JHomogenizer User's Guide

Version 1.0

Joe Koebbe
Department of Mathematics and Statistics
Utah State University
Logan, Utah 84322-3900

November 5, 2003

Contents

Preface:

This document describes a piece of software written by the author with the intent of providing an environment for comparison of homogenization methods. The interface allows users to implement homogenization methods and compare the results to other homogenization methods implemented in the environment. The methods can be implemented in native languages via hooks created in the java code written to implement the graphical user interface. Users can also include homogenization methods through the develpment of Java AveragingObjects. These objects are implemented to conform with an abstract object definition. Instructions are given as to how one can add methods to the environment. Examples are given that should help users get their methods implemented.

In addition, to the raw results of the homogenization methods users can see how the solution of a typical elliptic problem is effected by the homogenization process from different methods of averaging. Users can look at the elliptic solution, gradient, and flow through the heterogeneous and homogenized media. This part of the interface allows the user to get a good look at how the homogenization effects the flow in the map.

Finally, one can use this as an analysis tool to see how subregions behave under homogenization methods. Users can pull out subregions in a map and homogenize the data in that region.

It is hoped that people developing homogenization methods will benefit from this effort and have a common tool to use.

Joe Koebbe

JHomogenizer User's Guide

1 Introduction

This User's Guide is intended to get you the reader through the process of starting up the JHomogenizer tool and working through the use of homogenization methods. The application allows the user to apply a number of different methods of homogenization. Methods from simple schemes like computing the arithmetic or harmonic average to complicated methods couched in rigorous mathematical homogenization theory [3], [1] can be applied to heterogeneous media. The methods can be applied and compared on a full region or subregions of the original domain.

Another feature allows for the application of different homogenization methods to different subregions. For example, there may be regions that are relatively homogeneous on which one would apply a simple method like arithmetic averaging. Other subregions may have fractures or other complicated structures. Users can block off these regions and apply "better" averaging methods to the more complicated heterogeneities. One can also use this feature to set up a region with the same heterogeneous values periodically repeated in multiple macroscopic cells. Then apply different methods to each of the repeated regions to compare the averaging methods.

The code is multithreaded and the object oriented averaging objects allow the user a lot of flexibility in obtaining homogenized values. When a region is subdivided into blocks, each block is assigned an averaging method. An averaging object is created for each block that includes the data from the block and all the necessary information for the averaging process chosen. Each of the averaging objects is "runnable". That is, after the object is instantiated the object will create its own thread and run independently of the other blocks being homogenized. Each of the objects is assumed to be independent allowing for the independent computation described.

This user guide also shows you how to include your own averaging methods into the interface. There are two basic methods for doing this. The first way to do this is to extend a Java *AverageObject*. This approach to implementing a homogenization method requires that you implement your averaging method in the Java programming language. If you follow the instructions in this User's Guide you will be able to build an averaging object that implements your favorite technique. The downside of doing this is that you will need to follow the restrictions imposed by the author on the development of the Java objects. Actually, the AverageObject defines a Java abstract object. Any averaging object included in the package is assumed to extend the abstract object. This requires that all of the methods set out in the AverageObject interface must be implemented in your java object. The contents of this object are presented in Appendix B. If you choose to do this you will find it straight forward to modify existing averaging objects (e.g., ArithmeticAverageObject.java) to develop your own averaging objects.

The other method of implementing your favorite homogenization method involves interfacing your native code written in languages like Fortran and C/C++ with the interface. Again there are a few steps that must be followed. The input and output data structure are fixed. So you may need to modify the input and output in your code to fall in line with the data file formats used in the JHomogenizer application. Examples of how to translate input and output formats for the interface are given in this User's Guide. The down side of using native code is that native code must be compiled for each new platform or operating system you use. That is, you will need to recompile the native code on each new platform that you export your methods to. The Java code approach described above is platform independent[1].

Details related to the development and implementation of averaging methods into the code are given later in this User's Guide. If you do add a method of averaging the author would appreciate your inclusion of the method in future releases of the JHomogenizer application. You can send the new additions to the author. Please include instructions on how the method was included and a little documentation of the method - maybe a publication or reference to a paper in the literature with the details of the implementation. All contributions to this package will be given appropriate credit. It would be nice to be able to include references to all research articles published by contributors related to the homogenization methods in the JHomogenizer package. So multiple references and background references will certainly be welcome.

In the following sections the JHomogenizer tool is described along with examples of how to build heterogeneous maps and how to compute average values for these maps. The examples given in this document walk you through the steps of using the code.

[1]almost - you must have an appropriate Java Runtime Environment (JRE)

2 Obtaining and Installing the Software

This software package has been developed to run on unix and windows platforms. The main development of the interface and all Java code has been done within a Solaris operating system environment. The Java code for the interface and a number of the basic averaging objects has been developed under Solaris. The Java development uses Java 1.2 and does not use the swing classes offered by Sun Microsystems Inc. or any other specialized features of individual implementations of the Jave programming language. The main idea has been to restrict the development to a standard[2] Java implementation. The Java code should be able to run under most basic runtime environments. The windows development/implementation is being done on a Windows XP machine and tested on older windows computing environments (e.g., Windows 95, 98, 2000). At this point everything seems to work. Over time the testing on older windows operating systems will be dropped from the development process.

The interface is a self-contained piece of code that runs within a Java Runtime Environment (JRE). However, some of the methods of homogenization included in the package are implemented in native code (e.g., Fortran 77, Fortran 90, or C). These codes will need to be compiled locally to create appropriate executables. The installation script will try to compile the codes of specific types; f77, f90, C, and C++ codes. In the end it may be up to you to figure out how to get the code compiled. While we are on the topic of JRE's, the software package includes a windows JRE that is downloadable from Sun Microsystems Inc. The JRE shipped with the software is the JRE that the code has been tested with. The simple C++ executable used to start up the software on a windows desktop looks for the JRE shipped with the package. So, you should use the JRE that comes with the package.

There are a couple of ways that you can get the code. The easiest way to get the code is to download a compressed file over the internet. The author's web site (given below) will have the appropriate link to the software. Look for the JHomogenizer name somewhere on the homepage. The other way to obtain the software is to make a request for a CD of the software. The CD will have versions that run on Unix, Windows, and Macintosh (if you want) machines. The files can be copied directly to the machine or you can run the application from the CD. It is not a good idea to run the software from the CD since it will load and run a lot more slowly from the CD. The best idea is to copy the software onto the hard drive on your computer. Some performance increase will occur if only the JRE is loaded to the harddrive as described below. However, for best performance the entire package should be stored on your hard drive.

[2]Whatever that means!

2.1 Downloading the Application

The code described in this User's Guide is available for download from the web site maintained by the author. The code can be downloaded in one of three compressed formats; compress'd, gzip'd, or zip'd. The *compress* and *uncompress* commands are unix file compression commands that some people are comfortable using. The gzip and gunzip commands are the usual GNU file compression programs that one sees on most modern unix and linux machines. The zip and unzip commands are more typical on windows machines, although versions of these compression programs exist for unix and linux environments.

The site is:

```
http://www.math.usu.edu/~koebbe/
```

You will need to follow the links to the appropriate page to download the code and the application. Note that the author's homepage changes from time to time. That makes it difficult to give an exact location for the link that will be good for all time. So, you will probably need to look around on the author's homepage. There will always be a link to the software somewhere on the author's main webpage.

Note that you will need to make sure that an appropriate Java Runtime Environment (JRE) is installed on your computer. If a JRE is not installed on your computer you can find one contained within the compressed file you download. These are not hard to find; just search the web by including Java Runtime Environment and your platform or operating system in the search string. Note that Apple Computer Inc. maintains the JRE for Macintosh computers. Go to apple.com to search for the JRE for Macintosh computers. However, you may need to modify the startup script or the command you use to start up the software.

2.2 Ordering a CD

The real advantage in ordering a CD is that Java Runtime Environments (JREs) that work on Unix, Windows, and Macintosh machines comes along on the CD. If you download the software over the internet you will need to find an appropriate JRE for your machine.

You can order a CD that contains the Unix/Windows/Machintosh code burned on it. The cost of this will depend on when you order it. As of right now the cost will not exceed $25. It will depend on the cost of things (e.g., time, effort, shipping, and materials) at the time ordered. The price of the CD is intended to cover the cost of producing and shipping the CD. You can order the CD from


```
Joe Koebbe
Department of Mathematics and Statistics
USU
Logan, UT 84322-3900
```


Note that you can order by sending the author email at koebbe@math.usu.edu. You may need to wait a bit for this since the CDs will be burned one at a time in the near future. There probably won't be enough demand to justify making a number of copies of the CD all at once.

2.3 Installing the Application

The software installation will vary from one platform to the next - but not too much. In addition, there will be other ways that the software can be installed. However, the installation instructions described here will give a basic idea of how to proceed and have been tried out on each of the platforms.

2.3.1 Solaris Installation

The software has been developed primarily on a Solaris machine, a Sun Ultra 10[3]. The installation can be done in a users home directory and the installation is straight forward. Basically, there is a jar file that contains all of the code needed to run the application. In addition, there are some native codes (e.g., implementations written in Fortran) that must reside where the interface can find the executables. Finally, there is a simple script that will start up the Java interface using the jar file.

There are a couple of different scenarios under which you might need to install the software. The first scenario is when you install the software from a CD obtained from the author. The second situation occurs if you download a compressed tar file from the author's web site.

Installation from CD To install the software from the CD perform the following steps:

1. Insert the CD into the drive and find your way into the drive. Go to the unix directory on the CD.

2. In the unix version of the software move to the "bin" directory.

3. Execute the installation script. The script will copy a bunch of files to your a folder[4] that is created in your home directory. In fact, the script will create a folder named "java" in your home directory and then a subdirectory named "JHomogenizer" in the "java" folder. Once this is done the script will copy the files to the newly created directory and install the software.

Once the script completes running the software should be ready to run. The script will attempt to compile the native C, Fortran 77, and Fortran 90 codes created by users. Finally, the script will compile the java code for the interface.

If you are not worried about the compilation step you can avoid the installation script. You just need to copy the JHomogenizer folder on the CD to a directory named "java" in your home directory.

Installation from Compressed File If you download the software from the web site it is a simple task to install the software. Perform the following steps to install the software.

1. Download the appropriate file from the web site. If you are going to use the unzip utility download the JH.zip file from the site. Likewise, if you are using either the gunzip or uncompress utilities download the JH.tar.gz or JH.tar.Z, respectively.

[3]This is a relatively old machine by today's standards

[4]We will use folder, directory, home folder interchangeably in this User's Guide

2. Place the archive file of your choice (JH.zip, JH.tar.gz, or JH.tar.Z) in a folder named "java" in your home directory. To make the directory use

```
hostname% cd
hostname% mkdir java
```

3. Move to the "java" folder. For example, use

```
hostname% cd
hostname% cd java
```

4. Uncompress the archive file you downloaded. That is, use the command

```
hostname% unzip JH.zip
```

   for the zip'd file, or

```
hostname% gunzip JH.tar.gz
```

   for the gzip'd file, or

```
hostname% uncompress JH.tar.Z
```

   for the archive file in the unix compress format.

5. In the case of the zip'd file you are done already. However, if you use one of the other two formats you need to extract the file using

```
hostname% tar xvf JH.tar
```

   for either of the other two files you downloaded.

At this point the software is available for you to use. The jar file and the appropriate executable script to start the software are available.

