    ScatDens: [100x1 double]
```

Set scattering density as a non-standard model in `Atm`*. Attempt to change number of screens in* `Atm2` *but* `Atm2.ScatDens` *is still 100×1.*

```
>> m = addModelType('ScatDens')

m =

    'Cn2'
    'Abs'
    'Scat'
    'Ext'
    'Temp'
    'Wind'
    'WindHeading'
    'Press'
    'Dens'
    'Lin'
    'Lout'
    'RH'
    'DewPt'
    'ScatDens'
```

Adds model type 'ScatDens' for scattering density to the standard model types recognized within ATMTools. Now one can use ChangeAtm *with an Atm structure containing a model for 'ScatDens' and that model will be updated as well.*

```
>> Atm3 = AtmStruct(5000,10,10000,100,'ScatDens','UniformAtm',1);
>> Atm4 = ChangeAtm(Atm3,'screens',10)

Atm4 =

              hp: 5000
              ht: 10
              rd: 1.0000e+004
               L: 1.1179e+004
         LatLong: [1x1 struct]
          MaxAlt: Inf
          xRange: [0 1]
```

```
            z: [10x1 double]
           dz: [10x1 double]
            h: [10x1 double]
     ScatDens: [10x1 double]
ScatDensEval: {'UniformAtm'  'h'  [1]}
```

Now 'ScatDens' is recognized as a standard model type and changing number of screens updates values for `Atm3.ScatDens`.

3.4.8.2 AeroOpticsScaling

Syntax

[opdScaling, frqScaling, dimScaling] = AeroOpticsScaling(*[scalingType], ... hp, vp, Dt, lambda, [Dap], [hp0], [vp0], [Dt0], [lambda0], [Dap0]*)

```
SCALING_CODE_API int AeroOpticsScaling(char** errorChain,
    char scalingType, const int n, const double* hp, const double* vp,
    const double* turDiam, const double* lambda, const double* apDiam,
    const double* hp0, const double* vp0, const double* turDiam0,
    const double* lambda0, const double* apDiam0, double* opdScaling,
    double* frqScaling, double* dimScaling)
```

This function returns aero-optics scaling factors as a function of platform altitudes, velocities, and turret diameters.[37], [55] Allows for vectorized inputs, but each nonscalar input must be of the same length. Outputs will then be vectors of the same length as the input vectors. The Matlab inputs are:

scalingType [char]	(Optional) Type of scaling to compute. Specify either 'Tilt' or 'HO'. Defaults to 'HO'.
hp [vector]	New platform altitude (above sea level) (m).
vp [vector]	New platform velocity (m/s).
Dt [vector]	New turret diameter (m).
lambda [vector]	New wavelength (m).
Dap [vector]	(Optional) Aperture diameter (m). Not used for higher-order scaling.
hp0 [vector]	(Optional) Original platform altitude (m).
vp0 [vector]	(Optional) Original platform velocity (m/s).
Dt0 [vector]	(Optional) Original turret diameter (m).
lambda0 [vector]	(Optional) Original wavelength (m).
Dap0 [vector]	(Optional) Original aperture diameter (m). Not used for higher-order scaling.

If scaling from normalized data, only the new platform altitude, velocity, turret diameter and wavelength need to be specified as inputs. If scaling from one set of conditions (*hp0*,*vp0*,*Dt0*,*lambda0*,[*Dap0*]) to a new set of conditions (*hp*,*vp*,*Dt*,*lambda*,[*Dap0*]), all input values should be specified.

The Matlab outputs are:

opdScaling [vector]	Scaling factor applied to aero-optics OPDs.
frqScaling [vector]	Scaling factor applied to aero-optics frequency.
dimScaling [vector]	Scaling factor applied to aero-optics x/y dimensions.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages, deepest error is first.
scalingType [in]	Specify either 'T'/'t' for tilt or 'H'/'h' for higher order; if character is unrecognized, a warning is issued and higher order is used.
scaleFromNonDimFlag [in]	Indicates whether scaling from non-dimensionalized data.
hp [in]	New platform altitude (above sea level) (m).
vp [in]	New platform velocity (m/s).
turDiam [in]	New turret diameter (m).
lambda [in]	New wavelength (m).
apDiam [in]	Aperture diameter (m); ignored for higher-order.
hp0 [in]	Original platform altitude (m); ignored if *scaleFromNonDimFlag* = 0.
vp0 [in]	Original platform velocity (m/s); ignored if *scaleFromNonDimFlag* = 0.
turDiam0 [in]	Original turret diameter (m); ignored if *scaleFromNonDimFlag* = 0.
lambda0 [in]	Original wavelength (m); ignored if scaleFromNonDimFlag = 0.
apDiam0 [in]	Original aperture diameter (m); ignored for higher-order or if *scaleFromNonDimFlag* = 0.
opdScaling [out]	Scaling factor applied to aero-optics OPDs.
frqScaling [out]	Scaling factor applied to aero-optics frequency.
dimScaling [out]	Scaling factor applied to aero-optics x/y dimensions.

3.4.8.3 AreaAdditionStrehl

Syntax

S = AreaAdditionStrehl(*StrehlTerms*)

```
SCALING_CODE_API int AreaAdditionStrehl(char** errorChain,
    const double* strehlTerms, const int nInputVectors,
    const int nTermsPerVector, double* areaAdditionStrehl)
```

This function combines individual Strehl ratio terms via an area addition method [13]. The area addition method combines degradations that not only reduce the peak irradiance of the beam but also contribute to a spreading of the beam, or increase in area. The input, *StrehlTerms*, is a vector or matrix of individual Strehl ratio terms. If in matrix form, the area addition is computed across the second dimension. The output *S* is the composite Strehl ratio given by

$$\mathrm{S} = \left(\sum_{i=1}^{N} \mathrm{StrehlTerms}_i^{-1} - (N-1) \right)^{-1}$$

where N is the length of *StrehlTerms*. *S* will be scalar if *StrehlTerms* is a vector, and will be a vector if the input is a matrix.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
strehlTerms [in]	Strehl ratio terms to sum; summation takes place over sequential elements separately for each input vector.
nInputVectors [in]	Number of input vectors to independently reduce; this will also be the number of outputs.
nTermsPerVector [in]	The actual number of Strehl terms to combine; strehlTerms will be size *nInputVectors* x *nTermsPerVector*.
areaAdditionStrehl [out]	Composite Strehl ratio of individual terms; must be of length *nInputVectors*.

3.4.8.4 AtmStruct_Interface

Syntax

AtmOut = AtmStruct_Interface(*[AtmIn]*)

This function opens the Graphical User Interface (GUI) for interfacing with an Atm structure. While the GUI window is open, execution of commands at the MATLAB® command prompt will be blocked. The input *AtmIn* is a structure from AtmStruct and is optional. The output *AtmOut* is the modified structure as from AtmStruct. If the optional output is not declared, the variable *Atm* will be assigned to the base workspace when the "Commit & Close" button is pushed.

3.4.8.5 AtmStruct2Cell

Syntax

[CellOut, CellNames] = AtmStruct2Cell(*Atm, [LaunchExcel]*)

Converts an *Atm* structure or array of structures into cell arrays for viewing with Microsoft Excel. The inputs are:

Atm [struct]	Structure or array of structures from AtmStruct.
LaunchExcel [bool]	1 to display structures in an Excel Worksheet. Defaults to 0.

The outputs are:

CellOut [cell]	Cell array of cells containing structure field names and model data.
CellNames [cell]	Cell array of strings indicating ordering of data in *CellOut*.

If the input *Atm* structure is continuous, the output will be a single element cell array with the model evaluations that would get put into a single Excel spreadsheet. Otherwise, each element of the output *CellOut* will be a cell array of data. *CellOut*{*1*} is a cell array containing geometry information including *Atm.z* and *Atm.dz*. Each additional cell array in *CellOut*{*2:end*} contains data for each model in the input *Atm* structure. If sent to Excel by setting the input *LaunchExcel* to true, there will be an Excel spreadsheet titled 'Geom' containing *z* and *dz* and additional spreadsheets for each model in the input *Atm* structure.

Example 3.4.53 (AtmStruct2Cell) *Some examples of using* AtmStruct2Cell *for creating cells for display of an array of* Atm *structures.*

```
>> Atm = ChangeAtm(AtmStruct,'alpha',[0 1 2 3 4]);
>> CellOut = AtmStruct2Cell(Atm,true)
```

Display data from all Atm structures in Excel. CellOut{1} *is a cell array with z and dz.* CellOut{2:end} *contains data for the models in the* Atm *struct array.*

```
>> [CellOut,CellNames] = AtmStruct2Cell(AtmStruct,false);
>> Cn2 = CellOut{strcmp(CellNames,'Cn2')}
```

Get output from AtmStruct2Cell *and display the cell array containing* C_n^2 *data.*

3.4.8.6 ExtractBaseModel

Syntax

BaseModel = ExtractBaseModel(*ModelEval*)

This function extracts the base model from the model evaluation from an Atm structure. It removes the outer-most wrapper function, i.e. AverageAtm, BoundaryAtm, or TerrainAtm. The input is

ModelEval [cell]	Full model evaluation as from an Atm structure.

The output is

BaseModel [cell]	Base model evaluation.

This function is not recursive. If the model evaluation uses more than one wrapper function (i.e. AverageAtm(BoundaryAtm)), the function must be called multiple times to get to the primary base model.

Example 3.4.54 (Extracting models) *This example illustrates how to use* ExtractBaseModel.

```
>> Atm = AtmStruct(1230,5000,5000,10,'Cn2','TerrainAtm',1200,'HV57');
>> BaseModel = ExtractBaseModel(Atm.Cn2Eval)
```

Extract the base turbulence model.

```
>> Atm2 = ChangeAtm(Atm,'Cn2',ExtractBaseModel(Atm.Cn2Eval))
```

Create a new Atm structure using the base model of an existing Atm structure.

3.4.8.7 FAS_TAPE6

Syntax

[Trans v] = FAS_TAPE6(*lambda, FileDir*)

This function is used by FASCODE to extract the output of the FORTRAN executable FASCODE from the TAPE6 file. The inputs are

lambda [scalar]	Wavelength to extract the results for (m)
FileDir [string]	The directory in which to find the TAPE6 file.

The outputs are

Trans [scalar]	Transmission under conditions of the run.
v [string]	FASCODE version used for computing transmission.

3.4.8.8 GETALPHA

Syntax

alpha = GETALPHA(*Atm, [ModelName]*)

Given a structure or array of structures from ATMSTRUCT, GETALPHA returns a vector of multipliers applied to the base model function for the specified effect. The inputs are

Atm [struct]	Structure or array of structures from ATMSTRUCT.
ModelName [string]	(Optional) Name of effect in *Atm* for which to extract the multiplier. Defaults to 'Cn2'.

The ouput, *alpha*, is a vector of multipliers on *ModelName*.

Example 3.4.55 (getAlpha) *Examples of using* GETALPHA *to extract multipliers on models in* *Atm*.

```
>> alpha = getAlpha(Atm);
```

Get the multiplier on C_n^2 from the input Atm

```
>> Atm = ChangeAtm(Atm,'Wind',{'Bufton';'4*Bufton';'10*Bufton'}');
>> alpha = getAlpha(Atm,'Wind')

alpha =

     1     4    10
```

Return the multiplier applied to the Bufton model for wind speed.

3.4.8.9 mkFuncStr

Syntax

str = mkFuncStr(*cellArray*)

This function converts the model evaluation cell array from an atmospheric structure into a pretty function call string for display purposes. The input is

cellArray [cell]	Cell array holding the functional evaluation information for an atmospheric model.

The output is

str [char/cell]	Character array (or cell array in the case of array of cell arrays) with the displayable function call.

Example 3.4.56 (Converting Model Evaluation to a String) *This example illustrates how to convert the cell array model evaluation to a string.*

```
>> Atm = AtmStruct;
>> str = mkFuncStr(Atm.Cn2Eval)
```

Output a string containing the turbulence model evaluation. Useful for plot titles, etc.

```
>> Atm = ChangeAtm(AtmStruct,'alpha',[2,3,4]);
>> ca = mkFuncStr({Atm.Cn2Eval});
```

Output a cell array for an array of ATM structures. Useful for adding a legend to a atmospheric model plot.

3.4.8.10 MOD_TAPE6

Syntax

[ExtCoeff, Wn, v] = MOD_TAPE6(*FileName, lambda*)

This function is used by MODTRAN to extract the output of the FORTRAN executable MODTRAN from the TAPE6 file given *lambda* in microns. The outputs are

ExtCoeff [vector]	The extinction coefficients under the conditions of run, which has the following columns: total ext, mol ext, molabs, mol scat, aer ext, aer abs, aer scat.
Wn [scalar]	Wave number (1/cm).
v [string]	The version of MODTRAN used to create the TAPE6 file.

3.4.8.11 PhasorSumStrehl

Syntax

S = PhasorSumStrehl(*fld, [mask]*)

```
SCALING_CODE_API int PhasorSumStrehl(char** errorChain,
     const int Npts, const int Ng, const double* fldREAL,
     const double* fldIMAG, const double* mask, double* S)
```

This function computes the Strehl ratio of the input field with optional input mask using a phasor-sum Strehl method. Strehl is defined as on-axis intensity in the far field for the input field, normalized to the on-axis intensity in the far field for a uniform input field with the same total power as the input field. The Matlab inputs are:

fld [matrix]	Complex field value array (arbitrary units).
mask [matrix]	(Optional) Aperture *mask* to be applied to complex field value array.

If not input *mask* defaults to the entire grid. The Strehl is then calculated as

$$S = \frac{|\int M(x,y)F(x,y)dA|^2}{\int M(x,y)|F(x,y)|^2dA \int M(x,y)^2dA} \tag{3.11}$$

where $M(x,y)$ is the mask and $F(x,y)$ is the field.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
Npts [in]	Number of elements in each input field array, usually *Nx* x *Ny*.
Ng [in]	Number of inputs field arrays.

fldREAL [in]	Complex field value array (arbitrary units); must be *Npts* x *Ng*.
fldIMAG [in]	Complex field value array (arbitrary units); must be *Npts* x *Ng*.
mask [in]	Aperture mask to be applied to complex field value array; NULL ok, defaults to entire grid; must be length *Npts*.
S [out]	Strehl ratio of input field; allocate to length *Ng*.

3.4.8.12 PLOTATM

Syntax

ax = PLOTATM(*[ax], Atm, [xVar], [ModelName], [plotType], [xVarY]*)

This function creates plots of data from an input *Atm* structure. Also adds a Plot Options menu to the figure window allowing user to manipulate the plot as well as select different data to plot. The inputs are

Parameters

ax [handle]	(Optional) Axis handle for the plot.
Atm [struct]	Atmospheric structure as from ATMSTRUCT.
xVar [char]	(Optional) Name of independent variable for the plot. One of 'x', 'z', or 'h'. Defaults to 'z'.
ModelName [char/cell]	(Optional) Name of model to plot. Can be any of the models in the *Atm* structure or a cell array of models to plot together, e.g. {'Abs', 'Scat'}. Defaults to 'Cn2'. If both 'Abs' and 'Scat' are present, one can also plot 'Ext' (the sum of the two).
plotType [char]	(Optional) Plot type to display. Either 'Continuous', 'Discrete', or 'Integrated'.
xVarY [logical]	(Optional) Flag indicating whether to plot the independent variable on the y-axis. Default is false.

The only required input is the *Atm* structure. The function also can return the function handle to the axes for further manipulation.

ax [handle]	Handle to resulting axes.

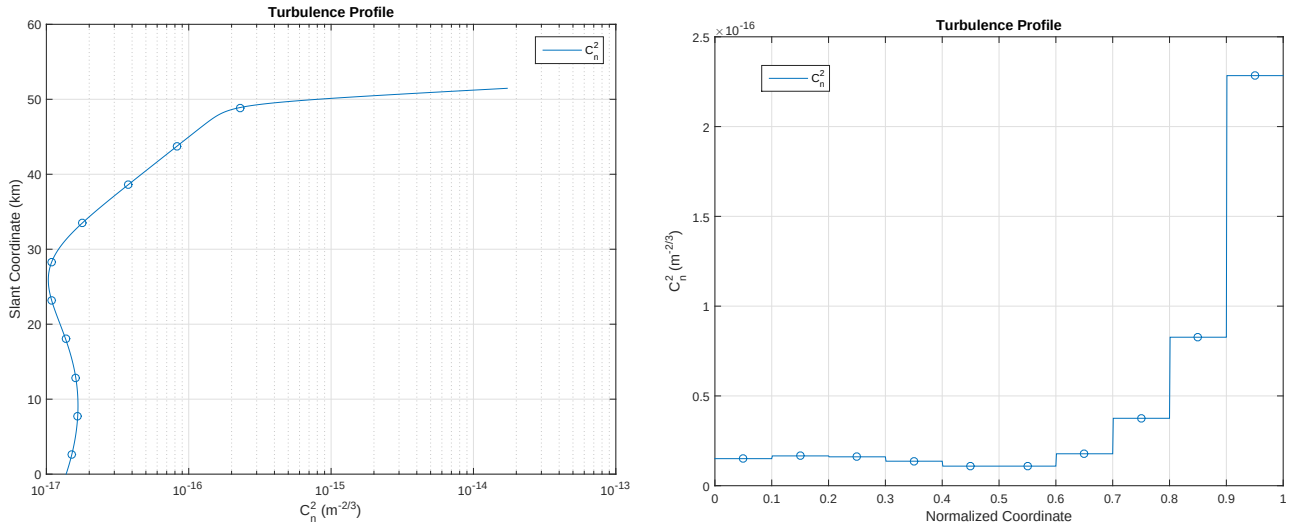
Example 3.4.57 (Using PLOTATM) *The following example creates plots using* PLOTATM.

```
>> Atm = AtmStruct(GeomStruct,10,'Cn2','HV57');
>> ax = plotAtm(Atm,'z','Cn2','Continuous',true)
>> set(ax,'XScale','log')
```

Create an atmospheric structure and use PLOTATM *to create a continuous plot of* C_n^2 *vs range. Then use the output axes handle to set the scale for the* C_n^2 *axis, which is the x-axis since* ***xVarY*** *is true, to logarithmic.*

```
>> plotAtm(Atm,'z','Cn2','Discrete')
```

Create a discrete plot of C_n^2 *vs normalized distance along the path.*



(a) (b)

Figure 3.35: Examples of plots from PLOTATM. (a) Continuous plot with C_n^2 on the horizontal axis and (b) a discrete plot of C_n^2 .

3.4.8.13 PROPCONFIG

Syntax

CNew = PROPCONFIG(*C*)

[AtmNew, GNew] = PROPCONFIG(*Atm, [G]*)

ttNew = PROPCONFIG(*tt*)

[AtmNew, GNew] = PROPCONFIG(*[fileName]*)

PC = PROPCONFIG(*'SaveFile', C, fileName*)

PROPCONFIG provides the user the ability to set up geometry and atmosphere, get guidance on setting propagation mesh parameters for a wave optics simulation and then run an open-loop simulation to test parameters. The user can set up the geometry and atmosphere from the GUI starting with some default settings or input either Atm and G structures or a C structure that has already been computed. One of the advantages of PROPCONFIG is that it allows novice users to setup atmosphere and geometry using some of the more advanced functionality of ATMTools that would be difficult to do via the command line. The optional inputs are

C [struct]	(Optional) An engagement structure from CaseStruct.
Atm [struct]	Atmospheric structure from AtmStruct.
G [struct]	(Optional) Geometry structure as from GeomStruct.
tt [struct]	Structure from TurbTool.
fileName [string]	(Optional) Name of MATLAB® data file saved from either PropConfig or TurbTool.

If no inputs are specified, the GUI is populated with default parameters.

The function can output either a *C* structure, *Atm* and *G* structures, or a structure as from TurbTool as follows:

CNew [struct]	Engagement structure as from CaseStruct with any changes made.
AtmNew [struct]	Atmospheric structure as from AtmStruct with any changes made.
GNew [struct]	Geometry structure as from GeomStruct with changes.
ttNew [struct]	TurbTool structure with changes.

If outputs are specified, MATLAB® will pause execution of anything other than callbacks using the function uiwait. With no outputs, other commands and functions can be run while the GUI is open. However, the primary output is a MATLAB® data file. The data file contains all information available in the GUI and can be loaded into PropConfig or MATLAB® later or loaded for a wave-optics simulation. Section 3.3.0.25 describes in more detail use of PropConfig and Section 3.3.0.26 describes how to use the data file saved by PropConfig in a WaveTrain™ simulation.

One can also save a data file without opening the GUI, using default options for computing mesh parameters by passing as the first argument the string `'SaveFile'` followed by a *C* structure and the name of the file to save. The file will be saved to the current working directory unless the input file name includes the path. If the file name is omitted or empty, a structure of data that would be saved, *PC*, will be returned.

3.4.8.14 PropControl_Interface

Syntax

[PC PropType] = PropControl_Interface(*PCin*)

User interface for setting fields of the propagation control structure for TBWaveCalc. Primarily used in PropConfig. The optional input is

PCin [struct]	(Optional) The propagation control structure to be modified. Defaults to *PC*=TBWaveCalc.

The outputs are

PC [struct]	Modified propagation control structure.
PropType [scalar]	Flag indicating whether spherical wave propagation is allowed with the specified DLL.

Figure 3.36 shows the PropControl_Interface window with default settings.

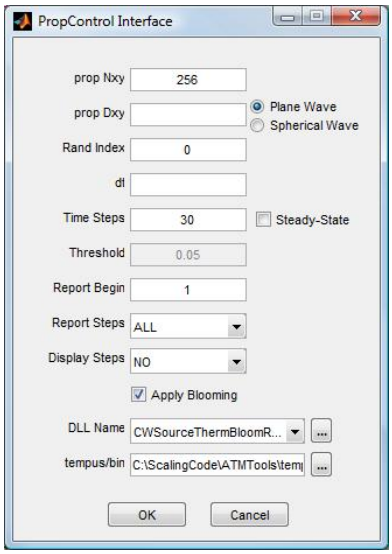


Figure 3.36: PropControl_Interface with default parameter settings.

3.4.8.15 PropMeshParams

Syntax

[nxy, dxy1, [dxy2]] = PropMeshParams(*lambda, D1, D2, Atm, ... [cturb], [nxy_in], [dxy1_in], [dxy2_in], [dxy1_max], [dxy2_max]*)

[nxy, dxy1, [dxy2]] = PropMeshParams(*lambda, D1, D2, L, [r0sph1], ... [r0sph2], [cturb], [nxy_in], [dxy1_in], [dxy2_in], ... [dxy1_max], [dxy2_max]*)

