



**US Army Corps
of Engineers®**
Engineer Research and
Development Center

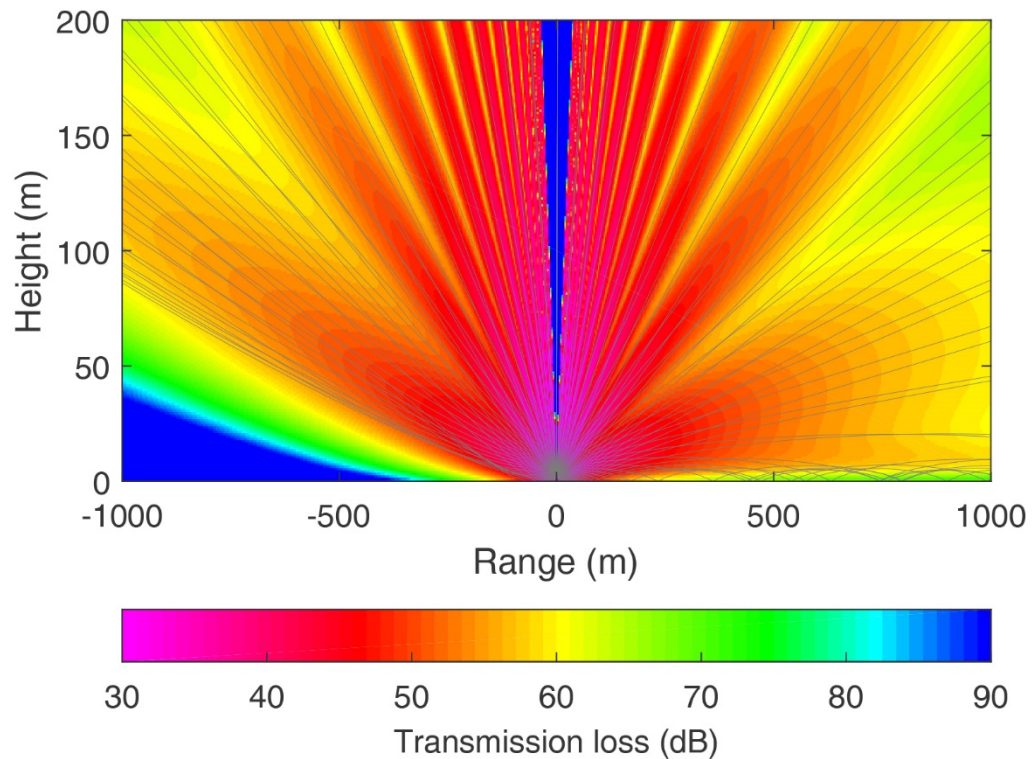


ERDC 6.2/6.3 Military Engineering (ME) RAFTER

Using EASEE's Acoustical Calculations in MATLAB

D. Keith Wilson, Ross E. Alter, Katrina M. Burch, and
Michelle E. Swearingen

January 2019



The U.S. Army Engineer Research and Development Center (ERDC) solves the nation's toughest engineering and environmental challenges. ERDC develops innovative solutions in civil and military engineering, geospatial sciences, water resources, and environmental sciences for the Army, the Department of Defense, civilian agencies, and our nation's public good. Find out more at www.erdclibrary.usace.army.mil.

To search for other technical reports published by ERDC, visit the ERDC online library at <http://acwc.sdp.sirsi.net/client/default>.

Using EASEE's Acoustical Calculations in MATLAB

D. Keith Wilson, Ross E. Alter, and Katrina M. Burch

*U.S. Army Engineer Research and Development Center (ERDC)
Cold Regions Research and Engineering Laboratory (CRREL)
72 Lyme Road
Hanover, NH 03755-1290*

Michelle E. Swearingen

*U.S. Army Engineer Research and Development Center (ERDC)
Construction Engineering Research Laboratory (CERL)
2902 Newmark Dr.
Champaign, IL 61822*

Final Report

Approved for public release; distribution is unlimited.

Prepared for Headquarters, U.S. Army Corps of Engineers
Washington, DC 20314-1000

Under ERDC 6.2/6.3 Military Engineering (ME), Remote Assessment of
Infrastructure for Ensured Maneuver (RAFTER), funded by 62784/T40/46,
“Propagation Effects”

Abstract

EASEE (Environmental Awareness for Sensor and Emitter Employment) is a Java-based software framework for modeling the impacts of the weather and terrain on signal propagation and sensor performance. EASEE includes extensive capabilities for representing the environment (atmosphere, land cover, terrain elevation, and soil properties), along with many different models related to acoustic, optical, radio frequency, and seismic signals. This report describes how to run EASEE from MATLAB, which is a popular software package for performing numerical calculations and displaying graphics. For this purpose, a simple installation configuration and MATLAB script were devised to set up and initialize EASEE. The focus of the report is on using EASEE for acoustic propagation calculations, which is its most mature signal modality. The report describes two general approaches to performing acoustical calculations in EASEE: one that involves using EASEE for its environmental representation only and then running the acoustic propagation calculation using a MATLAB toolbox and a second that uses EASEE for both its environmental layer and the acoustical calculation. Overall, this report shows that the MATLAB user interface provides convenient access to EASEE's powerful signal modeling capabilities.

DISCLAIMER: The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. All product names and trademarks cited are the property of their respective owners. The findings of this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

DESTROY THIS REPORT WHEN NO LONGER NEEDED. DO NOT RETURN IT TO THE ORIGINATOR.

Contents

Abstract	ii
Figures and Tables.....	v
Preface.....	vi
Acronyms and Abbreviations.....	vii
1 Introduction.....	1
1.1 Background	1
1.1.1 What is EASEE?.....	1
1.1.2 EASEE software design and capabilities	2
1.2 Objective.....	3
1.3 Approach	4
2 Preliminaries.....	5
2.1 Installing the EASEE files.....	5
2.2 Changing the MATLAB JVM to Java 8.....	5
3 Running EASEE.....	7
3.1 Getting started.....	7
3.2 Verifying the installation	8
3.3 Simple example	8
3.4 Some useful background	10
3.4.1 A very brief introduction to Java	10
3.4.2 Running Java from MATLAB	11
3.4.3 Importing packages and classes.....	12
3.4.4 Conversion of data types	14
3.4.5 The MATLAB disp command.....	16
3.4.6 Java enumerations and inner classes	16
4 Geographic Coordinates, Grids, and Elevation Maps	21
4.1 Geographic coordinates	21
4.2 Geographic grids.....	23
4.3 Digital elevation, surface, land cover, and soil grids	24
5 Environmental Representations	27
5.1 Atmospheric constants and conversions	29
5.2 Environmental components	30
5.2.1 Humid air representation.....	31
5.2.2 Solid-earth representation.....	32
5.2.3 Land cover representation	34
5.2.4 Soil representation.....	36
5.2.5 Snow representation.....	37
5.2.6 Atmospheric surface layer.....	38

5.2.7	Atmospheric vertical profiles.....	41
5.2.8	Clouds	45
5.2.9	Seismic vertical profiles.....	46
5.3	Environmental representations.....	48
5.3.1	Homogeneous environment	48
5.3.2	Vertical profile environment	52
5.4	Loading WRF Data	55
5.4.1	Single WRF Output File	55
5.4.2	Multiple WRF output files.....	60
6	Acoustic Propagation Calculations.....	66
6.1	Standard acoustic frequencies.....	66
6.2	AcousticMedium class.....	67
6.3	Calculation grids	70
6.4	Impedance-plane model.....	71
6.4.1	Flat ground	71
6.4.2	Uneven ground	75
6.5	Parabolic equation methods	78
6.6	BNOISE	83
6.7	Nord2000.....	83
6.8	MATLAB acoustic propagation interface.....	83
7	Full Example Script for Testing EASEE in MATLAB	88
8	Conclusion.....	93
	References	94

Figures and Tables

Figures

1	EASEE framework	3
2	Output from the methodview('HumidAir') command in MATLAB.....	14
3	Hierarchy of EASEE's environmental classes. Phrases in <i>bold</i> are the main environmental classes within EASEE, and <i>italicized</i> phrases are subclasses of the foremost environmental class— <i>EnvironScenario</i> . All other listings are fields (e.g., friction velocity is a field within <i>AtmosSfcLayer</i> , which is a field within <i>AtmosOneDim</i> , which is a field within <i>EnvironVertProf</i>). All phrases with only capitalized terms (e.g., <i>EnvironVertProf</i>) are instances of classes. <i>Blue</i> font color indicates recent additions.....	28
4	Vertical profiles of temperature, specific humidity, and wind speed as created by specification of atmospheric surface layer properties for a moderately windy, unstable case	43
5	Vertical profiles of air temperature (<i>blue</i>), the eastward wind component (<i>red</i>), and the northward wind component (<i>orange</i>) derived from a WRF test file in the EASEE repository	60
6	Vertical profiles of air temperature derived from three pseudoensemble test WRF files in the EASEE repository.....	65
7	Attenuation coefficient for air at 20°C, 40% relative humidity, and sea-level pressure	69
8	Transmission loss (TL) for propagation at a frequency of 100 Hz over several different ground surfaces.....	75
9	Terrain elevations (digital elevation model, or DEM) used for the transmission loss calculation shown in Fig. 10. The coordinate axes are the easting and northing relative to the southwest corner of the domain	77
10	Transmission loss (TL) calculation for a source in hilly terrain. The frequency is 100 Hz. The source is positioned at an easting of 7 km and a northing of 5 km. The DEM for the calculation is shown in Fig. 9.....	78
11	Comparison of transmission loss (TL) calculations by several different codes	82
12	Transmission loss resulting from a wide-angle parabolic equation calculation using the MATLAB interface	87
13	Same as Fig. 12 but using atmospheric output from the WRF weather model.....	92
14	Difference in transmission loss (TL) between Fig. 13 and Fig. 12.....	92

Preface

This study was conducted for the Assistant Secretary of the Army for Acquisition, Logistics, and Technology (ASA[ALT]) under the U.S. Army Engineer Research and Development Center (ERDC) 6.2/6.3 Military Engineering (ME), Remote Assessment of Infrastructure for Ensured Maneuver (RAFTER) program, funded by 62784/T40/46, under the “Propagation Effects” pillar. The technical monitor was Ms. Danielle Whitlow, Program Manager for RAFTER.

The work was performed by the Signature Physics Branch (CEERD-RRD) of the Research and Engineering Division (CEERD-RR), ERDC Cold Regions Research and Engineering Laboratory (CRREL), and the Ecological Processes Branch (CEERD-CNN) of the Installations Division (CEERD-CN), ERDC Construction Engineering Research Laboratory (CERL). At the time of publication, Dr. Andrew Niccolai was Chief, CEERD-RRD; Mr. J. D. Horne was Chief, CEERD-RR; Dr. Chris Rewerts was Chief, CEERD-CNN; and Ms. Michelle Hanson was Chief, CEERD-CN. The Deputy Director of ERDC-CRREL was Mr. David B. Ringelberg, and the Director was Dr. Joseph L. Corriveau. The Deputy Director of ERDC-CERL was Dr. Kirankumar V. Topudurti, and the Director was Dr. Lance D. Hansen.

COL Ivan P. Beckman was Commander of ERDC, and Dr. David W. Pittman was the Director.

Acronyms and Abbreviations

1-D	One-Dimensional
2-D	Two-Dimensional
3-D	Three-Dimensional
API	Application Programming Interface
ARW	Advanced Research WRF
ASA(ALT)	Assistant Secretary of the Army (Acquisition, Logistics, and Technology)
ASL	Atmospheric Surface Layer
BNOISE	Blast Noise
CERL	Construction Engineering Research Laboratory
CNPE	Crank-Nicholson Parabolic Equation
CRREL	U.S. Army Cold Regions Research and Engineering Laboratory
DEM	Digital Elevation Model
DSM	Digital Surface Model
EASEE	Environmental Awareness for Sensor and Emitter Employment
ERDC	U.S. Army Engineer Research and Development Center
FASST	Fast All-Season Strength
FFP	Fast-Field Program
GFPE	Green's Function Parabolic Equation
IR	Infrared
.jar	Java Archive file
JDK	Java SE Development Kit
JRE	Java SE Runtime Environment

JVM	Java Virtual Machine
KNEE	KNEE is Not EASEE in Its Entirety
ME	Military Engineering
MGRS	Military Grid Reference System
MOST	Monin-Obukhov Similarity Theory
NASA	National Aeronautics and Space Administration
NCAR	National Center for Atmospheric Research
NetCDF	Network Common Data Format
NLCD	National Land Cover Dataset
P-	Compressional
PE	Parabolic Equation
RAFTER	Remote Assessment of Infrastructure for Ensured Maneuver
RF	Radio Frequency
S-	Shear
SPEBE	Sensor Performance Evaluator for Battlefield Environments
TL	Transmission Loss
UTC	Coordinated Universal Time
UTM	Universal Transverse Mercator
WRF	Weather Research and Forecasting model

1 Introduction

1.1 Background

1.1.1 What is EASEE?

EASEE (Environmental Awareness for Sensor and Emitter Employment) provides a versatile software framework for modeling terrain and weather impacts on signal propagation and the performance of battlefield sensors. Two previous reports (Wilson et al. 2009; Wilson and Yamamoto 2014) describe the overall design and many technical aspects of EASEE. As Wilson and Yamamoto (2014) explain,

The performance and utility of battlefield and homeland security sensors depends on many complex environmental and mission-related factors. This is generally true whether the sensors are ground-based or airborne; whether the sensors are acoustic, seismic, optical, infrared (IR), radio frequency (RF), or magnetic; and whether the observable features of signal emitters originate from vehicles, humans, or electronic equipment. Realistic modeling and simulation of environmental factors can improve the effectiveness of mission planning and can further the development of more effective sensor system designs and doctrine for their usage.

The same report also summarizes the motivation underlying EASEE's software design and the history of its development:

The primary design goal in developing EASEE was to create a highly reusable software framework, which would provide realistic, physics-based simulations of terrain and weather impacts on all types of battlefield signals and sensors. EASEE is a successor to the SPEBE (Sensor Performance Evaluator for Battlespace Environments) software (Wilson et al. 2002), which had become widely used but accommodated only acoustics and seismics. SPEBE was also written in MATLAB, which limited options for interfacing it with other simulations and mission planning tools. The EASEE project began as an ERDC [U.S. Army Engineer Research and Development Center] applied research work package in 2006. It involved a complete adaptation of the capabilities of SPEBE into the Java program-

ming language, to make the code more reusable, and a new, object-oriented modeling paradigm that could accommodate signal modalities in addition to acoustic and seismic.

Thus EASEE, a Java-based code, was the successor to SPEBE, a MATLAB-based code. The motivation for making this transition, as described above, remains valid. Largely because EASEE is written in Java, it has been successfully integrated into many different user interfaces and deployed as a web service. However, there are situations, especially in research applications, where it is nonetheless desirable to access EASEE's modeling capabilities from MATLAB. This turns out to be very feasible, since Java can be directly called from MATLAB. From a programming perspective, EASEE appears to function in essentially the same manner as native MATLAB code. This report will describe how to use EASEE from MATLAB. First, however, it is useful to outline some aspects of EASEE's design and capabilities.

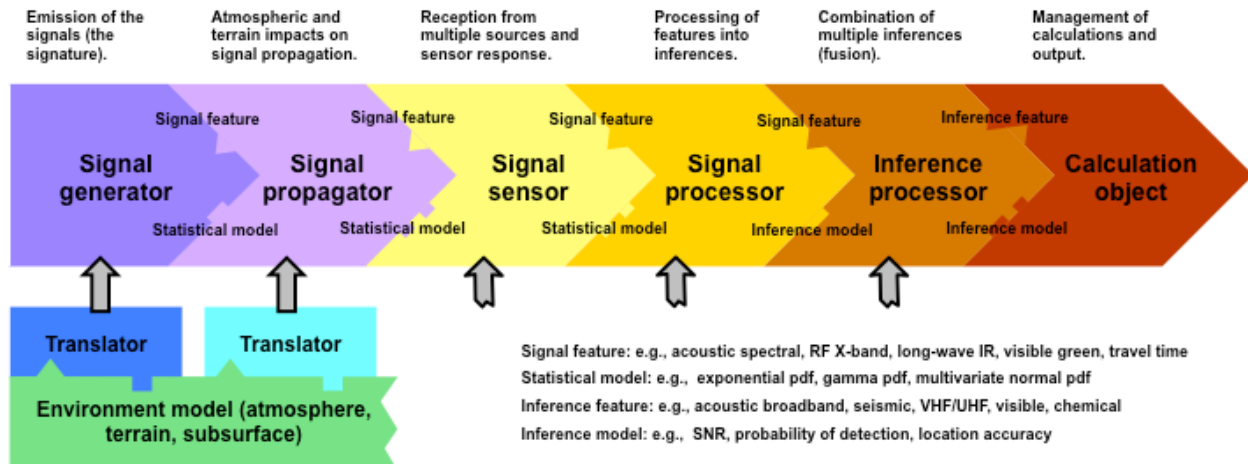
1.1.2 EASEE software design and capabilities

EASEE consists of two main components—EASEELib, which is the militarily sensitive and limited distribution parts of EASEE, and KNEE, which contains the nonsensitive parts. KNEE is short for “KNEE is Not EASEE in its Entirety.”* EASEELib and KNEE are compiled as separate .jar (Java archive) files. KNEE can run by itself, but EASEELib requires KNEE to run.

Figure 1 shows the basic software design of EASEE. The design involves a sequence of interlinked modeling stages, from generation of the signal to propagation through the environment, sensing of the signal, and processing. Each step in the sequence is called a *framework element*. An advanced Java technique called *generics* enforces compatibility between framework elements and data passed between them. The different models thus always fit together as designed.

* The KNEE acronym is an homage to the acronym for the well known GNU software project, which means “GNU is Not Unix.”

Figure 1. EASEE framework.



EASEE presently incorporates modeling capabilities related to many different signal modalities, including acoustic, seismic, RF, visible, IR, and chemical/biological. Additionally, EASEE includes its own detailed representations of the environment, by which we mean primarily the atmosphere, land cover, terrain elevation, and subsurface properties. Each model implemented as a framework element has its own class for translating environmental representations to parameters needed by the model. EASEE's environmental representation is an example of a *data abstraction layer*. (For a description of data abstraction layers, see, for example, Rouse 2014.) The purpose of this layer is to isolate the modeling capabilities from the environmental representation. Suppose we wish to interface a new environmental data resource to EASEE (e.g., a new atmospheric forecast model or a new land cover scheme). We thus would need to write new code that translates the data resource into EASEE's native environmental representation. But this only needs to happen once—then the new data resource will become available to all models in EASEE without modifying any code associated with the individual models or their parameter translation classes.

1.2 Objective

The primary purpose of this report is to provide an easy-to-use, step-by-step manual for those interested in using EASEE to model outdoor sound propagation. Specifically, our report shows how to run EASEE within MATLAB, a programming platform that is widely used among potential users of EASEE.

1.3 Approach

EASEE is too extensive to practically describe here all of its components and how they can be called from MATLAB. Rather, this report focuses on some particular aspects of setting up the environmental representations and running acoustic propagation models. The report also focuses on *dynamic* methods (those involving object instantiation) in EASEE, as opposed to *static* methods. In general, much of the functionality described here (such as humidity conversions, calculation of absorption coefficients, and calculation of solid material properties) is supported by static methods also. We do not attempt here to describe all the methods (static or dynamic) that are available in EASEE. Programmers may find additional useful information by examining the constructors and other methods in the original Java code (which are generally well documented) or by reading the EASEELib and KNEE Javadoc (computer-generated HTML documentation). The test packages in EASEELib and KNEE are also very helpful in providing examples of how to call the code.

This report is intended primarily for acousticians and software developers who wish to use EASEE to model outdoor sound propagation. However, it may also be of interest to developers desiring an introduction to the functionality of EASEE, complete with useful examples and test cases.

In this report, we first discuss some preliminaries about setting up MATLAB and installing EASEE. Next, we discuss the various classes in EASEE for representing the environment, by which is meant the atmosphere, surface state, and subsurface (geology). Lastly, we discuss how to run propagation calculations in EASEE.

2 Preliminaries

In this section, we describe the preliminary steps that the user must perform to run EASEE from MATLAB, beginning with how to configure the Java Virtual Machine (JVM) and install the EASEE .jar files.

2.1 Installing the EASEE files

The files needed to run EASEE from MATLAB are distributed in the file `easee-repo.zip`, which is available from the authors of this report. Place this file in a convenient location and then unzip the contents. You should see a few files and several directories, including `air`, `CNPE`, and `EASEEInterface`. If you navigate to the `EASEEInterface` directory, you will find a folder called `easee-repo`, and a number of other files, the most essential being `StartEASEE.m`, `SetPaths.m`, `RunCalc.m`, `PlotCalc.m`, `createMATLABShortcut.m`, and `Rainbow.mat`. In particular, the file `StartEASEE.m` automatically performs a number of functions related to configuring paths, thus simplifying the process of running EASEE from MATLAB. Also, the file `TestEASEE.m` contains all of the executable MATLAB code in this technical report (except for section 7, which is included in `CalcExampWithWRF.m`) and can be run sequentially (section by section) or all at once so users do not have to manually copy all of the code into MATLAB for testing purposes.

The folder `easee-repo` contains a number of .jar and executable files. These include the `KNEE` and `EASEELib` .jar files, as well as a number of other .jar files that are needed by EASEE.

The next subsection describes the final prerequisite to launching EASEE.

2.2 Changing the MATLAB JVM to Java 8

EASEE is compiled using Java version 8. As such, running EASEE from MATLAB requires configuring MATLAB to use a Java 8 JVM, rather than Java 7, which is the default JVM distributed with MATLAB as of version R2017a. When MATLAB is originally installed, its JVM is separate from the one by the operating system (Windows, MacOS, or Unix). To determine which Java version is installed on your operating system, type the following into a command prompt:

```
java -version
```

The output should look like `java version "1.8.0..."` if Java 8 is installed.

Users who do not have Java 8 running on their operating system will need to download and install either the Java SE Runtime Environment (JRE) or, if they wish to develop EASEE, the Java SE Development Kit (JDK, which contains the JRE) from <http://www.oracle.com/technetwork/java/javase/downloads> (Oracle, n.d.).

Once a Java 8 JVM has been installed on the target computer, it is still necessary to point MATLAB to this JVM. The procedure depends on which operating system you are using.

Instructions for Windows users: Create a `MATLAB_JAVA` environmental variable and set it to the JRE folder, as described at MathWorks (2018a). You will need to restart MATLAB.

Instructions for MacOS users: Several options are provided at MathWorks (2018b). The one we found to be most satisfactory is running the script `createMATLABShortcut`. For convenience, this script is included in the installation distribution. To use it, navigate to the `EASEEInterface` folder, and then type

```
createMATLABShortcut('/Library/Java/JavaVirtualMachines/  
jdk1.8.0_111.jdk/Contents/Home/jre')
```

(Substitute the location for the desired JRE in the quotes.) This will create a new desktop shortcut that will start MATLAB but with the specified JRE. We recommend that you retain this shortcut as well as the MATLAB icon on the Dock, which will start with the MATLAB default JVM as before.

All users: Now, if you restart MATLAB and type `ver` at the MATLAB prompt, you should see the desired version of the JVM listed. Note that you will need to repeat the preceding instructions (Windows or MacOS) if you install a new JVM.

Having installed EASEE, you are now ready to run it, as described in the next section.

3 Running EASEE

3.1 Getting started

The following three simple steps set the stage for running EASEE from MATLAB:

1. Launch MATLAB (e.g., by double-clicking on the desktop icon).
2. Navigate to the `EASEEInterface` folder (where `StartEASEE.m` resides), using either the `cd` command at the MATLAB prompt or the “Browse for Folder” button at the top of the command window.
3. At the MATLAB prompt, type

```
StartEASEE('easee-repo')
```

where `easee-repo` is the folder containing the EASEE/KNEE jar files. Or, if the `easee-repo` folder is a subfolder within the current one (as is the case with the normal file distribution), just type the following:

```
StartEASEE
```

The `StartEASEE` function will add these files to MATLAB’s Java class path (which you can verify by typing `javaclasspath` at the MATLAB prompt) and will attempt to call EASEE for the first time. If you see the following error,

```
Undefined variable "DirectoryUtils" or class  
    "DirectoryUtils.setRootDirectory".
```

```
Error in StartEASEE (line 16)
```

```
DirectoryUtils.setRootDirectory(repoPath);
```

then MATLAB could not find the .jar files. The likely cause is that the name of the folder was incorrectly specified to `StartEASEE`. The same error will be issued when a version of Java other than Java 8 is running.