2.3.2 A Note about C-shell and the Scripts Included

The scripts for installation and running the interface are written in C-shell. So, you will need to run the scripts after executing the csh command. If you want to use a different shell you will need to modify the script included with the software. If you get something running, let the author know and I will include the scripts in future incarnations of the package. For example, if someone is kind enough to write a ksh or tsh version of the scripts it would be great.

2.3.3 Windows Installation

The installation in this case is simple. If you have downloaded the file over the internet you need to do the following.

1. Uncompress/extract the file you have downloaded using the appropriate command or utility. The result should be a folder named "JHomogenizer"This folder contains the entire package.

2. Move the entire folder to the *Program Files* folder on the C drive on your computer.

3. (Optional) You can move the JRE included in the package to the *Program Files* folder. To move the JRE to the appropriate location all you need to do is to use Windows Explorer to go into the *Program Files* folder and then into the JHomogenizer folder. Cut the Java folder, move up one level to the *Program Files* folder, and paste the Java folder. You can also install the JRE from Sun Microsystems Inc. to your PC. The JRE downloaded directly from Sun will automatically be installed in *Program Files*. However, you risk using a JRE that was not used to develop the package. This may cause some problems.

4. (Optional) Create a shortcut to the software on your desktop. The executable for starting up the is in this folder and is a Win32 batch file named JHomogenizer.exe. It will appear in Windows Explorer as a batch file icon or JHomogenizer.exe in the list format. Right click on this file and click on the Create Shortcut option. Finally, drag the shortcut icon to your desktop.

Once this is done the software is ready to run. Double clicking on the executable icon for the JHomogenizer.exe file will start up the software. The driver program will look for the JRE and the jarfile containing the Java application for the interface.

If you have obtained a copy of the CD and want to install the software on your PC the steps are about the same.

1. Insert the CD into a CD drive on the PC.

2. Use Windows Explorer to access the CD. Go to the WINDOWS folder and click on the JHomogenizer folder.

3. Click on the Copy option on the Edit menu at the top of the Windows Explorer window.

4. Using Windows Explorer go to the *Program Files* folder on the C: drive.

5. Paste the folder into the folder. The JHomogenizer folder will be copied to this location.

6. (Optional) Do the last two steps described in the case above when the software was downloaded over the internet.

Again, the software is ready to run.

2.3.4 Macintosh Installation

The software will also work on Macintosh computers. However, the primary effort in the development has been for unix, linux, and windows platforms. The only way I want to release the software at this time is on a CD. To get the CD use the instructions above. Once you have the CD, copy the folder on the CD to the "Macintosh HD"drive. A script is included in the main folder that will start up the interface.

2.4 A Note on g77

You might be tempted to download and use the Fortran 77 compiler from the GNU site and use the g77 compiler in place of other Fortran 77 compilers. At this point in time the author recommends against this practice. During a portion of the development the g77 compiler was used and failed to compile some of the native codes. The g77 compiler failed to compile some of the standard f77 statements in the native code. When the Sun Microsystems Inc. fortran compilers were used, there were no problems.

3 Homogenization/Averaging Methods Implemented

Users of the software can compare a variety of methods of homogenization in the interface. The hope is that all of the methods will work. In addition, we will include a library of papers that can be used to understand the methods.

3.1 What is Currently Included

A number of methods come with the interface in a canned form. Some of the methods have been developed into Java Averaging Objects. For example, an arithmetic averaging object (ArithmeticAverageObject.java) is implemented. In addition, the following averaging objects are implemented.

1. ArithmeticAverageObject - implementation of the arithmetic average.
2. HarmonicAverageObject - implementation of the harmonic average.
3. GeometricAverageObject - implementation of the geometric average.
4. AperiodicWaveletAverage2D -
5. AperiodicWaveletAverage -
6. BVidalFracCodeDispAverage -
7. BVidalFracCodePermAverage -
8. HomCodeAverage -
9. HomogABAverage -
10. PeriodicWaveletAverage2D -
11. THLBCodeAverage -

When you start up the software the available methods will be listed in the upper left hand portion of the interface. You should expect to see the list above plus a few more as time goes on. The listed averaging methods will be augmented in the future with more methods from users and the author.

A few the current methods include use of wavelet analogies developed in research done by the author. For example, the PeriodicWaveletAverage2D.java object implements the standard periodic homogenization developed by Bourgeat, [3] and Amaziane [1]. Implementation details are given in [2].

3.2 What is Coming

The hope is that users of the software will include their own methods, test their methods, and finally send the methods to the author to include in future incarnations of the package. In addition, it is hoped that the users will include references and papers describing their work.

4 Java, Averaging Objects, and Multithreading

The interface and all of the averaging objects that are written in Java have been developed within an object oriented structure. This implementation allows different averaging methods to be applied to different user specified regions in a given map. That is, you can apply the arithmetic average to one region, the harmonic average region to another region, and periodic homogenization to yet another region to produce a final result. The regions must be rectangular in shape to match the data structure in the software.

The interface allows the user to set the methods used in various subregions. When choosing the averaging method as described later you can choose a block by block setting. A dialog pops up that allows you to click on a subregion and set the method in the region.

In addition, the code is multithreaded which allows separate regions in the original map to be averaged in parallel. This provides an advantage if you run the software on a multiprocessor computer. We have a 10 processor Sun Microsystems Inc. computer that takes advantage of this feature. The code is set to limit the number of processes spawned to 10. However, in theory this number can be set to any number you like provided the computer has enough memory to handle the extra objects, each of which has a copy of the data in the subregion.

4.1 Averaging Objects

Some of the methods discussed above have been implemented into extensions of the abstract Java object AverageObject.java. This object is described in Appendix B. Each averaging object (e.g., ArithmeticAverageObject.java) implements a set of methods that allow input and output of data and a method for computing the average value of the data internal to the object. If you want to include an averaging object in the package using a Java object you must implement all of the methods defined in the abstract AverageObject.java.

4.2 Multithreaded Object Implementation

The JHomogenizer interface uses a multithreaded approach to perform averaging operations. Suppose that you decide to subdivide a heterogeneous map into chunks and compute an average value on each of the portions of the region. The interface will send the appropriate data to a new instantiation of the user specified type of averaging object for that portion of the map. Each object is of type runnable which means that once the averaging object is instantiated and the data is loaded into the object the average will be computed. A thread is instantiated in the averaging object and the averaging process is run. So, each chunk of the heterogeneous map can be averaged independently.

The code is multithreaded which allows each of the objects to go ahead and compute an average value as soon as the data is available. The only restriction is that the code allows a maximum number of threads to run at any given time. This can be modified, but requires that the application be recompiled. The default setting is for 10 threads. So, at any given time up to 10 disjoint regions can be in the process of being homogenized. Note that if you have multiple processors on a computer you can take advantage of the parallelism in the hardware in this way.

You may want to change to the maximum number of threads to optimize the performance of the code on your computer. There is not a clean way to do this on a Windows or MacIntosh machine. However, on a unix or linux box the process is straightforward. This can be done by performing the following steps.

1. Move into the “src” folder inside the “JHomogenizer” directory in the software installation folder.

2. Edit the JHomogenizer.java file and find the line containing

   ```
   maxThreads = 10
   ```

3. Change the integer value to reflect the maximum number of independent homogenization processes you want to have running at any given time.

4. Change to the “bin” folder using

   ```
   cd ../bin
   ```

5. Execute the build script to create a new jar file. That is,

   ```
   ./build
   ```

   Again, realize that the script is written in C-shell.

6. Make sure that the new jar file is in the JHomogenizer folder. The script should show the current date when listed with

   ```
   ls -al
   ```

Keep in mind that the more processes you allow to run at a time the more memory the software will need. The problem is that a copy of the data for that block is created along with all of the parts of the Java object.

5 The Map Hierarchy

To allow comparison between various methods of homogenization a map hierarchy has been implemented in the package. For an input map a tree structure is created with the root or level zero map being the input map. Any maps created through actions on a level 0 map in the interface are linked to the level zero map and can be accessed from the level zero map. In addition, the relationship is maintained by a recursive structure in the Java implementation.

All maps are stored in a Java class called a MapObject. The MapObject class keeps track of all the information about the specific map including the domain, grid information, and data for the map. When a map file is opened the MapObject automatically looks for associated maps by looking through the folder in which the original map is stored. If more maps are found, the MapObject creates a vector of associated maps and loads these into the interface. This continues recursively until no more maps are found. The interface displays two additional map levels in addition to the level zero maps. However, this recursion could go on as long as you like.

For the purpose of illustration let's suppose that the file is named perm40.map. Maybe this is a permeability map on a 40×40 grid. Suppose that from this we want to compute a coarse 4×4 permeability map where the values in the coarse grid are average values from the original heterogeneous map. After a homogenization or averaging is performed a new MapObject is created with the homogenized data. The new MapObject is added to an associated map vector in the original MapObject that contains the original heterogeneous map. This associated map can then be worked with in the interface.

Next, we could choose to create a 4×4 map from the original map file perm40.map that contains the harmonic average on each piece. Suppose that we name this file hrmnc4.map. This would be stored along side the arithmetic average map arith4.map as a level one map. You can then compare the values to see how things look for different averaging methods.

Now suppose that we decide to compute the arithmetic average of the 4×4 map and name the new map arith1.map. The resulting map is a level two map and will be stored relative to the map arith4.map. You can mix and match methods any way you want, but there are certainly combinations that aren't too helpful to look at.

Examples of the process of setting up and creating files will be given below. The map hierarchy can also be used to store related files such as pressure distributions and flow fields that result from computing solutions of differential equations. More on this later.

5.1 The Root or Level 0 Map

When the interface is started up you will typically load or import a map or create a map using the MapEditor tool. Any map that is loaded or imported into the interface or created by the MapEditor tool and retrieved by the interface will be included as a level 0 map in the interface. This is a root map in the map hierarchy. Any maps generated off of the level map will become part of the tree structure that begins with the level one map.

5.2 Associated or Level 1+ Maps

Any map that is the result of the averaging or computation of elliptic solutions is considered an associated map and store in an appropriate location. The maps will appear in the Level 1 and Level 2 lists in the interface. The software is really not set up to handle more levels in the interface. Theoretically, this is possible. The problem is that the interface is not set up to display more than two associated level. The author thinks he knows how to implement this, but it will take a bit of coding to shift the levels. In any case, having two levels allows for two levels of homogenization and the computation of elliptic solutions between an original map and one level of homogenization.

5.2.1 Elliptic Solution Maps

As described later in this document you will be able to compute solutions of elliptic problems associated with the maps loaded in and created in the user interface. The output from the elliptic solver is retained as a map associated the map used to generate the elliptic solution. These maps will appear in the associated map list with specific names. So, for a given coefficient map, the elliptic solver will produce an approximation for a solution of the elliptic problem

$$\nabla \cdot \mathbf{K} \nabla h = f$$

and the gradient of the solution

$$v = -\mathbf{K} \nabla h$$

The approximation of the solution of h is stored in the map named "ellipsol" and the gradient is stored in the three component files "ellipxflux", "ellipxflux", and "ellipxflux".

When a map is loaded into the interface, the elliptic solution maps will also be loaded back into the application and can be viewed as before. You should realize that the interface allows you to change the boundary conditions on the elliptic solution. The code will load in the last computed solution. If you want an approximation associated with different boundary conditions you can recompute the solution in the interface. The next time the maps are saved, the new solution will overwrite the old approximate solution.