```
SCALING_CODE_API int PropMeshParams(char** errorChain,
    const double wavelength, const double apDiamA, const double apDiamB,
    const double pathLength, const double* r0SphA, const double* r0SphB,
    const double* cTurb, const double* nxyIn, const double* dxyInA,
    const double* dxyInB, const double* dxyMaxA, const double* dxyMaxB,
    double* nxy, double* dxyA, double* dxyB)
```

This function computes mesh spacings and dimensions for wave optics propagation based on [56]. Can be used for both plane wave and spherical wave propagation geometries. The Matlab inputs are:

lambda [scalar]	Wavelength (m).
D1 [scalar]	Diameter of aperture or region of interest at plane 1 (m).
D2 [scalar]	Diameter of aperture or region of interest at plane 2 (m).
Atm [struct]	Atmospheric structure from AtmStruct or a comma-separated list of (..., *L*, [*r0sph1*], [*r0sph2*], ...).
L [scalar]	Propagation distance (m).
r0sph1 [scalar]	(Optional) Spherical wave coherence radius at plane 1 (m). If not specified, *r0sph1*=1.0e20, i.e. effectively vacuum.
r0sph2 [scalar]	(Optional) Spherical wave coherence radius at plane 2 (m). If not specified, *r0sph2*=*r0sph1*.
cturb [scalar]	(Optional) Scale factor for turbulence blurring (see references) (m). If not specified, *cturb*=2.0.
nxy_in [scalar]	(Optional) Desired mesh dimension.
dxy1_in [scalar]	(Optional) Desired mesh spacing at plane 1.
dxy2_in [scalar]	(Optional) Desired mesh spacing at plane 2.
dxy1_max [scalar]	(Optional) Maximum mesh spacing at plane 1.
dxy2_max [scalar]	(Optional) Maximum mesh spacing at plane 2.

In order to specify any of the optional inputs, all preceding optional inputs must be specified.

The Matlab outputs are:

nxy [scalar]	Computed mesh dimension.
dxy1 [scalar]	Computed mesh spacing at plane 1.
dxy2 [scalar]	Computed mesh spacing at plane 2 (spherical wave propagation only).

If the output *dxy2* is not specified, the output will be for plane wave propagation. Otherwise, spherical wave propagation settings will be returned as shown in Example 3.4.58.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
wavelength [in]	Wavelength (m).
apDiamA [in]	Diameter of aperture or region of interest at plane 1 (m).
apDiamB [in]	Diameter of aperture or region of interest at plane 2 (m).
pathLength [in]	Propagation distance (m).
r0SphA [in]	(NULL OK) Spherical wave coherence radius at plane 1 (m); if not specified equal to 1.0e20 (i.e. effectively vacuum); only first element used.
r0SphB [in]	(NULL OK) Spherical wave coherence radius at plane 2 (m); if not specified equal to *r0SphA*; only first element used.
cTurb [in]	(NULL OK) Scale factor for turbulence blurring (see references); if not specified *cTurb* =2.0; only first element used.
nxyIn [in]	(NULL OK) Desired mesh dimension; only first element used.
dxyInA [in]	(NULL OK) Desired mesh spacing at plane 1; only first element used.
dxyInB [in]	(NULL OK) Desired mesh spacing at plane 2; only first element used.
dxyMaxA [in]	(NULL OK) Maximum mesh spacing at plane 1; only first element used.
dxyMaxB [in]	(NULL OK) Maximum mesh spacing at plane 2; only first element used.
nxy [out]	Computed mesh dimension; scalar.
dxyA [out]	Computed mesh spacing at plane 1; scalar.

dxyB [out]	(NULL OK) Computed mesh spacing at plane 2; scalar; if not NULL spherical wave propagation will be modeled; if NULL planar propagation will be modeled.

Example 3.4.58 (PropMeshParams) *Shows how to call* PropMeshParams *to determine either spherical or plane wave propagation mesh settings.*

```
>> Atm = AtmStruct(3000,50,10000,10,'Cn2','HV57');
>> [nxy_min, dxy1_max, dxy2_max] = PropMeshParams(1e-6, 0.5, 0.1, Atm)

nxy_min =

    33


dxy1_max =

    0.0337


dxy2_max =

    0.0094

>> [nxy, dxy1, dxy2] = PropMeshParams(1e-6, 0.5, 0.1, Atm, [],128)

nxy =

   128


dxy1 =

    0.0145


dxy2 =

    0.0040
```

Determine the nominal settings for grid size and spacing for spherical wave propagation, then call PropMeshParams *again to determine grid spacings based on a desired value for grid size.*

```
>> [nxy_min, dxy_max] = PropMeshParams(1e-6, 0.5, 0.1, Atm)

nxy_min =

    49


dxy_max =
```

```
    0.0145

>> [nxy, dxy] = PropMeshParams(1e-6, 0.5, 0.1, Atm, [], 128)

nxy =

   128


dxy =

    0.0079
```

Determine the minimum grid size and maximum grid spacing for plane wave propagation, then call PROPMESHPARAMS *again to determine grid spacings based on a desired value for grid size.*

3.4.8.16 RECOMMENDEDMESH

Syntax

mp = RECOMMENDEDMESH(*PropType, lambda, D1, D2, Atm, [cTurb], ... [minNpts], [n], [d1], [d2]*)

This function computes recommended mesh parameters based on nominal parameters returned from PROPMESHPARAMS. The inputs are

PropType [scalar]	Propagation type: 0 - plane wave, 1 - spherical wave.
lambda [scalar]	Wavelength for propagation (m).
D1 [scalar]	Diameter of interest in the source plane (m).
D2 [scalar]	Diameter of interest in the receiver/target plane (m).
Atm [struct]	Atmospheric structure from ATMSTRUCT.
cTurb [scalar]	(Optional) Scale factor for turbulence blurring (m). Defaults to 2.0.
minNpts [scalar/vector]	Minimum number of pixels in the diameter of interest. Specify as a 2-element vector to use different values for source aperture and for the receiver aperture.
n [scalar]	(Optional) Desired number of mesh points.
d1 [scalar]	(Optional) Desired pixel size for plane wave propagation or pixel size at the platform for spherical wave propagation (m).
d2 [scalar]	(Optional) Desired pixel size for the target/receiver plane for spherical wave propagation (m). Not used if *PropType*=0.

RecommendedMesh first calls PropMeshParams to obtain nominal propagation mesh parameters. If a desired mesh dimension is not input, the nominal mesh dimension is rounded up to the next power of 2 and the mesh spacing is recomputed. If a user does specify a desired number of mesh points and/or pixel size combination that is invalid, RecommendedMesh will compute and return values for mesh dimension and spacing, whereas PropMeshParams returns 0 for mesh dimension and *NaN* for mesh spacing. The output structure returns the computed mesh parameters, nominal parameters, and flags indicating whether computed mesh parameters are valid. The output is a structure as follows.

mp [struct]	Structure of parameters for help in setting mesh values.
mp.propnxy [scalar]	Number of mesh points based on inputs.
mp.propdxy [scalar/vector]	Pixel size based on inputs (m). Scalar for plane wave propagation or 2-element vector for spherical wave propagation.
mp.nxyValid [logical]	Flag indicating whether number of mesh points computed is valid.
mp.dxyValid [logical]	Flag indicating whether computed mesh spacing is valid.
mp.nxymin [scalar]	Minimum/nominal value for number of mesh points.
mp.dxymax [scalar/vector]	Maximum/nominal value for mesh spacing. Scalar for plane wave propagation or 2-element vector for spherical wave propagation.

3.4.8.17 ReverseAreaAddition

Syntax

S = ReverseAreaAddition(*S_Total, StrehlTerms*)

```
SCALING_CODE_API int ReverseAreaAddition(char** errorChain,
    const double* sTotal, const double* strehlTerms, const int rows,
    const int cols, double* strehlRatio)
```

This function is used to determine a component Strehl required for the area addition of all Strehl terms to yield the input composite Strehl [13]. The Matlab inputs are:

S_Total [vector]	Composite Strehl.
StrehlTerms [vector/matrix]	Vector (Nx1) or array of individual Strehl ratio terms. The area addition is computed across the second dimension.

The Matlab output is:

S [scalar/vector]	Component Strehl ratio of individual terms.

The output component Strehl is computed as

$$S = \left(\frac{1}{S_{Total}} + N - \sum_{i=1}^{N} S_i^{-1} \right)^{-1}$$

where S_i represents each of the input Strehl terms.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
sTotal [in]	The composite Strehl; must be length rows.
strehlTerms [in]	The array of individual Strehl ratio terms; must be length *rows* x *cols*.
rows [in]	The leading dimension of *strehlTerms*.
cols [in]	The second dimension of *strehlTerms*.
strehlRatio [out]	The component Strehl ratio of individual terms; allocate to length *cols*.

3.4.8.18 ReverseAtm

Syntax

[Atm G] = ReverseAtm(*AtmIn, [GIn], [xRangeIn]*)

This function outputs an atmospheric structure and a geometry structure for the propagation from target to platform for use in computing propagation parameters, such as r_0, for the reverse path. The inputs are

AtmIn [struct]	Atmospheric structure from AtmStruct.
GIn [struct]	(Optional) Geometry structure from GeomStruct. If not input, geometry structure is created from info in *AtmIn*.
xRangeIn [vector]	(Optional) The normalized range over which the atmospheric models in *AtmIn* are computed, separate from specifying maximum altitude. See AtmStruct for more info.

The outputs are

Atm [struct]	Atmospheric structure as from AtmStruct with reversed geometry, screen locations, and models.
G [struct]	Reversed geometry structure.

The function does not move screens, just reverses the order. So for example if *AtmIn* had screens concentrated near the beginning of the path in the forward geometry, they would be concentrated near the end of the path in the reverse geometry. In both cases, the screens are concentrated near the platform but the platform is at the end of the path in the reverse geometry.

3.4.8.19 ScreenData

Syntax

WT = ScreenData(*C, [intFlag]*)

WT = ScreenData(*[lambda], G, Atm, [intFlag]*)

This function returns a structure of computed screen data useful for input to a WaveTrain™ simulation. Velocity decomposition is done in the target "parallel" (P) and the target "transverse" (T) directions, both of which are transverse to the propagation direction. The inputs are

C [struct]	Engagement case structure from CaseStruct or a comma-separated list of ([*lambda*], *G*, *Atm*).
lambda [scalar]	(Optional) Wavelength for which to compute the screen r0 values (m). Defaults to 1 micron.
G [struct]	Geometry structure as from GeomStruct
Atm [struct]	A discrete atmospheric structure from AtmStruct with models for *Cn2*, Wind, *Abs*, *Scat*, and *Temp*. Any models not present will be set to zero.
intFlag [logical]	(Optional) Flag indicating the integral type. If *intFlag*=false (default) calculation will use *Atm.Cn2.*∗*Atm.dz*. If *intFlag*=true, calculation will integrate *Atm.Cn2Eval* over the phase screen segments.

The value of *ScrData.Cn2* is always equal to *Atm.Cn2*. If *intFlag* is false (the default), *ScrData.AverageCn2* is also equal to *Atm.Cn2*, i.e. the code assumes that the value is *Atm.Cn2* is meant to represent the average value of C_n^2 for the phase screen. If *intFlag* is true, *ScrData.AverageCn2* is computed using AverageAtm. *ScrData.IntegratedCn2* is always *ScrData.AverageCn2.*∗*Atm.dz* and *ScrData.r0Screens* is computed using *ScrData.IntegratedCn2*.

The output structure is

ScrData [struct]	Structure of computed screen data.
ScrData.platformAlt [scalar]	Platform altitude (m).
ScrData.targetAlt [scalar]	Target altitude (m).
ScrData.groundRange [scalar]	Ground range along Earth's surface (m).
ScrData.slantRange [scalar]	Slant range from platform to target (m).
ScrData.platformVp [scalar]	Component of platform velocity in the P direction (m/s).
ScrData.platformVt [scalar]	Component of platform velocity in the T direction (m/s).
ScrData.targetVp [scalar]	Component of target velocity in the P direction (m/s).
ScrData.targetVt [scalar]	Component of target velocity in the T direction (always 0).
ScrData.psPositions [vector]	Phase screen locations (m).
ScrData.psThicknesses [vector]	Phase screen slab thicknesses (m).
ScrData.Cn2 [vector]	C_n^2 values for the phase screens ($m^{-2/3}$).
ScrData.AverageCn2 [vector]	Average C_n^2 values for each phase screen ($m^{-2/3}$)
ScrData.IntegratedCn2 [vector]	Integrated C_n^2 values for each phase screen slab ($m^{1/3}$).
ScrData.Abs [vector]	Absorption coeff. for the phase screens (1/m).
ScrData.Scat [vector]	Scattering coeff. for the phase screens (1/m).
ScrData.Temp [vector]	Temperature for the phase screens (K).
ScrData.Lin [vector]	(Optional) Turbulence inner scale values for the phase screens (m).
crData.Lout [vector]	(Optional) Turbulence outer scale for the phase screens (m).
ScrData.r0Screens [vector]	r0 values for the phase screens (m).
ScrData.wavelength4r0s [scalar]	Wavelength at which *r0Screens* was computed (m).
ScrData.WindVelocityP [vector]	Component of natural wind velocity in the P direction (m/s).
ScrData.WindVelocityT [vector]	Component of natural wind velocity in the T direction (m/s).
ScrData.EffVelocityP [vector]	Component of the total (natural and slew) wind in the P direction (m/s).
ScrData.EffVelocityT [vector]	Component of the total (natural and slew) wind in the T direction (m/s).

3.4.8.20 STREHL2WFE

Syntax

wfe = STREHL2WFE(*S*)

```
SCALING_CODE_API int STREHL2WFE(char** errorChain,
    const double* strehlRatio, const int nStrehlRatio,
    double* wavefrontError)
```

This function converts Strehl ratio to rms waves of wavefront error according to the inverse of the approximation

$$S = \exp\left(-(2\pi wfe)^2\right)$$

where *S* is the input Strehl ratio and *wfe* is the output wavefront error.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
strehlRatio [in]	Strehl ratio; must be length *nStrehlRatio*.
nStrehlRatio [in]	Length of input and output arrays.
wavefrontError [out]	RMS wavefront error (waves).

3.4.8.21 WFE2STREHL

Syntax

S = WFE2STREHL(*wfe*)

```
SCALING_CODE_API int WFE2STREHL(char** errorChain, const double* wfe,
    const int nWfe, double* strehl)
```

This function converts rms waves of wavefront error to Strehl ratio using a standard approximation

$$S = \exp\left(-(2\pi wfe)^2\right)$$

where *wfe* is the input wavefront error and the output is *S*, the Strehl ratio.

The API function returns system error codes and the arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
wfe [in]	Vector of root mean square wavefront errors (waves).
nWfe [in]	Number of elements in *wfe* and *strehl*.

strehl [out]	Strehl ratio.

3.4.8.22 WIND2VXY

Syntax

[xWind yWind] = WIND2VXY(*upSpec, Atm, [G]*)

This function rotates components of natural wind transverse to the line of sight from the base coordinate frame to user-specified coordinates. The inputs are

upSpec [vector]	Desired projection of the "up" direction.
Atm [struct]	Atmospheric structure, as from ATMSTRUCT. Must be a discrete atmosphere.
G [struct]	(Optional) Geometry structure, as from GEOMSTRUCT.

Rotation is based on the angle between the projection of the zenith onto the base coordinate frame in the target plane and the input projection, *upSpec*. If the geometry structure is input, the base coordinate frame is the target T/P coordinates in the plane perpendicular to the propagation path. Otherwise, the base coordinate frame is such that the y direction is in the "up" direction and all horizontal components of wind are in the x-direction. Use of the user-specified "up" direction is under the assumption that it is defined in a right-handed coordinate frame where z is the direction of propagation from the platform to the target.

The outputs are

xWind [vector]	Wind in the x-direction (m/s).
yWind [vector]	Wind in the y-direction (m/s).

Example 3.4.59 (Rotate Natural Wind) *This illustrates various rotations of natural wind.*

```
>> Atm = AtmStruct(GeomStruct,10,'Wind','UniformAtm',5,'WindHeading','UniformAtm',90, ...
    'VerticalWind','UniformAtm',1);
>> [xWind yWind] = Wind2Vxy([0 1],Atm); [xWind yWind]

ans =

   -5.0000    0.9716
   -5.0000    0.9718
   -5.0000    0.9720
   -5.0000    0.9722
   -5.0000    0.9723
```

```
   -5.0000    0.9725
   -5.0000    0.9727
   -5.0000    0.9729
   -5.0000    0.9731
   -5.0000    0.9733
```

Setup an **Atm** *structure with default geometry (***hp** *= 12000 m South of the target,* **ht** *= 0,* **rd** *= 50000 m) and uniform wind speed and direction. Rotate wind such that all horizontal components are in the x-direction and vertical components are in the y-direction. Note that because we have a curved earth and because the propagation direction is a straight line, the angle between the zenith and the target plane varies along the path (in this case decreases from platform to target). Therefore, the component of vertical wind in the target plane (yWind) varies along the path as well.*

```
>> [xWind yWind] = Wind2Vxy([1 0],Atm); [xWind yWind]

ans =

    0.9716    5.0000
    0.9718    5.0000
    0.9720    5.0000
    0.9722    5.0000
    0.9723    5.0000
    0.9725    5.0000
    0.9727    5.0000
    0.9729    5.0000
    0.9731    5.0000
    0.9733    5.0000
```

Rotate wind component such that vertical wind is in the x-direction and horizontal wind is in the y-direction.

4.Application Programming Interface Functions

The functions in this section are functions that are available in the API and do not have MATLAB® equivalent functions in the SHaRE toolbox. Functions described in Section 4.1 contains functions that are available for calling from MATLAB® using the MATLAB® function CALLLIB. These functions are called from within MATLAB® toolbox functions. Section 4.2 describes functions that are available in the API and would be used internally in the API but aren't loaded for use in MATLAB®.

4.1 Functions Available in Matlab®

These are functions in the API that are available for calling directly from MATLAB®. Section 4.1.1 describes functions that help with geometry computations. Functions that facilitate atmospheric calculations in the API are described in Section 4.1.2. Functions in Section 4.1.4 are provided for working with xml files that describe the main structures used in the toolbox. There are parameter files for system parameter structures, geometry structure, atmospheric structures, and target structures. General utility functions in the API for setting paths and error checking are described in Section 4.1.5 and Section 4.1.6 contains general utility math functions.

4.1.1 Geometry

The following functions are API-only functions in ATMTools that support geometry calculations.

4.1.1.1 COMPUTEBILLPOINT

Syntax

```
SCALING_CODE_API int ComputeBILLPoint(char** errorChain, const int n,
    const double* trackPoint, const double* helPoint,
    const double* pathLength, const double* targetSpeed,
    const double* aoLatency, const double* beaconMargin,
    const double* stageLength, double* billPoint)
```

This function computes the optimal beacon illuminator point on the target, given constraints. Keeps the BILL point on the target section of interest and within a marginal distance from the track point. Here, the "optimal" point is such that the sensed beacon is traveling through the same atmosphere as the HEL. All of the arguments that are points are 2 × n, are in target parallel/transverse coordinates, and are stored as [p1,t1,...,pn,tn]. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Number of inputs.
trackPoint [in]	Track point in target parallel/transverse coordinates (m); must be length 2 × n.
helPoint [in]	Point in target parallel/transverse coordinates where high energy laser dwell takes place (m); must be length 2 × n.
pathLength [in]	Straight-line distance from platform to target (m); must be length n.

targetSpeed [in]	Speed of the target in target parallel direction (m/s); must be length n.
aoLatency [in]	Latency of transmit adaptive optics loop (s); must be length n.
beaconMargin [in]	The minimum distance the beacon must be separated from the track point in target parallel/transverse coordinates (m); must be length 2 × n.
stageLength [in]	Length of the section of the target that the BILL must stay on; must be length n.
billPoint [out]	Optimal beacon illuminator point in target parallel/transverse coordinates (m); must be length 2 × n.

The function returns system error codes.

4.1.1.2 CreateScreenLocations

Syntax

```
SCALING_CODE_API int CreateScreenLocations(char** errorChain,
     const int nScreens, const int nGeoms, const double* hp,
     const double* ht, const double* rd, const double* re, double* x,
     double* dx, double* z, double* dz, double* h)
```

This API function computes altitudes, normalized positions and spacing, and actual positions and spacing of phase screens along an engagement path. Only given the number of screens desired and engagement geometry, this assumes the screens should be evenly spaced. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nScreens [in]	Number of phase screen positions to generate per geometry.
nGeoms [in]	Number of geometry specifications input.
hp [in]	Altitude of transmit/receive platform (m); must be length *nGeoms*.
ht [in]	Altitude of target (m); must be length *nGeoms*.
rd [in]	Downrange of target along curved earth surface (m); must be length *nGeoms*.
re [in]	(NULL OK) Spherical earth radius (m); if passed in must be length *nGeoms*; the default value is the mean earth radius.

x [out]	Path-normalized screen location in range $0 \leq x < 1$; allocate length *nScreens* * *nGeoms* where screens belonging to a particular geometry are consecutive in memory.
dx [out]	Path-normalized screen thickness; allocate length *nScreens* * *nGeoms* where screens belonging to a particular geometry are consecutive in memory.
z [out]	Phase screen distances from transmitter (m); allocate length *nScreens* * *nGeoms* where screens belonging to a particular geometry are consecutive in memory.
dz [out]	Phase screen thicknesses (m); allocate length *nScreens* * *nGeoms* where screens belonging to a particular geometry are consecutive in memory.
h [out]	Altitude of each phase screen above earth surface (m); allocate length *nScreens* * *nGeoms* where screens belonging to a particular geometry are consecutive in memory.

4.1.1.3 CreateScreenLocationsEcf

Syntax

```
SCALING_CODE_API int CreateScreenLocationsECF(char** errorChain,
     const int nScreens, const int nGeoms, const double* posPlat,
     const double* posTarg, const double* earthRadius,
     const int nEarthRadii, double* x, double* dx, double* z, double* dz,
     double* h)
```

This function finds altitudes, normalized positions and spacing, and actual positions and spacing of phase screens along engagement path. Only given the number of screens desired and engagement geometry, this assumes the screens should be evenly spaced. The function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nScreens [in]	Number of phase screen positions to generate per geometry.
nGeoms [in]	Number of geometry specifications input.
posPlat [in]	Position of platform in ECF coords (m); must be length $3 \times$ *nGeoms* with leading dimension length 3.
posTarg [in]	Position of target in ECF coords (m); must be length $3 \times$ *nGeoms* with leading dimension length 3.

earthRadius [in]	(NULL OK) Spherical or oblate earth radius (m); if passed in must be length *nEarthRadii* × *nGeoms* with *nEarthRadii* being the length of the leading dimension; the default value is the mean earth radius.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate earth; ignored if *earthRadius* is NULL.
x [out]	Path-normalized screen location in range $0 \leq x < 1$; allocate length *nScreens* × *nGeoms* where screens belonging to a particular geometry are consecutive in memory.
dx [out]	Path-normalized screen thickness; allocate length *nScreens* × *nGeoms* where screens belonging to a particular geometry are consecutive in memory.
z [out]	Phase screen distances from transmitter (m); allocate length *nScreens* × *nGeoms* where screens belonging to a particular geometry are consecutive in memory.
dz [out]	Phase screen thicknesses (m); allocate length *nScreens* × *nGeoms* where screens belonging to a particular geometry are consecutive in memory.
h [out]	Altitude of each phase screen above earth surface (m); allocate length *nScreens* × *nGeoms* where screens belonging to a particular geometry are consecutive in memory.

This function is similar to CreateScreenLocations except that platform and target position are passed as ECF coordinate vectors and oblate earth model is supported.

4.1.1.4 CreateScreenNormPositions

Syntax

