CE 4333 Practical Computational Hydraulics using R

by
Theodore G. Cleveland
Department of Civil, Environmental, and Construction Engineering
Texas Tech University

Contents

1 Introduction and Getting Started

In the 1990s Civil Engineering programs reduced programming courses in a effort to recover hours for other topics – a logical decision at the time, but with some consequences. The philosophy was that engineers would not need to be able to write computer programs, but instead just use them. Microsoft Excel and Lotus 1-2-3 were the dominant spreadsheet software programs (Borland QuatroPro was a close third), and with macro instruction capability, much legitimate engineering computation could be performed within these tools. In fact I developed Excel spreadsheets that could solve multi-dimensional diffusion problems (3D groundwater flow) using fully implicit finite difference methods. These spreadsheets were slow relative to MODFLOW, but you could watch the solutions evolve – ultimately the process was deemed a waste, because of the ever present "... there is no longer a need for engineers to be able to write programs." Later on I developed spreadsheets to perform pressurized pipe network simulation, gradually varied flow simulation, and rudimentary water-hammer and St-Venant equation solutions. The spreadsheets were never really practical (yes they worked well, produced the same results as professional products, but were always intended a pedagogical tools), but they proved an important point – if you could teach a computer to follow an algorithm it made you a more self-help user of the professional tools.

In 2014 several of my students expressed desire to understand programming – they all know how to write code, but feel they don't know how to build algorithms (and implement them). This workbook is an attempt to remedy that student self-identified weakness. I conducted several one-to-one classes (as special topics); they learned a lot, I learned even more. This book is a tribute to their interests.

The workbook plan is to introduce a programming tool – I have selected **R** because it has a rich development environment already available, graphics is almost trivial, then apply that tool to selected hydraulics problems of practical value. In the end the reader ends up with a toolkit that can either stand-alone, or more likely supplement professional tools they will eventually use.

R is freeware, but it is built and maintained by a consortium of programmers and statisticians. They have evolved the environment to work on most of the main architectures (MacOS, Windows, Linux); there are even parallel processor and GPU builds available, and a company called RStudio provides the APIs to even run it server side. Much of the underlying code is C, C++, and well proven FORTRAN routines.

1.1 About R

R is a open source envrionment that runs on Windows, Linux/UNIX, and Mac OS X. The individual binaries are unique to each OS and architecture, but **R** "source" is interchangeable among machines. With very minor differences, an **R** script will run equally well on any machine.

R is a statistical analysis tool, it is also a programming tool and language, it is also a nearly "publication" quality graphics tool. Naturally all this capability comes at a cost (especially since the software is distributed for "free") — learning to do more than simple calculations takes some time (not much), but the skill is highly perishable. You will need to keep notes, or copies of your **R** scripts for future reference. Relearning after some time away from **R** is pretty simple, so the modeler only has to pay the steep learning cost once.

The remainder of this essay shows how to obtain and install **R** on a Windows machine. Macintosh and Linux installs are accomplished in a similar fashion. For the truly insane, the entire envrionment can be built from source on any machine with PERL, gcc, and gfortran compliers (default in Linux, easy to obtain for other architectures).

1.2 Getting Started

The first step required (for using **R** as a programming tool) is to install **R** on your computer. The source of the binary builds is the same regardless of the underlying operating system – the Comprehensive R Archive Network (CRAN for short). The remainder of this chapter shows how to get the tool running on the three main operating systems in current practice.

1.2.1 Windows Users

The purpose of this section is to demonstrate how to get **R** running on a Windows computer. This document assumes the following:

1. You have internet connection.
2. You have sufficient user privileges to install software on your machine. (If you need someone else to install, I did my install by running the installer as a local administrator obviously you need the password)
3. You have 60MB or so of vacant disk space on the system directory.

The step-by-step guide is presented as a series of screen captures. Obviously adjust inputs to fit your machine. The version in these screen captures is quite dated — use the most recent, stable version offered on CRAN (Comprehensive R Archive Network).

Yipee! It is running. You can install additional packages now or later. You should now have sufficient computation capability for much of the course. You may still need Python, Ruby, or Perl for data file formatting — these can wait, a lot of manipulation can be done in a text editor and using Excel.

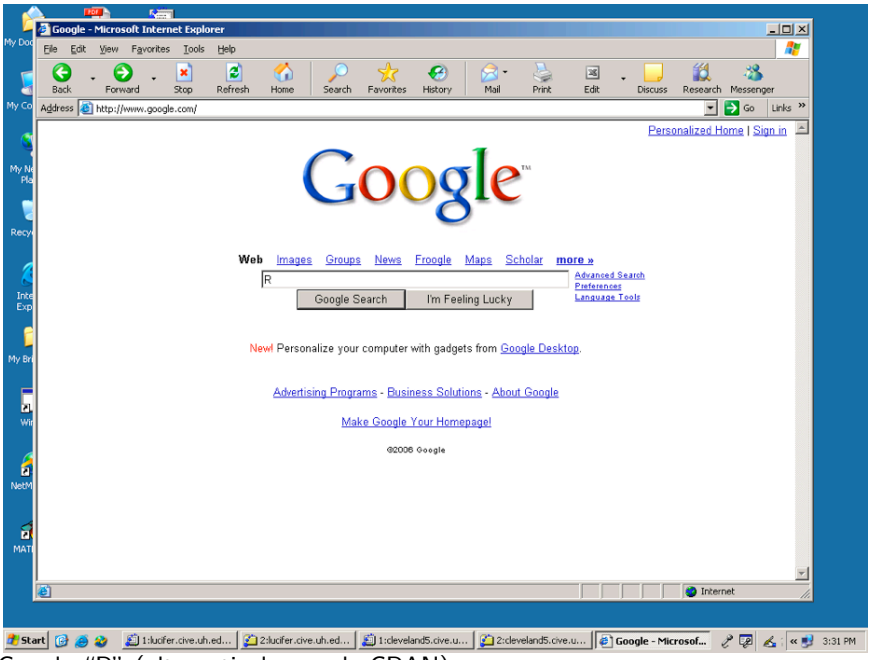

Figure 1. Google "R" (alternatively google CRAN).

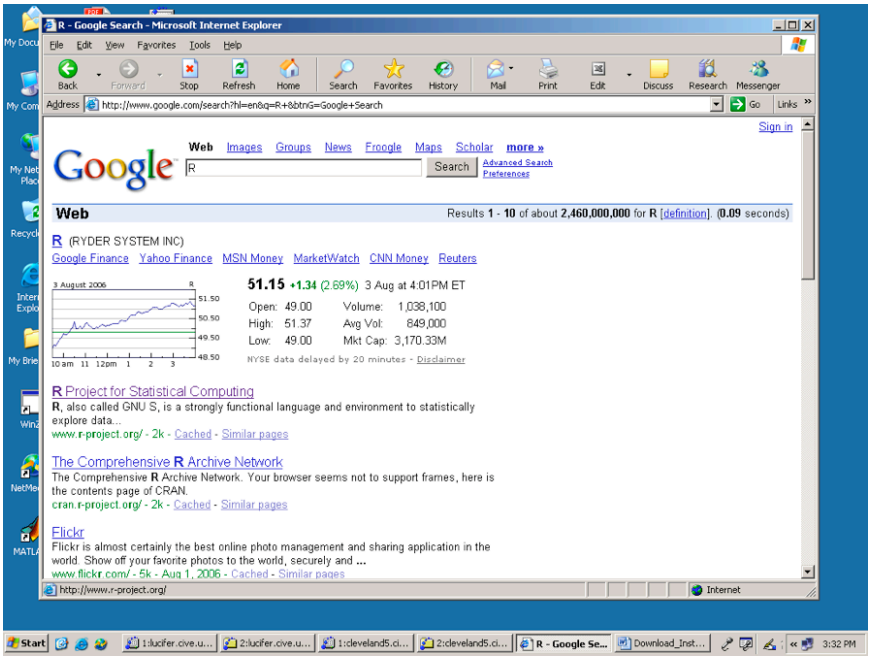

Figure 2. Select "R" GNU project.

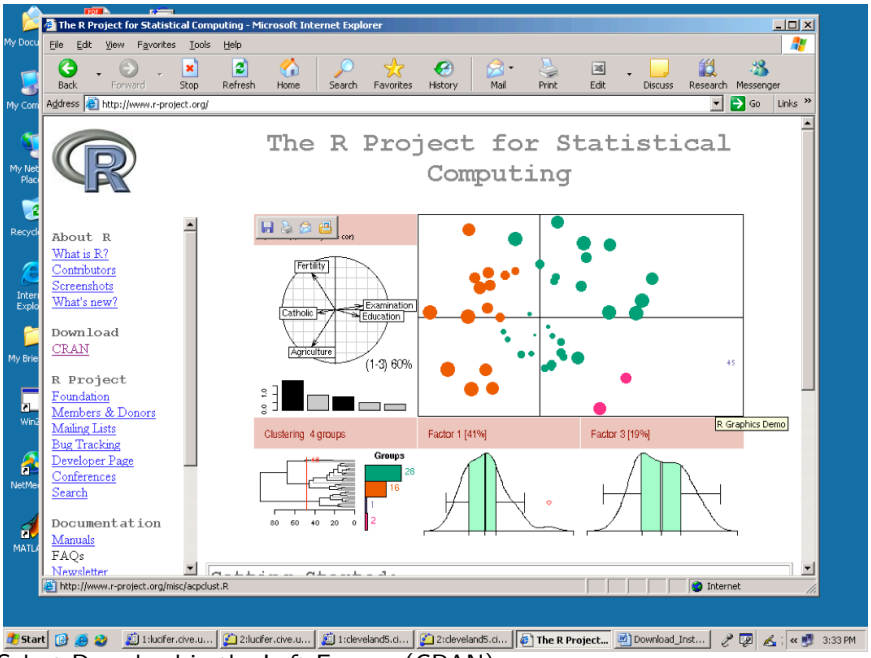

Figure 3. Select Download in the Left Frames (CRAN).

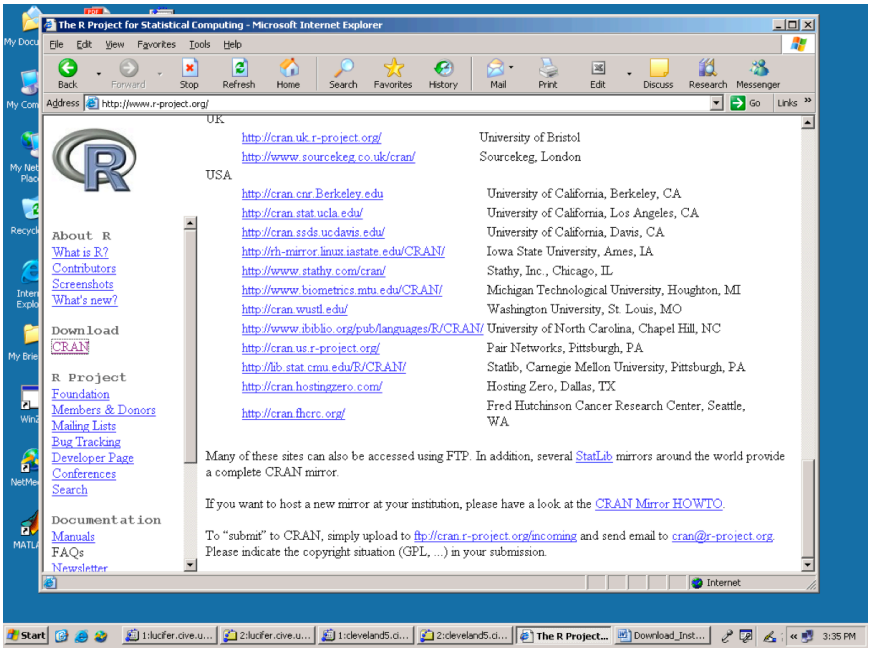

Figure 4. Choose a mirror. USA, Canada, and UK have fastest downloads.

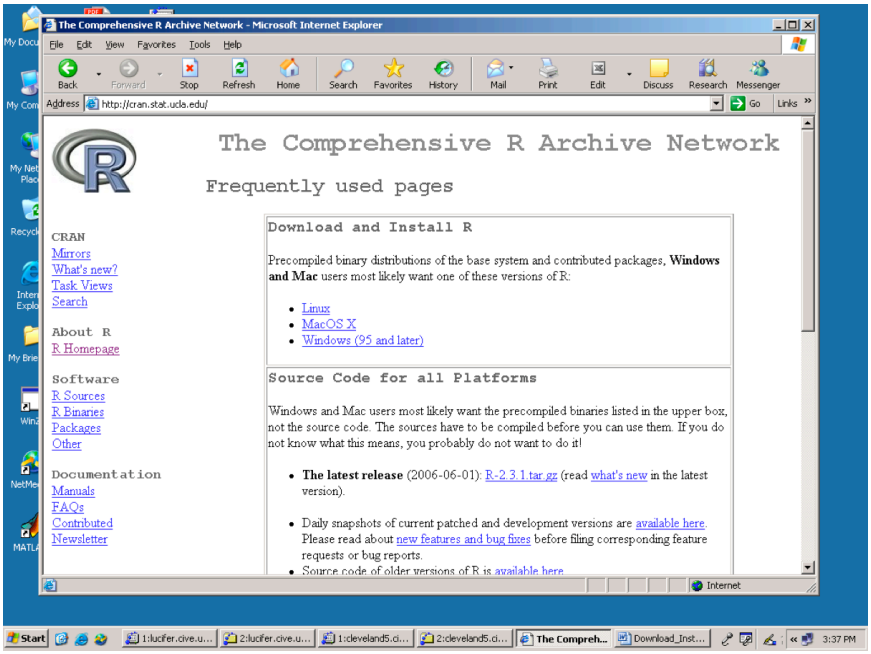

Figure 5. Select your platform : Windows, MacOSX, or Linux..

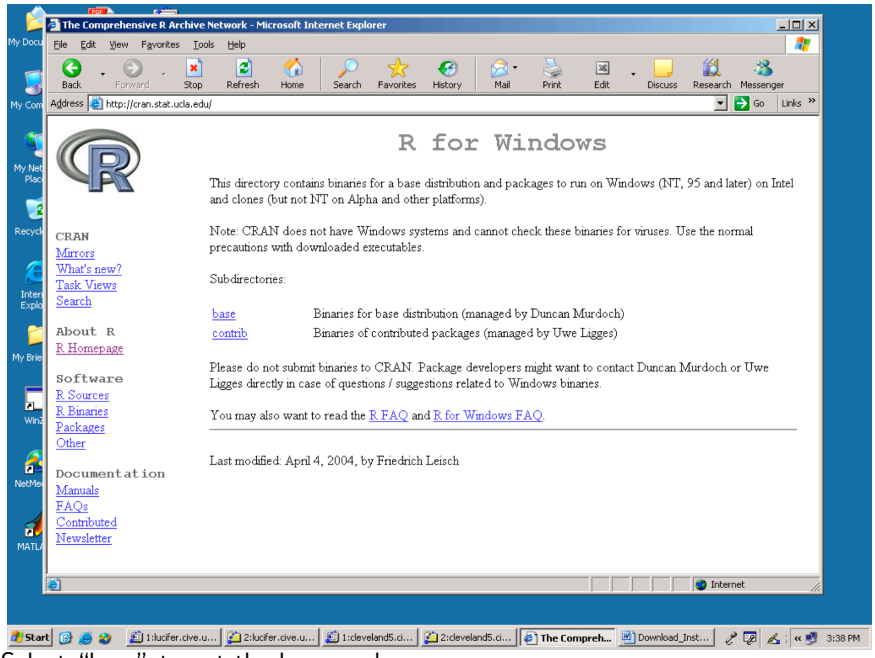

Figure 6. Select "base" to get the base packages..

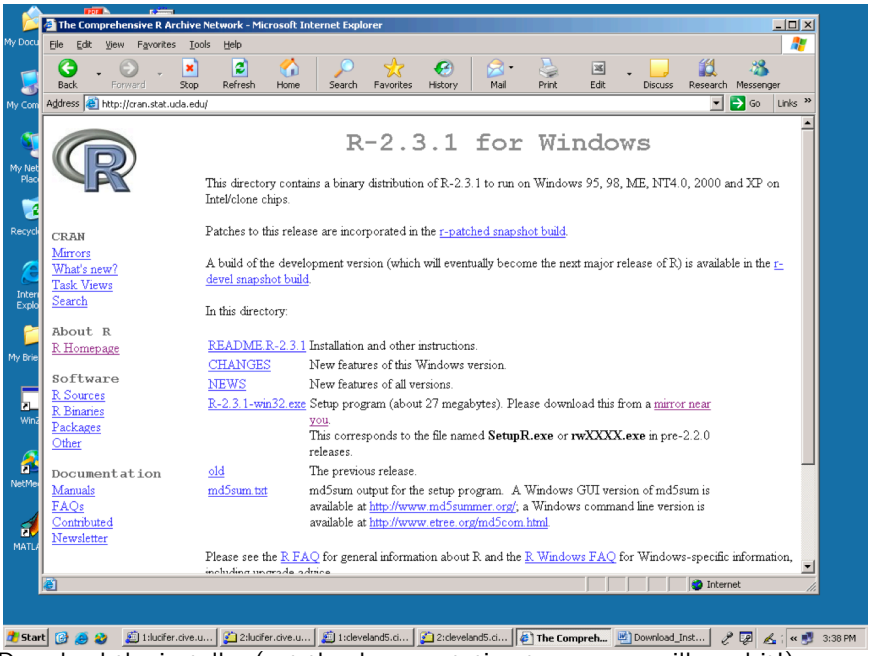

Figure 7. Download the installer (get the documentation too — you will need it!).

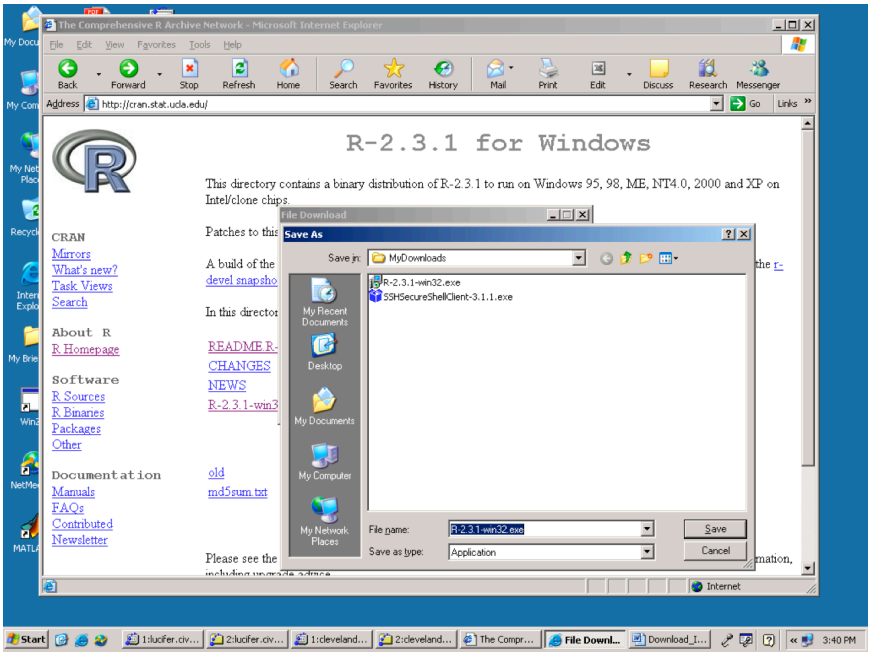

Figure 8. Downloading the system.

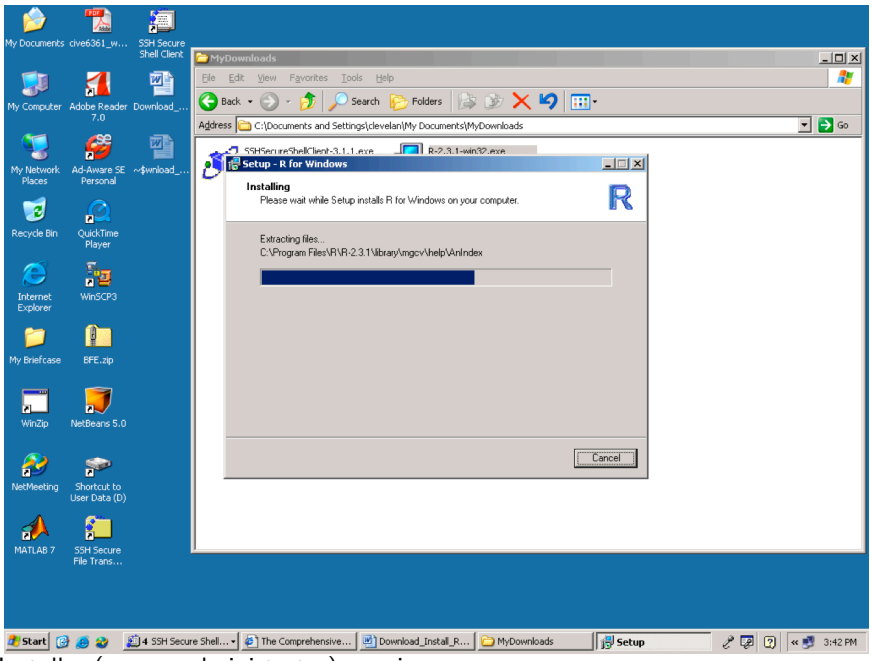

Figure 9. Installer (run as administrator) running..

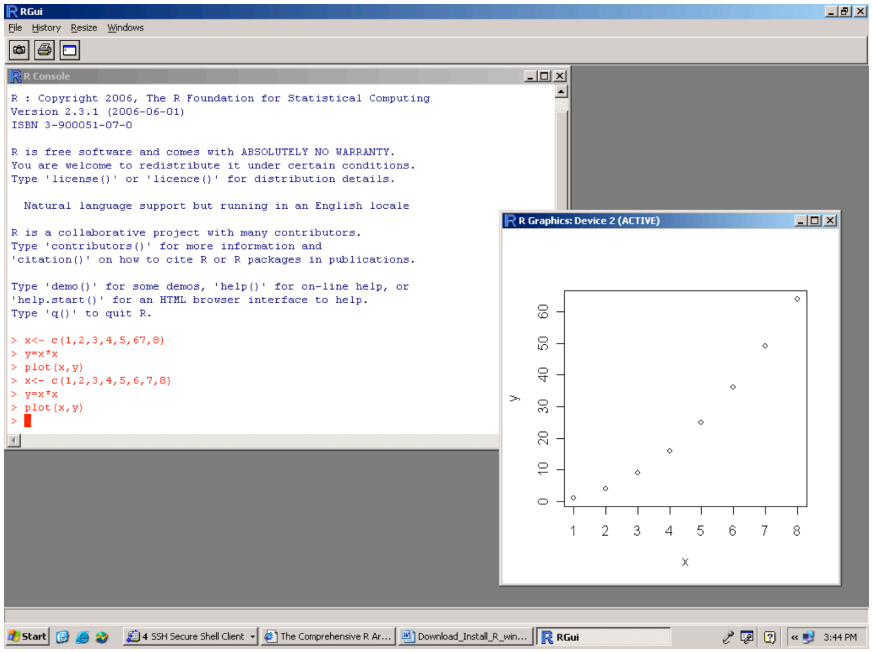

Figure 10. Verify the install by running the program and doing something simple..

1.2.2 Macintosh OSX Users

[Replicate Windows using MacOS screen captures]

1.2.3 Linux Users

The purpose of this section is to demonstrate how to get **R** and **R Studio** running on a Linux computer. This document assumes the following:

1. You have internet connection.
2. You have sufficient user privileges to install software on your machine. Generally, if it is your own machine then you have superuser (root) privileges. If it is some network machine maintained by someone else you probably don't have such priviliges. The examples here use the `sudo <command>` to do the installs – on my machine the password I enter is my user password (and not the root password). Alternativley you can switch user `su` to root, and run the installs as root – this approach is considerably more dangerous in terms of wrecking your operating system that using the `sudo` approach.
3. You have 60MB or so of vacant disk space on the system directory.

The step-by-step guide is presented as a series of screen captures. Obviously adjust inputs to fit your machine. Use the most recent, stable version offered on CRAN (Comprehensive R Archive Network).

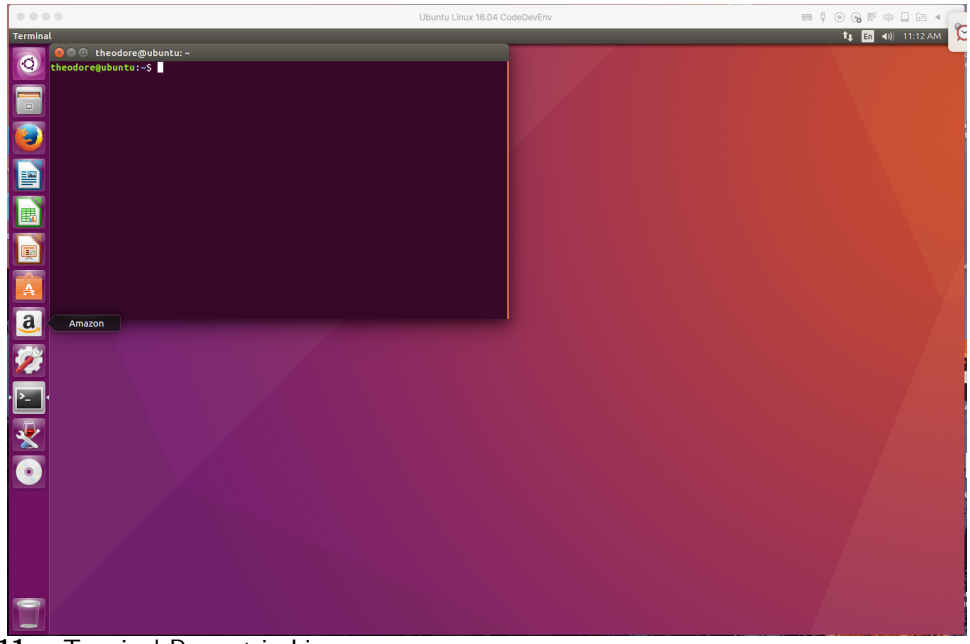

Figure 11. Terminal Prompt in Linux.

To get started we need to install **R**. The easiest method is to use the package manager – my Linux distribution is Ubuntu 16.XX. It is built from the debian distribution and uses the `apt` package manager. The package manager is pretty cool because when we request a package it finds the package and its dependencies, downloads everything needed and then we can install. In earlier times (the 1990's) using the Red Hat Package Manager (`rpm`) one would have to find the dependencies themselves (in all

fairness `rpm` would identify the dependencies and suggest where to find them!). So here we go, first open a terminal in Linux.

Next in the terminal window issue the command `sudo apt install r-cran-littler`.

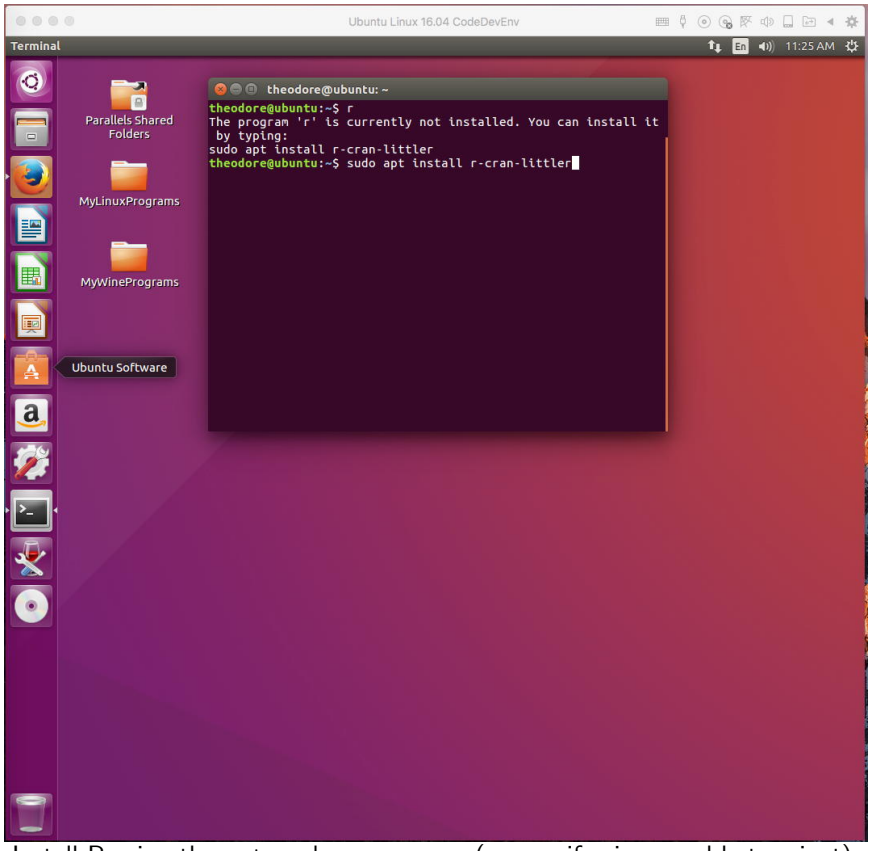

Figure 12. Install R using the `apt` package manager (or `rpm` if using a red hat variant).

When you press return, the computer should ask for a password – use your user password; if you have install privileges this will work. If not, you will have to switch user to root and either add your user account to the group `wheel`, return to your account – or just install as root.

Next the install will begin and may take a few minutes. Usually there is a lot of installer messages that run across the screen (kind of like in "The Matrix"). The `apt` utility is downloading files and binging them to resources on the system so that **R** can run. Eventually it should get to the end of the install and may look something like the next screen capture.

Our next task is to verify that the install was successful. Usually failure is obvious, but not always. I find the easiest way to verify that the operating system thinks the software is installed is issue the command to run the program with the version switch active. In this case, issue the command `r --version`. This command will try to run **R**, and will return the version number (or build number).

In the next screen capture when we run the command we see that the version number

is **R** version 3.2.2. The version numbering system is typical for all software – it identifies something like this is the second subversion, of the second revision, of the third stable release.

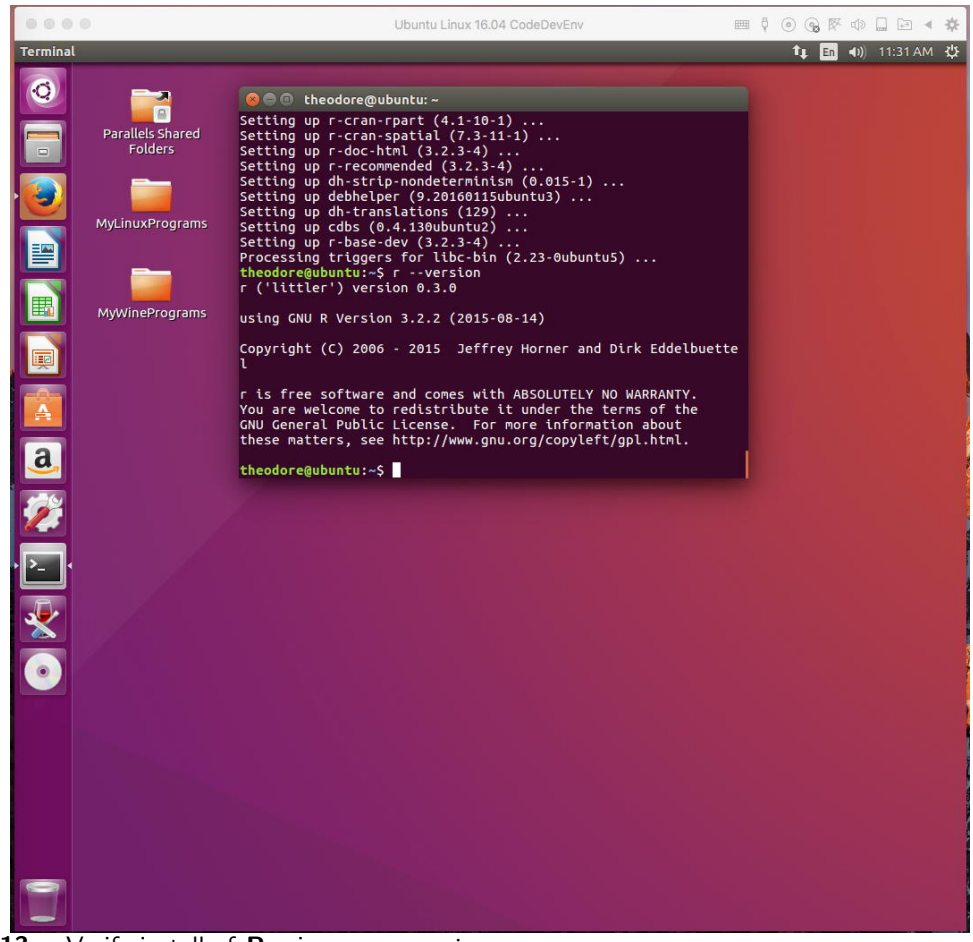

Figure 13. Verify install of **R** using `r --version`.

The next step I recommend is to go ahead and download a development environment. **R Studio** is an integrated development environment for **R**. We don't "need" it, but it enforces some discipline, gives us a place to store and modify **R**-scripts (little programs), and lets us see all of our work going on in one location.

To get **R Studio** we have to download it from the manufacturer, the copy we will use is free. Instead of `apt` this time I used Firefox to navigate to the **R Studio** website, then select download.

Next select the appropriate installer for your platform. Be sure you are selecting an installer and not the source codes for the program.[1]

We will download the 64-bit version (unless you have a 32-bit machine, then you need

[1] In theory we could build the program from source using the make utility and (hopefully) already installed compilers – this is for people with time, training, inclination and need. We are going to use the already built binaries!

32-bit software). We need to select our Linux platform – mine is Unbuntu/Debian. I also have a Red Hat/SuSe machine, so if I were using that machine, i would select its software.

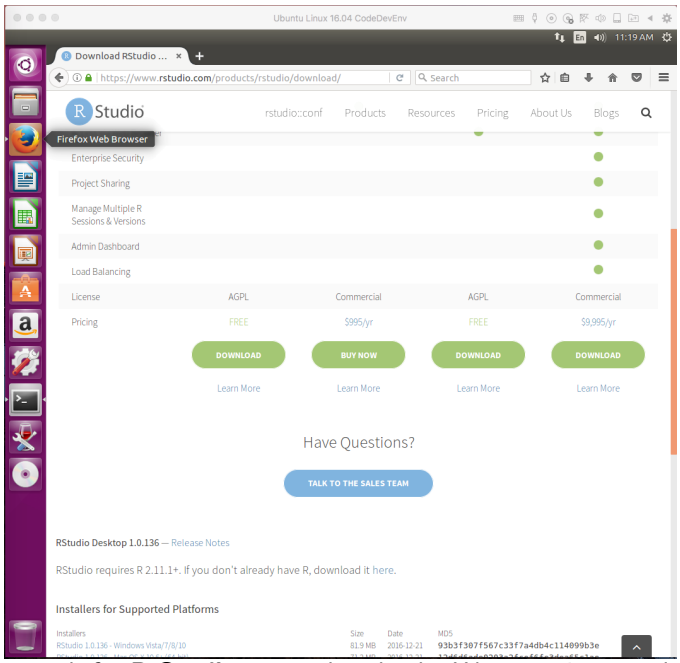

Figure 14. Browser search for **R Studio**; go to downloads. We are going to select the free (leftmost) column.

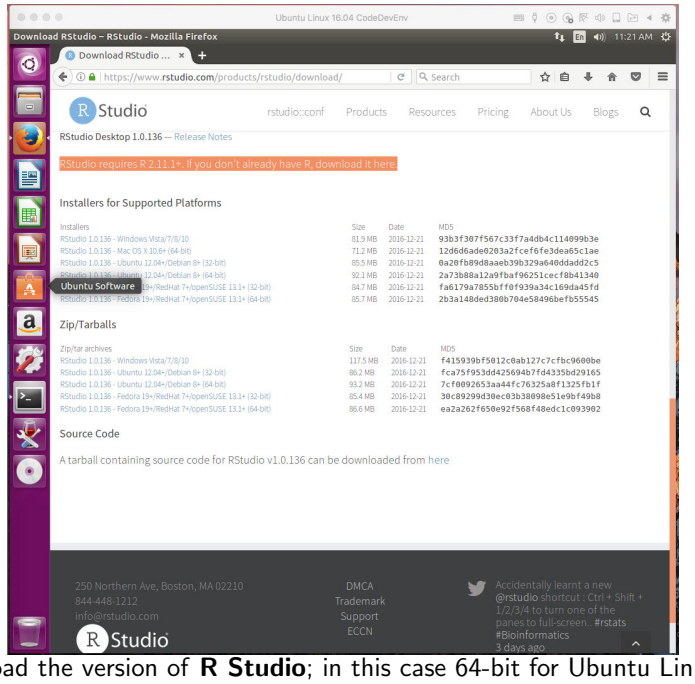

Figure 15. Download the version of **R Studio**; in this case 64-bit for Ubuntu Linux (what I am using). My computer asks if upon download if I want to run the installer, of course select YES.

Figure 15 is the selection page for selecting the installers. Once I select the package the computer starts the download, and asks me if I want to run the installer, as in Figure 16

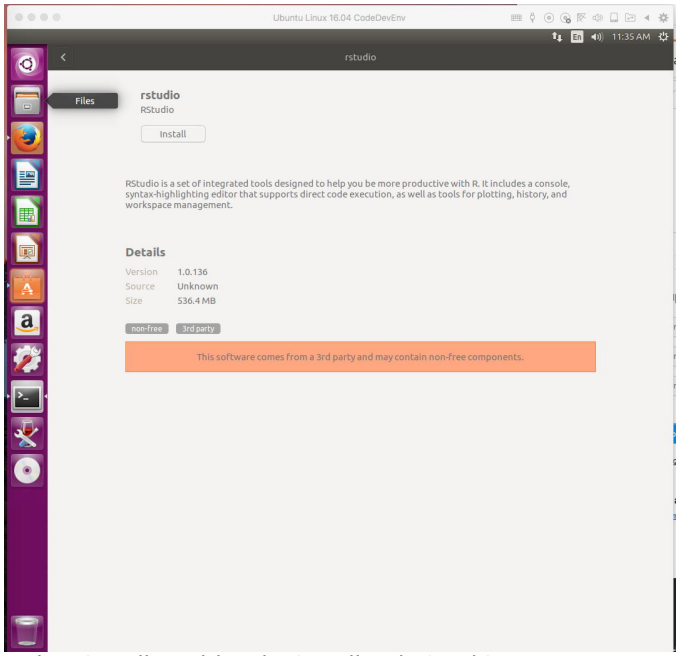

Figure 16. Here we select install, and let the installer do its thing.

Selecting yes, the installer will attempt to install **R Studio** onto my computer. Once installation is complete, the program is ready for a validation (of install) run.

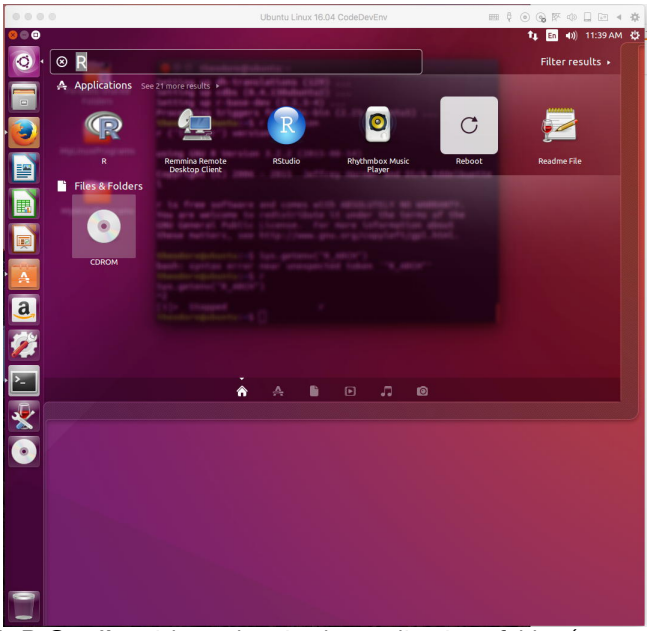

Figure 17. To launch **R Studio**, either select in the applications folder (or type in the terminal `rstudio`.