5.3 An Example Map Hierarchy

As an example of the map hierarchy, suppose that we have loaded in a map named elfdata and created a map in the MapEditor tool named unif30x30.

Level Zero Maps	Level One Maps	Level Two Maps
elfdata		
	arith5x5	
		arith1x1
		hrmnc1x1
	hrmnc5x5	
		hrmnc1x1
		ellipsol
		ellipxflux
		ellipyflux
		ellipyflux
	hrmnc1x1	
	arith1x1	
	ellipsol	
	ellipxflux	
	ellipyflux	
	ellipzflux	
unif30x30		
	arith5x5	

6 Using the Interface to Homogenize

The interface can be used to homogenize data sets easily. This section will walk you through how to use the interface to do the homogenization. The basic steps involve (1) building a map using the editor or loading an existing map into the program, (2) setting the parameters for the homogenization (e.g., the block sizes), (3) choosing the homogenization methods, and (4) doing the homogenization.

In the following a map is basically a data set that will be used in the interface. For example, a spatially varying permeability tensor in a flow problem can be sampled at a set of points. These data values define a map for our purposes. We will compute homogenized coefficients from the data values.

6.1 The Basic Steps

The following subsections describe the various operations that you will need to need to carry out with the interface to do the homogenization or averaging of interest to you. The following sections give some specific examples that you can work through.

6.1.1 Starting the Interface

To start the interface, you need to execute the JHomogenizer script from the command line in the Unix/Linux version, double click on the JHomogenizer command file in the Windows version, or on the link for the Macintosh version of the application. The script or appropriate executable can be found in the appropriate directory. In the Unix/Linux setting you may need to modify the script to make sure that the script correctly identifies the location of the interface code in the file JHomogenizer.jar. The executables for the Windows and Macintosh versions of the code are hard-wired and discussed below in the examples.

Note that the interface must be started in the directory in which the work is to be done. The reason for this is that the map hierarchy is set up to work from the root directory for the hierarchy. For example, suppose that you want to build maps and homogenize in a directory named HomogWork. If the directory has already been created, change the working directory to this and then start the interface. The maps that you load must be in this folder and the maps you create will be saved in this folder.

You can import and export files from and to other locations. However, imported and exported files will not be included in the map hierarchy. An imported file can be included as a map in the hierarchy. A copy will be made in the appropriate directory in the hierarchy.

6.1.2 Status Line and Running Information

Once the interface is up and running the first thing you should do is look at the status line that resides just above the buttons that control the interface. This line will let you know what is going on. In particular, if it seems that the interface is hung up you should see if the text in this textfield is changing. If the interface is working on any of the most common task you will see what the code is doing (e.g., loading a map file) and a set of dots that keeps increasing in length. When the task completes, a message to that effect will appear in this line.

If the line is not changing and the interface still seems hung up (e.g., clicking on items and buttons has no effect) the interface may still be working. For example, when certain problems occur (e.g., a file is not found) the interface will pop up a warning dialog box for you acknowledgement. Typically, these pop-ups will take a bit of time to bring up. This is especially true when the interface first starts up. The various types of pop-up windows have been instantiated, but the peers for the objects have not been created in the windowing system. The second time the pop-up window is requested it should come up a bit faster.

6.1.3 Loading a Map File

To load a map file you should change your working directory to the directory above the map hierarchy described in the previous section. For example, if you have a map file named perm40.map this should be contained in a directory named perm40 contained in the current working directory. So, you should currently be in the directory just above perm40 in this example. Start the interface in this directory.

To load the map file perm40.map, you start by clicking on the Load Map button in the JHomogenizer interface. A file dialog choice will pop up allowing you to choose the file. The next step is to double click on the directory named perm40. Once the folder is opened in the file dialog you should see a file perm40.map. To load the file double click on the file name or click once on the file named perm40.map and then click on OK to load the file.

As the the map is being loaded a message to that effect will appear in the status line. Once the map is loaded a message will appear to that effect and the name of the map, perm40 in this case, will appear in the interface in the level zero map list.

6.1.4 Importing a Map File

Importing a file is a bit different than loading a file. The file must be in the same format as a map file you load as described in Appendix A. However, the file can be anywhere on your computer as long as you can access the file. The file will be loaded into the interface and then if you save the map file a copy of the imported file will then be included in the map hierarchy structure as a level zero file that can be worked with.

To import a file click on the Import Map button. Again, a file dialog will pop up. The instructions at this point are the same. Find the folder in which the map file resides and use the file dialog to load the file. Again, this file can be anywhere. In the Load Map option the map file must be in the map hierarchy.

Note again that until you save the file it will not be included in the file hierarchy. So, if you just want to compute some numbers without saving any of the computations this option will allow you to do that. In fact, for this type of situation you can start the JHomogenizer application anywhere.

If you save the maps, the map hierarchy will be created in the directory in which the JHomogenizer application was started.

As in all cases if you exit without saving you will be prompted to see if you want to save the files as part of the hierarchy. If you do not want to save the map hierarchy you should indicate this in the dialog that pops up.

6.1.5 Building a Map File

Once the interface is up and running you can build a map by using the MapEditor tool. To do this click on the Show MapEditor button at the bottom of the interface. In a few seconds a window will pop up that contains a tool for building a map. The tool allows you to build a fully three dimensional map to work with. There are a number of ways to create a map. There are a couple of canned examples. These include buttons to create uniformly and normally distributed values. In addition, you can get a log-normally distributed set of values using the Exp button. Finally, to do some work on fractures, you can build a specific type of fracture proposed by Alain Bourgeat.

To get the data into a map click on the Get All Data button at the bottom of the JHomogenizer window. This will bring the data into the JHomogenizer application. Messages will appear in the Status Line to let you know what is happening. Once the data is loaded you can get rid of the MapEditor tool by clicking on the Kill MapEditor button at the bottom of the JHomogenizer window. Note that this is the same button the started the MapEditor tool. The label on the button changes when the button is used.

A section on using the MapEditor tool is given below. You should look at this section carefully to find out how to build and edit files.

6.1.6 Editing a Map File

You may want to change the map file. You can do this with the MapEditor tool. Click on the map that you want to edit at any level and then click on the Show MapEditor button. You can use the MapEditor tool to edit the map. Then get all the data as described above by clicking on the Get All Data button.

6.1.7 Saving a Map File

Saving is easy. All you need to do is click on the Save Map button. The save will be performed on the current level zero map. The original level zero map and all of the map hierarchy will be saved. This includes the files that are derived from the original map. If you exit without saving the maps you will be prompted to see if you want to save the maps in the level zero list one by one.

6.1.8 Exporting a Map File

Exporting a file is similar to saving. However, you can export any file to any directory location. The map file will be saved into a file name of your choice in any directory you want. This might be helpful if you want to use a homogenized file as input to a reservoir simulator or some other application.

6.1.9 Setting the Homogenization Method

After you have a map in place you will need to tell the code how the homogenization is to be done. To choose a method of homogenization you can just click on the desired method in the list in the upper left portion of the application window. The first option is to set the method of homogenization block by block. The rest of the options implement some sort of homogenization using Java objects or native code implemented through a Java interface.

If you choose the first option when you click on the Homogenize button as described below a window will pop up that allows you to set the homogenization method in each block separately. This will be described below. If you choose one of the specific homogenization methods listed, the method will be applied in all of the blocks in the coarser map. In this case each block will be considered to have a local problem defined on the block and the specific homogenization method chosen will be applied to the local problem.

6.1.10 Setting the Homogenization Block Size

Once the map is loaded into the interface and the method of homogenization has been chosen you will need to specify the number of blocks that will be homogenized. The number of blocks in each coordinate in the original map are shown in the top text fields on the right side of the interface. These cannot be changed unless you edit the file in the MapEditor tool. The line of text fields just below these allow you to specify the number of blocks in the output map.

Note that the numbers entered into the text fields must be an integer divisor of the values in the text fields showing the number of blocks in the original map. For example, if the original map has 24 blocks in the first coordinate direction and the 36 and 2 blocks in the other two coordinate directions, then the values entered into the text fields for the number of homogenization blocks must be integer divisors of 36, 24, and 2 respectively.

The bottom text fields allow you to set a number of computational refinement blocks in each coordinate direction. In some of the methods implemented it is necessary to refine the computational grid to make sure that the approximate numerical solutions obtained in the homogenization method are accurate. So, you can refine this. One should be careful in using a large computational refinement. This is especially true when a method implemented in native code (e.g., Fortran or C). The computational refinement may overwhelm the storage set out in the native code. Fortran 77 is notorious for having this problem since the code must set up the storage up front. It is up to the user to make sure that the native code handles storage correctly.

6.1.11 Computing the Homogenized Values

Computation of the homogenized values is easy. Clicking on the Homogenize button will start the process going. As the homogenization is going along you will see messages pop up letting you know what is happening. In the end you will see the homogenized values in the text area in the middle of the application. You will have to give the resulting homogenized map a name. A little input window will pop up to let you do this. The name will appear in the appropriate location in the hierarchy level.

If the method chosen for homogenization is the Setting Block By Block choice from the list the block by block set up will be handled at the beginning of the homogenization process. A dialog window will pop up and let you set specified methods of homogenization on each of the coarse grid blocks. An example of this will be given below.

6.1.12 Viewing the Results of the Homogenization

The interface will not display the numerical data unless you want this to happen. If the current map file is large, the amount of data displayed and the amount of time needed to display the data may be a problem. It makes sense to display small maps and maps created by homogenization to see what the homogenization produces. However, looking at a whole bunch of data is not very informative. You can use the graphics facilities for the elliptic solution to see the effect of the homogenization or the MapEditor tool to see the maps.

6.1.13 Saving the Output TextArea Information

We can save the information in the output textarea at any time by clicking on the "Print Output Area". This will prompt you for a file name in which the text in the output area will be saved. This file can then be printed. For now this is as close to direct printing as the software will get until the Java environment provides a platform independent manner in which printing can take place. Efforts have been made in this direction. However, the way this will be done is still not completely settled yet.

The main idea of this option is to allow you to get a hard copy of the information contained in the output area. This may or may not be important to you.

6.1.14 Exiting

To exit, click on the Exit button. If there are maps that have not been saved you will be prompted to see if you want to save the map file and all of the map hierarchy.

Off you go now to the examples.

6.2 Example One: The Basics

In this section we will go through the process of building a map file, viewing the file, and producing a simple homogenized result. We will also save the map file and also play around with the map hierarchy as described earlier in this user guide.

6.2.1 Starting Up the Homogenization Tool

The first thing to do is to figure out where you want to store the maps. Change to the folder in which you want to save your data. For example, on my computer I would create a directory "homog_work" in my main directory. So, on my Solaris machine I would be in the folder

```
/export/home/koebbe/
```

and would make the work directory by doing the following.

```
cd
mkdir homog_work
```

The first command will put me into my home directory and the second makes the directory for the work I want to do.

Next change into the appropriate directory using

```
cd homog_work
```

If you use a similar set of commands you will be in a position to work and save data in an appropriate place.

To start the tool type in the command that will execute the script to start up the application. The command would look something like

```
/export/home/koebbe/java/JHomogenizer/JHomogenizer
```

This assumes that the application script is stored in the directory

```
/export/home/koebbe/java/JHomogenizer/
```

The installation instructions are described earlier in this document.