```
SCALING_CODE_API int CreateScreenNormPositions(char** errorChain,
    const int nScreens, const int nGeoms, double* x, double* dx)
```

This function computes normalized positions and spacing of phase screens along engagement path given the number of screens desired. This assumes the screens are be evenly spaced and placed in the middle of each screen slab. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nScreens [in]	Number of phase screen positions to generate per geometry.
nGeoms [in]	Number of geometry specifications input.

x [out]	Path-normalized screen location in range $0 \leq x < 1$; allocate length *nScreens* × *nGeoms* where screens belonging to a particular geometry are consecutive in memory.
dx [out]	Path-normalized screen thickness; allocate length *nScreens* × *nGeoms* where screens belonging to a particular geometry are consecutive in memory.

The function returns system error codes. To compute screen positions and thickness in meters and screen altitudes, see CreateScreenLocations and CreateScreenLocationsECF.

4.1.1.5 EarthAltEcf

Syntax

```
SCALING_CODE_API int EarthAltECF(char** errorChain, const double* x,
    const int nX, const double* posPlat, const double* posTarg,
    const double* earthRadius, int nEarthRadii, const int nGeoms,
    double* h, double* l, double* el)
```

This function computes the altitude above the surface of the earth for a ray going from platform to target for normalized position *x* (x = 0 at platform, x = 1 at target). Also returns slant range (*L*) and elevation angle (*el*) in radians. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Normalized path position from platform (0) to target (1); must be length *nX* × *nGeoms* with the leading dimension being length *nX*.
nX [in]	Number of elements along path to compute altitude at.
posPlat [in]	Position of platform in ECF coords (m); must be length 3 × *nGeoms* with leading dimension length 3.
posTarg [in]	Position of target in ECF coords (m); must be length 3 × *nGeoms* with leading dimension length 3.
earthRadius [in]	(NULL OK) Spherical or oblate earth radius (m); if passed in must be length *nEarthRadii* × *nGeoms* with *nEarthRadii* being the length of the leading dimension; the default value is the mean earth radius.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate earth; ignored if *earthRadius* is NULL.
nGeoms [in]	Number of geometry specifications to compute heights over.

h [out]	Altitude above earth surface for position *x* (m); must be length *nX* × *nGeoms* with the leading dimension being length *nX*.
l [out]	Slant range of each engagement (m); must be length *nGeoms*.
el [out]	(NULL OK) Elevation angle of each engagement (rad); must be length *nGeoms*; if NULL this value is not computed.

The function returns system error codes. This function is similar to EarthAlt except that platform and target position are passed as ECF coordinate vectors and oblate earth model is supported.

4.1.1.6 FlyoutGeomGC

Syntax

```
SCALING_CODE_API int FlyoutGeomGC(char** errorChain,
    const double* requestedTime, const int nRequestedTimes,
    const double* geomValidTime, const int movePlatform,
    const double* posPlat, const double* posTarg, const double* velPlat,
    const double* velTarg, const double* earthRadius, int nEarthRadii,
    double* newPosPlat, double* newPosTarg, double* newVelPlat,
    double* newVelTarg)
```

This function computes an array of geometries for position and velocity of a target based on great circle navigation. This function is called by the MATLAB® function FlyoutGeom when the great circle method is specified. The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
requestedTime [in]	Points in time (with respect to *geomValidTime*) that geometry is requested (s); must be length *nRequestedTimes*.
nRequestedTimes [in]	Length of *requestedTime* vector.
geomValidTime [in]	(NULL OK) Points in time that the geometry points (specified next) are associated with (s), defaults to 0.0.
movePlatform [in]	If true platform moves according to velocity and requested time; if false only the target is moved.
posPlat [in]	Platform position in ECF coordinates (m); 3 element vector.
posTarg [in]	Target position in ECF coordinates (m); 3 element vector.
velPlat [in]	Velocity of platform in ECF coordinates (m/s); 3 element vector.

velTarg [in]	Velocity of target in ECF coordinates (m/s); 3 element vector.
earthRadius [in]	(NULL OK) Numeric and scalar if spherical Earth radius, or 2-element vector [a b] where a is the equatorial radius and b is the polar radius; default is the mean spherical earth radius from PhysicalConst.
nEarthRadii [in]	Must be 1 for spherical earth or 2 for oblate.
newPosPlat [out]	Platform positions evaluated at *requestedTime*; size is 3 * *nRequestedTime*.
newPosTarg [out]	Target positions evaluated at *requestedTime*.
newVelPlat [out]	Platform velocity positions evaluated at *requestedTime*.
newVelTarg [out]	Target velocity positions evaluated at *requestedTime*.

4.1.2 Atmosphere

The following functions are API-only functions in ATMTools that support atmospheric model evaluation.

4.1.2.1 AtmEval

Syntax

```
SCALING_CODE_API int AtmEval(char** errorChain, const char* modelType,
    const double modelAlpha, const void* atmsPtr, const int nscreens,
    double* h, double* lats, double* lons, double* lambda,
    double* modelData)
```

This API function generates model data at specified altitudes based on the XML file from the input pointer to an AtmStruct type. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.

modelType [in]	The type of atmospheric model; can be any one of the following (case insensitive): Cn2 - "Cn2", "Turb", or "C"; Abs - "Abs", "A", "Absorption", "TotalAbs", or "TotalAbsorption"; AeroAbs - "AeroAbs", "AA", "AAbs", or "AerosolAbsorption"; MolecAbs - "MolecAbs", "MA", "MAbs", or "MolecularAbsorption"; CloudAbs - "CloudAbs", "CA", "CAbs", "CldAbs", or "CloudAbsorption"; RainAbs - "RainAbs", "RA", "RAbs", or "RainAbsorption"; Scat - "Scat", "S", "Scattering", "TotalSca", "TotalScat", or "TotalScattering"; MolecScat - "MolecScat", "MS", "MScat", "MolecSca", or "MolecularScattering"; AeroScat - "AeroScat", "AS", "AScat", "AeroSca", or "AerosolScattering"; CloudScat - "CloudScat", "CS", "CScat", "CldSca", "CldScat", "CloudSca", or "CloudScattering"; RainScat - "RainScat", "RS", "RScat", "RainSca", or "RainScattering"; Ext - "Ext", "E", "Extinction", "TotalExt", or "TotalExtinction"; Temp - "Temp", "T", "Temperature", or "CurrentTemperature; Wind - "Wind", "W", "NaturalWind", or "WindSpeed"; WindHeading - "WindHeading", "WH", or "WindDir"; Press - "Press", "P", "Pressure", or "CurrentPressure"; RH - "RH", "R", "RelativeHumidity", or "CurrentRH"; DewPoint - "DewPoint", "DP", or "CurrentDewPoint" Desnity - "Dens", or "Density" InnerScale - "Lin", "InnerScale" OuterScale - "Lout", "OuterScale".
modelAlpha [in]	Multiplier of model profile, on top of any multiplier specified in *atmsPtr*.
atmsPtr [in]	Void pointer to an AtmStruct, as returned from ParamStructCreate.
nscreens [in]	Length of input *h* and output *modelData*.
h [in]	Altitude above sea level (m); vector of length *nscreens*.
lats [in]	(NULL OK) Latitudes of the screen locations if required by the model (deg).
lons [in]	(NULL OK) Longitudes of the screen locations if required by the model (deg).
lambda [in]	Desired wavelength for computation of absorption, scattering, and extinction (m); ignored if model is not related to transmission; if wavelength is required but LUT does not have the requested wavelength, an error is returned.
modelData [out]	Model values at specified altitudes; must be length *nscreens*.

The function returns system error codes. The function will return all zeros with a warning if the input *atmsPtr* does not have a model for the specified *modelType*.

4.1.2.2 extCoeffs

Syntax

```
SCALING_CODE_API int extCoeffs(char** errorChain, const void* atmsPtr,
    const int nScreens, double* h, const double* lats,
    const double* lons, double* lambda, double* totalExt, double* absExt,
    double* scatExt)
```

This function computes extinction coefficients given a pointer to an Atm XML, wavelength, and path altitudes. The arguments are

errorChain [in,out]	Holds error and warning messages - deepest error is first.
atmsPtr [in]	Void pointer to AtmStruct, as returned from ParamStructCreate.
nScreens [in]	Length of input *h* and output extinction data.
h [in]	Altitude above sea level (m); vector of length *nScreens*.
lambda [in]	Desired wavelength for computation of absorption, scattering, and extinction (m); if wavelength is required but model does not have the requested wavelength, an error may be returned.
totalExt [out]	Total extinction coefficient (1/m) for each input path; allocate to length *nScreens*.
absExt [out]	(NULL OK) Extinction coefficient due to absorption only for each input path; allocate to length *nScreens*.
scatExt [out]	(NULL OK) Extinction coefficient due to scattering only for each input path; allocate to length *nScreens*.

The function returns system error codes. If *atmsPtr* does not specify an extinction model, *totalExt* will be the sum of absorption and scattering.

4.1.2.3 boundaryAtmEval

Syntax

```
DLL_EXPORT void boundaryAtmEval(char** errorChain,
    const std::string modelName, const char* funcName, ParamStruct* atms,
    const std::string modelPrefix, int nscreens, const double* h,
    int* argOffset, double* hBL)
```

This function is a utility in AtmEval to compute new altitudes for BoundaryAtm and return the argument offset for remaining model terms. If there is an error calling BoundaryAlt, the altitudes will not be set. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
modelName [in]	Model function name from *atms*. Should be "BoundaryAtm" to adjust for boundary values.
funcName [in]	Name of calling function for error/warning messaging.
atms [in]	ParamStruct with atmosphere specification.
modelPrefix [in]	Model for evaluation to query *atms*, e.g. "Cn2", "Abs", "Scat", etc.
nscreens [in]	Number of phase screen altitudes.
h [in]	Original altitudes; must be length *nscreens*.
argOffset [out]	Argument offset for the base model.
hBL [out]	Altitudes adjusted for the boundary values; must allocated to length *nscreens*.

This function has no return value.

4.1.2.4 checkGenericModels

Syntax

```
DLL_EXPORT int checkGenericModels(char** errorChain,
     const std::string modelName, const char* funcName, ParamStruct* atms,
     const std::string modelPrefix, const int atmArgOffset,
     const int nscreens, const double* hBL, const double* lats,
     const double* lons, const double* lambda, double* modelData)
```

This function is a utility in AtmEval to evaluate generic models that can be applied to multiple model types, i.e. LogScaling, UniformAtm, LEEDRAtmos, LEEDRWxCube. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
modelName [in]	Model function name from *atms*.
funcName [in]	Name of calling function for error/warning messaging.
atms [in]	ParamStruct with atmosphere specification.

modelPrefix [in]	Model for evaluation to query atms, e.g. "Cn2", "Abs", "Scat", etc.
atmArgOffset [in]	Argument offset to start pulling out inputs for evaluation of the model function.
nscreens [in]	Number of phase screen altitudes.
hBL [in]	Altitudes of the screens, adjusted for boundary layers if necessary; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model function.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model function.
lambda [in]	(NULL OK) Wavelength for evaluation of extinction coefficients. NULL ok if model doesn't require wavelength or to get wavelength from *atms*.
modelData [out]	Evaluation of the model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

This function returns system error codes or -999 if the input model function was not recognized as one of the generic model functions.

4.1.2.5 LEEDRATMOSEVAL

Syntax

```
DLL_EXPORT int LEEDRATMOSEVAL(char** errorChain, ParamStruct* atms,
    const std::string modelPrefix, const int atmArgOffset,
    const int nscreens, const double* h, const double* lambda,
    double* modelData)
```

This function is a utility in ATMEVAL to evaluate the LEEDRATMOS model obtaining inputs from an AtmStruct. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
modelPrefix [in]	Model for evaluation to query *atms*, e.g. "Cn2", "Abs", "Scat", etc.
atmArgOffset [in]	Argument offset to start pulling out inputs for LEEDRATMOS.
nscreens [in]	Number of phase screen altitudes.

h [in]	Altitudes of the screens; must be length *nscreens*.
lambda [in]	(NULL OK) Wavelength for evaluation of extinction coefficients. NULL ok if model doesn't require wavelength or to get wavelength from *atms*.
modelData [out]	Evaluation of LEEDRAtmos at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes.

4.1.2.6 LEEDRWXCUBEEVAL

Syntax

```
DLL_EXPORT int LEEDRWXCUBEEVAL(char** errorChain, ParamStruct* atms,
    std::string modelPrefix, const int atmArgOffset, const int nscreens,
    const double* h, const double* lats, const double* lons,
    const double* lambda, double* modelData)
```

This function is a utility in ATMEVAL to evaluate the LEEDRWXCUBE model obtaining inputs from an AtmStruct. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
modelPrefix [in]	Model for evaluation to query atms, e.g. "Cn2", "Abs", "Scat", etc.
atmArgOffset [in]	Argument offset to start pulling out inputs for LEEDRWXCUBE.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	Latitude of the screens; must be length *nscreens*.
lons [in]	Longitude of the screens; must be length *nscreens*.
lambda [in]	(NULL OK) Wavelength for evaluation of extinction coefficients. NULL ok if model doesn't require wavelength or to get wavelength from *atms*.
modelData [out]	Evaluation of LEEDRWxCube at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes.

4.1.2.7 LOGSCALINGEVAL

Syntax

```
DLL_EXPORT int LOGSCALINGEVAL(char** errorChain, ParamStruct* atms,
     const std::string modelPrefix, const int atmArgOffset,
     const int nscreens, const double* h, double* modelData)
```

This function is a utility in ATMEVAL to evaluate the LOGSCALING model obtaining inputs from an AtmStruct. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
modelPrefix [in]	Model for evaluation to query *atms*, e.g. "Cn2", "Abs", "Scat", etc.
atmArgOffset [in]	Argument offset to start pulling out inputs for LOGSCALING.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
modelData [out]	Evaluation of LOGSCALING at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

4.1.2.8 UNIFORMATMEVAL

Syntax

```
DLL_EXPORT int UNIFORMATMEVAL(char** errorChain, ParamStruct* atms,
     const std::string modelPrefix, const int atmArgOffset,
     const int nscreens, const double* h, double* modelData)
```

This function is a utility in ATMEVAL to evaluate the UNIFORMATM model obtaining inputs from an AtmStruct. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
modelPrefix [in]	Model for evaluation to query *atms*, e.g. "Cn2", "Abs", "Scat", etc.
atmArgOffset [in]	Argument offset to start pulling out inputs for UNIFORMATM.

nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
modelData [out]	Evaluation of UniformAtm at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes.

4.1.2.9 CN2EVAL

Syntax

```
DLL_EXPORT int CN2EVAL(char** errorChain, ParamStruct* atms,
     const int nscreens, const double* h, const double* lats,
     const double* lons, double* modelData)
```

This function is a utility in ATMEVAL to evaluate the Cn2 model as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for Cn2.

4.1.2.10 COEFFEVAL

Syntax

```
DLL_EXPORT int COEFFEVAL(char** errorChain,
     const std::string modelType, ParamStruct* atms, const int nscreens,
     const double* h, const double* lats, const double* lons,
     const double* lambda, double* modelData)
```

This function is a utility in ATMEVAL to evaluate the model for extinction coefficients as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
modelType [in]	Model for evaluation to query *atms*, e.g. "A", "S", "MA", "MS", etc.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lambda [in]	(NULL OK) Wavelength for evaluation of extinction coefficients. NULL ok to get wavelength from *atms* or use the default.
modelData [out]	Evaluation of extinction model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for the specified *modelType*.

4.1.2.11 DENSEVAL

Syntax

```
DLL_EXPORT int DENSEVAL(char** errorChain, ParamStruct* atms,
     const int nscreens, const double* h, const double* lats,
     const double* lons, double* modelData)
```

This function is a utility in ATMEVAL to evaluate the model for density as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for density.

4.1.2.12 DEWPTEVAL

Syntax

```
DLL_EXPORT int DEWPTEVAL(char** errorChain, ParamStruct* atms,
     const int nscreens, const double* h, const double* lats,
     const double* lons, double* modelData)
```

This function is a utility in ATMEVAL to evaluate the dew point model as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.

modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for dew point.

4.1.2.13 innerScaleEval

Syntax

```
DLL_EXPORT int innerScaleEval(char** errorChain, ParamStruct* atms,
    int nscreens, const double* h, const double* lats,
    const double* lons, double* modelData)
```

This function is a utility in AtmEval to evaluate the inner scale model as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for inner scale.

4.1.2.14 outerScaleEval

Syntax

```
DLL_EXPORT int outerScaleEval(char** errorChain, ParamStruct* atms,
    int nscreens, const double* h, const double* lats,
    const double* lons, double* modelData)
```

This function is a utility in AtmEval to evaluate the outer scale model as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for outer scale.

4.1.2.15 pressEval

Syntax

```
DLL_EXPORT int pressEval(char** errorChain, ParamStruct* atms,
    const int nscreens, const double* h, const double* lats,
    const double* lons, double* modelData)
```

This function is a utility in AtmEval to evaluate the model for pressure as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.

lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for pressure.

4.1.2.16 RHEVAL

Syntax

```
DLL_EXPORT int RHEVAL(char** errorChain, ParamStruct* atms,
    const int nscreens, const double* h, const double* lats,
    const double* lons, double* modelData)
```

This function is a utility in ATMEVAL to evaluate the relative humidity model as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for relative humidity.

4.1.2.17 tempEval

Syntax

```
DLL_EXPORT int tempEval(char** errorChain, ParamStruct* atms,
     const int nscreens, const double* h, const double* lats,
     const double* lons, double* modelData)
```

This function is a utility in AtmEval to evaluate the model for temperature as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for temperature.

4.1.2.18 windEval

Syntax

```
DLL_EXPORT int windEval(char** errorChain, ParamStruct* atms,
     const int nscreens, const double* h, const double* lats,
     const double* lons, double* modelData)
```

This function is a utility in AtmEval to evaluate the model for wind speed as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BoundaryAtm is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.

atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for wind.

4.1.2.19 WINDHEADINGEVAL

Syntax

```
DLL_EXPORT int WINDHEADINGEVAL(char** errorChain, ParamStruct* atms,
    int nscreens, const double* h, const double* lats,
    const double* lons, double* modelData)
```

This function is a utility in ATMEVAL to evaluate model for wind heading as a function of altitude obtaining inputs from a pointer to an AtmStruct. Altitudes will be adjusted for boundary layers for input to the base model if BOUNDARYATM is specified as the model function name. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
atms [in]	ParamStruct with atmosphere specification.
nscreens [in]	Number of phase screen altitudes.
h [in]	Altitudes of the screens; must be length *nscreens*.
lats [in]	(NULL OK) Latitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
lons [in]	(NULL OK) Longitude of the screens; must be length *nscreens*; may be NULL if not required by the model.
modelData [out]	Evaluation of model at the input altitudes for the parameters obtained from *atms*; must allocate to length *nscreens*.

The function returns system error codes. The output *modelData* will be zeros with a warning if the input *atms* does not have a model for wind heading.

4.1.2.20 wlString

Syntax

```
DLL_EXPORT std::string wlString(double lambda)
```

This API function converts the input numeric value to a wavelength string for input to atmospheric models that require wavelength to be a string. The wavelength string is the wavelength in nm with a 'p' in the place of any decimal. The arguments are

lambda [in]	Wavelength value (m).

The function returns the wavelength string as a std string value.

4.1.2.21 OpenLoopStrehlReverseLookup

Syntax

```
SCALING_CODE_API int OpenLoopStrehlReverseLookup(char** errorChain,
    const double* strehl, const double* strehlTiltRemoved,
    const double* apDiam, const double* obsDiam, int nInputs,
    double* r0Eff)
```

This function implements the open-loop Strehl LUT data in reverse to compute an effective r_0 given the Strehl ratio of a laser propagated through a turbulent medium. Finds the r_0 for either the tilt-included Strehl or the tilt-removed Strehl, but not both. The function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
strehl [in]	Strehl ratio for tilt-included beam; must be length *nInputs* or NULL.
strehlTiltRemoved [in]	Strehl ratio for tilt-removed beam; must be length *nInputs* or NULL.
apDiam [in]	Diameter of transmit aperture (m); must be length *nInputs*.
obsDiam [in]	Diameter of transmitter central obscuration (m); must be length *nInputs*.
nInputs [in]	Number of Strehl ratios to compute.
r0Eff [out]	(NULL OK) Effective r0 for given Strehl ratio.

4.1.3 LEEDR Atmosphere

The following functions are API-only functions in ATMTools that support atmospheric model evaluation using LEEDR and LEEDR Wx cubes.

4.1.3.1 LEEDRAtmos_GetLUTFN

Syntax

```
SCALING_CODE_API int LEEDRAtmos_GetLUTFN(char** errorChain,
    char* lutFileName)
```

This function returns the name of the last LEEDR LUT file loaded. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutFileName [out]	Name of the last-loaded LUT file.

The function returns system error codes. Use LEEDRAtmos_GetLUTFNLength to get a value for the length to allocate *lutFileName*.

4.1.3.2 LEEDRAtmos_GetLUTFNLength

Syntax

```
SCALING_CODE_API int LEEDRAtmos_GetLUTFNLength(char** errorChain,
    int* lutFNLength)
```

This function returns the length of the char array associated with the name of the last LEEDR LUT file loaded. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutFNLength [out]	Length of char array for the last-loaded LUT file.

The function returns system error codes. If LEEDR LUT data has not been loaded, the function will return a warning and *lutFNLength* will be 0. Use the output to allocate the input array for LEEDRAtmos_GetLUTFN.

4.1.3.3 LEEDR_clearModelData

Syntax

```
SCALING_CODE_API int LEEDR_clearModelData()
```

Use this function to clear previously loaded LEEDR LUT data. The function has no arguments and returns 0 if successful.

4.1.3.4 LEEDR_setModelData

Syntax

```
SCALING_CODE_API int LEEDR_setModelData(char** errorChain,
    const char* dataName, const int nValues, double* dataVals)
```

Use this function to set the LEEDR data from computed values as opposed to loading from a file. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
dataName [in]	Identifier for the model data to set. One of "hH" (number of LUT altitudes), "siteAltitude" (site ground elevation MSL (m)), "siteLatitude" (site latitude (deg)), "siteLongitude" (site longitude (deg)), "h" (LUT altitudes AGL (m)), "nWavelength" (number of LUT wavelengths), "Wavelength" (LUT wavelengths (m)), or any LEEDR models - see LEEDRAtmos for a list of models.
nValues [in]	Number of values in the data.
dataVals [in]	Data values to set, must be length *nValues*.

This function returns system error codes.

4.1.3.5 LEEDRSiteAltitude

Syntax

```
SCALING_CODE_API int LEEDRSiteAltitude(char** errorChain,
    const char* lutFileName, // ouput double* siteAlt)
```

This function returns site altitude (m) for a given LEEDR LUT. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutFileName [in]	Name of the file containing LEEDR lookup table data; file must be in the directory specified by SetLEEDRLookupPath or must contain the full path to the file; pass in a single, null-terminating string.
siteAlt [out]	Site altitude (deg).

The function returns system error codes. The specified LUT file will be loaded if it hasn't been already.

4.1.3.6 LEEDRSiteLatLon

Syntax

```
SCALING_CODE_API int LEEDRSiteLatLon(char** errorChain,
    const char* lutFileName, // ouput double* siteLat, double* siteLon)
```

This function returns site latitude and longitude (deg) for a given LEEDR LUT. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutFileName [in]	Name of the file containing LEEDR lookup table data; file must be in the directory specified by SetLEEDRLookupPath or must contain the full path to the file; pass in a single, null-terminating string.
siteLat [out]	Site latitude (deg).
siteLon [out]	Site longitude (deg).

The function returns system error codes. The specified LUT file will be loaded if it hasn't been already.

4.1.3.7 LEEDRWxCube_clearModelData

Syntax