Figure 17 is a screen capture after installation using the Unbuntu program manager window. Both **R** and **R Studio** are available.

Either select **R Studio** to launch it, or type in a terminal window `rstudio` and the IDE should launch. The IDE itself is a bit complicated, but actually enables us to keep better track of our work and ultimately saves time. MatLab users should notice that the interface looks quite similar (at least it does to me) – its also the same concept.

Figure 18 is the result of launching the program. The left side of the IDE is an **R** console – exactly what would occur if we had just started **R**. The right side of the IDE is divided into an upper and lower panel. Each panel provides information about our programming environment at any instant – and the content is selectable from the icons at the top of each panel.

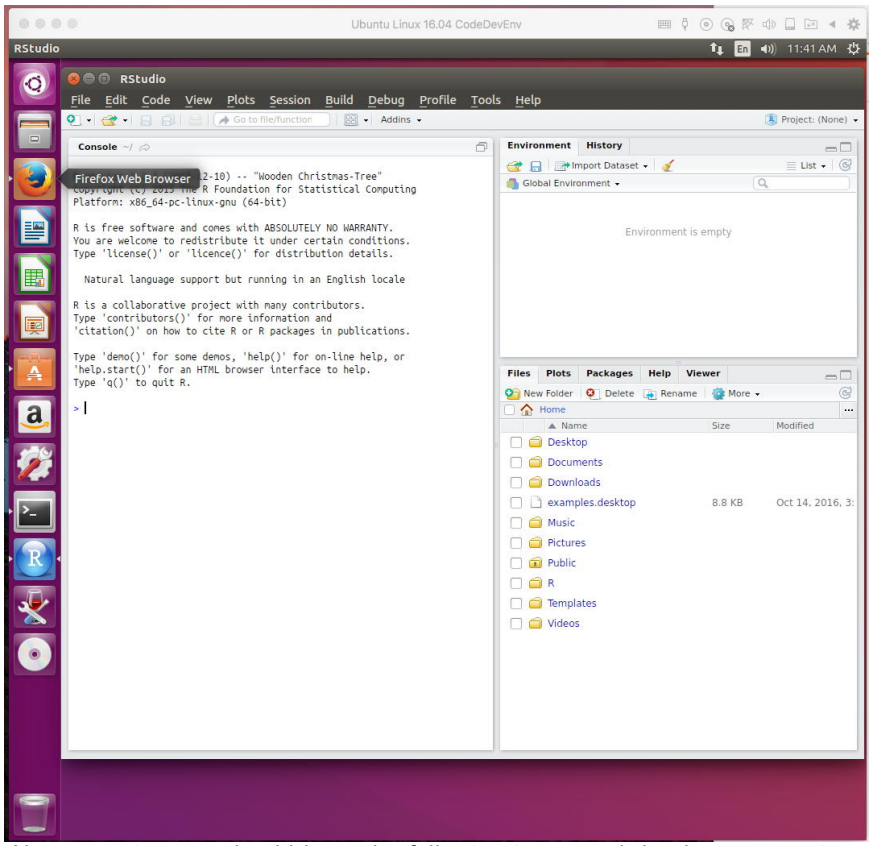

Figure 18. Upon opening you should have the following integrated development environment (IDE). The left panel is exactly what you would obtain if you just type `r` in the terminal window. .

The next few figures are a step-by-step example to introduce **R Studio** as well as test if it installed correctly. We could simply type in the **R** console within the IDE, but instead to get into the habit of saving our work, we will open a new scripting file.

Figure 19 shows the process of selecting FILE/OPEN to create a new file to store our scripts. We will type a few commands into the file and then run them.

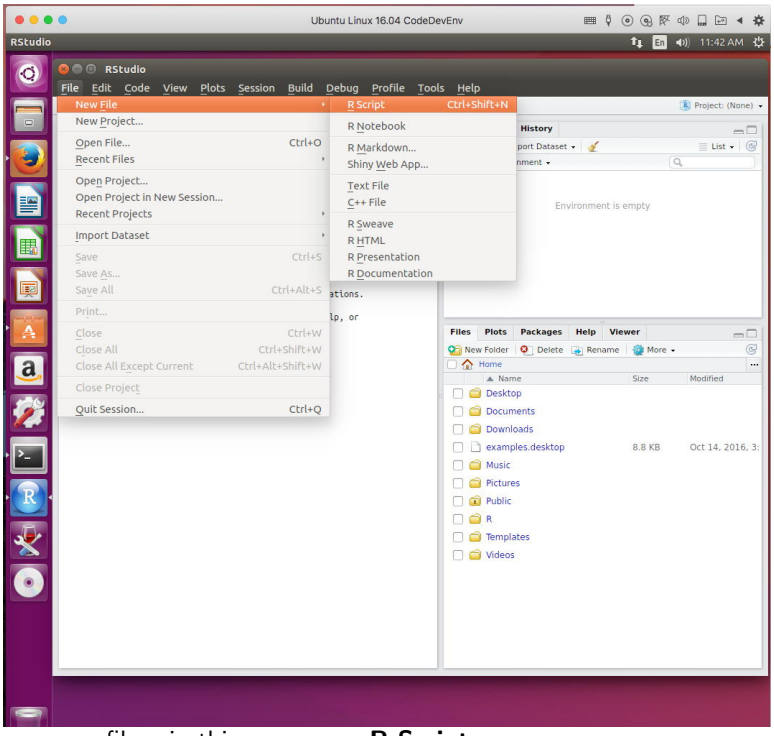

Figure 19. Open a new file – in this case as an **R-Script**.

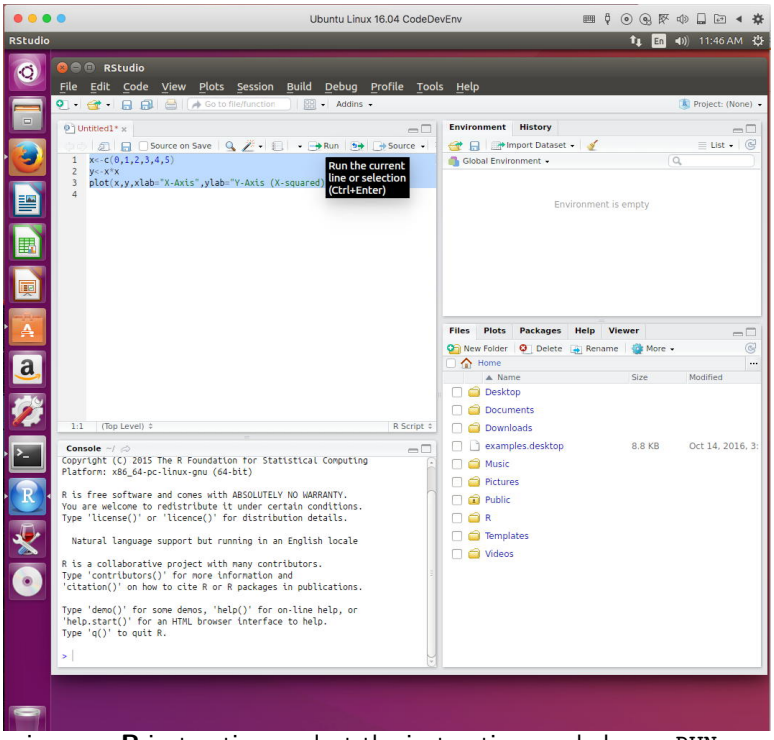

Figure 20. Type in some **R** instructions, select the instructions and choose `RUN`.

So once the file is open the left side of the IDE divides into two panels, an upper and lower panel. The lower panel is the console environment, and the upper panel is a script editor.

Figure 20 shows the upper panel with the **R** commands shown in Listing 1

Listing 1. R code demonstrating a few commands
This fragment of code generates two vectors X and Y and then plots them.

```
################ Some R Commands ########################
x <- c(0,1,2,3,4,5) # create a vector of 6 elements -- integers 0 to 5.
y <- x*x                  # square x, put result in y.
plot(x,y,xlab="X_Axis",ylab="Y-Axis (X-squared)",lwd=3, type="l") # make and label a plot
```

To actually run the code we can either highlight the portion of the script file we wish the program to execute (that's what done here), or we can save the file and run the entire file. Often we will do both – highlight portions to develop a model, then save and run the file as needed. The term "sourcing" a [file] in R is jargon for running the commands contained within a file named [file]. The ability to save and reuse commands is really useful and is what makes **R** (or any other stored program software) really useful.

Figure 21 is the result of highlighting these three lines of code and running them (notice the little run icon above the script editor).

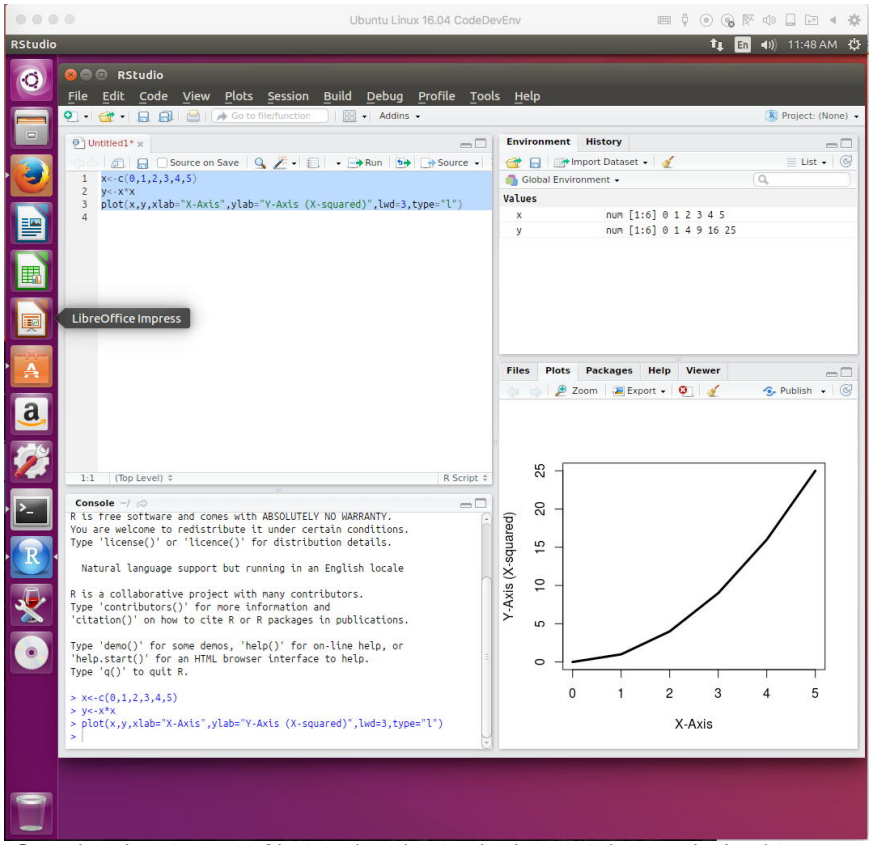

Figure 21. Completed script run. Notice the plot in the lower right panel. At this point you have a functioning programming tool..

1.3 Exercises

2 Algorithms and R for Computing

An algorithm is a recipe. A useful definition is

> An algorithm is a computational method or an ensemble of rules determining the order and form of numerical operations to be applied to a set of data $\mathbf{a}(a_1, a_2, \dots)$ in order to find a new set of values $\mathbf{x}(x_1, x_2, \dots)$ forming the solution of a problem.
>
> An algorithmic procedure can be represented as
>
> $$\mathbf{x} \;=\; f(\mathbf{a}) \tag{1}$$
>
> From a mathematical perspective the main concern is that the algorithm is well posed:
>
> 1. A solution exists for a given $\mathbf{a}$.
> 2. The computation must lead to a single solution for $\mathbf{x}$ given $\mathbf{a}$.
> 3. The results for $\mathbf{x}$ must be connected to the input $\mathbf{a}$ through the Lipschitz relation.
>
> $$|\delta\mathbf{a}| < \eta \;\text{ then }\; |\delta\mathbf{x}| < M|\delta\mathbf{a}| \tag{2}$$
>
> where M is a bounded natural number, $M = M(\mathbf{a}, \eta)$[2].
>
> Certain common problems are not well posed as stated but with reasonable assumptions can be forced into such a state.

Thus an algorithm is a recipe to take input data and produce output responses through some relationships. If a well posed problem then each result is related to the inputs, and the same inputs (in an algorithm) produce the same results. By the recipe analogy, if you follow the same recipe each time with the same raw materials then the cake should taste the same when it is baked.

An important concept is that an algorithm operates on data (procedure-oriented); an object-oriented view is that an algorithm performs a task (generate response) based on states established by the data. Both points of view are valid and equivalent.

Most computational hydraulics models are built (by a quirk of history) in a procedure-oriented perspective.

2.1 Tools

A practicing modeler needs a toolkit — these tools range from the actual computation engine (EPA-SWMM, HEC-RAS, FESWMS, HSPF, WSPRO, TR-20, etc.)

[2]Think of M as a mapping function.

to analysis tools for result interpretation (**R**, **Excel**) to actual programming tools (`FORTRAN`,`PERL`, etc.) to construct their own special purpose models or to test results from general purpose professional models.

In this book **R** will be used for programming, analysis, and presentation.

2.2 Programming

Why programming?

There are three fundamental reasons for requiring a programming experience:

1. Teaching someone else a subject or procedure forces the teacher to have a reasonable understanding of the subject or procedure. Teaching a computer (by virtue of programming) forces a very deep understanding of the underlying algorithm.
2. You will encounter situations that general purpose programs are not designed to address; if you have a moderate ability to build your own tools when you need to, then you can. In all likliehood, you will "trick" the professional program, but you cannot invent tricks unless you know a little bit about programming.
3. Programming a computer requires an algorithmic thought process — this process is valuable in many other areas of engineering, hence the act of programming is good discipline for other problems you will encounter.

2.3 Interpolating Tabular Data – A useful algorithm

Material properties in physical systems are usually tabulated values. A frequent task is to interpolate in a set of tabular values to approximate the value between rows (or columns) in the table. Linear interpolation is the common technique used; and the tables can are stores as either separate files, or, if the tables are small enough, they can be directly imbedded into the code.

2.4 Linear Interpolation

Figure 22 is a sketch of a set of ordered pairs (x, y).

These pairs (there are two in the sketch) represent values in a table, for instance x may represent water temperature, and y may represent vapor pressure at that particular temperature. Two adjacent values (in the table) are depicted in the sketch, and the pairs are ordered bases on the x-value.

Now suppose we want to estimate the value of y^* at some intermediate value x^* that lies between x_1 and x_2. As a computational task, the problem statement is to

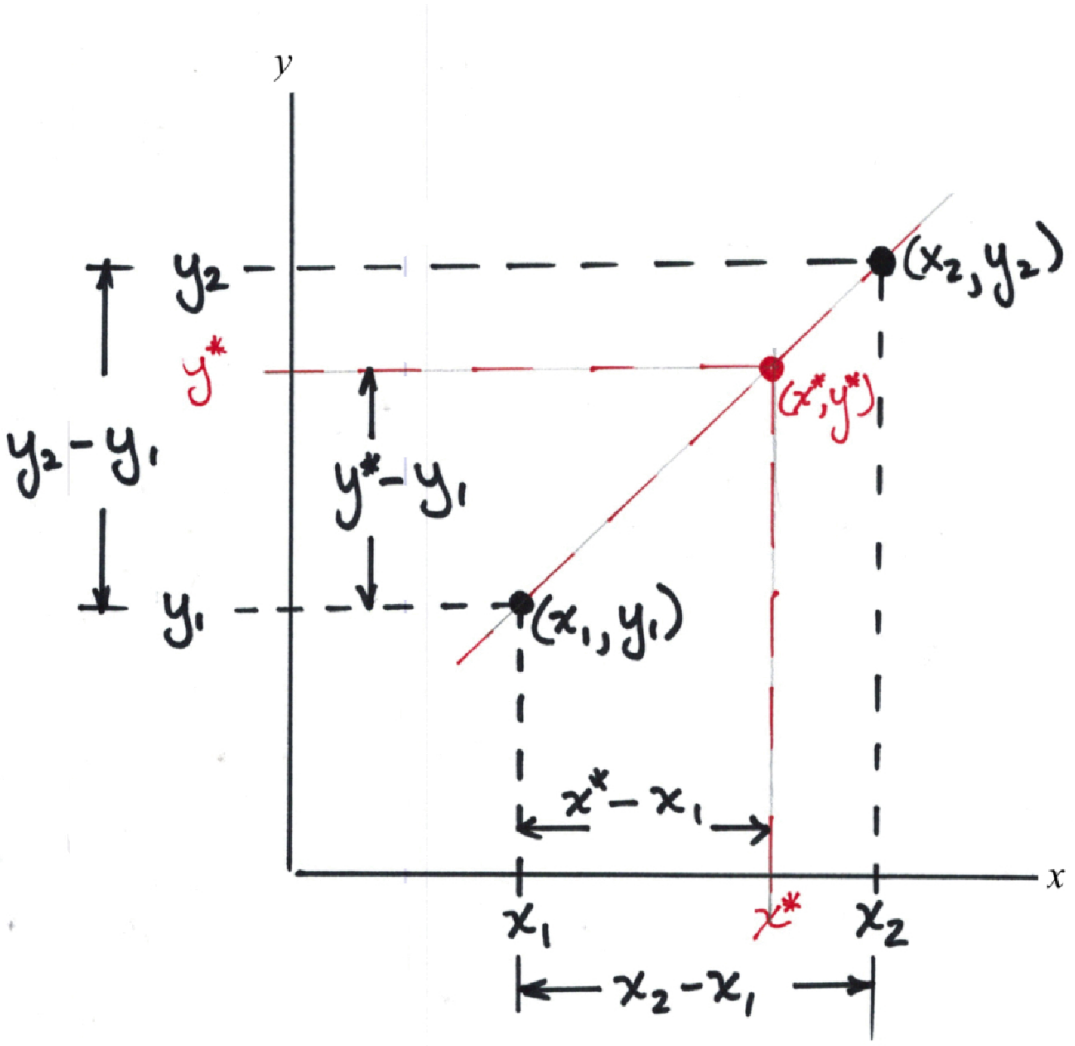

Figure 22. Sketch of two adjacent values from a table, plotted in Cartesian coordinate system..

"Estimate the value of y^* associated with the value x^* given the ordered pairs (x_1, y_1) and (x_2, y_2)."

Linear interpolation simply uses the concept of similar triangles to scale the x and y distances between the ordered pairs to the intermediate location. Equation 3 is the result of application of similar triangles to the situation described by Figure **??** and the problem statement.

$$\frac{x^* - x_1}{x_2 - x_1} = \frac{y^* - y_1}{y_2 - y_1} \tag{3}$$

Next, apply algebra to solve Equation 3 for y^*, to obtain Equation 4

$$y^* = y_1 + \frac{(y_2 - y_1)(x^* - x_1)}{(x_2 - x_1)} \tag{4}$$

Now we can use 4 to estimate values between any two data pairs.

2.4.1 Interpolation of Values in Two Pairs

Figure 23 is a table of water properties from (CITE), that represents typically how tabular data are presented. The temperature column is arranged in increasing order and the other properties associate with temperature across a row.

Temperature	Density	Specific Weight	Dynamic Viscosity	Kinematic Viscosity	Vapor Pressure
	kg/m^3	N/m^3	$N \cdot s/m^2$	m^2/s	N/m^2 abs
0°C	1000	9810	1.79×10^{-3}	1.79×10^{-6}	611
5°C	1000	9810	1.51×10^{-3}	1.51×10^{-6}	872
10°C	1000	9810	1.31×10^{-3}	1.31×10^{-6}	1,230
15°C	999	9800	1.14×10^{-3}	1.14×10^{-6}	1,700
20°C	998	9790	1.00×10^{-3}	1.00×10^{-6}	2,340
25°C	997	9781	8.91×10^{-4}	8.94×10^{-7}	3,170
30°C	996	9771	7.97×10^{-4}	8.00×10^{-7}	4,250
35°C	994	9751	7.20×10^{-4}	7.24×10^{-7}	5,630
40°C	992	9732	6.53×10^{-4}	6.58×10^{-7}	7,380
50°C	988	9693	5.47×10^{-4}	5.53×10^{-7}	12,300
60°C	983	9643	4.66×10^{-4}	4.74×10^{-7}	20,000
70°C	978	9594	4.04×10^{-4}	4.13×10^{-7}	31,200
80°C	972	9535	3.54×10^{-4}	3.64×10^{-7}	47,400
90°C	965	9467	3.15×10^{-4}	3.26×10^{-7}	70,100
100°C	958	9398	2.82×10^{-4}	2.94×10^{-7}	101,300

Figure 23. Table of water properties in SI units (from CITE).

Now suppose we wished to estimate the density of water at 44^o C. The two ordered pairs of temperature and density that surround 44^o C are $(40^o\ C, 992\ kg/m^3)$ and $(50^o\ C, 988\ kg/m^3)$. So, to estimate the unknown density we can apply Equation 4 and obtain the following result

$$y^* = 992 + \frac{(988 - 992)(44 - 40)}{(50 - 40)} = 990.4 \tag{5}$$

We might want to do this a lot, so we could write a simplistic script in R and remember to load it into our environment when we need it

Listing 2. R code demonstrating the interpolation equation.

```
# EXAMPLE # ** Interpolating Between Tabulated Pairs
interpolate2pairs<-function(xstar,x1,y1,x2,y2){
# apply interpolation equation
#   does not trap errors (divide by zero, etc)
  ystar <- y1 + (y2-y1)*(xstar-x1)/(x2-x1)
  return(ystar)
}
# In R Console
> interpolate2pairs(44,40,992,50,988)
[1] 990.4
>
```

For a single interrogation of a table we can stop here, but in many instances we have to interrogate a table a lot – we want some kind of program structure to handle the work so all we have to do is pass the temperature value and have the program return the density.

2.4.2 Interpolation of Values in Two Arrays

To accomplish repeated interpolation we will need to have: (1) an interpolating method (we have the beginning of one above in Listing 2), (2) the entire table so we don't have to enter the pairs, and (3) a way to automatically search the table so we don't have to look up values and supply them to the interpolator.

The table itself in this instance is relatively small, so we can simply assign values to some constant arrays in below in Listing 3.

Listing 3. R code assigning Liquid Properties.

```
# EXAMPLE # ** Assigning Constants
tempSI<-c(0.00,5.00,10.00,15.00,20.00,25.00,30.00,35.00,
   40.00,50.00,60.00,70.00,80.00,90.00,100.00)
densitySI<-c(1000.00,1000.00,1000.00,999.00,998.00,997.00,996.00,
   994.00,992.00,988.00,983.00,978.00,972.00,965.00,958.00)

# In R Console
> cbind(tempSI,densitySI)
      tempSI densitySI
 [1,]      0      1000
 [2,]      5      1000
 [3,]     10      1000
 [4,]     15       999
 [5,]     20       998
 [6,]     25       997
 [7,]     30       996
 [8,]     35       994
 [9,]     40       992
[10,]     50       988
[11,]     60       983
[12,]     70       978
[13,]     80       972
[14,]     90       965
[15,]    100       958
>
```

Returning to our example, the value 44 lies between `tempSI[9]` and `tempSI[10]`, so we desire an algorithm that starts at `tempSI[1]` and determines if the search value is between `tempSI[1]` and `tempSI[2]`, if not, then increment the row counter and determine if the search value is between `tempSI[2]` and `tempSI[3]`, and so on.

Once we locate in the searched array where the value lies then the interpolation uses the lower and upper elements of the range to interpolate. In the case of our example, once we determine the 44 lies between `tempSI[9]` and `tempSI[10]`, then the interpolation equation is

$$y^* = \texttt{densitySI[9]} + \frac{(\texttt{densitySI[10]} - \texttt{densitySI[9]})(44 - \texttt{tempSI[9]})}{(\texttt{tempSI[10]} - \texttt{tempSI[9]})} \quad (6)$$

Listing 4 is an **R** script that implements the search and interpolation just described. The script assumes that the searched array (x) is ordered and increasing – not a trivial assumption! The script has some limited error handling to test if the search value actually lies in the total range of the array before beginning the search. Once these tests are passed, then the code searches in the x array for the search value x^* and finds the two rows that contain the value. Once the rows are located, the interpolation equation is used.

Listing 4. R code to Search and Interpolate.

```
# EXAMPLE # ** Search and Interpolate
  getAvalue <- function(x,xvector,yvector){
    # returns a y value for x interpolated from (xvector,yvector)
    # xvector is assumed to be in a monotonic sequence
    # function performs limited error checks
    # NULL return is error indicator
    # T.G. Cleveland July 2007
    #
    xvlength <- length(xvector)
    yvlength <- length(yvector)
    # check that vector lengths same
    if(xvlength != yvlength){
      message("vectors xvector and yvector different lengths -- exiting function")
      return()
    }
    # check that x in range xvector
    if(x < min(xvector)){
      message(" x too small -- exiting function")
      return()
    }
    if(x > max(xvector)){
      message(" x too big -- exiting function")
      return()
    }
    #
    for (i in 1:(xvlength-1)){
      if( (x >= xvector[i]) & (x < xvector[i+1]) ){
        result = yvector[i]+(yvector[i+1]-yvector[i])*(x - xvector[i])/
        (xvector[i+1]-xvector[i])
        return(result)
      }
      # next row
    }
    # check if at endpoint
    if( (x >= xvector[xvlength-1]) & (x <= xvector[xvlength]) ){
      result = yvector[i]+(yvector[i+1]-yvector[i])*(x - xvector[i])/
      (xvector[i+1]-xvector[i])
      return(result)
    }
    # should never get to next line
    message("something is really wrong -- check the vectors!")
    return()
  }
 # In R Console:
> getAvalue(44,tempSI,densitySI)
[1] 990.4
>
```

Now we can load and run the `getAvalue` script and supply the two vectors plus the search value as shown in Listing 4

This look-up process is readily transferred to other cases, we do have to decide if the data will be coded as constants (as was done here) or read from a file – if the database is large the file read option is best. In terms of building a generic look-up tool several things actually happen in a particular order.

1. The function call loads in the table (of reading from a file, we would have to forward declare the vectors).
2. The function searches the first vector for the bounding location of the search variable.
3. Once the boundaries are located, the interpolation is performed – notice how the last boundary pair is handled.

Now we can combine the data assignment, the search and interpolate into a single function so when we want to evaluate in the future we only need the single function.

Listing 5 is an example of everything combined. Here I have embedded the `getAvalue` script into the function so the whole function itself is portbable (we don't have keep track of `getAvalue`). This embedding can be replaced with a load from a library (but then we must keep track of the path).

The library way is preferable; if `getAvalue` needs changing, we will have to change every instance of it in the code, if we miss one the code may still run and it could be years before we discover the error because a single instance of code fragment within a larger code was missed – its far better to only change a single instance of the function when maintenance is necessary.

Listing 5. R code to Return Water Density for Given Temperature.

```
# Script to return water density in SI units as a function of temperature
getDensitySI<-function(t){
# load the getAvalue() function ######################################################
  getAvalue <- function(x,xvector,yvector){
    # returns a y value for x interpolated from (xvector,yvector)
    # NULL return is error indicator
    #
    xvlength <- length(xvector)
    yvlength <- length(yvector)
    # check that vector lengths same
    if(xvlength != yvlength){
      message("vectors xvector and yvector different lengths -- exiting function")
      return()
    }
    # check that x in range xvector
    if(x < min(xvector)){
      message(" x too small -- exiting function")
      return()
    }
    if(x > max(xvector)){
      message(" x too big -- exiting function")
      return()
    }
    #
    for (i in 1:(xvlength-1)){
      if( (x >= xvector[i]) & (x < xvector[i+1]) ){
        result = yvector[i]+(yvector[i+1]-yvector[i])*(x - xvector[i])/
          (xvector[i+1]-xvector[i])
        return(result)
      }
      # next row
    }
    # check if at endpoint
    if( (x >= xvector[xvlength-1]) & (x <= xvector[xvlength]) ){
      result = yvector[i]+(yvector[i+1]-yvector[i])*(x - xvector[i])/
        (xvector[i+1]-xvector[i])
      return(result)
    }
    # should never get to next line
    message("something is really wrong -- check the vectors!")
    return()
  }
##############################################################################################
# load the data vectors, tempSI and densitySI
tempSI<-c(0.00,5.00,10.00,15.00,20.00,25.00,30.00,35.00,
  40.00,50.00,60.00,70.00,80.00,90.00,100.00)
```

```
densitySI<-c(1000.00,1000.00,1000.00,999.00,998.00,997.00,996.00,
  994.00,992.00,988.00,983.00,978.00,972.00,965.00,958.00)
# now call getAValue
result<-getAvalue(t,tempSI,densitySI)
return(result)
}
```

The "library" approach is demonstrated in Listing 6; in this listing the path in the `source()` command is unique to my machine – your path is likely to be different. I find it is useful to contain all the various codes into a single directory and source that directory once to find the path, then change all the source calls to that path. In fact that path can be a string variable and the referencing can be automatic (as long as the files exist!).

Once the look-up function is built then we can interrogate the table many times; and even build a plot of the table – these features are demonstrated in Listing 6.

Listing 6. R code demonstrating use of getDensitySI().

```
## In R Console
> # Example demonstrating use of functions
> # load in the functions (must exist -- use path on your machine)
> source('~/Dropbox/1-CE-TTU-Classes/UnderDevelopment/
   CE4333-PCH-R/6-RScripts/getAvalue.R')
> source('~/Dropbox/1-CE-TTU-Classes/UnderDevelopment/
   CE4333-PCH-R/6-RScripts/getDensitySI.R')
> # Now use them
> getDensitySI(44)
[1] 990.4
> getDensitySI(54)
[1] 986
> getDensitySI(88)
[1] 966.4
> t<-seq(0,100,2) # make a temperature vector 0 to 100 in 2 degree increments
> d<-numeric(0) # forward declare d to store results
> howMany<-length(t)
> for(i in 1:howMany){
+     d[i]<-getDensitySI(t[i])
+ }
> plot(t,d,type="l",xlab="Degrees Celsius",ylab="Density (kg/m^3)")
>
```

The resulting plot is shown on Figure 24 below.

2.5 Sorting

Another frequent task in engineering hydraulics is the seemingly mundane task of sorting or ordering things. Here we explore a couple of simple sorting algorithms, just to show some of the thoughts that go into such a task, then will ultimately resort to the internal sorting routines built into R.

2.5.1 Bubble Sort

The bubble sort is a place to start despite it's relative slowness. It is a pretty reviled algorithm (read the Wikipedia entry), but it is the algorithm that a naive programmer might cobble together in a hurry, and despite its shortcomings (its really slow and inefficient), it is robust.

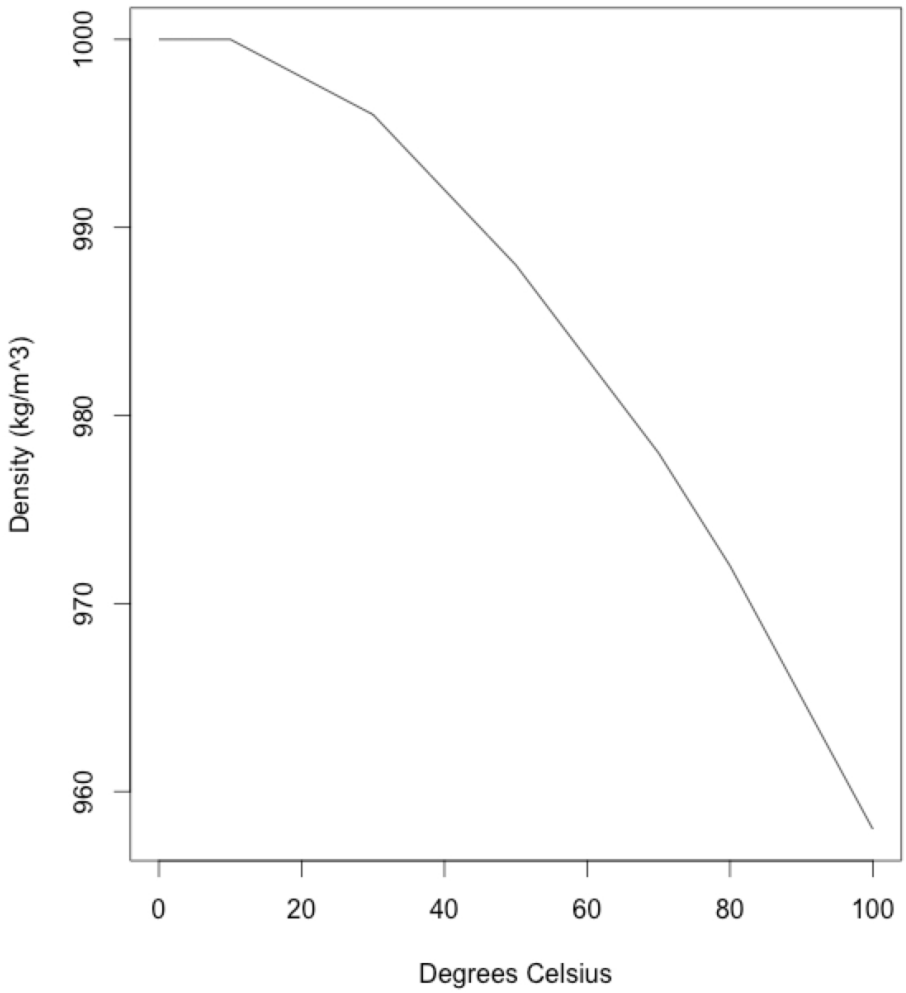

Figure 24. Plot of density versus temperature generated using the `getDensity()` function..

Here is a description of the sorting task as described by Christian and Griffiths (2016) (pg. 65):

> "Imagine you want to alphabetize your unsorted collection of books. A natural approach would be just to scan across the shelf looking for out-of-order pairs – Wallace followed by Pynchon, for instance – and flipping them around. Put Pynchon ahead of Wallace, then continue your scan, looping around to the beginning of the shelf each time you reach the end. When you make a complete pass without finding any more out-of-order pairs on the entire shelf, then you know the job is done.
>
> This process is a Bubble Sort, and it lands us in quadratic time. There are n books out of order, and each scan through the shelf can move each one at most one position. (We spot a tiny problem, make a tiny fix.) So in the worst case, where the shelf is perfectly backward, at least one book will need to be moved n positions. Thus a maximum of n passes through n books, which gives us $O(n2)$ in the worst case.[3] ..... For instance, it means that sorting five shelves of books will take not five times as long as sorting a single shelf, but twenty-five times as long."

Converting the word description into **R** is fairly simple. We will have a vector of n numbers (we use a vector because its easy to step through the different positions), and we will scan through the vector once (and essentially find the smallest thing), and put it into the first position. Then we scan again from the second position and find the smallest thing remaining, and put it into the second position, and so on until the last scan which should have the remaining largest thing. If we desire a decreasing order, simply change the sense of the comparison.

Listing 7 is an **R** script that implements the algorithm – in the script the actual sort is treated as a function (we may actually want to use it again someday) which is loaded into the programming environment first, then an array is defined, and sorted. The program (outside of the sorting algorithm) is really quite simple.

- Load the sorting function.
- Load contents into an array to be sorted.
- Echo (print) the array (so we can verify the data are loaded as anticipated).
- Sort the array, put the results back into the array (an in-place sort).
- Report the results.

Listing 7. R code demonstrating the naive bubble sort.

```
################################################################
rm(list=ls())  # clear the object list (i.e. deallocate and clear memory)
### Bubble Sort Function -- Needs to be defined before sending array to sort ###
# Bubble Sort Function
```

[3] Actually, the average running time for Bubble Sort isn't any better, as books will, on average, be $n/2$ positions away from where they?re supposed to end up. One would round the $n/2$ passes of n books up to $O(n2)$.

```
# MyLocation: ~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/1-Lectures/Lecture03/ScriptsInLecture
# Bubble Sort with array indexing starting at [1]
# Compare to Python or C where arrays start at [0])
# by: Theodore G. Cleveland 2017-0317
################################################################
bubble <- function(array)
{
  # Prepare the sort, need to know how many things and need a temporary store
  swap <- numeric(0) # temporary store (aka swap location)
  howMany <- length(array) # how many things to be sorted
  # The actual sorting process
  for (irow in 1:(howMany-1))
  {
    for (jrow in 1:(howMany-irow))
    {
      if( array[jrow] > array[jrow+1])
      {
        swap <- array[jrow];
        array[jrow] <- array[jrow+1];
        array[jrow+1] <- swap;
      }
    }
  }
  # return result (sort in-place)
  return(array)
}
################################################################

################################################################
xarray <- c(1003,3.2,55.5,-0.0001,-6,666.6,102)  # the array to sort
print(xarray)
xarray <- bubble(xarray)
print(xarray)
################################################################
```

Figure 25 is a screen capture of the script running. In the figure we see that the program (near the bottom of the file) assigns the values to the vector named array and the initial order of the array is $[1003, 3.2, 55.5, -0.0001, -6, 666.6, 102]$. The smallest value in the example is -6 and it appears in the 5-th position, not the 1-st as it should.

The first pass through the array will move the largest value, 1003, in sequence to the right until it occupies the last position. Repeated passes through the array move the remaining largest values to the right until the array is ordered. One can consider the values of the array at each scan of the array as a series of transformations (`irow`-th scan) – in practical cases we don't necessarily care about the intermediate values, but here because the size is manageable and we are trying to get our feet wet with algorithms, we can look at the values.

The sequence of results (transformations) after each pass through the array is shown in the following list:

1. Initial value: $[1003, 3.2, 55.5, -0.0001, -6, 666.6, 102]$.
2. First pass: $[3.2, 55.5, -0.0001, -6, 666.6, 102, 1003]$.
3. Second pass: $[3.2, -0.0001, -6, 55.5, 102, 666.6, 1003]$.
4. Third pass: $[-0.0001, -6, 3.2, 55.5, 102, 666.6, 1003]$.
5. Fourth pass: $[-6, -0.0001, 3.2, 55.5, 102, 666.6, 1003]$.
6. Fifth pass: $[-6, -0.0001, 3.2, 55.5, 102, 666.6, 1003]$. Sorted, fast scan.
7. Sixth pass: $[-6, -0.0001, 3.2, 55.5, 102, 666.6, 1003]$. Sorted, fast scan.

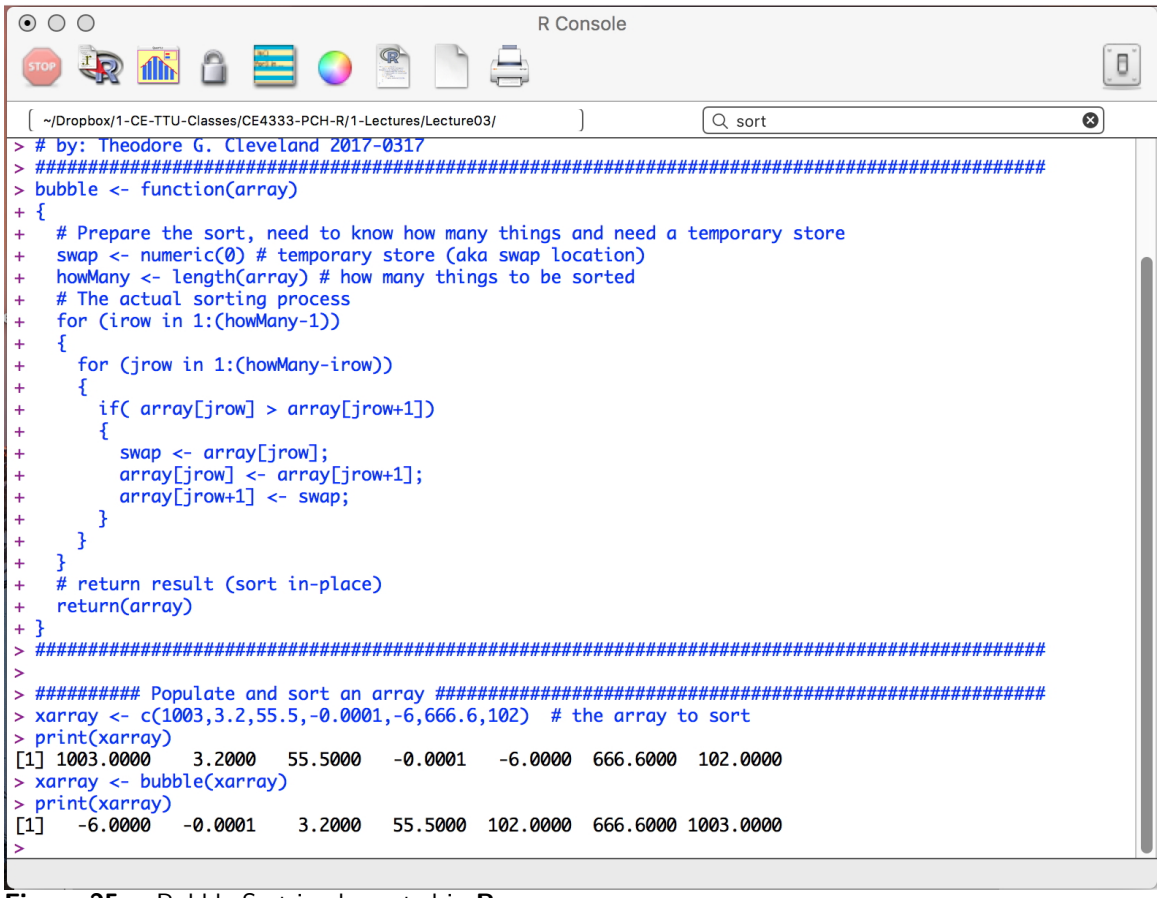

Figure 25. Bubble Sort implemented in **R**.