If all goes well the application should start running and you will be able to continue along with the example. If this does not work you should make sure the software is installed in the appropriate place and that you type the appropriate command.

Note: If you know how to set up an alias for the executable it can save you a lot time in starting up the software. I personally have an alias set up in my .cshrc file the defines JH to be the appropriate command to start up the software. The alias I have in my .cshrc file is

```
alias JH $HOME/java/JHomogenizer/JHomogenizer
```

Then after changing to the work directory I just type

```
JH
```

to start things up.

6.2.2 Building the Map File

There are a number of ways to build a file including getting the data from a separate application. However, to build a file within the application the process starts by showing the MapEditor tool. To show the MapEditor click on the button at the bottom labeled "Show MapEditor". It may take a lit bit of time, but a new window will pop up that has a map editing/viewing tool in it. To learn more about the MapEditor tool see the section included in this guide that describes the map editor tool.

Once the MapEditor window pops up the next step in this example is to create a uniformly distributed map. In the map editor, click on the "Uniform Random Map" towards the bottom of the MapEditor window. You will see the window display the values in a plot at the top of the window. Dragging the cursor over the map will display the values in the little pop up that comes with the MapEditor. The values will be uniformly distributed values between zero and one.

At this point we have a map in the MapEditor that we can work with. We just need to get the map into the Homogenization Tool. This is done by clicking on the "Get All Data Button" at the bottom of the JHomogenizer window. This will get a copy of the data from the MapEditor tool into JHomogenizer. You should notice that the text fields for the "Het. Input" will fill in with nonzero values. For this example, the values in these fields will be 20, 20, and 1.

6.2.3 Setting the Map File Name

As part of the process of getting the data you will be asked to input a file name for the map. You should realize that the name you input will be used to create a directory in which to store the original map we have created and in addition, store all maps derived from the original. That is, when we do an averaging on the original data we will generate a new coarser map. The data associated with this coarser map will be stored in a subdirectory of the directory associated with the original map.

So, in this example, we will name the original map

```
uniform20x20
```

To follow the example enter this name in the input dialog that pops up. Note that the directories will be created when the map files are saved.

You will see that the map name will appear in the text area in the middle column of the top part of the JHomogenizer window. In addition, as long as the file is not too large the data will also be displayed in a text form in the bottom half of the JHomogenizer window.

6.2.4 Setting the Homogenization Parameters

Now that the data is in the JHomogenizer tool we will need to set the parameters for the homogenization. In the Het. Input fields the number of blocks in the coordinate directions in the original map are given. We need to specify the number of blocks in the coarser map. Note that the number of blocks in each coordinate direction must be an integer divisor of the associated number of blocks in the original map. For our example, we will enter 4, 4, and 1, respectively (4 divides 20). The result is that a total of 16 local problems, each defined by a 5 by 5 set of blocks will be solved. If we left the values at 1, 1, and 1 the tool will compute the average value for the entire region that is currently in the JHomogenizer tool.

We could also increase the number of blocks use in the approximate solution of the local problem. By changing the values in the "Comp. Ref. Input" text fields we can get more accuracy in the numerical solutions of the local problems. For now we leave these at 1 which means no computational refinement. The averaging methods used in this example do not use the refinement parameter. So, we do not have to set this to anything other than 1.

6.2.5 Setting the Homogenization Method

The last thing to do is to set the homogenization method for the local problems. Choosing one of the methods will impose this method on all of the local problems. In this example, we will use the Arithmetic Average choice. So, click on this line in the list in the upper left portion of the JHomogenizer window. The entry should be highlighted. Note that if no choice is made the code will not do any work and returns a warning message that will let you know that you need to set the homogenization method.

6.2.6 Homogenization!

Now just click the Homogenize! button. The application will work its way through the local problems computing the average. In this example, the arithmetic average will be used to compute the average value for each local problem in lexicographically in order of the coordinates. You will see the work being done as things move along. As a coarse block defined by the local 5 by 5 region is processed a message is printed in the bottom half of the window. When all of the regions have been processed the application will prompt you for a map name. In this case, we will use the name "arith4x4". Type this in an continue.

Once the name is input the computed values will be displayed in the lower half of the window. In addition, the name of the new map will be displayed in the second text area in the middle of the top half of the window. Note that we can work on any of the maps in the current set up by clicking on that map. So, we could actually average the map named arith4x4 by clicking on the name and then modifying the homogenization input parameters.

6.2.7 Save Your Work

To save the data including the original map and all associated maps all you need to do is click on the "Save Map" button. The application will save all maps and create any necessary subdirectories for the associated maps. For the current example a directory named "uniform20x20" will be created along with the subdirectory "arith4x4". A file named uniform20x20.map will be created and stored in the top directory uniform20x20. A file named arith4x4.map will be created in the directory arith4x4 insider the directory uniform20x20. Take a look inside the current working directory to see the files and folders created.

6.2.8 Viewing the Coarser Map

You can display a crude version of the map by clicking on the name in the lists of original and associated files. The application will pop up a window showing the structure of the maps.

6.2.9 All Done

Now that we are all done we need to get out of the application. The obvious way is to click on the "Exit" button. If the files have not been saved since the last modification, you will be prompted to see if you want to save the files. Clicking on the save option will save the files and the other two options are clear. The structure of the averaging files is kept. That is, the original file defines a root folder in which the original file is stored. The files derived from this file are stored in subdirectories of the original map file directory.

And we are done with the first example. The next example will use the files you have just created to do more homogenization work.

6.3 Example Two: Loading Map Files

In this section we will load the map file created in Example One. Then we will work through a couple of short exercises.

6.3.1 Starting Up the Homogenization Tool

First change directories into the working directory. Based on the previous example change to the directory Homog_Work in your main directory. This is where the map from the previous example was saved.

As before, start the application up by typing in the command to execute the script.

```
/export/home/koebbe/java/JHomogenizer/JHomogenizer
```

assuming again that the application script is stored in the directory

```
/export/home/koebbe/java/JHomogenizer/
```

As usual, the software should start up and the window should appear on the screen.

6.3.2 Loading a Map File

When a map file is loaded the original map and any associated maps are also loaded. In the previous example we created a map named uniform20x20 using the MapEditor tool. An associated coarser scale map was created via the application of arithmetic averaging. The associated level one map was called arith4x4. Let's load this map structure into the software.

Start by clicking the "Load Map" button at the bottom of the JHomogenizer window. A file dialog will pop up to allow you to choose a file. If you started the Jhomogenizer application in the appropriate directory the file dialog will display the directory uniform20x20. Double click on the uniform20x20 entry in the folder. The file dialog should then display the contents of the folder which include the file uniform20x20.map. Double clicking on this file will load in the maps including the associated map arith4x4.map from the folder arith4x4.

Note that if you try to go down another folder and load the map arith4x4.map in the folder arith4x4 you will run into a file not found error. You could choose to load in the associated map by importing the file or by starting up the software in the uniform20x20 folder. Then load in the arith4x4.map file as described above. Importing a file will be described below.

6.3.3 Doing More Homogenization

After the files load the names of the map files will appear in the lists in the upper middle portion of the JHomogenizer application window. You can make any of the maps current by clicking on the map name in the list. The entries in the heterogeneous block parameter textfields will change to reflect the values for the map.

For this example, click on the map uniform20x20 to make it active. The crude map display window will pop up for the map. This may actually take a little time if the file is large. In any case, you can also look at the map in the MapEditor tool. To do this, click on the "Show MapEditor" button. The MapEditor window will pop up with the current map displayed. You can look at the map or edit the map. We will do more of this in later examples.

For this example, let's compute couple more averages. After making the map uniform20x20 current click on the Harmonic Average item in the list of choices in the upper left hand corner of the JHomogenizer window. Set the number of homogenization blocks to 5 and 5 for the first two coordinates. Then click on the "Homogenize" button to complete the homogenization calculations. Messages describing the process will be output to the JHomogenizer window. When the homogenization is done you will be asked to name the map file. Let's use "hrmnc5x5". The map name will appear the level one list of maps.

To do a bit more work click on the arithmetic average map, arith4x4, to make it current. Then click on the "Arithmetic Average" choice in the list of methods of homogenize. To compute the arithmetic average for the entire domain, click on the "Homogenize" button. The code will compute the arithmetic average and then ask for a map name. Let's use the name "arith1" The name of this map will appear in the level two map list in the interface.

6.3.4 Saving the Map

Again, we will need to save the maps for later use. To do this you just need to click on the "Save Map" button. The current level zero map (uniform20x20 in this example) is saved along with all of the associated maps will be saved.

6.3.5 Done Again

To quit the application just click on the "Exit" button and the application will close. Again, if you have not saved maps the application will prompt you to see if you want to save the maps.

6.4 Example Three: Setting Homogenization Methods

The application lets you apply different methods to different regions of the original heterogeneous map. This example will walk you through this process. In this case, we will assume that you already have the application running. If you don't have it running and need to know how to get it running see the previous two examples.

6.4.1 Building a Map - Same As Example One

There are a number of ways to build a file including getting the data from a separate application. However, to build a file within the application the process starts by showing the MapEditor tool. To show the MapEditor click on the button at the bottom labeled "Show MapEditor". It may take a lit bit of time, but a new window will pop up that has a map editing/viewing tool in it. To learn more about the MapEditor tool see the section included in this guide that describes the map editor tool.

Once the MapEditor window pops up the next step in this example is to create a uniformly distributed map. In the map editor, click on the "Uniform Random Map" towards the bottom of the MapEditor window. You will see the window display the values in a plot at the top of the window. Dragging the cursor over the map will display the values in the little pop-up that comes with the MapEditor. The values will be uniformly distributed values between zero and one.

At this point we have a map in the MapEditor that we can work with. We just need to get the map into the Homogenization Tool. This is done by clicking on the "Get All Data Button" at the bottom of the JHomogenizer window. This will get a copy of the data from the MapEditor tool into JHomogenizer. You should notice that the text fields for the "Het. Input" will fill in with nonzero values. For this example, the values in these fields will be 20, 20, and 1.

6.4.2 Setting the Homogenization Method

Let's build a coarse scale version of the map from the previous subsection. Set the number of homogenized blocks to 2 and 2 in the first coordinate directions. Then choose the "Set Block by Block" homogenization option in the list. Clicking on the "Homogenize" option will start the calculations. The next thing that will happen is that a dialog will pop up with the coarse grid displayed in the upper left hand corner of the dialog. Also, the list of methods available will be displayed on the right. To set a method in a block click on the method you want and then click on the block in the coarse grid representation. The block will change to the color associated with the method chosen. You can then choose the methods any way you want. Note that you can click and drag to set multiple blocks.

Once you are done setting the blocks click on the "Okay" button and the work will begin. Messages will appear in the JHomogenizer window that indicate the block being averaged and the method being used.

6.4.3 Done Again

In this case, we are done with the example. So click on the "Exit" button to finish. You will be prompted to see if you want to save the files. Do what you like here and on to the next example.

6.5 Example Four: Importing and Exporting Files

It is important to be able to work on files not included in the map structure and to be able to export the output of the work to files that can be used by other applications.

6.5.1 Importing a File

Start the JHomogenizer application as in the previous example. Then click on the "Import Map" button at the bottom of the JHomogenizer window. A file dialog will pop up to allow you to choose a file. In this case, the import will only load in the file you choose and will not include the file as part of the map hierarchy unless you save the file. The map in the file will be treated as a level zero map.