```
SCALING_CODE_API int LEEDRWxCube_clearModelData()
```

This function is a utility for LEEDRWxCube that clears any loaded LEEDR Wx cube data. This function has no arguments and returns 0 if successful.

4.1.3.8 LEEDRWxCube_GetAlts

Syntax

```
SCALING_CODE_API int LEEDRWxCube_GetAlts(char** errorChain,
    float* lutAlts)
```

This function returns the altitudes associated with the last-loaded LEEDR Wx Cube LUT. Use LEEDRWxCube_LoadLookupTabl to get the number of altitudes for allocating the output. The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutAlts [out]	Altitudes; allocate to length *lutDims[2]* output from LEEDRWxCube_LoadLookupTable.

Use the function LEEDRWxCube_GetData to get the model data.

4.1.3.9 LEEDRWxCube_GetData

Syntax

```
SCALING_CODE_API int LEEDRWxCube_GetData(char** errorChain,
    const char* lutModel, float* lutData)
```

This function is a utility for LEEDRWxCube that returns the data associated with the last-loaded LUT. Use LEEDRWxCube_LoadLookupTable to get the data dimensions for allocating output. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutModel [in]	Model name. One of "lats" (Wx cube latitudes), "lons" (Wx cube longitudes), "elLats" (elevation data latitudes), "elLons" (elevation data longitudes), "el" (elevation data DTED), or any valid model name. See LEEDRWxCube for a list of valid model names.
lutData [out]	Model data - must allocate to the size of the model data returned from LEEDRWxCube_LoadLookupTable.

The function returns system error codes.

4.1.3.10 LEEDRWxCube_GetLUTInfo

Syntax

```
SCALING_CODE_API int LEEDRWxCube_GetLUTInfo(char** errorChain,
    char* lutLoc, char* lutDateTime, bool* hasElevation)
```

This function returns the character arrays associated with last LUT location and last Date/Time used for loading a LEEDR Wx Cube and whether the LUT has elevation data associated with it. Use LEEDRWxCube_GetLUTInfoLengths to get lengths for allocating the outputs. The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutLoc [out]	Last-loaded LUT location.
lutDateTime [out]	Last-loaded LUT date/time.
hasElevation [out]	True if elevation data present.

4.1.3.11 LEEDRWxCube_GetLUTInfoLengths

Syntax

```
SCALING_CODE_API int LEEDRWxCube_GetLUTInfoLengths(char** errorChain,
    int* lutLocLength, int* lutDateTimeLength)
```

This function is a utility for LEEDRWxCube that returns the lengths of the char arrays associated with last LUT location and last Date/Time. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutLocLength [out]	Length of char array for the last-loaded LUT location.
lutDateTimeLength [out]	Length of char array for the last-loaded LUT date/time.

The function returns system error codes. If no LEEDR weather cube data has been loaded, the function will return a warning and both *lutLocLength* and *lutDateTimeLength* will be 0.

4.1.3.12 LEEDRWxCube_GetNumWavelengths

Syntax

```
SCALING_CODE_API int LEEDRWxCube_GetNumWavelengths(char** errorChain,
    int* nWvls)
```

This function is a utility for LEEDRWxCube that returns the number of wavelengths associated with last-loaded LUT. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nWvls [out]	Number of wavelengths.

The function returns system error codes. If no LEEDR weather cube data has been loaded, the function will return a warning and *nWvls* will be 0.

4.1.3.13 LEEDRWxCube_GetWavelength

Syntax

```
SCALING_CODE_API int LEEDRWxCube_GetWavelength(char** errorChain,
    double* lutWavelength)
```

This function returns the wavelengths associated with the last-loaded LEEDR Wx Cube LUT. Use LEEDRWxCube_GetNumWavel to get the number of wavelengths for allocating output. The API function returns system error codes and the arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutWavelength [out]	Wavelength; allocate to length returned by LEEDRWxCube_GetNumWavelengths.

4.1.3.14 LEEDRWxCube_GetSurfaceAlts

Syntax

```
SCALING_CODE_API int LEEDRWxCube_GetSurfaceAlts(char** errorChain,
    const char* lutModel, float* lutAlts)
```

This function returns the surface altitudes associated with the last-loaded LEEDR Wx cube LUT. Use LEEDRWxCube_LoadLookupTable to get the number of surface altitudes for allocating output. The function arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutModel [in]	Model name. See LEEDRWxCube for a list of valid model names.
lutAlts [out]	Surface altitudes for the specified model; allocate to length from lutDims for the model output from LEEDRWxCube_LoadLookupTable.

The API function returns system error codes.

4.1.3.15 LEEDRWxCube_GetSurfaceData

Syntax

```
SCALING_CODE_API int LEEDRWxCube_GetSurfaceData(char** errorChain,
    const char* lutModel, float* lutData)
```

This function returns the surface data for the specified model associated with the last-loaded LUT. Use LEEDRWxCube_LoadLookupTable to get the data dimensions for allocating output. The API function arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutModel [in]	Model name. See LEEDRWxCube for a list of valid model names.

lutData [out]	Model data - must allocate to the size of the model surface data returned from LEEDRWxCube_LoadLookupTable.

The API function returns system error codes.

4.1.3.16 LEEDRWxCube_LoadLookupTable

Syntax

```
SCALING_CODE_API int LEEDRWxCube_LoadLookupTable(char** errorChain,
    const char* lutLocation, const char* lutDateTime, int* lutDims)
```

This function is a utility for LEEDRWxCube. It returns the Wx cube data dimensions and loads the LEEDR weather cube if it hasn't been loaded already. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
lutLocation [in]	LUT location.
lutDateTime [in]	LUT date/time.
lutDims [out]	(NULL OK) Dimensions of the data in the cube. Allocate to size 6. lutDims[0] = Number of latitudes, lutDims[1] = Number of longitudes, lutDims[2] = Number of altitudes, lutDims[3] = Number of Wavelengths, lutDims[4] = Number of elevation data latitudes, lutDims[5] = Number of elevation data longitudes, lutDims[6] = Number of absorption surface altitudes, lutDims[7] = Number of air density surface altitudes, lutDims[8] = Number of albedo surface altitudes, lutDims[9] = Number of cn2 surface altitudes, lutDims[10] = Number of extinction surface altitudes, lutDims[11] = Number of pressure surface altitudes, lutDims[12] = Number of relative humidity surface altitudes, lutDims[13] = Number of scattering surface altitudes, lutDims[14] = Number of temperature surface altitudes, lutDims[15] = Number of wind direction surface altitudes, lutDims[16] = Number of wind speed surface altitudes, lutDims[17] = 0 if LUT is mesh file, 1 if H5 file

The function returns system error codes.

The LEEDR weather cube data is stored in memory until it is cleared using LEEDRWxCube_clearModelData or different LUT data is loaded.

4.1.4 Parameter Structure

workspaces to the same paramstruct. Could eliminate this function if we had a method to copy.

4.1.4.1 ParamStructClearLast

Syntax

```
SCALING_CODE_API void ParamStructClearLast()
```

Use this function to clear (set to NULL) the last loaded ParamStruct to avoid having multiple pointers in different workspaces to the same ParamStruct.

4.1.4.2 ParamStructCreate

Syntax

```
SCALING_CODE_API int ParamStructCreate(char** errorChain,
    const char* filename, const char* structType, void** pPSV)
```

This function creates a ParamStruct using the given XML file. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
filename [in]	File name of XML File.
structType [in]	Type of struct, such as 'ParamStruct' or 'TargStruct'.
pPSV [out]	Pointer to the ParamStruct.

The function returns system error codes; -1 for error, 1 for warning, 0 for success. Use ParamStructDestroy to destroy the ParamStruct object.

Example 4.1.1 (Create a ParamStruct) *This function illustrates loading an XML file into an API ParamStruct.*

```
char** errorChain = new char*[1];
errorChain[0] = new char[2000]{' '};
void *pParamS = NULL;
ParamStructCreate(errorChain, "source_generic.xml", "ParamStruct", &pParamS);
```

Load the source parameter structure from "source_generic.xml" into pParamS.

4.1.4.3 ParamStructDestroy

Syntax

```
SCALING_CODE_API void ParamStructDestroy(void **pPSV)
```

Use this function to delete a ParamStruct. The input, *pPSV*, is a void pointer to a ParamStruct pointer.

4.1.4.4 ParamStructGetChar

Syntax

```
SCALING_CODE_API int ParamStructGetChar(char** errorChain,
    void** pPSV, const char* paramName, char* paramValue)
```

This function returns the specified character array from a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
paramName [in]	Name of parameter to return.
paramValue [out]	Array for storage of the output, use ParamStructGetCharLen to determine size for allocation.

The function returns system error codes; -1 for error, 0 for success. Allocate *paramValue* using the output of ParamStructGetCharLen + 1 to include NULL terminating character. The function returns an error if the specified parameter is not a character array in the ParamStruct. Use ParamStructCreate to create a ParamStruct object. Use ParamStructSetChar to set or add a character array value to the ParamStruct.

4.1.4.5 ParamStructGetCharLen

Syntax

```
SCALING_CODE_API int ParamStructGetCharLen(char** errorChain,
    void** pPSV, const char* paramName, int* paramLength)
```

This function returns the length of a specified character array from a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
paramName [in]	Name of parameter to return.
paramLength [out]	Length of the character array, allocate array to *paramLength* + 1 to include NULL terminating character.

The function returns system error codes; -1 for error, 0 for success. The function returns an error if the specified parameter is not a character array in the ParamStruct. Use ParamStructCreate to create a ParamStruct object. Use ParamStructGetChar to obtain the character array value from the ParamStruct.

4.1.4.6 ParamStructGetDbl

Syntax

```
SCALING_CODE_API int ParamStructGetDbl(char** errorChain, void** pPSV,
    const char* paramName, double* paramVal)
```

This function returns a double value from a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
paramName [in]	Name of parameter to return.
paramVal [out]	Parameter value.

The function returns system error codes; -1 for error, 0 for success. The function returns an error if the specified parameter is not a double in the ParamStruct. Use ParamStructCreate to create a ParamStruct object. Use ParamStructSetDbl to set or add a double value to the ParamStruct. The function ParamStructGetDblArray can be used to get an array of double values from the ParamStruct.

4.1.4.7 ParamStructGetDblArray

Syntax

```
SCALING_CODE_API int ParamStructGetDblArray(char** errorChain,
    void** pPSV, const char* paramName, const int idxStart,
    int* arrayLen, double* paramVal)
```

This function returns a double array from a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
paramName [in]	Name of parameter to return.
idxStart [in]	Param index for the first value for an array, paramVal[0] will be the value of "paramName.idxStart"
arrayLen [in,out]	Expected number of elements in the array, will be updated to be the number of values read.
paramVal [out]	Array for storage of the output, must be allocated to at least *arrayLen*.

The function returns system error codes; -1 for error, 1 for warning, 0 for success. The function returns an error if the specified parameter is not a double array in the ParamStruct. A warning will be returned if the double array was shorter than the expected value and the value at ***arrayLen*** will be updated to the actual size. Use ParamStructCreate to create a ParamStruct object. Use ParamStructSetDbl to set or add a double array value to the ParamStruct. The function ParamStructGetDbl can be used to get a scalar double value from a ParamStruct.

4.1.4.8 ParamStructGetInt

Syntax

```
SCALING_CODE_API int ParamStructGetInt(char** errorChain, void** pPSV,
    const char* paramName, int* paramVal)
```

Use this function to retrieve an integer value from a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
paramName [in]	Name of parameter to return.
paramVal [out]	Parameter value.

The function returns system error codes; -1 for error, 0 for success. The function returns an error if the specified parameter is not an integer in the ParamStruct. Use ParamStructCreate to create a ParamStruct object. Use ParamStructSetInt to set or add an integer value to the ParamStruct. The function ParamStructGetIntArr can be used to get an array of integer values from the ParamStruct.

4.1.4.9 ParamStructGetIntArr

Syntax

```
SCALING_CODE_API int ParamStructGetIntArr(char** errorChain,
    void** pPSV, const char* paramName, const int idxStart,
    int* arrayLen, int* paramVal)
```

This function returns an integer array from a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
paramName [in]	Name of parameter to return.
idxStart [in]	Param index for the first value for an array, paramVal[0] will be the value of "paramName.idxStart"

arrayLen [in,out]	Expected number of elements in the array, will be updated to be the number of values read.
paramVal [out]	Array for storage of the output, must be allocated to at least *arrayLen*.

The function returns system error codes; -1 for error, 1 for warning, 0 for success. The function returns an error if the specified parameter is not an integer array in the ParamStruct. A warning will be returned if the integer array was shorter than the expected length and the value at *arrayLen* will be updated to the actual size. Use ParamStructCreate to create a ParamStruct object. Use ParamStructSetInt to set or add an integer array value to the ParamStruct. The function ParamStructGetInt can be used to get a scalar integer value from a ParamStruct.

4.1.4.10 ParamStructGetXML

Syntax

```
SCALING_CODE_API int ParamStructGetXML(char** errorChain, void** pPSV,
    char* xmlContents)
```

Use this function to get the full XML contents in a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
xmlContents [out]	Contents of the XML file; use ParamStructGetXMLLen for allocation size.

The function returns system error codes; 1 for error, 0 for success. Allocate *xmlContents* using the output of ParamStructGetXMLLen + 1 to include NULL terminating character.

4.1.4.11 ParamStructGetXMLLen

Syntax

```
SCALING_CODE_API int ParamStructGetXMLLen(char** errorChain,
    void** pPSV, int* xmlLength)
```

This function returns the length of the character array from a ParamStruct XML file. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.

pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
xmlLength [out]	Length of the character array, allocate array to *xmlLength* + 1 to include NULL terminating character.

The function returns system error codes; -1 for error, 0 for success. Use ParamStructGetXML to get the XML text.

4.1.4.12 ParamStructRemoveField

Syntax

```
SCALING_CODE_API int ParamStructRemoveField(char** errorChain,
    void** pPSV, const char* fieldName)
```

This function removes a parameter (i.e. field) or group of parameters from a ParamStruct object. Returns a warning if nothing has been removed. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
fieldName [in]	Name of parameter or group of parameters to remove.

The function returns system error codes.

4.1.4.13 ParamStructSetChar

Syntax

```
SCALING_CODE_API int ParamStructSetChar(char** errorChain,
    void** pPSV, const char* paramName, const char* paramVal)
```

Use this function to set or add a character array in a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
paramName [in]	Name of parameter to set.

paramVal [in]	Character array values of *paramName* to add to the ParamStruct object.

The function returns system error codes; 0 for success. Use ParamStructCreate to create a ParamStruct object. The specified value will be added to the ParamStruct and the XML text will be updated. Use ParamStructGetChar to return the value of the specified parameter or an error if it is not already there.

4.1.4.14 ParamStructSetDbl

Syntax

```
SCALING_CODE_API int ParamStructSetDbl(char** errorChain, void** pPSV,
    const char* paramName, const int* idxStart, const int arrayLen,
    const double* paramVal)
```

Use this function to set or add a double array in a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
paramName [in]	Name of parameter to set.
idxStart [in]	Param index for the first value for an array, NULL ok.
arrayLen [in]	Number of elements in the input *paramVal*.
paramVal [in]	Values of *paramName* to add to the ParamStruct object.

The function returns system error codes; 0 for success. If *idxStart* = NULL and *arrayLen* = 1, value added will be "paramName". If *idxStart* = NULL and *arrayLen* ¿ 1, *idxStart* will be set to 0 and values added will be "paramName.i" where $0 \leq i <$*arrayLen*. If *arrayLen* = 0 or *paramVal* == NULL, *paramName* will be removed from the ParamStruct object. If *paramName* is an array, all elements after *idxStart* will be removed.

The specified value(s) will be added to (or removed from) the ParamStruct and the XML text will be updated. Use ParamStructCreate to create a ParamStruct object. Use ParamStructGetDbl to return the value of the specified parameter or an error if it is not already there.

4.1.4.15 ParamStructSetInt

Syntax

```
SCALING_CODE_API int ParamStructSetInt(char** errorChain, void** pPSV,
    const char* paramName, const int* idxStart, const int arrayLen,
    const int* paramVal)
```

Use this function to set or add an integer array in a ParamStruct object. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
pPSV [in]	Void pointer to the ParamStruct, can be any supported type.
paramName [in]	Name of parameter to set.
idxStart [in]	Param index for the first value for an array, NULL ok.
arrayLen [in]	Number of elements in the input *paramVal*.
paramVal [in]	Values of *paramName* to add to the ParamStruct object.

The function returns system error codes; 0 for success. If *idxStart* = NULL and *arrayLen* = 1, value added will be "paramName". If *idxStart* = NULL and *arrayLen* ¿ 1, *idxStart* will be set to 0 and values added will be "paramName.i" where $0 \leq i <$ *arrayLen*. The specified value(s) will be added to the ParamStruct and the XML text will be updated. Use ParamStructCreate to create a ParamStruct object. Use ParamStructGetInt to return the value of the specified parameter or an error if it is not already there.

4.1.5 General Utility Functions

4.1.5.1 GetFieldDataPath

Syntax

```
SCALING_CODE_API int GetFieldDataPath(char** errorChain,
    char* fldDataPath)
```

Use this function to retrieve the path to near-field/far=field data files that was previously set with SetFieldDataPath. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
fldDataPath [out]	The path to the field data directory; must be allocated to at least the size of the path string; use GetFieldDataPathLength.

The function returns system error codes; -1 for error, 0 for success. Use SetFieldDataPath to set the location of the field data path.

4.1.5.2 GetFieldDataPathLength

Syntax

```
SCALING_CODE_API int GetFieldDataPathLength(char** errorChain,
    int* fldDataPathLength)
```

Use this function to get the length of the field data path string, including the null-terminating character. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
fldDataPathLength [out]	Length of the field data path string.

The function returns system error codes; -1 for error, 1 for warning, or 0 for success. Use the output to allocate the argument for GetFieldDataPath.

4.1.5.3 SetFieldDataPath

Syntax

```
SCALING_CODE_API int SetFieldDataPath(char** errorChain,
    const char* fldDataPath)
```

This function sets the path to the field data directory. This function should always be called before doing work with the Scaling Code library as several functions rely on this path. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
fldDataPath [in]	The path to the field data directory.

The function returns system error codes; -1 for error, 0 for success. Use GetFieldDataPath to retrieve the current setting. The user may set this path based on the location of project specific field data files.

4.1.5.4 GetLookupPath

Syntax

```
SCALING_CODE_API int GetLookupPath(char** errorChain,
    const char* LUTtype, char* lookupPath)
```

This function retrieves the path to the specified LUT directory that was previously set with SetLookupPath. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
LUTtype [in]	A null terminating character array with the type of LUT. Supports "AERO", "AIRS", "ATM", "LEEDR", "LEEDRWxCube", "SHARE".
lookupPath [out]	The path to the specified lookup table directory; must be allocated to at least the size of the set lookup path; use GetLookupPathLength.

The function returns system error codes; -1 for error, 0 for success. Use SetLookupPath to set the specified LUT directory.

4.1.5.5 GetLookupPathLength

Syntax

```
SCALING_CODE_API int GetLookupPathLength(char** errorChain,
    const char* LUTtype, int* lookupPathLength)
```

This function returns the length of the specified LUT path string, including the null-terminating character, that was previously set with SetLookupPath. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
LUTtype [in]	A null terminating character array with the type of LUT. Supports "AERO", "AIRS", "ATM", "LEEDR", "LEEDRWxCube", "SHARE".
lookupPathLength [out]	The length of the LUT path for allocation.

The function returns system error codes; -1 for error, 1 for warning, 0 for success.

4.1.5.6 SetLookupPath

Syntax

```
SCALING_CODE_API int SetLookupPath(char** errorChain,
    const char* LUTtype, const char* lookupPath)
```

This function sets the path to the specified lookup table directory. This should always be called before doing work with the Scaling Code library; several functions rely on this path. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
LUTtype [in]	A null terminating character array with the type of LUT. Supports "AERO", "AIRS", "ATM", "LEEDR", "LEEDRWxCube", "SHARE".
lookupPath [in]	The path to the lookup table directory.

The function returns system error codes; -1 for error, 0 for success. Use GetLookupPath to retrieve the path to a particular LUT directory. The "ATM" and "SHARE" LUT paths should be set to the directories included in the ScalingCodeAPI directory as several functions rely on these. The other LUT paths can be set to custom project-specific directories.

4.1.5.7 GetParamPath

Syntax

```
SCALING_CODE_API int GetParamPath(char** errorChain,
    char* paramFilePath)
```

This function retrieves the path to the parameter struct directory that was previously set with SetParamPath. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
paramFilePath [out]	The path to the parameter struct directory, must be allocated to at least the size of the path; use GetParamPathLength.

The function returns system error codes; -1 for error, 0 for success.

4.1.5.8 GetParamPathLength

Syntax

```
SCALING_CODE_API int GetParamPathLength(char** errorChain,
    int* paramPathLength)
```

This function gets the length of the parameter struct path string, including the null-terminating character. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
paramPathLength [out]	The length of the parameter struct path string for allocation.

The function returns system error codes; -1 for error, 1 for warning, 0 for success. Use GetParamPath to get the path.

4.1.5.9 SetParamPath

Syntax

```
SCALING_CODE_API int SetParamPath(char** errorChain,
    const char* paramPath)
```

This function sets the path to the parameter struct directory. This should always be called before doing work with the Scaling Code library; several functions rely on this path. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
paramPath [in]	The path to the parameter structs directory.

The function returns system error codes; -1 for error, 0 for success. Use GetParamPath to return the path. The user may set this path based on the location of project-specific parameter struct XML files.

4.1.5.10 GetTargDataPath

Syntax

```
SCALING_CODE_API int GetTargDataPath(char** errorChain,
    char* targDataPath)
```

This function retrieves the path to the target data directory that was previously set with SetTargDataPath. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
targDataPath [out]	The path to the target data directory, must be allocated to at least the size of the path; use GetTargDataPathLength.

The function returns system error codes; -1 for error, 0 for success.

4.1.5.11 GetTargDataPathLength

Syntax

```
SCALING_CODE_API int GetTargDataPathLength(char** errorChain,
    int* targDataPathLength)
```

This function gets the length of the target data path string, including the null-terminating character. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
targDataPathLength [out]	The length of the target data path string for allocation.

The function returns system error codes; -1 for error, 1 for warning, 0 for success. Use GetTargDataPath to get the path.

4.1.5.12 SetTargDataPath

Syntax

```
SCALING_CODE_API int SetTargDataPath(char** errorChain,
    const char* targDataPath)
```

This function sets the path to target data directory. This should always be called before doing work with the Scaling Code library as several functions rely on this path. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
targDataPath [in]	The path to the target data directory.

The function returns system error codes; -1 for error, 0 for success. Use GetTargDataPath to return the path. The user may set this path based on the location of project-specific target object data files.

4.1.5.13 errorCheck

Syntax