3.2 Verifying the installation

Next, type the following six lines at the MATLAB prompt. If they execute without error, then the EASEE and KNEE files have been properly installed:

```
import mil.army.usace.knee.acoustic.*;
test1=ImpedancePlaneParamsHomo;
disp(test1)
import mil.army.usace.easee.acoustic.*;
test2=ParabolicEqParamsHomo;
disp(test2)
```

As output, you should see

```
terrain diffraction included, por=0.38, tort=1.27, vort=1.758e-04,
...

terrain diffraction included, narrow angle, range res = 0.1,
height res = 0.1
```

You are now running EASEE!

3.3 Simple example

Let us consider a simple, but useful, example in which we calculate the sound pressure above a rigid plane. When entering the following code, it is important to remember that both MATLAB and Java are case sensitive.

First, we import the Java packages needed for the calculation by entering the following at the MATLAB prompt:

```
import mil.army.usace.knee.acoustic.*;
import mil.army.usace.knee.envIRON.*;
import mil.army.usace.knee.grids.*;
```

Next, we will set the frequency, source height, receiver height, and range (horizontal distance):

```
freq = 100.0; % frequency
srcHgt = 5.0; % source height
zmesh = 1.0; % receiver height
```

```
rmesh = 0.0:100.0:1000.0; % horizontal ranges for output
```

The following sequence of commands sets up the parameters for the impedance-plane model:

```
refAir = ImpedancePlaneModelTypes.RIGID.getRefAir();  
ground = ImpedancePlaneModelTypes.RIGID.getGround();  
paramsImp = ImpedancePlaneParamsHomo();  
paramsImp.setMedia(refAir, ground);
```

Next, we create an impedance-plane calculation model with the parameters specified above:

```
impp = ImpedancePlaneModel(paramsImp);
```

Lastly, we set up the grid for the calculation and run the model:

```
tgV = TransmitGridVert(srcHgt, zmesh, rmesh);  
calcVals = impp.calcTransGridStruct(freq, tgV);
```

Now we can display the results of the calculation:

```
disp(10*log10(calcVals))
```

which yields transmission loss relative to the level that would be observed at 1 m in free space. Note that EASEE calculates the mean-square sound pressure; so in the preceding, we converted to decibels by multiplying by 10 rather than by 20. The following values will be displayed:

```
-17.4677  
-34.1577  
-40.2729  
-43.9186  
-46.5455  
-48.6131  
-50.3266  
-51.7957  
-53.0857  
-54.2391  
-55.2846
```

The complete code for this example can be found in the script `CalcExampleImp.m`, which is included in the installation distribution.

3.4 Some useful background

This section provides some background information on Java and MATLAB that is useful for understanding how to employ EASEE.

3.4.1 A very brief introduction to Java

Here we provide a *very* brief introduction to Java for readers of this report who might be familiar with MATLAB but not with Java. The main purpose is to define Java terminology, such as *objects*, *classes*, *methods*, and *constructors*. For readers interested in a full tutorial on Java, we would recommend the book *Head First Java* (Sierra and Bates 2005) for those who are learning Java as their first object-oriented language; *Thinking in Java* (Eckel 2006) for those who have already learned an object-oriented language; and *Java All-in-One Desk Reference for Dummies* (Lowe and Burd 2007) for a more complete reference. There are also many excellent, free, online resources, such as the IBM Introduction to Java programming course found at Perry (2010).

Java is a general-purpose, object-oriented programming language and was designed to be simple and reusable on different computer architectures. As the name would suggest, the focus of object-oriented programming is on the concept of *objects* as opposed to *procedures* or *functions*, which was the dominant programming paradigm in popular programming languages such as BASIC and FORTRAN, which preceded Java.

Objects include *attributes* (data) and *methods* (procedures) that manipulate the attributes of the object. Objects are typically *constructed* (instantiated) and then are dynamically modified during program execution. Methods that change the properties of an object are commonly called *setter* methods; methods that retrieve existing properties are *getter* methods.

Java also supports the procedural or functional programming through *static*, as opposed to *instance*, methods. The static methods do not require construction of an object and can be thought of in the conventional sense of receiving an output for a set of inputs. For some programming problems, static methods arguably provide a more natural solution than object-oriented programming.

Java objects are described by *classes*. The class defines *fields*, which represent the attributes of the object. The fields may be simple data types, such as integers, floating point values, and Strings, or they may be other objects. The class also defines the instance methods for the object. Static methods may also be defined, but these exist independently of the construction of objects of the class.

Typically, classes are defined in a hierarchy, with parent classes and subclasses, or *extensions*. For example, a class for animals might be defined, a subclass of which could be cats, which might have subclasses for lions, tigers, and housecats.

3.4.2 Running Java from MATLAB

Since Java can be directly called from MATLAB, the EASEE Java code is called very similarly to native MATLAB functions. The main differences are that the desired Java packages must be imported and that many of the operations produce or operate on Java objects, rather than on MATLAB native data types. The Java objects generally cannot be directly manipulated (e.g., retrieving, setting, and performing operations on their properties) within MATLAB; the objects must be manipulated by calling the EASEE Java methods from MATLAB.

Java code, when called from within MATLAB (as opposed to when it is called directly from Java), appears essentially the same as the original Java, with a few notable differences:

1. *Data types:* The data types are not explicitly declared (since MATLAB, unlike Java, is not a type-controlled language). Hence many data conversions are performed automatically according to the MATLAB application programming interface (API). This topic will be discussed later in its own subsection.
2. *Object construction:* The `new` keyword in Java is omitted in MATLAB when calling a constructor method. Alternatively, one can construct a Java object with MATLAB's `javaMethod` function, but this is only necessary in some special cases as described in the MATLAB documentation.
3. *Invoking Java methods:* Just as in Java, static methods are called by specifying the name of the class, followed by a period and then the name of the method. Instance methods are called the same way, with the name of a

valid object replacing the class name. Alternatively, the `javaMethod` function can be used, but this is usually unnecessary. If there are no arguments to a method, MATLAB allows the empty parentheses to be omitted.

3.4.3 Importing packages and classes

The MATLAB `import` command, like the Java `import` statement, provides a way to abbreviate the names of packages and classes. For example, KNEE's `geo` package has a class called `GeoCoord`, which represents geographic coordinates. To construct a geographic coordinate and assign the result to the variable `gc`, one could either type

```
gc = mil.army.usace.knee.geo.GeoCoord(33.0, -62.2);
```

where the two numbers represent the latitude and longitude in degrees, respectively, or

```
import mil.army.usace.knee.geo.GeoCoord;  
gc = GeoCoord(33.0, -62.2);
```

A wild card (asterisk) can also be used to import all classes in a Java package. Thus the following will also work:

```
import mil.army.usace.knee.geo.*;  
gc = GeoCoord(33.0, -62.2);
```

MATLAB does not recommend this last version of the `import` command due to the potential for creating conflicts in the name space. For example, if one of the imported Java classes implied by the wildcard is named “close”, it could conflict with the MATLAB `close` function.

However, it is important to note that `StartEASEE` must be executed *at the MATLAB command prompt* before MATLAB is able to import individual classes in EASEE. Otherwise, if `StartEASEE` is included in a script that also contains an `import` command for an individual class (e.g., `import mil.army.usace.knee.geo.GeoCoord`), then MATLAB will produce an error. Therefore, to avoid confusion (and for convenience), we will use the wildcard version of the `import` command in the following examples.

After a package has been imported, one can list the methods within a Java class that is located within that package (such as `HumidAir` within `mil.army.usace.knee.envIRON`) by using `methods('class')`, for example

```
methods('HumidAir')
```

which displays

Methods for class `HumidAir`:

<code>HumidAir</code>	<code>getDewPoint</code>	<code>getSpecHumDef</code>	<code>notify</code>
<code>thermCond</code>	<code>convHum</code>	<code>getMixRat</code>	<code>getTemp</code>
<code>notifyAll</code>	<code>toString</code>	<code>convRelHumToSpecHum</code>	<code>getMolHum</code>
<code>getTempCelsius</code>	<code>prandtl</code>	<code>viscosity</code>	<code>convSpecHumToRelHum</code>
<code>getPress</code>	<code>getTempDef</code>	<code>pressSat</code>	<code>wait</code>
<code>density</code>	<code>getPressAtm</code>	<code>getTempFahr</code>	<code>setPress</code>
<code>waterVaporPressure</code>	<code>equals</code>	<code>getPressDef</code>	<code>getTempKelvin</code>
<code>setPressAtm</code>	<code>getCaveats</code>	<code>getPressMbar</code>	<code>getThermCond</code>
<code>setPressMbar</code>	<code>getClass</code>	<code>getRelHum</code>	<code>getViscosity</code>
<code>setSpecHum</code>	<code>getDensity</code>	<code>getSpecHum</code>	<code>hashCode</code>
<code>setTemp</code>			

To find out more information about these methods, one can use `methodsview('class')`. For example, typing

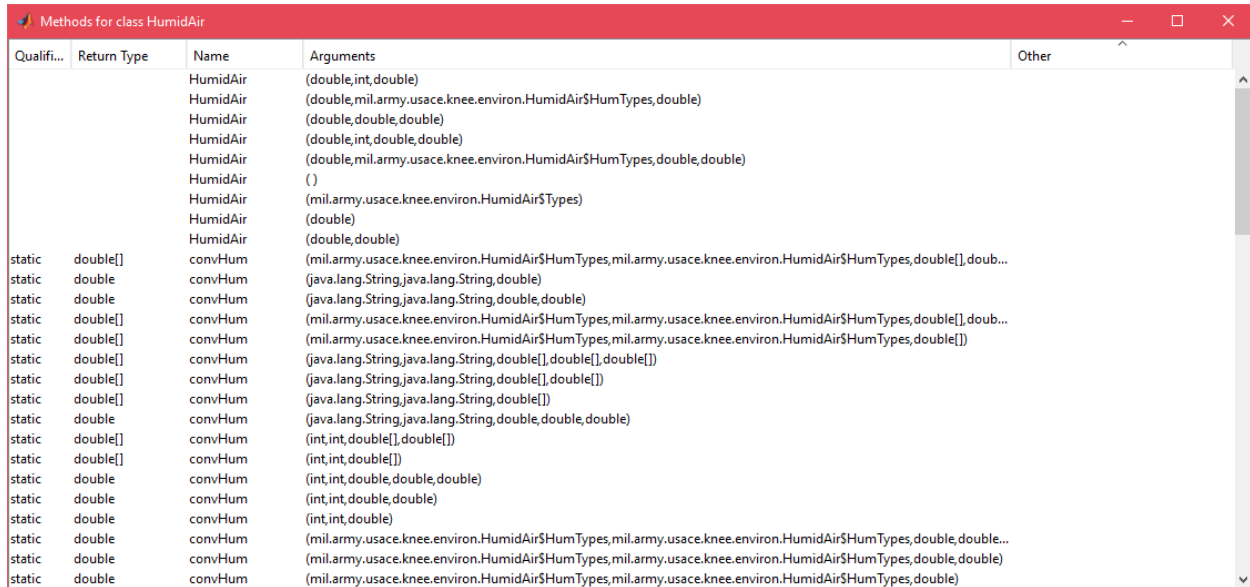
```
methodsview('HumidAir')
```

results in the following popup window (Figure 2), which shows the name, return type, qualifier (e.g., static), and arguments for each method in the `HumidAir` class.

The package containing the desired class should be imported before typing `methods` or `methodsview` in the MATLAB command prompt. If not, then the entire package name needs to be specified in the MATLAB command prompt, for example:

```
methodsview('mil.army.usace.knee.envIRON.HumidAir')
```

Figure 2. Output from the methodsview('HumidAir') command in MATLAB.



Qualifi...	Return Type	Name	Arguments	Other
		HumidAir	(double,int,double)	
		HumidAir	(double,mil.army.usace.knee.environ.HumidAir\$HumTypes,double)	
		HumidAir	(double,double,double)	
		HumidAir	(double,int,double,double)	
		HumidAir	(double,mil.army.usace.knee.environ.HumidAir\$HumTypes,double,double)	
		HumidAir	()	
		HumidAir	(mil.army.usace.knee.environ.HumidAir\$Types)	
		HumidAir	(double)	
		HumidAir	(double,double)	
static	double[]	convHum	(mil.army.usace.knee.environ.HumidAir\$HumTypes,mil.army.usace.knee.environ.HumidAir\$HumTypes,double[],doub...	
static	double	convHum	(java.lang.String,java.lang.String,double)	
static	double	convHum	(java.lang.String,java.lang.String,double,double)	
static	double[]	convHum	(mil.army.usace.knee.environ.HumidAir\$HumTypes,mil.army.usace.knee.environ.HumidAir\$HumTypes,double[],doub...	
static	double[]	convHum	(mil.army.usace.knee.environ.HumidAir\$HumTypes,mil.army.usace.knee.environ.HumidAir\$HumTypes,double[])	
static	double[]	convHum	(java.lang.String,java.lang.String,double[],double[],double[])	
static	double[]	convHum	(java.lang.String,java.lang.String,double[],double[])	
static	double[]	convHum	(java.lang.String,java.lang.String,double[])	
static	double	convHum	(java.lang.String,java.lang.String,double,double,double)	
static	double[]	convHum	(int,int,double[],double[])	
static	double[]	convHum	(int,int,double[])	
static	double	convHum	(int,int,double,double,double)	
static	double	convHum	(int,int,double,double)	
static	double	convHum	(int,int,double)	
static	double	convHum	(mil.army.usace.knee.environ.HumidAir\$HumTypes,mil.army.usace.knee.environ.HumidAir\$HumTypes,double,double...	
static	double	convHum	(mil.army.usace.knee.environ.HumidAir\$HumTypes,mil.army.usace.knee.environ.HumidAir\$HumTypes,double,double)	
static	double	convHum	(mil.army.usace.knee.environ.HumidAir\$HumTypes,mil.army.usace.knee.environ.HumidAir\$HumTypes,double)	

3.4.4 Conversion of data types

The MATLAB interface to Java performs a number of automatic conversions between MATLAB and Java data types. These are fully described in the MATLAB help page for “Pass Data to Java Methods.” The main thing to keep in mind is that MATLAB’s basic data type for variables is a two-dimensional (2-D) array of complex numbers that may, in practice, hold scalar data, a one-dimensional (1-D) array, or a 2-D array. The values may be integer, real, or complex. Hence, when there are overloaded Java methods (methods with the same name) having the same number of arguments but with different types, MATLAB must choose which method is the closest match to a specified argument list. The process involves internal rules for selecting the most appropriate Java method at runtime.

In practice, the rules used by MATLAB usually lead to intuitive results. In particular, if a MATLAB variable holds a real-valued scalar, MATLAB will first search for a Java method with an argument of the Java type `double`. If such a method does not exist, it will search for a method with type `float`, then type `long`, then type `int`, and so forth. Analogously, the Java `float` type corresponds to MATLAB `single`; thus MATLAB will attempt to find a method with Java type `float` given a variable of type `single`.

The Java `int` type corresponds to the MATLAB `int32` type, whereas `long` corresponds to `int64`. Thus, if the programmer’s intent is to call a Java method with argument of type `int` and there are overriding methods that

take precedence, the variable must be first converted to the MATLAB type `int32`.

A MATLAB variable of type `logical` will be mapped to a Java `boolean` if possible. MATLAB `char` maps to a Java `String`.

Other nonsimple Java data types (i.e., Java objects) are not converted by MATLAB. Rather, MATLAB retains a reference to the Java object, just as Java would.

As a rather contrived example, suppose we have a Java class that defines two methods called `add`, as follows:

```
public double add (double x, double y)
public int add(int x, int y)
```

Suppose we then set

```
val1 = 3.69;
val2 = 4.2;
```

If we then enter

```
disp(add(val1, val2))
```

MATLAB will call the `add` method for double arguments, as expected. But, what happens if we instead set

```
val1 = 3;
val2 = 4;
```

Since MATLAB automatically stores these values in its native double-precision floating-point format, the preceding `disp` command will automatically call the Java `add` method for double arguments. To call the `int` method, we must explicitly enter

```
disp(add(int32(val1), int32(val2)))
```

MATLAB also performs automatic conversions based on array dimensions. If the MATLAB variable is a scalar (dimensions 1 by 1), it will be converted

to a scalar value in Java. If the MATLAB variable has a one nonsingleton dimension (e.g., it is $N \times 1$ or $1 \times N$), it will be mapped to a 1-D Java array. If it has two nonsingleton dimensions, it will be mapped to a 2-D Java array. For example,

```
z = [1.1 1.3; 2.4 3.5];
```

will map to a Java `double[][]` array.

Arrays of nonsimple data types are handled using MATLAB's `javaArray` function.

3.4.5 The MATLAB `disp` command

When a Java object is called by the MATLAB `disp` command, MATLAB will call the Java class's `toString` method, which provides customized output depending on the definition within each Java class. Throughout the EASEE `environ` and `geo` packages (and most others packages in EASEE), an effort has been made to provide `toString` methods that usefully summarize the properties of the object. What this means, in practice, is that `disp` can be used to display helpful information about the object. For example, typing

```
atm = HumidAir(18.1, 0.012);
```

creates an instance of the `HumidAir` class (which will be discussed later). When this object is displayed by typing

```
disp(atm)
```

we see the following information:

```
Humid air: temperature 18.1 (C), specific humidity 0.012,  
           pressure 101.33 (kPa)
```

3.4.6 Java enumerations and inner classes

Many classes in EASEE involve a programming pattern in which a Java enumeration class is used to define a number of *types* of an object. For example, the `Helicopter` class defines types for Apache, Blackhawk, and other helicopters. The `soil` class defines types for sandy soil, clayey soil,

etc. While these enumerations are very useful and convenient when programming in Java, unfortunately they can be a bit awkward to use from MATLAB, particularly when they are defined inside another class. Thus, we describe here how to handle the Java enumerations from MATLAB.

As an example of an enumeration that is implemented in its own separate class (as opposed to an inner class), let us first consider land cover definitions in EASEE. (The land cover definitions will be discussed more fully later in this report; here our focus is on describing the programming pattern.) One of the land cover systems implemented in EASEE is the National Land Cover Dataset (NLCD) 2001. This system consists of 20 different land cover types. The class in EASEE that implements these types is called `NationalLandcoverDataset2001Types`, where “Types” is generally added to the end of an enumeration name. `NationalLandcoverDataset2001Types`, in this case, is itself an enumeration; that is, we do not have to deal with the complication of an inner class.

To import `NationalLandcoverDataset2001Types`, we type

```
import mil.army.usace.knee.envIRON.NationalLandcoverDataset2001Types;
```

The `values` method is used to create an array holding all of the available enumerations in the order they are defined:

```
NLCDTypes = NationalLandcoverDataset2001Types.values();
```

If one were to simply type `disp(NLCDTypes)`, MATLAB would provide a rather uninformative listing of references to Java objects. To actually list the land cover enumerations, one can loop through the array while calling the `name` method, as follows:

```
for m=1:length(NLCDTypes), disp(NLCDTypes(m).name); end
```

This command will yield the following output:

```
OPEN_WATER  
ICE_SNOW  
DEVELOPED_OPEN  
DEVELOPED_LOW  
DEVELOPED_MEDIUM
```

```

DEVELOPED_HIGH
BARREN_LAND
FOREST_DECIDUOUS
FOREST_EVERGREEN
FOREST_MIXED
SHRUB_DWARF
SHRUB_SCRUB
GRASSLAND
SEDGE
LICHENS
MOSS
PASTURE
CULTIVATED_CROPS
WETLANDS_WOODY
WETLANDS_HERBACEOUS

```

The previous listing shows the actual names of the enumerations as they would appear in the Java code. More useful, descriptive names can be listed using the `toString` method in place of the `name` method. For example,

```
for m=1:length(NLCDTypes), disp(NLCDTypes(m).toString); end
```

yields

```

Open Water (11)
Perennial Ice/Snow (12)
Developed, Open Space (21)
Developed, Low Intensity (22)
Developed, Medium Intensity (23)
Developed, High Intensity (24)
Barren Land (31)
Deciduous Forest (41)
Evergreen Forest (42)
Mixed Forest (43)
Dwarf Scrub (51)
Shrub/Scrub (52)
Grassland/Herbaceous (71)
Sedge/Herbaceous (72)
Lichens (73)

```

```

Moss (74)
Pasture/Hay (81)
Cultivated Crops (82)
Woody Wetlands (90)
Emergent Herbaceous Wetlands (95)

```

The one-to-one correspondence of these descriptive names to the original enumerations should be clear. The values in parentheses correspond to an integer coding system for the land cover, which will be discussed later. Since the MATLAB `disp` command automatically invokes the `toString` method, the following code would produce the same descriptive listing:

```

for m=1:length(NLCDTypes), disp(NLCDTypes(m)); end

```

If we wish to create a new variable in MATLAB corresponding to one of these enumerations, for example, `FOREST_DECIDUOUS`, we can enter

```

lcCode = NationalLandcoverDataset2001Types.FOREST_DECIDUOUS;

```

Alternatively, we could count down the listing and note that `FOREST_DECIDUOUS` is the eighth element of the array. Thus, equivalent to the preceding, we could use

```

lcCode = NLCDTypes(8);

```

Programming with the Java enumerations becomes a bit more cumbersome when the enumeration is implemented as a Java nested class. An example is accessing the predefined types in the `HumidAir` class, which represent the properties of humid air (a mixture of dry air and water vapor). A number of predefined characteristic air masses are defined within `HumidAir` by an inner class called `Types`. They can be listed by entering

```

HumidAirTypes = javaMethod('values',
    'mil.army.usace.knee.envIRON.HumidAir$Types');
for m=1:length(HumidAirTypes), disp(HumidAirTypes(m)); end

```

This will result in

```

Polar, winter
Polar, summer

```

```
Cold continental, winter
Cold continental, summer
Temperate, winter
Temperate, summer
Arid, winter
Arid, summer
Tropical, winter
Tropical, summer
ISA mean sea level
ISA tropopause
ISA stratosphere (20 km)
ISA stratosphere (32 km)
ISA stratopause
ISA mesosphere (51 km)
ISA mesosphere (71 km)
ISA mesopause
```

Note that the inner class is invoked from MATLAB using the \$ operator, which *must* be passed via the `javaMethod` function. Therefore the `Types` in the `HumidAir` class cannot be accessed directly; the approach of creating an array to access the enumerations, as described above with respect to the land cover, must be used. Thus to create a `HumidAir` object with properties characteristic of temperate summer air, we would thus enter

```
airCode = HumidAirTypes(6);
atm = HumidAir(airCode);
```

or, more simply,

```
atm = HumidAir(HumidAirTypes(6));
```

4 Geographic Coordinates, Grids, and Elevation Maps

In this section, we consider some features of EASEE that, while they do not pertain directly to acoustical calculations, do support those calculations in important ways. Specifically, we describe KNEE's `geo` package, which provides capabilities for setting and manipulating Earth coordinates and for representing data on geographic grids. To use the `geo` package, first import it by entering

```
import mil.army.usace.knee.geo.*;
```

4.1 Geographic coordinates

The `geo` package supports Earth coordinate specifications in latitude/longitude, Universal Transverse Mercator (UTM), and Military Grid Reference System (MGRS). The latitude/longitude coordinates are generally specified in decimal degrees, whereas UTM and MGRS are specified in meters. The WGS84 Earth ellipsoid (datum) is used by default for all coordinate systems.

The `GeoCoord` class in `geo` is used to create a geographic coordinate. Here are the primary four constructors for this purpose:

```
public GeoCoord(double lat, double lng)
public GeoCoord(double easting, double northing, char latZone,
               int lngZone)
public GeoCoord(double easting, double northing, String mgrsZone)
public GeoCoord(GeoCoord origin, double relEasting, double
               relNorthing)
```

The first three of these correspond to the three coordinate systems mentioned above. The fourth specifies a coordinate relative to an origin coordinate but offset by indicated distances to the east and north of the origin coordinate. The offsets are interpreted based on the UTM projection in the grid zone of the origin.