6.5.2 Exporting a File

To export a file start by making the map of interest current. Then click on the "Export Map" button at the bottom of the JHomogenizer window. A file dialog will again pop up and allow you to choose a file in which to save the file. This will not save the file as part of a map hierarchy. That way you can place the file anywhere that you want and then use the file as input to other applications.

6.6 Example Five: Batch Homogenization

The interface allows you the ability to compute homogenized values for a number of files that reside in the same directory. If you have placed a number of files in a directory by clicking on the "Batch

Homogenize" you can compute the average value for the data in each of the map files in the directory you specify.

6.6.1 Setting Up the Averaging Method

Assuming that you have the interface up and running, start this example by choosing the method you want to apply to all the files. For this example, let's choose the arithmetic average which is the second choice. Click on this choice and proceed to the next step.

6.6.2 Getting Some Data

Now show the MapEditor tool by clicking the "Show MapEditor" button. Once the button is up and running, create a uniformly distributed data set by clicking on the "Uniform Random Map"button. The map will appear in the display in the MapEditor tool. Next, click on the "Get All Data" button in the JHomogenizer tool. Set the map name to uniform20.

Now generate a second map set. Start by changing the number of blocks to 30 and 30 in each coordinate direction. To do this click on the "Build A Map" button in the MapEditor tool. A dialog will pop up that allows you to enter the number of blocks in each direction and the background value. So, set the number of blocks in the first two coordinates to 30 and then set the background value for the map to 1.0. Finally, click on the "OKay" button to complete the map definition.

The next step is to click on the "Normal Random Map" button to generate data in the map with normally distributed values. Now we have a 30 by 30 by 1 map with data in it. Now, click on the "Get All Data" button for a second time and name the map normal20 when the dialog pops up.

At this point, we need to save the data sets. If we save the maps the map hierarchy will get in the way of the example. So, instead of a simple save let's export the files to the current working directory. So, click on each of the two maps listed in the JHomogenizer application and then click on the "Export Map" button and give a name for each of the files. Let's call the two files uniform20x20 and normal30x30, respectively. Note that this will create the two files uniform20x20.map and normal30x30.map in the current directory.

6.6.3 Doing the Batch Homogenization

Now we can do the batch homogenization. Click on the "Batch Homogenization" button to start the process. A file dialog will pop up. Double click on the name uniform20x20.map to get things going. The batch homogenization will start up and the results of each of the averaging calculations will appear in the output text area in the JHomogenizer application.

6.6.4 Outputting the Results

To get the output to a file and then possibly a printer we can click on the "Print Output Area" button to bring up a file dialog box. Then type in the name of a file in which the results will be stored. For this example let's type in "BatchResults.out". This file will contain text that looks about the same as the text in the output area in the JHomogenizer application.

7 Adding Methods and Modifying Existing Methods

The interface implements a number of methods for use. This section describes how to interface your codes written in a variety of languages that implement homogenization methods. Even though details of how to modify the existing codes are given it is strongly recommended that you do not modify the code. Instead, if you feel that you want to make modifications, copy the code into another object or native code (e.g., Fortran, C, C++, etc.), rename the file to reflect the fact that it is different and then follow the instructions for adding methods to the interface.

You will need to have the source code for the interface available for editing to include the homogenization methods you implement. So, you should copy the the source directory into an appropriate location for editing and compiling.

7.1 Adding A Java Object

A number of the methods have been implemented in Java Objects as mentioned earlier in this document. The structure of the Java Objects is to implement the methods in a separate self-contained piece of code within a Java object.

7.1.1 Creating the Java Object

The steps for developing an averaging object are the following.

1. Make a copy of an averaging object such as ArithmeticAverage.java.

2. Modify the name of the constructors to be the same as the name of the file. For example, if you choose a file name of MyAverage.java in the first step then the constructor names must be MyAverage. The lines

```
public class ArithmeticAverage extends AverageObject {
public ArithmeticAverage(Double[][] d,int nx,int ny, int nz, int t)
public ArithmeticAverage(Double[][] d,int nx, int ny, int nz, int t,
                         Double xl, Double xr,
                         Double yl, Double yr,
                         Double zl, Double zr)
public ArithmeticAverage(Double[][]d, int nx, int ny, int nz, int t,
                         Double [] xg, Double [] yg, Double [] zg)
public ArithmeticAverage(Double[][]d, int nx, int ny, int nz, int t,
```

```
                                  Double xl, Double xr,
                                  Double yl, Double yr,
                                  Double zl, Double zr,
                                  Double [] xg, Double [] yg, Double [] zg)
```

in ArithmeticAverage.java should read as follows.

```
public class MyAverage extends AverageObject {
public MyAverage(Double[][] d,int nx,int ny, int nz, int t)
public MyAverage(Double[][] d,int nx, int ny, int nz, int t,
                 Double xl, Double xr,
                 Double yl, Double yr,
                 Double zl, Double zr)
public MyAverage(Double[][]d, int nx, int ny, int nz, int t,
                 Double [] xg, Double [] yg, Double [] zg)
public MyAverage(Double[][]d, int nx, int ny, int nz, int t,
                 Double xl, Double xr,
                 Double yl, Double yr,
                 Double zl, Double zr,
                 Double [] xg, Double [] yg, Double [] zg)
```

If you do not make these changes you will have problems compiling the objects you create.

3. Modify the computeValue() method in your averaging object to reflect the computations needed to compute the averaging method you are implementing.

4. Change all of the constructor invocations in the main() method near the bottom of the object you have created. If you made a copy of the file ArithmeticAverage.java you will need to change all occurences of the string "ArithmeticAverage" to the name of your object. For example, if the name of your object is "MyAverage" you will need to change all occurences of "ArithmeticAverage" to "MyAverage". So, search for the main() method and change the appropriate lines.

These steps are fairly simple and the code will basically look like C or C++ code. Keep the return type in mind that the object you create is runnable and observable. Once the object is instantiated (created by the interface) the object will run on the data and compute the value automatically. The object will notify the interface that the computation is complete. This is done in the line

```
notifyObservers();
```

at the end of the computeValue() method. Do not remove this line when you create the averaging object implementing your method. If the notification line is not included the code will not work. In fact, it will most likely crash the code since when the object is instantiated the interface will never be able to tell if the process of computing the value is complete. So, if you use this method it will create objects until the maximum number of threads is attained and then since the objects never appear to complete the code will just sit there with nothing happening.

The return type of the method is boolean. If the method fails you should return false. However, the last line must be

```
return true;
```

This will tell the interface that the computation was successful. Note that you are responsible for catching any errors in your averaging methods.

As a final piece of information to keep track of in the development of your Java AveragingObject, the computeValue() method should create an array of nine values that represent the entries in a three dimensional tensor. The values are assumed to be stored in the following manner.

```
kxx:     value[0]
kxy:     value[1]
kxz:     value[2]
kyx:     value[3]
kyy:     value[4]
kyz:     value[5]
kzx:     value[6]
kzy:     value[7]
kzz:     value[8]
```

where the tensor is defined by

$$\mathbf{K} = \begin{bmatrix} k_{xx} & k_{xy} & k_{xz} \\ k_{yx} & k_{yy} & k_{yz} \\ k_{zx} & k_{zy} & k_{zz} \end{bmatrix}$$

It is important to leave things this way since the interface accesses the computed data in this format. Use ArithmeticAverage.java as a template.

7.1.2 Testing Your Java Object

After developing the method you might want to test the method. This requires the use of a Java compiler on the Java object. Suppose again that the object is named MyAverage.java. The main() method contains code to test the method. As long as you only change the names of the methods appropriately, running the object as a stand-alone Java application will output all the appropriate computed values.

To test the object do the following:

1. Compile the object using a command like:

   ```
   javac MyAverage.java
   ```

   This will create a file named MyAverage.class.

2. Next run the code using a command like

```
java MyAverage
```

This will output the values computed by the object.

3. Make sure the output is what it should be.

This test will make sure that the code you have written will work in the JHomogenizer interface. If the main() method works and produces good results the interface should be able to interact with your object.

7.1.3 Getting the Java Object into the JHomogenizer Interface

Once the object has been created you will need to add the object to the list of objects in the interface. This requires a little bit of care. You will need to modify the main routine in the source code, JHomogenizer. So, to do the next steps open up the file JHomogenizer.java in your favorite editor. and perform the following modifications.

1. Modify the array of methods that are contained in the package. The methods array is located towards the bottom of the file. Search for methods or for "Original HomCode" in the file. The easiest way is to search the JHomogenizer.java file for the string "STEP I". Add an appropriate string to the list for your specific homogenization method. Keep track of the location of the item you include. You should really only use the

2. Next head to the homogenize() method. There is a switch statement in the code you will need to find. To get there search for the string "STEP II". You will need to add your option to the switch statement. For example, if your method is named MyAverage as described above you will need to add lines like

   ```
   case 1:
     averageObject[index] =
       new ArithmeticAverage(d, nxhet, nyhet, nzhet, l-1);
     break;
   ```

   This is the way the interface sends data from a block to the arithmetic averaging Java object that comes with the application. So, for example, you might end up adding the lines

   ```
   case 10:
     averageObject[index] =
       new MyAverage(d, nxhet, nyhet, nzhet, l-1);
     break;
   ```

   if the your method appears as the tenth item in the methods array as modified in the first step this is appropriate.

At this point the method you implemented should be ready to go.

7.1.4 Rebuilding the JHomogenizer Interface - Unix/Linux

You can rebuild the code by reinstalling the code. However, do not use the installation script on the CD or in the package as this will overwrite the JHomogenizer.java file and undo your work. Instead, you should use the build script that gets installed when you run the script. You can do this on a Unix machine by executing the command

```
./build
```

from within the bin directory. When the script is done running the new version of the interface will be available to run.

7.1.5 Rebuilding the JHomogenizer Interface - Windows

7.1.6 Rebuilding the JHomogenizer Interface - Macintosh

7.2 Native Codes

By native codes we mean programs written in a high level language like Fortran or C. The interface works with Fortran 77, Fortran 90, C, and C++ codes written by users of the program. This section works through some examples of high level language implementations of homogenization methods.

Note that the native code you produce should take input defining values of a three dimensional tensor of the form

$$\mathbf{K} = \begin{bmatrix} k_{xx} & k_{xy} & k_{xz} \\ k_{yx} & k_{yy} & k_{yz} \\ k_{zx} & k_{zy} & k_{zz} \end{bmatrix}$$

defined on a three dimensional rectangular grid. The output must be a single tensor value. The interface objects you create will output the data in a format defined as in Appendix A. The output produced by your native code should be of the form

```
1           *** number of grid block in first coordinate
1           *** number of grid block in second coordinate
1           *** number of grid block in third coordinate
 0.576659E+00
 0.468313E+00
 0.000000E+00
 0.495063E+00
 0.501637E+00
 0.000000E+00
 0.000000E+00
 0.000000E+00
 0.520427E+00
```

This indicates that the data is defined on a $1 \times 1 \times 1$ block and the nine values are the components of the three dimensional tensor. That is, the corresponding symbolic version of the file is

```
nx
ny
```

```
nz
kxx
kxy
kxz
kyx
kyy
kyz
kzx
kzy
kzz
```

The interface expects things in this format. If you choose to change the format you will be on your own in terms of getting the code to your; both yours and the author's.