```
SCALING_CODE_API int errorCheck(char** errorChain, const char* funcID,
    const int errNum, const char* errorMsg)
```

This function provides error handling for the ScalingCode API. Input error/warning messages are appended to errorChain. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first; Pass as NULL to not append *errorMsg*, message may still be displayed.
funcID [in]	Name of the function with an error.
errNum [in]	Takes value of -1 or 1 to represent error or warning, respectively.
errorMsg [in]	The message explaining the reason for error or warning; pass as NULL to not update *errorChain* and not display message.

This function returns *errNum*.

4.1.5.14 getWarning

Syntax

```
SCALING_CODE_API bool getWarning(const char* msgId)
```

This function returns the current warning status for a particular warning identifier. The char array input, *msgId*, is the warning message identifier. The function outputs a boolean value indicating the warning status. Use setWarning to turn on or off specified messages. Warning message display is handled by the function errorCheck.

4.1.5.15 outputOn

Syntax

```
SCALING_CODE_API bool outputOn()
```

Use this function to query flag for error/warning message display. The function has no arguments and returns a boolean value to indicate the setting. Use seeOut to set the message display flag.

4.1.5.16 seeOut

Syntax

```
SCALING_CODE_API void seeOut(const bool setting)
```

This function can be used to set the flag for error/warning message display. The boolean input, *setting*, should be set to true to turn std::out on and false to turn display off. Use outputOn to query the flag setting.

4.1.5.17 setWarning

Syntax

```
SCALING_CODE_API void setWarning(const char* msgId,
    const bool setting)
```

Use this function to set the warning status for a particular warning identifier. The arguments are

msgId [in]	Warning message identifier.
setting [in]	Flag indicating whether the warning should be on or off.

Use getWarning to set the warning display status for a particular warning identifier. The function errorCheck handles the warning message display.

4.1.6 MzaSciComp General Utility Functions

This section contains specialized mathematical functions available in the API, in addition to functions described in Section 3.4.6.

4.1.6.1 CROP

Syntax

```
SCALING_CODE_API int CROP(char** errorChain, int n0, int n1,
    const double* img, int ctr0, int ctr1, int m0, int m1,
    double* imgOut)
```

This function crops an array to *NR*×*NC* by taking the center portion of the array. Center pixel is assumed to be floor(N/2)+1, where N is the size of the input array - typical FFT definition. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n0 [in]	Number of elements in the sequential dimension in the input *img*.
n1 [in]	Number of elements in the non-sequential dimension in the input *img*.
img [in]	Input array to be cropped; must be size *n0* × *n1*.
ctr0 [in]	Center point around which image is to be cropped in sequential dimension; if zero, defaults to floor(*n0*/2).
ctr1 [in]	Center point around which image is to be cropped in non-sequential dimension; if zero, defaults to floor(n1/2).
m0 [in]	Number of elements in sequential dimension to keep.
m1 [in]	Number of elements in non-sequential dimension to keep.
imgOut [out]	Output cropped array; must allocated to *m0* × *m1*.

The function returns system error codes; 1 for warning, -1 for error, 0 for success.

4.1.6.2 PAD

Syntax

```
SCALING_CODE_API int PAD(char** errorChain, const int n0,
    const int n1, const double* img, const double* padVal, const int m0,
    const int m1, double* imgOut)
```

This function will pad the input array to *NRC*×*NRC* pixels such that the center of the image is at floor(*NRC*/2)+1 - matching the FFT convention. Input image is assumed to be centered at [floor(*nx*/2)+1 floor(*ny*/2)+1]. If *NRC* is not input, it will be set to `max(nextpow2(nx),nextpow2(ny))`. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n0 [in]	Number of elements in the sequential dimension in the input *img*.
n1 [in]	Number of elements in the non-sequential dimension in the input *img*.
img [in]	Input array to be padded; must be size *n0* × *n1*.
padVal [in]	Value to use for padding; NULL ok, defaults to zero.
m0 [in]	Number of elements in the sequential dimension of *imgOut*.
m1 [in]	Number of elements in the non-sequential dimension of *imgOut*.
imgOut [out]	Output padded array; must allocated to *m0* × *m1*.

The function returns system error codes.

4.1.6.3 UNWRAP

Syntax

```
SCALING_CODE_API int UNWRAP(char** errorChain, const int nx,
    const double* x, double* ux)
```

This function unwraps the radian phase for a vector of wrapped phase values. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	The number of elements in *x*.
x [in]	The phase array to unwrap; must be size *nx*.
ux [out]	The unwrapped phase; allocate to size *nx*.

The function returns system error codes; -1 for error, 0 for success.

4.2 Functions Only in API

These are functions in the API that are not available from MATLAB® but are available in other sources when working in the API. Many of these functions have MATLAB® equivalents that we needed to reproduce in order to use the API outside of MATLAB®. There are a few general utility functions described in Section 4.2.2 and Section 4.2.3 describes general numeric functions. Section 4.2.4 describes functions that operate on scalar values, Section 4.2.5 describes functions that operate on vectors, and Section 4.2.6 contains functions that operate on matrices.

4.2.1 Parameter Structure Methods

This section describes methods available for ParamStruct objects in the API.

4.2.1.1 addDouble

Syntax

```
DLL_EXPORT void ParamStruct::addDouble(const char* name, double value)
```

This function adds a double value to a ParamStruct. The arguments are

name [in]	Name of the double to add to the ParamStruct.
value [in]	Value for the double.

The function does not have a return value. If the specified double already exists, the value will be updated. The XML text will also be updated.

4.2.1.2 addInt

Syntax

```
DLL_EXPORT void ParamStruct::addInt(const char* name, const int value)
```

This function adds an integer value to a ParamStruct. The arguments are

name [in]	Name of the integer to add to the ParamStruct.
value [in]	Value for the integer.

The function does not have a return value. If the specified integer already exists, the value will be updated. The XML text will also be updated.

4.2.1.3 addString

Syntax

```
DLL_EXPORT void ParamStruct::addString(const char* name, const char*
value)
```

This function adds a string value to a ParamStruct. The arguments are

name [in]	Name of the string to add to the ParamStruct.
value [in]	Value for the string.

The function does not have a return value. If the specified string already exists, the value will be updated. The XML text will also be updated.

4.2.1.4 CLEAR

Syntax

```
DLL_EXPORT void ParamStruct::clear(void)
```

Use this function to clear the static data in ParamStruct. ParamStructCreate stores the last loaded ParamStruct for quicker access. It may be necessary to clear this data when creating and destroying ParamStruct data.

4.2.1.5 GETDOUBLE

Syntax

```
DLL_EXPORT double ParamStruct::getDouble(const char* name,
    const double valueIfNotFound)
```

This function returns the specified double value from the ParamStruct. The arguments are

name [in]	Name of the double to get from the ParamStruct.
valueIfNotFound [in]	Value for the double if not found. Defaults to 0.

The function returns the requested double value. If the specified value is not there, it will be added with the value in *valueIfNotFound*. The XML text will also be updated.

4.2.1.6 GETDOUBLENAMES

Syntax

```
DLL_EXPORT int ParamStruct::getDoubleNames(const char* nameStartsWith,
    std::vector<std::string> &doubleNames)
```

Use this function to get the names for all double values from the ParamStruct that start with the specified char array. The arguments are

nameStartsWith [in]	Array to look for at the beginning of names in the ParamStruct.
doubleNames [out]	A vector of names that start with the input array.

The function returns the number of names found.

4.2.1.7 getInt

Syntax

```
DLL_EXPORT int ParamStruct::getInt(const char* name, const int
valueIfNotFound)
```

This function returns the specified integer value from the ParamStruct. The arguments are

name [in]	Name of the integer to get from the ParamStruct.
valueIfNotFound [in]	Value for the integer if not found. Defaults to 0.

The function returns the requested integer value. If the specified value is not there, it will be added with the value in *valueIfNotFound*. The XML text will also be updated.

4.2.1.8 getIntNames

Syntax

```
DLL_EXPORT int ParamStruct::getIntNames(const char* nameStartsWith,
    std::vector<std::string> &integerNames)
```

Use this function to get the names for all integer values from the ParamStruct that start with the specified char array. The arguments are

nameStartsWith [in]	Array to look for at the beginning of names in the ParamStruct.
integerNames [out]	A vector of names that start with the input array.

The function returns the number of names found.

4.2.1.9 getString

Syntax

```
DLL_EXPORT std::string ParamStruct::getString(const char* name,
    const char* valueIfNotFound)
```

This function returns the specified string value from the ParamStruct. The arguments are

name [in]	Name of the string to get from the ParamStruct.
valueIfNotFound [in]	Value for the string if not found. If NULL, name will not be added.

The function returns the requested string value. If the specified value is not there, it will be added with the value in *valueIfNotFound*. The XML text will also be updated.

4.2.1.10 getStringNames

Syntax

```
DLL_EXPORT int ParamStruct::getStringNames(const char* nameStartsWith,
    std::vector<std::string> &stringNames)
```

Use this function to get the names for all string values from the ParamStruct that start with the specified char array. The arguments are

nameStartsWith [in]	Array to look for at the beginning of names in the ParamStruct.
stringNames [out]	A vector of names that start with the input array.

The function returns the number of names found.

4.2.1.11 getXMLtext

Syntax

```
DLL_EXPORT std::string ParamStruct::getXMLtext(void)
```

This function returns the XML text associated with the ParamStruct.

4.2.1.12 isDouble

Syntax

```
DLL_EXPORT bool ParamStruct::isDouble(const char* name)
```

Query whether a specified double value exists in the ParamStruct. The input char array, *name*, is the double to find. The function returns a boolean value indicating whether the specified double value is contained in the ParamStruct.

4.2.1.13 isInt

Syntax

```
DLL_EXPORT bool ParamStruct::isInt(const char* name)
```

Query whether a specified integer value exists in the ParamStruct. The input char array, *name*, is the integer to find. The function returns a boolean value indicating whether the specified integer value is contained in the ParamStruct.

4.2.1.14 isString

Syntax

```
DLL_EXPORT bool ParamStruct::isString(const char* name)
```

Query whether a specified string value exists in the ParamStruct. The input char array, *name*, is the string to find. The function returns a boolean value indicating whether the specified string value is contained in the ParamStruct.

4.2.1.15 loadXML

Syntax

```
   DLL_EXPORT int ParamStruct::loadXML(char** errorChain, const char*
filename,
        const char* structType)
```

This function loads the specified xml file into a ParamStruct. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
filename [in]	Name of the xml file to load. Must include full path to the file.
structType [in]	Structure type to load. One of "ParamStruct", "GeomStruct", "TargSTruct" or "AtmStruct".

The function returns the number of elements loaded or -1 to indicate an error.

4.2.1.16 print

Syntax

```
DLL_EXPORT void ParamStruct::print(char** errorChain, const char* name)
```

Use this function to print to screen the value of a specified parameter. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
name [in]	Name of parameter value to display.

4.2.1.17 printAll

Syntax

```
DLL_EXPORT void ParamStruct::printAll(void)
```

Use this function to print to screen all the values in a ParamStruct.

4.2.1.18 remove

Syntax

```
DLL_EXPORT int ParamStruct::remove(const char* name)
```

Use this function to remove a specified value from the ParamStruct. Will be removed from strings, doubles, and integers. The char array input, *name*, is the name of the parameter to remove from the ParamStruct. The function returns the total number of elements removed.

4.2.1.19 removeGroup

Syntax

```
DLL_EXPORT int ParamStruct::removeGroup(const char* groupName)
```

This function is used to remove a specified group from the ParamStruct. This will remove all values from strings, doubles, and integers in the group and update the XML text to remove the group. The function arguments are

groupName [in]	Name of the group to remove from the ParamStruct.

The API function returns the number of elements removed.

4.2.2 General Utility Functions

4.2.2.1 getFullFilePath

Syntax

```
DLL_EXPORT int getFullFilePath(char** errorChain,
    std::string filename, std::string fileType, const char* LUTtype,
    std::string* fullPath)
```

This function returns the full path to a parameter file. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
filename [in]	Name of the file; file must be in the directory specified by default path or must contain the full path to the file; pass in a std::string.
fileType [in]	Type identifier for where default file location - either "Field", "Lookup", "Parameter", or "Targ".
LUTtype [in]	(NULL OK) A null terminating character array with the type of Lookup. Supports "AERO", "AIRS", "ATM", "LEEDR", "SHARE". Ignored if *fileType* is not "Lookup".
fullPath [out]	Full path to the file - pointer to a std::string.

The function returns system error codes; -1 for error, 0 for success, 1 for warning.

4.2.2.2 LowerCase

Syntax

```
DLL_EXPORT std::string LowerCase(const char* inputString)

DLL_EXPORT std::string LowerCase(std::string inputString)
```

This function converts the input null-terminated string, *inputString*, to all lower case. The function returns the lower case string.

4.2.3 Sci Comp Numerics

4.2.3.1 IsInfinite

Syntax

```
DLL_EXPORT bool IsInfinite(const double x)
```

This function is used to check if a value is one of the two IEEE representations for infinity. The only input is the value to check, *x*, and the function returns true if the input is positive or negative infinity and false otherwise.

4.2.3.2 IsNan

Syntax

```
DLL_EXPORT bool IsNan(const double x)
```

This function is used to check if a value is the IEEE "not a number" representation. This is important because using comparison operators oftentimes return false when both values actually are NaN, because "NaN is not equal to itself". The only input is the value to check and the function returns true if the input is NaN and false otherwise.

4.2.3.3 Nan

Syntax

```
DLL_EXPORT double Nan()
```

This function returns IEEE 754 standard representation of NaN for a double-precision floating point number.

4.2.3.4 NegativeInfinity

Syntax

```
DLL_EXPORT double NegativeInfinity()
```

This function returns IEEE 754 standard representation of negative infinity for a double-precision floating point number.

4.2.3.5 One

Syntax

```
DLL_EXPORT int One(char** errorChain, const int nx, double* x)
```

This function fills the provided array with ones. The arguments are

errorChain	[in,out]	Holds error and warning messages- deepest error is first.

nx [in]	The number of elements in *x*.
x [in,out]	The array to fill; must be size *nx*.

The function returns system error codes.

4.2.3.6 PositiveInfinity

Syntax

```
DLL_EXPORT double PositiveInfinity()
```

This function returns the IEEE 754 standard representation of positive infinity for a double-precision floating point number.

4.2.3.7 Zero

Syntax

```
DLL_EXPORT int Zero(char** errorChain, const int nx, double* x)
```

This function zeros the provided double array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	The number of elements in *x*.
x [in,out]	The array to zero; must be size *nx*.

The function returns system error codes; -1 for error, 0 for success.

4.2.4 Scalar Functions

4.2.4.1 BesselJ

Syntax

```
DLL_EXPORT double BesselJ(int n, double x)
```

This function computes the value for the Bessel function of the first kind, order *n*. The arguments are

n [in]	Bessel function order.
x [in]	Argument of the Bessel function.

The function returns $J_n(x)$.

4.2.4.2 CartesianToPolar

Syntax

```
DLL_EXPORT int CartesianToPolar(char** errorChain, double x, double y,
    double z, double angle[1], double radius[1], double height[1])
```

This function converts 3D Cartesian coordinates to 3D polar (or cylindrical) coordinates. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	First Cartesian coordinate (units).
y [in]	Second Cartesian coordinate (units).
z [in]	Third Cartesian coordinate (units).
angle [out]	Angle around the cylinder (radians); angle in the x-y plane; must be allocated length 1.
radius [out]	Distance at height zero from the origin (units); distance in the x-y plane; must be allocated length 1.
height [out]	Distance along the straight dimension of the cylinder (units); corresponds exactly to z; must be allocated length 1.

The function returns system error codes; -1 for error, 0 for success. Use PolarToCartesian to reverse the conversion.

4.2.4.3 CartesianToSpherical

Syntax

```
DLL_EXPORT int CartesianToSpherical(char** errorChain, double x,
    double y, double z, double azimuth[1], double elevation[1],
    double radius[1])
```

This function converts Cartesian coordinates to 3D spherical coordinates. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	First Cartesian coordinate (units).
y [in]	Second Cartesian coordinate (units).
z [in]	Third Cartesian coordinate (units).
azimuth [out]	Angle (radians); must be allocated length 1.

elevation [out]	Angle (radians); must be allocated length 1.
radius [out]	Distance of spherical coordinate from origin (units); must be allocated length 1.

The function returns system error codes; -1 for error, 0 for success. Use SphericalToCartesian to reverse the conversion.

4.2.4.4 ComplexDivide

Syntax

```
DLL_EXPORT int ComplexDivide(char** errorChain, double xReal,
    double xImag, double yReal, double yImag, double* zReal,
    double* zImag)
```

This function implements the division operator between complex numbers. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
xReal [in]	Real component of x in x/y=z.
xImag [in]	Imaginary component of x in x/y=z.
yReal [in]	Real component of y in x/y=z.
yImag [in]	Imaginary component of y in x/y=z.
zReal [out]	Real component of z in x/y=z; allocate one element.
zImag [out]	Imaginary component of z in x/y=z; allocate one element.

The function returns system error codes; -1 for error, 0 for success. Use ComplexMultiply to multiply complex numbers.

4.2.4.5 ComplexMultiply

Syntax

```
DLL_EXPORT int ComplexMultiply(char** errorChain, double xReal,
    double xImag, double yReal, double yImag, double* zReal,
    double* zImag)
```

This function is used to multiply two complex numbers.

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$$

The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
xReal [in]	Real component of x in x*y=z.
xImag [in]	Imaginary component of x in x*y=z.
yReal [in]	Real component of y in x*y=z.
yImag [in]	Imaginary component of y in x*y=z.
zReal [out]	Real component of z in x*y=z; allocate one element.
zImag [out]	Imaginary component of z in x*y=z; allocate one element.

The function returns system error codes; -1 for error, 0 for success.

4.2.4.6 DegreesToRadians

Syntax

```
DLL_EXPORT double DegreesToRadians(const double deg)
```

This function converts a spread represented as the number of parts of a circle equally divided into 360 parts to a spread represented as the amount of circumference covered by the unit circle. The only input, *deg*, is the value in degrees. The function returns $deg \times (\pi/180)$.

4.2.4.7 ErrorFunction

Syntax

```
DLL_EXPORT double ErrorFunction(const double x)
```

This function evaluates the error function for the input, *x*. The function returns the evaluation of the error function at *x*. This function is being added to the C++ standard (2011), but this is not so common yet so this function uses MKL for now.

4.2.4.8 Fix

Syntax

```
DLL_EXPORT double Fix(const double x)
```

This function finds the next integer value on the number line towards zero. The only input is a number value. The function returns an integer (represented in double precision format) fixed towards zero.

4.2.4.9 Gamma

Syntax

```
DLL_EXPORT double Gamma(const double x)
```

This function evaluates the Gamma function for the input. The only input is the input to the Gamma function. The function returns the evaluation of Gamma at *x*. This function is being added to the C++ standard (2011), but this is not so common yet so this function uses MKL for now.

4.2.4.10 GAMMALN

Syntax

```
DLL_EXPORT int GAMMALN(char** errorChain, const double x,
    double y[1])
```

This function computes the natural logarithm of the gamma function. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Function input.
y [out]	Function evaluated at *x*.

The function returns system error codes; -1 for error, 0 for success. MKL version return natural log of abs value.

4.2.4.11 GETNEXTPOWER2

Syntax

```
DLL_EXPORT int GETNEXTPOWER2(const int n)
```

This function finds the next integer greater than *n* that is a power of 2. The only input is *n*, the minimum value of the output. The function returns 2^P where P is the minimum integer that satisfies $2^P > n$.

4.2.4.12 POLARTOCARTESIAN

Syntax

```
DLL_EXPORT int POLARTOCARTESIAN(char** errorChain, const double angle,
    const double radius, const double height, double x[1], double y[1],
    double z[1])
```

This function converts 3D polar (or cylindrical) coordinates to 3D Cartesian. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
angle [in]	Angle around the cylinder (radians); angle in the x-y plane.
radius [in]	Distance at height zero from the origin (units); distance in the x-y plane.
height [in]	Distance along the straight dimension of the cylinder (units); corresponds exactly to z.
x [out]	First Cartesian coordinate (units); must be allocated length 1.
y [out]	Second Cartesian coordinate (units); must be allocated length 1.

z [out]	Third Cartesian coordinate (units); must be allocated length 1.

The function returns system error codes.

4.2.4.13 RadiansToDegrees

Syntax

```
DLL_EXPORT double RadiansToDegrees(const double rad)
```

This function converts a spread represented as the amount of circumference covered by the unit circle to a spread represented as the number of parts of a circle equally divided into 360 parts. The arguments are

rad [in]	An angle radians.

The function returns *rad* $\times$ (180 / π).

4.2.4.14 Remainder

Syntax

```
DLL_EXPORT double Remainder(const double num, const double den)
```

This function finds and returns the remainder of a/b. The arguments are

num [in]	Dividend.
den [in]	Divisor.

The result makes the most sense if *num* and *den* are integers.

4.2.4.15 Round

Syntax

```
DLL_EXPORT double Round(const double x)
```

This function rounds a value to the nearest integer. The arguments are

x [in]	Value to round.

The function returns the rounded value as a double.

4.2.4.16 Sign

Syntax

```
DLL_EXPORT double Sign(const double x)
```

This function determines whether the input, x, is positive, negative, or zero. The function returns 0 if x is 0, 1 if x is positive, -1 if x is negative, and NaN if x is NaN.

4.2.4.17 SphericalToCartesian

Syntax

```
DLL_EXPORT int SphericalToCartesian(char** errorChain,
     const double azimuth, const double elevation, const double radius,
     double x[1], double y[1], double z[1])
```

This function converts 3D spherical coordinates to 3D Cartesian. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
azimuth [in]	Angle (radians).
elevation [in]	Angle (radians).
radius [in]	Distance of spherical coordinate from origin (units).
x [out]	First Cartesian coordinate (units); must be allocated length 1.
y [out]	Second Cartesian coordinate (units); must be allocated length 1.
z [out]	Third Cartesian coordinate (units); must be allocated length 1.

The function returns system error codes; -1 for error, 0 for success. Use CartesianToSpherical to reverse the conversion.

4.2.5 Vector Functions

4.2.5.1 AbsElementwise

Syntax

```
DLL_EXPORT int AbsElementwise(char** errorChain, const int n,
     const double* a, double* z)
```

This function computes the absolute or non-negative values of the values in the provided input array. The arguments are:

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Number of inputs and outputs.
a [in]	Array to take the absolute values of; must be length n.
z [out]	Elementwise absolute values of a; allocate to length n.

The API function returns system error codes.

4.2.5.2 AcosElementwise

Syntax

```
DLL_EXPORT int AcosElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the arccosine of each element in an input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values to take the arccosine of; must be length *n*.
z [out]	The result of $\cos(a)$; must be length *n*.

The API function returns system error codes.

4.2.5.3 arrayFillD

Syntax

```
DLL_EXPORT int arrayFillD(char** errorChain, const int nx,
    const double v, double* x)
```

This function fills the provided double array with a single value. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	The number of elements in *x*.

v [in]	The value to place in the array.
x [in,out]	The array to fill; must be size *nx*.

The API function returns system error codes.

4.2.5.4 arrayFilli

Syntax

```
DLL_EXPORT int arrayFilli(char** errorChain, const int nx,
    const int v, int* x)
```

This function fills the provided integer array with a single value. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx[in]	The number of elements in *x*.
v [in]	The value to place in the array.
x [in,out]	The array to fill; must be size *nx*.

The API function returns system error codes.