We could probably add additional code to break from the scans when we have a single pass with no exchanges – while meaningless in this example, for larger collections of things, being able to break out when the sorting is complete is a nice feature.

2.5.2 Insertion Sort

The next type of sorting would be to select one item and locate it either left or right of an adjacent item based on its size – like sorting a deck of cards, or perhaps a better description – again using the bookshelf analog from Christian and Griffiths (2016) (pg. 65):

> "..... You might take a different tack – pulling all the books off the shelf and putting them back in place one by one. You'd put the first book in the middle of the shelf, then take the second and compare it to the first, inserting it either to the right or to the left. Picking up the third book, you'd run through the books on the shelf from left to right until you found the right spot to tuck it in. Repeating this process, gradually all of the books would end up sorted on the shelf and you'd be done. Computer scientists call this, appropriately enough, Insertion Sort. The good news is that it's arguably even more intuitive than Bubble Sort and doesn't have quite the bad reputation. The bad news is that it's not actually that much faster. You still have to do one insertion for each book. And each insertion still involves moving past about half the books on the shelf, on average, to find the correct place.
>
> Although in practice Insertion Sort does run a bit faster than Bubble Sort, again we land squarely, if you will, in quadratic time. Sorting anything more than a single bookshelf is still an unwieldy prospect."

Listing 8 is an **R** implementation of a straight insertion sort. The script is quite compact, and I used indentation and extra line spacing to keep track of the scoping delimiters. The sort works as follows, take the an element of the array (start with 2 and work to the right) and put it into a temporary location (called `swap` in my script). Then compare locations to the left of `swap`. If smaller, then break from the loop, exchange values, otherwise the values are currently ordered. Repeat (starting at the next element) , when all elements have been traversed the resulting vector is sorted. Here are the transformations for each pass through the outer loop:

1. Pass 0: $[1003, 3.2, 55.5, -0.0001, -6, 666.6, 102]$, Initial array.
2. Pass 1: $[3.2, 1003, 55.5, -0.0001, -6., 666.6, 102]$.
3. Pass 2: $[3.2, 55.5, 1003, -0.0001, -6., 666.6, 102]$.
4. Pass 3: $[-0.0001, 3.2, 55.5, 1003, -6., 666.6, 102]$.
5. Pass 4: $[-6, -0.0001, 3.2, 55.5, 1003., 666.6, 102]$.
6. Pass 5: $[-6, -0.0001, 3.2, 55.5, 666.6, 1003, 102]$.
7. Pass 6: $[-6, -0.0001, 3.2, 55.5, 102, 666.6, 1003]$, Sorted array.

Figure 26 is a screen capture of the insertion sort in operation. Insertion sorting is

reasonably fast for small lists (about 50 or so elements) and forms the basis of the internal sorts in other routines that divide up the overall list into smaller lists, sort the smaller lists, then uses a merge to collate back to the overall list (now sorted).

Listing 8. R code demonstrating the insertion sort.

```
### Straight Insertion Sort Function by: Theodore G. Cleveland 2017-0317
rm(list=ls())  # clear the object list (i.e. deallocate and clear memory)
################################################################
insertSort <- function(array){
# Prepare the sort, need to know how many things and need a temporary store
  swap <- numeric(0) # temporary store (aka swap location)
  howMany <- length(array) # how many things to be sorted
  for (j in 2:howMany)  # select each position in turn
    {
    test <- 0           # set a test value, used to insert later
    swap <- array[j]    # current position to swap
    for (i in seq(j-1,1,by=-1)) #find place to insert by ...
      {
      if (array[i] <= swap)  # test if current position is bigger
        {
        test <- 1            # if true set test to 1, break inner loop
        break
        }
      array[i+1] <- array[i]  # otherwise exchange postions
      }
    if(test == 1)             # if broke from loop, insert swap
      array[i+1] <- swap
    else
      i = 0
      array[i+1] <- swap      # otherwise swap goes to first position
    }
  return(array) }    # return result (sort in-place)
################################################################
xarray <- c(1003,3.2,55.5,-0.0001,-6,666.6,102)  # the array to sort
print(xarray)
xarray <- insertSort(xarray)
print(xarray)
################################################################
```

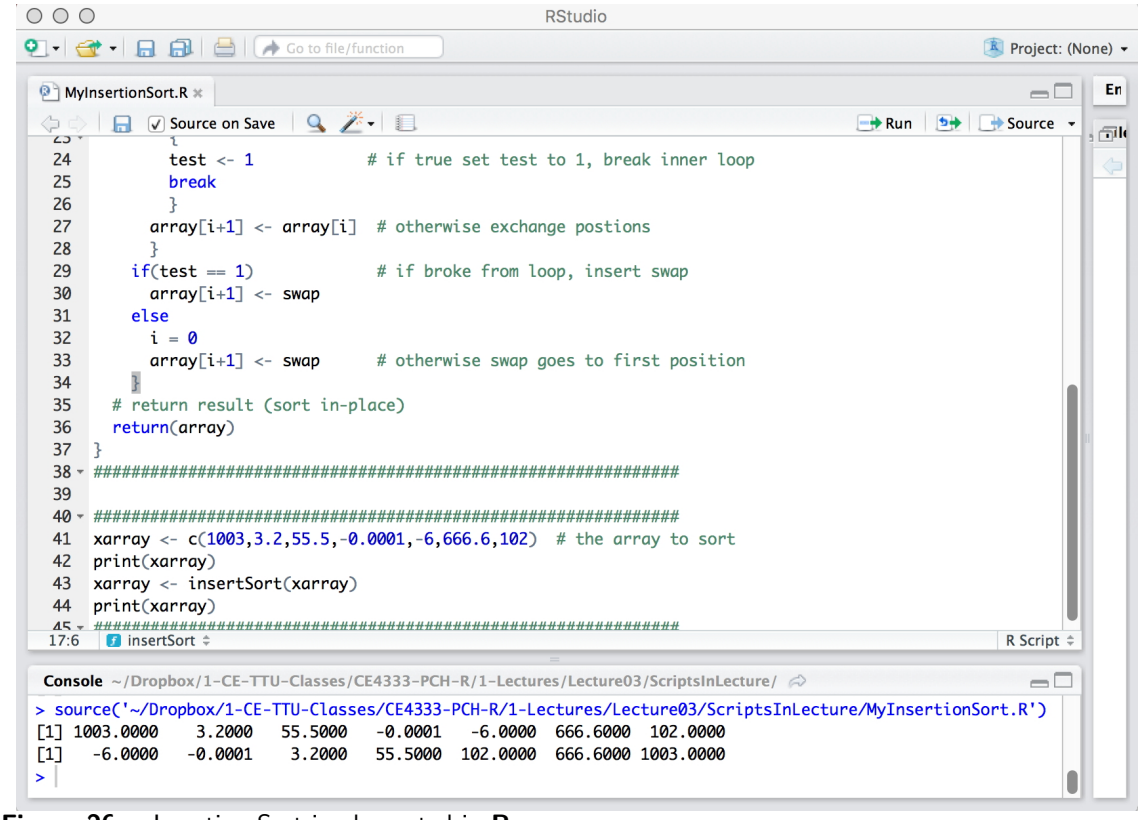

Figure 26. Insertion Sort implemented in **R**.

2.5.3 Merge Sort

A practical extension of these slow sorts is called the Merge Sort. It is an incredibly useful method. One simply breaks up the items into smaller arrays, sorts those arrays - then merges the sub-arrays into larger arrays (now already sorted), and finally merges the last two arrays into the final, single, sorted array.

Here is a better description, again from Christian and Griffiths (2016):

> "..... information processing began in the US censuses of the nineteenth century, with the development, by Herman Hollerith and later by IBM, of physical punch-card sorting devices. In 1936, IBM began producing a line of machines called "collators" that could merge two separately **ordered** stacks of cards into one. *As long as the two stacks were themselves sorted*, the procedure of merging them into a single sorted stack was incredibly straightforward and took linear time: simply compare the two top cards to each other, move the smaller of them to the new stack you're creating, and repeat until finished.
>
> The program that John von Neumann wrote in 1945 to demonstrate the power of the stored-program computer took the idea of collating to its beautiful and ultimate conclusion. Sorting two cards is simple: just put the smaller one on top. And given a pair of two-card stacks, both of them sorted, you can easily collate them into an ordered stack of four. Repeating this trick a few times, you'd build bigger and bigger stacks, each one of them already sorted. Soon enough, you could collate yourself a perfectly sorted full deck – with a final climactic merge, like a riffle shuffle's order-creating twin, producing the desired result. This approach is known today as Merge Sort, one of the legendary algorithms in computer science."

There are several other variants related to Merge Sort; Quicksort and Heapsort being relatives. The creation of a Merge Sort is left to the reader if there is a need, and at this point we can just use the built-in `sort()` and/or `order()` functions in **R** – which implements either a Shellsort (useful if character strings are to be sorted) or Quicksort (used if numeric values are supplied). We also have to supply if we want increasing or decreasing sorts.

2.5.4 Built-In R Sorts

Figure 27 illustrates using the built-in functions. For an ordinary sort, we simply use the function name `sort()` and direct its output into an object (it can even be the same vector as shown in the figure).

If we wish to sort several related columns, based on values in one of the columns, it is easiest to construct a data frame (like a matrix), then order the contents based on one of the columns, and send the results to another data frame, or we can send

the result back to itself. Usually when we are manipulating multiple columns, we are operating in a "relational database" kind of mindset, and it is probably to our benefit to not destroy the original association structure. Be aware of the syntax of a dataframe function, you will notice there is a comma that appears at the end of the function that is important for the script to function.

For example, `z <- z[order(xarray),]` will function as shown, whereas `zztop <- z[order(xarray)]` will not.

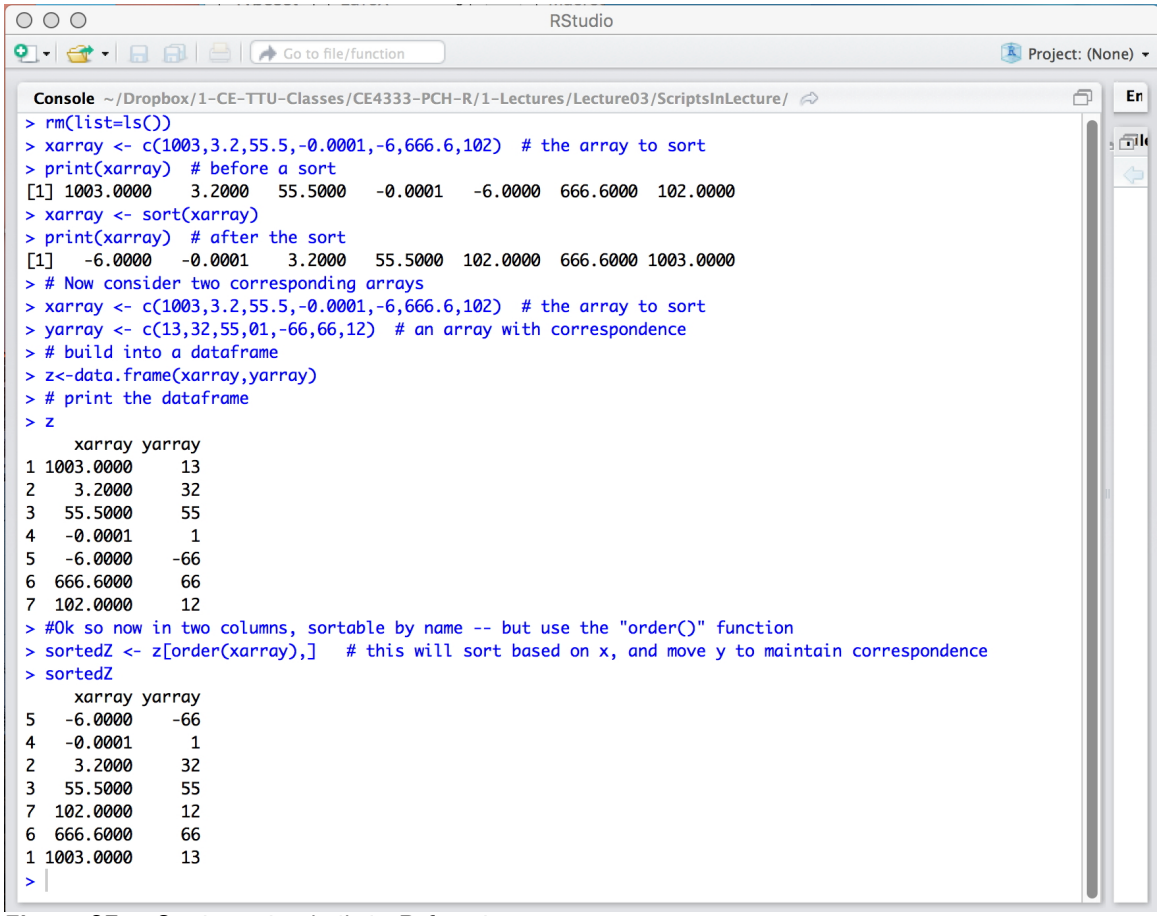

Figure 27. Sorting using built-in **R** functions.

Now if we return to the interpolation chapter just before this one, we can immediately see a need for sorting. The interpolation algorithm *assumes* that the explanatory structure (x-axis) is ordered, otherwise the interpolation equation will return garbage.

I conclude the section on sorting with one more quoted section from Christian and Griffiths (2016) about the value for sorting – which is already relevant to a lot of computational hydraulics:

> "The poster child for the advantages of sorting would be an Internet search engine like Google. It seems staggering to think that Google can take the search phrase you typed in and scour the entire Internet for it in less than

half a second. Well, it can't – but it doesn't need to. If you're Google, you are almost certain that (a) your data will be searched, (b) it will be searched not just once but repeatedly, and (c) the time needed to sort is somehow "less valuable" than the time needed to search. (Here, sorting is done by machines ahead of time, before the results are needed, and searching is done by users for whom time is of the essence.) All of these factors point in favor of tremendous up-front sorting, which is indeed what Google and its fellow search engines do."

2.6 Exercises

1. Build a function `getDensityUS()` that searches the table in Figure 23 and returns the density of water in US customary units for a value of temperature supplied in degrees Farenheight.

   Submit your code and screen captures of the density for temperatures of $43^o\ F$, $146^o\ F$, and $210^o\ F$.

2. Later in the class we will need functions to return viscosity to compute head losses in pipe networks.

   Build and test a function `getKinViscosityUS()` that searches the table in Figure 23 and returns the kinematic viscosity of water in US customary units for a value of temperature supplied in degrees Farenheight

   Submit the code and screen captures of the kinematic viscosity for temperatures of $43^o\ F$, $146^o\ F$, and $210^o\ F$.

3. Build and test a function `getKinViscositySI()` that searches the table in Figure 23 and returns the kinematic viscosity of water in SI units for a value of temperature supplied in degrees Celsius

   Submit the code and screen captures of the kinematic viscosity for temperatures of $13^o\ C$, $66^o\ C$, and $97^o\ C$.

4. Imagine you have two arrays, `array1` and `array2` that are linked in the sense that each element of `array1` is associated with the corresponding element of `array2`. You wish to sort based on contents of `array1` and maintain the correspondence of `array2`. In words we would simply modify the script to move and element of `array2` whenever you move an element of `array1`.

   Modify the Insertion Sort script to accept two arrays, the sort is based on contents of the first array and you are to maintain correspondence with the second array.

Test your script on the following arrays:
`array1` = $[5, 6, 8, 2, 3, 4, 1]$
`array2` = $[24, 35, 63, 3, 8, 15, 0]$

when these are reordered, the result should be:
`array1` = $[1, 2, 3, 4, 5, 6, 8]$
`array2` = $[0, 3, 8, 15, 35, 63]$

Then modify your script to read input from an ASCII file that contains the two arrays as columns (like in a spreadsheet) where you will sort on the first column, and carry along the correspondence with the second column.

Apply the script on the file `es3-pr1.txt`.

5. Repeat the exercise above but use built-in **R** functions.[4]

[4] My solution uses the `data.frame()` and `order()` functions. There are probably several other ways to accomplish the goal – corresponding sorts are hugely important in many practical situations.

3 Numerical Methods – Integrals, Derivatives, and Newton's Method

3.1 Numerical Integration of Functions

Numerical integration is the numerical approximation of

$$I = \int_a^b f(x)dx \tag{7}$$

Consider the problem of determining the shaded area under the curve $y = f(x)$ from $x = a$ to $x = b$, as depicted in Figure 28, and suppose that analytical integration is not feasible.

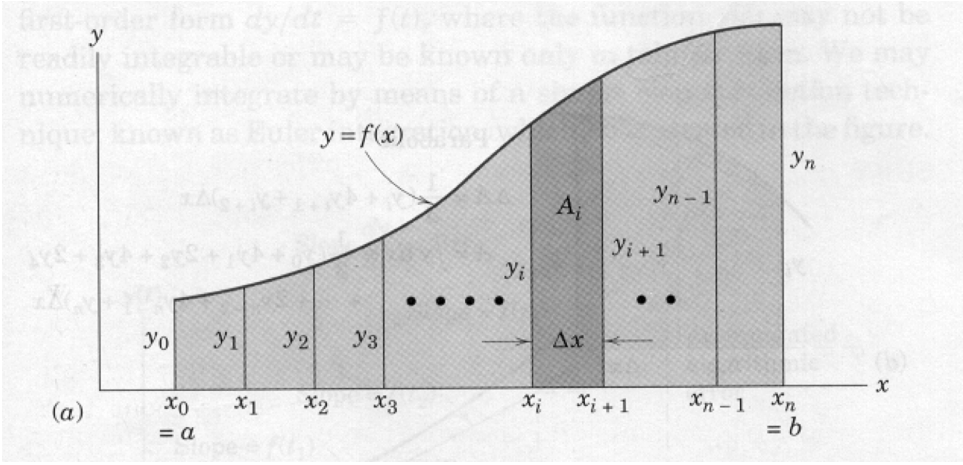

Figure 28. Schematic of Panels for Numerical Integration. .

The function may be known in tabular form from experimental measurements or it may be known in an analytical form. The function is taken to be continuous within the interval $a < x < b$. We may divide the area into n vertical panels, each of width $\Delta x = (b - a)/n$, and then add the areas of all strips to obtain $A \approx \int ydx$.

A representative panel of area A_i is shown with darker shading in the figure. Three useful numerical approximations are listed in the following sections. The approximations differ in how the function is represented by the panels — in all cases the function is approximated by known polynomial models between the panel end points.

In each case the greater the number of strips, and correspondingly smaller value of Δx, the more accurate the approximation. Typically, one can begin with a relatively small number of panels and increase the number until the resulting area approximation stops changing.

3.1.1 Rectangular Panels

Figure 29 is a schematic of a rectangular panels. The figure is assuming the function structure is known and can be evaluated at an arbitrary location in the Δx dimension. Each panels is treated as a rectangle, as shown by the representative panel whose

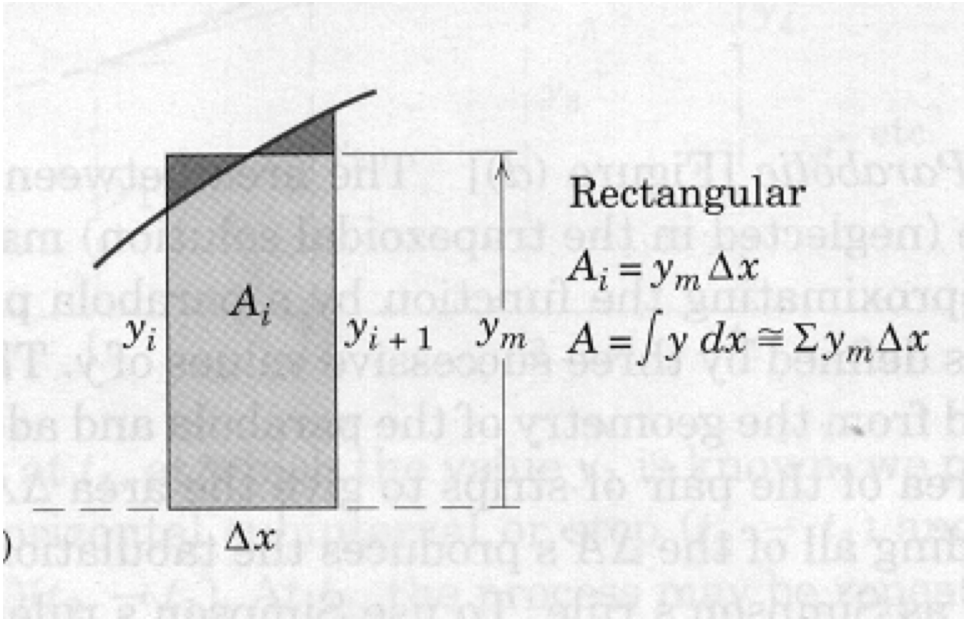

Figure 29. Rectangular Panel Schematic..

height y_m is chosen visually so that the small cross-hatched areas are as nearly equal as possible. Thus, we form the sum $\sum y_m$ of the effective heights and multiply by Δx. For a function known in analytical form, a value for y_m equal to that of the function at the midpoint $x_i + \Delta x/2$ may be calculated and used in the summation.

For tabulated functions, we have to choose to either take y_m as the value at the left endpoint or right endpoint. This limitation is often quite handy when we are trying to integrate a function that is integrable, but undefined on one endpoint.

Lets try some examples in **R**.

Problem: Find the area under the curve $y = x\sqrt{1+x^2}$ from $x = 0$ to $x = 2$.

Solution: One solution is shown in Figure 30,which is a screen capture of a rudimentary code that implements the rectangular panel method.[5]

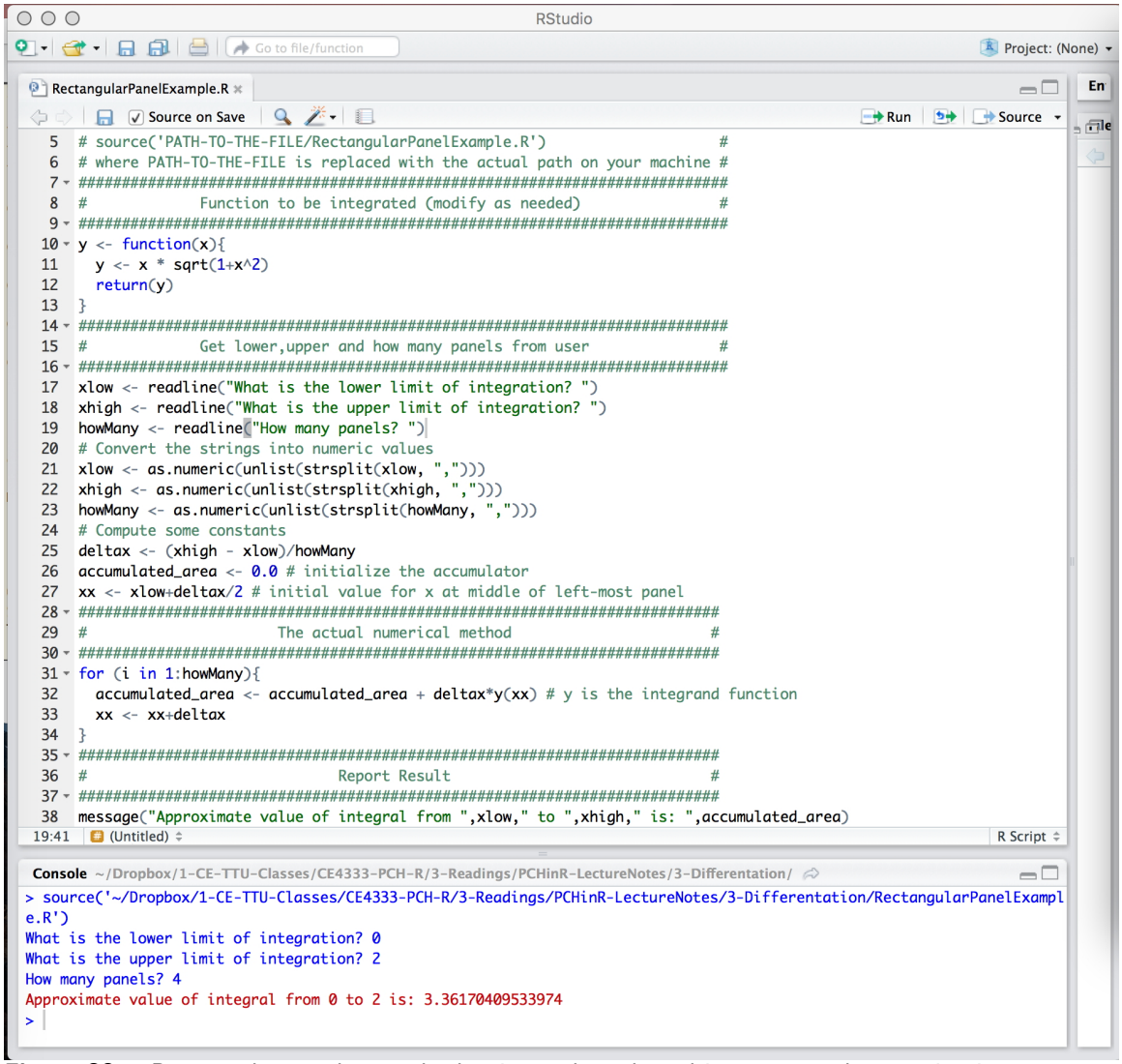

Figure 30. Rectangular panel example showing code and resulting computed area using just 4 panels..

The script does not implement any kind of error checking – we could enter text values for the lower and upper values of x as well as the number of panels to use, and the script would attempt to run. A better version would force us to enter numeric values, and check for undefined ranges and such; devotion to error trapping is typical for professional programs where you are going to distribute executable modules and not expect the end user to be a programmer.

[5]The exact solution is A=3.393477

For the time being, we will accept this approach (error trapping is left as an exercise), however in your own scripts you should implement error traps where possible – you may start without them but as you maintain your scripts, you will learn where data entry errors occur and trap them.[6]

The actual listing depicted in Figure 30 is shown in Listing 9

Listing 9. R code demonstrating Rectangular Panel Numerical Integration.

```
# R script to implement rectangular panel numerical integration
################################## NOTE ###################################
## The interactive input requires the script to be sourced               #
# In R console the command line would be                                 #
# source('PATH-TO-THE-FILE/RectangularPanelExample.R')                   #
# where PATH-TO-THE-FILE is replaced with the actual path on your machine #
###########################################################################
#             Function to be integrated (modify as needed)               #
###########################################################################
y <- function(x){
  y <- x * sqrt(1+x^2)
  return(y)
}
###########################################################################
#             Get lower,upper and how many panels from user              #
###########################################################################
xlow <- readline("What is the lower limit of integration? ")
xhigh <- readline("What is the upper limit of integration? ")
howMany <- readline("How many panels? ")
# Convert the strings into numeric values
xlow <- as.numeric(unlist(strsplit(xlow, ",")))
xhigh <- as.numeric(unlist(strsplit(xhigh, ",")))
howMany <- as.numeric(unlist(strsplit(howMany, ",")))
# Compute some constants
deltax <- (xhigh - xlow)/howMany
accumulated_area <- 0.0 # initialize the accumulator
xx <- xlow+deltax/2 # initial value for x at middle of left-most panel
###########################################################################
#                        The actual numerical method                      #
###########################################################################
for (i in 1:howMany){
  accumulated_area <- accumulated_area + deltax*y(xx) # y is the integrand function
  xx <- xx+deltax
}
###########################################################################
#                              Report Result                              #
###########################################################################
message("Approximate value of integral from ",xlow," to ",xhigh," is: ",accumulated_area)
```

Figure 31 is the same program run using 4,400, and 4000 panels observe the difference in computed area as well as the results closeness to the exact solution.

3.1.2 Trapezoidal Panels

The trapezoidal panels are approximated as shown in Figure 32. The area A_i is the average height $(y_i + y_{i+1})/2$ times Δx. Adding the areas gives the area approximation as tabulated. For the example with the curvature shown, the approximation will be on the low side. For the reverse curvature, the approximation will be on the high side. The trapezoidal approximation is commonly used with tabulated values.

The same example as presented for rectangular panels is repeated, except using trapezoidal panels. The code is changed because we will evaluate at each end of the panel (so no fussing to find an intermediate estimate for where to evaluate the function).

[6] For example, traps are used to force the user to enter a value that actually is meaningful, or select a default value, or internally prevent division by zero.

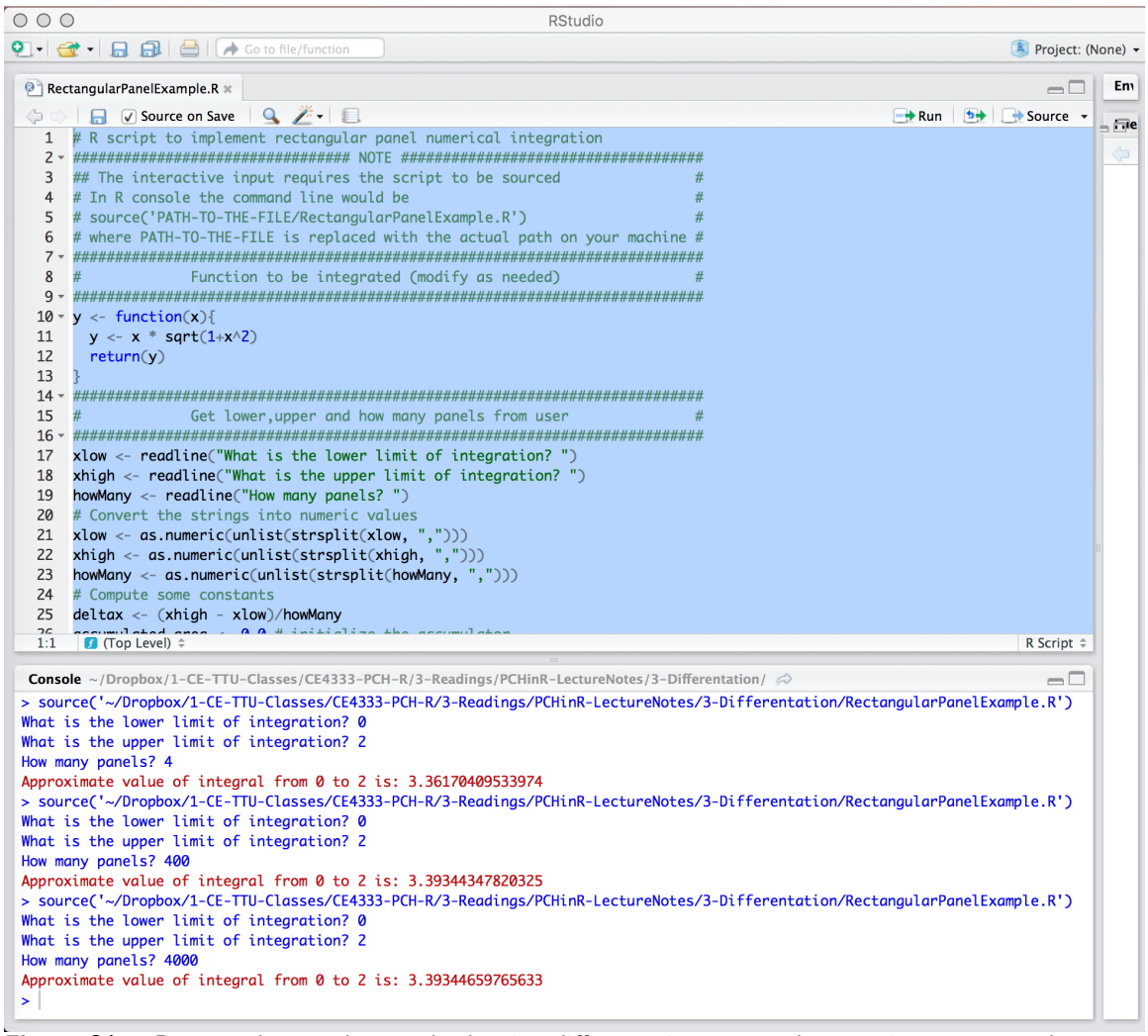

Figure 31. Rectangular panel example showing difference in computed area using 4, 400, and 4000 panels. The 4000 panel result is essentially equivalent to the exact solution. Such convergence to exact values is typical.

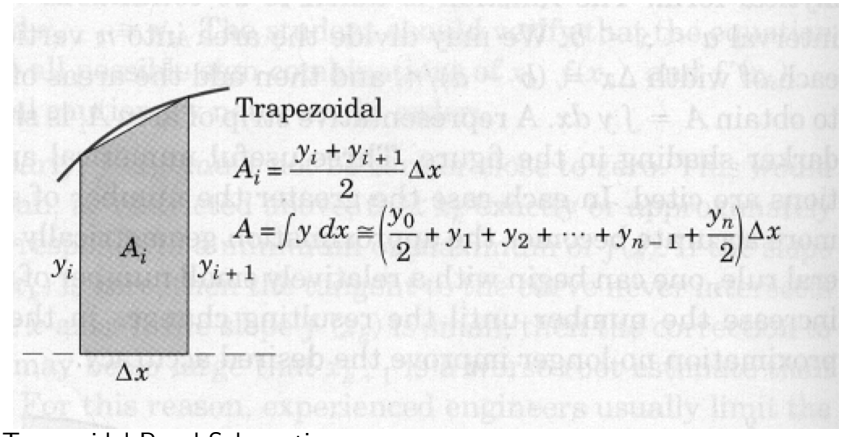

Figure 32. Trapezoidal Panel Schematic..

Figure 33 illustrates the trapezoidal method for approximating an integral. In the example, the left and right panel endpoints in x are set as separate variables x_{left} and x_{right} and incremented by Δx as we step through the count-controlled repetition to accumulate the area. The corresponding y values are computed within the loop and averaged, then multiplied by Δx and added to the accumulator. Finally the x values are incremented.

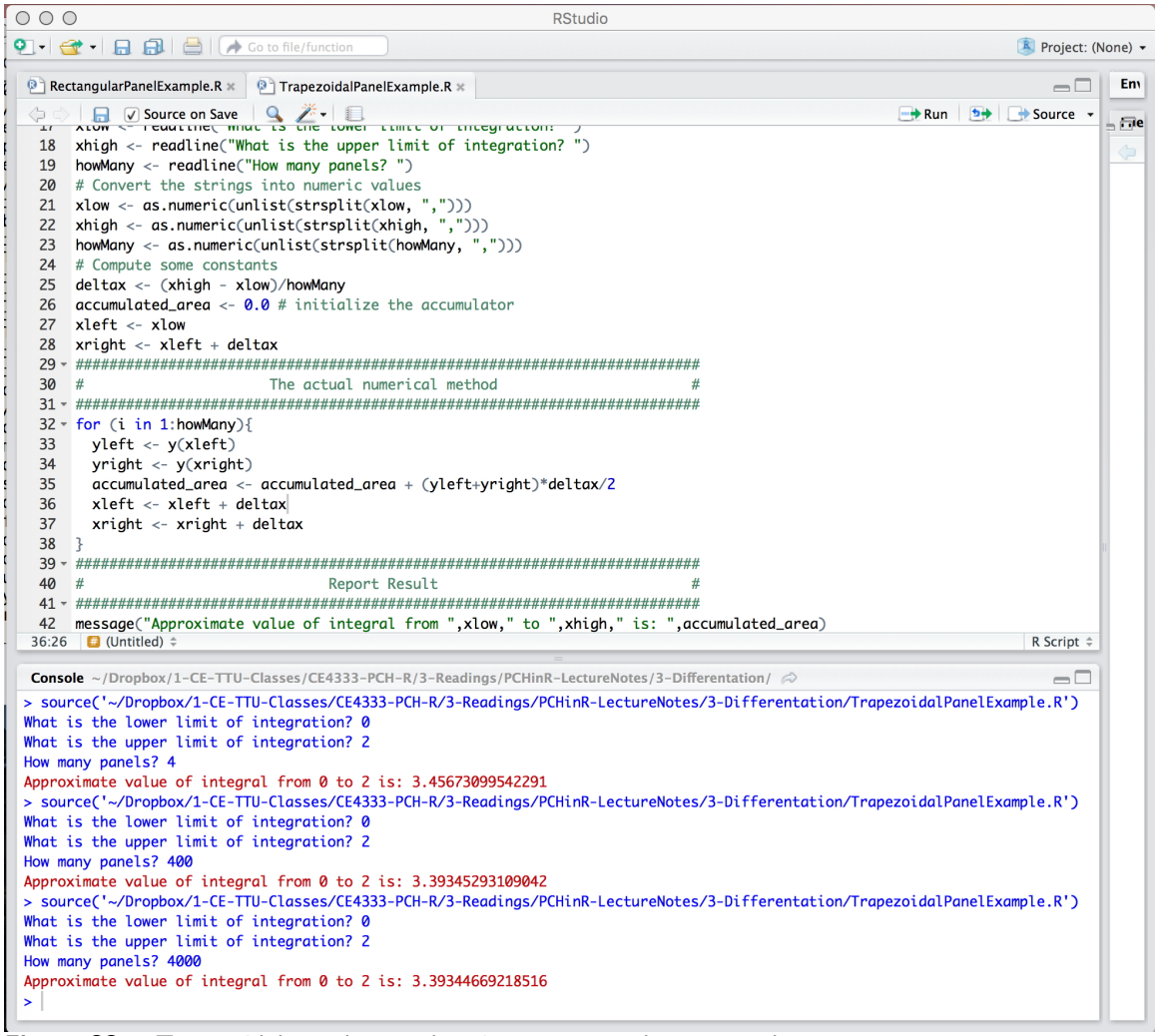

Figure 33. Trapezoidal panel example using 4, 400, and 4000 panels..

The actual listing depicted in Figure 33 is shown in Listing 10. Observe (at least for this example) the method appears more accurate that the rectangular method for the same number of panels, however also observe we are making twice as many function calls.