For example, to create a coordinate at a latitude of 43.5° and a longitude of -72.4° (i.e., 72.4°W), we would enter

```
gc1 = GeoCoord(43.5, -72.4);
```

Once a coordinate has been constructed, we can use the following methods to retrieve its properties:

- `getLat` for the latitude (decimal degrees)
- `getLng` for the longitude (decimal degrees)
- `getUTMEasting` for the UTM easting (m)
- `getUTMNorthing` for the UTM northing (m)
- `getLatZone` for the UTM latitude zone (a character)
- `getLngZone` for the longitude zone number (an integer)
- `getMGRSEasting` for the MGRS easting (m)
- `getMGRSNorthing` for the MGRS northing (m)
- `getMGRSZone` for the MGRS grid zone (a string)

Furthermore, the following methods convert values to different units:

- `toString` or `toStringDecDeg` (which are equivalent) displays the coordinate in latitude/longitude in decimal degrees.
- `toStringDMS` displays the coordinate in degrees/minutes/seconds.
- `toStringUTM` displays the coordinate in the UTM system.
- `toStringMGRS` displays the coordinate in the MGRS system.

Some other useful methods in the `GeoCoord` class calculate quantities of one geographic coordinate relative to another. For each of these methods, the current instance of the geographic coordinate is the origin, whereas the specified argument is the termination. For example,

```
gc1 = GeoCoord(43.5, -72.4);  
gc2 = GeoCoord(43.3, -72.2);  
disp(gc1.getDistance(gc2))
```

displays the distance between the geographic coordinates `gc1` and `gc2`. The calculation is based on the UTM projection in the grid zone of `gc1`. The output in this case is 27.47 km. The method `getRelBearing` returns the relative bearing (azimuth) of the termination relative to the origin. Hence,

```
disp(gc1.getRelBearing(gc2))
```


returns 142.04° , where the orientation is specified in the geographic convention, namely where 0° is northward and the angle increases clockwise. Therefore, in this case, `gc2` is to the southeast of `gc1`. Similarly, `getRelNorthing` and `getRelEasting` return the relative distance to the north and relative distance to the east (as based on the UTM projected coordinates).

In addition to `GeoCoord`, there is a class called `GeoCoord3D`, which is very similar to `GeoCoord` except that the constructors take an additional argument that indicates the height above ground level in meters.

4.2 Geographic grids

The `geo` package also enables the creation of geographic grids, by which is meant a regular (Cartesian) grid of geographic coordinates. The coordinates can be specified using latitude and longitude, UTM, or MGRS easting and northing. Note that a regularly spaced grid in latitude and longitude does *not* equate to a regularly spaced grid in distance. A 2-D grid can be created by specifying the coordinates of the southwest and northeast corners and the number of points in each direction:

```
SWcorner = GeoCoord(43.0, -72.0); % lat and lng of SW corner
NEcorner = GeoCoord(43.4, -71.7);
ptsLat = 50; % number of points in latitude direction
ptsLng = 40; % number of points in longitude direction
grid = GeoGridCart2D(SWcorner, NEcorner, ptsLat, ptsLng);
```

Note that the previous `GeoGridCart2D` constructor defaults to a latitude/longitude grid. It is immaterial that the corner coordinates were specified in latitude/longitude; they can be specified in any system. The following constructor is equivalent to the previous although the grid system and datum (Earth ellipsoid model) are specified explicitly:

```
grid = GeoGridCart2D(GeoGridTypes.LAT_LNG, DatumTypes.WGS84,
    SWcorner, NEcorner, ptsLat, ptsLng);
```

Alternatively, if we desire a UTM grid, we may specify:

```
grid = GeoGridCart2D(GeoGridTypes.UTM, DatumTypes.WGS84,
    SWcorner, NEcorner, ptsNorth, ptsEast);
```

where `ptsNorth` and `ptsEast` are the number of points to the north and to the east, respectively. The grid may also be specified by indicating the spacing between grid points in each direction:

```
SWcorner = GeoCoord(43.0, -72.0); % lat and lng of SW corner
ptsLat = 50; % number of points in latitude direction
ptsLng = 40; % number of points in longitude direction
delLat = 0.01; % spacing in latitude direction
delLng = 0.01; % spacing in longitude direction
grid = GeoGridCart2D(GeoGridTypes.LAT_LNG, DatumTypes.WGS84,
    SWcorner, delLat, delLng, ptsLat, ptsLng);
```

Within Java, the grid data are, by convention, stored such that latitude (or northing) corresponds to the inner, or faster, varying array index, whereas longitude (or easting) corresponds to the outer, or slower, varying grid index. That is, for a 2-D array `f`, use `f[i][j]` to retrieve a point at longitude index `i` and latitude index `j`.

There are two extensions of `GeoGridCart2D`, namely `GeoGridCart2DInt` and `GeoGridCart2DDouble`. These include grids to hold 2-D integer array data (Java `int[][]`) and 2-D double array data (Java `double[][]`), respectively. For example, one might store a digital elevation model as a `GeoGridCart2DDouble` (as will be discussed momentarily). `GeoGridCart2DInt` includes methods called `setDataGrid` and `getDataGrid2DInt`, which set and retrieve the integer data array. Similarly, `GeoGridCart2DDouble` includes methods called `setDataGrid` and `getDataGrid2D`, which set and retrieve the double data array.

The `geo` package also includes a class called `GeoGridCart3D`, which accommodates a three-dimensional (3-D) grid. The third coordinate is altitude.

4.3 Digital elevation, surface, land cover, and soil grids

A digital elevation model (DEM) represents the terrain elevation as a function of the geographic coordinates. Normally, the DEM is understood to represent the “bare Earth” elevation (i.e., the elevation when all the objects on the surface, such as vegetation and buildings, are removed). The digital surface model (DSM) represents the elevation of the bare Earth plus the surface elements. Currently, the acoustical calculations in EASEE use only the DEM; the surface elements are ignored.

Typically, the DEM and DSM would be read from a file, but to illustrate the concept here for how a DEM is constructed and used in an EASEE calculation, we contrive a DEM using the MATLAB `peaks` function for the terrain elevations:

```
SWcorner = GeoCoord(43.0, -72.0); % lat and lng of SW corner
NEcorner = GeoCoord(SWcorner, 10000.0, 10000.0); %NE corner
          10 km N, 10 km E
nPts = 50; % number of points in each direction of grid
dem = GeoGridCart2DDouble(SWcorner, NEcorner, nPts);
dem.setDataGrid(peaks(nPts));
```

This code snippet creates a domain with the southwest corner at the specified latitude and longitude and with the northeast corner 10 km to the north and 10 km to the east of the southwest corner, as based on a UTM coordinate projection. A geographic grid of size of 50×50 points is then created, and the values of that grid are set using the `peaks` function.

Geographic grids are also used to represent land cover and soil properties, which vary along the Earth-air interface. Although we have not yet discussed in detail how land cover and soil types are represented (that will be the topic of sections 5.2.3 and 5.2.4, respectively), for present purposes it is sufficient to understand that these properties are represented by integers, which encode the type of land cover or soil according to standardized systems. The standardized system is described by a “decoder” class in EASEE. Specifically, suppose `lcDecoder` and `soilDecoder` are objects specifying the mapping of integer data to land cover and soil types, respectively, and that `lcGrid` and `soilGrid` contain the actual integer data grids for the land cover and soil. Normally, these land cover or soil grids would be read from a file, such as a GeoTiff format file. However, for illustrative purposes, it is also worthwhile to consider how such data could be specified manually, as we did with the DEM. Building on the previous example where we used the MATLAB `peaks` function to construct a DEM, let us suppose that the land cover type consists of woody wetlands at negative elevations and mixed forest at positive elevations. Referring to the list of NLCD 2001 land cover types in section 3.4.6, we see that the codes for these land cover types (shown in the parentheses) are 90 and 43, respectively. Hence, we set

```
lcData = 90*ones(nPts);  
lcData(peaks(nPts)>0) = 43;  
lcGrid = GeoGridCart2DInt(SWcorner, NEcorner, nPts);  
lcGrid.setDataGrid(int32(lcData));
```

Here, the MATLAB `int32` function converts the double array to a 32-bit integer array, which corresponds to the data type `int[][]` in Java. Similarly, we can construct a soil grid consisting of organic silts (FASST [Fast All-Season Strength] soil type number 11 [see section 5.2.4 for information about FASST]) at negative altitudes and silty clayey sand (FASST soil type number 16) at positive altitudes:

```
soilData = 11*ones(nPts);  
soilData(peaks(nPts)>0) = 16;  
soilGrid = GeoGridCart2DInt(SWcorner, NEcorner, nPts);  
soilGrid.setDataGrid(int32(soilData));
```

5 Environmental Representations

In this section, we discuss EASEE's `environ` package, which represents environmental data. By *environment*, we mean the atmosphere, Earth surface characteristics (terrain), and solid-earth (subsurface) properties. In this section, we show how to create (*instantiate*, in Java terminology) objects representing the environment, for the eventual purpose of using them in acoustical calculations. EASEE supports a great variety of approaches to creating the environmental objects. Information on approaches not explicitly described here can be obtained by examining the original Java code or Javadoc.

From a software-design perspective, EASEE has its own, complete internal representation of the atmosphere, surface, and subsurface, which serves as a *data abstraction layer* between the various types of external environmental models and data and the signal and sensor models found in EASEE. The main components of this representation are the atmospheric profiles, atmospheric surface layer (part of the atmosphere adjacent to the ground), subsurface profiles (primarily intended for seismic calculations), terrain elevations, land cover properties, soil type, soil moisture, snow type, and snow depth. The value of such an abstraction layer is that it isolates the signal and sensor models from the external environmental models and data; when a new environmental model or data source is interfaced with EASEE, all of the signal and sensor models will continue to function without modification.

EASEE currently has three basic environmental representations. All three are subclasses of the abstract class `EnvironScenario`. Figure 3 shows the hierarchy of environmental representations and the data elements at each stage in the hierarchy. In order of increasing complexity, the three subclasses are homogeneous (the `EnvironHomo` class), 1-D (the `EnvironVertProf` class), and 3-D (the `EnvironThreeDim` class). Here, “homogenous” means that the atmosphere and subsurface have constant properties. Terrain properties, such as elevation and land cover, may nonetheless vary along the Earth surface. The 1-D representation includes atmospheric and subsurface profiles that vary vertically. The 3-D representation includes both vertical and horizontal variability. At the time of the writing of this report, the 3-D representation was nearly com-

plete although modeling capabilities using the 3-D atmospheric and sub-surface fields had yet to be implemented. Thus, we do not describe the `EnvironThreeDim` class here.

Figure 3. Hierarchy of EASEE's environmental classes. Phrases in *bold* are the main environmental classes within EASEE, and *italicized* phrases are subclasses of the foremost environmental class—`EnvironScenario`. All other listings are fields (e.g., friction velocity is a field within `AtmosSfcLayer`, which is a field within `AtmosOneDim`, which is a field within `EnvironVertProf`). All phrases with only capitalized terms (e.g., `EnvironVertProf`) are instances of classes. *Blue* font color indicates recent additions.

EnvironScenario (common to all representations)

Digital elevation map

Digital surface map

Land cover type grid

Soil type grid

Snow type grid

Snow depth grid

Soil moisture grid

➤ *EnvironHomo (homogeneous atmosphere/ground)*

Homogeneous air layer

Homogeneous solid layer

➤ *EnvironVertProf (horizontally stratified environment)*

`AtmosOneDim` (1-D atmosphere)

`AtmosSurfLayer`

Friction velocity, sensible/latent heat fluxes, surface temperature, etc.

`AtmosVertProf`

Profiles of wind, temperature, humidity, and pressure

`Clouds`

Low-, mid-, and high-cloud layers, each with height and cover fraction

`SubSurfOneDim` (1-D subsurface)

`SeismicVertProf`

Profiles for density, compressional and shear wave speed, attenuation

➤ *EnvironThreeDim (3-D environment)*

`AtmosThreeDim` (3-D atmosphere)

2-D grid of atmospheric profile locations

`AtmosSurfLayer[][]` (2-D array of `AtmosSurfLayer`)

`AtmosVertProf[][]` (2-D array of `AtmosVertProf`)

`Clouds[][]` (2-D array of `Clouds`)

`SubSurfThreeDim` (3-D subsurface)

2-D grid of subsurface profile locations

`SeismicVertProf[][]` (2-D array of `SeismicVertProf`)

In this section, we describe construction of the `EnvironHomo` and `EnvironVertProf` objects. Before actually getting to that point, we discuss construction of a number of objects of many types that are *components* of the environmental classes (e.g., air and solid representations).

For the examples in this section, KNEE's `environ` and `geo` packages must both be imported using the commands (typed at the MATLAB prompt):

```
import mil.army.usace.knee.environ.*;  
import mil.army.usace.knee.geo.*;
```

5.1 Atmospheric constants and conversions

As a preliminary, we describe in this subsection a simple, but useful, capability in EASEE, namely the availability of a number of constants related to significant physical properties, particularly the thermodynamic properties of air and water. These are found in the `AtmosConstants` class in KNEE's `environ` package. To display gravitational acceleration, for example, enter

```
disp(AtmosConstants.GRAV_ACC)
```

To display standard sea-level pressure, enter

```
disp(AtmosConstants.SEA_LEVEL_PRESS)
```

Or, to display the freezing point in Kelvin, enter

```
disp(AtmosConstants.FREEZING_PT_KELVIN)
```

Many other properties are available, which can be seen by examining the Javadoc or the Java code for the `AtmosConstants` class. Alternatively, one may list the names of each physical property within `AtmosConstants` by typing the following:

```
fieldnames(AtmosConstants)
```

This produces

```
'GRAV_ACC'  
'SEA_LEVEL_PRESS'  
'STAND_PRESS_MBAR'  
'FREEZING_PT_FAHR'  
'FREEZING_PT_KELVIN'  
'ROOM_TEMP_CELSIUS'  
'CELSIUS_TO_FAHR'  
'TRIPLE_PT_KELVIN'  
'GAS_CONST_DRY_AIR'  
'MOL_MASS_DRY_AIR'  
'MOL_MASS_WATER_VAPOR'
```

```
'MOL_MASS_RATIO'
'SPEC_HEAT_RATIO_DRY_AIR'
'SPEC_HEAT_DRY_AIR'
'SPEC_HEAT_WATER_VAPOR'
'LATENT_HEAT_WATER_VAPOR'
'POISSON_CONST'
'DRY_LAPSE_RATE'
```

The `AtmosConstants` class also provides static methods for converting temperature and pressure between various units. Either scalar values or arrays may be passed to these methods. The method `degCToDegF`, for example, converts degrees Celsius to degrees Fahrenheit. Thus

```
disp(AtmosConstants.degCToDegF([0.0 100.0]))
```

displays an array with values of 32.0 and 212.0. Similarly,

- `degCToDegK` converts temperature from degrees Celsius to Kelvin,
- `degFToDegC` converts temperature from degrees Fahrenheit to degrees Celsius, and
- `degKToDegC` converts temperature from Kelvin to degrees Celsius.
- `paToMbar` converts pressure from Pascals to millibars,
- `mbarToPa` converts pressure in the other direction,
- `paToAtm` converts pressure from Pascals to atmospheres, and
- `atmToPa` converts pressure in the other direction.

The `AtmosConstants` class also has static methods for converting between ordinary temperature and potential temperature:

- `tempKToPotTempK` converts from ordinary to potential temperature.
- `potTempKToTempK` converts in the other direction.

These methods each take two arguments, the first being the temperature in Kelvin, the second being the pressure in Pascal.

5.2 Environmental components

In this subsection, we describe the most important components of the EASEE environmental representations: the `EnvironHomo` and `EnvironVertProf` classes.

5.2.1 Humid air representation

The `HumidAir` class represents a mixture of dry air and water vapor. Naturally, it plays a particularly important role in modeling the atmosphere and its impact on sound propagation. A `HumidAir` object is constructed by specifying temperature, humidity, and pressure. The most basic constructor is

```
HumidAir(double temp, double specHum, double press)
```

where `temp` is the temperature in degrees Celsius, `specHum` is the specific humidity (ratio of the mass of water vapor [kg] to the total mass of the air parcel [kg]), and `press` is the pressure in Pascals. For example, entering

```
atm = HumidAir(18.1, 0.012, 99000.0);
```

creates a MATLAB variable called `atm`, which holds a Java `HumidAir` object representing air at a temperature of 18.1°C, specific humidity of 0.012, and pressure of 99000.0 Pa.

The constructor may also be called with no arguments, a single argument (temperature), or two arguments (temperature and specific humidity). In these cases, default values are used for the unspecified variables. The default values are a temperature of 20°C, zero humidity, and standard sea-level pressure (101325 Pa). For example,

```
atm = HumidAir(18.1, 0.012);
```

would assume standard sea-level pressure.

If we wish to specify the humidity in some way other than specific humidity, we can use `HumidAir`'s conversion capabilities. Conversions are specified using the following integer codes: 1 = relative humidity (%), 2 = molar concentration, 3 = specific humidity, 4 = mixing ratio, and 5 = dew-point temperature (°C). The code precedes the humidity value. Thus, if the input value is a relative humidity of 40%, the corresponding constructor (assuming sea-level pressure) would be

```
atm = HumidAir(18.1, int32(1), 40.0);
```

A number of predefined characteristic air masses are also available. These are defined by a nested class in `HumidAir` called `Types`, which can be used to construct a `HumidAir` object, as described in section 3.4.6.

The following are some useful getter methods for the `HumidAir` class:

- `getTemp`, which gets the temperature in degrees Celsius
- `getTempKelvin`, which gets the temperature in Kelvin
- `getTempFahr`, which gets the temperature in degrees Fahrenheit
- `getPress`, which gets the pressure in Pascals
- `getPressMbar`, which gets the pressure in millibars
- `getPressAtm`, which gets the pressure in atmospheres
- `getDensity`, which gets the density in kilograms per cubic meter
- `getSpecHum`, which gets the specific humidity
- `getRelHum`, which gets the relative humidity in percent (%)
- `getDewPoint`, which gets the dew-point temperature in degrees Celsius
- `getMixRat`, which gets the mixing ratio
- `getMolHum`, which gets the molar humidity

For example, the density can be displayed by entering

```
disp(atm.getDensity)
```

The `HumidAir` class also contains a static method called `convHum`, which converts between different measures of humidity. The humidity measure is specified using the previously described integer codes. Thus, to convert from a relative humidity of 52% to mixing ratio, for example, we would enter

```
disp(HumidAir.convHum(1, 4, 52.0))
```

The preceding assumes a temperature of 20.0°C at sea-level pressure. To explicitly specify the temperature and pressure, add them as additional arguments. For a temperature of 3.0°C and pressure of 10^5 Pa, enter

```
disp(HumidAir.convHum(1, 4, 52.0, 3.0, 10^5))
```

5.2.2 Solid-earth representation

The `SolidIsoLinear` class represents the properties of isotropic, linear (primarily solid) materials. This `SolidIsoLinear` class has an associated

class called `SolidIsoLinearTypes`, which defines several common solid materials of interest. The types available can be listed by entering

```
solidTypes = SolidIsoLinearTypes.values;  
for m=1:length(solidTypes), disp(solidTypes(m).name); end
```

which results in

```
AIR  
WATER  
ICE  
SOIL  
SAND_UNSAT  
SAND_SAT  
CONCRETE  
SANDSTONE  
SHALE  
GRANITE  
METAMORPHIC  
BASALT
```

Omitting the `name` method above produces the following descriptive list:

```
air  
water  
ice  
soil  
unsaturated sand  
saturated sand  
concrete  
sandstone  
shale  
granite  
metamorphic rock  
basalt
```

To construct a `SolidIsoLinear` object, we can simply pass the desired `SolidIsoLinearTypes` enumeration (e.g., for unsaturated sand)

```
solid = SolidIsoLinear(solidTypes(5));
```

or, more directly,

```
solid = SolidIsoLinear(SolidIsoLinearTypes.SAND_UNSAT);
```

Similarly, if we wish to construct a subsurface of granite (the tenth element in the array), we would enter

```
subSurf = SolidIsoLinear(solidTypes(10));
```

or

```
solid = SolidIsoLinear(SolidIsoLinearTypes.GRANITE);
```

Alternatively, if we do not wish to use one of the predefined types, the solid properties can be specified directly using one of the constructors

```
SolidIsoLinear(double rho, double K, double G)
```

or

```
SolidIsoLinear(double rho, double cp, double cs, double Qp,  
               double Qs)
```

The first constructor specifies the density, bulk modulus, and shear modulus and assumes the medium is nonattenuative. The second constructor specifies the density, compressional (P-) wave speed, shear (S-) wave speed, quality factor for compressional waves, and quality factor for shear waves. The quality factors characterize the wave attenuation and may be set to 1000 for a nonattenuative medium.

5.2.3 Land cover representation

The land cover is normally set using one of the predefined types, which are based on a number of standardized systems for enumerating land cover. The systems currently supported by EASEE include NLCD 1992 and 2001 (Java classes `NationalLandcoverDataset1992Types` and `NationalLandcoverDataset2001Types`, respectively), `GeoCover` (`GeoCoverLCTypes`), `WorldView` (`WorldView2LCTypes`), and `VisNav` (`VisnavLCTypes`). Earlier, we discussed how to list the land cover enumerations associated with `NationalLandcoverDataset2001Types`. The same

procedure applies to the other land cover classes. For example, to see the GeoCoverLCTypes, enter

```
GCTypes = GeoCoverLCTypes.values();
for m=1:length(GCTypes), disp(GCTypes(m)); end
```

which yields

```
Forest, Deciduous (1)
Forest, Evergreen (2)
Shrub/Scrub (3)
Grassland (4)
Barren/Minimal Vegetation (5)
Urban/Built-Up (6)
Agriculture, General (7)
Agriculture, Rice/Paddy (8)
Wetland, Permanent/Herbaceous (9)
Wetland, Mangrove (10)
Water (11)
Permanent or Nearly Permanent Ice/Snow (12)
Cloud/Cloud Shadow/No Data (13)
```

Once we have decided on the land cover system and particular enumeration of interest, it is straightforward to construct the actual land cover of interest. Namely, we would set

```
landCov = Landcover(lcCode);
```

where `lcCode` is the enumeration and `landCov` is a new variable, specifically an instance of EASEE's `Landcover` class. For example, to create a `landcover` for the GeoCover evergreen forest type, we would enter

```
landCov = Landcover(GeoCoverLCTypes.FOREST_EVERGREEN);
```

If we now type `disp(landCov)`, output similar to the following will be displayed:

```
Landcover: Forest, Evergreen (2)
          Ar/Br root distribution: 6.706/2.175
          Min/max coverage (%): 70.0/80.0
```

```

Min/max leaf area index: 5.0/6.0
Min/max stomatal Resistance: 200.0/500.0
Maximum dew depth: 0.25
Roughness height: 1.0
Displacement height: 6.0
Default temperature: 20 (C)
Default specific humidity: 0

```

In the listings of the land cover types, the values in parentheses correspond to integer codes, which represent the land cover type in data files; for example, a code of 2 in the `GeoCover` system is “Forest, Evergreen,” whereas 6 would correspond to “Urban/Built-Up.” The integer code can be displayed using the `getLcCode` method:

```
disp(lcCode.getLcCode)
```

5.2.4 Soil representation

The soil properties are set in much the same manner as the land cover. For soil types, EASEE currently supports only one system, namely the Fast All-Season Strength (FASST) soil types, which are an extension of the U.S. Geological Survey soil classes (Frankenstein and Koenig 2004). These are enumerated in the class `FASSTSoilTypes`. We can list the predefined soil types by entering

```

FASSTTypes = FASSTSoilTypes.values();
for m=1:length(FASSTTypes), disp(FASSTTypes(m)); end

```

This results in the following output:

```

Well graded gravel (GW) (1)
Poorly graded gravel (GP) (2)
Silty gravel (GM) (3)
Clayey gravel (GC) (4)
Well graded sand (SW) (5)
Poorly graded sand (SP) (6)
Silty sand (SM) (7)
Clayey sand (SC) (8)
Inorganic silts (ML) (9)
Low-plasticity inorganic clays (CL) (10)
Organic silts (OL) (11)

```

```

Inorganic silts with fine sands (CH) (12)
High-plasticity inorganic clays (MH) (13)
High-plasticity organic clays (OH) (14)
Peat (PT) (15)
Silty clayey sand (MC) (16)
Inorganic silty clay (CM) (17)
Evaporites (EV) (18)
Concrete (CO) (20)
Asphalt (AS) (21)
Bedrock (RO) (25)
Air (AI) (27)
Water (US) (28)
Snow (SN) (30)

```

A `Soil` object is constructed by passing the desired enumeration. Thus, if we wish to set the soil type to silty clayey sand, which is the sixteenth element in the preceding array, we would enter

```
soil = Soil(FASSTypes(16));
```

where `soil` (lowercase) is now an instance of EASEE's `Soil` (initial cap) Java class.