7.2.1 Building the Java Object Interface

To interface the native code to the JHomogenizer interface you will need to build a Java object that will allow the interface to communicate with your native code. An example of such an object is HomCodeAverage.java. This object interfaces with a f77 code used to compute the periodic homogenized values on a given region. There are a number of steps to follow in creating an appropriate Java object.

Do the following:

1. Create a copy of the HomCodeAverage.java file with the appropriate name. For illustration purposes let's use the name MyNativeAverage.java.

2. Change all the strings that are used to identify the original method of averaging to the new method. In this example, you will need to search for all occurences of the string "HomCodeAverage"and change these to "MyNativeAverage". Again remember that the name of the file (minus the .java extenstion) must be the same as the class name and the constructor name in the Java code.

3. In the file you have created, MyNativeAverage.java, you will find documentation on how to modify the object to incorporate an interface to your native code. The lines look like

```
// the following list identifies the steps you need to take care of to build
// an averaging object for your favorite method. the steps can be searched for
// using your editor. there is a comment with the step name (e;g;, Step 1) at
// each appropriate location in the code.
//
//   Step 1. Identify the location of the executable image and the names of
//           the input and output files that the executable expects when it
//           is run. these are listed close to the top of this file.
//   Step 2. (optional) Set up methods for setting and getting the location
//           of the executable and input and output file names. If you want
//           to hard-wire these in one place you do not need this step.
```

```
//   Step 3. Set the name of the working directories to be generated by the
//           java code in which to run your executable. the input data file
//           and the shell script automatically generated by this code will be
//           thrown into directories with this name and the tag value at the
//           end. so each process (the code is multithreaded) will have a
//           unique directory in which to work.
//   Step 4. Set up the formatting for the input file to the executable. this
//           is dependent on the format needed by the executable. in the case
//           below a separate method was developed to produce an output file
//           in the correct format.
//   Step 5. Set up a means for getting the averaged values back into this
//           code. The input needs to read in a total of 9 values for this
//           code. So, you need to make sure that the 9 values are set in the
//           input. In some cases, you may not have the information in the
//           code. For example, If your code is a 2D code, you may only have
//           the two component direction values. in this case you might want
//           to set the third component values to 0 on the off diagonal and to
//           some appropriate value on the diagonal, say 1.
//
```

Follow these instructions before proceeding.

7.2.2 Setting up the Native Code Executable

Once the object is set up you will need to compile your code and place an appropriate location so that the interface can use your code. If your code is an f77, f90, C, or C++ code you can place them in the appropriate directory in the src directory. There is a directory for each of these flavors of code. Do the following:

1. Create a directory in the appropriate directory for your code. For example, there is a subdirectory under the src/f77 directory named HomCode. Place your code and any additional files needed in the folder your create.

2. Compile the code using the appropriate compiler. Make sure the name of the resulting executable matches the name you modified into your Java object (e.g., MyNativeAverage). Also, make sure that the path name is set correctly. For the HomCode example, the path is src/f77/HomCode and the executable name is homcode.

Once this is done, the code should be available.

7.2.3 Rebuilding the JHomogenizer Interface

Follow the instructions in the Section 7.1 to rebuild the application. From this point things are the same.

7.2.4 Other Languages

Codes written in other languages can be handled in the same fashion. However, the interface installation and building scripts will not by default compile your code. The only languages that are handled by default are Fortran 77, Fortran 90, C, and C++. Java code is handled as described in the previous section.

8 MapEditor Tool

I will try very hard to put a section in on this later.... Sorry for the problem here...ran out of time for now.

9 The Elliptic Solver

The interface has an elliptic solver object attached to the Java code. The solver computes a numerical approximation of the solution of the elliptic problem

$$\nabla \cdot \mathbf{K} \nabla h = f \tag{1}$$

where the coefficient $\mathbf{K}$ is taken from the current map selected in the interface. So, as an example, a numerical approximation of the elliptic problem can be computed for the original heterogeneous map or for any homogenized version of the original map. In terms of the interface, this means we can compute an approximate solution for the solution of this elliptic problem for the maps in any of the levels. Once the solutions have been computed the results can be compared with other solutions as described below.

The elliptic solver is an implementation of the lowest order mixed finite element (MFE) method (see [5]). The specific implementation takes advantage of a more efficient implementation as described in [4]. In the more efficient implementation auxiliary variables are introduced with a projection approach to handle full tensors in an efficient manner. The system can be reduced by redefining the flux/velocity variable in the system of first order differential equations (see [4]).

The data for the elliptic problem other than the coefficient, $\mathbf{K}$, (e.g., the boundary conditions) is hardwired for the most part. However, the boundary conditions and forcing function can be set to a few different values. The data that can be set is accessible through a button on the interface as described below. The forcing function can be set to zero or to a pair of point sources at the corners of the domain (lower lefthand and upper right hand corners in two dimensions). The point sources are set up to simulate the quarter-five spot pattern that appears in the reservoir engineering literature. The boundary conditions for the problem can be set to no-flow or one-dimensional flows. The one-dimensional flows are set up in a user specified coordinate direction. The no-flow conditions on the boundary are the default and if a point source forcing function is chosen the boundary conditions are reset to no-flow conditions.

The use of the MFE method allows access to approximations of the fluxes

$$\mathbf{v} = -\mathbf{K} \nabla h$$

at grid lines on the computational grid. The solution of the elliptic problem, h, as well as the aproximations of the fluxes at the grid lines are stored if the solution process is successful. The fluxes can be used in a flow solver as described in the next section to see how flow will occur in the heterogeneous and homogenized problems.

The output from the solution process is stored in map files name as shown in the following table

elliptic solution	ellipsol
x component of the gradient	ellipxflux
y component of the gradient	ellipyflux
z component of the gradient	ellipzflux

These map names will appear in the associated map lists with the maps. For example, if a level 0 map is named "elfdata", the solution will be stored in a map called "ellipsol" and will appear in the level 1 list of maps. If you click on these maps they will not be loaded into the interface. The interface keeps track of these specific maps and uses the files differently.

When a map file is loaded into the interface the associated approximate solution and approximate gradient files will be loaded with the other files. These files will be used as the solution of the elliptic problem. The elliptic solution can be recomputed at any time with alternate boundary conditions. All you need to do is to click on the "Elliptic Solution" button. A prompt will appear to see if you really want to recompute the solution. For example, you may want to see the solution based on a quarter-five spot configuration and then when the boundary conditions in a coordinate direction are nonzero. The software will retain and save only the last solution computed.

9.1 Setting the Boundary Conditions

To set the boundary conditions for the elliptic solution process you click on the "BC's" button next to the "Elliptic Solver" button in the interface. A small dialog will pop up that allows you to modify the forcing function or the boundary conditions for the elliptic problem. To change the conditions used in the problem you simply click on the appropriate choice and then click on the "Okay" button on the dialog box. The conditions can be chosen at any time and will be used the next time an elliptic solution is computed. When the dialog box is opened the current boundary condition choice is checked.

9.2 Getting a Solution in the GUI

To get a solution for the flow problem associated with a given map we start by clicking on the map of interest in the lists in the interface. This will make that chosen map the current working map. Then all we need to do is to click on the "Elliptic Solution"button on the interface. The actions associated with the computation of the elliptic solution appear on the bank of buttons just below the middle of the interface. As the work progresses the status line will indicate that the code is working. Once the solution has been computed the status line will indicate that solution has been computed and several new files will appear in the lists. These files are the elliptic solution along with the three components of the gradient. The solution can be retrieved from these files in the future so that we do not have to recompute the solution. When the current working map is saved the elliptic solution and gradient components will also be saved. So, to save the solution values you can click on the "Save Map(s)" button to keep the solutions to the elliptic solutions.

9.3 Grid Refinement Issues

The elliptic solution component of the interface is set up so that the solution of the elliptic problem for a homogenization is performed on the same grid as the level zero or original heterogeneous maps in the interface list. This will give an apples to apples comparison for the behavior of flow through both the heterogeneous and homogeneous maps. In addition, the results from numerous homogenization approaches can be compared to the heterogeneous map. Note that the refinement text areas on the interface do not refer to the computational refinement in the elliptic solution. The refinement values in the text areas refer to the computational refinement used in the computation of the homogenized results be some other numerical method. For example, the HomCode homogenization option allows one to specify a computational refinement in the computation of the approximate solution of the local problem.

9.4 Displaying an Elliptic Solution in the GUI

Once the solution has been obtained we can create graphical representations using the "Show Elliptic Image". This button will build an image that shows the results of the calculations in the elliptic solution. The graphics are created in the plane defined by the first and second variables. If the number of grid blocks in the third coordinate is larger than one the interface will animate the images in the third coordinate direction. That is, images will be created for each slice in the third coordinate direction and displayed in sequence using a timing loop.

The "Start Flow Solver" button will produce an animation of flow of fluids through the map using a simple upwinding scheme. This button cannot be activated until the elliptic solution has been created for the map. The next section discusses this option in the interface.

9.5 Showing Comparisons of Elliptic Solutions in the GUI

Up to three elliptic solutions can be displayed at the same time. On the lower right hand side of the interface a list appears. As elliptic solutions are computed the map names will appear in the list. Once a map name is listed in this area it is availble to be shown in a comparison window with up to two other results. The first three names in this list will be used in the comparison graphics. If more map names are on the list they are not included. After clicking "Show Comparison" will bring up a window with three rows of graphics; one for each map. Once the window comes up you can see a simulation of the flow through the map by clicking on the "Start Flow Solver" button. The results of the approximation for all three maps using the same time steps will be shown in the window.

To remove a map from the list, simply click on the name in the list. The name will disappear and the results of the elliptic solution for this map will no longer be available for comparison with the other maps in the list. If a map has been removed and you want it back on the list, all that you need to do is to click on the map name in the level map lists and then click on the "Elliptic Solver" button. You will be prompted to see if you want to recompute the elliptic solution. Clicking on the "Cancel" button will just reinstate the previous solution and the name of the map will appear again on the comparison list.

To kill the map just click on the button at the bottom of the comparison list which will now read "Kill Comparison". If you click on the "Show Elliptic Image" button, the comparison window will also disappear.

9.6 Exporting a PPM Images for Elliptic Solutions

The images created can be exported in a PPM format as desired. Once the elliptic solution has been computed and the images created clicking on the Create Elliptic Image or Create Elliptic Flux Image will export a PPM formatted version of the image. This image can be viewed in a standard image viewer like xview and translated to other graphics file formats.

10 The Flow Solver

Once the elliptic solution is available (see the previous section) you can take a look at the flow (or transport) through the region based on the coefficients chosen in the interface. To access the flow solution you need to display the elliptic solution using the “Show Elliptic Solution” or “Show Elliptic Comparison” buttons in the interface. Once the graphics window pops up you can start the flow solution calculations. Clicking on the “Start Flow Solver” button will start up the flow solution calculations. This action will also create objects that will be used to simulate the solution of a simple conservation law.

The “Start Flow Solver” button will change into two buttons and a textfield for running the flow computations. The textfield is used to set the number of time steps taken between displaying the results of the calculations. Clicking on the “Next Step” button will go through the number of steps displayed in the textfield and then display the results next to the elliptic solution graphics. The “Quit”button will cancel the calculations. Clicking on the “Start Flow Solver” button again will start the process up again.