Listing 10. R code demonstrating Trapezoidal Panel Numerical Integration.

```
# R script to implement trapezoidal panel numerical integration
################################## NOTE ###################################
## The interactive input requires the script to be sourced               #
# In R console the command line would be                                 #
# source('PATH-TO-THE-FILE/TrapezoidalPanelExample.R')                   #
# where PATH-TO-THE-FILE is replaced with the actual path on your machine #
###########################################################################
#             Function to be integrated (modify as needed)               #
###########################################################################
y <- function(x){
  y <- x * sqrt(1+x^2)
  return(y)
}
###########################################################################
#             Get lower,upper and how many panels from user              #
###########################################################################
xlow <- readline("What is the lower limit of integration? ")
xhigh <- readline("What is the upper limit of integration? ")
howMany <- readline("How many panels? ")
# Convert the strings into numeric values
xlow <- as.numeric(unlist(strsplit(xlow, ",")))
xhigh <- as.numeric(unlist(strsplit(xhigh, ",")))
howMany <- as.numeric(unlist(strsplit(howMany, ",")))
# Compute some constants
deltax <- (xhigh - xlow)/howMany
accumulated_area <- 0.0 # initialize the accumulator
xleft <- xlow
xright <- xleft + deltax
###########################################################################
#                        The actual numerical method                      #
###########################################################################
for (i in 1:howMany){
  yleft <- y(xleft)
  yright <- y(xright)
  accumulated_area <- accumulated_area + (yleft+yright)*deltax/2
  xleft <- xleft + deltax
  xright <- xright + deltax
}
###########################################################################
#                              Report Result                              #
###########################################################################
message("Approximate value of integral from ",xlow," to ",xhigh," is: ",accumulated_area)
```

3.1.3 Parabolic Panels

Parabolic panels approximate the shape of the panel with a parabola. The area between the chord and the curve (neglected in the trapezoidal solution) may be accounted for by approximating the function with a parabola passing through the points defined by three successive values of y.

This area may be calculated from the geometry of the parabola and added to the trapezoidal area of the pair of strips to give the area ΔA of the pair as illustrated. Adding all of the ΔAs produces the tabulation shown, which is known as Simpson's rule. To use Simpson's rule, the number n of strips must be even.

The same example as presented for rectangular panels is repeated, except using parabolic panels. The code is changed yet again because we will evaluate at each end of the panel as well as at an intermediate value.

Figure 35 is a screen capture of a parabolic panel integration. The actual script is also listed in Listing 11. In the script, I substituted $\frac{\Delta x}{2}$ for Δx from Figure 34, so the accumulation line has a 6 in the denominator (rather than the 3 in the figure).[7]

Observe that the estimated integral for 400 and 4000 panels is nearly the same,

[7] $\ldots \frac{\Delta x}{2} \times \frac{1}{3} = \ldots \frac{\Delta x}{6}$

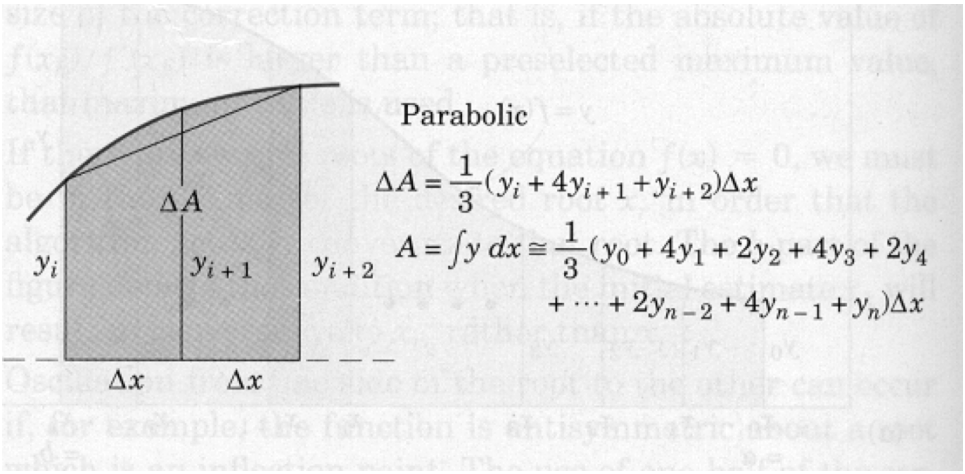

Figure 34. Parabolic Panel Schematic..

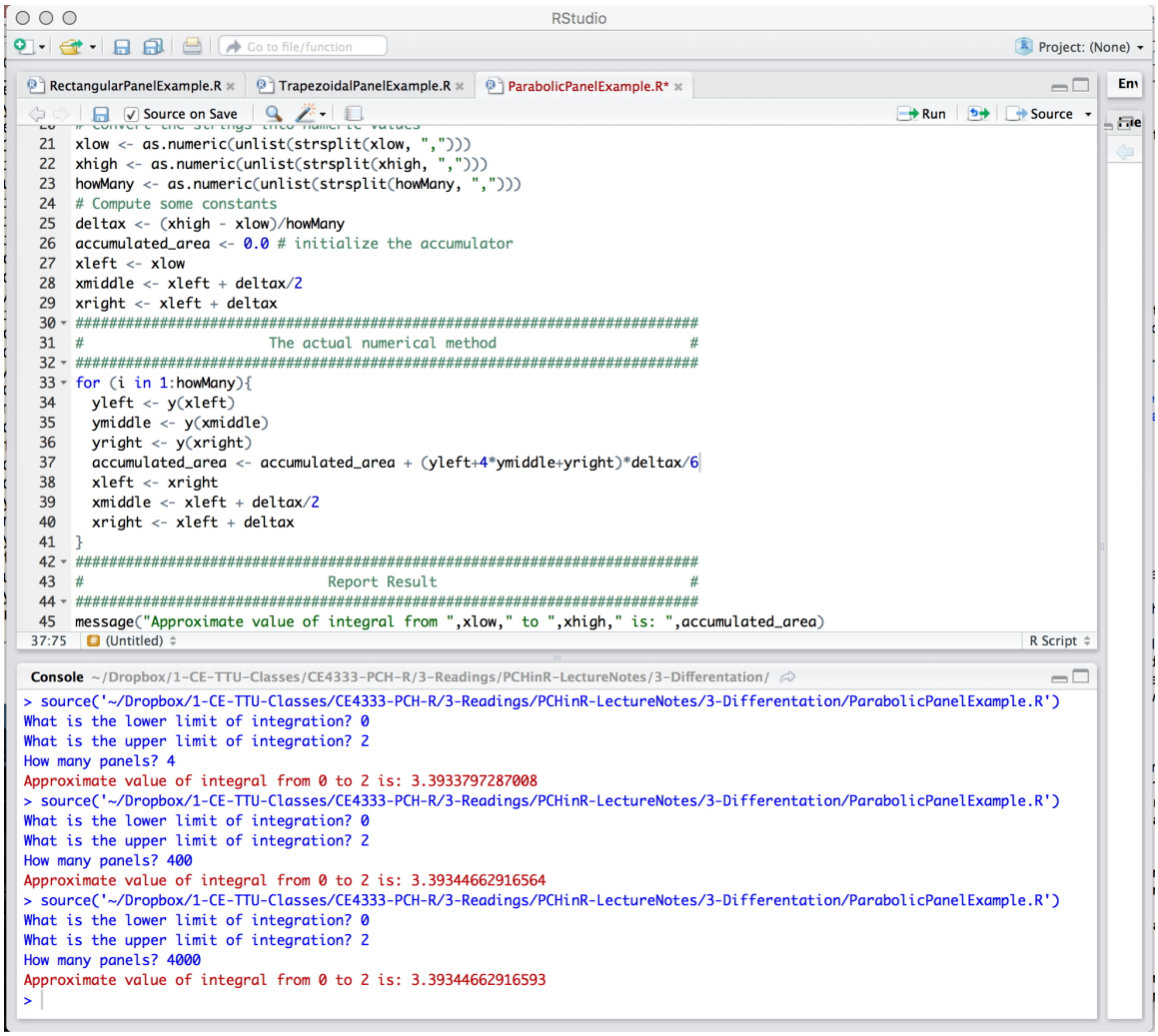

Figure 35. Parabolic panel example using 4, 400, and 4000 panels.

suggesting no need to go beyond a certain number of panels. Algorithms that detect when to stop adding panels exist and would be implemented in many scientific and engineering programming applications.

Listing 11. R code demonstrating Parabolic Panel Numerical Integration.

```
# R script to implement trapezoidal panel numerical integration
################################### NOTE ###################################
## The interactive input requires the script to be sourced                #
# In R console the command line would be                                  #
# source('PATH-TO-THE-FILE/RectangularPanelExample.R')                    #
# where PATH-TO-THE-FILE is replaced with the actual path on your machine #
###########################################################################
#             Function to be integrated (modify as needed)                #
###########################################################################
y <- function(x){
  y <- x * sqrt(1+x^2)
  return(y)
}
###########################################################################
#             Get lower,upper and how many panels from user               #
###########################################################################
xlow <- readline("What is the lower limit of integration? ")
xhigh <- readline("What is the upper limit of integration? ")
howMany <- readline("How many panels? ")
# Convert the strings into numeric values
xlow <- as.numeric(unlist(strsplit(xlow, ",")))
xhigh <- as.numeric(unlist(strsplit(xhigh, ",")))
howMany <- as.numeric(unlist(strsplit(howMany, ",")))
# Compute some constants
deltax <- (xhigh - xlow)/howMany
accumulated_area <- 0.0 # initialize the accumulator
xleft <- xlow
xmiddle <- xleft + deltax/2
xright <- xleft + deltax
###########################################################################
#                       The actual numerical method                       #
###########################################################################
for (i in 1:howMany){
  yleft <- y(xleft)
  ymiddle <- y(xmiddle)
  yright <- y(xright)
  accumulated_area <- accumulated_area + (yleft+4*ymiddle+yright)*deltax/6
  xleft <- xright
  xmiddle <- xleft + deltax/2
  xright <- xleft + deltax
}
###########################################################################
#                              Report Result                              #
###########################################################################
message("Approximate value of integral from ",xlow," to ",xhigh," is: ",accumulated_area)
```

If we study all the forms of the numerical method we observe that the numerical integration method is really the sum of function values at specific locations in the interval of interest, with each value multiplied by a specific weight. In this development the weights were based on polynomials, but other method use different weighting functions. An extremely important method is called gaussian quadrature, which is outside the scope of the discussion herein — Gaussian quadrature routines are readily available within **R**. The method is valuable because one can approximate convolution integrals quite effectively using quadrature routines, while the number of function evaluations for a polynomial based approximation could become hopeless.

When the function values are tabular, we are going to have to accept the rectangular (with adaptations) and trapezoidal as our best tool to approximate an integral because we don't have any really effective way to evaluate the function between the tabulated values – if we were to use our interpolation routine from earlier, its really going to be a kind of trapezoidal rule anyway.

3.2 Exercises

1. Write a script to approximate $\int_{1.8}^{3.4} e^x dx$ using rectangular panels.
   Run your script using 6 and 600 panels.
   (a) What is the analytical solution (e.g. do the calculus!)?
   (b) What is the percent error between the analytical solution and the approximation using 6 panels?
   (c) What is the percent error between the analytical solution and the approximation using 600 panels?
2. Write a script to approximate $\int_{1.8}^{3.4} e^x dx$ using trapezoidal panels.
   Run your script using 6 and 600 panels.
   (a) What is the analytical solution (e.g. do the calculus!)?
   (b) What is the percent error between the analytical solution and the approximation using 6 panels?
   (c) What is the percent error between the analytical solution and the approximation using 600 panels?
3. Based on the previous two exercises, which method do you think is more accurate for a given panel count? Why (do you think so)?
4. Write a script to approximate $\int_0^1 ln(x) dx$ using rectangular panels.
   Run your script using 6 and 600 panels.
   (a) What is the analytical solution (e.g. do the calculus!)?
   (b) What is the percent error between the analytical solution and the approximation using 6 panels?
   (c) What is the percent error between the analytical solution and the approximation using 600 panels?
5. Write a script to approximate $\int_0^1 ln(x) dx$ using trapezoidal panels.
   Run your script using 6 and 600 panels.
   (a) Did you get an error message — why?

3.3 Numerical Integration of Tabular Data

This subsection is going to work with tabular data — different from function evaluation, but similar. To be really useful, we need to learn how to read data from a file — manually entering tabular data is really time consuming, error prone, and just plain idiotic.

So in this subsection we will first learn how to read data from a file into a list, then we can process the list as if it were a function and integrate its contents.

3.3.1 Reading from a file – open, read, close files

R can read from an ASCII file (or even an Excel .csv file) using a multitude of methods. Common methods are `read.table(...)`, `read.table(...)`, `read.table(...)`, `read.table(...)`, and `read.table(...)`.

One can also use primatives[8] to read individual rows in a file and process them.[9]

First, lets create a file named `MyFile.txt`. The extension is important so that we can examine the file with other tools (a text editor) and remember that it is an ASCII file. The contents of `MyFile.txt` are:

```
1 , 1
2 , 4
3 , 9
4 , 16
5 , 25
```

To read the contents into an **R** script we have to do the following:

1. Open a connection to the file — this is a concept common to all languages, it might be called something different, but the program needs to somehow know the location and name of the file.
2. Read the contents into an object — we have a lot of control on how this gets done, for the time being we won't exercise much control yet. When you do substantial programs, you will depend on the control of the reads (and writes).
3. Disconnect the file — this too is common to all languages. Its a really easy step to forget. Not a big deal if the program ends as planned but terrible if there is a error in the program and the connection is still open. Usually noting bad happens, but with an open connection it is possible for the file to get damaged. If that file represents millions of customers data, that's kind of a problem.

[8] Jargon to describe lower level tools within **R**

[9] We will use this approach later in the book – the interactive prompt and reads in the prior subsection are similar to this approach where input is read into a string, then the string is converted into the appropriate type of object (numeric or text).

The `read.table` class of functions handles all three of these steps for us, we do have to provide the filename and some information about the file structure. Later when we are doing network simulation and other hydraulic techniques, different parts of an input file will be read line-by-line and processed — for this task we will need to handle these three steps using primatives.

Figure 36 illustrates the process. The input file has 5 lines, these get read then echoed (printed) back to us. The actual script is pretty simple, notice how the `filepath` and `filename` character variables are defined, then pasted together to produce a full *absolute* file name.[10]

Listing 12 is a listing of the script used in Figure 36. The analyst should be able to deduce that the filenames could be read from user input using the prompting technique used in the earlier subsections, so if one is going to process a lot of similar files the explicit naming could be replaced with variable naming – it would probably be a good idea to confine the files to a reasonably memorable path.

Listing 12. R code demonstrating Reading from a File.

```
# R script to illustrate reading from a file using read.table
# The script is intentionally complicated to illustrate the
# three steps: open connection, read into object, close connection
filepath <- "~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/3-
    Differentation/RScripts"
filename <- "MyFile.txt"
fileToRead <- paste(filepath,filename,sep="/") # build the user absolute filename
# Here we open the connection to the file (within read.table)
# Then the read.table attempts to read the entire file into an object named zz
# Upon either fail or success, read.table closes the connection
zz <- read.table(fileToRead,header=FALSE,sep=",") # comma seperated ASCII, No header
# Echo zz
print(zz)
```

Now that we can read a file, we are now able to integrate tabular data.

3.4 Integrating tabular data

Suppose instead of a function we only have tabulations and wish to estimate the area under the curve represented by the tabular values. Then our integration rules from the prior sections still work more or less, except the rectangular panels will have to be shifted to either the left edge or right edge of a panel (where the tabulation exists).

Lets just examine an example. Suppose some measurement technology produced Table 1 a table of related values. The excitation variable is x and $f(x)$ is the response.

[10] A file name can specify all the directory names starting from the root of the tree; then it is called an absolute file name. Or it can specify the position of the file in the tree relative to a default directory; then it is called a relative file name. On the computer I used to write this workbook, the symbol ~ , is the root to my user account, then the remaining directories from that location are explicitly listed. The actual absolute name is /Users/cleveland/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/3-Differentation/RScripts

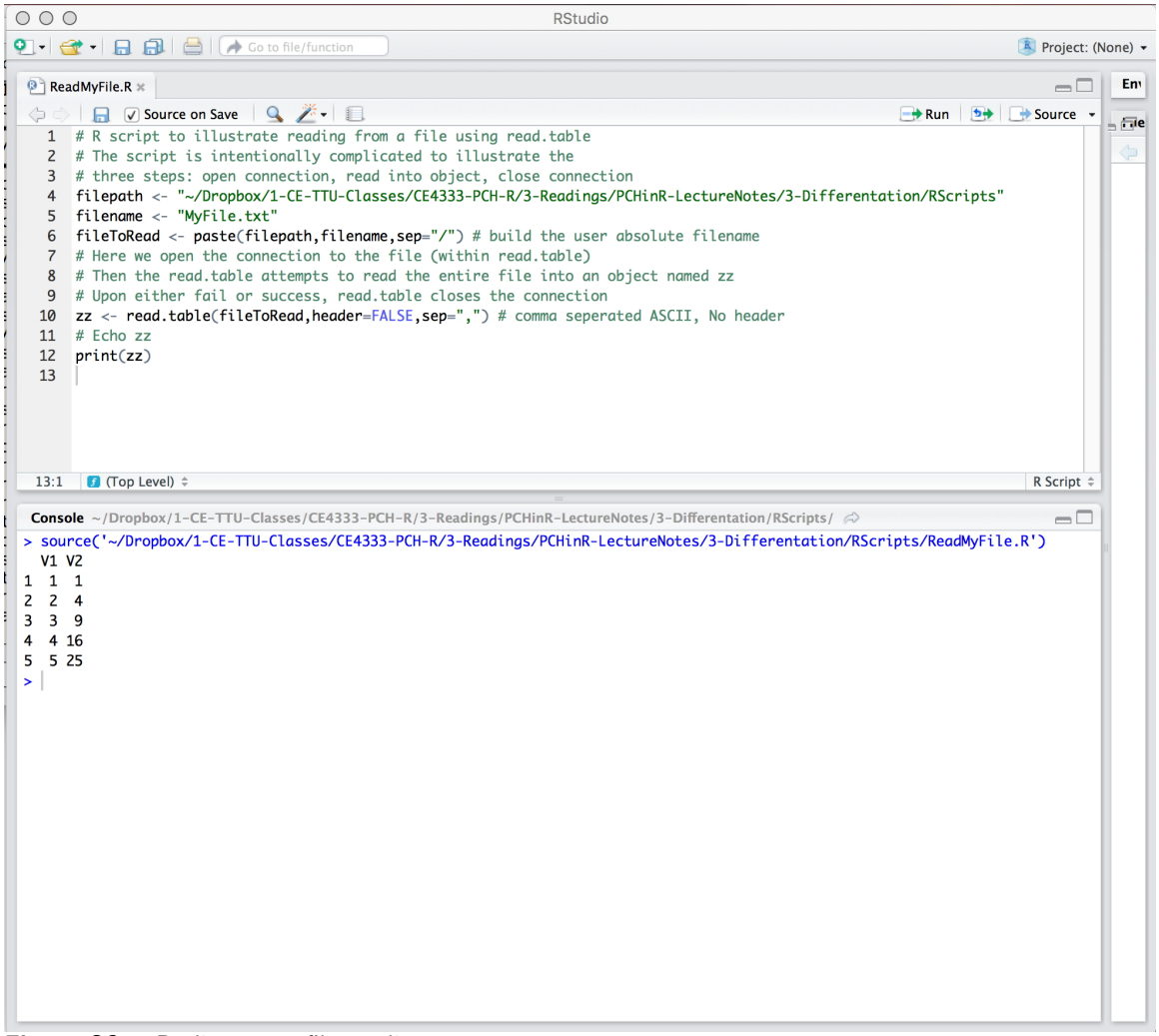

Figure 36. Rudimentary file reading.

Table 1. Tabular values of an excitation–response relationship.

x	$f(x)$
1.0	1.543
1.1	1.668
1.2	1.811
1.3	1.971
1.4	2.151
1.5	2.352
1.6	2.577
1.7	2.828
1.8	3.107

To integrate this table using the trapezoidal method is straightforward. We will modify our earlier code to read the table (which we put into a file), and compute the integral.

Figure 37 is a screen capture of a script that implements the file read and the numerical integration. The conversion of the method from the functional form in the previous section is pretty straightforward. The main nusiance here is the syntax required to access the "x" values and the "y" values. In **R** the most generic approach is `object$name`, where *object* is the data frame name, and *name* is the variable (column) name. If you don't use headers, **R** assigns names as `V1,V2,...,`V_{max}.

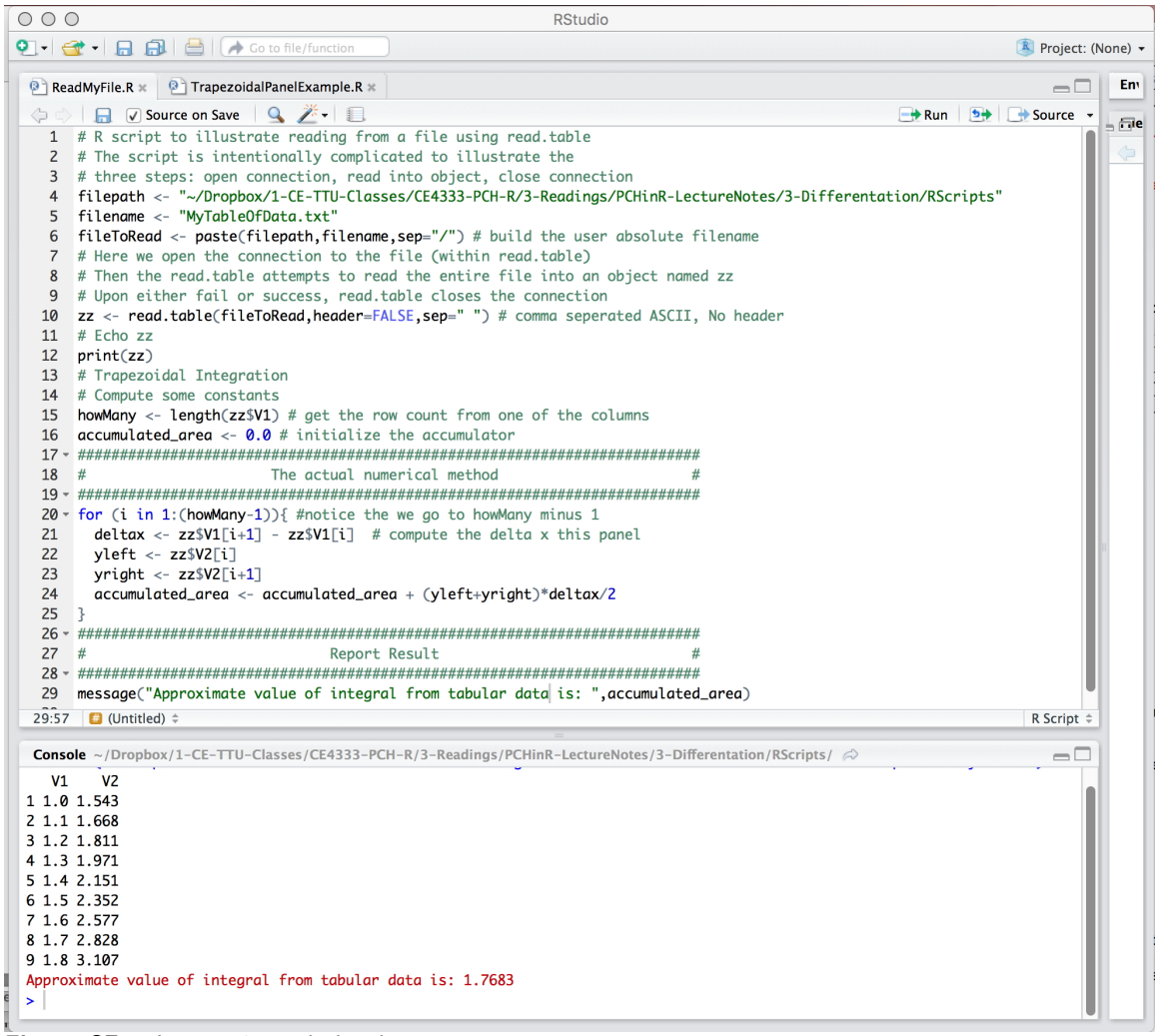

Figure 37. Integrating tabular data..

Realistically the only other simple integration method for tabular data is the rectangular rule, either using the left edge of a panel or the right edge of a panel (and you could do both and average the result which would result in the same outcome as the trapezoidal method). For the sake of completeness lets do both and then compare the results from all four approaches (trapezoidal, rectangular-left, rectangular-right, average rectangular).

First, Figure 38 implements the file read and tabular integration using the rectangular panel method, evaluating the function at the left edge of each panel.

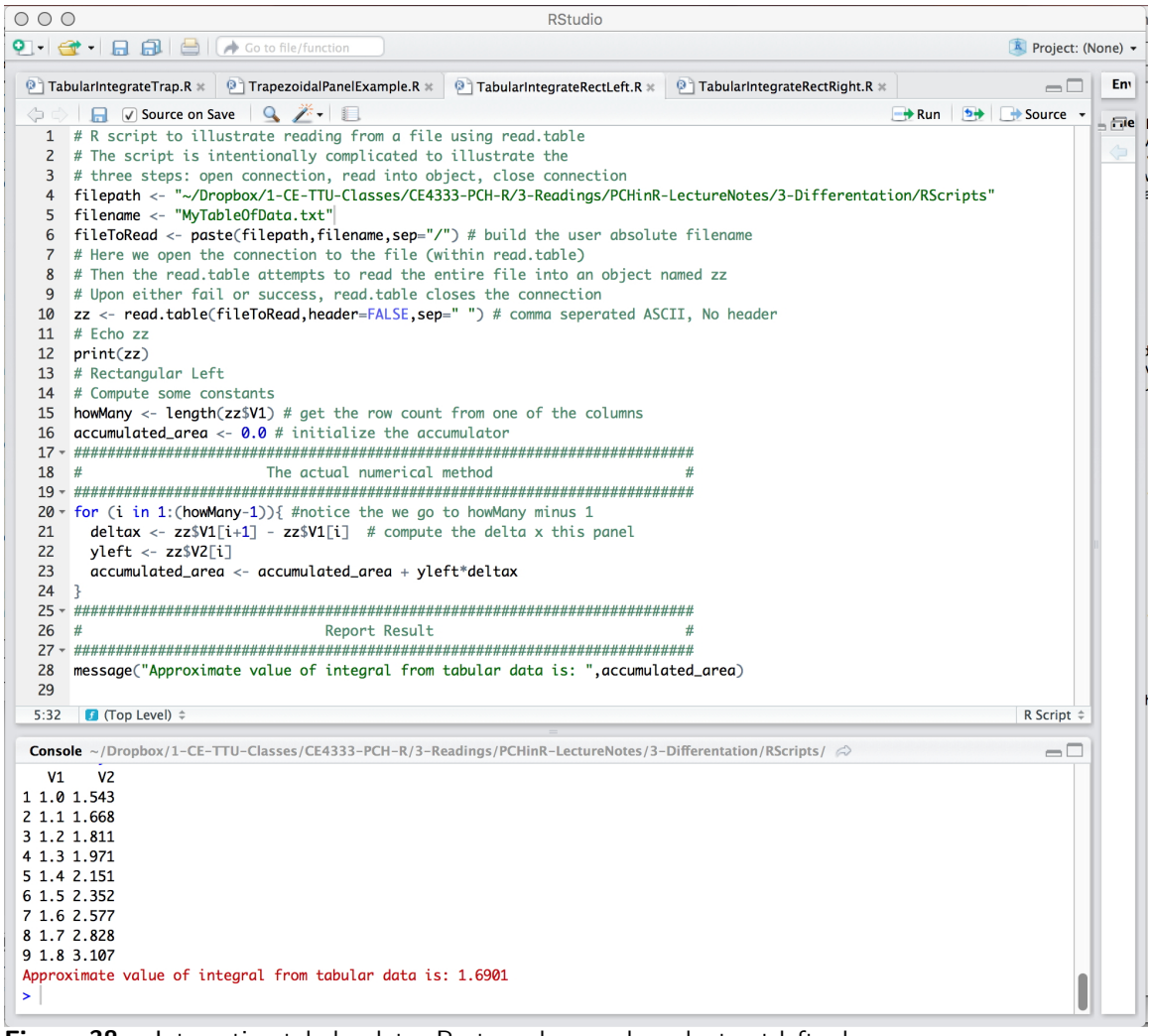

Figure 38. Integrating tabular data. Rectangular panel, evaluate at left edge..

Next, Figure 39 implements the file read and tabular integration using the rectangular panel method, evaluating the function at the left edge of each panel.

Now lets compare the results from using the three (four) approaches. Table 2 are the results by method.

Table 2. Comparison of tabular integration.

Method	Computed Area
Trapezoidal Panels	1.7683
Rectangular - Left Edge	1.6901
Rectangular - Right Edge	1.8465
Arithmetic Mean Rectangular	1.7683

What Table 2 illustrates is that the trapezoidal rule is simply the average of the rectangular rule evaluated at first the left-edge then the right-edge of a panel.

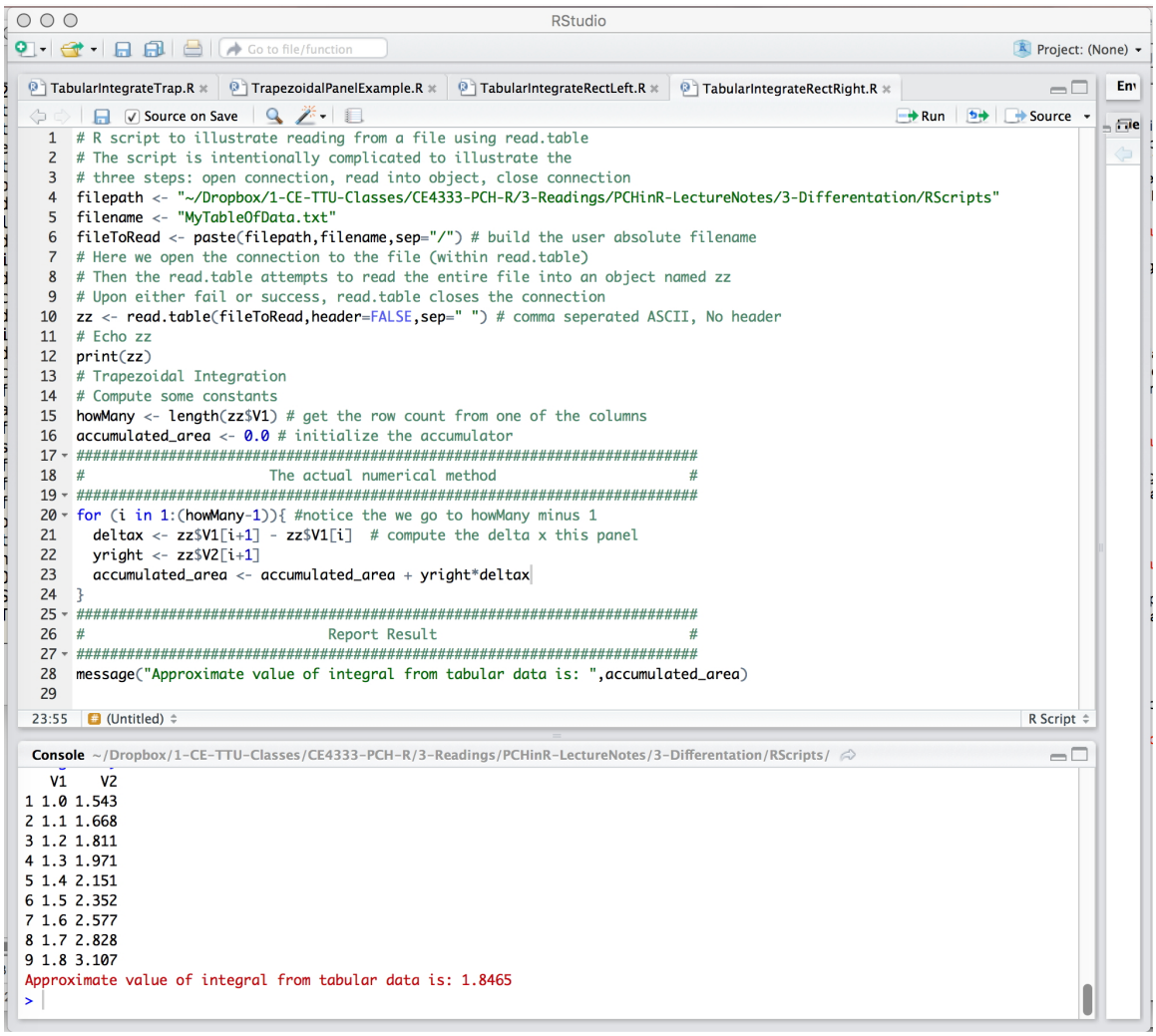

Figure 39. Integrating tabular data. Rectangular panel, evaluate at right edge..

3.5 Exercises

1. Approximate $\int_0^2 f(x)dx$ from the tabulation in Table 3.

Table 3. Tabular values of a function – non-uniform spacing in x.

x	$f(x)$
0.00	1.0000
0.12	0.8869
0.53	0.5886
0.87	0.4190
1.08	0.3396
1.43	0.2393
2.00	0.1353

2. Table 4 is a tabulation of various values of the hyperbolic cosine function.
   (a) Approximate $\int_1^{9.0} f(x)dx$ from the tabulation in Table 4.
   (b) Approximate $\int_1^{4.2} f(x)dx$ from the tabulation in Table 4.
   (c) Approximate $\int_1^{4.0} f(x)dx$ from the tabulation in Table 4.
   (d) Briefly explain how you choose to handle starting and stopping the integration from values that are intermediate and within the tabulation.
   (e) Also explain how you choose to handle working with values that fall between tabulated values.

Table 4. Tabular values of the hyperbolic cosine.

x	$f(x)$
1.0	1.54308063481524
1.1	1.66851855382226
1.2	1.81065556732437
1.3	1.97091423032663
1.4	2.15089846539314
1.5	2.35240961524325
1.6	2.57746447119489
1.7	2.82831545788997
1.8	3.10747317631727
2.0	3.76219569108363
2.2	4.56790832889823
2.4	5.55694716696551
2.6	6.76900580660801
2.8	8.25272841686113
3.0	10.0676619957778
3.3	13.5747610440296
3.6	18.3127790830626
3.9	24.711345508488
4.2	33.3506633088728
4.6	49.7471837388392
5.0	74.2099485247878
5.5	122.348009517829
6.0	201.715636122456
7.0	548.317035155212
8.0	1490.47916125218
9.0	4051.54202549259

3.6 Numerical Differentiation

Similar in context to numerical integration is approximation of derivatives. If the functions are representable as functions, then differencing degenerates into the selection of an appropriate difference formula. If the function is tabular, the same decision is presented, but we have to pay additional attention to the quantity of observations available.

3.7 Difference Approximations for Tabulated Data

Here we will introduce differencing by an example. Suppose we want to convert a cumulative data series into an incremental data series. It is operationally related to numerical differentiation.

As an example (leading to an algorithm) consider the cumulative rainfall time series in Table 5.

We shall import this data into **R**, then plot the data, then construct a computational procedure to extract the incremental values from the cumulative values. To load the data into **R** we start the program and then read the contents of a data file that contains the data into **R**, then we will introduce the plotting tools in **R**. In addition to plotting, we will also learn about headers and attaching an object (which gives access to header names rather than the `object$name` structure.

Table 5. Cumulative Rainfall Time Series.

hours	cumulative_rain
0.0	0.00
0.5	1.06
1.0	2.99
1.5	4.80
2.0	4.80
2.5	4.80
3.0	4.80
3.5	4.80
4.0	4.80
4.5	4.80
5.0	4.80
5.5	4.80

The plot suggests that the values are accumulated at the end of the time interval, thus the value accumulated is some average "rate" multiplied by the time interval. The line segment between each point is called a "secant" line. The slope of each secant line, provides that average "rate". So a fundamental computational step will be a

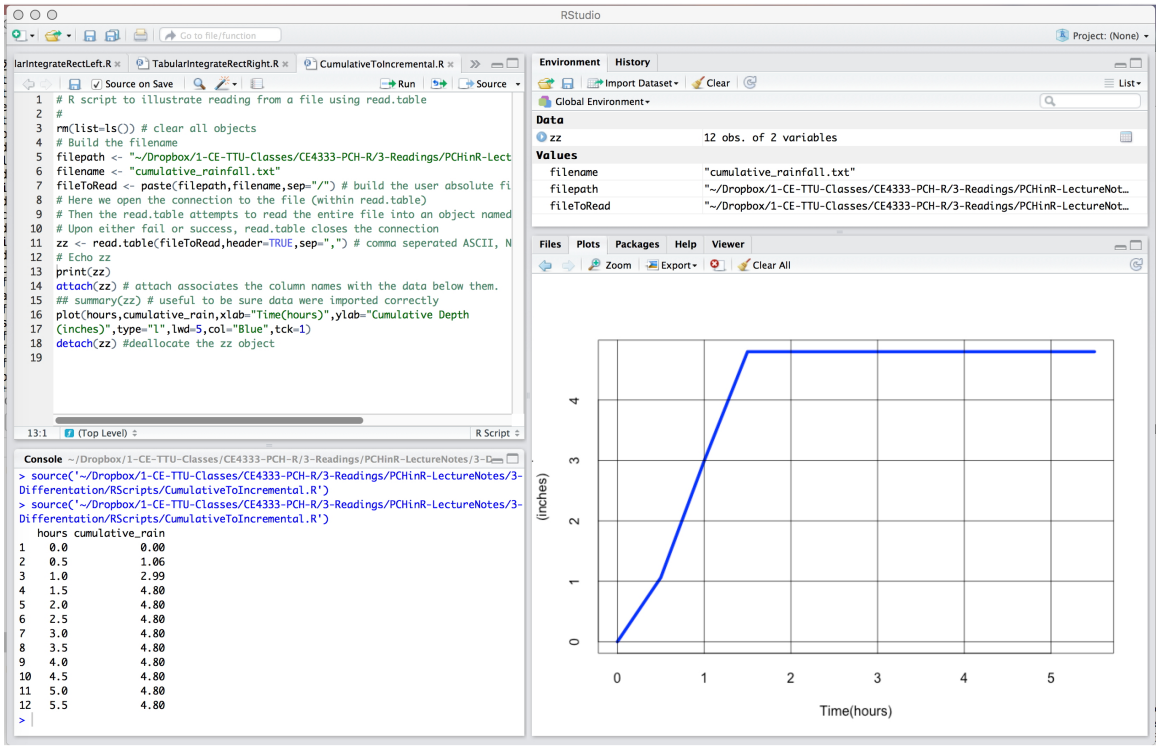

Figure 40. Plot of Cumulative Rainfall Time Series.

function that computes the slope given any two points (assumed to be adjacent — so that's why sorting can become important, although the program doesn't actually care).

3.7.1 Slope of a Secant Line

`slopeOfSecant` is a prototype function that we write to that computes the slope of the secant line through two known points on a function $f(x_1)$ and $f(x_2)$. The function could be tabular or evaluated. The script assumes tabular in that the function is evaluates external to `slopeOfSecant`.

Listing 13. R code demonstrating the prototype function `slopeOfSecant`.

```
############### slope function prototype ####################
slopeOfSecant<-function(f1,f2,x1,x2){
  slopeOfSecant <- (f2-f1)/(x2-x1);
  return(slopeOfSecant)
}
############################################################
```

This slope is also a first order approximation of a derivative (forward, backward, and centered differences depending on values supplied). This function can then be used to compute "derivatives" of data series using a disaggregation function.

As an illustrative example, if we present parts of the cumulative rainfall data series we can recover the average rate between the inputs. Figure 41 is a screen capture of such a test.