As with the land cover enumerations, the values in parentheses in the previous listing correspond to the integer codes, which represent the soil type.

5.2.5 Snow representation

The `SnowTypes` class represents five different characteristic types of snow, based on data provided by Dr. Don Albert of the ERDC Cold Regions Research and Engineering Laboratory (pers. comm.). To list these, enter

```

snowTypes = SnowTypes.values();
for m=1:length(snowTypes), disp(snowTypes(m)); end

```

The result is

```

Snow, fresh (1)
Snow, mature always cold (2)
Snow, mature near freezing (3)
Snow, melting (4)
Snow, patchy (5)

```

Snow is actually regarded by EASEE as a class of soil types; that is, the snow layer is, in effect, a type of soil on top of the normal soil layer. Thus, to create a snow object, pass the desired enumeration to the `Soil` constructor. For example, to designate melting snow,

```
snow = Soil(snowTypes(4));
```

5.2.6 Atmospheric surface layer

The atmospheric surface layer (ASL) is defined as the part of the atmosphere in which the fluxes (momentum, heat, and moisture) are nearly equal to their values at the surface (Stull 1988). Typically, the ASL extends from the Earth's surface to about 50–200 m above. The environmental layer in EASEE includes a representation of the ASL, namely the `AtmosSurfLayer` class.

The following `AtmosSurfLayer` constructor specifies the most important surface-layer parameters directly:

```
AtmosSurfLayer(double frictionVel, double sensibleHeatFlux,
               double latentHeatFlux, double windDir, double surfTemp,
               double surfSpecHum, double roughHgt, double dispHgt)
```

Here,

- `frictionVel` is the friction velocity,
- `sensibleHeatFlux` is the sensible heat flux,
- `latentHeatFlux` is the latent heat flux,
- `windDir` is the wind direction in radians (in the Cartesian convention, where 0 is to the east and $\pi/2$ is to the north),
- `surfTemp` is the “surface” temperature, and
- `surfSpecHum` is the “surface” specific humidity.

We included the quotes around “surface” because what is actually meant here is the temperature and humidity at 2 m height, which is the standard meteorological height for surface temperature and humidity observations. Although the roughness and displacement heights (`roughHgt` and `dispHgt`) are not part of the surface-layer representation per se, they are needed internally by the constructor to deduce the wind speed at the standard wind observation height, 10 m. Finally, the `AtmosSurfLayer` class includes static

methods that convert the above Cartesian wind direction to the more familiar meteorological convention (0° from the north, increasing clockwise—`convToMetConv`) and vice versa (`convToCartConv`).

An alternative approach to specifying the surface layer involves the constructor

```
AtmosSurfLayer(WindTypes windEnum, StabilityTypes stabEnum,
               double bowen, double windDir, double surfTemp, double
               surfSpecHum, double roughHgt, double dispHgt)
```

This constructor is similar to the previous, except that the first three arguments have been replaced by an enumerator for the wind category (`windEnum`), an enumerator for the stability category (`stabEnum`), and the value of the Bowen ratio. The Bowen ratio is the ratio of the sensible to the latent heat flux at the surface, which is often assumed to be a constant for a particular land cover type. Alternatively, we could use the following constructor:

```
AtmosSurfLayer(WindTypes windEnum, StabilityTypes stabEnum,
               Landcover landCov, double windDir,
               double surfTemp, double surfSpecHum)
```

in which case the Bowen ratio, roughness height, and displacement height are automatically set to default values based on the land cover type.

To set a value for `windEnum`, first list the various wind categories as follows:

```
windTypes = javaMethod('values',
                       'mil.army.usace.knee.envIRON.AtmosSurfLayer$WindTypes');
for m=1:length(windTypes), disp(windTypes(m)); end
```

This will display the available categories:

```
Very, very low wind
Very low wind
Low wind
Moderate wind
High wind
Very high wind
```

Thus we see that `windTypes(4)` would correspond to the moderate wind category, etc. For the stability categories,

```
stabTypes = javaMethod('values',
    'mil.army.usace.knee.environ.AtmosSurfLayer$StabilityTypes');
for m=1:length(stabTypes), disp(stabTypes(m)); end
```

yields

```
Very stable (clear, night)
Stable (some clouds, night)
Neutral (cloudy)
Unstable (some clouds, day)
Very unstable (clear, day)
Highly unstable (clear, day)
```

Thus, to construct a surface layer with moderate wind and very unstable stratification, with a grassy ground surface, wind blowing to the east, a surface temperature of 20°C, and surface specific humidity of 0.009, we would set

```
landCov = Landcover(NationalLandcoverDataset2001Types.GRASSLAND);
asl = AtmosSurfLayer(windTypes(4), stabTypes(5), landCov, 0.0,
    20.0, 0.009);
```

The `AtmosSurfLayer` class has a rich set of getter methods:

- `getFrictionVel` and `getUStar` are equivalent and both return the friction velocity.
- `getSensibleHeatFlux` returns the sensible heat flux.
- `getLatentHeatFlux` returns the latent heat flux.
- `getTStar` returns the surface-layer temperature scale.
- `getQStar` returns the surface-layer humidity scale.
- `getBowenRatio` returns the Bowen ratio.
- `getWindDirCart` returns the wind direction in the Cartesian convention (radians).
- `getWindDirMet` returns the wind direction in the meteorological convention (degrees).
- `getSurfWindSpeed` returns the surface wind speed.
- `getObsHgtWind` returns the observation height for the surface wind.

- `getSurfTemp` returns the surface temperature.
- `getObsHgtTemp` returns the observation height for the surface temperature.
- `getSurfSpecHum` returns the surface specific humidity.
- `getSurfRelHum` returns the surface relative humidity.
- `getSurfDewpoint` returns the surface dew-point temperature.
- `getObsHgtHum` returns the observation height for the surface humidity.
- `getObukhovLength` returns the Obukhov length.
- `getCV2` returns the structure-function parameter for velocity.
- `getCT2` returns the structure-function parameter for temperature.

Methods are also available to calculate statistics related to turbulence; these can be found by typing `methods('AtmosSurfLayer')` or by examining the Javadoc or the original code.

5.2.7 Atmospheric vertical profiles

Many constructors are available to specify the atmospheric profiles. The following constructor, which assumes a homogeneous, motionless atmosphere, is the simplest:

```
AtmosVertProf(double[] profHgt, HumidAir air)
```

Here, `profHgt` contains the heights in meters above sea level at which the profiles are to be specified, and `air` is a humid air object specifying the properties of the homogeneous atmosphere.

The following constructor is used to specify the full atmospheric profiles directly:

```
AtmosVertProf(double[] profHgt, double[] vx, double[] vy,
              double[] T, double[] q, double[] P)
```

Here, `vx` is the eastward component of the wind (m/s), `vy` is the northward component (m/s), `T` is the temperature (°C), `q` is the specific humidity, and `P` is the ambient pressure (Pa). All of the arrays passed to this constructor must be the same length as `profHgt`.

In many applications, it can be useful to model the profiles using the Monin-Obukhov similarity theory (MOST). (See, for example, Stull 1988.) The following two constructors are available for this purpose:

```
AtmosVertProf(double[] profHgt, AtmosSurfLayer asl,
              double roughHgt, double dispHgt)
```

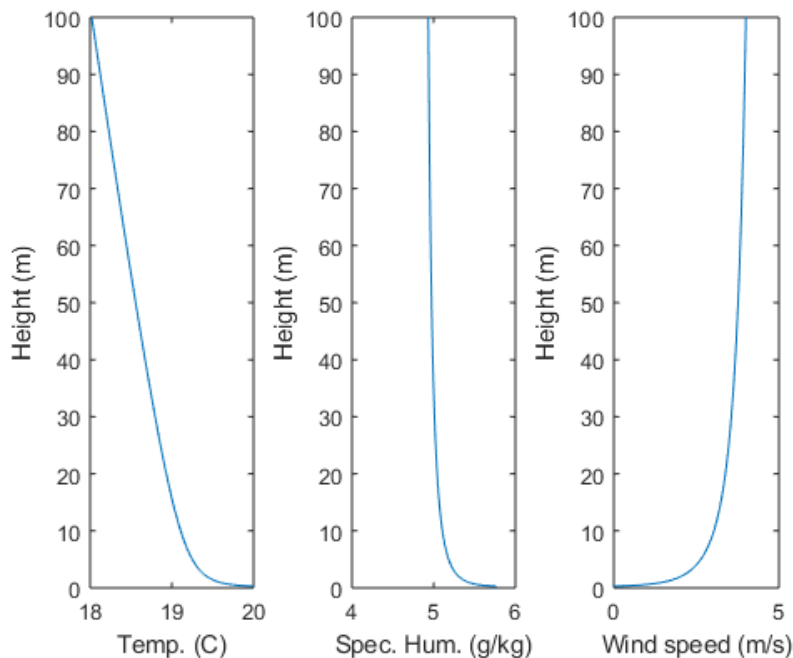
and

```
AtmosVertProf(double[] profHgt, AtmosSurfLayer asl, Landcover
              landCov)
```

Here, `asl` is an atmospheric surface layer object as described in section 5.2.5. The first of these constructors specifies the roughness and displacement heights directly; the second infers them from a land cover specification. The following code illustrates the usage of this constructor, including the creation of a plot of the temperature, specific humidity, and wind speed profiles, as shown in Figure 4:

```
surfTemp = 20.0; % in degrees Celsius
surfRelHum = 40.0;
surfSpecHum = HumidAir.convHum(1, 3, surfRelHum, surfTemp);
% 1 = relative humidity, 3 = specific humidity
windDir = 0.0; % blowing to the east
windTypes = javaMethod('values',
                        'mil.army.usace.knee.envIRON.AtmosSurfLayer$WindTypes');
windEnum = windTypes(4); % moderate wind
stabTypes = javaMethod('values',
                        'mil.army.usace.knee.envIRON.AtmosSurfLayer$StabilityTypes');
stabEnum = stabTypes(4); % unstable
landCov = Landcover(NationalLandcoverDataset2001Types.GRASSLAND);
asl = AtmosSurfLayer(windEnum, stabEnum, landCov, windDir,
                    surfTemp, surfSpecHum);
profHgt = logspace(-1, 2);
atmos = AtmosVertProf(profHgt, asl, landCov);
figure;
subplot(1,3,1)
plot(atmos.getTempProf, profHgt);
xlabel('Temp. (C)'); ylabel('Height (m)')
subplot(1,3,2)
plot(atmos1.getSpecHumProf*1000.0, profHgt);
xlabel('Spec. Hum. (g/kg)'); ylabel('Height (m)')
subplot(1,3,3)
plot(atmos.getWindSpeedProf, profHgt);
xlabel('Wind speed (m/s)'); ylabel('Height (m)')
```

Figure 4. Vertical profiles of temperature, specific humidity, and wind speed as created by specification of atmospheric surface layer properties for a moderately windy, unstable case.



Profiles are also available for benchmark acoustical calculations. The constructor in this case is

```
AtmosVertProf(double[] profHgt, AtmosVertProf.  
    BenchmarkProfileTypes profType)
```

where `profType` specifies the particular benchmark case. To list the available benchmark cases (which are a nested class of `AtmosVertProf`), enter

```
benchTypes = javaMethod('values', 'mil.army.usace.knee.envIRON.  
    AtmosVertProf$BenchmarkProfileTypes');  
for m=1:length(benchTypes), disp(benchTypes (m)); end
```

The following benchmark cases are then displayed:

```
Adiabatic lapse rate (neutral)  
Shallow temperature inversion  
Deep temperature inversion  
Homogeneous (no refraction)  
Weak upward refraction  
Strong upward refraction
```

```
Downward refraction
Acoustic benchmark 1 (from Attenborough et al. 1995)
Acoustic benchmark 2 (from Attenborough et al. 1995)
Acoustic benchmark 3 (from Attenborough et al. 1995)
Acoustic benchmark 4 (from Attenborough et al. 1995)
```

If we want to construct the benchmark profiles for Case 2 from Attenborough et al. (1995) with the profile evaluated at 5 m increments up to an altitude of 100 m, we would enter the following:

```
profHgt = 0.0:5.0:100.0;
atmos = AtmosVertProf(profHgt, benchTypes(9));
```

Once the atmospheric profiles have been constructed, the `AtmosVertProf` class offers many convenient getter methods to retrieve their properties. Besides the profiles specified explicitly by the constructor, additional profiles can be determined based on the ideal gas law and other relationships:

- `getProfHgt` returns the heights at which the profiles are specified (in meters, starting at the lowermost point and increasing upward).
- `getPressProf` returns the pressure profile in Pascals.
- `getPressProfMbar` returns the pressure profile in millibars.
- `getDensity` returns the air density profile.
- `getTempProf` returns the temperature profile in degrees Celsius.
- `getTempProfFahr` returns the temperature profile in degrees Fahrenheit.
- `getInvHgt` returns the height of the temperature inversion at the top of the boundary layer in meters.
- `getSpecHumProf` returns the specific humidity profile.
- `getMixRatProf` returns the water-vapor mixing-ratio profile.
- `getRelHumProf` returns the relative humidity profile.
- `getWindSpeedProf` returns the wind profile.
- `getWindDirCartProf` returns the wind direction in the Cartesian convention (radians).
- `getWindDirMetProf` returns the wind direction in the meteorological convention (degrees).
- `getWindProfE` returns the component of the wind velocity directed to the east.
- `getWindProfN` returns the component of the wind velocity directed to the north.

All the methods with `Prof` in their names return `double[]` arrays with one element for each profile height.

5.2.8 Clouds

The `Clouds` class defines cloud layers in the atmosphere. The most basic constructor has no arguments, namely

```
Clouds()
```

This constructor specifies clear skies (absence of cloud cover). The following is the next simplest constructor:

```
Clouds(double cloudCoverTotal, double cloudBaseHeight, double
        cloudTopHeight)
```

which specifies a single cloud layer with the indicated fractional coverage (`cloudCoverTotal`, a value between 0 and 1), cloud base height, and cloud top height. The constructor

```
Clouds(CloudLayer cloudsLow, CloudLayer cloudsMid, CloudLayer
        cloudsHigh)
```

specifies three separate cloud layers at low, medium, and high altitudes. Each of these cloud layers is constructed using

```
CloudLayer(double cloudCover, double cloudHeightBase,
            double cloudHeightTop, Clouds.CLOUD_TYPE cloudType)
```

where the parameters are the fractional cloud coverage; the cloud base height; the cloud top height; and the type of cloud, which is a nested class within the `Clouds` class.

Some of the useful getter methods in the `Clouds` class include `getCloudsLow`, `getCloudsMid`, and `getCloudsHigh`, which retrieve the lower, middle, and upper cloud layers (`CloudLayer` objects), respectively. For the `CloudLayer` objects, the method `getCloudCoverTotal` gets the total cloud cover fraction, `getCloudBaseHeight` gets the cloud base height in meters, and `getCloudTopHeight` gets the cloud top height in meters.

5.2.9 Seismic vertical profiles

The seismic vertical profiles are defined by the `SeismicVertProf` class and contain information needed to model seismic wave propagation in the subsurface. The following constructor for `SeismicVertProf` specifies the profiles directly:

```
SeismicVertProf(double[] profDepth, double[] cp, double[] cs,
                double[] qp, double[] qs, double[] rho)
```

Here, `profDepth` is an array indicating the profile depths (meters below ground level); and the remaining arguments are arrays of the same length, which specify the material properties of the seismic layers. Specifically, `cp` is the compressional wave phase speed (m/s), `cs` is the shear wave phase speed (m/s), `qp` is the quality factor for compressional waves, `qs` is the quality factor for shear waves, and `rho` is the density (kg/m³). From these basic five profiles, other material profiles of interest in seismological calculations readily follow.

Alternatively, the profiles can be specified using layers of `SolidIsoLinear` objects (e.g., `SOIL` or `GRANITE`). The available constructors include

```
SeismicVertProf(SolidIsoLinear layer)
SeismicVertProf(double[] profDepth, SolidIsoLinear layer)
```

and

```
SeismicVertProf(double[] profDepth, SolidIsoLinear[] layers,
                double[] transDepth)
```

The first two of these constructors specify a single solid type, which is used throughout the entire subsurface. In the first version, default profile depths are used, whereas in the second, they are specified explicitly. Although the seismic profiles will not vary as a function of the depth, the evaluation points can impact numerical solvers used for seismic wave propagation. In the third constructor above, an array with material layer descriptions is passed along with an array giving the depths of the interface layers between the materials. The latter array should have a length one less than the number of layers. As an example, let us construct a subsurface with a 1 m thick layer of soil above granite. Referring to the listing

of `SolidIsoLinear` from section 5.2.2, `SOIL` is type 4 and `GRANITE` is type 10. Hence

```
profDepth = 0.2:0.2:5.0;
SolidIsoLinearTypes = javaMethod('values',
    'mil.army.usace.knee.envIRON.SolidIsoLinearTypes');
layers = javaArray('mil.army.usace.knee.envIRON.SolidIsoLinear', 2);
layers(1) = SolidIsoLinear(SolidIsoLinearTypes(4));
layers(2) = SolidIsoLinear(SolidIsoLinearTypes(10));
transDepth = 1.0;
svp = SeismicVertProf(profDepth, layers, transDepth);
```

Once the seismic profiles have been constructed, a number of getter methods are available to retrieve their properties, including profiles that are calculated from those specified. The following methods all return `double[]` arrays:

- `getProfDepth` returns the depths at which the profiles are specified (in meters, where the depth closest to the surface is first and then increases downward).
- `getDensityProf` returns the density profile.
- `getBulkModProf` returns the bulk modulus profile.
- `getShearModProf` returns the shear modulus profile.
- `getPWaveProf` returns the compressional wave speed profile.
- `getSWaveProf` returns the shear wave speed profile.
- `getPQProf` returns the profile of quality factors for compressional waves.
- `getSQProf` returns the profile of quality factors for shear waves.

For all of the preceding methods with a name ending in `Prof`, there is also a method where `Prof` is omitted, which will return the value near the surface. For example, `getDensity` returns the density at the uppermost profile point. The method `getLayer` takes an integer argument and returns a `SolidIsoLinear` object corresponding to the profiles at the corresponding profile depth. If the argument is omitted, `getLayer` returns the uppermost profile point. This can be very useful for obtaining other quantities of interest. For example, `getSeismicProf.getLayer.getRayleighSpeed` returns the Rayleigh wave speed at the surface.

5.3 Environmental representations

Having discussed the individual components of EASEE's environmental representations, we now put these pieces together into the full product. As mentioned in the introduction to this section, EASEE has two main environmental representations: homogeneous atmosphere and subsurface (`EnvironHomo`) and vertical profile atmosphere and subsurface (`EnvironVertProf`). These will now be described.

5.3.1 Homogeneous environment

The `EnvironHomo` class has a great variety of constructor methods, which enable the environmental properties to be specified by various approaches and to varying levels of complexity. The following is one of the simplest constructors, which is pertinent to a flat, homogeneous ground surface:

```
EnvironHomo(double groundHgt, HumidAir atmos, SolidIsoLinear
            subsurf)
```

The arguments are (1) the height of the ground in meters (usually relative to mean sea level); (2) an object representing a humid atmosphere (as discussed in section 5.2.1); and (3) a solid, isotropic, linear object representing the subsurface (as discussed in section 5.2.2). This constructor uses defaults for the land cover (`FASSTVegetationTypes.GRASS_SHORT`) and soil (`FASSTSoilTypes.SAND_CLAYEY`), which are assumed to apply throughout the domain.

The following constructor is similar but enables the land cover and soil to be specified explicitly:

```
EnvironHomo(double groundHgt, Landcover landCov, Soil soil,
            HumidAir atmos, SolidIsoLinear subsurf)
```

As described in sections 5.2.3 and 5.2.4, respectively, many options are available for specifying particular properties of the soil and land cover.

Consider the following slightly more complicated constructor for `EnvironHomo`:

```
EnvironHomo(GeoGridCart2DDouble dem, Landcover landCov,
            Soil soil, HumidAir atmos, SolidIsoLinear subsurf)
```

Compared to the previously considered version of the constructor, the initial argument is a digital elevation model (DEM), in the form of a `GeoGridCart2DDouble` object, which stores data on a 2-D geographic grid. The DEM is constructed as described in section 4.3.

Lastly, let us consider the following `EnvironHomo` constructor, which allows specification of land cover and soil that vary across the interface between the ground and air:

```
EnvironHomo(GeoGridCart2DDouble dem, LandcoverDecoder lcDecoder,
            GeoGridCart2DInt lcGrid, SoilDecoder soilDecoder,
            GeoGridCart2DInt soilGrid, HumidAir atmos,
            SolidIsoLinear subsurf)
```

Here, `lcDecoder` and `soilDecoder` are objects specifying the mapping of integer data to land cover and soil types, respectively, as described in section 4.3. After creating `lcGrid` and `soilGrid`, we would then construct the desired environment using the command

```
env = EnvironHomo(dem, NationalLandcoverDataset2001Types.
                OPEN_WATER, lcGrid, FASSTSoilTypes.GRAVEL_WELL_GRADED,
                soilGrid, atm, subSurf);
```

It is important to keep in mind that the specified land cover and soil decoders indicate only which decoding algorithm to use, rather than the actual land cover or soil types (those are specified by `lcGrid` and `soilGrid`, respectively). Though `NationalLandcoverDataset2001Types.OPEN_WATER` and `FASSTSoilTypes.GRAVEL_WELL_GRADED` were specified above, the decoder can be any valid *instance* of the Java class for representing the land cover or soil system. In this case, for the land cover, we could have alternatively used `NationalLandcoverDataset2001Types.OPEN_WATER`, `NationalLandcoverDataset2001Types.WETLANDS_WOODY`, or any other valid instance of `NationalLandcoverDataset2001Types`, as opposed to, say, an instance of `GeoCoverLCTypes`.

Once a homogeneous environment has been constructed, a snow layer and soil moisture may be additionally specified. A single value for the soil moisture fraction throughout the domain may be set using the `setSingleSoilMoisture` method. The moisture is a value between 0 and 1, although if a value is specified that exceeds the soil porosity (which can be

retrieved using `FASSTSoilTypes.getPorosity()`, the value will be reduced to the porosity (since the soil moisture fraction cannot exceed the porosity). To set the soil moisture to 0.2, for example, we would enter

```
env.setSingleSoilMoisture(0.2);
```

The `setMultiSoilMoisture` method specifies soil moisture that varies across the domain. Specifically, it takes as input a `GeoGridCart2DDouble` object, which stores the soil moisture on a geographic grid.

The snow layer includes both the snow type and depth. The `setSingleSnowCover` method specifies a snow cover that does not vary over the domain. It takes two arguments, snow depth and snow type. For example,

```
env.setSingleSnowCover(0.1, SnowTypes.SNOW_FRESH);
```

specifies 0.1 m of fresh snow throughout the domain. The `setMultiSnowCover` method is used to specify a spatially varying snow cover. It has the form

```
setMultiSnowCover(GeoGridCart2DDouble snowDepth, SoilDecoder  
    snowDecoder, GeoGridCart2DInt snowGrid)
```

where `snowDepth` is a geographic grid specifying the snow depth, `snowDecoder` is an instance of the decoder class for the snow (e.g., `SnowTypes.SNOW_FRESH`), and `snowGrid` is a grid of integer codes indicating the snow types.