4.2.5.5 AsinElementwise

Syntax

```
DLL_EXPORT int AsinElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the arcsine of each element in an input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values to take the arcsine of; must be length *n*.
z [out]	The result of $\sin(a)$; must be length *n*.

The API function returns system error codes.

4.2.5.6 AtanElementwise

Syntax

```
DLL_EXPORT int AtanElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the arctangent for each element in an input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values to take the acrtangent of; must be length *n*.
z [out]	The result of tan(a); must be length *n*.

The API function returns system error codes.

4.2.5.7 Atan2Elementwise

Syntax

```
DLL_EXPORT int Atan2Elementwise(char** errorChain, const int n,
    const double* a, const double* b, double* z)
```

This function computes the four quadrant arctangent of each element of a/b. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values; must be length *n*.
b [in]	An array of values; must be length *n*.
z [out]	The result of atan(a,b); must be length *n*.

The API function returns system error codes.

4.2.5.8 ComplexElementwiseDivide

Syntax

```
DLL_EXPORT int ComplexElementwiseDivide(char** errorChain, int n,
    const double* xReal, const double* xImag, const double* yReal,
    const double* yImag, double* zReal, double* zImag)
```

This function performs element-by-element division for complex numbers. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of independent complex numbers to multiply.
xReal [in]	Real component of x in x/y=z; must be length *n*.
xImag [in]	Imaginary component of x in x/y=z; must be length *n*.
yReal [in]	Real component of y in x/y=z; must be length *n*.
yImag [in]	Imaginary component of y in x/y=z; must be length *n*.
zReal [out]	Real component of z in x/y=z; must allocate to length *n*.
zImag [out]	Imaginary component of z in x/y=z; must allocate to length *n*.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.9 ComplexElementwiseMultiply

Syntax

```
DLL_EXPORT int ComplexElementwiseMultiply(char** errorChain, int n,
    const double* xReal, const double* xImag, const double* yReal,
    const double* yImag, double* zReal, double* zImag)
```

This function does element-by-element multiplication for complex numbers. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of independent complex numbers to multiply.
xReal [in]	Real component of x in x*y=z; must be length *n*.
xImag [in]	Imaginary component of x in x*y=z; must be length *n*.
yReal [in]	Real component of y in x*y=z; must be length *n*.
yImag [in]	Imaginary component of y in x*y=z; must be length *n*.

zReal [out]	Real component of z in x*y=z; must allocate to length *n*.
zImag [out]	Imaginary component of z in x*y=z; must allocate to length *n*.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.10 CopyVector

Syntax

```
DLL_EXPORT int CopyVector(char** errorChain, const int n,
    const int incSrc, const double* src, const int incDest,
    double* dest)
```

This function copies vector *a* (or portion thereof) into vector *z*. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements to copy.
incSrc [in]	Increment for the elements of *src*.
src [in]	An array of values; must be at least 1 + (*n* - 1)**incSrc*.
incDest [in]	Increment for the elements of *dest*.
dest [out]	An array of values; must be at least 1 + (*n* - 1)**incDest*.

The API function returns system error codes.

4.2.5.11 CosElementwise

Syntax

```
DLL_EXPORT int CosElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the cosine of each element in the input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.

a [in]	An array of values to take the cosine of; must be length *n*.
z [out]	The result of cos(*a*); must be length *n*.

The function returns system error codes.

4.2.5.12 CrossProduct

Syntax

```
DLL_EXPORT int CrossProduct(char** errorChain, const double* a,
    const double* b, double* c)
```

This function calculates the cross product of two input vectors, $c = a \times b$. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
a [in]	3-element vector, first operand of cross product.
b [in]	3-element vector, second operand of cross product.
c [out]	Allocate 3-element vector for storing the result.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.13 CrossProductArray

Syntax

```
DLL_EXPORT int CrossProductArray(char** errorChain, const int n,
    const double* a, const double* b, double* c)
```

This function calculates the cross product, $c = a \times b$, of multiple three element vectors. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Number of 3-element vectors in *a* and *b*.
a [in]	3-element vector, first operand of cross product; must be length $3 \times n$.
b [in]	3-element vector, second operand of cross product; must be length $3 \times n$.
c [out]	Result of the cross product; must be length $3 \times n$.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.14 DegreesToRadiansVector

Syntax

```
DLL_EXPORT int DegreesToRadiansVector(char** errorChain, const int n,
    const double* deg, double* rad)
```

This function converts each element of the input array from degrees to radians. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Number of inputs and outputs.
deg [in]	An angle (degrees); must be length *n*.
rad [out]	An angle (radians); must be length *n*.

The function returns system error codes.

4.2.5.15 Difference

Syntax

```
DLL_EXPORT int Difference(char** errorChain, const double* x,
    const int nx, double* result)
```

This function computes the difference between elements in a vector. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Vector to find difference between elements.
nx [in]	Number of elements in vector *x*.
result [out]	Allocate to length *nx*-1; only values from *result*[0] to *result*[*nx*-2] (inclusive) are used.

The API function returns system error codes.

4.2.5.16 DifferenceElementwise

Syntax

```
DLL_EXPORT int DifferenceElementwise(char** errorChain, const int n,
    const double* a, const double* b, double* z)
```

This function subtracts each element in an input array from the corresponding element in another input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
`n` [in]	The number of elements in `a`, `b`, and `z`.
`a` [in]	An array from which to subtract, element-by-element; must be length `n`.
`b` [in]	An array to subtract, element-by-element, from `a`; must be length `n`.
`z` [out]	The result of $a - b$; must be length `n`.

The API function returns system error codes.

4.2.5.17 DotProduct

Syntax

```
DLL_EXPORT int DotProduct(char** errorChain, const int n,
    const double* a, const double* b, double* c)
```

This function calculates dot product of two input vectors, $c = a\dot{b}$. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
`n` [in]	Number of elements in vectors `a` and `b`.
`a` [in]	Vector, first operand of dot product.
`b` [in]	Vector, second operand of dot product.
`c` [out]	Result of the dot product; allocate 1-element (i.e. scalar) for result.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.18 ErfElementwise

Syntax

```
DLL_EXPORT int ErfElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the error function for each element of an input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values; must be length *n*.
z [out]	The result of erf(a); must be length *n*.

The API function returns system error codes.

4.2.5.19 ExpElementwise

Syntax

```
DLL_EXPORT int ExpElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the exponential of each element in the input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of powers; must be length *n*.
z [out]	The result of $\exp a$; must be length *n*.

The function returns system error codes.

4.2.5.20 GammaElementwise

Syntax

```
DLL_EXPORT int GammaElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the gamma function for each element of an input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values; must be length *n*.

z [out]	The result of gamma(a); must be length *n*.

The API function returns system error codes.

4.2.5.21 GammalnElementwise

Syntax

```
DLL_EXPORT int GammalnElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the natural logarithm of the absolute value of gamma function for each element of an input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values; must be length *n*.
z [out]	The result of erf(a); must be length *n*.

The API function returns system error codes.

4.2.5.22 GenerateRandomNormal

Syntax

```
DLL_EXPORT int GenerateRandomNormal(char** errorChain,
    const double mean, const double sigma, const int* seed,
    const int nSamples, double* samples)
```

Use this function to populate a given vector with random samples from Gaussian distribution. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
mean [in]	Mean of the underlying probability distribution.
sigma [in]	Standard deviation of the underlying probability distribution.
seed [in]	(NULL OK) Seed for setting the random number generator; if NULL, continues where the last generator left off; if NULL and there was no previous generator set, sets seed according to the current time.

nSamples [in]	Number of samples to generate and the size allocated for samples array.
samples [out]	Array to store randomly generated values.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.23 InverseElementwise

Syntax

```
DLL_EXPORT int InverseElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the inverse of the values in the provided input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Number of elements in the input array.
a [in]	Array to take the inverse values of; must be length *n*.
z [out]	The result of $1/a$; allocate to length *n*.

The API function returns system error codes.

4.2.5.24 Laguerre

Syntax

```
DLL_EXPORT int Laguerre(char** errorChain, const int n, const int m,
    const int nx, const double* x, double* v)
```

This function returns values of the associated Laguerre polynomials functions.[8] The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Radial mode number.
m [in]	Angular mode number.
nx [in]	Number of elements in *x*.
x [in]	Evaluation points; length *nx*.
v [out]	Associated Laguerre polynomial; length *nx*.

This function returns system error codes; -1 for error, 0 for success.

4.2.5.25 LnElementwise

Syntax

```
DLL_EXPORT int LnElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the natural logarithm of each element in the input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values to take the log of; must be length *n*.
z [out]	The result of $\ln(a)$; must be length *n*.

The function returns system error codes.

4.2.5.26 Log10Elementwise

Syntax

```
DLL_EXPORT int Log10Elementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the base 10 logarithm of each element in the input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values to take the log10 of; must be length *n*.
z [out]	The result of $\log 10(a)$; must be length *n*.

The function returns system error codes.

4.2.5.27 LSPolyFit

Syntax

```
DLL_EXPORT int LSPolyFit(char** errorChain, const int nvals,
    const double* x, const double* y, const int order, double* p)
```

This function computes a least-squares polynomial fit to the input data. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nvals [in]	Number of data point pairs to fit.
x [in]	X-values for the data points; must be length *nvals*.
y [in]	Y-values for the data points; must be length *nvals*.
order [in]	Order for the polynomial fit.
p [out]	Polynomial coefficients; must allocated to length *order* + 1.

The function returns system error codes; 1 for warning, -1 for error, 0 for success.

4.2.5.28 Maximum

Syntax

```
DLL_EXPORT int Maximum(char** errorChain, const double* x,
    const int nx, double* result)
```

This function finds the highest or most positive value in an array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Array to find the maximal value in.
nx [in]	Number of elements in *x*.
result [out]	The maximal value in the array *x*; must be length 1.

The API function returns system error codes.

4.2.5.29 MaximumV

Syntax

```
DLL_EXPORT int MaximumV(char** errorChain, const double* x,
    const double* y, const int nx, double* result)
```

This function finds the highest or most positive value between two input arrays. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Array to find the maximal value in.
y [in]	Array to find the maximal value in.
nx [in]	Number of elements in *x*.
result [out]	The maximal value in the array *x* or *y*; must be length *nx*.

The API function returns system error codes.

4.2.5.30 MaximumWithIndex

Syntax

```
DLL_EXPORT int MaximumWithIndex(char** errorChain, const double* x,
    const int nx, double* result, int* maxIndex)
```

This function returns the maximum value in the input array and the index where this particular max value was found. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Array to find the maximal value in.
nx [in]	Number of elements in *x*.
result [out]	The maximal value in the array *x*; allocate to length 1.
maxIndex [out]	Zero-based index into result where the maximum value was found; if there is more than one max value, returns the first one; allocate to length 1.

The API function returns system error codes.

4.2.5.31 MaxOfColumns

Syntax

```
DLL_EXPORT int MaxOfColumns(char** errorChain, const double* matrix,
    const int nCols, const int nRows, double* columnMaxs)
```

This function finds the highest or most positive value of each column of a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
matrix [in]	A row-major matrix to take the max of over the columns; must be length *nCols* × *nRows*.
nCols [in]	Number of elements in leading or consecutive dimension of *matrix*.
nRows [in]	Number of elements in second dimension of *matrix*.
columnMaxs [out]	Maximum value of each column; must be length *nCols*.

The API function returns system error codes.

4.2.5.32 MaxOfRows

Syntax

```
DLL_EXPORT int MaxOfRows(char** errorChain, const double* matrix,
    const int nCols, const int nRows, double* rowMaxs)
```

This function finds the highest or most positive value of each row of a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
matrix [in]	A row-major matrix to take the max of over the rows; must be length *nCols* × *nRows*.
nCols [in]	Number of elements in leading or consecutive dimension of *matrix*.
nRows [in]	Number of elements in second dimension of *matrix*.
rowMaxs [out]	Maximum value of each row; must be length *nRows*.

The API function returns system error codes.

4.2.5.33 Mean

Syntax

```
DLL_EXPORT int Mean(char** errorChain, const double* x, const int nx,
    double* result)
```

This function finds the average value of the vector. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Vector to find the average value over.
nx [in]	Number of elements in vector *x*.
result [out]	The mean value; length is always 1.

The API function returns system error codes.

4.2.5.34 MeanOfColumns

Syntax

```
DLL_EXPORT int MeanOfColumns(char** errorChain, const double* matrix,
    const int nCols, const int nRows, double* columnMeans)
```

This function finds the average value of each column of a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
matrix [in]	A row-major matrix to take the mean of over the columns; must be length *nCols* × *nRows*.
nCols [in]	Number of elements in leading or consecutive dimension of *matrix*.
nRows [in]	Number of elements in second dimension of *matrix*.
columnMeans [out]	Average value of each column; must be length *nCols*.

The API function returns system error codes.

4.2.5.35 MeanOfRows

Syntax

```
DLL_EXPORT int MeanOfRows(char** errorChain, const double* matrix,
    const int nCols, const int nRows, double* rowMeans)
```

This function finds the average value of each row of a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.

matrix [in]	A row-major matrix to take the mean of over the rows; must be length *nCols* × *nRows*.
nCols [in]	Number of elements in leading or consecutive dimension of *matrix*.
nRows [in]	Number of elements in second dimension of *matrix*.
rowMeans [out]	Average value of each row; must be length *nRows*.

The API function returns system error codes.

4.2.5.36 MINIMUM

Syntax

```
DLL_EXPORT int MINIMUM(char** errorChain, const double* x,
    const int nx, double* result)
```

This function finds the lowest or most negative value in an array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Array to find the minimal value in.
nx [in]	Number of elements in *x*.
result [out]	The minimal value in the array *x*; must be length 1.

The API function returns system error codes.

4.2.5.37 MINIMUMV

Syntax

```
DLL_EXPORT int MINIMUMV(char** errorChain, const double* x,
    const double* y, const int nx, double* result)
```

This function finds the lowest or most negative value between two input arrays. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Array to find the minimal value in.
y [in]	Array to find the minimal value in.
nx [in]	Number of elements in *x*.

result [out]	The minimal value in the array *x* or *y*; must be length *nx*.

The API function returns system error codes.

4.2.5.38 MinimumWithIndex

Syntax

```
DLL_EXPORT int MinimumWithIndex(char** errorChain, const double* x,
    const int nx, double* result, int* minIndex)
```

This function returns the minimum value in an array and the index of the result is also output. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Array to find the minimal value in.
nx [in]	Number of elements in *x*.
result [out]	The minimal value in the array *x*; allocate to length 1.
minIndex [out]	Index into *x* where the minimum result was found; if there are several equal minimum values the index to the first is returned; allocate to length 1.

The API function returns system error codes.

4.2.5.39 MinOfColumns

Syntax

```
DLL_EXPORT int MinOfColumns(char** errorChain, const double* matrix,
    const int nCols, const int nRows, double* columnMins)
```

This function finds the smallest or most negative value of each column of a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
matrix [in]	A row-major matrix to take the minimum of over the columns; must be length *nCols* × *nRows*.
nCols [in]	Number of elements in leading or consecutive dimension of *matrix*.

nRows [in]	Number of elements in second dimension of *matrix*.
columnMins [out]	Minimum value of each column; must be length *nCols*.

The API function returns system error codes.

4.2.5.40 MinOfRows

Syntax

```
DLL_EXPORT int MinOfRows(char** errorChain, const double* matrix,
    const int nCols, const int nRows, double* rowMins)
```

This function finds the lowest or most negative value of each row of a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
matrix [in]	A row-major matrix to take the minimum of over the rows; must be length *nCols* × *nRows*.
nCols [in]	Number of elements in leading or consecutive dimension of matrix.
nRows [in]	Number of elements in second dimension of matrix.
rowMins [out]	Minimum value of each row; must be length *nRows*.

The API function returns system error codes.

4.2.5.41 Mode

Syntax

```
DLL_EXPORT int Mode(char** errorChain, const double* x, const int nx,
    double* result)
```

This function finds the most frequently occurring value in the vector, the mode. If multiple values occur equally frequently, the minimum of those is used. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Vector to find the mode value over.
nx [in]	Number of elements in vector *x*.

result [out]	The most frequently occurring value; length is always 1.

The API function returns system error codes.

4.2.5.42 MZA_SORT

Syntax

```
DLL_EXPORT int MZA_SORT(char** errorChain, const int nx,
    const double* x0, double* x, int* ix)
```

This function sorts an array of data and returns sorted data and the indices for the sort. The function arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx in]	Number of elements to process.
x0 [in]	Vector of data to sort; must be length *nx*.
x [out]	Vector of sorted data; allocate to length *nx*.
ix [out]	Vector of index values for the sort such that `x[i] = x0[ix[i]]`; allocate length *nx*.

The API function returns system error codes.

4.2.5.43 MZA_VDPACKM

Syntax

```
DLL_EXPORT int MZA_VDPACKM(char** errorChain, const int nx,
    const double* x, const int* mask, double* y)
```

This function copies elements of an array with mask indexing to a vector with unit increment. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements to process.
x [in]	Array of values to pack; must be length *nx*.
mask [in]	Mask of values; must be length *nx*.

y [out]	Array with incremental values of set to *x*[*mask* != 0]; allocate to length sum(*mask* != 0).

The API function returns system error codes.

4.2.5.44 MZA_VDPACKV

Syntax

```
DLL_EXPORT int MZA_VDPACKV(char** errorChain, const int nx,
    const double* x, const int* idx, double* y)
```

This function copies elements of an array with vector indexing to a vector with unit increment. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements to process.
x [in]	Array of values to pack; must be at least length max(*idx*).
idx [in]	Index vector for the elements of *x*; must be at least length *nx*.
y [out]	Array with incremental values set to *x*[*idx*]; allocate to at least length *nx*.

The API function returns system error codes.

4.2.5.45 MZA_VDUNPACKM

Syntax

```
DLL_EXPORT int MZA_VDUNPACKM(char** errorChain, const int nx,
    const double* x, const int* mask, double* y)
```

This function copies elements of a vector with unit increment to an array with mask indexing. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements to process.
x [in]	Vector with unit indexing to unpack; length must be at least number of non-zero elements in *mask*.

mask [in]	Mask of values; must be at least length *nx*.
y [out]	Array with elements where *mask* != 0 set to incremental values of *x*; allocate to length *nx*.

The API function returns system error codes.

4.2.5.46 MZA_VDUNPACKV

Syntax

```
DLL_EXPORT int MZA_VDUNPACKV(char** errorChain, const int nx,
     const double* x, const int* idx, double* y)
```

This function copies elements of a vector with unit increment to an array with vector indexing. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements to process.
x [in]	Vector with unit indexing to unpack; must be at least length *nx*.
idx [in]	Vector of index values; must be at least length *nx*.
y [out]	Array with elements referenced by *idx* set to incremental values of *x*; allocate to length max(*idx*).

The API function returns system error codes.

4.2.5.47 NEGATE

Syntax

```
DLL_EXPORT int NEGATE(char** errorChain, const int nx, double* x)
```

This function reverses the sign on the input vector, in place. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	The number of elements in *x*.
x [in,out]	The array to negate; must be size *nx*.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.48 Normalize

Syntax

```
DLL_EXPORT int Normalize(char** errorChain, const int n,
    const double* v, double* x)
```

This function sets output vector to a vector in the same direction as the input vector but with Euclidean length 1. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Length of input and output vectors.
v [in]	Vector to find normal for; must be length *n*.
x [out]	Vector that is the normal of *v*; must be length *n*.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.49 NormalizeInPlace

Syntax

```
DLL_EXPORT int NormalizeInPlace(char** errorChain, const int n,
    double* x)
```

This function keeps vector's direction but sets the magnitude or Euclidean length to 1. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Length of vector *x*.
x [in,out]	Vector to be normalized; must be length *n*.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.50 NormalizeRows

Syntax

```
DLL_EXPORT int NormalizeRows(char** errorChain, const int n0,
    const int n1, const double* v, double* x)
```

This function sets the output array of vectors each to a vector in the same direction as the input vector but with Euclidean length 1. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n0 [in]	Number of elements in each row of *v*.
n1 [in]	Number of vectors in *v* of length *n0* to normalize.
v [in]	Array of vectors to find normal for; must be length *n0* × *n1*.
x [out]	Array of vectors that is the normal of *v*; must be length *n0* × *n1*.

The function returns system error codes.

4.2.5.51 NormalizeRowsInPlace

Syntax

```
DLL_EXPORT int NormalizeRowsInPlace(char** errorChain, const int n0,
    const int n1, double* x)
```

This function sets output array of vectors each to a vector in the same direction as the input vector but with Euclidean length 1. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n0 [in]	Number of elements in each row of *v*.
n1 [in]	Number of vectors in *v* of length *n0* to normalize.
x [in,out]	Array of vectors that is the normal of *v*; must be length *n0* × *n1*.

The function returns system error codes.

4.2.5.52 PowElementwise

Syntax

```
DLL_EXPORT int PowElementwise(char** errorChain, const int n,
    const double* a, const double b, double* z)
```

This function raises each element from the input array to a constant power. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a*, *b*, and *z*.
a [in]	An array to be raised, element-by-element, to *b*; must be length *n*.
b [in]	Power to raise *a* to, element-by-element; must be length *n*.
z [out]	The result of a^b; must be length *n*.

The function returns system error codes.