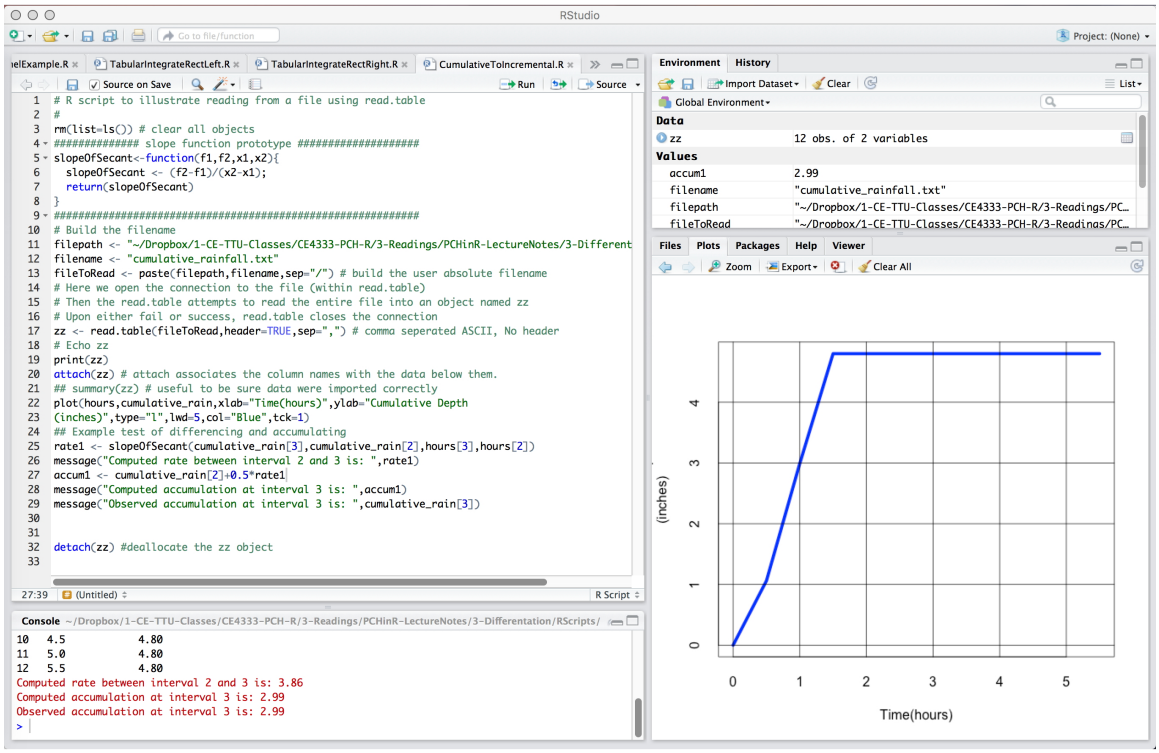

Figure 41. Script with slopeOfSecant prototype inserted and validation that we recover rate and can re-accumulate correctly.

3.7.2 Disaggregation

`disaggregate` is a prototype function that computes the slopes of the secant lines joining adjacent pairs of input data. Depending on the way the input arrays are presented to the `disaggregate` function, the function returns either the backward difference approximation to the function's derivative or if an index is presented instead of actual t values, then the function returns the incremental values that when aggregated reconstruct the original input function.

Listing 14. R code demonstrating the prototype function `disaggregate()`.

```
########      disaggregate function prototype    ############
# returns a vector of slopes computed by sloepOfSecant
disaggregate<-function(f,x,dfdx){
  n<-length(x) # length of vectors
  dfdx<-rep(0,n); # zero dfdx
  for (i in 2:n){dfdx[i]<-slopeOfSecant(f[i-1],f[i],x[i-1],x[i]);};
  dfdx[1]<-0;
  return(dfdx)}
################################################################
```

3.7.3 Numerical Differentiation

A related concept is to determine the average rate for the time interval, the principal difference is that the rate occurs during the entire time interval and should be assigned to the beginning of the interval instead of the end of the interval. A subtle change in

the `disaggregate` function can accomplish the task, we will name that new function `brbt`. The name is a nemonic for "backward-rate, backward-time" differencing.

Listing 15. R code demonstrating the prototype function `brbt()`.

```
############ backward rate, backward time prototype ############
brbt<-function(f,x,dfdx){
  n<-length(x) # length of vectors
  dfdx<-rep(0,n); # zero dfdx
  for (i in 1:(n-1)){dfdx[i]<-slopeOfSecant(f[i],f[i+1],x[i],x[i+1]);};
  dfdx[n]<-0;
  return(dfdx)}
################################################################
```

Finally, putting everything together, we have the toolkit to determine the incremental rates (which is an approximation to the derivative of the cumulative rates) and incremental depths which are these individual rates multiplied by the length of the time interval. Figure 42 is a screen capture of the **R** script that implements these functions on the tabular data.

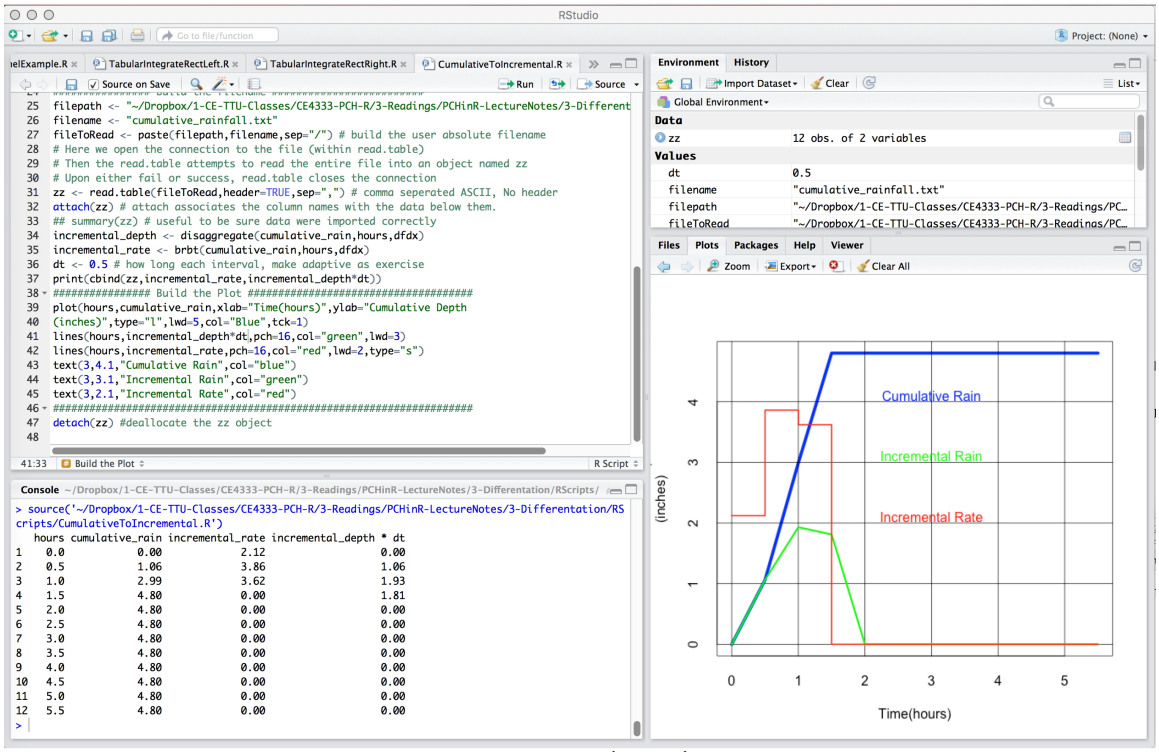

Figure 42. Plot of Cumulative Rainfall Time Series (BLUE), Incremental Depth Time Series (GREEN), and Average Rate Time Series (RED).

Listing 16 is a listing of the **R** script that produced Figure 42. The intermediate steps from Figure 41 is removed in this listing.

Listing 16. R code demonstrating Numerical Differencing.

```
# R script to illustrate numerical differencing
rm(list=ls()) # clear all objects
############# Prototype (forward define) Functions #########
############### slope function prototype ####################
slopeOfSecant<-function(f1,f2,x1,x2){
  slopeOfSecant <- (f2-f1)/(x2-x1);
  return(slopeOfSecant)}
```

```
########      disaggregate function prototype    ############
disaggregate<-function(f,x,dfdx){
  n<-length(x) # length of vectors
  dfdx<-rep(0,n); # zero dfdx
  for (i in 2:n){dfdx[i]<-slopeOfSecant(f[i-1],f[i],x[i-1],x[i]);};
  dfdx[1]<-0;
  return(dfdx)}
############ backward rate, backward time prototype ############
brbt<-function(f,x,dfdx){
  n<-length(x) # length of vectors
  dfdx<-rep(0,n); # zero dfdx
  for (i in 1:(n-1)){dfdx[i]<-slopeOfSecant(f[i],f[i+1],x[i],x[i+1]);};
  dfdx[n]<-0;
  return(dfdx)}
################################################################

################ Build the filename ########################
filepath <- "~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/3-
    Differentation/RScripts"
filename <- "cumulative_rainfall.txt"
fileToRead <- paste(filepath,filename,sep="/") # build the user absolute filename
# Here we open the connection to the file (within read.table)
# Then the read.table attempts to read the entire file into an object named zz
# Upon either fail or success, read.table closes the connection
zz <- read.table(fileToRead,header=TRUE,sep=",") # comma seperated ASCII, No header
attach(zz) # attach associates the column names with the data below them.
## summary(zz) # useful to be sure data were imported correctly
incremental_depth <- disaggregate(cumulative_rain,hours,dfdx)
incremental_rate <- brbt(cumulative_rain,hours,dfdx)
dt <- 0.5 # how long each interval, make adaptive as exercise
incremental_depth <- incremental_depth*dt
print(cbind(zz,incremental_rate,incremental_depth))
################ Build the Plot ######################################
plot(hours,cumulative_rain,xlab="Time(hours)",ylab="Cumulative Depth
(inches)",type="l",lwd=5,col="Blue",tck=1)
lines(hours,incremental_depth*dt,pch=16,col="green",lwd=3)
lines(hours,incremental_rate,pch=16,col="red",lwd=2,type="s")
text(3,4.1,"Cumulative Rain",col="blue")
text(3,3.1,"Incremental Rain",col="green")
text(3,2.1,"Incremental Rate",col="red")
#####################################################################
detach(zz) #deallocate the zz object
```

3.7.4 Aggregation

Aggregation is the compliment of disaggregation; instead of finding differences we are trying to produce cumulatives from incremental values or rates. Aggregation is to integration as disaggregation is to differentiation. Simple aggregation functions are straightforward to build. Numerical integration (already introduced) is a bit more challenging because there are many different ways to compute areas from tabular data – we will illustrated rectangular, trapezoidal, and parabolic panels.

We can insert a prototype `aggregate` function that simply adds elements in a series to prior elements and stores the value in another series. Another name for this kind of arithmetic is a running sum. Functionally, it is rectangular panel (evaluate from the left), numerical integration.

Listing 17. R code demonstrating the prototype function `aggregate()`.

```
############ aggregate function prototype ############
aggregate<-function(vector1,vector2){
n<-length(vector1)
# fill vector2 with zeros
vector2<-rep(0,n)
vector2[1]<-vector1[1]+0.0
for(i in 2:n)vector2[i]<-vector2[i-1]+vector1[i]
return(vector2)}
################################################
```

We add this function to the prototype list ate the top of the script and can run To illustrate the use of `aggregate` we will aggregate the incremental depths into the

cumulative rainfall – we should recover the original cumulative rainfall series that was originally supplied.

3.8 Exercises

1. Add the `aggregate` prototype function to collection of prototype functions in the script. The add some code like:

   ```
   new_cum_rain<-aggregate(incremental_depth,dummy)
   plot(hours,cumulative_rain,xlab="Time(hours)",ylab="Cumulative Depth
   (inches)",type="l",lwd=5,col="Blue",tck=1)
   lines(hours,new_cum_rain,col="red",lwd=1.5)
   ```

   Demonstrate that the two series are identical (e.g. plotting on top of one another).

2. (Advanced) Modify the `disaggregate()` prototype function to automatically determine the time spacing (x) and perform the correct multiplication within the function to return the correct increments. You only have to add one line of code to the prototype function at

   ```
   for (i in 2:n){dfdx[i]<-slopeOfSecant(f[i-1],f[i],x[i-1],x[i]);
   deltax <-   # you need to define this in terms of x[i] and x[i-1]!
   dfdx[i]<- deltax*dfdx[i];};
   ```

3.9 Finite-Difference Formulas

What we have just done is to explore the use of finite-difference approximations for derivatives. Some common formulas for difference formulas are listed below (without derivation – you should be able to find explanation in any numerical methods text). All the difference equations presented here are the result of truncated Taylor series expansions about x. The "order" refers to the magnitude of truncation error, and this magnitude is proportional to the step size (Δx) raised to a power (the order). Truncation error decreases as the step size is decreased, but one is approaching a divide-by-zero situation (because numerical methods don't do limits just yet!).

3.9.1 First Derivatives

Equation 8 is a first-order backwards difference.

$$\frac{df}{dx} \approx \frac{f(x) - f(x - \Delta x)}{\Delta x} \tag{8}$$

Equation 9 is a first-order backwards difference.

$$\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{9}$$

Equation 10 is a second-order central difference.

$$\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} \tag{10}$$

3.9.2 Second Derivatives

Equation 11 is a second-order central difference.

$$\frac{d^2f}{dx^2} \approx \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2} \tag{11}$$

3.9.3 Third Derivatives

Equation 12 is a sixth-order central difference.

$$\frac{d^3f}{dx^3} \approx \frac{f(x+2\Delta x) - 2f(x+\Delta x) + 2f(x-\Delta x) + f(x-2\Delta x)}{2\Delta x^3} \tag{12}$$

The procedure to generate such difference formulas is general and can supply estimates with approximations of any degree. The accuracy depends on the location and number of field variable values involved in the approximation. The selection of a formula is not at all trivial (especially with tabulations), but beyond the scope of this handbook.

Despite known complications, this is a general tool used in computational hydraulics and we will use it throughout the remainder of the handbook – in some examples it will not be obvious that it is finite differencing, and in others it will be explicitly obvious. The next section introduces Newton's method, and finite-differences will be used to approximate the derivative (Quasi-Newton) to implement the method. The procedure is really quite common and imbedded in a lot of the computational tools we use professionnaly.

3.10 Single Variable Quasi-Newton Methods

The application of fundamental principles of modeling and mechanics often leads to an algebraic or transcendental equation that cannot be easily solved and represented in a closed form. In these cases a numerical method is required to obtain an estimate of the root or roots of the expression.

Newton's method is an iterative technique that can produce good estimates of solutions to such equations. The method is employed by rewriting the equation in the form $f(x) = 0$, then successively manipulating guesses for x until the function evaluates to a value close enough to zero for the modeler to accept.

Figure 43 is a graph of some function whose intercept with the $x-$axis is unknown. The goal of Newton's method is to find this intersection (root) from a realistic first guess. Suppose the first guess is x_1, shown on the figure as the right-most specific value of x. The value of the function at this location is $f(x_1)$. Because x_1 is supposed to be a root the difference from the value zero represents an error in the estimate. Newton's method simply provides a recipe for corrections to this error.

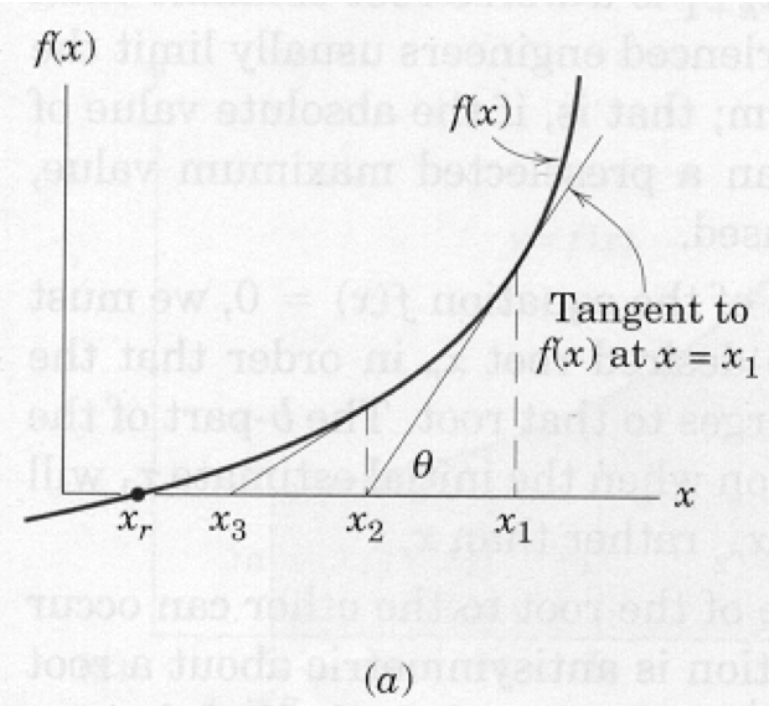

Figure 43. Graph of Arbitrary Function..

Provided x_1 is not near a minimum or maximum (slope of the function is not zero) then a better estimate of the root can be obtained by extending a tangent line from $x_1, f(x_1)$ to the x-axis. The intersection of this line with the axis represents a better estimate of the root.

This new estimate is x_2. A formula for x_2 can be derived from the geometry of the triangle x_2,$f(x_1)$,x_1. Recall from calculus that the tangent to a function at a

particular point is the first derivative of the function. Therefore, from the geometry of the triangle and the definition of tangent we can write,

$$tan(\theta) = \left.\frac{df}{dx}\right|_{x_1} = \frac{f(x_1)}{x_1 - x_2} \tag{13}$$

Solving the equation for x_2 results in a formula that expresses x_2 in terms of the first guess plus a correction term.

$$x_2 = x_1 - \frac{f(x_1)}{\frac{df}{dx}|_{x_1}} \tag{14}$$

The second term on the right hand side is the correction term to the estimate on the right hand side. Once x_2 is calculated we can repeat the formula substituting x_2 for x_1 and x_3 for x_2 in the formula. Repeated application usually leads to one of three outcomes:

1. a root;
2. divergence to $\pm\infty$; or
3. cycling.

These three outcomes are discussed below in various subsections along with some remedies.

The generalized formula is

$$x_{k+1} = x_k - \frac{f(x_k)}{\frac{df}{dx}|_{x_k}} \tag{15}$$

If the derivative is evaluated using analytical derivatives the method is called Newton's method, if approximations to the derivative are used, it is called a quasi-Newton method.

3.10.1 Newton's Method — Using analytical derivatives

This subsection presents an example in **R** of implementing Newton's method with analytical derivatives. The algorithm itself is:

1. Write the function in proper form, and code it into a computer.
2. Write the derivative in proper form and code it into a computer.
3. Make an initial guess of the solution (0 and 1 are always convenient guesses).
4. Evaluate the function, evaluate the derivative, calculate their ratio.
5. Subtract the ratio from the current guess and save the result as the update.

6. Test for stopping:
   (a) Did the update stay the same value? Yes, then stop, probably have a solution.
   (b) Is the function nearly zero? Yes, then stop we probably have a solution.
   (c) Have we tried too many updates? Yes, then stop the process is probably cycling, stop.
7. If stopping is indicated proceed to next step, otherwise proceed back to step 4.
8. Stopping indicated, report last update as the result (or report failure to find solution), and related information about the status of the numerical method.

The following example illustrates these step as well as a **R** implementation of Newton's method.

Suppose we wish to find a root (value of x) that satisfies Equation 16.

$$f(x) = e^x - 10cos(x) - 100 \tag{16}$$

Then we will need to code it into a script. Here is a code fragment that will work:

Listing 18. R code fragment for the function calculation.

```
# Define Function Here
func <- function(x)
{
  func <- exp(x)-10*cos(x)-100;
  return(func);
}
```

The next step is to code the derivative. In this case, Equation 17 is the derivative of Equation 16.

$$\frac{df}{dx}|(x) = e^x + 10\sin(x) \tag{17}$$

A code fragment to compute the value of the derivative at any value of x that will work is:

Listing 19. R code fragment for the derivative calculation.

```
# Define Derivative Here
dfdx <- function(x)
{
  dfdx <- exp(x) + 10*sin(x);
  return(dfdx);
}
```

Next we will need script to read in an initial guess, and ask us how many trials we will use to try to find a solution, as well as how close to zero we should be before we declare victory.

Listing 20. R code fragment for reading input data from the programmer.

```
# Read some values from the console
message('Enter an initial guess for X for Newton method :  ')
xnow <- as.numeric(readline())
message('Enter iteration maximum :  ')
HowMany <- as.numeric(readline())
message('Enter a tolerance value for stopping (e.g. 1e-06) :  ')
HowSmall <- as.numeric(readline())
## There are several other ways to make these reads!  The scan() function would probably
    also work.
```

The use of `HowSmall`; is called a zero tolerance. We will use the same numerical value for two tolerance tests. Also notice how we are using error traps to force numeric input. Probably overkill for this example, but we already wrote the code in an earlier chapter, so might as well use the code. Professional codes do a lot of error checking before launching into the actual processing — especially of the processing part is time consuming, its worth the time to check for obvious errors before running far a few hours then at some point failing because of an input value error that was predictable.

Now back to the tolerance tests. The first test is to determine if the update has changed or not. If it has not, we may not have a correct answer, but there is no point continuing because the update is unlikely to move further. The test is something like

IF $|x_{k+1} - x_k| <$ Tol. THEN Exit and Report Results

The second test is if the function value is close to zero. The structure of the test is similar, just an different argument. The second test is something like

IF $|f(x_{k+1})| <$ Tol. THEN Exit and Report Results

One can see from the nature of the two tests that a programmer might want to make the tolerance values different. This modification is left as a reader exercise.

Checking for maximum iterations is relatively easy, we just include code that checks for normal exit the loop.[11]

Now we simply connect the three fragments, and we have a working **R** script that implements Newton's method for Equation 16. Listing 21 is the entire code module that implements the method, makes the various tests, and reports results. Figure 44 is a screen capture of the program run in **R**.

The example is specific to the particular function provided, but the programmer could move the two functions `func` and `dfdx` into a user specified module, and then load that module in the program to make it even more generic. The next section will use such an approach to illustrate the ability to build a generalized Newton method and only have to program the function itself.

[11] Rather than breaking from the loop.

Listing 21. R code demonstrating Newton's Method calculations.

```
# Newtons Method in R
# Define Function Here
func <- function(x)
{
  func <- exp(x)-10*cos(x)-100;
  return(func);
}
# Define Derivative Here
dfdx <- function(x)
{
  dfdx <- exp(x) + 10*sin(x);
  return(dfdx);
}
# Newton's Method Here
# Read some values from the console
message('Enter an initial guess for X for Newton method :  ')
xnow <- as.numeric(readline())
message('Enter iteration maximum :  ')
HowMany <- as.numeric(readline())
message('Enter a tolerance value for stopping (e.g. 1e-06) :  ')
HowSmall <- as.numeric(readline())
# Now start the iterations
for (i in 1:HowMany) {
  xnew <- xnow - func(xnow)/dfdx(xnow)
# test for stopping
  if (abs(xnew-xnow) < HowSmall){
    message('Update not changing')
    xnow <- xnew
    print(cbind(xnow,xnew,func(xnew)))
    break
  }
  if (abs(func(xnew) < HowSmall)) {
    message('Function value close to zero')
    xnow <- xnew
    print(cbind(xnow,xnew,func(xnew)))
    break
  }
# next iteration
xnow <- xnew
}
if (i >= HowMany){
  message('Iteration limit reached')
  print(cbind(xnow,xnew,func(xnew)))
}
```

3.10.2 Newton's Method — Using Finite-Differences to estimate derivatives

A practical difficulty in using Newton's method is determining the value of the derivative in cases where differentiation is difficult. In these cases we can replace the derivative by a difference equation and then proceed as in Newton's method.

Recall from calculus that the derivative was defined as the limit of the difference quotient:

$$\frac{df}{dx}|_x = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{18}$$

A good approximation to the derivative should be possible by using this formula with a small, but non-zero value for Δx.

$$\frac{df}{dx}|_x \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{19}$$

When one replaces the derivative with the difference formula the root finding method the resulting update formula is

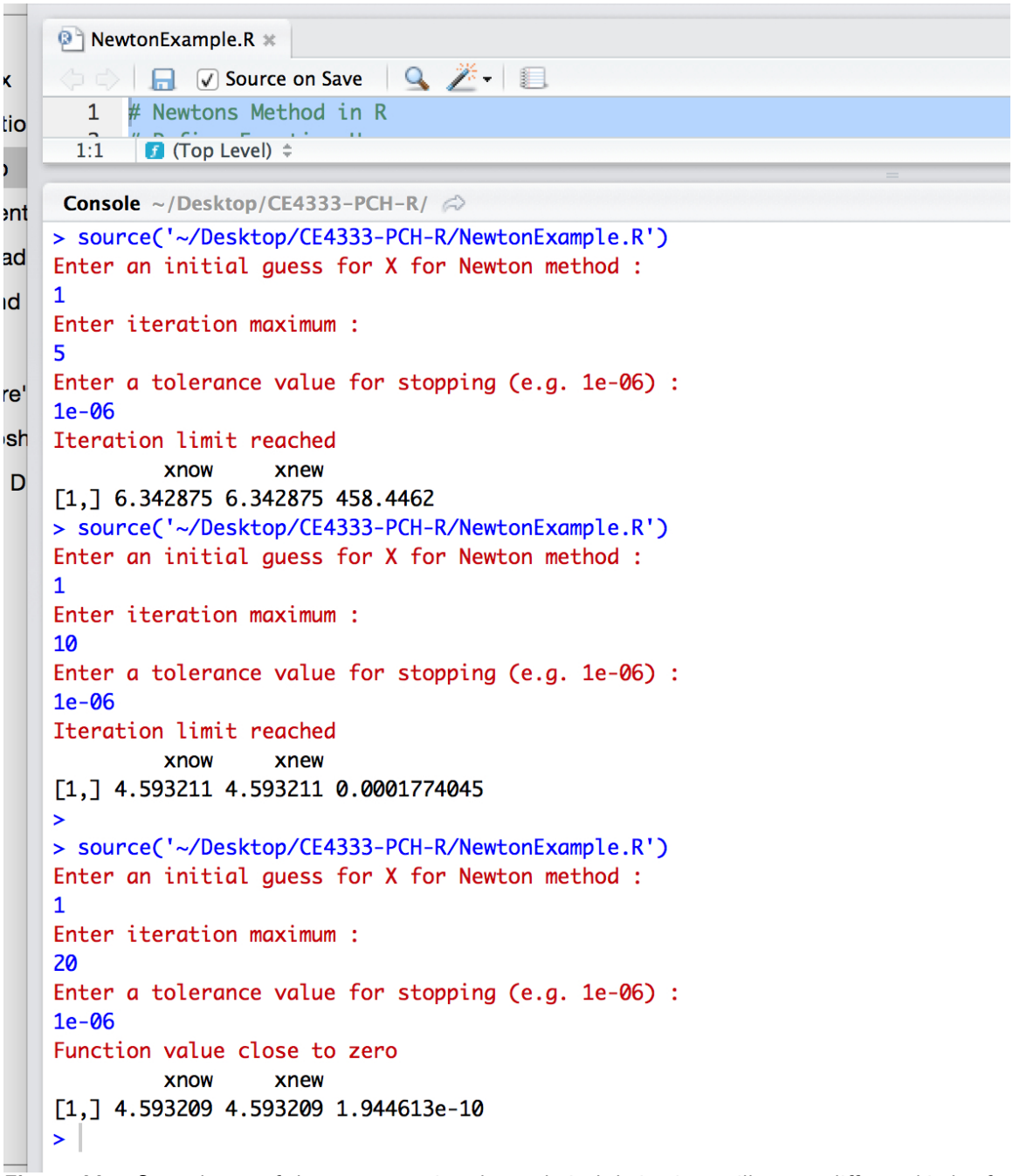

Figure 44. Several runs of the program using the analytical derivative to illustrate different kinds of responses..

$$x_{k+1} = x_k - \frac{f(x_k)\Delta x}{f(x_k + \Delta x) - f(x_k)} \tag{20}$$

This root-finding method is called a quasi-Newton method.

Listing 22 is the code fragment that we change by commenting out the analytical

derivative and replacing it with a first-order finite difference approximation of the derivative. The numerical value $1e-06$ is called the step size (Δx) and should be an input value (rather than built-in to the code as shown here) like the tolerance test values, and be passed to the function as another argument.

Listing 22. R code demonstrating Newton's Method calculations.

```
# Define Derivative Here
dfdx <- function(x)
{
#  dfdx <- exp(x) + 10*sin(x);
    dfdx <- (func(x + 1e-06) - func(x) )/ (1e-06);
# func must already exist before first call!
  return(dfdx);
}
```

Starting with the last example lets modify the analytical version of the code by inserting the above fragment in place of the analytical derivative. Listing 23 is the listing with the modification in place. Notice we have only changed a single line, and not have a more flexible tool. The next modification (left as an exercise) is to detach the creation of the function from the main algorithm, then we would have a general purpose Quasi-Newton's method.

Listing 23. R code demonstrating Newton's Method calculations using finite-difference approximation for the derivative.

```
# Newtons Method in R
# Define Function Here
func <- function(x)
{
  func <- exp(x)-10*cos(x)-100;
  return(func);
}
# Define Derivative Here
dfdx <- function(x)
{
#  dfdx <- exp(x) + 10*sin(x);
  dfdx <- (func(x + 1.0e-06) - func(x))/(1.0e-06)
  return(dfdx);
}
# Newton's Method Here
# Read some values from the console
message('Enter an initial guess for X for Newton method :  ')
xnow <- as.numeric(readline())
message('Enter iteration maximum :  ')
HowMany <- as.numeric(readline())
message('Enter a tolerance value for stopping (e.g. 1e-06) :  ')
HowSmall <- as.numeric(readline())
# Now start the iterations
for (i in 1:HowMany) {
  xnew <- xnow - func(xnow)/dfdx(xnow)
# test for stopping
  if (abs(xnew-xnow) < HowSmall){
    message('Update not changing')
    xnow <- xnew
    print(cbind(xnow,xnew,func(xnew)))
    break
  }
  if (abs(func(xnew) < HowSmall)) {
    message('Function value close to zero')
    xnow <- xnew
    print(cbind(xnow,xnew,func(xnew)))
    break
  }
# next iteration
xnow <- xnew
}
if (i >= HowMany){
  message('Iteration limit reached')
  print(cbind(xnow,xnew,func(xnew)))
}
```

Listing 23 is the main code. Notice how the function definitions are changed, in particular `dfdx`.

Figure 45 is a screen capture of the program run after the code modification above.

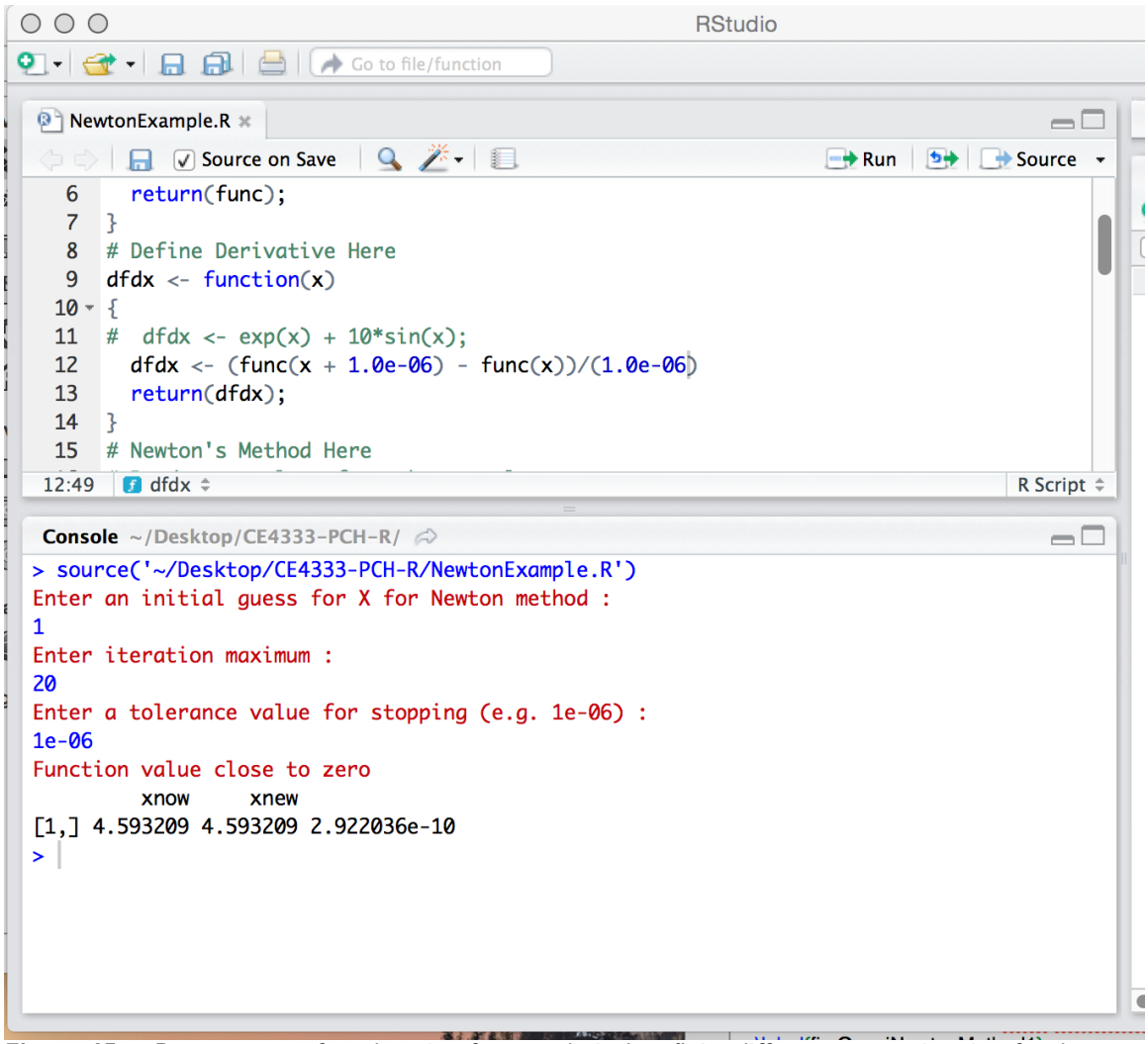

Figure 45. Program run after changing from analytical to finite-difference approximation for the derivative..

The advantage of the approximate derivative is that we don't have to do the calculus — just code in the function.

The obvious advantage of modular coding is to protect the parts of the code that are static, and just modify the function definitions. We can keep a working example around in case we break something and use that to find what we broke.

3.10.3 Method Fails

The three subsections below describe the ways that the method routinely fails, along with some suggestions for remedy. Generally we should plot the function before trying

to find a root, but sometimes the root finding is a component of a more complex program and we just want it to work. In that situation, the programmer would build in many more tests that the three above to try to force a result before giving up.

3.10.4 Multiple Roots

Figure 46 illustrates the behavior in the presence of multiple roots. When there are multiple roots the method will converge on the root that that is defined by the initial guess.[12] The initial estimate must be close enough to the desired root to converge to the root.[13] Another challenge is what happens if the initial guess is at the divide

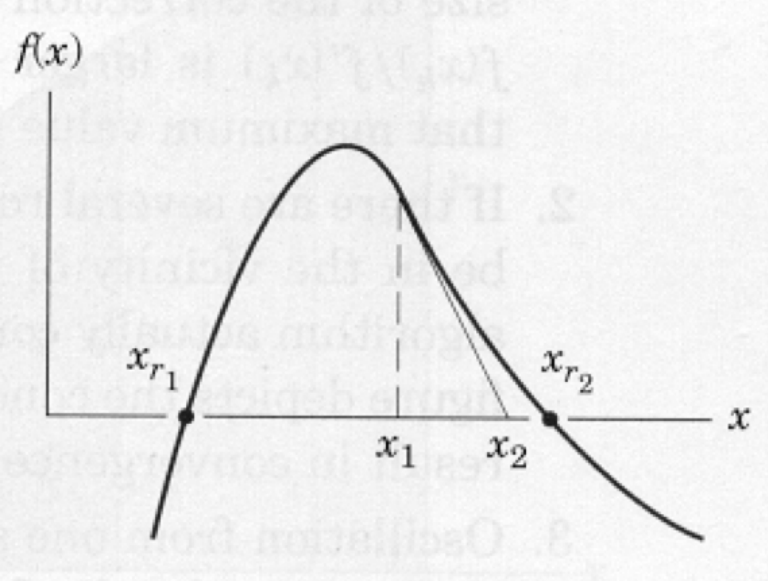

Figure 46. Multiple roots..

(the peak of the function in Figure 46); in such cases we may actually get a divergent solution because the slope of the function at that peak is nearly zero.

3.10.5 Cycling

Cycling can occur when the root is close to an inflection point of the function. Usual practice is to again limit the step size to prevent such behavior. Figure 47 is an illustration of cycling. A good remedy for cycling is to first detect the cycling, then provide a small "shove" to the guess. Examination of root finding codes often reveals a pseudo-random number generator within the code that will provide this shove when cycling is detected.

[12]This behavior is called "sensitive dependence on initial conditions".
[13]Ironically, we need a good idea of the answer before we start the method.

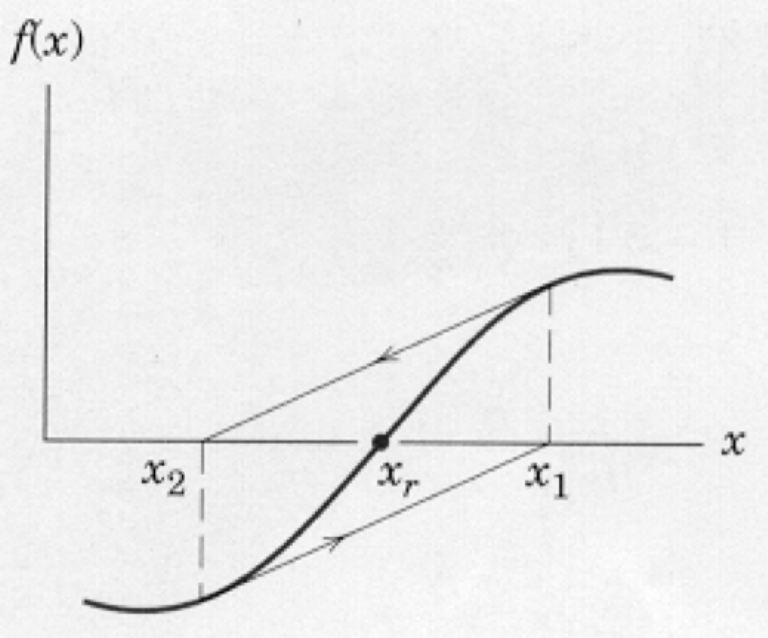

Figure 47. Estimates cycling around a root..

3.10.6 Near-zero derivatives

The derivative in Equation 86 must not be zero, otherwise the guess corresponds to a maximum or minimum of the function and the tangent line will never intersect the x-axis. The derivative must not be too close to zero, otherwise the slope will be so small as to make the correction too large to produce a meaningful update. Usual practice is to limit the size of the correction term to some maximum and to use this maximum value whenever the formula prescribes a larger step. Divergence to $\pm\infty$ is usually explained by near-zero derivatives at the sign change. The bi-section method is a little more robust in this respect.

3.11 Related Concepts

A couple of other root finding methods are worth mentioning because they can sometimes serve as a fallback when Newton's method fails. Two robust methods are bisection and false-positioning.

3.12 Exercises

1. Build a Newton's Method program (or use mine) and make the program request a tolerance value for "how close to zero" is the function, and "how small is the change in update values." Build your code using the modular approach (two files). Test your code using the same example in the notes.

2. Now modify the main and the function module to use approximate derivatives (the finite-difference formulation) and require the user supply a step size. Test the code using the same example in the notes.

   For each of the exercises above, prepare documentation similar to the notes where you describe the salient points of your program.

3. Now use your program to find roots for the following equations:

   (a) $\exp(x) - 3x^2 = 0$

   (b) $\ln(x) - x + 2 = 0$

   (c) $\tan(x) - x - 1 = 0$

   For these three equations, document your search for roots. Identify if there are bad initial guesses that cause the program to fail to find a root. Also the equations may have multiple roots. If you discover multiple roots, identify the starting values one needs to use to converge to a particular root.

4 Tank Drain Simulation using Finite-Differences

This section examines how to simulate the time to drain a tank. This particular example is a common exercise in fluid mechanics and hydraulic engineering courses to motivate the concept of control volumes and the conservation of mass. The equations of motion are usually given (as they are here) based on an assumption of negligible energy loss as the tank drains[14].

The purpose is to gain some practice with numerical modeling techniques before trying more complex methods in hydraulics.