10.1 The Simple Conservation Law

The simple conservation law used to show the effect of homogenization on a heterogeneous map is the following.

$$u_t + \mathbf{v} \cdot \nabla f(u) = u_t + \mathbf{v} \cdot \nabla u = 0 \tag{2}$$

The parameter u is some conserved quantity that is moved around the region based on the gradient of the solution of the elliptic equation (1). The function, $f(u)$, is called a flux function and is chosen to be the simple case of $f(u) = u$ which can be thought of as a flux function that represents a linear flood process in a reservoir.

The variable, $\mathbf{v} = -\mathbf{K}\nabla h$, is the gradient of the solution of the elliptic equation (1). As described below, an approximation of the gradient is obtained at the grid block boundaries using a simple mixed finite element method.

10.2 The Numerical Method: A Simple Upwind Scheme

The method used to compute the flow solution is a simple upwind solver. The interface passes the gradient of the solution returned from the elliptic solver to compute grid line fluxes for the simple hyperbolic conservation law defined by (2). The mixed method provides the approximations for $\mathbf{v}$. These grid line values are used to determine the upwind direction and thus the values of u at the grid lines for a simple conservative numerical method. To be specific, let's consider the definition of the numerical method in two spatial dimensions. The discretization used for (2) is of the form

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = -\frac{1}{\Delta x}[(v_x u)_{i+1/2,j} - (v_x u)_{i-1/2,j}] - \frac{1}{\Delta y}[(v_y u)_{i,j+1/2} - (v_y u)_{i,j-1/2}]$$

where $u_{i,j}^n \approx u(x_i, y_j, t_n)$. Also, the values $(v_x u)_{i\pm 1/2,j}$ and $(v_y u)_{i,j\pm 1/2}$ are the fluxes with the values of u at the grid lines are determined by the upwind value of the unknown, u, neighboring the grid line.

The CFL is enforced on all of the calculations. In addition, in the comparison graphics, the CFL condition is computed over all of the maps being compared. So, if three maps are being compared the CFL condition is enforced at each time step by determining the bound over all three of the flow simulations.

In summary, the equation and numerical methods used are simple and only intended to give some idea of how things change when homogenization is used. In the future more complicated equations may be added with better numerical methods for approximating the solutions.

References

[1] B. Amaziane and A. Bourgeat, *Effective behavior of two-phase flow in heterogeneous reservoir*, IMA, 11 (1988), pp. 1–22.

[2] B. Amaziane, A. Bourgeat, and J. Koebbe, *Numerical simulation and homogenization of two-phase flow in heterogeneous porous media*, in Transport in Porous Media II, Hornung, Dogan, and Knaber, eds., Kluwer Academic Publishers, 1990.

[3] A. Bourgeat, *Homogenization method applied to the behavior of a naturally fissured reservoir*, in Mathematical Methods in Energy Research, K. Gross, ed., SIAM, 1984, pp. 181–193.

[4] J. Koebbe, *A computationally efficient modification of mixed finite element methods for flow problems with full transmissivity tensors*, Numerical Methods for Partial Differential Equations, 9 (1993), pp. 339–355.

[5] P. Raviart and J. Thomas, *Mixed finite element methods for 2^{nd} order elliptic problems*, in Lecture Notes in Mathematics Vol. 606, Springer-Verlag, 1977.

A Map File Format

To allow ease of use and the merging of files from other applications the file format for the map files is given in this appendix. In addition, an example of a (small) map file is given. The format is basically to give the number of blocks in each of the three coordinate directions on a separate line and then include the data for the three dimensional tensor by component after this.

A.1 Description of the Map Format

The basic form of the file is the following.

```
nx          the number of blocks in the first coordinate direction
ny          the number of blocks in the second coordinate direction
nz          the number of blocks in the third coordinate direction
kxx         the (1,1) entry constant values defined on the rectangular grid
kxy         the (1,2) entry constant values defined on the rectangular grid
kxz         the (1,3) entry constant values defined on the rectangular grid
kyx         the (2,1) entry constant values defined on the rectangular grid
kyy         the (2,2) entry constant values defined on the rectangular grid
kyz         the (2,3) entry constant values defined on the rectangular grid
kzx         the (3,1) entry constant values defined on the rectangular grid
kzy         the (3,2) entry constant values defined on the rectangular grid
kzz         the (3,3) entry constant values defined on the rectangular grid
```

Note that the data for each component is included. The constant values are given in lexicographical order using the first, second, and third coordinate directions, respectively.

The following is an example of a map file defined on a $4 \times 4 \times 1$ grid. Notice that there are nine sets of 16 numerical values. The first set corresponds to the 16 values of the $(1,1)$ entry in the tensor defining the map.

```
4           *** number of grid block in first coordinate
4           *** number of grid block in second coordinate
1           *** number of grid block in third coordinate
 0.576659E+00  0.468313E+00  0.495063E+00  0.501637E+00  0.520427E+00
 0.395964E+00  0.515561E+00  0.565033E+00  0.487239E+00  0.612207E+00
 0.507643E+00  0.482643E+00  0.455966E+00  0.523867E+00  0.499941E+00
 0.480598E+00
```

```
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00
0.576659E+00  0.468313E+00  0.495063E+00  0.501637E+00  0.520427E+00
0.395964E+00  0.515561E+00  0.565033E+00  0.487239E+00  0.612207E+00
0.507643E+00  0.482643E+00  0.455966E+00  0.523867E+00  0.499941E+00
0.480598E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00  0.000000E+00
0.000000E+00
0.576659E+00  0.468313E+00  0.495063E+00  0.501637E+00  0.520427E+00
0.395964E+00  0.515561E+00  0.565033E+00  0.487239E+00  0.612207E+00
0.507643E+00  0.482643E+00  0.455966E+00  0.523867E+00  0.499941E+00
0.480598E+00
```

To take the illustration above a bit further suppose that we are working on permeability files where the permeability tensor is the parameter $\mathbf{K}$ in the partial differential equation

$$\nabla \cdot \mathbf{K} \nabla h = f$$

If the tensor is of the form

$$\mathbf{K} = \begin{bmatrix} k_{xx} & k_{xy} & k_{xz} \\ k_{yx} & k_{yy} & k_{yz} \\ k_{zx} & k_{zy} & k_{zz} \end{bmatrix}$$

In the data format above the first three lines give the number of grid blocks in the coordinate directions. Then the values of k_{xx} on the entire grid are given followed by the values of k_{xy}, the values of k_{xz}, and so on as indicated at the beginning of this section.

A.2 Code to Import/Export the Map Data

To read and write the data file format as described in the previous section you may need a bit of code. For example, if you have an application that produces a map you will need to format or possibly reformat the output from your application. In the end you will have to decide how to do thing. A couple of examples of code used to input and output the map files are given in this section.

A.2.1 Example Fortran Code

The following bit of code is the code used to output the map file from a Fortran routine.

```
      program main
c
c all values store in a two dimensional array - note that this does not work
c for three dimensional maps
c
      real a(40,40,9)
c
c values of the tensor stored in separate arrays - the values are ordered
c lexicographically in the coordinates
      real kxx(1600),kxy(1600),kxz(1600)
      real kyx(1600),kyy(1600),kyz(1600)
      real kzx(1600),kzy(1600),kzz(1600)
c
c values of the tensor stored in an array that is ordered lexicographically in
c the spatial part of things and then the second index keeps track o the entry
c in the tensor
c
      real tns(1600,9)
c
      real zeros(1600)
      real ones(1600)
      do 1 i=1,1600
        zeros(i) = 0.0
        ones(i) = 1.0
    1 continue


c
c set up the dimensions for the first examples
c
      nx = 30
      ny = 30
      nz = 1
      nnp = nx * ny * nz
c
c isotropic tensor example
```

```
c
c note: the following loops show the relationship between methods of storing
c the array values. the two dimensional and one dimensional examples are shown
c and used to output the data.
c
      do 2 j=1,ny
         j2 = ( j - 1 ) * nx
      do 2 i=1,nx
         ind = i + j2
c
         a(i,j,1) = ran3(ind)
         kxx(ind) = a(i,j,1)
         tns(ind,1) = a(i,j,1)
         a(i,j,2) = 0.0
         kxy(ind) = a(i,j,2)
         tns(ind,2) = a(i,j,2)
         a(i,j,3) = 0.0
         kxz(ind) = a(i,j,3)
         tns(ind,3) = a(i,j,3)
c
         a(i,j,4) = 0.0
         kyx(ind) = a(i,j,4)
         tns(ind,4) = a(i,j,4)
         a(i,j,5) = a(i,j,1)
         kyy(ind) = a(i,j,5)
         tns(ind,5) = a(i,j,5)
         a(i,j,6) = 0.0
         kyz(ind) = a(i,j,6)
         tns(ind,6) = a(i,j,6)
c
         a(i,j,7) = 0.0
         kzx(ind) = a(i,j,7)
         tns(ind,7) = a(i,j,7)
         a(i,j,8) = 0.0
         kzy(ind) = a(i,j,8)
         tns(ind,8) = a(i,j,8)
         a(i,j,9) = a(i,j,1)
         kzz(ind) = a(i,j,9)
         tns(ind,9) = a(i,j,9)
c
    2 continue
c
c output the isotropic tensor
c
c this is the correct format for the map file
```

```
c
      open(unit=11,file='isotropic30.map')
      write(11,*) nx
      write(11,*) ny
      write(11,*) nz
      write(11,10) (kxx(i),i=1,nnp)
      write(11,10) (zeros(i),i=1,nnp)
      write(11,10) (zeros(i),i=1,nnp)
      write(11,10) (zeros(i),i=1,nnp)
      write(11,10) (kyy(i),i=1,nnp)
      write(11,10) (zeros(i),i=1,nnp)
      write(11,10) (zeros(i),i=1,nnp)
      write(11,10) (zeros(i),i=1,nnp)
      write(11,10) (kzz(i),i=1,nnp)
      close(unit=11)
c
c the next is the format that lists the tensor values one value on each line
c using a lexicographical ordering
c
      open(unit=11,file='isotropic30.dat1')
      write(11,*) nx
      write(11,*) ny
      write(11,*) nz
      do 3 i=1,nnp
         write(11,*) (tns(i,l),l=1,9)
    3 continue
      close(unit=11)
c
c formats
c
   10 format(5(e14.8,2x))
      stop
      end
```

A.2.2 Example Fortran Code Output - Correct Format

There are two output files that are generated from this code in two different formats. The correct format is the following.

```
   4
   4
   1
0.29822734E+00  0.71511912E+00  0.33021107E-01  0.87439358E+00  0.53419441E+00
0.63158619E+00  0.89102978E+00  0.25754768E+00  0.93163508E+00  0.27724633E+00
0.71581721E+00  0.48340783E+00  0.53109354E+00  0.18343967E+00  0.27134272E+00
```

```
0.60305482E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00
0.29822734E+00  0.71511912E+00  0.33021107E-01  0.87439358E+00  0.53419441E+00
0.63158619E+00  0.89102978E+00  0.25754768E+00  0.93163508E+00  0.27724633E+00
0.71581721E+00  0.48340783E+00  0.53109354E+00  0.18343967E+00  0.27134272E+00
0.60305482E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00  0.00000000E+00
0.00000000E+00
0.29822734E+00  0.71511912E+00  0.33021107E-01  0.87439358E+00  0.53419441E+00
0.63158619E+00  0.89102978E+00  0.25754768E+00  0.93163508E+00  0.27724633E+00
0.71581721E+00  0.48340783E+00  0.53109354E+00  0.18343967E+00  0.27134272E+00
0.60305482E+00
```