The `EnvironHomo` class provides many getter methods that can be used to obtain and display the properties of an object. For example, the DEM is retrieved by entering

```
demOut = env.getDem();
```

The method `getMeanElev` returns the mean elevation for the DEM. The local terrain height can be retrieved at a particular coordinate by specifying a `GeoCoord` as the argument to the `getDem` method. That is,

```
disp(env.getDem(coord))
```

displays the terrain elevation at `coord`, where `coord` is an instance of `GeoCoord`. The following command displays the elevation at a distance 100 m north and 200 m east of the specified coordinate:

```
disp(env.getDem(GeoCoord(coord, 100.0, 200.0)))
```

The `getDsm` method functions similarly to `getDem`, except that it pertains to the digital surface map (DSM) rather than the DEM. Here, DSM is the elevation of the bare Earth plus the height of the objects (e.g., buildings and vegetation) on the surface (DEM generally is for the bare Earth.) The `getDsm` method without argument returns the entire DSM grid, `getDsm` with a coordinate returns the DSM at that coordinate, and `getMeanSurfElev` returns the mean for the DSM.

Similar getters are provided for the land cover (`getLandcover`), soil (`getSoil`), snow type (`getSnow`), snow depth (`getSnowDepth`), and soil moisture (`getSoilMoisture`). All of these methods can accept a coordinate as an argument, for which they return the value at that coordinate. However, the `getLandcover`, `getSoil`, and `getSnow` methods, when used without an argument, return the *most prevalent* land cover, soil, and snow type, respectively, since it would make no sense to return a mean value for these quantities. The `getLandcoverGrid`, `getSoilGrid`, and `getSnowGrid` methods return the entire grids. The `getSnowDepth` and `getSoilMoisture` methods, without arguments, return the entire grids, whereas `getMeanSnowDepth` and `getMeanSoilMoisture` return the means.

A variety of getters are also available to retrieve the atmospheric and sub-surface properties. In this regard, it should be kept in mind that, for the `EnvironHomo` class, the atmospheric and subsurface properties are constant throughout the domain.

The atmospheric properties are retrieved with the `getAtmos` method, followed by a getter for the desired quantity. To retrieve the temperature in degrees Celsius and store the result in the variable `temp`, one would enter

```
temp = env.getAtmos().getTemp();
```

Similarly, instead of `getTemp`, one could use `getTempKelvin` to get the temperature in Kelvin and `getTempFahr` to get the temperature in degrees Fahrenheit. Setting

```
hum = env.getAtmos().getSpecHum();
```

retrieves the specific humidity, whereas `getRelHum`, `getMolHum`, and `getMixRat` return the relative humidity, molar humidity, and mixing ratio, respectively. The `getDensity` method returns the air density, `getPress` returns pressure in Pascals, and `getPressMbar` returns the pressure in millibar.

The subsurface properties are retrieved with the `getSubSurf` method, followed by the desired quantity. In particular,

- `getDensity` returns the density,
- `getBulkMod` returns the bulk modulus,
- `getPWaveSpeed` returns the compressional (P-) wave speed,
- `getSWaveSpeed` returns the shear (S-) wave speed,
- `getRayleighSpeed` returns the Rayleigh wave speed,
- `getPQualFac` returns the quality factor for P-waves, and
- `getSQualFac` returns the quality factor for S-waves.

5.3.2 Vertical profile environment

The `EnvironVertProf` class represents a horizontally stratified environment, that is, an environment in which the atmospheric and subsurface profiles depend on the vertical coordinate only. Internally, `EnvironVertProf` is the same as `EnvironHomo`, except that the homogeneous atmosphere and subsurface representations are replaced by representations of the classes `AtmosOneDim` and `SubSurfOneDim`. So, most of the description in the previous section regarding `EnvironHomo` still applies to `EnvironVertProf`, except regarding setting of the properties of the air and subsurface and retrieval of data using the `getAtmos` and `getSubSurf` methods.

The `AtmosOneDim` class includes the vertical profiles in the atmosphere, specifically, the temperature, specific humidity, pressure, wind speed, and wind direction, which are encapsulated within an `AtmosVertProf` object (section 5.2.7). `AtmosOneDim` also includes a representation of the atmospheric surface layer (by an `AtmosSurfLayer` object, section 5.2.6) and cloud layers (by a `Clouds` object, section 5.2.8). Internally, the `SubSurfOneDim` contains the vertical density profile and the seismic vertical profiles of compressional (P-) wave speed, shear (S-) wave speed, and quality factors for P- and S-waves, as represented by a `SeismicVertProf` object (section 5.2.9).

Here are three of the most basic and flexible constructor methods for the vertical profile environment:

```
EnvironVertProf(double groundHgt, AtmosOneDim atmos,
                Landcover landCov, Soil soil, SeismicVertProf seis)
```

```
EnvironVertProf(AtmosOneDim atmos, Landcover landCov, Soil soil,
                GeoGridCart2DDouble dem, SeismicVertProf seis)
```

and

```
EnvironVertProf(AtmosOneDim atmos, GeoGridCart2DDouble dem,
                LandcoverDecoder lcDecoder, GeoGridCart2DInt landcover,
                SoilDecoder soilDecoder, GeoGridCart2DInt soil,
                SeismicVertProf seis)
```

The first of these constructors assumes flat ground and no terrain. The second and third allow the DEM to be explicitly specified. Furthermore, the first and second constructors apply to an environmental model in which the land cover and soil properties are the same throughout the domain; the third allows them to vary horizontally. Construction of the DEM, land cover, and soil objects appearing in these constructors follows from the discussion in the previous subsections. The new aspect of these constructors is the `AtmosOneDim` object. (Because the `SubSurfOneDim` is basically just a wrapper around the `SeismicVertProf` class, the constructors take the `SeismicVertProf` object directly.)

`AtmosOneDim` has a rich variety of constructors, which enables the atmospheric profiles to be specified directly by certain benchmark cases or by MOST. These constructors are often quite similar to `AtmosVertProf`; however, using the `AtmosOneDim` constructor is preferable in such cases to ensure that the `AtmosSurfLayer` and `Clouds` objects are constructed in a manner that is compatible with the vertical profiles.

The following two constructors specify the vertical profiles and clouds explicitly while inferring information about the surface layer:

```
AtmosOneDim(AtmosVertProf atmos, double frictionVel, double
            sensibleHeatFlux, double latentHeatFlux, double roughHgt,
            double dispHgt, Clouds clouds)
```

or

```
AtmosOneDim(AtmosVertProf atmos, double roughHgt, double dispHgt,
            Clouds clouds)
```

For both forms, one would ordinarily construct the profiles using one of the methods in section 5.2.7 and the clouds using one of the methods in section 5.2.8. The first form specifies the vertical profiles directly as an `AtmosVertProf` object and builds the surface layer from the specified friction velocity (m/s), sensible heat flux (W/m²), latent heat flux (W/m²), roughness height (m), and displacement height (m). The lowermost height in the profiles is used to determine the surface temperature and humidity. The second form is similar but infers the friction velocity and heat fluxes by applying MOST to the *two* lowermost heights of the profiles.

The following constructor yields an atmosphere based solely on profiles constructed from MOST:

```
AtmosOneDim(double[] profHgt, AtmosSurfLayer asl,
            double roughHgt, double dispHgt, Clouds clouds)
```

Here, `AtmosSurfLayer` is the class for representing atmospheric surface layers, as described in section 5.2.6. The arguments `roughHgt` and `dispHgt` are the surface roughness and displacement heights, respectively. They may be retrieved from the land cover by using the `getRoughHgt` and `getDispHgt` methods.

`AtmosOneDim` also has a constructor for benchmark profiles, which is of the same form as that for `AtmosVertProf` (section 5.2.7), namely

```
AtmosOneDim(double[] profHgt, AtmosVertProf.BenchmarkProfileTypes
            profType)
```

The previous section discussed a number of getter methods with regard to the `EnvironHomo` class that enable examination of the properties of the environment. Those methods generally apply to `EnvironVertProf` as well, with the main exceptions being `getAtmos` and `getSubSurf` (methods return objects representing the 1-D atmosphere and subsurface, respec-

tively, as opposed to homogeneous representations). Some methods specific to `EnvironVertProf` are also defined in this class, namely `getAtmosSL`, `getAtmosProf`, `getClouds`, and `getSeismicProf`, which return the atmospheric surface layer, atmospheric profiles, clouds, and seismic profiles, respectively. One can then append additional getter methods to obtain the quantity of interest. For example, if `env` is an object of the `EnvironVertProf` class, `env.getAtmosSL.getFrictionVel` would return the friction velocity.

5.4 Loading WRF Data

As described in section 5.3.2, EASEE supports a great many methods to construct vertical profile environments. However, these all assume that the user has already obtained information on the atmosphere from other sources. We previously circumvented the question of what that source might be. In this subsection, we focus on a method that enables EASEE to ingest a particular source of atmospheric data, namely output from the Weather Research and Forecasting (WRF) weather model.

The WRF model (with the Advanced Research WRF [ARW] solver) is a regional weather model that solves fully compressible, Eulerian, nonhydrostatic equations (Skamarock et al. 2008) and can produce atmospheric forecasts at a variety of horizontal, vertical, and temporal resolutions. It also includes many parameterization options for planetary boundary-layer physics, surface physics, and other atmospheric domains. Because WRF is a community model, it is free and publicly available at the WRF Model Users' Page (NCAR 2018) and is used widely by the atmospheric community, with over 39,000 users in over 160 countries (NCAR 2017). Overall, WRF's versatility and large user base make it desirable as a source of atmospheric data for EASEE.

EASEE supports the usage of either (1) a single WRF output file or (2) an *ensemble* of WRF output files (i.e., a set of model output files from nearly identical simulations differing only in the initial or boundary conditions). The following two subsections describe the procedures for working with single or ensemble forecast files, respectively.

5.4.1 Single WRF Output File

To use data from the WRF output file's native format—NetCDF (Network Common Data Format)—EASEE uses methods from the NetCDF-Java API

v4.6. This API requires the creation of a Java-NetCDF object (`NetcdfFile`) from the WRF output file that can be readily used within EASEE. To do this, first open the WRF output file in MATLAB:

```
ncfile = WrfLoader.openNetcdf(fileLocation);
```

where `fileLocation` is the path to the desired WRF output file and `ncfile` is the `NetcdfFile` object derived from the WRF output file that can be used within EASEE.

Generally, users will provide their own WRF files for use within EASEE. However, there are also some example WRF files that can be accessed through the online GitLab repository that contains EASEE; KNEE can be cloned from this repository, and the WRF test files can then be obtained. Please contact the authors for more information about accessing the GitLab repository.

If you would like to use the single WRF test file and you have cloned the KNEE portion of the EASEE GitLab repository, then the test file can be found at the following file path:

```
\path\to\KNEE\KNEE\src\test\resources\TestWeatherFiles\WRF\  
wrfout_d01_2017-01-01_00_00_00
```

where `\path\to\KNEE\` is the path to the EASEE/KNEE directory on your local machine. The test file specified by the above path has 17 time steps, with output at 3-hour intervals over 2 model days (from 2017.01.01 at 0000 UTC [Coordinated Universal Time] to 2017.01.03 at 0000 UTC). The simulation domain has a 5 km horizontal resolution and 29 vertical model layers (model top is at 100 hPa [10 kPa]). The domain is centered at 40.°N, 75.°W and spans 150 grid cells west–east and 120 grid cells south–north.

If you would like to use your own WRF output file instead, or if you receive the above test file through means other than cloning the GitLab repository, simply substitute the appropriate path to the file for `fileLocation`.

For the case of a single file, the `WrfLoader` class is used to construct an object that extracts weather data from a WRF output file. `WrfLoader` extends the `AtmosOneDim` class and thus comes preequipped with default versions of

`AtmosVertProf`, `Clouds`, and `AtmosSurfLayer` objects. Specifying this constructor updates these default objects by using data from the WRF output file, allowing a more accurate `AtmosOneDim` object to be created.

There is one main constructor available for `WrfLoader`:

```
WrfLoader(NetcdfFile ncfile, int nest, double lat, double lon,
          int timeIndex, double roughHgt, double dispHgt)
```

This constructor requires a WRF output file and the approximate location in latitude (-90° to 90°) and longitude (-180° to 180°) at which you would like to extract a vertical atmospheric profile. The latitude/longitude values are specified in decimal degrees. Alternatively, a variation of the constructor allows you to instead specify the latitude and longitude indices (e.g., 1, 2, 3, . . .) corresponding to the grid cell numbers in the south–north and west–east directions, respectively. WRF simulations can be conducted with multiple nested horizontal domains (e.g., with a parent domain at 9 km horizontal resolution and a smaller, nested domain at a finer 3 km horizontal resolution). The variable `nest` refers to the horizontal model domain in which you would like to sample a vertical profile (i.e., the parent domain would be 1, the next nested domain would be 2, etc.). (Note that `nest = 1` for a WRF simulation with a single domain, such as the aforementioned WRF test file). Finally, `roughHgt` and `dispHgt` represent the roughness height (m) and displacement height (m), respectively.

The following is an example of loading data from the single WRF test file into EASEE and plotting vertical profiles of air temperature and wind.

First, import KNEE's `environ` and `geo` packages:

```
import mil.army.usace.knee.environ.*;
import mil.army.usace.knee.geo.*;
```

Next, open the WRF test file using its path on your local machine:

```
ncfile = WrfLoader.openNetcdf('path\to\KNEE\KNEE\src\test\
resources\TestWeatherFiles\WRF\wrfout_d01_2017-01-
01_00_00_00');
```

where `\path\to\KNEE\` is the path to the EASEE/KNEE directory on your local machine.

At this point, you can test whether your chosen latitude and longitude values (e.g., 40, -75) fit within the estimated boundaries of the domain. This can be accomplished by running `WrfLoader.getWrfLatLonIndices(ncfile, 40, -75)`. You must adjust your latitude/longitude values until the last line of output from this command reads, for example, `Closest lat/lon indices: (59, 74)`, where 59 and 74 are the vertical and horizontal positions of the closest grid cell, respectively.

Then, use the `WrfLoader` constructor to initiate an object with WRF data:

```
wrfFile = WrfLoader(ncfile, 1, 40, -75, 1, 0.1, 0.2);
```

where 1 refers to the chosen domain within the WRF simulation (only one domain in this file, so `nest = 1`), 40 (40°N) is the chosen latitude, -75 (75°W) is the chosen longitude, 1 is the second time step (0-based Java indices), 0.1 is the roughness height in meters (typical for grassland), and 0.2 is the displacement height in meters (estimated as twice the roughness height).

The following output should be generated after running the `WrfLoader` constructor:

```
Boundaries of the model domain: [36.98319625854492,
    42.858577728271484, -79.8861083984375, -70.49801635742188]
Specified lat/lon: (40.0, -75.0)
Closest lat/lon: (40.000, -75.000)
Closest lat/lon indices: (59, 74)
```

Additionally, typing `wrfFile` at the MATLAB command prompt will print summaries of the objects for the atmospheric surface layer, atmospheric vertical profiles, and clouds:

```
wrfFile =
    Atmospheric surface layer: friction velocity 0.695 m/s,
    sensible heat flux -46.446 W/m^2, latent heat flux 0 W/m^2, wind
    direction 216.824 deg (cw from N)
```

Atmospheric vertical profiles: 29 profile points, inversion height 732.0818481445312 m

Low cloud layer: (coverage = 0%, base height = 2.0 (km)),
mid cloud layer: (coverage = 0%, base height = 2.0 (km)), high
cloud layer: (coverage = 0%, base height = 2.0 (km)), aerosol =
AEROSOL_RURAL

Now that the WRF data have been successfully imported, use methods from EASEE to extract the vertical profiles of temperature and wind, and then use MATLAB commands to plot them:

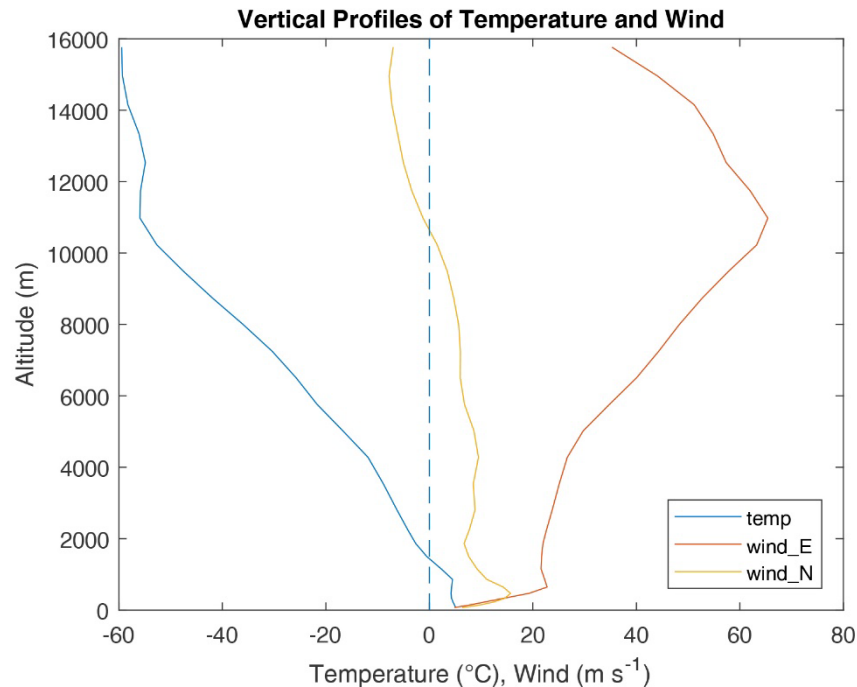
```
atm = wrfFile.getAtmosProf; % Atmospheric vertical profile
    object
profhgts = atm.getProfHgt; % Vertical profile heights
proftemps = atm.getTempProf; % Air temperature
profwinde = atm.getWindProfE; % Eastward component of wind
profwindn = atm.getWindProfN; % Northward component of wind
plot(proftemps, profhgts, profwinde, profhgts, profwindn,
     profhgts)
line([0 0], ylim, 'LineStyle','--'); % vertical ref line at
    temp/wind = 0
legend('temp','wind\_E','wind\_N','Location','southeast')
xlabel(['Temperature (' 176 'C), Wind (m s-1)']) % 176 is the
    character code for the "degrees" symbol
ylabel('Altitude (m)')
title('Vertical Profiles of Temperature and Wind')
```

The above block of code should produce Figure 5.

Alternatively, you can plot wind speed and/or wind direction. Simply substitute the following lines into the larger chunk of code above and adjust the variable names and axis labels:

```
profwindspeed = profhgts.getWindSpeedProf;
profwinddir = profhgts.getWindDirMetProf; % direction from which
    wind is blowing in degrees (0-360), starting at N then
    clockwise
```

Figure 5. Vertical profiles of air temperature (*blue*), the eastward wind component (*red*), and the northward wind component (*orange*) derived from a WRF test file in the EASEE repository.



5.4.2 Multiple WRF output files

If you would like to instead use multiple WRF output files (i.e., a forecast ensemble), then first identify the path of each WRF output file, and open the files in MATLAB using the following commands:

```
ncfile1 = WrfLoader.openNetcdf(fileLocation1);
ncfile2 = WrfLoader.openNetcdf(fileLocation2);
ncfile3 = WrfLoader.openNetcdf(fileLocation3);

...ncfile"n" = WrfLoader.openNetcdf(fileLocation''n'');
```

where `fileLocation1,2,...,n` are the paths to the desired WRF output files and `ncfile1,2,...,n` are the `NetcdfFile` objects derived from the WRF output files that can be used within EASEE.

As with the single WRF output file case, users will generally provide their own WRF files for use within EASEE. However, there are also some example WRF files within the EASEE repository that can be used for testing purposes.

If you would like to use the multiple WRF test files and you have cloned the KNEE portion of the EASEE GitLab repository, then the test files can be found at the following file paths:

```
\path\to\KNEE\KNEE\src\test\resources\TestWeatherFiles\WRF\
wrfout_d01_2017-01-01_00_00_00_file#
```

where `\path\to\KNEE\` is the path to the EASEE/KNEE directory on your local machine and where `#` refers to an integer from 1 to 3. The three test WRF files located at the above file paths are “pseudoensemble” members (i.e., they are different temporal subsets of the original file, `wrfout_d01_2017-01-01_00_00_00`).

The test files specified by the above path each have four time steps, with each file containing output at 3-hour intervals that sequentially cover 1.5 model days (i.e., file1 is 2017.01.01 at 0000, 0300, 0600, and 0900 UTC; file2 is 2017.01.01 at 1200, 1500, 1800, and 2100 UTC; etc.). All of the other domain metadata are the same as those for the original `wrfout_d01_2017-01-01_00_00_00` file from the “Single WRF Output File” section.

If you would like to use your own WRF output files instead, or if you receive the above test files through means other than cloning the GitLab repository, simply substitute the appropriate paths to the files for `fileLocation1,2,...n`.

After the WRF output files have been opened, you can use the `WrfEnsemble` constructor, which constructs an environmental ensemble from a list of WRF output files:

```
WrfEnsemble(List<NetcdfFile> ncfile, int nest, int latIndex, int
lonIndex, int[] timeIndex, EnvironVertProf env)
```

This constructor is similar to the `WrfLoader` constructor in that WRF output data are specified along with the location at which vertical atmospheric profiles should be extracted. However, the main differences are that `ncfile` in this constructor consists of multiple `NetcdfFile` objects, the `timeIndex` field is an array of time steps rather than a single time step, and a default `EnvironVertProf` object is specified instead of other component parts such as `roughHgt`. Furthermore, this constructor formulation re-

quires latitude and longitude indices corresponding to the grid cell numbers in the south–north and west–east directions, respectively, rather than exact values of latitude and longitude. These are obtained using a method called `getWrfLatLonIndices`, which is described later in this section.

Similar to the `WrfLoader` constructor, `WrfEnsemble` extends the `EnvironModelEnsemble` class and thus contains a default ensemble (`EnvironEnsemble`) of `EnvironVertProf` objects. Specifying this constructor updates the component parts of these default `EnvironVertProf` objects using data from the WRF output files, allowing more accurate `EnvironVertProf` objects to be created.

The following is an example of loading data from multiple WRF test files into EASEE and plotting vertical profiles of air temperature from each “ensemble” member (noting that the three files are actually “pseudoensembles” derived from the same original simulation).

First, import KNEE’s `environ` and `geo` packages and Java’s `ArrayList` package using the following commands:

```
import mil.army.usace.knee.environ.*;
import mil.army.usace.knee.geo.*;
import java.util.ArrayList;
```

Next, open the three WRF test files using their paths on your local machine:

```
ncfile1 = WrfLoader.openNetcdf('\path\to\KNEE\KNEE\src\test\
    resources\TestWeatherFiles\WRF\wrfout_d01_2017-01-01_00_
    00_00_file1');
ncfile2 = WrfLoader.openNetcdf('\path\to\KNEE\KNEE\src\test\
    resources\TestWeatherFiles\WRF\wrfout_d01_2017-01-01_00_
    00_00_file2');
ncfile3 = WrfLoader.openNetcdf('\path\to\KNEE\KNEE\src\test\
    resources\TestWeatherFiles\WRF\wrfout_d01_2017-01-
    01_00_00_00_file3');
```

where `\path\to\KNEE\` is the path to the EASEE/KNEE directory on your local machine.

Then, create a Java `ArrayList` to hold the multiple WRF files. The WRF files are added to the `ArrayList` one at a time:

```
fileList = ArrayList;  
fileList.add(ncfile1);  
fileList.add(ncfile2);  
fileList.add(ncfile3);
```

The `WrfEnsemble` constructor requires several parameters that must be created before the constructor is used. First, instantiate a default `EnvironVertProf` object:

```
env = EnvironVertProf();
```

Next, you can use an EASEE method called `getWrfLatLonIndices` to convert your chosen latitude and longitude values to integer indices on a grid:

```
latlon = WrfLoader.getWrfLatLonIndices(ncfile1, 40, -75);  
    % latlon contains two elements:  a latitude index and a  
    longitude index
```

Note that the output from the above command may indicate that your chosen latitude/longitude values are not within the estimated bounds of the simulation domain. You must adjust your latitude/longitude values until the last line of output from this command reads, for example, `Closest lat/lon indices: (59, 74)`.