4.1 Problem Description

Consider the tank depicted in Figure 48. The water depth in the tank at any instant is $z(t)$. The tank cross-section area is A_{tank} and the outlet cross section area is A_{out}. The product of A_{tank} and $z(t)$ is the volume in storage in the tank at some instant, and the outflow from the tank is $Q(t)$.

As a first model consider the computation of

1. Time to drain the tank from a known initial depth.
2. Storage in the tank at some given time.
3. Depth in tank at some given time.
4. Discharge rate at some instant in time.
5. How long to drain the tank to $\frac{1}{2}$ or $\frac{1}{4}$ full.

All these questions are reasonable kinds of questions that could arise in some engineering design situation or (more likely) in an operations scenario. In this example the tank has constant cross section, but it is not a far stretch to imagine that the tank could represent a reservoir of variable geometry. A practical reason to ask such questions arises in storm-water detention pond design (as well as reactor design in wastewater when dynamic flows are considered) — the time to drain impacts the remaining capacity in a detention pond, the remaining capacity is what provides some measure of control for back-to-back storms. If the tank drains too slowly, there will not be enough reserve for a back-to-back storm, whereas if the tank drains too fast it is hydraulically irrelevant.

While this problem seems detached from an open channel flow course, it [the problem] is not. Each node in the open channel equations where flow depth is computed is essentially such a tank. The tank area is not constant (instead the area is related to depth and reach distance), but like this tank the inflows and outflows are, in-part, governed by the depth already in the tank.

[14] Bernoulli's equation.

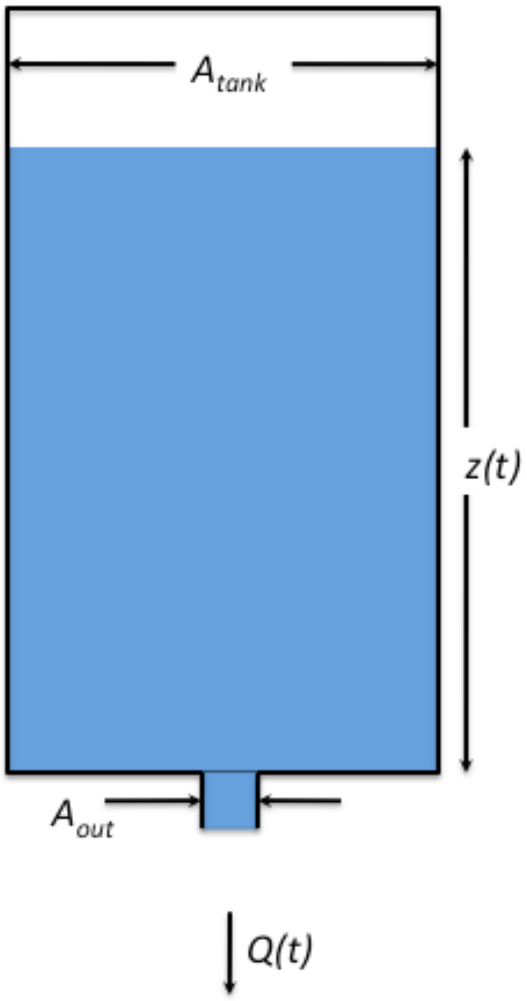

Figure 48. Storage tank with drain on bottom..

4.2 Computational Approach

This example will be handled first analytically (because it is simple to do so), then by a simple finite-difference scheme implemented in **R**

4.3 Problem Analysis — Development of the Analytical Solution

To write an equation or set of equations for this problem, the relationship of tank depth, areas, and discharges must be specified.

If we write Bernoulli's equation along a streamline in the tank from the free surface

to the outlet we would arrive at something like

$$\frac{p_{fs}}{\gamma} + \frac{V_{fs}^2}{2g} + z_{fs} = \frac{p_{out}}{\gamma} + \frac{V_{out}^2}{2g} + z_{out} \tag{21}$$

If we invoke the following reasonable assumptions:

1. $p_{fs} = 0$ Free surface at zero gage pressure.
2. $p_{out} = 0$ Pressure in jet is ≈ 0.
3. $V_{fs} << V_{out}$ Free surface velocity is small relative to the outlet velocity.
4. $z_{out} = 0$ the outlet is the datum.

The resulting relationship between depth, and discharge is

$$Q(t) = A_{out} \times \sqrt{2gz(t)} \tag{22}$$

Now we have an equation of motion. A mass balance in the tank requires that the storage decrease as the tank drains. Thus relating the change in tank depth and discharge produces the ordinary differential equation (ODE) that governs (at least in our model world) the tank.

$$A_{out} \times \sqrt{2gz(t)} = -A_{tank} \times \frac{dz}{dt} \tag{23}$$

If one collects the constants $\frac{-A_{out}}{A_{tank}}\sqrt{2g} = \alpha$ the ODE is a little simpler to examine[15]

$$\alpha[z(t)]^{\frac{1}{2}} = \frac{dz}{dt} \tag{24}$$

Equation 24 can be separated and integrated

$$\int_0^T \alpha \; dt = \int_{z(0)}^0 \frac{dz}{[z(t)]^{\frac{1}{2}}} \tag{25}$$

The solution is

$$z(t) = (z(0)^{\frac{1}{2}} - \frac{\alpha}{2}t)^2 \tag{26}$$

[15]The constants can be collected in this case because the geometry stays constant in the tank — changing geometries would have areas as functions of depth and could not be collected in this fashion.

4.3.1 Application of Analytical Solution in R

This section presents the analytical solution coded in Excel, **R**, and `FORTRAN`.

Analytical solutions are usually straightforward to represent in environments like **R**. In this example, a single line of code builds the relationship and a couple more lines for a plot.

```
> depth<-function(alpha,time,initialDepth){(sqrt(initialDepth)-0.5*alpha*time)^2}
> tt<-seq(1,1000)
> plot(tt,depth(alpha,tt,5),xlab="Time (minutes)",ylab="Depth (meters)")
```

The actual "plot" is left as an exercise.

4.4 Problem Analysis — Development of the Finite-Difference Approximation

The finite difference approximation follows the same principles up to Equation 24 then approximates the derivative term by its difference quotient for small time steps[16]

$$\alpha[z(t)]^{\frac{1}{2}} \approx \frac{\Delta z}{\Delta t} \tag{27}$$

Notice some features of 27. First the equal sign is changed to "approximately" equal; second the derivative is changed to a related rate. The remainder of the "equation" is unchanged. The trick here is to understand how to interpret the equation.

> The approximation states that the rate of change of depth with time is approximately equal to the product of the geometric and gravitational constant and the square root of the current depth.

Using this approximation, and knowing the depth in the tank to start produces an algorithm to explore the tank behavior. First expand the difference equation.

$$\alpha[z(t)]^{\frac{1}{2}} = \frac{z(t+\Delta t) - z(t)}{\Delta t} \tag{28}$$

Then rearrange to isolate values at time $t + \Delta t$,

$$z(t+\Delta t) = z(t) + \Delta t \alpha[z(t)]^{\frac{1}{2}} \tag{29}$$

All the terms on the right hand side are known and the equation[17] tells the modeler

[16] Recall the fundamental theorem of calculus where the difference quotient is taken to the limit and this limit is called the derivative — here we don't go to the limit, instead using small but finite steps.

[17] Now known as an update equation

how to approximate depth at a future time. Because the analysis assumed the difference quotient is close to the derivative, the time steps need to be kept pretty small (in fact in many problems the time step is constrained by the physics for stability and by the computation regime for precision).

The particular update here is an implementation of Euler's method[18]. There are other methods — the reader should observe that the time difference scheme could be backwards so that the discharge could be at an unknown time, or a weighted average of the two times, or a variety of other ways to approximate the derivative.

4.5 Application of the Finite-Difference Approximation in R

This section presents the same set of "solutions" using the finite-difference model. In these cases the underlying algorithm is that expressed by Equation 29.

To model the tank using **R** the modeler needs to write the necessary equation structure in the proper order. Listing 24 is a crude implementation — note how the update quotient is actually built with a test for near zero values to prevent attempted square root of a negative number. Also note how there is some flexibility to change the time step.

Listing 24. R code demonstrating time to drain calculations.

```
> # Time to Drain Model
> AreaTank<-987
> AreaPipe<-1
> alpha<-sqrt(2*9.8)*AreaPipe/AreaTank
> z<-numeric(0)  # define the depth array
> z[1]<-5.0 # initial depth
> dt<-5.0 # time step size
> # program the update difference quotient
> dzdt<-function(alpha,depth){if(depth >= 0.001)alpha*sqrt(depth) else 0}
> # update many times
> for (i in 1:500){z[i+1]=z[i]-dt*dzdt(alpha,z[i])}
> length(z) # get length for plotting
[1] 501
> time<-seq(0,500)*dt
> plot(time,z,xlab="Time (minutes)",ylab="Depth (meters)")
```

The result of the plot call is displayed in Figure 49

The plot displays a lot of zero values, the student should explore tricks to plot only the interesting portion of the behavior. The student should also explore how to plot both the analytical and numerical solution on the same graph.

What this section presented is actually quite simple. The engineer in any case will have to conceptualize the physical system into a structure amenable to mathematical representation. The tools are the conservation of mass, momentum, and energy. Then relationships between components must be established — generally time, lengths, and forces are somehow related. These relationships, whether empirical or fundamental constitute the basis to build a model to reality.

[18] One of many methods to approximate solutions to ordinary differential equations

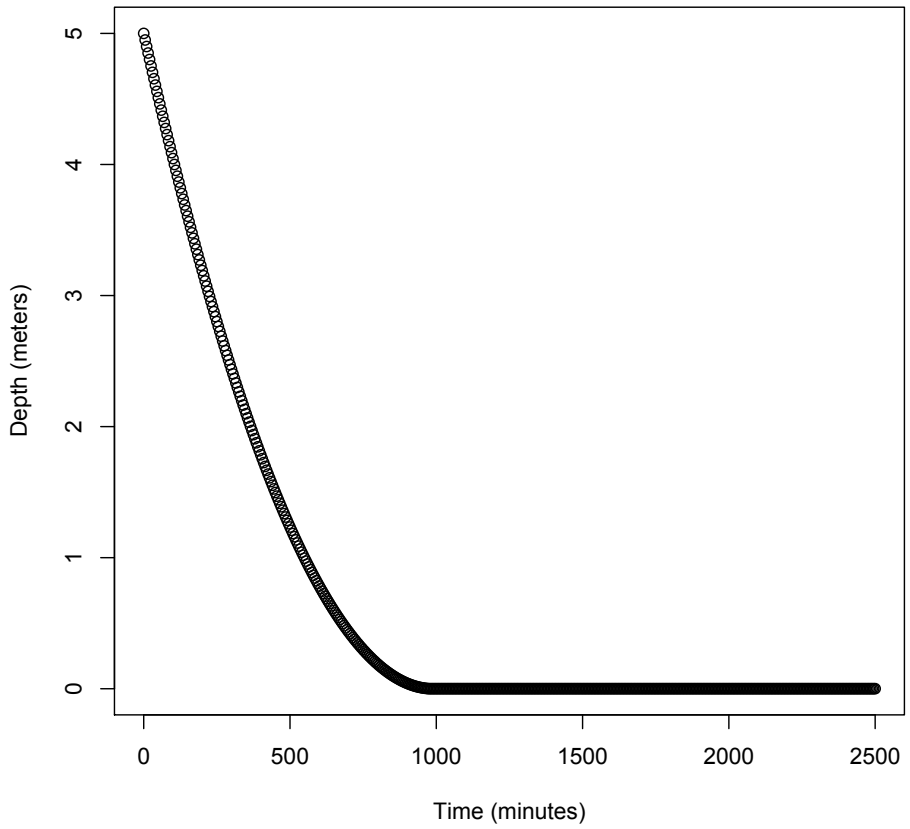

Figure 49. Plot of relationship of elapsed time and depth of water in tank..

Next the modeler has to convert these relationships into a structure that can be solved using the tools at hand: in this case **R**. Finally the application is built and used for the problem of interest. While intellectually desirable to have a fairly general tool, the engineer should never be afraid to use a purpose built tool if a general tool is unavailable. Be sure to test the tool and know its limitations.

In the example a simple hydraulics computation was presented to motivate these modeling steps. Solutions were constructed in the **R** programming environments.

4.6 Exercises

1. Adapt the **R** script and build a simulator that approximates the depth of water in a cylindrical drum lying on its side as a function of time. Figure 50 is a sketch of the drum of interest. Also desired is the volume in storage in the drum as a function of time.

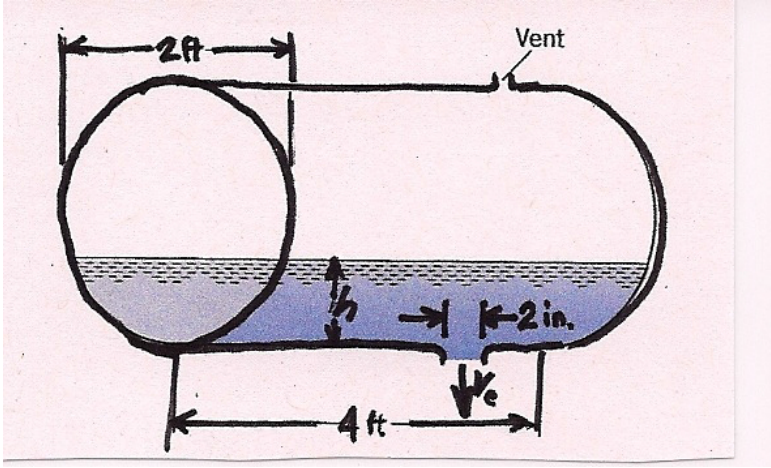

Figure 50. Sketch of drum dimensions.

The drum is drained by a 2-inch diameter short pipe at the bottom of the drum. The velocity of water in the pipe is $V_e = \sqrt{2\,g\,h}$ where g is gravitational acceleration and h is water depth in the tank above the outlet. The drum is 4-feet long and 2-feet in diameter. Simulate the time to drain from $\frac{1}{2}$ full to empty, then generalize to any starting depth (up to the tank diameter).

Produce a time versus depth and time versus storage plot for the drum using **R**.

Document your work in a short modeling report that includes conceptualization, problem analysis (development of requisite equations), coding, any testing, and finally the application of the model.

2. Adapt the **R** script and build a simulator for a trough of arbitrary dimensions as shown in Figure 51.

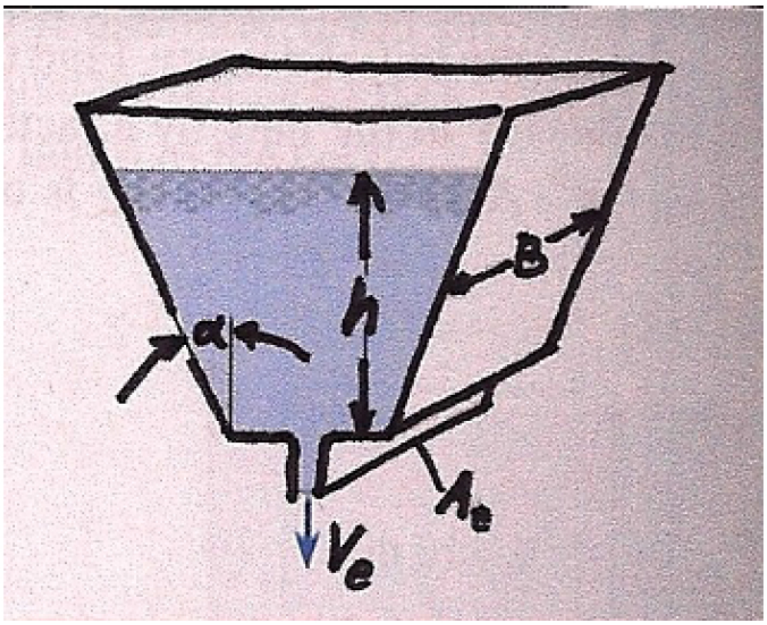

Figure 51. Sketch of trough dimensions.

The angle with the vertical of the sloping sides is α, and the distance between the parallel ends is B. The width of the trough is $W_0 + 2h\ tan\ \alpha$, where h is the distance from the trough bottom. The velocity of water issuing from the opening in the bottom of the trough is equal to $V_e = \sqrt{2\ g\ h}$. The area of the water stream at the bottom of the trough is A_e.

Produce a drainage curve (time versus depth) for the case where $h_0 = 5m$, $W_0 = 1m$, $\alpha = 30^o$, $B = 10m$, and $A_e = 1m^2$.

Document your work in a short modeling report that includes conceptualization, problem analysis (development of requisite equations), coding, any testing, and finally the application of the model.

5 Simultaneous Linear Systems of Equations

Many engineering simulations require the solution of simultaneous algebraic equations. These algebraic equation systems are either linear or non-linear in the unknown variables. Many computation schemes have been developed to solve the resulting systems, mostly depending on the structure of the systems (and the corresponding coefficient matrices).

The solution of linear (or non-linear for that matter) can be accomplished using either direct methods or iterative (successive approximation) methods. The method choice depends on:

1. The amount of computation required (size of the problem) and computer memory available.[19]
2. The accuracy of the solution required.
3. The ability to control accuracy (i.e. find accurate enough solutions) to improve overall computation speed and throughput.

Direct solution methods lead to results by means of finite and predictable operations count, but at the expense of error amplification and difficulty to deal with near-singular systems. Iterative methods can converge to exact solutions, are robust in near-singular cases, but at the expense of a non-predictable number of operations.

In this chapter we will see how to solve systems using built-in method(s) in **R** and will also see the simplest of the iterative methods, Jacobi iteration. Jacobi iteration is presented for several reasons: it is simple to program, it shows the beauty of iteration when it works, and introduces a concept called pre-conditioning. For problems in this workbook, the built in `solve(...)` is recomended; we will use Jacobi iteration later on the the aquifer flow models, because the model equation structure is quite amenable to this kind of solution method.

For really large systems of equations iterative methods probably dominate because they are quite amenable to out-of-core solution — Jacobi iteration is ideal for parallel processing in a GPU[20]

[19] In the past, the memory was indeed an issue – its less so today; a really big problem of thousands of equations and thousands of variables might indeed be too big for any single computer array and would require out-of-core solver techniques, which I suspect are a slowly dying art.

[20] Graphics Processing Unit — Nearly all our laptops have GPU; either an Intel, NVIDIA, or AMD. These are intended for rendering graphics, but can be directly accessed with the proper software tools and can perform floating point operations really quickly. For example on my laptop I have an NVIDIA GeForce GT750M which I can program using a CUDA toolkit. If I had a really large system to solve, I would try Jacobi iteration, make each equation a thread, the solution guess a thread, and the update a thread. Its relatively easy to multiply, add, and divide threads, so one could compute the update directly from parallel thread multiplication using the guess, then thread addition to update the guess, and repeat. GPU programming is beyond this handbook, but remember that one can trade efficiency for speed if the operations are simple vector arithmetic.

5.1 Numerical Linear Algebra – Matrix Manipulation

This section introduces use of matrices in **R** to learn how to address particular elements of a matrix – once that is understood, the remaining arithmetic is reasonably straightforward.

5.2 The Matrix — A data structure

Listing 25 is script fragment that reads in two different matrices **A** and **B**, and writes them back to the screen. While such an action alone is sort of meaningless, the code does illustrate how to read the two different files, and write back the result in a row wise fashion.

The two matrices are

$$\mathbf{A} = \begin{pmatrix} 12 & 7 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{30}$$

and

$$\mathbf{B} = \begin{pmatrix} 5 & 8 & 1 & 2 \\ 6 & 7 & 3 & 0 \\ 4 & 5 & 9 & 1 \end{pmatrix} \tag{31}$$

Now that we have a way (albeit pretty arcane) for getting matrices into our program from a file[21] we can explore some elementary matrix arithmetic operations, and then will later move on to some more sophisticated operations, ultimately culminating in solutions to systems if linear equations (and non-linear systems in the next chapter).

[21]The read from a file is a huge necessity — manually entering values will get old fast. I have written matrix generators whose purpose in life is to construct matrices and put them into files for subsequent processing — often these programs are pretty simple because of structure in a problem, at other times they rival the solution tool in complexity; once for a Linear Programming model (circa 1980's) I developed a code to write a 1200 X 1200 matrix to a file, which would be functionally impossible to enter by hand.

Listing 25. R code demonstrating reading in two matrices.

```
# R script for some matrix operations
############### READ IN DATA FROM A FILE ####################
filepath <- "~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/5-
    LinearSystems/RScripts"
filename <- "MatrixA.txt"
fileToRead <- paste(filepath,filename,sep="/") # build the user absolute filename
# Read the first file
yy <- read.table(fileToRead,header=FALSE,sep=",") # comma seperated ASCII, No header
filename <- "MatrixB.txt"   # change the filename
fileToRead <- paste(filepath,filename,sep="/") # build the user absolute filename
# Read the second file
zz <- read.table(fileToRead,header=FALSE,sep=",") # comma seperated ASCII, No header
############### Get Row and Column Counts ####################
HowManyColumnsA <- length(yy)
HowManyRowsA    <- length(yy$V1)
HowManyColumnsB <- length(zz)
HowManyRowsB    <- length(zz$V1)
###############  Build A and B Matrices  ####################
Amat <- matrix(0,nrow = HowManyRowsA, ncol = HowManyColumnsA)
Bmat <- matrix(0,nrow = HowManyRowsB, ncol = HowManyColumnsB)
for (i in 1:HowManyRowsA){
  for(j in 1:(HowManyColumnsA)){
    Amat[i,j] <- yy[i,j]
  }
}
rm(yy) # deallocate zz and just work with matrix and vectors
for (i in 1:HowManyRowsB){
  for(j in 1:(HowManyColumnsB)){
    Bmat[i,j] <- zz[i,j]
  }
}
rm(zz) # deallocate zz and just work with matrix and vectors
############# Echo Input ###################################
print(Amat)
print(Bmat)
```

5.3 Matrix Arithmetic

Analysis of many problems in engineering result in systems of simultaneous equations. We typically represent systems of equations with a matrix. For example the two-equation system,

$$\begin{aligned} 2x_1 \quad &+ \quad 3x_2 \;=\; 8 \\ 4x_1 \quad &- \quad 3x_2 = \quad -2 \end{aligned} \tag{32}$$

Could be represented by set of vectors and matrices[22]

$$\mathbf{A} = \begin{pmatrix} 2 & 3 \\ 4 & -3 \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 8 \\ -2 \end{pmatrix} \tag{33}$$

and the linear system then written as

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{34}$$

[22]Usually called "vector-matrix" form. Additionally, a vector is really just a matrix with column rank = 1 (a single column matrix).

So the "algebra" is considerably simplified, at least for writing things, however we now have to be able to do things like multiplication (indicated by $\cdot$) as well as the concept of addition and subtraction, and division (multiplication by an inverse). There are also several kinds of matrix multiplication – the inner product as required by the linear system, the vector (cross product), the exterior (wedge), and outer (tensor) product are a few of importance in both mathematics and engineering.

The remainder of this section will examine the more common matrix operations.

5.3.1 Matrix Definition

A matrix is a rectangular array of numbers.

$$\begin{pmatrix} 1 & 5 & 7 & 2 \\ 2 & 9 & 17 & 5 \\ 11 & 15 & 8 & 3 \end{pmatrix} \tag{35}$$

The size of a matrix is referred to in terms of the number of rows and the number of columns. The enclosing parenthesis are optional above, but become meaningful when writing multiple matrices next to each other. The above matrix is 3 by 4.

When we are discussing matrices we will often refer to specific numbers in the matrix. To refer to a specific element of a matrix we refer to the row number (i) and the column number (j). We will often call a specific element of the matrix, the $a_{i,j}$ -th element of the matrix. For example $a_{2,3}$ element in the above matrix is 17. In **R** we would refer to the element as `a_matrix[i][j]` or whatever the name of the matrix is in the program.

5.3.2 Multiply a matrix by a scalar

A scalar multiple of a matrix is simply each element of the matrix multiplied by the scalar value. Consider the matrix **A** below.

$$\mathbf{A} = \begin{pmatrix} 12 & 7 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{36}$$

If the scalar is say 2, then $2 \times \mathbf{A}$ is computed by doubling each element of **A**, as

$$2\mathbf{A} = \begin{pmatrix} 24 & 14 & 6 \\ 8 & 10 & 12 \\ 17 & 16 & 18 \end{pmatrix} \tag{37}$$

In **R** we can simply perform the arithmetic as

Listing 26. R code demonstrating scalar multiplication.

```
########################
twoA <- 2 * Amat
print(twoA)
```

Figure 52 is an example using the earlier **A** matrix and multiplying it by the scalar value of 2.0.

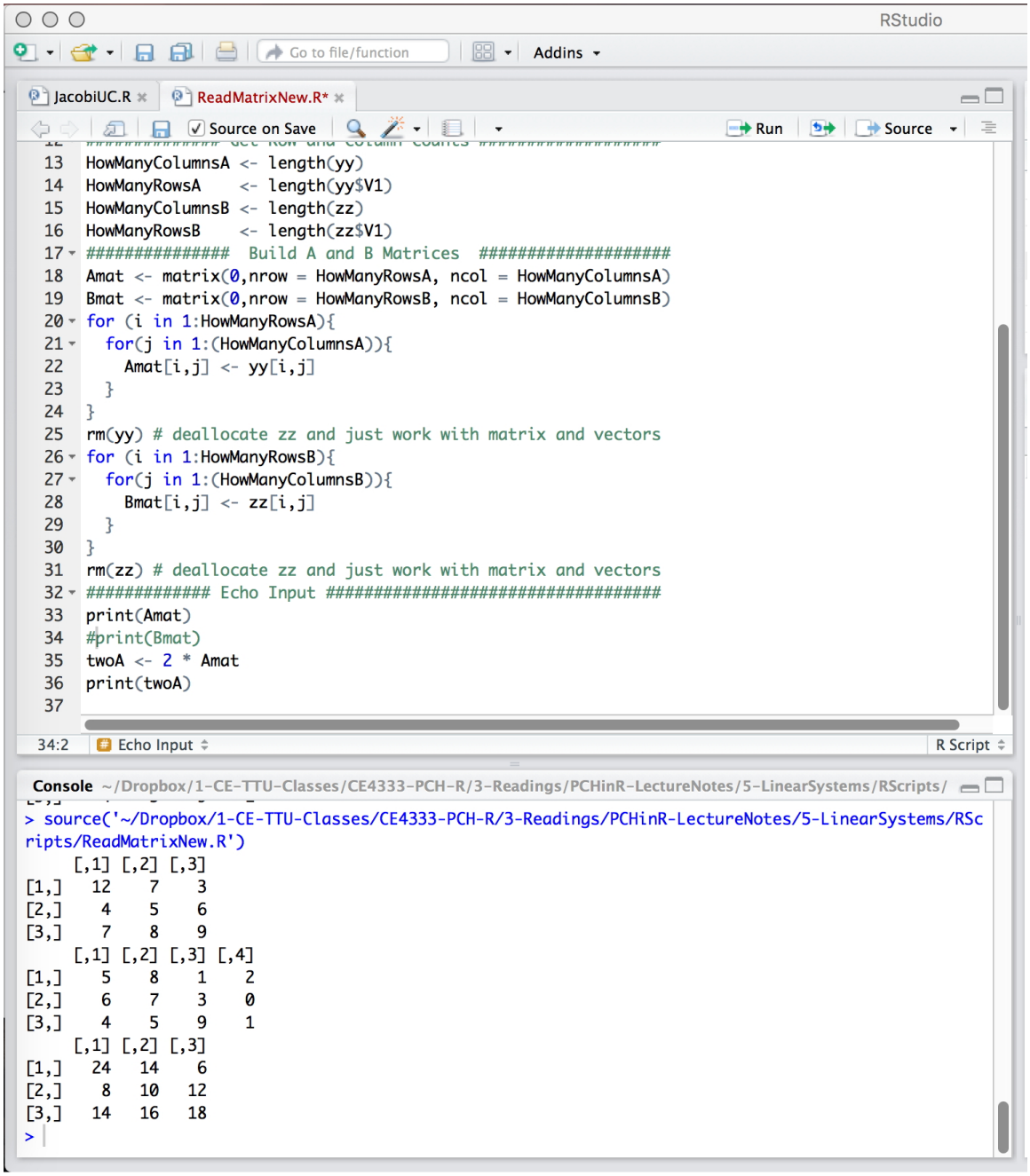

Figure 52. Multiply each element in `amatrix` by a scalar .

5.3.3 Matrix addition (and subtraction)

Matrix addition and subtraction are also element-by-element operations. In order to add or subtract two matrices they must be the same size and shape. This requirement means that they must have the same number of rows and columns. To add or subtract a matrix we simply add or subtract the corresponding elements from each matrix.

For example consider the two matrices **A** and **2A** below

$$\mathbf{A} = \begin{pmatrix} 12 & 7 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad 2\mathbf{A} = \begin{pmatrix} 24 & 14 & 6 \\ 8 & 10 & 12 \\ 17 & 16 & 18 \end{pmatrix} \tag{38}$$

For example the sum of these two matrices is the matrix named **3A**, shown below:

$$\mathbf{A} + \mathbf{2A} = \begin{pmatrix} 12+24 & 7+14 & 3+6 \\ 4+8 & 5+10 & 6+12 \\ 7+14 & 8+16 & 9+18 \end{pmatrix} = \begin{pmatrix} 36 & 21 & 9 \\ 12 & 15 & 18 \\ 21 & 24 & 27 \end{pmatrix} \tag{39}$$

Now to do the operation in **R**, we need to read in the matrices, perform the addition, and write the result. In the code example in 53 I added a third matrix to store the result – generally we don't want to clobber existing matrices, so we will use the result instead.

Subtraction is performed in a similar fashion, except the subtraction operator is used.

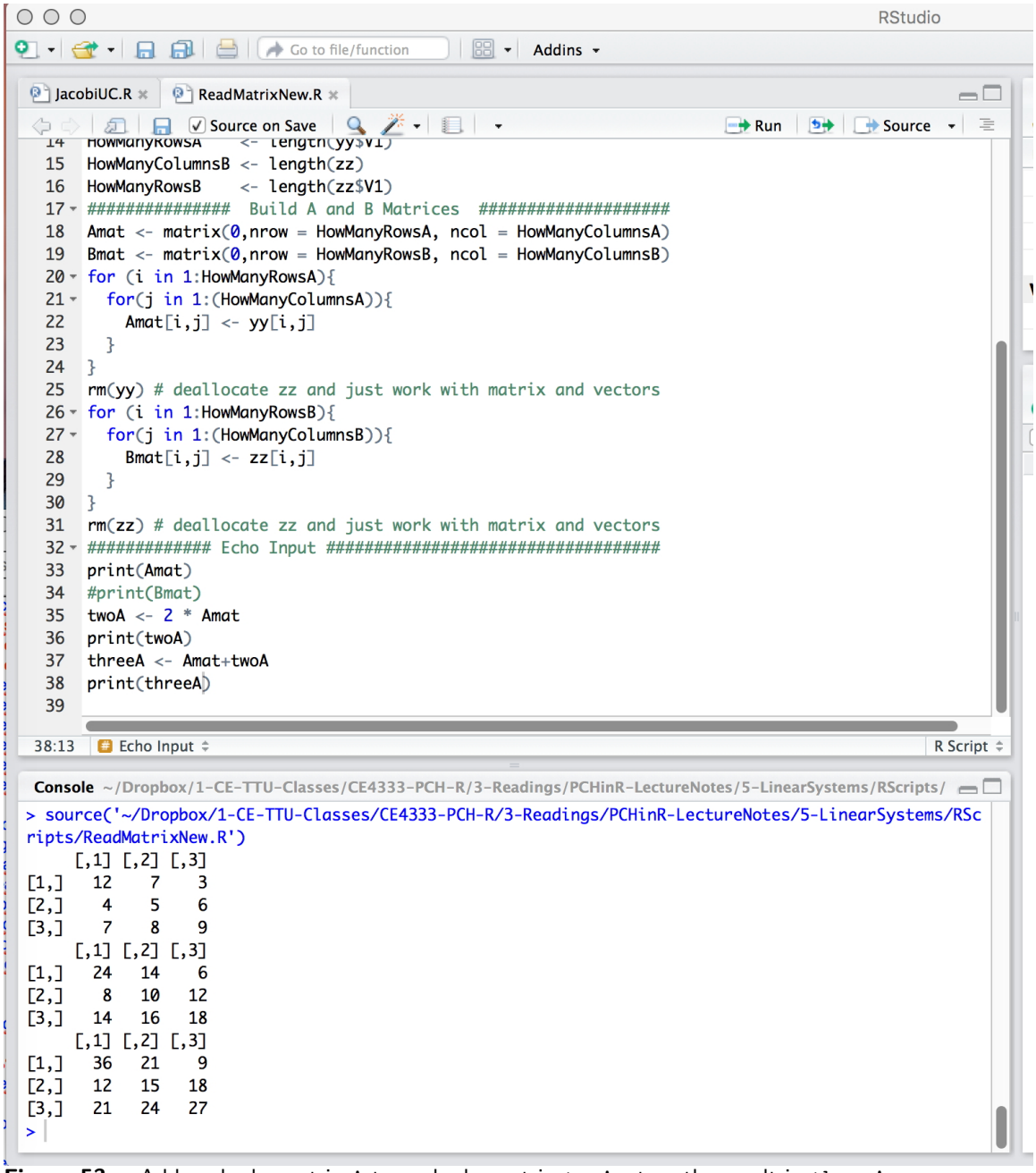

Figure 53. Add each element in `A` to each element in `twoA`, store the result in `threeA`..

5.3.4 Multiply a matrix

One kind of matrix multiplication is an inner product. Usually when matrix multiplication is mentioned without further qualification ,the implied meaning is an inner product of the matrix and a vector (or another matrix).

Matrix multiplication is more complex than addition and subtraction. If two matrices such as a matrix **A** (size l x m) and a matrix **B** (size m x n) are multiplied together, the resulting matrix **C** has a size of l x n. The order of multiplication of matrices is extremely important[23].

To obtain $\mathbf{C} = \mathbf{A}\,\mathbf{B}$, the number of columns in **A** must be the same as the number of rows in **B**. In order to carry out the matrix operations for multiplication of matrices, the i, j-th element of **C** is simply equal to the scalar (dot or inner) product of row i of **A** and column j of **B**.

Consider the example below

$$\mathbf{A} = \begin{pmatrix} 1 & 5 & 7 \\ 2 & 9 & 3 \end{pmatrix} \; \mathbf{B} = \begin{pmatrix} 3 & -2 \\ -2 & 1 \\ 1 & 1 \end{pmatrix} \tag{40}$$

First, we would evaluate if the operation is even possible, **A** has two rows and three columns. **B** has three rows and two columns. By our implied multiplication "rules" for the multiplication to be defined the first matrix must have the same number of rows as the second matrix has columns (in this case it does), and the result matrix will have the same number of rows as the first matrix, and the same number of columns as the second matrix (in this case the result will be a 2X2 matrix).

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix} \tag{41}$$

And each element of **C** is the dot product of the row vector of **A** and the column vector of **B**.

[23] Matrix multiplication is not transitive; $\mathbf{A}\,\mathbf{B} \neq \mathbf{B}\,\mathbf{A}$.

$$c_{1,1} = \begin{pmatrix} 1 & 5 & 7 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix} = \Big((1)(3) + (5)(-2) + (7)(1) \Big) = 0 \tag{42}$$

$$c_{1,2} = \begin{pmatrix} 1 & 5 & 7 \end{pmatrix} \cdot \begin{pmatrix} -2 \\ 1 \\ 1 \end{pmatrix} = \Big((1)(-2) + (5)(1) + (7)(1) \Big) = 10 \tag{43}$$

$$c_{2,1} = \begin{pmatrix} 2 & 9 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix} = \Big((2)(3) + (9)(-2) + (3)(1) \Big) = -9 \tag{44}$$

$$c_{2,2} = \begin{pmatrix} 2 & 9 & 3 \end{pmatrix} \cdot \begin{pmatrix} -2 \\ 1 \\ 1 \end{pmatrix} = \Big((2)(-2) + (9)(1) + (3)(1) \Big) = 8 \tag{45}$$

Making the substitutions results in :

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} 0 & 10 \\ -9 & 8 \end{pmatrix} \tag{46}$$

So in an algorithmic sense we will have to deal with three matrices, the two source matrices and the destination matrix. We will also have to manage element-by-element multiplication and be able to correctly store through rows and columns. In **R** this manipulation is handled for us by the matrix multiply operator `% * %`.

Figure 54 is a script that multiplies the two matrices above and prints the result.[24]

5.3.5 Identity matrix

In computational linear algebra we often need to make use of a special matrix called the "Identity Matrix". The Identity Matrix is a square matrix with all zeros except the $i, i0$-th element (diagonal) which is equal to 1:

[24] Internal to **R** the actual code for the multiplication is three nested for-loops. The outer loop counts based rows of the first matrix, the middle loop counts based on columns of the second matrix, and the inner most loop counts based on columns of the first matrix (or rows of the second matrix). In many practical cases we may actually have to manipulate at the element level — similar to how the `zz` object was put into a matrix explicitly above.

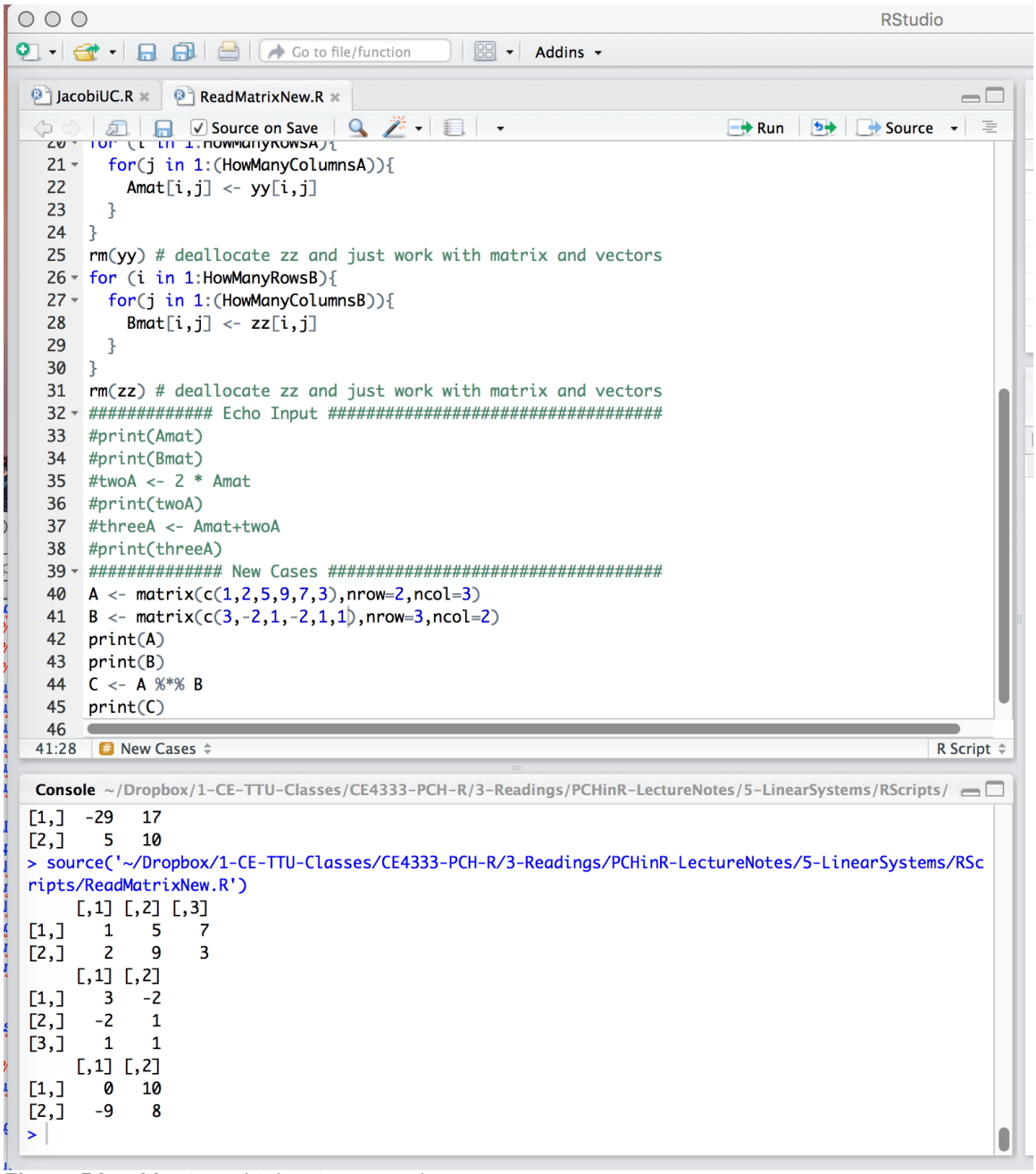

Figure 54. Matrix multiplication example.

$$\mathbf{I}_{3\times 3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{47}$$

Usually we don't bother with the size subscript i used above and just stipulate that the matrix is sized as appropriate. Multiplying any matrix by (a correctly sized) identity matrix results in no change in the matrix. $\mathbf{IA} = \mathbf{A}$

In **R** the identity matrix is easily created using `<matrix_name> <- diag(dimension)`.