4.2.5.53 Pow2o3Elementwise

Syntax

```
DLL_EXPORT int Pow2o3Elementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function raises each element from the input array to the constant power 2/3. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array to be raised to 2/3, element-by-element; must be length *n*.
z [out]	The result of $a^{2/3}$; must be length *n*.

The function returns system error codes.

4.2.5.54 Product

Syntax

```
DLL_EXPORT int Product(char** errorChain, const int nx,
    const double* x, double* result)
```

This function computes the product of all of the elements of the input vector. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements in the input vector.
x [in]	Vector of values to multiply; must be length *nx*.
result [out]	Resulting product of elements in the input vector.

The API function returns system error codes.

4.2.5.55 ProductElementwise

Syntax

```
DLL_EXPORT int ProductElementwise(char** errorChain, const int n,
    const double* a, const double* b, double* z)
```

This function multiplies each element from an input array with the corresponding element in a second input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a*, *b*, and *z*.
a [in]	An array to be multiplied, element-by-element, with *b*; must be length *n*.
b [in]	An array to be multiplied, element-by-element, with *a*; must be length *n*.
z [out]	The result of a × b; must be length *n*.

The API function returns system error codes.

4.2.5.56 ProductElementwiseIncWithScalarIP

Syntax

```
DLL_EXPORT int ProductElementwiseIncWithScalarIP(char** errorChain,
    const int n, const int incx, double* x, const double y)
```

This function multiplies every *incx* element from an array with scalar value. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in x.
incx [in]	Increment for the elements of x.
x [in,out]	An array to be multiplied, every *incx* element, with y; must be length $1 + (n-1)$*incx*.
y [in]	Scalar value to be multiplied with x.

This function returns system error codes.

4.2.5.57 ProductElementwiseWithScalar

Syntax

```
DLL_EXPORT int ProductElementwiseWithScalar(char** errorChain,
    const int n, const double* x, const double y, double* z)
```

This function multiplies each element from an array by scalar value. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in x and z.
x [in]	An array to be multiplied, element-by-element, with y; must be length n.
y [in]	Scalar value to be multiplied with x.
z [out]	The result of $x \times y$; must be length n.

For this algorithm it is safe to have z and x point to the same array. The function returns system error codes.

4.2.5.58 ProductElementwiseWithScalarIP

Syntax

```
DLL_EXPORT int ProductElementwiseWithScalarIP(char** errorChain,
    const int n, double* x, const double y)
```

This function multiplies each element from vector x with scalar y. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *x*.
x [in,out]	An array to be multiplied, element-by-element, with *y*; must be length *n*.
y [in]	Scalar value to be multiplied with *x*.

This function returns system error codes.

4.2.5.59 QuotientElementwise

Syntax

```
DLL_EXPORT int QuotientElementwise(char** errorChain, const int n,
    const double* a, const double* b, double* z)
```

This function divides each element from vector a with the corresponding element in vector b. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a*, *b*, and *z*.
a [in]	An array to be divided, element-by-element, by *b*; must be length *n*.
b [in]	An array to divide *a* by, element-by-element; must be length *n*.
z [out]	The result of a/b; must be length *n*.

The API function returns system error codes.

4.2.5.60 RSSElementwise

Syntax

```
DLL_EXPORT int RSSElementwise(char** errorChain, const int n,
    const double* a, const double* b, double* z)
```

This function computes elementwise the square root of the sum of the squares of the input arrays. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a*, *b*, and *z*.
a [in]	An array to use to find the hypotenuse, element-by-element, with *b*; must be length *n*.
b [in]	An array to use to find the hypotenuse, element-by-element, with *a*; must be length *n*.
z [out]	The result of $\sqrt{a^2 + b^2}$; must be length *n*.

The function outputs system error codes.

4.2.5.61 RectangularGrid

Syntax

```
DLL_EXPORT int RectangularGrid(char** errorChain, const int nx,
    const double* x, const int ny, const double* y, double* gx,
    double* gy)
```

This function returns the coordinates of a rectangular grid of points. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements in vector *x*.
x [in]	Vector of x-points; length *nx*.
ny [in]	Number of elements in vector *y*.
y [in]	Vector of y-points; length *ny*.
gx [out]	Grid of x-points; length $nx \times ny$.
gy [out]	Grid of y-points; length $nx \times ny$.

The function returns system error codes.

4.2.5.62 SinElementwise

Syntax

```
DLL_EXPORT int SinElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the sine of each element in the input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values to take the sine of; must be length *n*.
z [out]	The result of sin(a); must be length *n*.

The function returns system error codes.

4.2.5.63 SpaceEvenly

Syntax

```
DLL_EXPORT int SpaceEvenly(char** errorChain, const double start,
    const double end, const int n, double* x)
```

This function populates a vector with values evenly spaced over the specified interval, similar to MATLAB® function `linspace`. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
start [in]	First value in interval.
end [in]	Last value in interval.
n [in]	The total number of values in the interval.
x [out]	The evenly spaced values from start to end inclusive; allocate to length *n*.

The function returns system error codes; -1 for error, 0 for success.

4.2.5.64 SqrtElementwise

Syntax

```
DLL_EXPORT int SqrtElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the square root of each element in the input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array to take the square root of, element-by-element; must be length *n*.
z [out]	The result of $\sqrt{(a)}$; must be length *n*.

This function returns system error codes.

4.2.5.65 SquareElementwise

Syntax

```
DLL_EXPORT int SquareElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function squares each element of the input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array to be squared, element-by-element; must be length *n*.
z [out]	The result of a^2; must be length *n*.

The function returns system error codes.

4.2.5.66 Sum

Syntax

```
DLL_EXPORT int Sum(char** errorChain, const int nx, const double* x,
    double* result)
```

This function computes the sum of all of the elements of the input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	The number of elements in *x*.

x [in]	The array to sum; must be size *nx*.
result [out]	Resulting sum of values in *x*.

The API function returns system error codes.

4.2.5.67 SumElementwise

Syntax

```
DLL_EXPORT int SumElementwise(char** errorChain, const int n,
    const double* a, const double* b, double* z)
```

This function sums each element in an input array with the corresponding element in a second array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a*, *b*, and *z*.

a [in]An array to be summed, element-by-element, with *b*; must be length *n*.
b [in]An array to be summed, element-by-element, with *a*; must be length *n*.
z [out]The result of $a + b$; must be length *n*.

The API function returns system error codes.

4.2.5.68 SumElementwiseWithScalar

Syntax

```
DLL_EXPORT int SumElementwiseWithScalar(char** errorChain,
    const int n, const double* x, const double y, double* z)
```

This function adds a scalar value to each element in an array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *x*, *y*, and *z*.
x [in]	An array to be summed, element-by-element, with *y*; must be length *n*.
y [in]	Scalar value to be added to *x*; must be length *n*.

z [out]	The result of $x + y$; must be length *n*.

For this algorithm it is safe to have *z* and *x* point to the same array. The API function returns system error codes.

4.2.5.69 SumOfColumns

Syntax

```
DLL_EXPORT int SumOfColumns(char** errorChain, const double* matrix,
    const int nCols, const int nRows, double* columnSums)
```

This function finds the sum of each column of a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
matrix [in]	A row-major matrix to take the sum of over the columns; must be length *nCols* × *nRows*.
nCols [in]	Number of elements in leading or consecutive dimension of *matrix*.
nRows [in]	Number of elements in second dimension of *matrix*.
columnSums [out]	Sum of each column; must be length *nCols*.

The API function returns system error codes.

4.2.5.70 SumOfRows

Syntax

```
DLL_EXPORT int SumOfRows(char** errorChain, const double* matrix,
    const int nCols, const int nRows, double* rowSums)
```

This function finds the sum of each row of a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
matrix [in]	A row-major matrix to take the sum of over the rows; must be length *nCols* × *nRows*.
nCols [in]	Number of elements in leading or consecutive dimension of *matrix*.
nRows [in]	Number of elements in second dimension of *matrix*.

rowSums [out]	Sum of each row; must be length *nRows*.

The API function returns system error codes.

4.2.5.71 TanElementwise

Syntax

```
DLL_EXPORT int TanElementwise(char** errorChain, const int n,
    const double* a, double* z)
```

This function computes the tangent of each element in the input array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a* and *z*.
a [in]	An array of values to take the tangent of; must be length *n*.
z [out]	The result of $\tan(a)$; must be length *n*.

The function returns system error codes.

4.2.5.72 VectorScalarProductWithSumIP

Syntax

```
DLL_EXPORT int VectorScalarProductWithSumIP(char** errorChain,
    const int n, const double a, double* x, const int incx, double* b,
    double* y, const int incy)
```

This function multiplies each element from an input array with a scalar and adds result to a scalar times another input array ($ax + by$). The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *x* and *y*.
a [in]	Scalar multiplier on *x*.
x [in]	An array to be multiplied, element-by-element, with *a*; must be length *n*.
incx [in]	Increment for the elements of *x*.

b [in]	(NULL OK) Scalar multiplier on *y*; if null, *b* = 1.
y [in,out]	Result of $ax + by$; must be length *n*.
incy [in]	Increment for the elements of *y*.

The function returns system error codes.

4.2.5.73 VECTOR2NORM

Syntax

```
DLL_EXPORT int VECTOR2NORM(char** errorChain, const int n,
     const double* x, const int incx, double* z)
```

This function computes the Euclidean norm of a vector as

$$z = \sqrt{\sum x_i^2}$$

The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	The number of elements in *a*.
x [in]	An array to be normed; must be length *n*.
incx [in]	Increment for the elements of *a*.
z [out]	The result of norm.

The function returns system error codes.

4.2.6 Matrix Functions

4.2.6.1 CHOL

Syntax

```
DLL_EXPORT int CHOL(char** errorChain, const int nRowsCols, double* a,
     const char* lu)
```

This function performs Cholesky factorization for real numbers. The input matrix assumed to be row major (i.e. elements in a row are stored consecutively) and must be positive definite. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nRowsCols [in]	Number of rows in matrix *a* and result.
a [in,out]	Matrix to be factored, must be stored in row major format and be positive definite.
lu [in]	String indicating lower ("L") or upper ("U") triangular for result.

The function returns system error codes; -1 for error, 0 for success. Note that the result is returned in *a*.

4.2.6.2 Fft

Syntax

```
DLL_EXPORT int Fft(char** errorChain, const bool isForward,
     const int nx0, const int nx1, double* xRe, double* xIm)
```

This function performs in-place FFT with real and imaginary numbers. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
isForward [in]	True to negate the sum in the DFT exponent and false otherwise.
nx0 [in]	Number of elements along sequential dimension in 2D DFT.
nx1 [in]	Number of elements along 2nd dimension in 2D DFT.
xRe [in,out]	Real component of complex input/result; must be size *nx0* $\times$ *nx1*.
xIm [in,out]	Imaginary component of complex input/result; must be size *nx0* $\times$ *nx1*.

The function returns system error codes; -101 to -103 for errors setting up FFT, -1 for other error, 0 for success.

4.2.6.3 FftShift

Syntax

```
DLL_EXPORT int FftShift(char** errorChain, const double* x,
     const int nx0, const int nx1, double* y)
```

This function swaps rows and columns about midpoint in array. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Matrix to swap around; must have *nx0* × *nx1* elements.
nx0 [in]	Number of elements in sequential dimension.
nx1 [in]	Number of elements in 2nd dimension.
y [in]	Result of swapping; must have *nx0* × *nx1* elements; will be same shape as *x*.

The function returns system error codes; -1 for error, 0 for success.

4.2.6.4 FftShiftInPlace

Syntax

```
DLL_EXPORT int FftShiftInPlace(char** errorChain, const int nx0,
    const int nx1, double* xy)
```

This function performs in-place FFT shift. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx0 [in]	Number of elements in sequential dimension.
nx1 [in]	Number of elements in 2nd dimension.
xy [in,out]	Matrix to swap around and result of swapping.

The function returns system error codes; -1 for error, 0 for success.

4.2.6.5 IdentityMatrix

Syntax

```
DLL_EXPORT int IdentityMatrix(char** errorChain, const int n,
    double* x)
```

This function initializes a square matrix to identity. Here, "identity matrix" is defined as a square matrix with ones along the diagonal and zeros elsewhere. Row/column major doesn't matter here because the transpose of x is equal to x. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Number of rows and columns in the square matrix; the output array should be length $n \times n$.
x [out]	Identity matrix; allocate to size $n \times n$.

The function returns system error codes; -1 for error, 0 for success.

4.2.6.6 InverseLU

Syntax

```
DLL_EXPORT int InverseLU(char** errorChain, const int nRowsCols,
     double* matrixA, int* ipiv)
```

This function computes the matrix inverse for real numbers; input matrix assumed to be row major (i.e. elements in a row are stored consecutively) and must be square. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nRowsCols [in]	Number of rows and columns in *matrixA* and *result*.
matrixA [in/out]	Matrix to be factored, must be stored in row major format and square.
ipiv [in/out]	array of pivot info, allocate to *nRowsCols* × *nRowsCols*.

The function returns system error codes; -1 for error, 0 for success.

4.2.6.7 LinearInterpolate1D

Syntax

```
DLL_EXPORT int LinearInterpolate1D(char** errorChain, const int nx,
     const double* x, const int ny, const double* y, const int nxx,
     const double* xx, const double* extrapVal, double* yy)
```

This function performs linear interpolation for [y1,y2,...,yny]=[f1(x),f2(x),...fny(x)], where x is a vector and y is a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements in *x*.

x [in]	The independent variable; must be in order.
ny [in]	The number of y vectors length x.
y [in]	The dependent variable; there can be multiple y values for each x value; stored [y(1,1),y(1,2),. . . ,y(1,nx),. . . ,y(ny,1),y(ny,2),. . . ,y(ny,nx)].
nxx [in]	Number of elements in *xx*, the lookups into f(x).
xx [in]	Lookup values or interpolation sites; for each *xx* value there will be *ny* × *nxx* results.
extrapVal [in]	(NULL OK) Extrapolation value. If NULL, use linear extrapolation.
yy [out]	Result of interpolation; must be size *ny* × *nxx*; stored in the same way as *y*.

The function returns system error codes; -1 for error, 0 for success.

4.2.6.8 LinearInterpolate2D

Syntax

```
DLL_EXPORT int LinearInterpolate2D(char** errorChain, const int nx,
     const int ny, const double* x, const double* y, const double* z,
     const int nxx, const int nyy, const double* xx, const double* yy,
     const double* extrapVal, double* zz)
```

This function implements linear interpolation for f(x,y)=z at points xx and yy. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements in vector *x*; number of elements along sequential dimension of *z*.
ny [in]	Number of elements in vector *y*; number of elements along 2nd dimension of *z*.
x [in]	Vector of x positions at which *z* is defined; must be size *nx*; assumed to be in ascending order.
y [in]	Vector of y positions at which *z* is defined; must be size *ny*; assumed to be in ascending order.
z [in]	Values of f(x0 element of x, y0 element of y); must be size *nx* × *ny*.
nxx [in]	Number of elements in vector *xx*; number of elements along sequential dimension of *zz*.

nyy [in]	Number of elements in vector *yy*; number of elements along 2nd dimension of *zz*.
xx [in]	Vector of x positions at which *zz* values are desired; must be size *nxx*.
yy [in]	Vector of y positions at which *zz* values are desired; must be size *nyy*.
extrapVal [in]	(NULL OK) Extrapolation value. If NULL, use linear extrapolation.
zz [out]	Values of f(xx,yy) for each xx,yy pair via linear interpolation; must be allocated to size *nxx* × *nyy*.

The function returns system error codes; -1 for error, 0 for success.

4.2.6.9 LinearInterpolate2DVectorOut

Syntax

```
DLL_EXPORT int LinearInterpolate2DVectorOut(char** errorChain,
     const int nx, const int ny, const double* x, const double* y,
     const double* z, const int nxxyy, const double* xx, const double* yy,
     double* zz)
```

This function is a variation of LinearInterpolate2D where the output is a vector and not a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements in vector *x*; number of elements along sequential dimension of *z*.
ny [in]	Number of elements in vector *y*; number of elements along 2nd dimension of *z*.
x [in]	Vector of x positions at which z is defined; must be size *nx*; assumed to be in ascending order.
y [in]	Vector of y positions at which z is defined; must be size *ny*; assumed to be in ascending order.
z [in]	Values of table defined precisely at points formed by combinations of (x,y); must be size *nx* × *ny*; stored with x values varying the quickest and z values varying the slowest.
nxxyy [in]	Number of elements in vectors *xx*, *yy*, and *zz*.
xx [in]	Vector of x positions at which *zz* values are desired; must be size *nxxyy*.

yy [in]	Vector of y positions at which *zz* values are desired; must be size *nxxyy*.
zz [out]	Values of f(xx,yy) for each xx,yy pair via linear interpolation; must be allocated to size *nxxyy*.

The function returns system error codes; -1 for error, 0 for success. The function fills in NaN when values are outside the bounds of the table.

4.2.6.10 LinearInterpolate3DVectorOut

Syntax

```
DLL_EXPORT int LinearInterpolate3DVectorOut(char** errorChain,
    const int nx, const int ny, const int nz, const double* x,
    const double* y, const double* z, const double* v, const int nxx,
    const double* xx, const double* yy, const double* zz, double* vv)
```

This function performs linear lookup into a 3D table. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx [in]	Number of elements in vector *x*; number of elements along sequential dimension of *v*.
ny [in]	Number of elements in vector *y*; number of elements along 2nd dimension of *v*.
nz [in]	Number of elements in vector *z*; number of elements along 3rd dimension of *v*.
x [in]	Vector of *x* positions at which *v* is defined; must be size *nx*; assumed to be in ascending order.
y [in]	Vector of *y* positions at which *v* is defined; must be size *ny*; assumed to be in ascending order.
z [in]	Vector of *z* positions at which *v* is defined; must be size *nz*; assumed to be in ascending order.
v [in]	Values of table defined precisely at points formed by combinations of (x,y,z); must be size *nx* × *ny* × *nz*; stored with x values varying the quickest and z values varying the slowest.
nxx [in]	Number of elements in vectors *xx*, *yy*, *zz*, and *vv*.
xx [in]	Vector of x positions at which *vv* values are desired; must be size *nxx*.
yy [in]	Vector of y positions at which *vv* values are desired; must be size *nxx*.

zz [in]	Vector of z positions at which *vv* values are desired; must be size *nxx*.
vv [out]	Values of f(xx,yy,zz) for each xx,yy,zz pair via linear interpolation; must be allocated to size *nxx*.

This function returns system error codes; -1 for error, 0 for success. The function fills in NaN when values are outside the bounds of the table.

4.2.6.11 LinearSolve

Syntax

```
DLL_EXPORT int LinearSolve(char** errorChain, const int n,
    const double* a, const double* b, double* x)
```

This function solves a system of linear equations for a square matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
n [in]	Number of rows and columns in matrix *a*; number of elements in vectors *b* and *x*.
a [in]	Matrix in the equation $Ax = b$ where each row contains coefficients of the left-hand-side of the system of equations; this matrix is stored in row-major format which means that the items along each row are stored consecutively in memory.
b [in]	Column vector in the equation $Ax = b$ which is the right-hand-side of the equations.
x [out]	Solution to the system of linear equations, x in $Ax = b$.

This function returns system error codes; -1 for error, 0 for success.

4.2.6.12 MatrixMultiply

Syntax

```
DLL_EXPORT int MatrixMultiply(char** errorChain, const double alpha,
    const int nRowsA, const int nColsA, const double* a,
    const int nRowsB, const int nColsB, const double* b, double* result)
```

This function computes general matrix multiply for real numbers; input matrices assumed to be row major (i.e. elements in a row are stored consecutively). The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
alpha [in]	Scalar to multiply matrices by.
nRowsA [in]	Number of rows in matrix *a* and *result*.
nColsA [in]	Number of columns in matrix *a*; must be equal to *nRowsB*.
a [in]	First matrix in $a \times b = result$, must be stored in row major format.
nRowsB [in]	Number of rows in matrix *b*; must be equal to *nColsA*.
nColsB [in]	Number of columns in matrix *b* and *result*.
b [in]	Second matrix in $a \times b = result$, must be stored in row major format.
result [out]	Result of multiplication, $alpha \times a \times b$; must be allocated to *nRowsA* × *nColsB*; will be stored row major.

The function returns system error codes; -1 for error, 0 for success.

4.2.6.13 MatrixMultiplyAndTranspose

Syntax

```
DLL_EXPORT int MatrixMultiplyAndTranspose(char** errorChain,
     const double alpha, int nRowsA, int nColsA, const double* a,
     const bool transposeA, int nRowsB, int nColsB, const double* b,
     const bool transposeB, double* result)
```

This function performs general transpose and matrix multiply for real numbers; input matrices assumed to be row major (i.e. elements in a row are stored consecutively). The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
alpha [in]	Scalar to multiply matrices by.
nRowsA [in]	Number of rows in matrix *a* and *result*.
nColsA [in]	Number of columns in matrix *a*; must be equal to *nRowsB*.
a [in]	First matrix in $a \times b = result$, must be stored in row major format.

transposeA [in]	True if matrix *a* should be transposed before multiplying; in this case *nRowsA* and *nColsA* will automatically be swapped by the routine, so set them to the original shape of *a*.
nRowsB [in]	Number of rows in matrix *b*; must be equal to *nColsA*.
nColsB [in]	Number of columns in matrix *b* and *result*.
b [in]	Second matrix in $a \times b = result$, must be stored in row major format.
transposeB [in]	True if matrix *a* should be transposed before multiplying; in this case *nRowsB* and *nColsB* will automatically be swapped by the routine, so set them to the original shape of *b*.
result [out]	Result of multiplication, $alpha \times a \times b$; must be allocated to *nRowsA* × *nColsB*; will be stored row major.

The function returns system error codes; -1 for error, 0 for success.

4.2.6.14 PiecewiseCubicHermiteInterpolate1D

Syntax

```
DLL_EXPORT int PiecewiseCubicHermiteInterpolate1D(char** errorChain,
    const int nxy, const double* x, const double* y, const int nxxyy,
    const double* xx, double* yy)
```

This function implements cubic polynomial interpolation, first derivative continuity.[57] Extrapolation does not occur if *xx* is outside the bounds of *x* then NaN is stored in *yy*. Smoother than LinearInterpolate1D since continuity is achieved on the first derivative; however, continuity is not guaranteed on the second derivative. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nxy [in]	Number of elements in array *x* and array *y*; must be at least 3 for the algorithm to work.
x [in]	Known samples in lookup table function f(x)=y; does not need to be sorted - sorting will take place automatically; must be length *nxy*.
y [in]	Values stored in lookup table at points *x*; must be length *nxy*.
nxxyy [in]	Number of elements in arrays *xx* and *yy*.
xx [in]	Points at which values are desired; must be length *nxxyy*.

yy [out]	Interpolated values, f(xx) =yy; must allocate to length *nxxyy*.

The function returns system error codes.

4.2.6.15 QRDecomp

Syntax

```
DLL_EXPORT int QRDecomp(char** errorChain, const int nRows,
    const int nCols, double* matrixQ, double* matrixR)
```

This function calculates the Q-R decomposition of a matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nRows [in]	Number of rows in input matrix.
nCols [in]	Number of columns input matrix.
matrixQ [in, out]	Matrix to decompose of size *nRows* $\times$ *nCols*; is overwritten to become output matrix Q of QR decomposition.
matrixR [out]	Decomposed matrix R of QR decomposition.

The function returns system error codes. Will return -101 if call to MKL has an error flag (most common issue is that the matrix is not invertible).

4.2.6.16 SingularValueDecomp

Syntax

```
DLL_EXPORT int SingularValueDecomp(char** errorChain,
    const int nRowsA, const int nColsA, double* matrixA, double* matrixU,
    double* matrixS, double* matrixVT)
```

This function performs Singular Value Decomposition for real numbers; input matrices assumed to be row major (i.e. elements in a row are stored consecutively). The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nRowsA [in]	Number of rows in matrix *matrixA* and result.
nColsA [in]	Number of columns in matrix *matrixA*.

matrixA [in]	Matrix to be factored, must be stored in row major format.
matrixU [out]	Matrix of left singular vectors, stored in row major format.
matrixS [out]	Array of singular values.
matrixVT [out]	Matrix, transpose of the right singular vectors, stored in row major format.

The function returns system error codes; -1 for error, 0 for success.

4.2.6.17 TRANSPOSE

Syntax

```
DLL_EXPORT int TRANSPOSE(char** errorChain, const double* x,
    const int nx0, const int nx1, double* t)
```

This function performs transpose and stores result in separate memory. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
x [in]	Matrix to transpose.
nx0 [in]	Number of elements in original sequential dimension.
nx1 [in]	Number of elements in original non-consecutive dimension.
t [out]	Transposed matrix, size *nx1* $\times$ *nx0* (must be allocated by caller, as always).

The function returns system error codes; -1 for error, 0 for success. Use TRANSPOSEINPLACE to perform in-place transpose.

4.2.6.18 TRANSPOSEINPLACE

Syntax

```
DLL_EXPORT int TRANSPOSEINPLACE(char** errorChain, const int nx0,
    const int nx1, double* xt)
```

This function performs in-place transpose of an input matrix. The arguments are

errorChain [in,out]	Holds error and warning messages- deepest error is first.
nx0 [in]	Number of elements in original sequential dimension.
nx1 [in]	Number of elements in original non-consecutive dimension.
xt [in,out]	Matrix to transpose and transposed matrix, originally size *nx0* × *nx1* and changed to size *nx1* × *nx0*.