Finally, determine the number of time steps available in the WRF files so that you do not accidentally use time indices that are out of range:

```
timesize = WrfLoader.getTimeSize(ncfile1);
```

If you type `timesize` into the MATLAB command prompt, you will see that there are four time steps in each WRF file. Thus, your Java-based time indices must be within the range `[0, 3]`, inclusive.

Now you can use the `WrfEnsemble` constructor with the above parameters:

```
wrfFiles = WrfEnsemble.makeEnvironEnsemble(fileList, 1,  
    latlon(1), latlon(2), 0:2, env);
```

where 1 refers to the chosen domain within the WRF simulations (only one domain in these files, so `nest = 1`); `latlon(1)` and `latlon(2)` are the grid indices of latitude and longitude, respectively; and 0:2 are indices of the time steps (first through third) to be sampled from the original files.

From this `WrfEnsemble` object, you can extract separate ensemble members:

```
wrfEnsMem1 = wrfFiles.getEnsembleMember(0);
wrfEnsMem2 = wrfFiles.getEnsembleMember(1);
wrfEnsMem3 = wrfFiles.getEnsembleMember(2);
```

The three `wrfEnsMem` variables above represent the three ensemble members within the `WrfEnsemble` set of simulations.

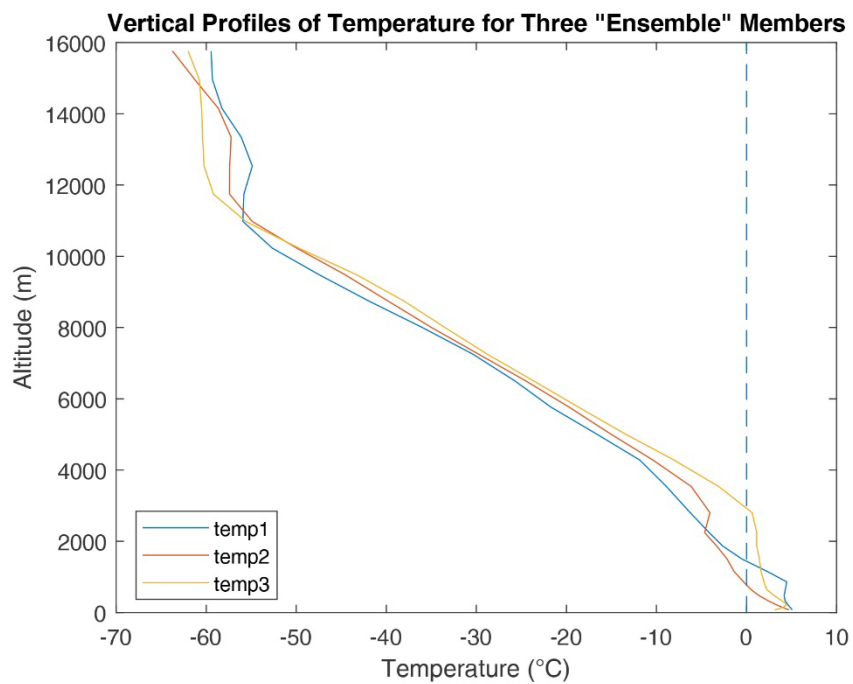
After the individual files have been extracted, it is easier to analyze the WRF data. Now you can plot temperature profiles for each ensemble member:

```
wrfEnsMem1_t2 = wrfEnsMem1.get(1); % Extracts WRF data from the
    first ensemble member ("Mem1"), second time step
wrfEnsMem2_t2 = wrfEnsMem2.get(1);
wrfEnsMem3_t2 = wrfEnsMem3.get(1);
wrfEnsAtm1 = wrfEnsMem1_t2.getAtmosProf; % Atmospheric vertical
    profile object
wrfEnsAtm2 = wrfEnsMem2_t2.getAtmosProf;
wrfEnsAtm3 = wrfEnsMem3_t2.getAtmosProf;
profhgts = wrfEnsAtm1.getProfHgt; % Same profile heights can be
    used for all ensemble members
proftemps1 = wrfEnsAtm1.getTempProf; % Air temperature
proftemps2 = wrfEnsAtm2.getTempProf;
proftemps3 = wrfEnsAtm3.getTempProf;
plot(proftemps1, profhgts, proftemps2, profhgts, proftemps3,
    profhgts)
line([0 0], ylim, 'LineStyle','--'); % Vertical reference line
    at temp = 0
legend('temp1','temp2','temp3','Location','southwest')
xlabel(['Temperature (' 176 'C)']) % 176 is the character code
    for the "degrees" symbol
ylabel('Altitude (m)')
```

```
title('Vertical Profiles of Temperature for Three "Ensemble"  
Members')
```

The preceding block of code should produce Figure 6.

Figure 6. Vertical profiles of air temperature derived from three pseudoensemble test WRF files in the EASEE repository.



6 Acoustic Propagation Calculations

KNEE's `acoustic` package provides a number of data representations and basic calculations related to acoustics. These include, for example, calculation of air absorption, porous media models for ground impedance, and calculation of the sound field above an impedance ground. Assuming the `StartEASEE.m` script has been run successfully (section 3.1), the KNEE `acoustic` package can be imported into MATLAB with the following command:

```
import mil.army.usace.knee.acoustic.*;
```

More advanced and militarily sensitive capabilities, such as the National Aeronautics and Space Administration (NASA) Green's Function Parabolic Equation (GFPE) method, are found in the `EASEELib acoustic` package, which is imported with the following command:

```
import mil.army.usace.easee.acoustic.*;
```

We begin this section with a discussion of several useful preliminaries: namely, how to construct standard octave and one-third octave bands, how to construct acoustic media for air and porous media, and how to set up calculation grids to store the model output. We will then consider several particular propagation models. Lastly, we will discuss an alternative approach to performing acoustical calculations, which involves constructing the environmental properties using EASEE and then performing the actual propagation calculations in MATLAB.

6.1 Standard acoustic frequencies

KNEE's `spectra` package contains Java classes for representing and manipulating power spectra. Although we will not describe the functionality of this package in detail here, we do describe one particularly useful feature, namely the availability of standard frequencies for octave and one-third octave bands. To access these frequencies, first import the `BandedPowerLawSpec` class:

```
import mil.army.usace.knee.spectra.BandedPowerLawSpec;
```

Then, to retrieve standard octave band center frequencies, for example, enter

```
freq = BandedPowerLawSpec.STANDARD_OCTAVE_CENTER_FREQS;
```

This will create a MATLAB array containing all of the standard frequencies. To retrieve standard one-third octave-band center frequencies, enter

```
freq = BandedPowerLawSpec.STANDARD_ONE_THIRD_OCTAVE_CENTER_FREQS;
```

Similarly, the `STANDARD_OCTAVE_LOWER_FREQS` and `STANDARD_OCTAVE_UPPER_FREQS` provide the lower and upper frequency bounds for the standard octave bands, whereas `STANDARD_ONE_THIRD_OCTAVE_LOWER_FREQS` and `STANDARD_ONE_THIRD_OCTAVE_UPPER_FREQS` provide the analogous results for one-third octave bands.

6.2 AcousticMedium class

The `AcousticMedium` class plays an important role in EASEE's implementation of acoustics by enabling the construction of the acoustical properties (impedance, complex wave number, attenuation, phase speed, etc.) of media. In particular, EASEE's models for the acoustical properties of air and for porous media are extensions of this class.

To use the acoustic medium class, you will first need to import the Apache Commons complex-number class by typing

```
import org.apache.commons.math3.complex.Complex;
```

This class is needed because Java, unlike MATLAB, does not directly support complex numbers. The `AcousticMedium` class defines methods to retrieve

- the characteristic impedance (`getImped` and `getNormImped`, for the unnormalized impedance and the impedance normalized by the reference value of $\rho_0 c_0$, where ρ_0 is the reference density and c_0 is the reference sound speed);
- the characteristic admittance (`getAdmit` and `getNormAdmit`);
- the characteristic resistance (`getResist` and `getNormResist`);
- the characteristic reactance (`getReact` and `getNormReact`);
- the complex wavenumber (`getWaveNumber` and `getNormWavenumber`, the latter being normalized by $2\pi f/c_0$, where f is the acoustic frequency);
- the phase speed (`getPhaseSpeed`);

- the attenuation coefficient in Np/m (`getAtten`);
- the complex bulk modulus (`getCompBulkMod`); and
- the complex density (`getCompDensity`).

The methods for the impedance, admittance, complex wavenumber, complex bulk modulus, and complex density return complex values of the Apache complex number type. The methods for resistance, reactance, phase speed, and attenuation return real values, which can be more convenient to manipulate in MATLAB.

All of the previously described methods can accept either a single value for the frequency or an array of values. The output would correspondingly be a single value or an array of values providing results for each specified frequency.

The `AirMedium` class is an extension of `AcousticMedium`, which provides the acoustical properties of air at a particular temperature, humidity, and pressure. In this regard, `AirMedium` is complementary to the `HumidAir` class in the `environ` package, which was discussed in section 5.2.1. In fact, one can construct an `AirMedium` object either by specifying the air properties directly or by passing a `HumidAir` object:

```
air = AirMedium(19.1, 0.01, 1e5); % air at 19.1 C, 0.01 spec
    hum, 10^5 Pa
```

or

```
air = AirMedium(HumidAir(19.1, 0.01, 1e5));
```

The two preceding constructors produce equivalent results. The pressure can be omitted from either one, in which case sea-level pressure is assumed. With the second form, any of the `HumidAir` constructors described earlier in this report may be used.

Once the `AirMedium` has been constructed, we can call the methods previously described for the `AcousticMedium` class. For example, the following code creates a plot of attenuation between the frequencies of 20 Hz and 20 kHz for air at 18.1°C, 40% relative humidity, and sea-level pressure:

```
air = AirMedium(HumidAir(18.1, int32(1), 40.0));
freq = logspace(log10(20), log10(20000));
```

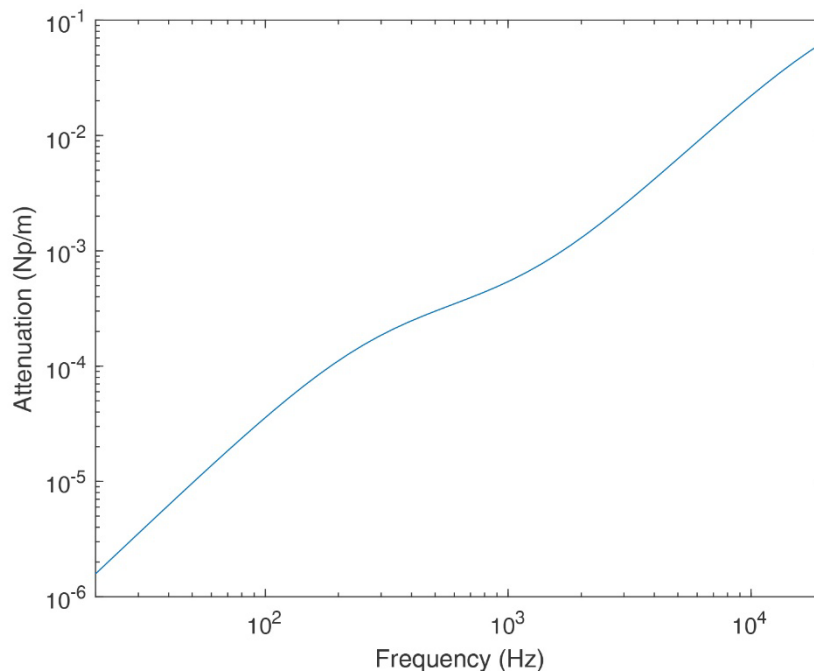
```

loglog(freq, air.getAtten(freq))
xlabel('Frequency (Hz)')
ylabel('Attenuation (Np/m)') % 1 Np = 8.69 dB
xlim([20 20000])

```

Figure 7 shows the result.

Figure 7. Attenuation coefficient for air at 20°C, 40% relative humidity, and sea-level pressure.



The `AcousticMedium` class also serves as the basis for EASEE's representations of the acoustical properties of porous media. The models currently supported are Attenborough's four-parameter model (`AttenboroughFourParamMedium`), the Delany-Bazley model (`DelanyBazleyMedium`), the relaxation model (`RelaxMedium`), and the Zwikker-Kosten model (`ZwikkerKostenMedium`).

As an example, let us consider the Zwikker-Kosten model, which represents the porous medium using three parameters: static flow resistivity, porosity (void fraction), and the so-called structure constant, which is related to pore tortuosity. Two primary constructors are provided:

```
ZwikkerKostenMedium(double por, double flowres)
```

and

```
ZwickerKostenMedium(double ks, double por, double flowres)
```

The first of these constructors accepts the porosity (a fraction between 0 and 1) and static flow resistivity and then estimates the structure constant based on the porosity. The second allows all three parameters to be set directly.

Information on the other porous models and their constructors can be found in the Javadoc or by directly examining the Java code.

The `BenchmarkMedium` class, which extends `AcousticMedium`, sets the ground properties as appropriate to certain acoustical benchmark calculations found in Attenborough et al. (1995) and Di and Gilbert (1993).

6.3 Calculation grids

EASEE calculations are performed on a grid (array) of specified source and receiver positions. The structure of the grid depends on the underlying symmetry of the underlying propagation calculation. Two main types of structured grids are of interest for running the acoustic propagation models: *vertical* and *polar*. The vertical grid is used for models for which the output depends on source height, receiver height, and horizontal range between the source and receiver. The polar grid introduces a dependence on the azimuth (e.g., the direction relative to the wind). For both the vertical and polar grids, the horizontal source position is assumed to be zero (meters for the vertical grid, degrees for the polar grid).

Let us consider an example for setting up a vertical grid. In this example, the grid has a single source height of 5.0 m and a single receiver height of 1.0 m, and the range varies from 0 m to 1000 m in increments of 10 m:

```
import mil.army.usace.knee.grids.*;
srcHgt = 5.0;
rcvHgt = 1.0;
range = [0.0:10.0:1000.0];
tgv = TransmitGridVert(srcHgt, rcvHgt, range);
```

The polar grid is constructed with one additional argument, representing the azimuth in radians. By convention, the azimuth is bounded between $-\pi$ and π . The following code sets up a polar grid with 16 evenly spaced angles:


```
azi = linspace(-pi, pi-pi/8, 16);
tgp = TransmitGridPolar(srcHgt, rcvHgt, range, azi);
```

6.4 Impedance-plane model

6.4.1 Flat ground

The impedance-plane propagation model, implemented in the KNEE package by the Java class `mil.army.usace.easee.knee.ImpedancePlaneModel`, solves for the sound field above a flat, impedance boundary (Attenborough et al. 1980). The atmosphere above the plane is assumed to be a homogeneous half-space, represented using the `AirMedium` class.

As discussed in the Background section, each model in EASEE has an associated Java class (or classes) defining the parameters needed by that model. A propagation model is created by first constructing the parameters and then constructing the propagation model using these parameters. Consider, for example, the following code:

```
params = ImpedancePlaneParamsHomo;
impModel = ImpedancePlaneModel(params);
```

or, more simply,

```
impModel = ImpedancePlaneModel(ImpedancePlaneParamsHomo);
```

Here we have constructed the parameters based on default values, which in this case correspond to impedance typical of grass-covered soil and dry air at room temperature and sea-level pressure. The `Homo` at the end of the `ImpedancePlaneParamsHomo` signifies that the parameters are constructed for a homogeneous environment (`EnvironHomo`). The impedance-plane model also has a parameter class called `ImpedancePlaneParamsVertProf`, which is for a vertical profile environment.

The following constructor for the impedance-plane model parameters passes the properties for the air and ground layers explicitly:

```
ImpedancePlaneParamsHomo(AcousticMedium refAir, AcousticMedium
    ground)
```

Any valid `AcousticMedium` may be used for both the air and ground. However, only the phase speed, density, and attenuation of the air layer and the specific impedance of the ground layer will impact calculations made with the impedance-plane model. `AirMedium` is a nonabstract subclass of `AcousticMedium` and is generally used for the air layer `refAir`. Similarly, one of the porous-media models mentioned in section 6.2 (such as `ZwikkerKostenMedium`) is used for the ground.

The following constructor explicitly passes an `EnvironHomo` object, which may be constructed using the procedures described in section 5.2:

```
ImpedancePlaneParamsHomo(EnvironHomo scene)
```

In this version of the constructor, the surface temperature, humidity, and pressure values from this environment are used to construct the air properties, whereas the soil properties are used to construct the ground impedance. All other information in the `EnvironHomo` object is ignored.

Another useful approach to constructing parameters for the impedance-plane model uses predefined acoustical ground types and benchmark cases. The following code lists the available definitions:

```
modelTypes = ImpedancePlaneModelTypes.values;
for m=1:length(modelTypes), disp(modelTypes(m)); end
```

This results in

```
FREE_SPACE
RIGID
RELEASE
AIR
BENCHMARK_10_HZ
BENCHMARK_100_HZ
BENCHMARK_1000_HZ
DG_BENCHMARK_100_HZ
DG_BENCHMARK_200_HZ
DG_BENCHMARK_500_HZ
DG_BENCHMARK_1000_HZ
ASPHALT
FOREST
```

```
GRASS
GRAVEL
ICE
SAND
SNOW
WATER
```

Hence, to construct a model for a grass-covered ground surface, we would enter

```
params = ImpedancePlaneParamsHomo(modelTypes(14));
impModel = ImpedancePlaneModel(params);
```

In the previous listing of predefined acoustical ground types, the types starting with `BENCHMARK_` are from Attenborough et al. (1995), whereas those starting with `DG_BENCHMARK_` are from Di and Gilbert (1993).

The basic procedure for running the propagation model is to create a structured grid, which specifies the geometry of the problem, and then call the calculation using this grid and other input parameters, such as the acoustic frequency. Assuming a vertical grid has been constructed as described in section 6.3, we calculate and display the transmission loss using

```
freq = 100.0;
pmag2 = impModel.calcTransGridStruct(freq, tgv);
transLossGrass = 10*log10(pmag2);
plot(range, transLossGrass)
```

The key to the preceding code is the call to the `calcTransGridStruct` method, which takes two arguments, namely the frequency and the transmission grid, and returns the sound field for receiver positions on the specified grid. Note that the output of EASEE acoustical calculations is the *squared magnitude* of the pressure, normalized by the pressure that would be observed at 1 m in free space. Hence, we convert the output to decibels by taking ten times the base-ten algorithm. This quantity is often called the transmission loss, or TL. (Strictly speaking, it would make more sense to call the *negative* of this quantity the transmission loss, since a loss would then be positive. The TL as defined here might more properly be called a transmission *gain*. Nonetheless, this quantity is conventionally called the transmission loss.)

To rerun the previous calculation with a rigid ground surface, we would enter

```
params = ImpedancePlaneParamsHomo(modelTypes(2));
impModel = ImpedancePlaneModel(params);
transLossRigid = 10*log10(impModel.calcTransGridStruct(freq, tgv));
```

And to rerun the calculation for snow,

```
params = ImpedancePlaneParamsHomo(modelTypes(18));
impModel = ImpedancePlaneModel(params);
transLossSnow = 10*log10(impModel.calcTransGridStruct(freq, tgv));
```

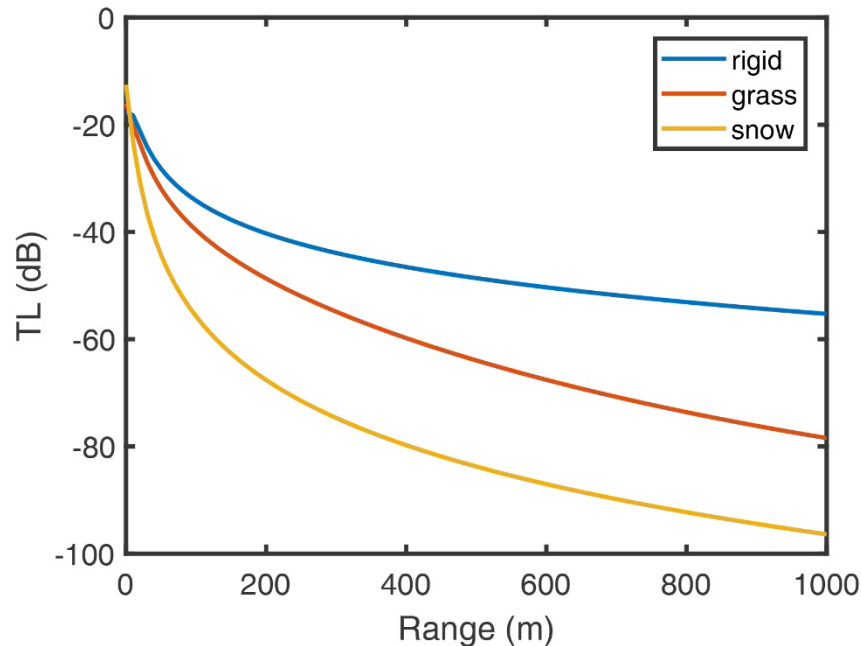
The following code listing brings together all of the previous steps to perform TL calculations for a rigid surface, grass-covered ground, and snow and then plots the results:

```
import mil.army.usace.knee.grids.*;
import mil.army.usace.knee.acoustic.*;
srcHgt = 5.0;
rcvHgt = 1.0;
range = [0.0:10.0:1000.0];
tgv = TransmitGridVert(srcHgt, rcvHgt, range);
modelTypes = ImpedancePlaneModelTypes.values;
params = ImpedancePlaneParamsHomo(modelTypes(14));
impModel = ImpedancePlaneModel(params);
freq = 100.0;
pmag2 = impModel.calcTransGridStruct(freq, tgv);
transLossGrass = 10*log10(pmag2);
params = ImpedancePlaneParamsHomo(modelTypes(2));
impModel = ImpedancePlaneModel(params);
transLossRigid = 10*log10(impModel.calcTransGridStruct(freq, tgv));
params = ImpedancePlaneParamsHomo(modelTypes(18));
impModel = ImpedancePlaneModel(params);
transLossSnow = 10*log10(impModel.calcTransGridStruct(freq, tgv));
h = plot(range, transLossRigid, range, transLossGrass, range,
         transLossSnow);
set(h, 'linewidth', 2)
xlabel('Range (m)', 'fontsize', 14)
ylabel('TL (dB)', 'fontsize', 14)
```

```
set(gca, 'fontsize', 14, 'linewidth', 2)
legend('rigid', 'grass', 'snow')
```

Figure 8 shows the resulting plot.

Figure 8. Transmission loss (TL) for propagation at a frequency of 100 Hz over several different ground surfaces.



6.4.2 Uneven ground

The impedance-plane model can also be run in conjunction with a wedge terrain diffraction model (Wilson and Yamamoto 2014). This model, in effect, replaces the terrain between the source and receiver by a wedge shape that approximates the effect of the intervening terrain. Internally, EASEE runs the impedance-plane and diffraction models separately and then adds the loss due to the impedance ground to the loss due to terrain diffraction. The terrain diffraction calculation is turned on using the `setTerrainEffect` method:

```
params.setTerrainEffect(true);
```

The terrain diffraction calculation requires a DEM. The DEM, as discussed earlier, is specified on a geographic grid. In this case, it is necessary to explicitly specify the geographic coordinates of the source and receiver. The `calcTransGridGeo` method is used for this purpose. This method has the following form:

```
GeoGridCart2DDouble calcTransGridGeo(double freq, GeoCoord srcCoord,
    double srcHgt, GeoGridCart2D rcvGrid, double rcvHgt)
```

The input arguments are the frequency, a geographic coordinate for the horizontal source position, the source height, a geographic grid for the horizontal receiver position, and the receiver height. The output is a 2-D geographic grid, with a `double[][]` data array, holding the squared pressure magnitude as calculated for each position in `rcvGrid`.