5.3.6 Matrix Inverse

In many practical computational and theoretical operations we employ the concept of the inverse of a matrix. The inverse is somewhat analogous to"dividing" by the matrix. Consider our linear system

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{48}$$

If we wished to solve for $\mathbf{x}$ we would "divide" both sides of the equation by $\mathbf{A}$. Instead of division (which is essentially left undefined for matrices) we instead multiply by the inverse of the matrix[25]. The inverse of a matrix $\mathbf{A}$ is denoted by $\mathbf{A}^{-1}$ and by definition is a matrix such that when $\mathbf{A}^{-1}$ and $\mathbf{A}$ are multiplied together, the identity matrix $\mathbf{I}$ results. e.g. $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$

Lets consider the matrixes below

$$\mathbf{A} = \begin{pmatrix} 2 & 3 \\ 4 & -3 \end{pmatrix} \tag{49}$$

$$\mathbf{A}^{-1} = \begin{pmatrix} \frac{1}{6} & \frac{1}{6} \\ \\ \frac{2}{9} & -\frac{1}{9} \end{pmatrix} \tag{50}$$

We can check that the matrices are indeed inverses of each other using **R** and matrix multiplication — it should return an identity matrix.

Figure 55 is our multiplication script modified where $\mathbf{A} = \mathbf{A}$ and $\mathbf{B} = \mathbf{A}^{-1}$ perform the multiplication and then report the result. The result is the identity matrix regardless of the order of operation.[26]

Now that we have some background on what an inverse is, it would be nice to know how to find them — that is a remarkably challenging problem. Here we examine a classical algorithm for finding an inverse if we really need to — computationally we only invert if necessary, there are other ways to "divide" that are faster.

5.3.7 Gauss-Jordan method of finding $\mathbf{A}^{-1}$

There are a number of methods that can be used to find the inverse of a matrix using elementary row operations. An elementary row operation is any one of the three operations listed below:

[25] The matrix inverse is the multiplicative inverse of the matrix – we are defining the equivalent of a division operation, just calling it something else. This issue will be huge later on in our workbook, especially when we are dealing with non-linear systems

[26] Why do you think this is so, when above we stated that multiplication is intransitive?

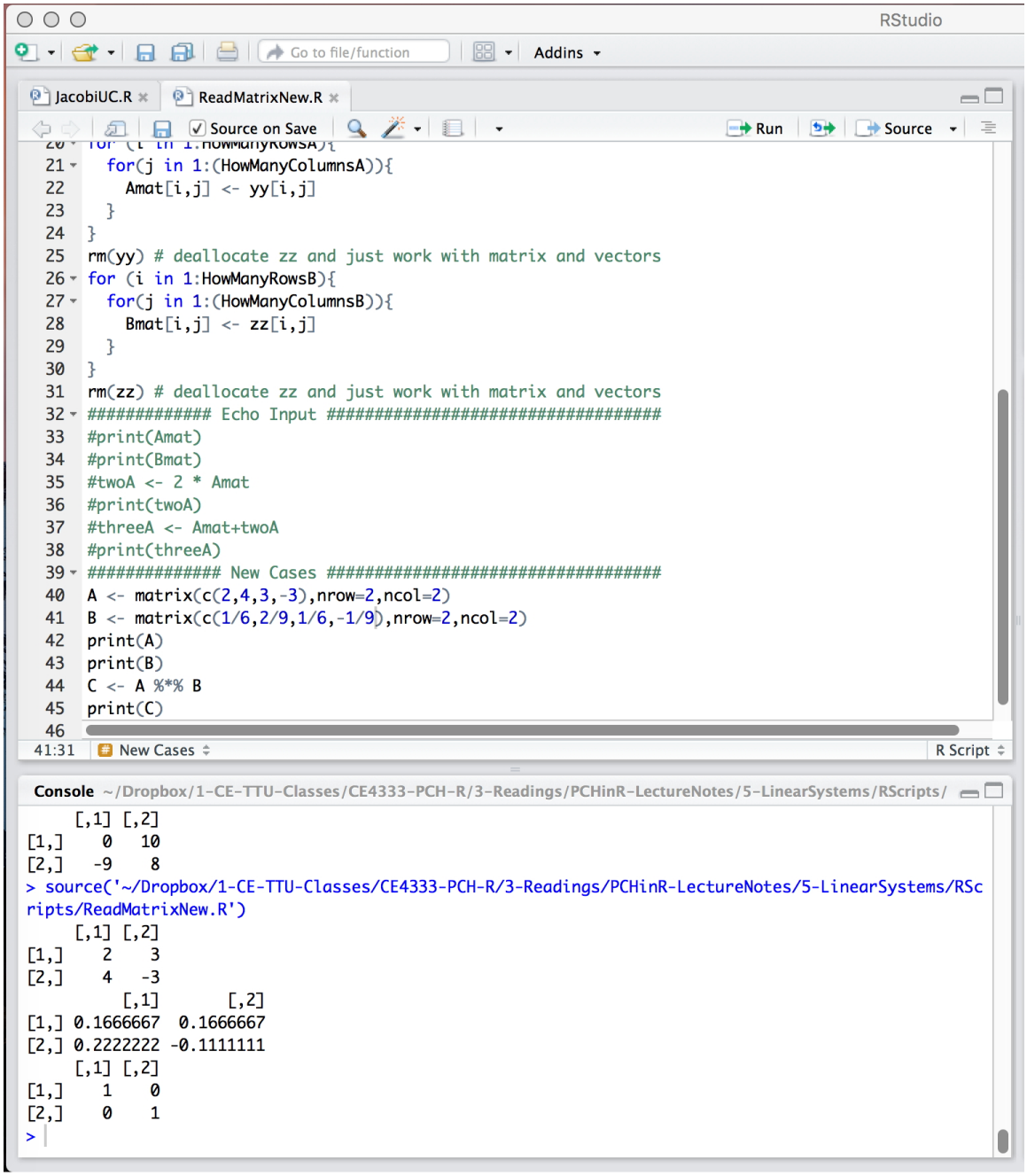

Figure 55. Matrix multiplication used to check an inverse..

1. Multiply or divide an entire row by a constant.
2. Add or subtract a multiple of one row to/from another.
3. Exchange the position of any 2 rows.

The Gauss-Jordan method of inverting a matrix can be divided into 4 main steps. In order to find the inverse we will be working with the original matrix, augmented with the identity matrix – this new matrix is called the augmented matrix (because

no-one has tried to think of a cooler name yet).

$$\mathbf{A}|\mathbf{I} = \begin{pmatrix} 2 & 3 & | & 1 & 0 \\ 4 & -3 & | & 0 & 1 \end{pmatrix} \tag{51}$$

We will perform elementary row operations based on the left matrix to convert it to an identity matrix – we perform the same operations on the right matrix and the result when we are done is the inverse of the original matrix.

So here goes – in the theory here, we also get to do infinite-precision arithmetic, no rounding/truncation errors.

1. Divide row one by the $a_{1,1}$ value to force a 1 in the $a_{1,1}$ position. This is elementary row operation 1 in our list above.

$$\mathbf{A}|\mathbf{I} = \begin{pmatrix} 2/2 & 3/2 & | & 1/2 & 0 \\ 4 & -3 & | & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 3/2 & | & 1/2 & 0 \\ 4 & -3 & | & 0 & 1 \end{pmatrix} \tag{52}$$

2. For all rows below the first row, replace row_j with $row_j - a_{j,1} * row_1$. This happens to be elementary row operation 2 in our list above.

$$\mathbf{A}|\mathbf{I} = \begin{pmatrix} 1 & 3/2 & | & 1/2 & 0 \\ 4-4(1) & -3-4(3/2) & | & 0-4(1/2) & 1-4(0) \end{pmatrix} = \begin{pmatrix} 1 & 3/2 & | & 1/2 & 0 \\ 0 & -9 & | & -2 & 1 \end{pmatrix} \tag{53}$$

3. Now multiply row_2 by $\frac{1}{a_{2,2}}$. This is again elementary row operation 1 in our list above.

$$\mathbf{A}|\mathbf{I} = \begin{pmatrix} 1 & 3/2 & | & 1/2 & 0 \\ 0 & -9/-9 & | & -2/-9 & 1/-9 \end{pmatrix} = \begin{pmatrix} 1 & 3/2 & | & 1/2 & 0 \\ 0 & 1 & | & 2/9 & -1/9 \end{pmatrix} \tag{54}$$

4. For all rows above and below this current row, replace row_j with $row_j - a_{2,2} * row_2$. This happens to again be elementary row operation 2 in our list above. What we are doing is systematically converting the left matrix into an identity matrix by multiplication of constants and addition to eliminate off-diagonal values and force 1 on the diagonal.

$$\mathbf{A}|\mathbf{I} = \tag{55}$$

$$\begin{pmatrix} 1 & 3/2-(3/2)(1) & | & 1/2-(3/2)(2/9) & 0-(3/2)(-1/9) \\ 0 & 1 & | & 2/9 & -1/9 \end{pmatrix} = \tag{56}$$

$$\begin{pmatrix} 1 & 0 & | & 1/6 & 1/6 \\ 0 & 1 & | & 2/9 & -1/9 \end{pmatrix} \tag{57}$$

5. As far as this example is concerned we are done and have found the inverse. With more than a 2X2 system there will be many operations moving up and down the matrix to eliminate the off-diagonal terms.

So the next logical step is to build an algorithm to perform these operations for us.

In **R** inversion is simply performed using the `solve(...)` function where the only argument passed to the function is the matrix.[27]

Figure 56 is a screen capture of using `solve(...)` to find the inverse of $\mathbf{A}$. The result is identical to the input matrix $\mathbf{A}^{-1}$ above. While we now have the ability to solve linear systems by rearrangement into

$$\mathbf{x} = \mathbf{A^{-1}} \cdot \mathbf{b} \tag{58}$$

this is generally not a good approach (we are solving n linear systems to obtain the inverse, instead of only the one we seek!).

Instead to solve a linear system, we would supply the coefficient matrix $\mathbf{A}$ and the right hand side $\mathbf{b}$, and then supply these two matrices to the solve routine (e.g. `x <- solve(A,b)`).

[27] If we have to write code ourselves, its not terribly hard, but is lengthy and consequently error-prone. Sometimes we have no choice, but in this workbook, we will use the built-in tool as much as possible. **R** does not use Gaussian reduction unless we tell it to do so, it implements a factorization called LU (or Cholesky) decomposition, then computes the inverse by repeated solution of a linear system with the right hand side being selected from one of the identify matrix columns (as was done above).

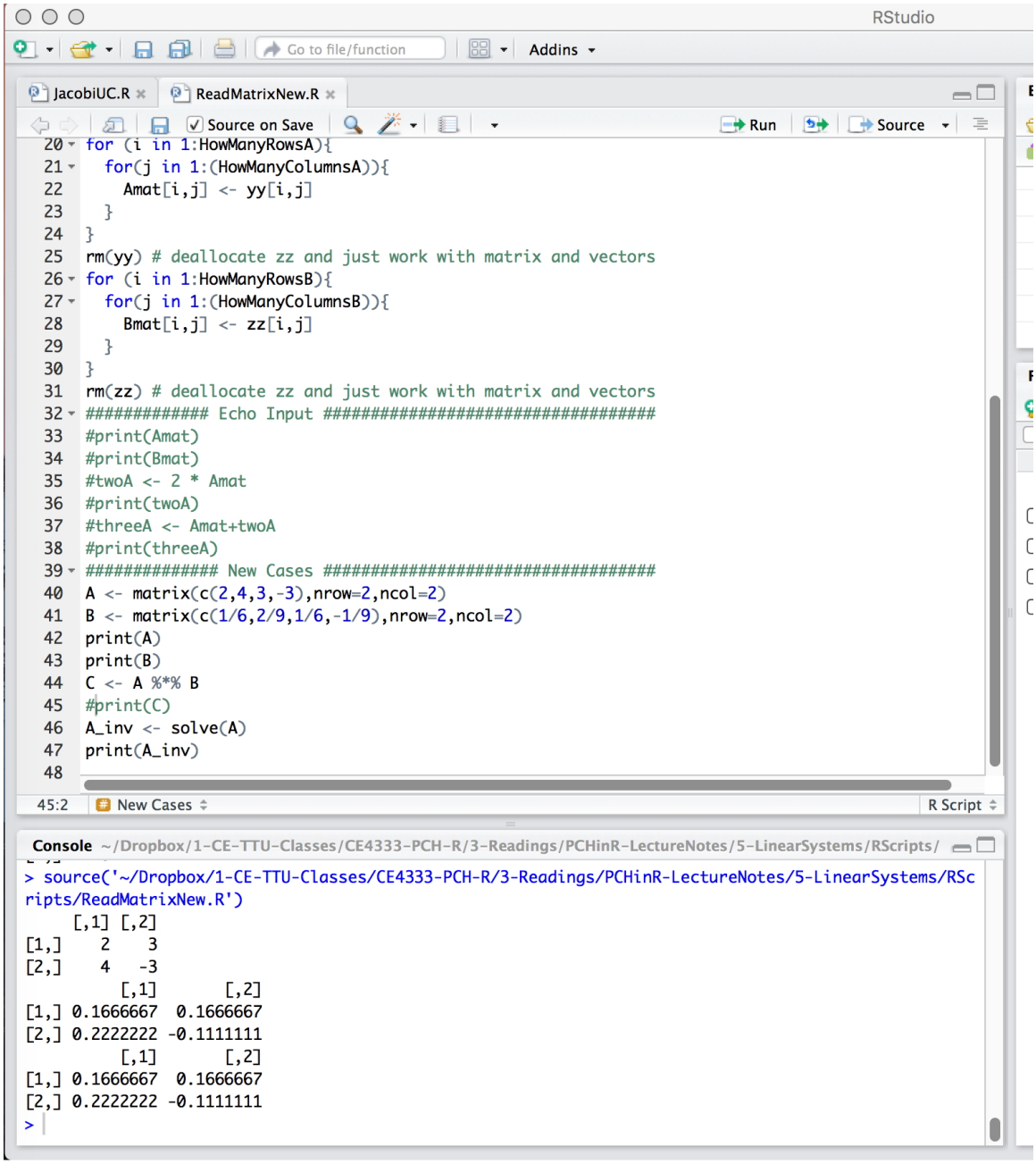

Figure 56. The matrix inversion script showing results of a run and various input and output..

5.4 Exercises

1. Develop a script to read and multiply two matrices.

   (a) Apply the script to find $\mathbf{Ax}$ where.

$$\mathbf{A} = \begin{pmatrix} 4.0 & 1.5 & 0.7 & 1.2 & 0.5 \\ 1.0 & 6.0 & 0.9 & 1.4 & 0.7 \\ 0.5 & 1.0 & 3.9 & 3.2 & 0.9 \\ 0.2 & 2.0 & 0.2 & 7.5 & 1.9 \\ 1.7 & 0.9 & 1.2 & 2.3 & 4.9 \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} 0.595194878133 \\ 0.507932173989 \\ 0.831708392507 \\ 0.630365599089 \\ 1.03737526565 \end{pmatrix} \tag{59}$$

   (b) Use your matrix multiplication program to demonstrate that the inversion example (the 5X5 system above) does indeed produce an inverse matrix.

   (c) Use the matrix multiplication program again to demonstrate that $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ where

$$\mathbf{A} = \begin{pmatrix} 4.0 & 1.5 & 0.7 & 1.2 & 0.5 \\ 1.0 & 6.0 & 0.9 & 1.4 & 0.7 \\ 0.5 & 1.0 & 3.9 & 3.2 & 0.9 \\ 0.2 & 2.0 & 0.2 & 7.5 & 1.9 \\ 1.7 & 0.9 & 1.2 & 2.3 & 4.9 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 5.0 \\ 6.0 \\ 7.0 \\ 8.0 \\ 9.0 \end{pmatrix} \tag{60}$$

2. Develop a script to solve the following linear system of equations

$$\begin{array}{rcrcrcl} 2x_1 & - & 2x_2 & + 3x_3 & = & 10 \\ 2x_1 & + & 1x_2 & - 2x_3 & = & -2 \\ 4x_1 & - & 1x_2 & - 3x_3 & = & 0 \end{array} \tag{61}$$

3. Develop a script to solve the following linear system of equations[28]

$$\begin{array}{rrrrrrcr} 0.866x_1 & 0x_2 & -0.5x_3 & 0x_4 & 0x_5 & 0x_6 & = & 0 \\ 0.5x_1 & 0x_2 & 0.866x_3 & 0x_4 & 0x_5 & 0x_6 & = & -1000 \\ -0.866x_1 & -1x_2 & 0x_3 & -1x_4 & 0x_5 & 0x_6 & = & 0 \\ -0.5x_1 & 0x_2 & 0x_3 & 0x_4 & -1x_5 & 0x_6 & = & 0 \\ -0.5x_1 & 0x_2 & 0x_3 & x_4 & x_5 & x_6 & = & 0 \\ 0x_1 & 0x_2 & -0.866x_3 & 0x_4 & 0x_5 & -1x_6 & = & 0 \end{array} \tag{62}$$

[28] This can be a somewhat tricky problem, if you have troubles try to interchange the rows need so that the diagonal is non-zero. If you use the matrix inverter with the system as presented you will get an error, but if you rearrange the rows so that the diagonal term is non-zero for each row, then the solver works. An improved version of the inverter would detect the singular nature and attempt to pivot (swap rows) to generate an investable matrix. If you just solve the system in **R** it will probably work.

5.4.1 Application Project: Static Truss Analysis

Figure 57 is a simply supported, statically determinate truss with pin connections (zero moment transfer connections). Find forces in each member for the loading shown.

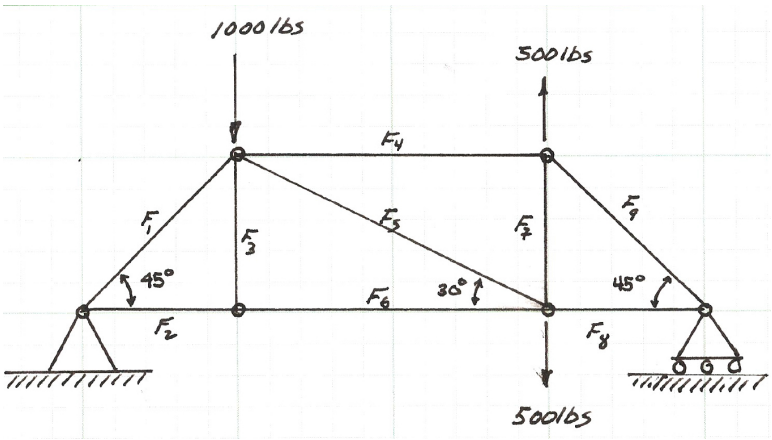

Figure 57. Sketch of simply supported, static truss.

Static Analysis Before even contemplating writing/using a program we need to build a mathematical model of the truss and assemble the system of linear equations that result from the model. So the first step is to sketch a free-body-diagram as in Figure 58 and build a node naming convention and force names.

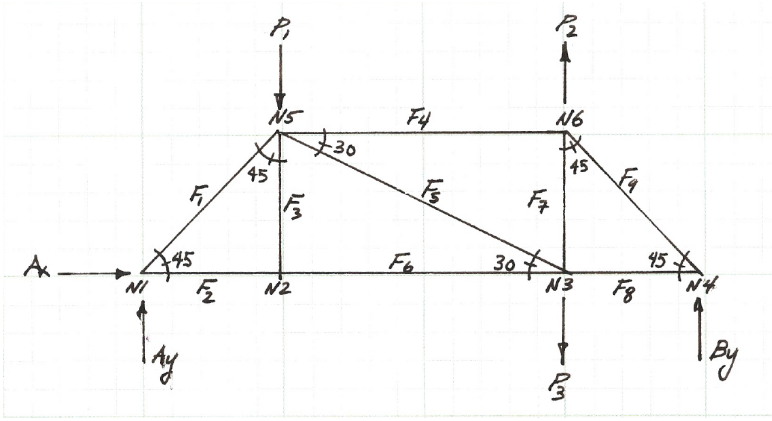

Figure 58. Static truss free-body-diagram showing forces and node naming convention..

Next we will write the force balance for each of the six nodes ($N1$-$N6$), which will produce a total of 12 equations in the 12 unknowns (the 9 member forces, and 3 reactions).

Figure 59 is the force balance for node $N1$, the two force equations (for the horizontal, x, direction and the vertical, y, direction) are listed below the figure.

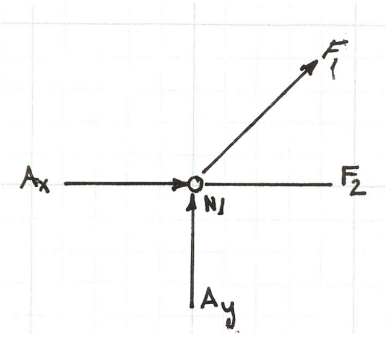

Figure 59. Force diagram at node N1.

$$\begin{array}{lllll} \sum F_x = 0 = & +F_1 cos(45) & +F_2 & +A_x & \\ \sum F_y = 0 = & +F_1 sin(45) & & & +A_y \end{array} \tag{63}$$

Equation 63 is the force balance equation pair for the node. The x component equation will later be named $N1_x$ to indicate it arises from Node 1, x component equation. A similar notation convention will also be adopted for the y component equation. There will be an equation pair for each node.

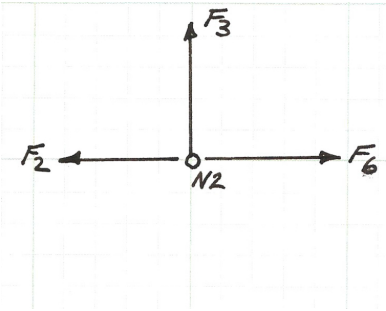

Figure 60. Force diagram at node N2.

$$\begin{array}{lllll} \sum F_x = 0 = & -F_2 & & +F_6 & \\ \sum F_y = 0 = & & +F_3 & & \end{array} \tag{64}$$

Equation 64 is the force balance equation pair for node $N2$.

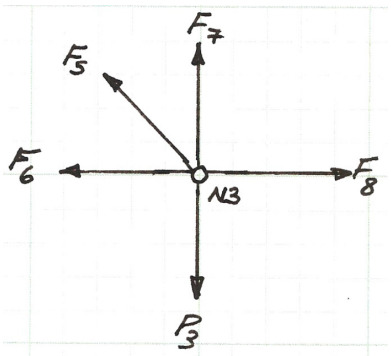

Figure 61. Force diagram at node N3.

$$\begin{array}{lccccc} \sum F_x = 0 = & & -F_5 cos(30) & -F_6 & +F_8 & \\ \sum F_y = 0 = & & F_5 sin(30) & +F_7 & & -P_3 \end{array} \tag{65}$$

Equation 65 is the force balance equation pair for node $N3$.

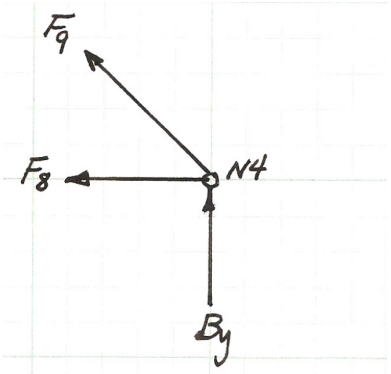

Figure 62. Force diagram at node N4.

$$\begin{array}{lccc} \sum F_x = 0 = & -F_8 & -F_9 cos(45) & \\ \sum F_y = 0 = & & F_9 sin(45) & +B_y \end{array} \tag{66}$$

Equation 66 is the force balance equation pair for node $N4$.

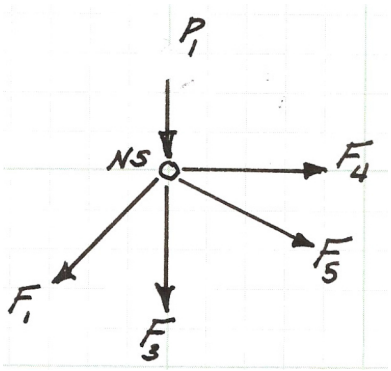

Figure 63. Force diagram at node N5.

$$\begin{array}{lllll} \sum F_x = 0 = & -F_1 cos(45) & +F_4 & +F_5 cos(30) & \\ \sum F_y = 0 = & -F_1 sin(45) \quad -F_3 & & -F_5 sin(30) & -P_1 \end{array} \tag{67}$$

Equation 67 is the force balance equation pair for node $N5$.

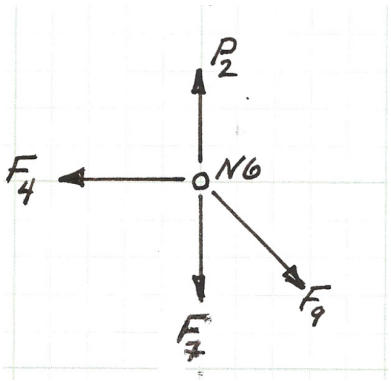

Figure 64. Force diagram at node N6.

$$\begin{array}{llll} \sum F_x = 0 = & -F_4 & F_9 sin(45) & \\ \sum F_y = 0 = & -1 & -F_9 cos(45) & P_2 \end{array} \tag{68}$$

Equation 68 is the force balance equation pair for node $N6$.

The next step is to gather the equation pairs into a system of linear equations. We will move the known loads to the right hand side and essentially construct the matrix equation $\mathbf{Ax} = \mathbf{b}$. Equation 69 is a matrix representation of the equation pairs with the forces moved to the right hand side $\mathbf{b} = RHS$.

$$\left(\begin{array}{c|cccccccccccc|c}
 & F_1 & F_2 & F_3 & F_4 & F_5 & F_6 & F_7 & F_8 & F_9 & A_x & A_y & B_y & RHS \\
\hline
N1_x & 0.707 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
N1_y & 0.707 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
N2_x & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
N2_y & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
N3_x & 0 & 0 & 0 & 0 & -0.866 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
N3_y & 0 & 0 & 0 & 0 & 0.5 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & P_3 \\
N4_x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -0.707 & 0 & 0 & 0 & 0 \\
N4_y & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.707 & 0 & 0 & 0 & 0 \\
N5_x & -0.707 & 0 & 0 & 1 & 0.866 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
N5_y & -0.707 & 0 & -1 & 0 & -0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & P_1 \\
N6_x & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0.707 & 0 & 0 & 0 & 0 \\
N6_y & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -0.707 & 0 & 0 & 0 & -P_2
\end{array}\right) \tag{69}$$

In Equation 69, the rows are labeled on the left-most column with their node-related equation name. Thus each row of the matrix corresponds to an equation derived from a node. The columns are labeled with their respective unknown force (except the last column, which represents the right-hand-side of the system of linear equations). Thus the coefficient in each column corresponds to a force in each node equation. The sign of the coefficient refers to the assumed direction the force acts. In the analysis all the members were assumed to be in tension (except for the reaction forces). If a coefficient has a value of zero in a particular row, then than force does no act at the node to which the row corresponds.

From this representation the transition to the formal vector-matrix representation is straightforward. Equation 70, Equation 71, Equation 72 are the **A** matrix, the **x** vector, and the **b** vector, respectively.

Finally, like the last problem in the exercises, the rows need to be re-ordered so the main diagonal contains non-zero values before attempting to solve the linear system. The remaining project task is to rearrange the rows (by-hand is fine, although writing code to generate an ordering would be a worthy endeavor!), then find the unknown forces using the matrix routines we have already constructed.[29]

One pretty cool feature of this analysis, is we solve for the reactions without needing to know member lengths (no moments).

[29] In a later chapter full pivoting will be presented, and with full pivoting this system can be solved without the human having to rearrange rows.

$$\mathbf{A} = \begin{pmatrix} 0.707 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0.707 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.866 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -0.707 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.707 & 0 & 0 & 0 \\ -0.707 & 0 & 0 & 1 & 0.866 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.707 & 0 & -1 & 0 & -0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0.707 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -0.707 & 0 & 0 & 0 \end{pmatrix} \tag{70}$$

$$\mathbf{x} = \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \\ F_7 \\ F_8 \\ F_9 \\ A_x \\ A_y \\ B_y \end{pmatrix} \tag{71}$$

$$\mathbf{b} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ P_3 \\ 0 \\ 0 \\ 0 \\ P_1 \\ 0 \\ -P_2 \end{pmatrix} \tag{72}$$

Using your scripts from the Exercises, the resulting linear system and determine the forces in each member.

5.5 Jacobi Iteration – An iterative method to find solutions

Iterative methods are often more rapid and economical in storage requirements than the direct methods in `solve(...)`.[30] The methods are useful (necessary) for non-linear systems of equations — we will use this feature later when we find solutions to networks of pipelines.

Lets consider a simple example:

$$\begin{array}{rrrr} 8x_1 & +1x_2 & -1x_3 = & 8 \\ 1x_1 & -7x_2 & +2x_3 = & -4 \\ 2x_1 & +1x_2 & +9x_3 = & 12 \end{array} \tag{73}$$

The solution is $x_1 = 1$, $x_2 = 1$, $x_3 = 1$. We begin the iterative scheme by refactoring each equation in terms of a single variable (there is a secret pivot step to try to make the system diagonally dominant – the example above has already been pivoted, or "pre-conditioned" for the solution method):

$$\begin{array}{rrrrr} x_1 = & 1.000 & & -0.125x_2 & 0.125x_3 \\ x_2 = & 0.571 & 0.143x_1 & & 0.286x_3 \\ x_3 = & 1.333 & -0.222x_1 & -0.111x_2 & \end{array} \tag{74}$$

Then supply an initial guess of the solution (e.g. $(0, 0, 0)$) and put these values into the right-hand side, the resulting left-hand side is an improved (hopefully) solution. Repeat the process until the solution stops changing, or goes obviously haywire.

This sequence of operation for the example above produces the results listed in Table 6.

Table 6. Jacobi Iteration Solution Sequence.

Iteration:	1-st	2-nd	3-rd	4-th	5th	6-th	7-th	8-th
x_1	0	1.000	1.095	0.995	0.993	1.002	1.001	1.000
x_2	0	0.571	1.095	1.026	0.990	0.998	1.001	1.000
x_3	0	1.333	1.048	0.969	1.000	1.004	1.001	1.000

As a practical matter, refactoring the equations can instead be accomplished by computing the inverse of each diagonal coefficient – and matrix multiplication, scalar division, and vector addition are all that is required to find a solution (if the method will actually work).

In linear algebra terms the Jacobi iteration method (without refactoring) performs the following steps:

[30] The **R** solve routine is pretty robust, if you tell it `sparse=TRUE` it has a lot of internal methods to pre-condition the problem for fast solution. But for really big systems we may wish to program our own solver — especially if these systems have some special and predictable structure.

1. Read in $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{x_{guess}}$.
2. Construct a vector from the diagonal elements of $\mathbf{A}$. This vector, $\mathbf{W}$, will have one column, and same number of rows as $\mathbf{A}$.
3. Perform matrix arithmetic to compute an error vector, $\mathbf{residual} = \mathbf{A} \cdot \mathbf{x_{guess}} - \mathbf{b}$.
4. Divide this error vector by the diagonal weights $\mathbf{update} = \mathbf{residual}/\mathbf{W}$
5. Update the solution vector $\mathbf{x_{new}} = \mathbf{x_{guess}} - \mathbf{update}$
6. Test for stopping, if not indicated, move the new solution into the guess and return to step 3.
7. If time to stop, then report result and stop.

Listing 27 implements in **R** the algorithm described above to find solutions by the Jacobi iteration method. The script does not pre-condition the linear system (so we have to do that ourselves).

Listing 27. R code demonstrating Jacobi Iteration.

```
# R script to implement Jacobi Iteration Method to
#   find solution to simultaneous linear equations
#   assumes matrix is pre-conditioned to diagional dominant
#   assumes matrix is non-singular
############## READ IN DATA FROM A FILE ####################
filepath <- "~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/5-
    LinearSystems/RScripts"
filename <- "LinearSystem000.txt"
fileToRead <- paste(filepath,filename,sep="/") # build the user absolute filename
# Here we open the connection to the file (within read.table)
# Then the read.table attempts to read the entire file into an object named zz
# Upon either fail or success, read.table closes the connection
zz <- read.table(fileToRead,header=FALSE,sep=",") # comma seperated ASCII, No header
############## Row and Column Counts ######################
HowManyColumns <- length(zz)
HowManyRows    <- length(zz$V1)
tolerance <- 1e-12 #stop when error vector is small
itermax <- 200 # maximum number of iterations
############### Build A, x, and B ##############################
Amat <- matrix(0,nrow = HowManyRows, ncol = (HowManyColumns-2) )
xguess <- numeric(0)
Bvec <- numeric(0)
Wvec <- numeric(0)
#############################################################
for (i in 1:HowManyRows){
  for(j in 1:(HowManyColumns-2)){
    Amat[i,j] <- zz[i,j]
  }
  Bvec[i] <- zz[i,HowManyColumns-1]
  xguess[i] <- zz[i,HowManyColumns]
  Wvec[i] <- Amat[i,i]
}
rm(zz) # deallocate zz and just work with matrix and vectors
#####################Implement Jacobi Iteration ##############
for(iter in 1:itermax){
Bguess <- Amat %*% xguess
residue <- Bguess - Bvec
xnew <- xguess - residue/Wvec
xguess <- xnew
testval <- t(residue) %*% residue
if (testval < tolerance) {
  message("sum squared error vector small : ",testval);
  break
}
}
if( iter == itermax) message("Method Fail")
message(" Number Iterations : ", iter)
message(" Coefficient Matrix : ")
print(cbind(Amat))
message(" Solution Vector : ")
print(cbind(xguess))
message(" Right-Hand Side Vector : ")
print(cbind(Bvec))
```

Figure 65 is a screen capture of the script in Listing 27 applied to the example problem.

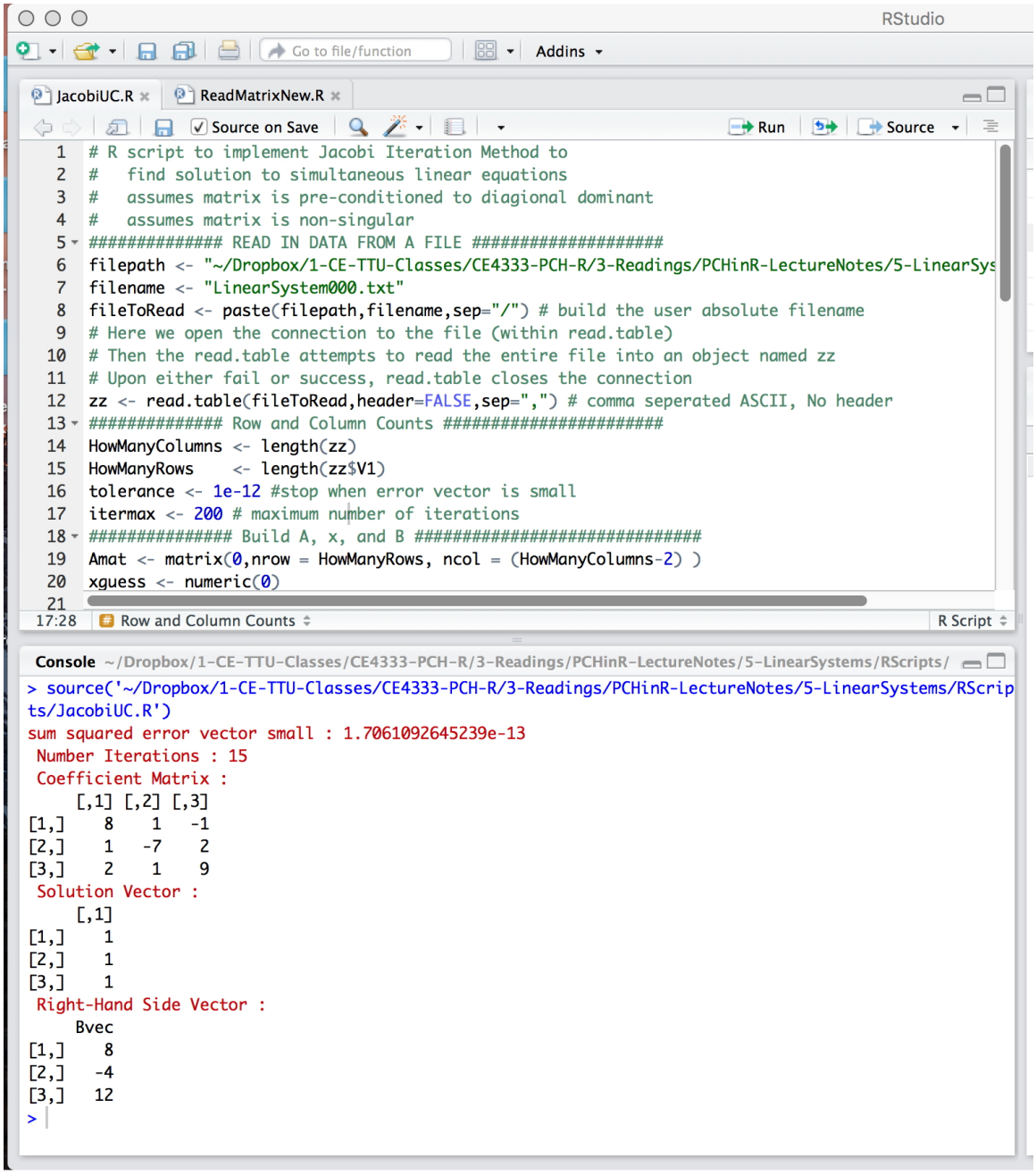

Figure 65. Jacobi Iteration applied to Example Problem.

5.6 Exercises

1. Develop a script to implement the Jacobi iteration method.
2. Apply your Jacobi script to find $\mathbf{x}$ in $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ where.

$$\mathbf{A} = \begin{pmatrix} 5 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 5 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 9 \\ 4 \\ -6 \end{pmatrix} \tag{75}$$

3. Solve the same system using `solve(...)` in **R**.
4. Apply your Jacobi script to find $\mathbf{x}$ in $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ where.

$$\mathbf{A} = \begin{pmatrix} 1.48 & 0.93 & -1.30 \\ 2.68 & 3.04 & -1.48 \\ 2.51 & 1.48 & 4.53 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 1.03 \\ -0.53 \\ 0.05 \end{pmatrix} \tag{76}$$

5. Solve the same system using `solve(...)` in **R**.
6. Apply your Jacobi script to find $\mathbf{x}$ in $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ where.

$$\mathbf{A} = \begin{pmatrix} 2.51 & 1.48 & 4.53 \\ 1.48 & 0.93 & -1.30 \\ 2.68 & 3.04 & -1.48 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 0.05 \\ 1.03 \\ -0.53 \end{pmatrix} \tag{77}$$

7. Solve the same system using `solve(...)` in **R**.
8. Problem 6 failed using Jacobi – however it is identical in a linear algebra sense to Problem 4. How should the system have been pre-conditioned before attempting the Jacobi solution?[31]

[31] An internet search on pre-conditioning would be helpful in answering this problem.