The function returns system error codes; -1 for error, 0 for success. Use TRANSPOSE to perform a not-in-place transpose.

Index

A

AbsElementwise ... 433
AcosElementwise ... 434
addDouble ... 420
addInt ... 420
addModelType ... 350
AddPolyFit ... 109
addString ... 420
AeroOpticsScaling ... 352
AFGLAMOS ... 249
AFRLDE_Atmos ... 250
AIRS ... 251
AnisoplanaticJitter ... 167
AnisoplanaticStrehl ... 169
APImessage ... 123
APIstruct ... 123
  getChar ... 124
  getDbl ... 124
  getInt ... 124
  getXML ... 124
  setChar ... 124
  setDbl ... 124
  setInt ... 124
  struct ... 125
  struct2xml ... 125
  writeXML ... 125
  xml2struct ... 125
APIwarning ... 125
ApparentGravity ... 252
AreaAdditionStrehl ... 353
arrayFillD ... 434
arrayFillI ... 435
AsinElementwise ... 435
Atan2Elementwise ... 436
AtanElementwise ... 435
AtmEval ... 381
ATMLookupPath ... 349
AtmModels ... 253
AtmosphericOTF ... 171
AtmStruct ... 161
AtmStruct2Cell ... 354
AtmStruct_Interface ... 354
AverageAtm ... 255

B

BeaconD0 ... 173
BeamIrradianceCovariance ... 174
BeamSpread ... 174
BesselJ ... 427
bilLoad ... 110
Boost2Flyout ... 110
BoostLoad ... 111
BoundaryAlt ... 176
BoundaryAtm ... 256
boundaryAtmEval ... 383
Bufton ... 257
buildLEEDRAtmos ... 326
buildLEEDRWxCube ... 327

C

CalcHRange ... 258
CartesianToPolar ... 427
CartesianToSpherical ... 428
CaseStruct ... 90
cell2csv ... 127
cell2Excel ... 128
CFLOSSites ... 329
ChangeAtm ... 260
ChangeRd ... 33
checkGenericModels ... 384
Chol ... 468
CircFresnel ... 177
clear ... 421
Clear1Night ... 262
Clear2Night ... 263
Clear2Wind ... 264
clickableLegend ... 128
CloudFreeHT ... 36
cn2Eval ... 388
coeffEval ... 388
CommonSites ... 37
ComplexDivide ... 429
ComplexElementwiseDivide ... 436
ComplexElementwiseMultiply ... 437
ComplexMultiply ... 429
CompStruct ... 129
ComputeBILLPoint ... 375
ConvertTZ ... 112
CopyVector ... 438
CosElementwise ... 438
CreateScreenLocations ... 376
CreateScreenLocationsECF ... 377
CreateScreenNormPositions ... 378
crop ... 418
CrossProduct ... 439
CrossProductArray ... 439

D

DegreesToRadians ... 430
DegreesToRadiansVector ... 439
densEval ... 389
dewPt2RH ... 265
dewPtEval ... 390
Difference ... 440
DifferenceElementwise ... 440
DistortionNumber ... 179
DLRmodHV57 ... 266
DotProduct ... 441
downloadGrbData ... 330
dynamicPiecewiseMaritimeAIRS ... 267
DynamicViscosity ... 267

E

EarthAlt 37
EarthAltECF 379
ecf2eci 41
ecf2enu 42
ecf2LLA 45
eci2ecf 48
EdlenOwens67 268
EngagementStruct 29
enu2ecf 49
EqualCn2Screens 269
ErfElementwise 441
errorCheck 415
ErrorFunction 430
ExpElementwise 442
extCoeffs 383
Extract 130
ExtractBaseModel 355

F

FacetNorms 51
FAS_TAPE6 356
FASCODE 270
Fft 469
FftShift 469
FftShiftInPlace 470
FindStructNaN 132
Fix 430
FlyoutGeom 53
FlyoutGeomGC 380
FocusStrehl 183
FortRead 111
FresnelInt 344
FWReckon 55

G

G_Interface 93
GADSAtmos 271
Gamma 430
GammaElementwise 442
Gammaln 430
GammalnElementwise 443
GenerateRandomNormal 443
GeomStruct 22
getAlpha 357
getDouble 421
getDoubleNames 421
GetFieldDataPath 411
GetFieldDataPathLength 411
getFullFilePath 425
getInt 421
getIntNames 422
GetLookupPath 412
GetLookupPathLength 412
getNextPower2 431
GetParamPath 413
GetParamPathLength 414
getString 422
getStringNames 422
GetTargDataPath 414
GetTargDataPathLength 415
getTargetSize 94
getWarning 416
getXMLtext 423
GrdAltProfile 58
GreenwoodCn2 273
GreenwoodFreq 184
Gregorian2Julian 113

H

HigherOrderPSD 186
hline 132
HufnagelValley 274
HV57 276
HYPER 345
hyper2f3as 345

I

IdentityMatrix 470
InnerScale 276
innerScaleEval 391
inpaint_nans 133
InverseElementwise 444
InverseLU 471
IrradianceCovariancePathWeight 189
IsClose 134
isDaylight 134
isDouble 423
IsGoodGeom 59
IsGoodHitPoint 60
IsInfinite 426
isInt 423
IsNan 426
isString 423

J

Julian2Gregorian 113

L

Laguerre 444
LaserField 95
LEEDR_clearModelData 397
LEEDR_setModelData 397
LEEDRAtm 278
LEEDRAtmos 279
LEEDRAtmos_GetLUTFN 397
LEEDRAtmos_GetLUTFNLength 397
leedrAtmosEval 385
LEEDRLookupPath 349
LEEDRLUT 330
LEEDRSiteAltitude 398
LEEDRSiteLatLon 398
LEEDRSites 332
LEEDRWxCube 282
LEEDRWxCube_clearModelData 399
LEEDRWxCube_GetAlts 399
LEEDRWxCube_GetData 399
LEEDRWxCube_GetLUTInfo 400

LEEDRWxCube_GetLUTInfoLengths 400
LEEDRWxCube_GetNumWavelengths 401
LEEDRWxCube_GetSurfaceAlts 402
LEEDRWxCube_GetSurfaceData 402
LEEDRWxCube_GetWavelength 401
LEEDRWxCube_LoadLookupTable 403
LEEDRWxCubeAtm 284
leedrWxCubeEval 386
LEEDRWxCubeLookupPath 349
LEEDRWXCUBELUT 332
LinearInterpolate1D 471
LinearInterpolate2D 472
LinearInterpolate2DVectorOut 473
LinearInterpolate3DVectorOut 474
LinearSolve 475
LineIntersect 135
LLA2ecf 60
LnElementwise 444
loadH5 126
loadLEEDRAtmos 335
loadLEEDRWxCube 335
loadMZAScalingCodeAPI 126
LoadStampParams 112
loadXML 423
LocalEarthRadius 63
Log10Elementwise 445
LogScaling 285
logScalingEval 386
Lommel 346
LookupPath 349
LOS 114
losAngle 65
LowerCase 425
LSPolyFit 445
LUTAtm 286

M

MarkFig 135
MatrixMultiply 475
MatrixMultiplyAndTranspose 476
Maui3Cn2 287
Maximum 446
MaximumV 446
MaximumWithIndex 447
MaxOfColumns 447
MaxOfRows 448
mcodeHelp 136
Mean 448
MeanOfColumns 449
MeanOfRows 449
meanRefract 335
mergeLEEDRAtmos 336
MIEV0 288
Minimum 450
MinimumV 450
MinimumWithIndex 451
MinOfColumns 451
MinOfRows 452
mkFuncStr 357
MOD_TAPE6 358
Mode 452
MODFAS 291
ModHufnagelValley 294
MODTRAN 295
MoonPhase 114
MoonPos 115
mza_sort 453
mza_vdPackM 453
mza_vdPackV 454
mza_vdUnpackM 454
mza_vdUnpackV 455
MZAneuralNetCn2Scaling 340
MZAsurfCn2 341

N

Nan 426
Negate 455
NegativeInfinity 426
NollMatrix 190
Normalize 455
NormalizeInPlace 456
NormalizeRows 456
NormalizeRowsInPlace 457
NSLOT_v2 297

O

ObjectIncidence 66
ObjectIntersect 69
OceanAtmos 298
One 426
OpenLoopJitter 191
OpenLoopStrehl 192
OpenLoopStrehlReverseLookup 396
OuterScale 299
outerScaleEval 391
outputOn 416

P

pad 418
ParamPath 126
ParamStruct 98
ParamStructClearLast 403
ParamStructCreate 404
ParamStructDestroy 404
ParamStructGetChar 404
ParamStructGetCharLen 405
ParamStructGetDbl 405
ParamStructGetDblArray 406
ParamStructGetInt 407
ParamStructGetIntArr 407
ParamStructGetXML 408
ParamStructGetXMLLen 408
ParamStructRemoveField 409
ParamStructSetChar 409
ParamStructSetDbl 410
ParamStructSetInt 410
PathDisp 302
pCFLOS 337

pdfOpen 136
PhasorSumStrehl 359
PhysicalConst 71
PiecewiseCubicHermiteInterpolate1D 477
PlaneD0 194
PlaneIrradianceCovariance 194
PlaneR0 195
PlaneRytov 196
PlaneWaveStrFcn 197
plotAtm 360
plotEng 99
plotGeom 101
plotObject 102
PolarToCartesian 431
PositiveInfinity 427
Pow2o3Elementwise 458
PowElementwise 457
pressEval 392
print 424
printAll 424
processAIRShdf 300
Product 458
ProductElementwise 459
ProductElementwiseIncWithScalarIP 459
ProductElementwiseWithScalar 460
ProductElementwiseWithScalarIP 460
progressbar 136
Propagation Geometry 33
PropConfig 361
PropControl_Interface 362
PropMeshParams 363
PropParams 197
propTLE 115
PSDGenRandom 138
PupilProp 199

Q

QRDecomp 478
QuotientElementwise 461

R

RadiansToDegrees 432
RandCn2 306
readMetData 342
RecommendedMesh 367
RectangularGrid 462
RefIndex 308
RefractAlt 199
Remainder 432
remove 424
removeGroup 424
ReverseAreaAddition 368
ReverseAtm 369
ReverseGeom 71
rhEval 393
RotateObject 72
rotateticklabel 138
Round 432
RSSElementwise 461

S

saveH5 126
saveLEEDRAtmos 338
saveMODFASAsH5 350
ScreenData 370
ScreenECF 74
ScreenR0 201
SDP4 116
seeOut 416
SetFieldDataPath 412
setLEEDRboundaryLayerAlt 338
setLEEDRInputs 338
setLEEDRpath 326
SetLookupPath 413
SetParamPath 414
SetTargDataPath 415
setWarning 416
sgamma 347
SGP4 117
SGPPrep 118
SiderealTime 119
Sign 432
SimpleGeom 76
SinElementwise 462
SingularValueDecomp 478
Slant2Down 79
Slant2DownEL 80
SLCDay 308
SLCNight 309
SOR2miAtm 310
SORWind 311
SpaceEvenly 463
SpeedOfSound 311
SphericalIrradianceCovariance 202
SphericalIrradianceVariance 203
SphericalLogIrrCovariance 204
SphericalLogIrrPathWeight 205
SphericalR0 206
SphericalRytov 208
SphericalTheta0 210
SphericalToCartesian 433
SphericalWaveStrFcn 213
SqrtElementwise 463
SquareElementwise 464
Strehl2wfe 372
StrehlEquivFocus 213
struct2vec 139
structArray2arrayStruct 140
Sum 464
SumElementwise 465
SumElementwiseWithScalar 465
SumOfColumns 466
SumOfRows 466
SunPos 119
SunRiseSet 120
syntaxHelp 140

T

TALOEngage 112

TanElementwise 467
TargCoordVec 81
TargDataPath 127
TargGUI 103
TargPtAngle 82
TargStruct 103
TBWaveCalc 215
tempEval 393
TerrainAtm 313
TerrainProfiler 84
ThermalBlooming 218
TiltAngCovScale 224
TiltCovariance 226
TiltCovariancePathWeight 227
TiltTempCovScale 228
TiltVariance 230
TLEExtract 121
TLEParse 122
Transmission 232
Transpose 479
TransposeInPlace 479
TransVelocity 85
TurbConst 233
TylerFreq 234

U

UniformAtm 314
uniformAtmEval 387
UniformCn2 315
unloadMZAScalingCodeAPI 127
Unwrap 419
updateLEEDRAtmos 339
updateLEEDRInputs 340
US_Standard76 316

V

VaryStruct 141
vector2Norm 468
VectorArgCheck 141
VectorScalarProductWithSumIP 467
vline 142
VTP2VLLA 317
VTP2Vxy 88

W

WaveStrFcnPathWeight 236
WeightFunctionRef 236
WeightMat 238
wfe2Strehl 372
WhichCommonSite 90
Wind2Vxy 373
WindDirCos 238
windEval 394
windHeadingEval 395
wlString 396
WSMRatm 320
WSMRCn2 321
WSMRWind 322
WSMRWindHeading 322
wtBLS900 241

Z

ZernikeCoeffCov 241
ZernikeCoefficient 242
ZernikeCoord 242
ZernikeCovariancePathWeight 243
ZernikeOrder 348
ZernikePolynomial 245
ZernikeReconstruct 245
Zero 427
ZTiltPSD 246

Bibliography

[1] P. J. Berger, J. Herrmann, R. J. Sasiela, R. Stock, and S. A. Tirrell, "Description and Assesment of the Airborne Laser Engagement (ABLE) Code," Technical Report VVA-LL-003, Rev A, Appendix A, 10 September 2002. FOR OFFICIAL USE ONLY.

[2] P. K. Seidelmann, *Explanatory supplement to the astronomical almanac*. University Science Books, 1992.

[3] K. M. Borkowski, "Accurate algorithms to transform geocentric to geodetic coordinates," *Bull. Geod.*, vol. 63, no. 1, pp. 50–56, 1989.

[4] T. Vincenty, "Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations," *Survey Review*, vol. 23, no. 176, pp. 88–93, 1975.

[5] D. H. Maling, *Coordinate Systems and Map Projections*. Pergamon Press, 2nd edition ed., 1992.

[6] D. E. Gray, ed., *American Institute of Physics Handbook*. McGraw-Hill, 3rd ed., 1972.

[7] L. C. Andrews and R. L. Phillips, *Laser Beam Propagation through Random Media*. Bellingham, Wash.: SPIE Press, 2nd ed., 2005.

[8] L. C. Andrews, *Special Functions of Mathematics for Engineers*. Oxford: Oxford University Press, 2nd ed., 1998.

[9] D. A. Vallado, *Fundamentals of Astrodynamics and Applications*. Space Technology Series, McGraw-Hill Primis Custom Publishing, 1997.

[10] F. R. Hoots and R. L. Roehrich, "Spacetrack report no. 3: Models for propagation of NORAD element sets," tech. rep., Aerospace Defense Center, Peterson Air Force Base, 1980.

[11] Nautical Almanac Office (U.S.) and U.S. Nautical Almanac Office, *Astronomical Almanac for the Year 2013 and Its Companion, the Astronomical Almanac Online*. ASTRONOMICAL ALMANAC FOR THE YEAR, Bernan Assoc, 2012.

[12] D. Kirk, *Graphic Gems III*. San Diego, CA: Academic Press Professional, Inc, 1992.

[13] P. J. Berger, J. Herrmann, R. J. Sasiela, R. Stock, and S. A. Tirrell, "Description and Assessment of the Airborne Laser Engagement (ABLE) Code," Technical Report VVA-LL-003, Rev A, 10 September 2002. FOR OFFICIAL USE ONLY.

[14] M. C. Roggemann and B. M. Welsh, *Imaging Through Turbulence*. The CRC Press Laser and Optical Science and Technology Series, Boca Raton: CRC Press, 1996.

[15] D. L. Fried, "Optical resolution through a randomly inhomogeneous medium for very long and very short exposures," *J. Opt. Soc. Am.*, vol. 56, pp. 1372–1379, 1966.

[16] G. A. e. a. Tyler, "Adaptive optics: Theory and applications," Technical Report AFRL-DE-PS-TR-1998-1054, Air Force Research Laboratory, Directed Energy Directorate, 1999.

[17] K. D. Mielenz, "Algorithms for Fresnel diffraction at rectangular and circular apertures," *J. Res. Natl. Inst. Stand. Technol.*, vol. 103, no. 5, pp. 497–509, 1998.

[18] L. C. Bradley and J. Herrmann, "Phase compensation for thermal blooming," *Appl. Opt.*, vol. 13, no. 2, pp. 331–334, 1974.

[19] A. M. Ngwele, "Scaling law modeling of thermal blooming in wave optics," MZA Technical Report NGIT ABL A&AS, November 2010.

[20] D. P. Greenwood, "Bandwidth specification for adaptive optics systems," *J. Opt. Soc. Am.*, vol. 67, pp. 390–393, 1979.

[21] R. J. Noll, "Zernike polynomials and atmospheric turbulence," *J. Opt. Soc. Am.*, vol. 66, pp. 207–211, 1976.

[22] J. Y. Wang and J. K. Markey, "Modal compensation of atmospheric turbulence phase distortion," *J. Opt. Soc. Am.*, vol. 68, pp. 78–87, 1978.

[23] R. J. Sasiela, *Electromagnetic Wave Propagation in Turbulence*. Berlin: Springer-Verlag, 1994.

[24] J. W. Goodman, *Introduction to Fourier Optics*. McGraw-Hill Electrical and Computer Engineering Series, New York: McGraw-Hill, 2nd ed., 1996.

[25] D. L. Fried, "Limiting resolution looking down through the atmosphere," *J. Opt. Soc. Am.*, vol. 56, pp. 1380–1384, 1966.

[26] S. F. Clifford, *The Classical Theory of Wave Propagation in a Turbulent Medium*, vol. 25 of *Topics in Applied Physics (Laser Beam Propagation in the Atmosphere)*, pp. 9–43. New York: Springer-Verlag, 1978.

[27] L. C. Andrews, R. L. Phillips, and C. Y. Hopen, "Scintillation model for a satellite communication link at large zenith angles," vol. 39, pp. 3272–3280, 12 2000.

[28] L. Andrews, R. Phillips, and C. Hopen, *Laser Beam Scintillation with Applications*. SPIE Press monograph, SPIE Press, 2001.

[29] D. L. Fried, "Limiting resolution looking down through the atmosphere," *J. Opt. Soc. Am.*, vol. 56, no. 10, pp. 1380–1384, 1966.

[30] D. L. Fried, "Anisoplanatism in adaptive optics," *J. Opt. Soc. Am.*, vol. 72, no. 1, pp. 52–61, 1982.

[31] T. R. Berkopec, "Thermal Blooming in WaveTrain," Technical Report AFRL-DE-PS-TR-2004-1099, MZA Associates Corporation, April 2003.

[32] M. R. Whiteley, V. S. R. Gudimetla, R. L. Beauchamp, T. L. Klein, and J. T. Riley, "Simulation and Analysis of Directed Energy Beam Control: Advanced Airborne Laser, Relay Mirror, and Tactical Laser," AFRL/DE Technical Report AFRL-DE-PS-TR-2005-1101, ATK Mission Research Corporation, 29 July 2005. Contract Number F29601-02-D-0256.

[33] E. P. Magee, "Derivations of Constants used in `ATMTools.tbx`," Technical Report, July 2006.

[34] G. A. Tyler, "Bandwidth considerations for tracking through turbulence," *J. Opt. Soc. Am. A*, vol. 11, no. 1, p. 358, 1994.

[35] R. R. Beland, *Propagation Through Atmospheric Optical Turbulence*, vol. 2, Chap. 2 of *The Infrared and Electro-Optical Systems Handbook*. Bellingham, WA: SPIE, 1993.

[36] R. F. Walter and S. A. Mani, "Laser Engineering and Technology Support (LETS) Advanced Tactical Laser (ATL) Power-Aperture Trade Studies," Technical Report AFRL-DE-TR-2003-1018, Schafer Corporation, March 2004.

[37] "U.S. Standard Atmosphere, 1976," Tech. Rep. ADA-035728, National Oceanic and Atmospheric Administration, 1976.

[38] P. Nicolas, "Atmospheric occultation of optical intersatellite links: coherence loss and related parameters," *Appl. Opt.*, vol. 48, no. 12, 2009.

[39] G. M. Anderson, *Development of a Standard Maritime Cn2 Profile Using Satellite Measurements*. M.s. thesis, Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, 2015.

[40] B. Edlen, "The dispersion of standard air," *J. Opt. Soc. Am.*, vol. 43, no. 5, pp. 339–344, 1953.

[41] J. C. Owens, "Optical refractive index of air: Dependence on pressure, temperature and composition," *Appl. Opt.*, vol. 6, no. 1, pp. 51–58, 1967.

[42] B. Edlen, "The refractive index of air," *Metrologica*, vol. 2, no. 2, pp. 71–80, 1966.

[43] R. K. Tyson, *Principles of Adaptive Optics*. Boston: Academic Press, 2nd ed., 1998.

[44] L. Bradford and R. Holmes, "Maui cn2 profiles," Technical Memo, AFRL, Feb 21 2003.

[45] W. Wiscombe, "Mie scattering calculations–advances in technique and fast, vector-speed computer codes," Ncar Tech Note TN-140+STR, National Center For Atmospheric Research, Boulder, Colorado, June 1979.

[46] W. Wiscombe, "Improved mie scattering algorithms," vol. 19, pp. 1505–9, 05 1980.

[47] H. C. Van De Hulst, *Light Scattering by Small Particles*. New York: Dover Press, 1957, 1982.

[48] M. Kerker, *The Scattering of Light and Other Electromagnetic Radiation*, vol. 16 of *Physical Chemistry: A Series of Monographs*. Academic Press, Oct 22 2013.

[49] P. A. Frederickson, K. L. Davidson, C. R. Zeisse, and C. S. Bendall, "Estimating the refractive index structure parameter (cn2) over the ocean using bulk methods," *J. Appl. Meteorol.*, vol. 39, pp. 1770–1783, 2000.

[50] B. et al, "Modeling and measurements of near-ground atmospheric optical turbulence according to weather for middle east environments," *Proc. SPIE*, vol. 5612, pp. 350–361, 2004.

[51] F. V. Hansen, "Surface roughness lengths," Technical Report ARL-TR-61, ARL, August 1993.

[52] D. H. Tofsted, "Modeling turbulence generation in the atmospheric surface and boundary layers," Technical Report ARL-TR-7503, ARL, October 2015.

[53] K. D. Mielenz, "Computation of Fresnel integrals. II," *J. Res. Natl. Inst. Stand. Technol.*, vol. 4, no. 105, pp. 589–590, 2000.

[54] Y. L. Luke, *The Special Functions and Their Approximations*. Mathematics in science and engineering, v. 53, New York,: Academic Press, 1969.

[55] D. T. Kyrazis, "Optical degradation by turbulent free shear layers," *Proc. SPIE*, vol. 2005, pp. 170–181, 1993.

[56] S. Coy, "Choosing mesh spacings and mesh dimensions for wave optics simulation," 2005.

[57] C. B. Moler, *Numerical Computing with Matlab*. Philadelphia, PA: Society for the Industrial and Applied Mathematics.