The following code listing illustrates how to perform a propagation calculation in complex terrain using the `calcTransGridGeo` method. In this example, the `peaks` function is used to create the terrain elevations, which is shown in Figure 9. The impedance-plane model, with terrain effects, is then called to calculate the TL, with the result shown in Figure 10.

```
import mil.army.usace.knee.geo.*;
import mil.army.usace.knee.grids.*;
import mil.army.usace.knee.acoustic.*;
SWcorner = GeoCoord(43.0, -72.0); % lat and lng of SW corner
NEcorner = GeoCoord(SWcorner, 10000.0, 10000.0);
    %NE corner 10 km N, 10 km E
nPts = 50; % number of points in each direction of grid
rcvHgt = 1.0;
rcvGrid = GeoGridCart2D(GeoGridTypes.UTM, DatumTypes.WGS84,
    SWcorner, NEcorner, nPts);
srcHgt = 5.0;
srcCoord = GeoCoord(SWcorner, 5000.0, 7000.0);
dem = GeoGridCart2DDouble(rcvGrid);
demData = 100.0*peaks(nPts);
dem.setDataGrid(demData);
params = ImpedancePlaneParamsHomo(ImpedancePlaneModelTypes.GRASS);
params.setTerrainEffect(true);
params.setTerrainElev(dem);
impModel = ImpedancePlaneModel(params);
freq = 100.0;
pmag2 = impModel.calcTransGridGeo(freq, srcCoord, srcHgt,
    rcvGrid, rcvHgt);
transLoss = 10*log10(pmag2.getDataGrid2D);
xaxis = linspace(0.0, 10.0, nPts);
figure(1)
```

```

imagesc(xaxis, xaxis, dem.getDataGrid2D);
hc = colorbar('vert');
hc.Label.String = 'Height (m)'; hc.Label.FontSize = 14;
set(gca, 'fontsize', 14)
axis('equal'); axis('xy'); caxis([-800 800])
xlabel('easting (km)', 'fontsize', 14)
ylabel('northing (km) ', 'fontsize', 14)
figure(2)
imagesc(xaxis, xaxis, transLoss);
xlabel('easting (km)', 'fontsize', 14)
ylabel('northing (km)', 'fontsize', 14)
hc = colorbar('vert');
hc.Label.String = 'TL (dB)'; hc.Label.FontSize = 14;
set(gca, 'fontsize', 14)
axis('equal'); axis('xy'); caxis([-120 -50])

```

Figure 9. Terrain elevations (digital elevation model, or DEM) used for the transmission loss calculation shown in Fig. 10. The coordinate axes are the easting and northing relative to the southwest corner of the domain.

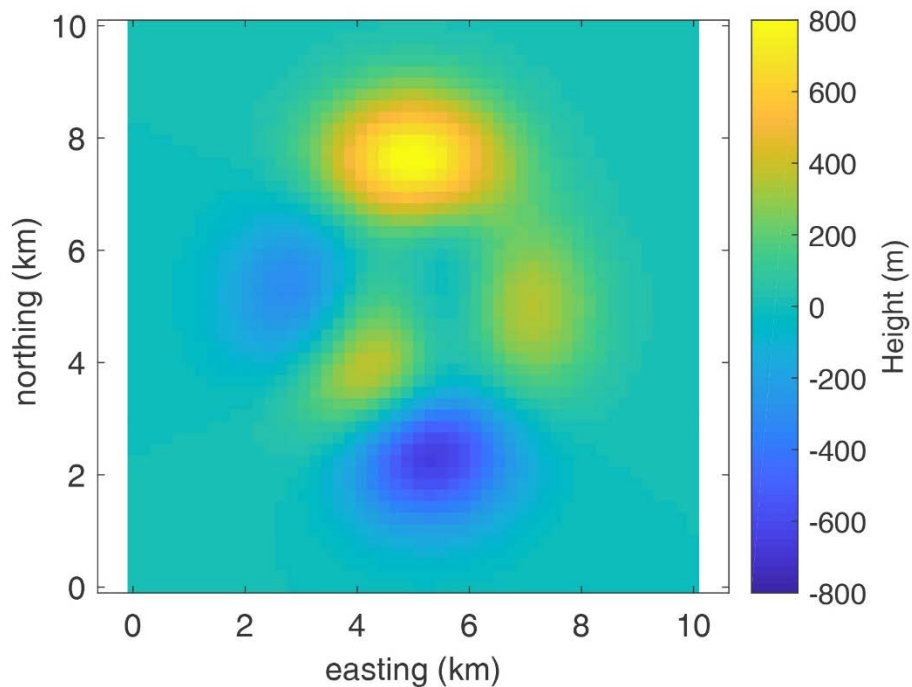
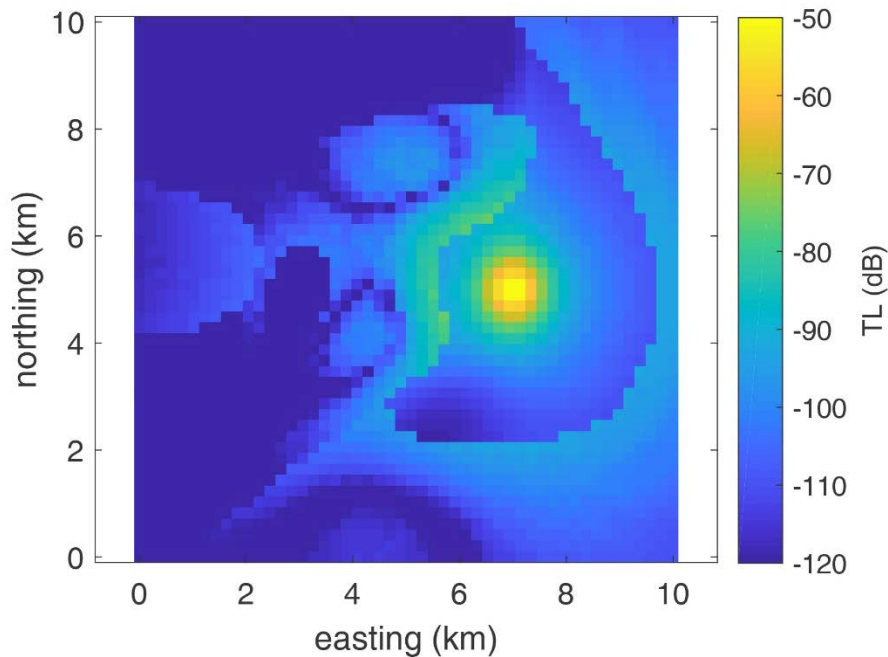


Figure 10. Transmission loss (TL) calculation for a source in hilly terrain. The frequency is 100 Hz. The source is positioned at an easting of 7 km and a northing of 5 km. The DEM for the calculation is shown in Fig. 9.



6.5 Parabolic equation methods

Parabolic equations (PE) efficiently calculate the impacts of weather and ground impedance on sound propagation in the atmosphere (Gilbert and Di 1993; Salomons 2001). Two PE codes are currently available in EASEE. These are Crank-Nicholson PE (CNPE), which is written in Java, and NASA's GFPE, which was originally written in FORTRAN and then compiled for Windows. Both are found in the EASEELib `acoustic` package.

The CNPE and GFPE share the same classes for defining parameters. Specifically, two classes are available, `ParabolicEqParamsHomo` and `ParabolicEqParamsVertProf`. As the names of these classes suggest, they are used for homogeneous and vertical profile environments, respectively. The basic constructors are

```
ParabolicEqParamsHomo(EnvironHomo scene)
```

and


```
ParabolicEqParamsVertProf(EnvironVertProf scene)
```

For example, to create the parameters for a vertical profile calculation, enter

```
scene = EnvironVertProf;  
params = ParabolicEqParamsVertProf(scene);
```

These classes also have methods to set the computational parameters. The `setComp` method controls the parabolic approximation, range step, and height step:

```
setComp(boolean wideAngle, double delr_norm, double delz_norm)
```

The first argument specifies whether a wide-angle calculation is to be used (if available); the second and third arguments are the range and height steps, as normalized by the wavelength. By default, a narrow-angle calculation is performed with the step values both set to 1/10 of a wavelength. For example, to set up a narrow-angle calculation with normalized range and height steps of 1/20 of a wavelength, enter

```
params.setComp(false, 0.05, 0.05);
```

The following method controls the characteristics of the absorbing (sponge) layer at the top of the calculation domain:

```
setAbsLayer(double layerHgt, double layerThick)
```

The first of these arguments is the height at which the absorbing layer begins as a fraction of the maximum horizontal range in the calculation. For example, if the maximum horizontal range is 10 km and `layerHgt` is 0.1, the absorbing layer will begin at 1 km. The second argument is the thickness of the absorbing layer in wavelengths.

The CNPE and the GFPE currently handle only flat ground with constant ground properties. Both perform calculations using the polar transmission grid (`TransGridPolar` class) as described in section 6.3.

Once the parameters and calculation grid have been constructed, it is straightforward to call either of the PE calculations. The name of the calculation class for the CNPE is `CNParabolicEqn`, whereas the GFPE is `NASA3DGfpe`. For example, to instantiate a CNPE calculation, enter

```
cnpe = CNParabolicEqn(params);
```

The following MATLAB code (which is a listing of a complete script) illustrates how to construct an environment (in this case, based on profiles constructed with MOST), run the CNPE and GFPE calculations, and then plot the results. For comparison purposes, we also run the impedance-plane model (section 6.4.1) and an alternative version of the CNPE available through the MATLAB interface (to be discussed in section 6.8).

```
SetPaths
%% Set the class paths and import the needed packages.
import mil.army.usace.knee.envIRON.*;
import mil.army.usace.knee.acoustic.*;
import mil.army.usace.easee.acoustic.*;
import mil.army.usace.knee.grids.*;

%% Set up the vertical profile environment.
surfTemp = 20.0;
surfRelHum = 40.0;
surfSpecHum = HumidAir.convHum(1, 3, surfRelHum, surfTemp);
windDir = 0.0;
windTypes = javaMethod('values',
    'mil.army.usace.knee.envIRON.AtmosSurfLayer$WindTypes');
windEnum = windTypes(4); % moderate wind
stabTypes = javaMethod('values',
    'mil.army.usace.knee.envIRON.AtmosSurfLayer$StabilityTypes');
stabEnum = stabTypes(4); % unstable
landCov = Landcover(NationalLandcoverDataset2001Types.GRASSLAND);
asl = AtmosSurfLayer(windEnum, stabEnum, landCov, windDir,
    surfTemp, surfSpecHum);
groundHgt = 0.0;
profHgt = logspace(-1, 2);
clouds = Clouds;
atmos = AtmosOneDim(profHgt, asl, landCov.getRoughHgt,
    landCov.getDispHgt, clouds);
```

```

soil = Soil(FASSTSoilTypes.SAND_POOR_GRADED);
seis = SeismicVertProf(SolidIsoLinear
    (SolidIsoLinearTypes.SAND_UNSAT));
env = EnvironVertProf(groundHgt, atmos, landCov, soil, seis);

%% Parameters and grid for the calculation.
freq = 250;
srcHgt = 5.0;
maxRange = 1000.0;
maxHgt = 199;
azi = 0.0;
rcvHgt = 2.0;
range = 0.0:10.0:maxRange;
tgV = TransmitGridVert(srcHgt, rcvHgt, range);
tgP = TransmitGridPolar(srcHgt, rcvHgt, range, azi);

%% Perform the calculations.
params = ParabolicEqParamsVertProf(env);
cnpeModel = CNParabolicEqn(params);
tlCnpe = 10*log10(cnpeModel.calcTransGridStruct(freq, tgP));
gfpeModel = NASA3DGfpe(params);
tlGfpe = 10*log10(gfpeModel.calcTransGridStruct(freq, tgV));
params = ImpedancePlaneParamsVertProf(env);
impModel = ImpedancePlaneModel(params);
tlImp = 10*log10(impModel.calcTransGridStruct(freq, tgV));
[rmesh, zmesh, pTL] = RunCalc(4, env, freq, srcHgt, maxRange,
    maxHgt, azi, false, true, false, 1, false);

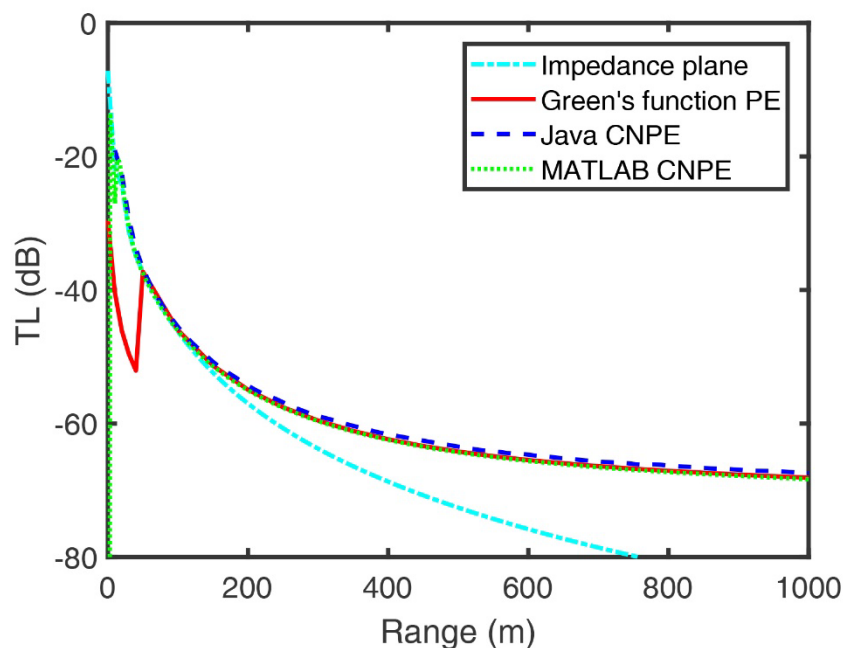
%% Plot the results.
fontsize = 14;
linewidth = 2.0;
h = plot(range, tlImp, 'c-.', range, tlGfpe, 'r-', range,
    tlCnpe, 'b--', rmesh, -pTL(3,:), 'g:');
set(h, 'linewidth', linewidth)
xlabel('Range (m)', 'fontsize', fontsize)
ylabel('TL (dB)', 'fontsize', fontsize)
ylim([-80 0])
set(gca, 'fontsize', fontsize, 'linewidth', linewidth)
legend('Impedance plane', 'Green''s function PE', 'Java CNPE',
    'MATLAB CNPE')

```

Figure 11 shows the resulting plot. At distances less than about 200 m, the CNPEs and impedance-plane model are in close agreement. This is to be expected since the meteorological effects are not very large near to the source; hence, the impedance-plane model, which does not include the meteorology, gives a good result in this range. However, the GFPE differs substantially from the other calculations at these distances, even though previous works (e.g., Gilbert et al. 1990; Salomons 1998; Lihoreau et al. 2006) illustrate the accuracy of both the GFPE and other PE methods. This anomaly may occur because solutions of the GFPE may only come into agreement at the end of the first range step that separates applications of the Fourier transform. Thus, at these short distances (<200 m), the CNPE seems to be preferable to the GFPE.

For longer distances, greater than about 200 m, the PE calculations all give nearly the same result. The predictions from the PEs differ substantially from the impedance-plane model due to the importance of the meteorological effects, in this case downward refraction in the downwind direction. The Java CNPE has a slightly smaller TL than the other two PEs (about 1 dB less). The reason for this discrepancy is unclear, but it is not very significant in a numerical sense.

Figure 11. Comparison of transmission loss (TL) calculations by several different codes.



6.6 BNOISE

The ERDC Construction Engineering Research Laboratory's BNOISE (Blast Noise) model is based on look-up tables generated by a fast-field program (FFP) (Salomons 2001). The table look-up is relatively fast although the accuracy of the calculation depends on the availability of a table generated with profiles similar to those in EASEE's environmental object.

Use of BNOISE is very similar to the PEs. Classes for setting up the model parameters are available for the homogeneous and vertical profile environments, which are named `BNOISEParamsHomo` and `BNOISEParamsVertProf`, respectively. The name of the actual calculation class is `BNOISE`. For example, once a vertical profile environment object and calculation grid have been constructed (by an approach such as that illustrated by the code listing in section 6.5), a BNOISE calculation can be run using

```
bnoiseModel = BNOISE(BNOISEParamsVertProf(env));
tlBnoise = 10*log10(bnoiseModel.calcTransGridStruct(freq, tgp));
```

6.7 Nord2000

The Nord2000 model (Plovsing 2006) was originally developed for noise-control applications. Unlike the impedance-plane model, the PEs, and BNOISE, it is a heuristic (approximate) model rather than a numerical method for solving the wave equation or a variant thereof. However, Nord2000 does have the advantage of being able to handle terrain diffraction and spatially varying ground-impedance.

Nord2000 is run by the same general approach described for the other models. The classes for setting up the model parameters are `Nord2000ParamsHomo` and `Nord2000ParamsVertProf`, and the propagation model class is `Nord2000Model`. The Nord2000 model is imported using

```
import com.brrc.easee.acoustic.*;
```

6.8 MATLAB acoustic propagation interface

In addition to the propagation models available in EASEE, a number of models have been coded directly in MATLAB. An interface has also been developed to run these MATLAB models using the environmental representations in EASEE. This can be regarded as a hybrid approach to using

EASEE in MATLAB; we use EASEE to define the inputs to the calculation but then run the actual calculation in MATLAB (rather than in Java).

The MATLAB models currently support only flat ground and a single type of land cover and soil. But several useful capabilities are provided that are unavailable in the Java code. For one, random turbulent fields, including temperature and wind velocity fluctuations, can be synthesized and incorporated into propagation calculations by using the parabolic equation (PE) and ray-tracing methods. This can be useful in cases of upward refraction, in which scattering by turbulence is particularly important (Wilson et al. 2015). Second, a capability exists for smoothly transitioning from the PE at narrow angles (within about 15° – 20° of horizontal, where the PE is accurate) to an impedance-plane model at high angles (where the PE is inaccurate). The MATLAB interface also includes a ray-tracing calculation, which can be overlaid on the TL calculation if desired.

The MATLAB interface involves two primary functions (m-files):

`RunCalc.m` and `PlotCalc.m`. These functions run the propagation calculation and plot the output, respectively. The header for `RunCalc.m`, which shows the input and output arguments, is

```
function [rmesh, zmesh, pTL, r_ray, z_ray, t_ray] =
    RunCalc(CalcType, env, freq, zsrc, maxRange, maxHgt,
    thmesh, biDirCalc, useEff, incTurb, numReal,
    findRayPaths, useImpAtHighAngle, limitAngle, useMOST)
```

The first input argument, `CalcType`, specifies the method for calculating the TL. The allowed values are 1 = impedance plane model, 2 = ray tracing, 3 = fast-field program, 4 = narrow-angle PE, and 5 = wide-angle PE. The second argument is an `EnvironVertProf` object constructed using EASEE, which specifies the atmospheric vertical profiles and ground properties.

The remaining input arguments are as follows:

- `freq`: acoustic frequency (Hz)
- `zsrc`: source height (m)
- `maxRange`: maximum horizontal distance (m)
- `maxHgt`: maximum height (m)
- `thmesh`: azimuthal angles at which the calculation will be performed (deg)

- `biDirCalc: true` to perform a calculation at `thmesh` and at `thmesh + pi`, where the calculations are joined together with the source at zero range
- `useEff: true` to use the effective sound-speed approximation
- `incTurb: true` to include random atmospheric turbulence
- `numReal: number of turbulence realizations to include prior to averaging`
- `findRayPaths: true` if ray paths are also to be calculated and overlaid on the TL display
- `useImpAtHighAngle: true` if the impedance plane model is to be substituted for high elevation angles when a PE calculation is used
- `limitAngle: elevation angle (deg) above which the impedance model is to be used (when useImpAtHighAngle=true)`
- `useMOST: use the Monin-Obukhov Similarity Theory for the profiles (regardless of whether MOST was used to construct the profiles in the EnvironVertProf object)`

The input arguments starting with `biDirCalc` are all optional, in which case they will be set to defaults (`biDirCalc = false, useEff = false, incTurb = false, numReal = 1, findRayPaths = false, useImpAtHighAngle = false, limitAngle = 30.0, useMOST = false`).

The output arguments of `RunCalc` are as follows:

- `rmesh: horizontal coordinate mesh for TL`
- `zmesh: vertical coordinate mesh for TL`
- `pTL: TL (in decibels relative to a distance of 1 m from the source)`
- `r_ray: horizontal coordinates of rays`
- `z_ray: vertical coordinates of rays`
- `t_ray: time coordinates of rays`

The final three arguments are empty if no ray-tracing calculation was requested (i.e., `CalcType` was set to 2 and `findRayPaths` was set to `false`).

The header for `PlotCalc.m` is

```
function h = PlotCalc(fig, rmesh, zmesh, pTL, r_ray, z_ray,
    t_ray, fontsize, clim, cbarlabel)
```

Here, `fig` is the number of the figure window for the plot. If an empty array is passed, a new figure window will be opened. The next six inputs

match the output of the `PlotCalc` function. The argument `fontsize` specifies the font size to be used for the axis labels; `clims` specifies the color limits; and `cbarlabel` specifies the label for the color bar. These last three arguments can be omitted, in which case they will be set to defaults (12, [30 90], and 'TL (dB)', respectively). The single output argument `h` is a handle to the axes of the plot, which can be used to tailor other graphics properties as desired.

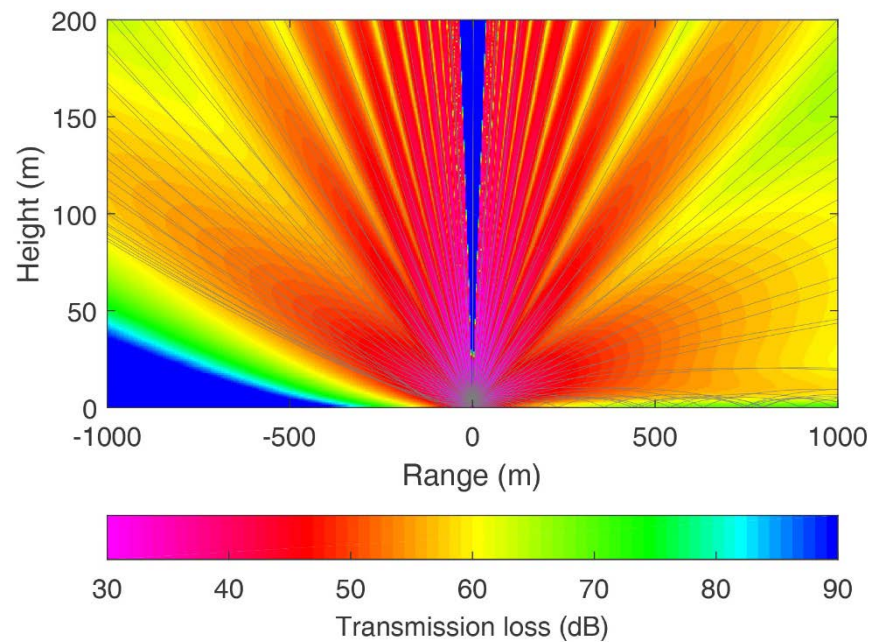
The following code, which is a complete MATLAB script, illustrates how to construct an environment, run a wide-angle parabolic equation calculation through the MATLAB interface, and then plot the results as a 2-D image (vertical plane). Figure 12 shows the plot that results from running this code:

```
SetPaths
import mil.army.usace.knee.envIRON.*;
surfTemp = 20.0;
surfRelHum = 40.0;
surfSpecHum = HumidAir.convHum('relative', 'specific',
    surfRelHum, surfTemp);
windDir = 0.0;
windTypes = javaMethod('values',
    'mil.army.usace.knee.envIRON.AtmosSurfLayer$WindTypes');
windEnum = windTypes(4); % moderate wind
stabTypes = javaMethod('values',
    'mil.army.usace.knee.envIRON.AtmosSurfLayer$StabilityTypes');
stabEnum = stabTypes(4); % unstable
landCov = Landcover(NationalLandcoverDataset2001Types.GRASSLAND);
asl = AtmosSurfLayer(windEnum, stabEnum, landCov, windDir,
    surfTemp, surfSpecHum);
groundHgt = 0.0;
profHgt = logspace(-1, 2);
clouds = Clouds;
atmos = AtmosOneDim(profHgt, asl, landCov.getRoughHgt,
    landCov.getDispHgt, clouds);
soil = Soil(FASSTSoilTypes.SAND_POOR_GRADED);
seis = SeismicVertProf(SolidIsoLinear(SolidIsoLinearTypes.
    SAND_UNSAT));
env = EnvironVertProf(groundHgt, atmos, landCov, soil, seis);
CalcType = 5; % wide-angle parabolic equation
```



```
freq = 250;  
zsrc = 5.0;  
maxRange = 1000.0;  
maxHgt = 0.2*maxRange;  
thmesh = 0.0;  
[rmesh, zmesh, pTL, r_ray, z_ray, t_ray] = RunCalc(CalcType, env,  
    freq, zsrc, maxRange, maxHgt, thmesh, true, false, false,  
    1, true);  
PlotCalc([], rmesh, zmesh, pTL, r_ray, z_ray, t_ray);
```

Figure 12. Transmission loss resulting from a wide-angle parabolic equation calculation using the MATLAB interface.