A.2.3 Example Fortran Code Output - Some Other Format

The other output is the following which will not be read correctly by the JHomogenizer application.

```
4
4
1
 0.298227  0.  0.  0.   0.298227  0.  0.  0.   0.298227
 0.715119  0.  0.  0.   0.715119  0.  0.  0.   0.715119
  3.30211E-02  0.  0.  0.   3.30211E-02  0.  0.  0.   3.30211E-02
 0.874394  0.  0.  0.   0.874394  0.  0.  0.   0.874394
```

0.534194	0.	0.	0.	0.534194	0.	0.	0.	0.534194
0.631586	0.	0.	0.	0.631586	0.	0.	0.	0.631586
0.891030	0.	0.	0.	0.891030	0.	0.	0.	0.891030
0.257548	0.	0.	0.	0.257548	0.	0.	0.	0.257548
0.931635	0.	0.	0.	0.931635	0.	0.	0.	0.931635
0.277246	0.	0.	0.	0.277246	0.	0.	0.	0.277246
0.715817	0.	0.	0.	0.715817	0.	0.	0.	0.715817
0.483408	0.	0.	0.	0.483408	0.	0.	0.	0.483408
0.531094	0.	0.	0.	0.531094	0.	0.	0.	0.531094
0.183440	0.	0.	0.	0.183440	0.	0.	0.	0.183440
0.271343	0.	0.	0.	0.271343	0.	0.	0.	0.271343
0.603055	0.	0.	0.	0.603055	0.	0.	0.	0.603055

B Java Averaging Objects

All of the averaging objects are based on an interface in the Java programming language. The code in the object is given in this appendix. Note that any averaging object in this package must implement all of the methods in this interface in order to be valid. If the methods are not implemented, compilation of the code will result in compilation errors. The best idea is to start with an object that already works (e.g. ArithmeticAverageObject.java) and modify this object to do what you want.

Note that the following code is just the interface and as such cannot be instantiated. All this interface is intended to do is to define a set of methods that must be implemented in any averaging object that a person might want to create and to make sure that the objects are consistent. The consistency is important since we must make sure that if we want to apply different averaging methods to different regions in the original map that the application is implemented in the same way.

```
import java.util.Observable;
abstract class AverageObject extends Observable implements Runnable {
  abstract boolean testData();
  abstract boolean computeValue();
  abstract Double [] getFunctionValue(Double x, Double y, Double z);
  abstract Double [] getFunctionValue(double x, double y, double z);
  abstract double [] getfunctionValue(Double x, Double y, Double z);
  abstract double [] getfunctionValue(double x, double y, double z);
  abstract Double [] getDoubleValue();
  abstract double [] getdoubleValue();
  abstract Double [][] getData();
  abstract int getNumberOfXBlocks();
  abstract int getNumberOfYBlocks();
  abstract int getNumberOfZBlocks();
  abstract int getTotalBlocks();
  abstract int getTag();
  abstract boolean isValueOK();
  abstract boolean getStatus();
}
```

C ArithmeticAverage.java

The actual Java object that implements the arithmetic average is given below. The idea is to give some idea of what the code involves. There are a lot of lines of code that stretch over a number of pages. However, you should look for the method computeValue() that is the core of the entire object. In creating your own Java object you can copy the file ArithmeticAverage.java to a file name of your choice that shares the name of the Java object. Then modify the computeValue() method to reflect the averaging method you want included. It is a good thing to leave the rest of the methods alone. Most of these are run of the mill routines for computing grid dependent quantities like the cell center for the new coarse cell. The rest of the instructions for adding a method of averaging are included in the User's Guide.

The following is block of code is a portion of the code. In particular, the constructors for the method and the methods for computing the averaged values. There are a lot of other methods that allow access to the data in the object. You can find these in the code.

```
/*
 * SimulatorGUI class: ArithmeticAverage
 *
 */

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.Arrays;
import java.util.Observable;

/*
 * this is an object that will compute the arithmetic average and store the
 * appropriate value for retrieval
 */
public class ArithmeticAverage extends AverageObject {

  // uniform grid on the unit cubic constructor for the class
  public ArithmeticAverage(Double [][] d, int nx, int ny, int nz, int t)
       throws BadArraySizeException {
    this(d, nx, ny, nz, t,
```

```
      null, null, null, null, null, null,
      null, null, null);
}

// uniform grid on a specified region constructor for the class
public ArithmeticAverage(Double [][] d, int nx, int ny, int nz, int t,
                         Double xl, Double xr,
                         Double yl, Double yr,
                         Double zl, Double zr)
    throws BadArraySizeException {
  this(d, nx, ny, nz, t,
      xl, xr, yl, yr, zl, zr,
      null, null, null);
}

// specified grid constructor for the class
public ArithmeticAverage(Double [][] d, int nx, int ny, int nz, int t,
                         Double [] xg, Double [] yg, Double [] zg)
    throws BadArraySizeException {
  this(d, nx, ny, nz, t,
      null, null, null, null, null, null,
      xg, yg, zg);
}

// full constructor for the class
public ArithmeticAverage(Double [][] d, int nx, int ny, int nz, int t,
                         Double xl, Double xr,
                         Double yl, Double yr,
                         Double zl, Double zr,
                         Double [] xg, Double [] yg, Double [] zg)
    throws BadArraySizeException {
  data = d;
  nxhet = nx;
  nyhet = ny;
  nzhet = nz;
  tag = t;
  nxnyhet = nxhet * nyhet;
  nnhet = nxnyhet * nzhet;
  if(data.length != nnhet) throw
      new BadArraySizeException("Array the wrong size in ArithmeticAverage");

  // set up the grid
  if(xl != null) {
    xleft = xl;
  } else {
```

```
  xleft = new Double(0.0);
}
if(xr != null) {
  xright= xr;
} else {
  xright = new Double(1.0);
}
if(yl != null) {
  yleft = yl;
} else {
  yleft = new Double(0.0);
}
if(yr != null) {
  yright= yr;
} else {
  yright = new Double(1.0);
}
if(zl != null) {
  zleft = zl;
} else {
  zleft = new Double(0.0);
}
if(zr != null) {
  zright= zr;
} else {
  zright = new Double(1.0);
}

// make sure the end points are ok
if(xleft.doubleValue() > xright.doubleValue()) {
  Double temp = xleft;
  xleft = xright;
  xright = temp;
}
if(yleft.doubleValue() > yright.doubleValue()) {
  Double temp = yleft;
  yleft = yright;
  yright = temp;
}
if(zleft.doubleValue() > zright.doubleValue()) {
  Double temp = zleft;
  zleft = zright;
  zright = temp;
}
```

```
// set or compute the grid lines in the domain
if(xg != null) {
  xgrid = xg;
  nxhet = xgrid.length;
} else {
  xgrid = computeGrid(xleft, xright, nxhet);
}
Arrays.sort(xgrid);
if(yg != null) {
  ygrid = yg;
  nyhet = ygrid.length;
} else {
  ygrid = computeGrid(yleft, yright, nyhet);
}
Arrays.sort(ygrid);
if(zg != null) {
  zgrid = zg;
  nzhet = zgrid.length;
} else {
  zgrid = computeGrid(zleft, zright, nzhet);
}
Arrays.sort(zgrid);

// test the end points against the grid - the only time this matters is if
// the grid lines are specified
if(xleft.doubleValue() != xgrid[0].doubleValue()) xgrid[0] = xleft;
if(xright.doubleValue() != xgrid[nxhet].doubleValue())
        xgrid[nxhet] = xright;
if(yleft.doubleValue() != ygrid[0].doubleValue()) ygrid[0] = yleft;
if(yright.doubleValue() != ygrid[nyhet].doubleValue())
        ygrid[nyhet] = yright;
if(zleft.doubleValue() != zgrid[0].doubleValue()) zgrid[0] = zleft;
if(zright.doubleValue() != zgrid[nzhet].doubleValue())
        zgrid[nzhet] = zright;

// set up the computational parameters
done = false;
ok = false;

// set up the value to return
value = new Double[9];

// set up the thread and run it to compute the average value for the data.
computeThread = new Thread(this);
computeThread.start();
```

```
}

public void run() {
  if(testData()) {
    computeCellCenter();
    if(computeValue()) {
      ok = true;
    } else {
      ok = false;
    }
  } else {
    ok = false;
  }
  done = true;
  computeThread = null;
}

/*
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                                         *
 *    methods for simple averaging                                         *
 *                                                                         *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 */

public boolean testData() {
  for (int k=1;k<=nzhet;k++) {
    int k2 = ( k - 1 ) * nxnyhet;
    for (int j=1;j<=nyhet;j++) {
      int j2 = ( j - 1 ) * nxhet + k2;
      for (int i=1;i<=nxhet;i++) {
        int index = i + j2 - 1;
        for(int l=0; l<9; l++) {
          if(data[index][l] == null) {
            System.out.println("Arithmetic Average: data test failure:"
                              + "\n   x index =       " + i
                              + "\n   y index =       " + j
                              + "\n   z index =       " + k
                              + "\n   spatial index = " + index
                              + "\n   component index = " + l);
            return false;
          }
        }
      }
```

```
    }
  }
  return true;
}

public boolean computeValue() {
  int j2, k2, index;
  double blksize;
  double totalVolume = ( xgrid[nxhet].doubleValue() - xgrid[0].doubleValue() )
                     * ( ygrid[nyhet].doubleValue() - ygrid[0].doubleValue() )
                     * ( zgrid[nzhet].doubleValue() - zgrid[0].doubleValue() );
  double [] sum = new double[9];
  for(int l=0; l<9; l++) sum[l] = 0.0;
  for (int k=1;k<=nzhet;k++) {
    k2 = ( k - 1 ) * nxnyhet;
    double dz = zgrid[k].doubleValue() - zgrid[k-1].doubleValue();
    for (int j=1;j<=nyhet;j++) {
      j2 = ( j - 1 ) * nxhet + k2;
      double dy = ( ygrid[j].doubleValue() - ygrid[j-1].doubleValue() );
      for (int i=1;i<=nxhet;i++) {
        index = i + j2 - 1;
        double dx = ( xgrid[i].doubleValue() - xgrid[i-1].doubleValue() );
        blksize = dx * dy * dz;
        for(int l=0; l<9; l++) {
          sum[l] = sum[l] + data[index][l].doubleValue() * blksize;
        }
      }
    }
  }
  for(int l=0; l<9; l++) value[l] = new Double(sum[l] / totalVolume);
  notifyObservers();
  return true;
}

public Double [] computeGrid(Double xl, Double xr, int nx) {
  Double [] grid = null;
  grid = new Double [nx+1];
  double dx = (xr.doubleValue() - xl.doubleValue()) / ((double) nx);
  grid[0] = xl;
  for(int i=1; i<nx; i++) {
    grid[i] = new Double(grid[i-1].doubleValue() + dx);
  }
  grid[nx] = xr;
  return grid;
}
```

```
public void computeCellCenter() {
  xcenter = new Double(0.5 * ( xleft.doubleValue() + xright.doubleValue() ));
  ycenter = new Double(0.5 * ( yleft.doubleValue() + yright.doubleValue() ));
  zcenter = new Double(0.5 * ( zleft.doubleValue() + zright.doubleValue() ));
}


            *
            *
            *
 a bunch of utility routines
            *
            *
            *
```