7 Full Example Script for Testing EASEE in MATLAB

The following example script—entitled `CalcExampWithWRF.m`—can be copied and pasted into MATLAB to test EASEE and its functionality with data from the WRF weather model. It is also located in the folder where `StartEASEE.m` resides. It produces many of the plots shown previously and two additional plots (Figures 13 and 14) that illustrate the difference in TL between simulations that include and exclude WRF weather model data.

```
% This script allows one to fully test EASEE for use in MATLAB
% with data
% from the WRF weather model.
% The script initializes EASEE, tests its functionality with WRF
% weather
% model output, runs calculations with the impedance plane model,
% and
% calculates 2D acoustic propagation with the wide-angle
% parabolic equation
% with and without WRF weather data. Figures 1-4 are found in
% the
% technical report entitled "Using EASEE's Acoustical
% Calculations in
% MATLAB," by Keith Wilson et al. Figures 5-6 are additional
% figures that
% represent, respectively, transmission loss from a wide-angle
% parabolic
% equation simulation that includes WRF model output, and the
% difference
% between this simulation and one that does not include WRF model
% output.

clear
close all

% Initializing EASEE for MATLAB
StartEASEE

%~~~~~
% USER INPUT
% Please change the example path below to your local parent
% directory for KNEE
KNEEdir = 'C:\Users\*username*\Documents\EASEE_Git';
lat = 40; % latitude
lon = -75; % longitude
timeInd = 1; % time index (0 = first time step)
%~~~~~

% Testing the installation
import mil.army.usace.knee.acoustic.*;
test1=ImpedancePlaneParamsHomo;
```

```

disp(test1)
import mil.army.usace.easee.acoustic.*;
test2=ParabolicEqParamsHomo;
disp(test2)

%% Testing functionality of EASEE with WRF weather data

% One WRF file
import mil.army.usace.knee.envIRON.*;
import mil.army.usace.knee.geo.*;
ncfile = WrfLoader.openNetcdf(['KNEEdir '\KNEE\src\test\
resources\TestWeatherFiles\WRF\wrfout_d01_2017-01-
01_00_00_00_file1']);
WrfLoader.getWrfLatLonIndices(ncfile, lat, lon)
wrfFile = WrfLoader(ncfile, 1, lat, lon, timeInd, 0.1, 0.2);
atm = wrfFile.getAtmosProf; % Atmospheric vertical profile
object
profhgts = atm.getProfHgt; % Vertical profile heights
proftemps = atm.getTempProf; % Air temperature
profwinde = atm.getWindProfE; % Eastward component of wind
profwindn = atm.getWindProfN; % Northward component of wind
geocoord = WrfLoader.getWrfGeoCoord(ncfile, lat, lon);

figure(1)
plot(proftemps, profhgts, profwinde, profhgts, profwindn,
profhgts)
line([0 0], ylim, 'LineStyle','--'); % Vertical reference line
at temp/wind = 0
legend('temp','wind\_E','wind\_N','Location','southeast')
xlabel(['Temperature (' 176 'C), Wind (m s^{-1})']) % 176 is the
character code for the "degrees" symbol
ylabel('Altitude (m)')
title('Vertical Profiles of Temperature and Wind')

% Multiple WRF files
import mil.army.usace.knee.envIRON.*;
import mil.army.usace.knee.geo.*;
import java.util.ArrayList;
ncfile1 = WrfLoader.openNetcdf(['KNEEdir '\KNEE\src\test\resources\
TestWeatherFiles\WRF\wrfout_d01_2017-01-
01_00_00_00_file1']);
ncfile2 = WrfLoader.openNetcdf(['KNEEdir '\KNEE\src\test\resources\
TestWeatherFiles\WRF\wrfout_d01_2017-01-
01_00_00_00_file2']);
ncfile3 = WrfLoader.openNetcdf(['KNEEdir '\KNEE\src\test\resources\
TestWeatherFiles\WRF\wrfout_d01_2017-01-
01_00_00_00_file3']);
fileList = ArrayList;
fileList.add(ncfile1);
fileList.add(ncfile2);
fileList.add(ncfile3);
env = EnvironVertProf();
latlon = WrfLoader.getWrfLatLonIndices(ncfile1, lat, lon); %
latlon contains two elements: a latitude index and a
longitude index
timesize = WrfLoader.getTimeSize(ncfile1);

```

```

wrfFiles = WrfEnsemble.makeEnvironEnsemble(fileList, 1, latlon(1),
    latlon(2), 0:2, env);
wrfEnsMem1 = wrfFiles.getEnsembleMember(0);
wrfEnsMem2 = wrfFiles.getEnsembleMember(1);
wrfEnsMem3 = wrfFiles.getEnsembleMember(2);
wrfEnsMem1_t2 = wrfEnsMem1.get(1); % Extracts WRF data from the
    first ensemble member ("Mem1"), second time step
wrfEnsMem2_t2 = wrfEnsMem2.get(1);
wrfEnsMem3_t2 = wrfEnsMem3.get(1);
wrfEnsAtm1 = wrfEnsMem1_t2.getAtmosProf; % Atmospheric vertical
    profile object
wrfEnsAtm2 = wrfEnsMem2_t2.getAtmosProf;
wrfEnsAtm3 = wrfEnsMem3_t2.getAtmosProf;
profhgts = wrfEnsAtm1.getProfHgt; % Same profile heights can be
    used for all ensemble members
proftemps1 = wrfEnsAtm1.getTempProf; % Air temperature
proftemps2 = wrfEnsAtm2.getTempProf;
proftemps3 = wrfEnsAtm3.getTempProf;

figure(2)
plot(proftemps1, profhgts, proftemps2, profhgts, proftemps3,
    profhgts)
line([0 0], ylim, 'LineStyle','--'); % Vertical reference line
    at temp = 0
legend('temp1','temp2','temp3','Location','southwest')
xlabel(['Temperature (' 176 'C)']) % 176 is the character code
    for the "degrees" symbol
ylabel('Altitude (m)')
title('Vertical Profiles of Temperature for Three "Ensemble"
    Members')

%% Impedance plane model
import mil.army.usace.knee.grids.*;
import mil.army.usace.knee.acoustic.*;
srcHgt = 5.0;
rcvHgt = 1.0;
range = [0.0:10.0:1000.0];
tgv = TransmitGridVert(srcHgt, rcvHgt, range);
modelTypes = ImpedancePlaneModelTypes.values;
params = ImpedancePlaneParamsHomo(modelTypes(14));
impModel = ImpedancePlaneModel(params);
freq = 100.0;
pmag2 = impModel.calcTransGridStruct(freq, tgv);
transLossGrass = 10*log10(pmag2);
params = ImpedancePlaneParamsHomo(modelTypes(2));
impModel = ImpedancePlaneModel(params);
transLossRigid = 10*log10(impModel.calcTransGridStruct(freq, tgv));
params = ImpedancePlaneParamsHomo(modelTypes(18));
impModel = ImpedancePlaneModel(params);
transLossSnow = 10*log10(impModel.calcTransGridStruct(freq, tgv));

figure(3)
h = plot(range, transLossRigid, range, transLossGrass, range,
    transLossSnow);
xlabel('Range (m)');
ylabel('TL (dB)');
legend('rigid', 'grass', 'snow')

```

```

title('Transmission loss over different ground surfaces')

%% Wide-angle parabolic equation calculations (2D plots)
% Without weather data from the WRF model
SetPaths
import mil.army.usace.knee.enviro.*;
surfTemp = 20.0;
surfRelHum = 40.0;
surfSpecHum = convhum('relative', 'specific', surfRelHum, surfTemp);
windDir = 0.0;
windTypes = javaMethod('values',
    'mil.army.usace.knee.enviro.AtmosSurfLayer$WindTypes');
windEnum = windTypes(4); % moderate wind
stabTypes = javaMethod('values',
    'mil.army.usace.knee.enviro.AtmosSurfLayer$StabilityTypes');
stabEnum = stabTypes(4); % unstable
landCov = Landcover(NationalLandcoverDataset2001Types.GRASSLAND);
asl = AtmosSurfLayer(windEnum, stabEnum, landCov, windDir,
    surfTemp, surfSpecHum);
groundHgt = 0.0;
profHgt = logspace(-1, 2);
clouds = Clouds;
atmos = AtmosOneDim(profHgt, asl, landCov.getRoughHgt,
    landCov.getDispHgt, clouds);
soil = Soil(FASSTSoilTypes.SAND_POOR_GRADED);
seis = SeismicVertProf(SolidIsoLinear(SolidIsoLinearTypes.
    SAND_UNSAT));
env = EnvironVertProf(groundHgt, atmos, landCov, soil, seis);
CalcType = 5; % wide-angle parabolic equation
freq = 250;
zsrc = 5.0;
maxRange = 1000.0;
maxHgt = 0.2*maxRange;
thmesh = 0.0;
[rmesh, zmesh, pTL1, r_ray, z_ray, t_ray] = RunCalc(CalcType, env,
    freq, zsrc, maxRange, maxHgt, thmesh, true, false, false, 1,
    true);
PlotCalc([], rmesh, zmesh, pTL1, r_ray, z_ray, t_ray);
title('Without WRF')

% With weather data from the WRF model
SetPaths
import mil.army.usace.knee.enviro.*;
env = EnvironVertProf(groundHgt, wrfFile, landCov, soil, seis);
% Using wrfFile instead of atmos
[rmesh, zmesh, pTL2, r_ray, z_ray, t_ray] = RunCalc(CalcType, env,
    freq, zsrc, maxRange, maxHgt, thmesh, true, false, false, 1,
    true);
PlotCalc([], rmesh, zmesh, pTL2, r_ray, z_ray, t_ray);
title('With WRF')

% Plotting the difference between simulations that include and
% do not include weather data from the WRF model
pTLdiff = pTL2 - pTL1;
PlotCalc([], rmesh, zmesh, pTLdiff, [], z_ray, t_ray, [],
    [-20 20], 'TL Difference (dB)');
title('With WRF - Without WRF')

```

Figure 13. Same as Fig. 12 but using atmospheric output from the WRF weather model.

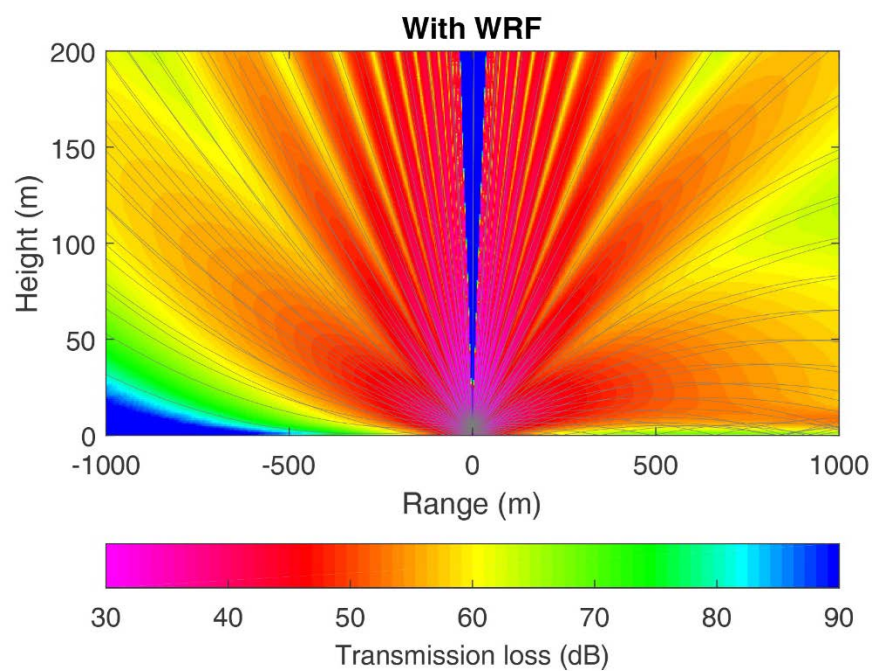
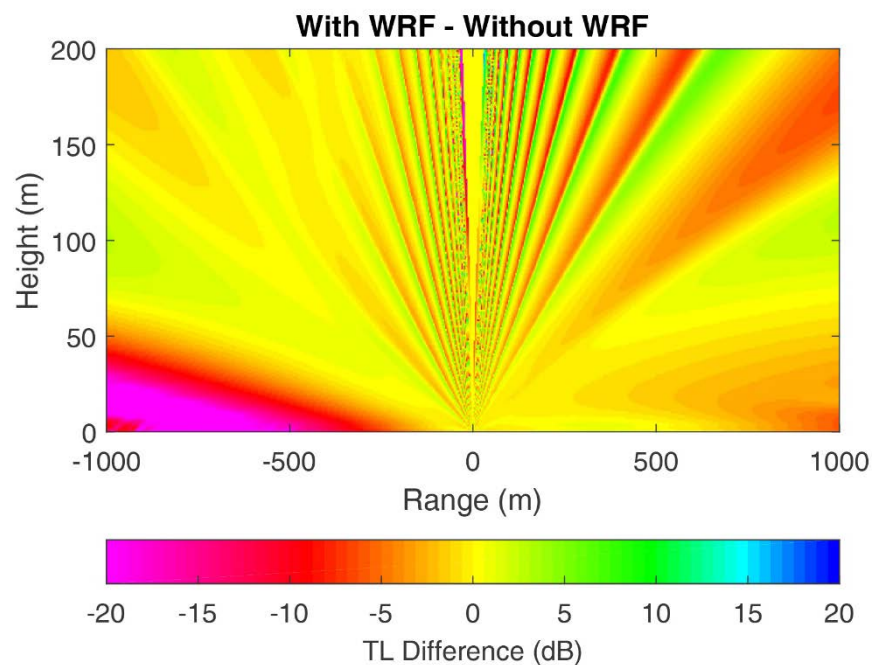


Figure 14. Difference in transmission loss (TL) between Fig. 13 and Fig. 12.



8 Conclusion

This report has shown how EASEE can provide a powerful MATLAB toolbox for creating representations of the atmosphere and terrain and for calculating sound propagation based on these representations. This toolbox is potentially useful for many projects involving acoustics and noise control.

We described two general approaches, both of which involved using EASEE to represent the propagation environment (atmosphere and terrain): In one approach, we used EASEE to perform the sound propagation calculations, while in the other, we used MATLAB. Both approaches are potentially useful, depending on whether the user wishes to code their propagation algorithms in Java or MATLAB.

We envision follow-ons to this effort in two main directions. First, the atmospheric representations are already being extended to 3-D, and it would be desirable to have a library of sound propagation models that can perform calculations with these 3-D data. Second, toolboxes for other signal modalities, such as radio frequency (RF), visible, and infrared (IR) propagation could be created.

Overall, this MATLAB interface can be helpful for those who would like to access the already numerous signal modeling capabilities of EASEE, and its utility will continue to improve as more features are added to EASEE in future versions.

References

- Attenborough, K., S. I. Hayek, and J. M. Lawther. 1980. Propagation of Sound over a Porous Half-Space. *Journal of the Acoustical Society of America* 68 (5): 1493–1501. <https://doi.org/10.1121/1.385074>.
- Attenborough, K., S. Taherzadeh, H. E. Bass, X. Di, R. Raspet, G. R. Becker, A. Güdesen, A. Chrestman, G. A. Daigle, A. L'Espérance, Y. Gabillet, K. E. Gilbert, Y. L. Li, M. J. White, P. Naz, J. M. Noble, and H. A. J. M. van Hoof. 1995. Benchmark Cases for Outdoor Sound Propagation Models. *Journal of the Acoustical Society of America* 97 (1): 173–191. <https://doi.org/10.1121/1.412302>.
- Di, X., and K. E. Gilbert. 1993. An Exact Laplace Transform Formulation for a Point Source above a Ground Surface. *Journal of the Acoustical Society of America* 93 (2): 714–720. <https://doi.org/10.1121/1.405435>.
- Eckel, B. 2006. *Thinking in Java*. 4th ed. Upper Saddle River, NJ: Prentice Hall.
- Frankenstein, S., and G. G. Koenig. 2004. *Fast All-Season Soil Strength (FASST)*. ERDC/CRREL SR-04-1. Hanover, NH: U.S. Army Engineer Research and Development Center.
- Gilbert, K. E., R. Raspet, and X. Di. 1990. Calculation of Turbulence Effects in an Upward-Refracting Atmosphere. *Journal of the Acoustical Society of America* 87 (6): 2428–2437. <https://doi.org/10.1121/1.399088>.
- Gilbert, K. E., and X. Di. 1993. A Fast Green's Function Method for One-Way Sound Propagation in the Atmosphere. *Journal of the Acoustical Society of America* 94:2343–2352. <https://doi.org/10.1121/1.407454>.
- Lihoreau, B., B. Gauvreau, M. Bérengier, P. Blanc-Benon, and I. Calmet. 2006. Outdoor Sound Propagation Modeling in Realistic Environments: Application of Coupled Parabolic and Atmospheric Models. *Journal of the Acoustical Society of America* 120 (1): 110–119. <https://doi.org/10.1121/1.2204455>.
- Lowe, D., and B. Burd. 2007. *Java All-in-One Desk Reference for Dummies*. 2nd ed. Indianapolis, IN: Wiley Publishing, Inc.
- MathWorks. 2018a. How do I change the Java Virtual Machine (JVM) that MATLAB is using on Windows? MATLAB Answers. Last modified 9 May 2018. <https://www.mathworks.com/matlabcentral/answers/130359-how-do-i-change-the-java-virtual-machine-jvm-that-matlab-is-using-on-windows>.
- MathWorks. 2018b. How do I change the Java Virtual Machine (JVM) that MATLAB is using on macOS? MATLAB Answers. Last modified 13 June 2018. <https://www.mathworks.com/matlabcentral/answers/103056-how-do-i-change-the-java-virtual-machine-jvm-that-matlab-is-using-on-macos>.
- NCAR (National Center for Atmospheric Research). 2017. The Weather Research and Forecasting Model. <https://www.mmm.ucar.edu/weather-research-and-forecasting-model> (accessed 2 June 2017).

- NCAR (National Center for Atmospheric Research). 2018. WRF Model Users' Page. <http://www2.mmm.ucar.edu/wrf/users/> (accessed 26 December 2018).
- Oracle. n.d. Java SE Downloads. <http://www.oracle.com/technetwork/java/javase/downloads>.
- Perry, S. J. 2010. Introduction to Java Programming, Part 1: Java Language Basics, Object-Oriented Programming on the Java Platform. IBM Developer. <https://www.ibm.com/developerworks/java/tutorials/j-introtojava1/index.html>.
- Plovsing, B. 2006. *Nord2000. Validation of the Propagation Model*. Delta Acoustics AV 1117/06. Hørsholm, Denmark: Danish Electronics, Light & Acoustics.
- Rouse, M. 2014. Data Abstraction Layer. WhatIs.com. Newton, MA: TechTarget. <https://whatis.techtarget.com/definition/database-abstraction-layer>.
- Salomons, E. M. 1998. Improved Green's Function Parabolic Equation Method for Atmospheric Sound Propagation. *Journal of the Acoustical Society of America* 104 (1): 100–111. <https://doi.org/10.1121/1.423260>.
- Salomons, E. M. 2001. *Computational Atmospheric Acoustics*. Dordrecht, the Netherlands: Kluwer Academic.
- Sierra, K., and B. Bates. 2005. *Head First Java*. 2nd ed. Sebastopol, CA: O'Reilly Media, Inc.
- Skamarock, W. C., J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, M. G. Duda, X.-Y. Huang, W. Wang, and J. G. Powers. 2008. *A Description of the Advanced Research WRF Version 3*. NCAR/TN-475+STR. Boulder, CO: National Center for Atmospheric Research.
- Stull, R. B. 1988. *An Introduction to Boundary Layer Meteorology*. Boston: Kluwer Academic Publishers.
- Wilson, D. K., J. T. Kalb, N. Srour, T. Pham, and B. M. Sadler. 2002. *Sensor Algorithm Interface and Performance Simulations in an Acoustical Battlefield Decision Aid*. ARL-TR-2860. Adelphi, MD: U.S. Army Research Laboratory.
- Wilson, D. K., R. Bates, and K. K. Yamamoto. 2009. *Object-Oriented Software Model for Battlefield Signal Transmission and Sensing*. ERDC/CRREL TR-09-17. Hanover, NH: U.S. Army Engineer Research and Development Center.
- Wilson, D. K., and K. K. Yamamoto. 2014. *Environmental Awareness for Sensor and Emitter Employment (EASEE): Software Design Version 2*. ERDC/CRREL TR-14-27. Hanover, NH: U.S. Army Engineer Research and Development Center.
- Wilson, D. K., C. L. Pettit, and V. E. Ostashev. 2015. Sound Propagation in the Atmospheric Boundary Layer. *Acoustics Today* 11 (2): 44–53.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE (DD-MM-YYYY) January 2019			2. REPORT TYPE Technical Report/Final		3. DATES COVERED (From - To)	
4. TITLE AND SUBTITLE Using EASEE's Acoustical Calculations in MATLAB					5a. CONTRACT NUMBER	
					5b. GRANT NUMBER	
					5c. PROGRAM ELEMENT NUMBER 62784/T40/46	
6. AUTHOR(S) D. Keith Wilson, Ross E. Alter, Katrina M. Burch, and Michelle E. Swearingen					5d. PROJECT NUMBER	
					5e. TASK NUMBER	
					5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Engineer Research and Development Center (ERDC) Cold Regions Research and Engineering Laboratory (CRREL) 72 Lyme Road Hanover, NH 03755-1290					8. PERFORMING ORGANIZATION REPORT NUMBER ERDC TR-19-1	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Headquarters, U.S. Army Corps of Engineers Washington, DC 20314-1000					10. SPONSOR/MONITOR'S ACRONYM(S)	
					11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.						
13. SUPPLEMENTARY NOTES ERDC 6.2/6.3 Military Engineering (ME) RAFTER						
14. ABSTRACT EASEE (Environmental Awareness for Sensor and Emitter Employment) is a Java-based software framework for modeling the impacts of the weather and terrain on signal propagation and sensor performance. EASEE includes extensive capabilities for representing the environment (atmosphere, land cover, terrain elevation, and soil properties), along with many different models related to acoustic, optical, radio frequency, and seismic signals. This report describes how to run EASEE from MATLAB, which is a popular software package for performing numerical calculations and displaying graphics. For this purpose, a simple installation configuration and MATLAB script were devised to set up and initialize EASEE. The focus of the report is on using EASEE for acoustic propagation calculations, which is its most mature signal modality. The report describes two general approaches to performing acoustical calculations in EASEE: one that involves using EASEE for its environmental representation only and then running the acoustic propagation calculation using a MATLAB toolbox and a second that uses EASEE for both its environmental layer and the acoustical calculation. Overall, this report shows that the MATLAB user interface provides convenient access to EASEE's powerful signal modeling capabilities.						
15. SUBJECT TERMS EASEE (Computer program), Computer software, Detectors--Environmental aspects, Detectors--Computer simulation, Remote sensing--Atmospheric effects, Digital communications						
16. SECURITY CLASSIFICATION OF:				17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	19b. TELEPHONE NUMBER (include area code)			