6 Simultaneous Non-Linear Systems of Equations

Non-linear systems are extensions of the linear systems cases except the systems involve products and powers of the unknown variables. Non-linear problems are often quite difficult to manage, especially when the systems are large (many rows and many variables).

The solution to non-linear systems, if non-trivial yet alone even possible, are iterative.

Within the iterative steps is a linearization component – these linear systems which are intermediate computations within the overall solution process are treated by an appropriate linear system method (direct or iterative).

In **R** it is sometimes successful to solve by the nonlinear minimization tool built-in, but neither efficient, nor particularly useful when the system gets large. On the `CRAN` there are a couple of packages devoted to non-linear systems, and these would be reasonable places to consider.

In this chapter we will illustrate an iterative technique called Quasi-Linearization, and the next chapter we will formally extend Newton's method to multi-dimensional cases.

$$\begin{aligned} x^2 &+ y^2 = 4 \\ e^x &+ y = 1 \end{aligned} \tag{78}$$

Suppose we have a solution guess x_k, y_k, which of course could be wrong, but we could linearize about that guess as

$$\mathbf{A} = \begin{pmatrix} x_k & + y_k \\ 0 & + 1 \end{pmatrix} \; \mathbf{x} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} \; \mathbf{b} = \begin{pmatrix} 4 \\ 1 - e^{x_k} \end{pmatrix} \tag{79}$$

Now the system is linear, and we can solve for $\mathbf{x_{k+1}}$ much like the Jacobi iteration of the previous chapter. If the system is convergent (not all are) then we update in the same fashion, and repeat until complete.

Listing 28 is a script that implements the quasi-linearization method. The starting vector is crucial, and the next several screen captures illustrate good starting vectors (resulting in a solution) and poor ones.

Listing 28. R code demonstrating Non-Linear by quasi-linearization.

```
# R script to solve non-linear example by quasi-linearization
Amat <- matrix(0,nrow=2,ncol=2)
Brhs <- numeric(0)
x_guess <- c(-1.9, 0.8)
maxiter <- 20
message("Initial Guess"); print(x_guess); message("Original Equations - x_guess ");
message( x_guess[1]^2 + x_guess[2]^2, " : should be 4 ")
message( exp(x_guess[1]) + x_guess[2], " : should be 1 ")
# Construct the current quasi-linear model
for (iter in 1:maxiter){
Amat[1,1] <- x_guess[1]; Amat[1,2] <- x_guess[2];
Amat[2,1] <- 0         ; Amat[2,2] <- 1;
Brhs[1] <- 4
Brhs[2] <- 1-exp(x_guess[1])
# Solve for the new guess
x_new <- solve(Amat,Brhs)
# Update
x_guess <- x_new
}
print(Amat); print(Brhs);
message("Current Guess"); print(x_new)
message("Original Equations - x_new")
message( x_new[1]^2 + x_new[2]^2, " : should be 4 ")
message( exp(x_new[1]) + x_new[2], " : should be 1 ")
```

Figure 66 is a screen capture of the algorithm started near a solution, that sort-of converges to the solution. Not really satisfying, but at least not divergent.

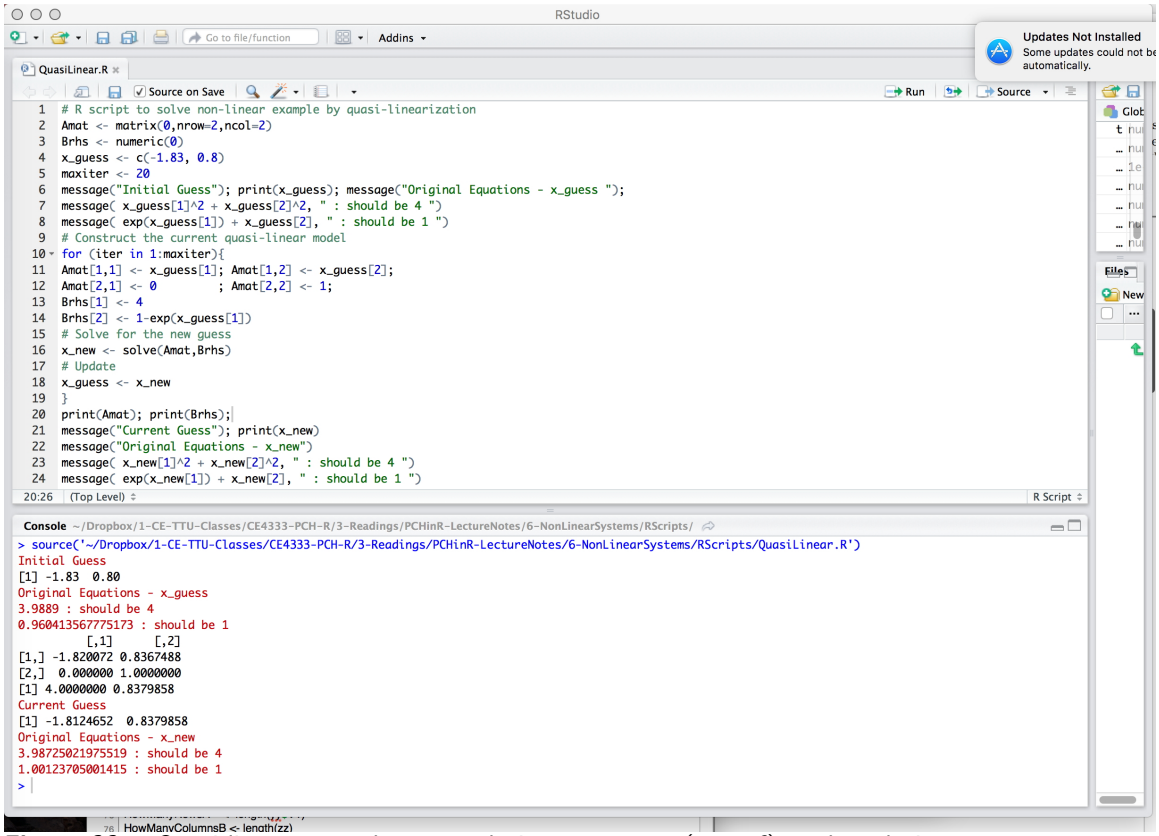

Figure 66. Quasi-linear, started near a solution, converges (sort-of) to the solution.

Figure 67 is a screen capture of the algorithm started near a solution, that fails to converge — it actually diverged.

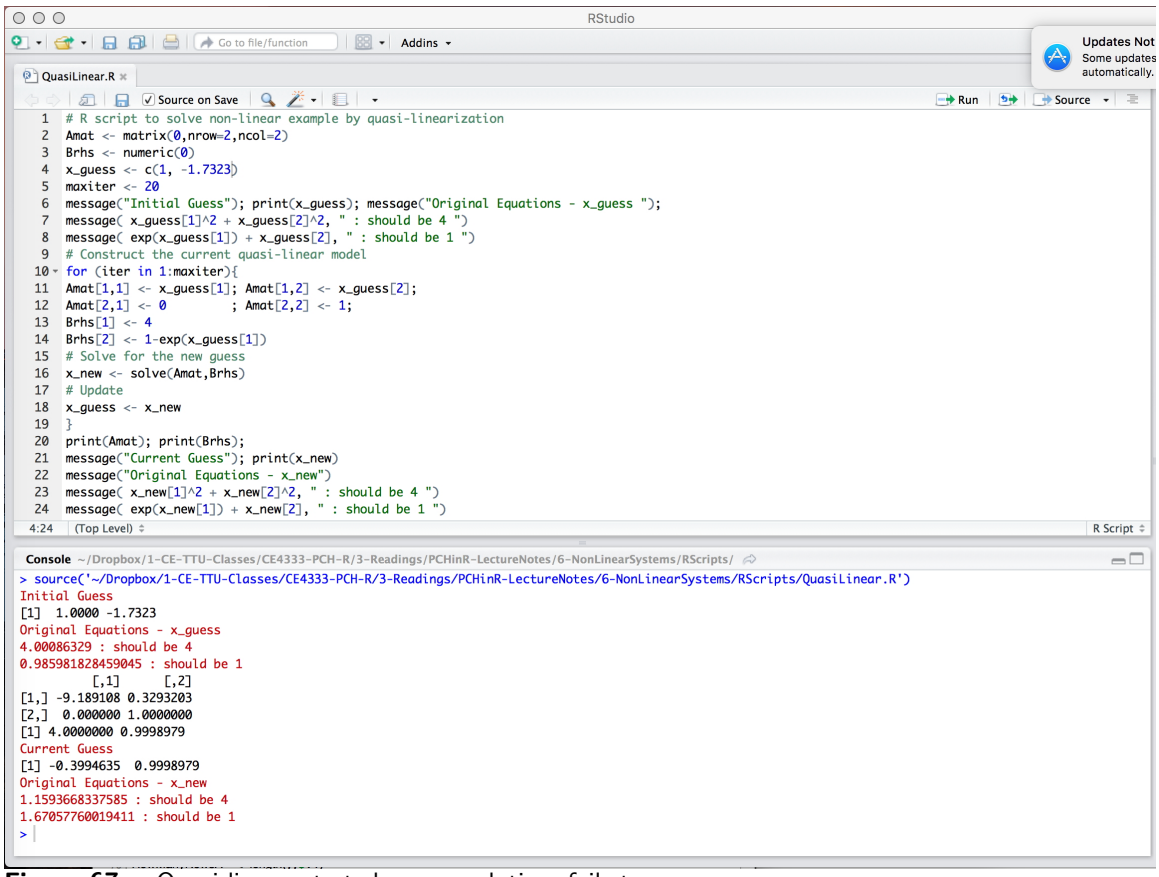

Figure 67. Quasi-linear, started near a solution, fails to converge.

What is really needed is a much more reliable algorithm. Sometimes the non-linear minimization tools can successfully be used. We will try that next.

Lets restructure our equation system a bit into

$$\mathbf{f}(\mathbf{x}) = \begin{matrix} f_1(x,y) = & x^2 & + y^2 & -4 \\ f_2(x,y) = & e^x & + y & -1 \end{matrix} \tag{80}$$

At the solution $\mathbf{x}$, the result should be $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. But if we are not at a solution, then the result will be non-zero (and represents the error) — one tool we have is a non-linear minimization tool in **R** that can minimize functions. So now we need the sum-of-squared errors, which with vectors is simply the inner product of the vector with itself:

$$\mathbf{F}(\mathbf{x}) = \mathbf{f}(\mathbf{x})^T \cdot \mathbf{f}(\mathbf{x}) \tag{81}$$

So lets rewrite the script to construct $\mathbf{f}(\mathbf{x})$ and $\mathbf{F}(\mathbf{x})$, then implement the non-linear minimizer, `nlm(...)` in **R**. Listing 29 is a listing that implements these changes. Notice the two prototype functions, the first takes vector input and returns vector output — internal (to the function) definition of a vector using `func <- numeric(0)` provides the memory space in the program.[32]

Listing 29. R code demonstrating Non-Linear by Minimization.

```
# R script for system of non-linear equations using minimization
# WARNING -- This is not recomended for large systems
# forward define the functions
####### f(x) ##################
func <- function(x_vector){
  func <- numeric(0)
  func[1] <- x_vector[1]^2 + x_vector[2]^2 - 4
  func[2] <- exp(x_vector[1]) + x_vector[2] - 1
  return(func)
}
######### F(x) ###############
bigF <- function(x_vector){
  vector <- numeric(0)
  vector <- func(x_vector)
  bigF <- t(vector) \%*\% vector
  return(bigF)
}
############################
# forward define some variables
# starting guess
x_guess <- c(1,-1.7)
result <- nlm(bigF,x_guess)
message(" Estimated bigF Value : ",result$minimum)
message(" Estimated x_vector Value : ")
print(result$estimate)
message(" Estimated func Value : ")
print(func(result$estimate))
```

Figure 68 is a screen capture of the script for the first solution to the system of equations, we have started quite close to a solution and the method converges to the correct solution. The object named `result` contains several items of which we have only accessed two. Notice how we have addressed these items using the `name$attribute` method.

Figure 69 is a screen capture of the script for the second solution to the system of equations, we have started quite close to a solution and the method converges to the correct solution.

Naturally, to be really useful we should test the method for starting values relatively far from the solution; Figure 70 is a screen capture of such testing for a few different start vectors. Observe that the solution at (-1.8,0.8) is the preferred solution in most cases unless we start very close to the second solution at (1,-1.7). This kind of preference to one solution over another is quite common in non-linear systems (sometimes these particular solutions are called attractors). The related observation is that we can find starting vectors that simply fail — this phenomenon is also quite common (sometimes called sensitive dependence on initial conditions).

[32] If you get an error message with the words `... Atomic ....`, it means that something in a function is trying to address a variable for which there is no space, or trying to address a global (external to the function) variable directly. These are pretty hard errors to debug (fix), so I have gotten into the habit of building and testing the prototype functions before I even try to get the rest of the program to run.

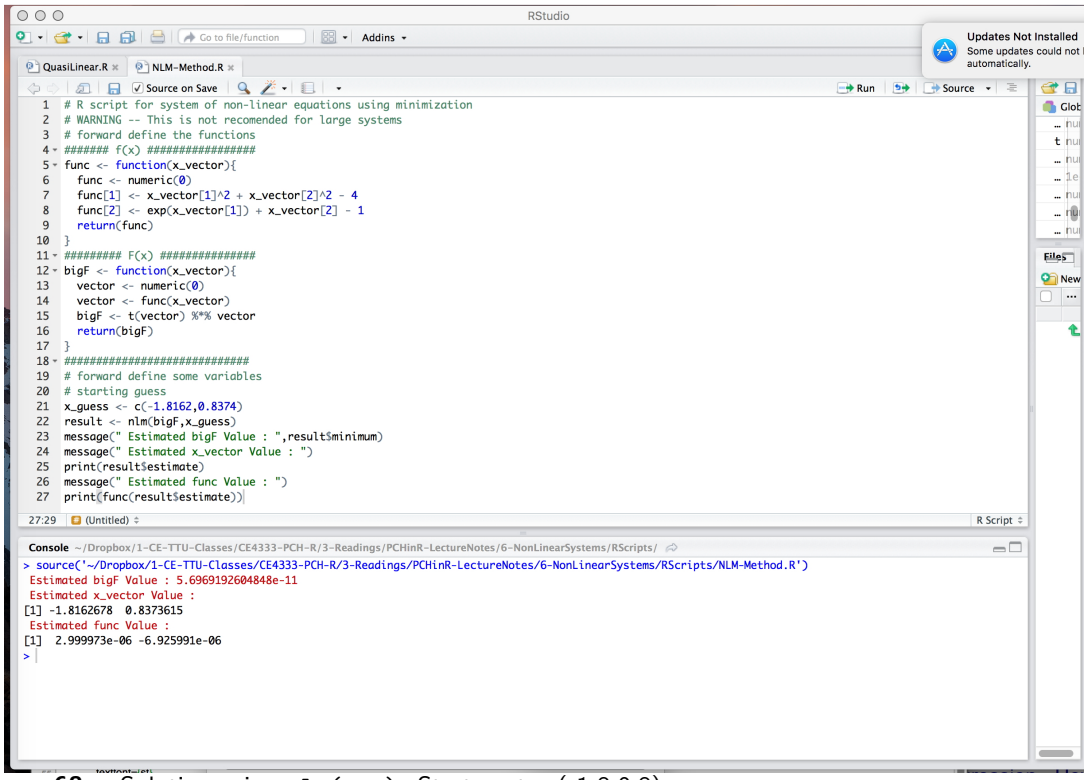

Figure 68. Solution using `nlm(...)`. Start vector (-1.8,0.8).

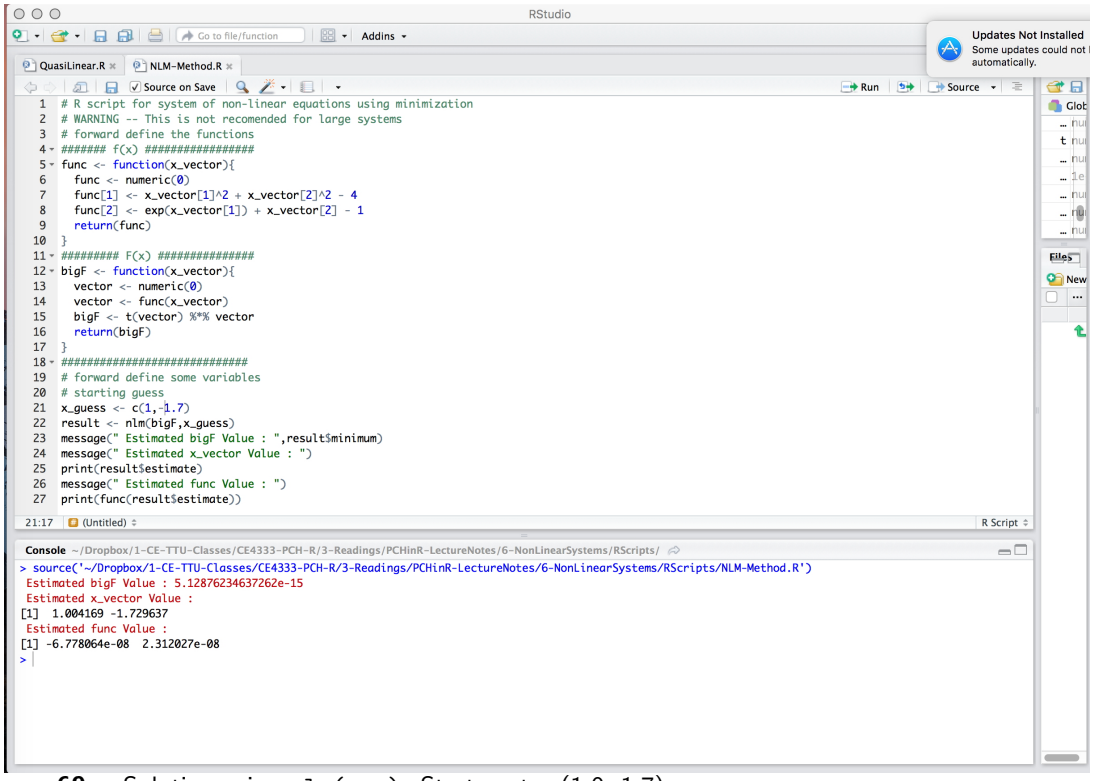

Figure 69. Solution using `nlm(...)`. Start vector (1.0,-1.7).

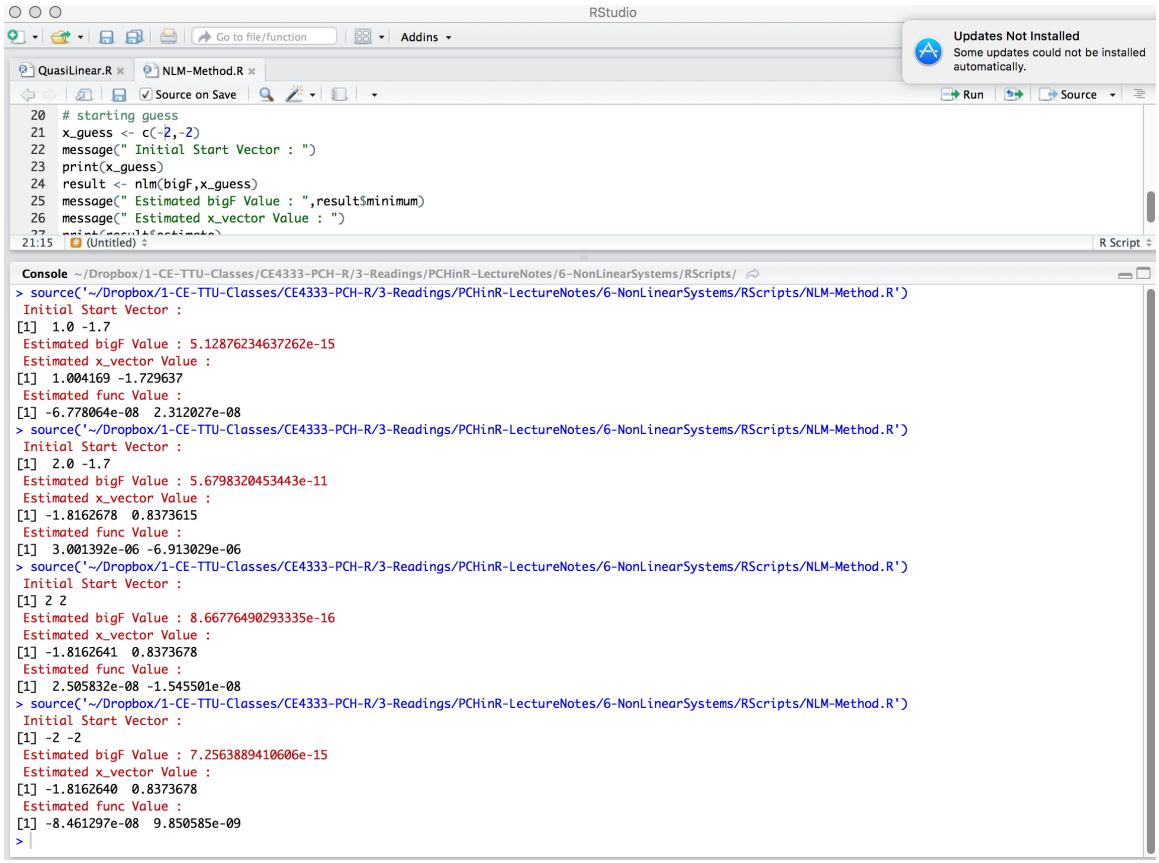

Figure 70. Solution using `nlm(...)`. Varying start vectors.

Using a non-linear minimization technique to solve systems of non-linear equations is not recommended for anything bigger than a few equations (maybe as many as 8 or 9). Quasi-linearization is a good technique — the example here is intentionally pathological. The next chapter presents a better technique than quasi-linearization that can be used for large systems (assuming they will converge at all), and it is the method that will be used for pipeline networks.

6.1 Exercises

1. Write a script that forward defines the multi-variate functions and implements the non-linear minimization technique in **R**. Implement the method and find a solution to:

$$\begin{array}{rcl} x^3 + 3y^2 & = & 21 \\ x^2 + 2y & = & -2 \end{array} \tag{82}$$

2. Write a script that forward defines the multi-variate functions and implements the non-linear minimization technique in **R**. Implement the method and find a solution to:

$$\begin{array}{rcl} x^2 + y^2 + z^2 & = & 9 \\ xyz & = & 1 \\ x + y - z^2 & = & 0 \end{array} \tag{83}$$

3. Write a script that forward defines the multi-variate functions and implements the non-linear minimization technique in **R**. Implement the method and find a solution to:

$$\begin{array}{rcl} xyz - x^2 + y^2 & = & 1.34 \\ xy - z^2 & = & 0.09 \\ e^x - e^y + z & = & 0.41 \end{array} \tag{84}$$

7 Numerical Methods – Multiple Variable Quasi-Newton Method

This chapter formally presents the Newton-Raphson method as the preferred alternative to using an optimizer routine to solve systems of non-linear equations. The method is used later in the document to solve for flows and heads in a pipeline network.

Lets return to our previous example where the function **f** is a vector-valued function of a vector argument.

$$\mathbf{f}(\mathbf{x}) = \begin{matrix} f_1 = & x^2 & + y^2 & -4 \\ f_2 = & e^x & + y & -1 \end{matrix} \tag{85}$$

Lets also recall Newtons method for scalar valued function of a single variable.

$$x_{k+1} = x_k - \frac{f(x_k)}{\frac{df}{dx}|_{x_k}} \tag{86}$$

Extending to higher dimensions, the value x become the vector $\mathbf{x}$ and the function $f()$ becomes the vector function **f()**. What remains is an analog for the first derivative in the denominator (and the concept of division of a matrix).

The analog to the first derivative is a matrix called the Jacobian which is comprised of the first derivatives of the function **f** with respect to the arguments **x**. For example for a 2-value function of 2 arguments (as our example above)

$$\frac{df}{dx}|_{x_k} => \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix} \tag{87}$$

Next recall that division is replaced by matrix multiplication with the multiplicative inverse, so the analogy continues as

$$\frac{1}{\frac{df}{dx}|_{x_k}} => \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix}^{-1} \tag{88}$$

Lets name the Jacobian $\mathbf{J(x)}$.

So the multi-variate Newton's method can be written as

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}(\mathbf{x})^{-1}|_{x_k} \cdot \mathbf{f}(\mathbf{x})|_{x_k} \tag{89}$$

In the linear systems chapter we did find a way to solve for an inverse, but its not necessary – a series of rearrangement of the system above yields a nice scheme tthat does not require inversion of a matrix.

First, move the $\mathbf{x}_k$ to the left-hand side.

$$\mathbf{x}_{k+1} - \mathbf{x}_k = -\mathbf{J}(\mathbf{x})^{-1}|_{x_k} \cdot \mathbf{f}(\mathbf{x})|_{x_k} \tag{90}$$

Next multiply both sides by the Jacobian.

$$\mathbf{J}(\mathbf{x})|_{x_k} \cdot (\mathbf{x}_{k+1} - \mathbf{x}_k) = -\mathbf{J}(\mathbf{x})|_{x_k} \cdot \mathbf{J}(\mathbf{x})^{-1}|_{x_k} \cdot \mathbf{f}(\mathbf{x})|_{x_k} \tag{91}$$

Recall a matrix multiplied by its inverse returns the identity matrix (the matrix equivalent of unity)

$$-\mathbf{J}(\mathbf{x})|_{x_k} \cdot (\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{f}(\mathbf{x})|_{x_k} \tag{92}$$

So we now have an algorithm:

1. Start with an initial guess $\mathbf{x}_k$, compute $\mathbf{f}(\mathbf{x})|_{x_k}$, and $\mathbf{J}(\mathbf{x})|_{x_k}$.
2. Test for stopping. Is $\mathbf{f}(\mathbf{x})|_{x_k}$ close to zero? If yes, exit and report results, otherwise continue.
3. Solve the linear system $\mathbf{J}(\mathbf{x})|_{x_k} \cdot (\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{f}(\mathbf{x})|_{x_k}$.
4. Test for stopping. Is $(\mathbf{x}_{k+1} - \mathbf{x}_k)$ close to zero? If yes, exit and report results, otherwise continue.
5. Compute the update $\mathbf{x}_{k+1} = \mathbf{x}_k - (\mathbf{x}_{k+1} - \mathbf{x}_k)$, then
6. Move the update into the guess vector $\mathbf{x}_k <= \mathbf{x}_{k+1}$ =and repeat step 1. Stop after too many steps.

Now to repeat the example from the previous chapter, except we will employ this algorithm.

The function (repeated)

$$\mathbf{f}(\mathbf{x}) = \begin{matrix} f_1 = & x^2 & + y^2 & -4 \\ f_2 = & e^x & + y & -1 \end{matrix} \tag{93}$$

Then the Jacobian, here we will compute it analytically because we can

$$\mathbf{J}(\mathbf{x}) => \begin{pmatrix} 2x & 2y \\ \\ e^x & 1 \end{pmatrix} \tag{94}$$

Listing 30 is a listing that implements the Newton-Raphson method with analytical derivatives.

Listing 30. R code demonstrating Newton's Method calculations.

```
# R script for system of non-linear equations using Newton-Raphson with analytical
    derivatives
# forward define the functions
######## f(x) ##########################
func <- function(x_vector){
  func <- numeric(0)
  func[1] <- x_vector[1]^2 + x_vector[2]^2 - 4
  func[2] <- exp(x_vector[1]) + x_vector[2] - 1
  return(func)
}
######## J(x) ##########################
jacob <- function(x_vector){
  jacob <- matrix(0,nrow=2,ncol=2)
  jacob[1,1] <- 2*x_vector[1]   ; jacob[1,2] <- 2*x_vector[2];
  jacob[2,1] <- exp(x_vector[1]); jacob[2,2] <- 1 ;
  return(jacob)
}
######## Solver Parameters #############
x_guess <- c(2.,-0.8)
tolerancef <- 1e-9  # stop if function gets to zero
tolerancex <- 1e-9  # stop if solution not changing
maxiter <- 20 # stop if too many iterations
x_now <- x_guess
###### Newton-Raphson Algorithm ########
for (iter in 1:maxiter){
  funcNow <- func(x_now)
  testf <- t(funcNow) %*% funcNow
  if(testf < tolerancef){
    message("f(x) is close to zero : ", testf);
    break
  }
  dx <- solve(jacob(x_now),funcNow)
  testx <- t(dx) %*% dx
  if(testx < tolerancex){
    message("solution change small : ", testx);
    break
  }
  x_now <- x_now - dx
}
########################################
if( iter == maxiter) {message("Maximum iterations -- check if solution is converging : ")}
message("Initial Guess"); print(x_guess);
message("Initial Function Value: "); print(func(x_guess));
message("Exit Function Value : ");print(func(x_now));
message("Exit Vector : "); print(x_now)
```

Figure 71 implements the script in Listing 30 for the example problem.

The next variant is to approximate the derivatives – usually a Finite-Difference approximation is used, either forward, backward, or centered differences – generally determined based on the actual behavior of the functions themselves or by trial and error. For really huge systems, we usually make the program itself make the adaptions as it proceeds.

The coding for a finite-difference representation of a Jacobian is shown in Listing 31. In constructing the Jacobian, we observe that each column of the Jacobian is simply the directional derivative of the function with respect to the variable associated with the column. For instance, the first column of the Jacobian in the example is first derivative of the first function (all rows) with respect to the first variable, in this case x. The second column is the first derivative of the second function with respect to the second variable, y. This structure is useful to generalize the Jacobian construction method because we can write (yet another) prototype function that can take the directional derivatives for us, and just insert the returns as columns. The example listing is specific to the 2X2 function in the example, but the extension to more general cases is evident.

Listing 31. R code demonstrating Newton's Method calculations using finite-difference approximations to the partial derivatives.

```
# R script for system of non-linear equations using Newton-Raphson with
#    finite-difference approximated derivatives
#    forward define the functions
####### f(x) ##########################
func <- function(x_vector){
  func <- numeric(0)
  func[1] <- x_vector[1]^2 + x_vector[2]^2 - 4
  func[2] <- exp(x_vector[1]) + x_vector[2] - 1
  return(func)
}
######## J(x) ##########################
jacob <- function(x_vector,func){  #supply a vector and the function name
# the columns of the jacobian are just directional derivatives
  dv <- 1e-06 #perturbation value for finite difference
  df1 <- numeric(0);
  df2 <- numeric(0);
  dxv <- x_vector;
  dyv <- x_vector;
# perturb the vectors
  dxv[1] <- dxv[1]+dv;
  dyv[2] <- dyv[2]+dv;
  df1 <- (func(dxv) - func(x_vector))/dv;
  df2 <- (func(dyv) - func(x_vector))/dv;
  jacob <- matrix(0,nrow=2,ncol=2)
# for a more general case should put this into a loop
  jacob[1,1] <- df1[1]   ;  jacob[1,2] <- df2[1]  ;
  jacob[2,1] <- df1[2]   ;  jacob[2,2] <- df2[2]  ;
  return(jacob)
}
####### Solver Parameters #############
x_guess <- c(2.,-0.8)
tolerancef <- 1e-9  # stop if function gets to zero
tolerancex <- 1e-9  # stop if solution not changing
maxiter <- 20 # stop if too many iterations
x_now <- x_guess
###### Newton-Raphson Algorithm ########
for (iter in 1:maxiter){
  funcNow <- func(x_now)
  testf <- t(funcNow) %*% funcNow
  if(testf < tolerancef){
    message("f(x) is close to zero : ", testf);
    break
  }
  dx <- solve(jacob(x_now,func),funcNow)
  testx <- t(dx) %*% dx
  if(testx < tolerancex){
    message("solution change small : ", testx);
    break
  }
  x_now <- x_now - dx
}
########################################
if( iter == maxiter) {message("Maximum iterations -- check if solution is converging : ")}
message("Initial Guess"); print(x_guess);
message("Initial Function Value: "); print(func(x_guess));
message("Exit Function Value : ");print(func(x_now));
message("Exit Vector : "); print(x_now)
```

7.1 Exercises

1. Write a script that forward defines the multi-variate functions and implements the Newton-Raphson technique in **R**. Implement the method, using analytical derivatives, and find a solution to:

$$\begin{aligned} x^3 &+ 3y^2 &&= 21 \\ x^2 &+ 2y &&= -2 \end{aligned} \tag{95}$$

2. Repeat the exercise, except use finite-differences to approximate the derivatives.

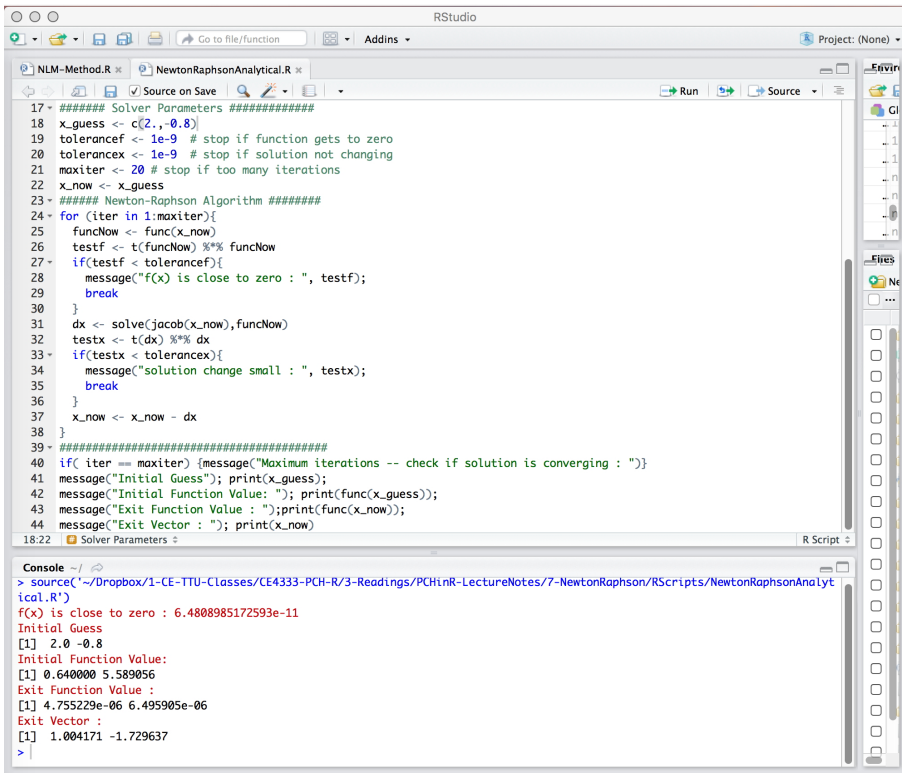

Figure 71. Newton-Raphson using Analytical Derivatives.

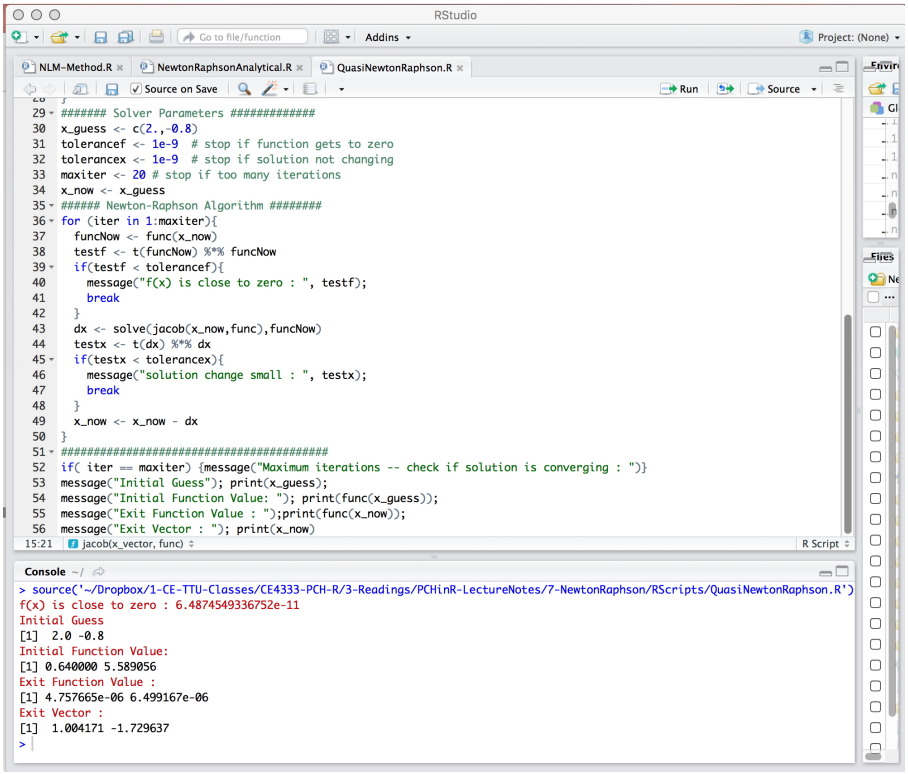

Figure 72. Newton-Raphson using Finite-Difference Approximated Derivatives.

3. Write a script that forward defines the multi-variate functions and implements the non-linear minimization technique in **R**. Implement the method, using analytical derivatives, and find a solution to:

$$\begin{array}{rrrcl} x^2 & + y^2 & + z^2 & = & 9 \\ & & xyz & = & 1 \\ x & + y & -z^2 & = & 0 \end{array} \tag{96}$$

4. Repeat the exercise, except use finite-differences to approximate the derivatives.

5. Write a script that forward defines the multi-variate functions and implements the non-linear minimization technique in **R**. Implement the method, using analytical derivatives, and find a solution to:

$$\begin{array}{rrrcl} xyz & - x^2 & + y^2 & = & 1.34 \\ & xy & - z^2 & = & 0.09 \\ e^x & - e^y & +z & = & 0.41 \end{array} \tag{97}$$

6. Repeat the exercise, except use finite-differences to approximate the derivatives.

8 Pipelines and Networks

Pipe networks, like single path pipelines, are analyzed for head losses in order to size pumps, determine demand management strategies, and ensure minimum pressures in the system. Conceptually the same principles are used for steady flow systems: conservation of mass and energy; with momentum used to determine head losses.

8.1 Pipe Networks – Topology

Network topology refers to the layout and connections. Networks are built of nodes (junctions) and arcs (links).

8.1.1 Continunity (at a node)

Water is considered incompressible in steady flow in pipelines and pipe networks, and the conservation of mass reduces to the volumetric flow rate, Q,

$$Q = AV \tag{98}$$

where A is the cross sectional of the pipe, and V is the mean section velocity. Typical units for discharge is liters per second (lps), gallons per minute (gpm), cubic meters per second (cms), cubic feet per second (cfs), and million gallons per day (mgd). The continuity equation in two cross-sections of a pipe as depicted in Figure 73 is

$$A_1 V_1 = A_2 V_2 \tag{99}$$

Junctions (nodes) are where two or more pipes join together. A three-pipe junction node with constant external demand is shown in Figure 10. The continuity equation for the junction node is

$$Q_1 - Q_2 - Q_3 - D = 0 \tag{100}$$

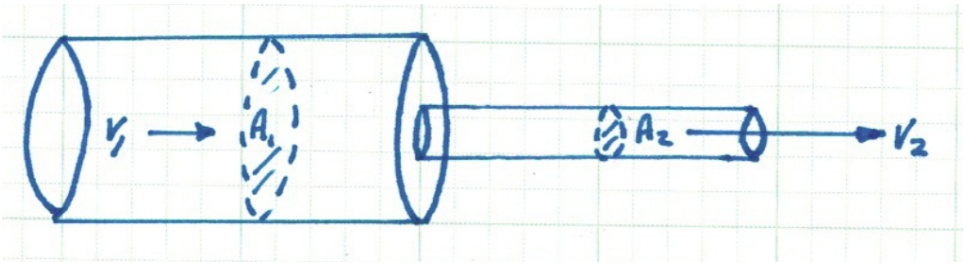

Figure 73. Continuity of mass (discharge) across a change in cross section.

In design analysis, all demands on the system are located at junctions (nodes), and the flow connecting junctions is assumed to be uniform across the cross sections (so

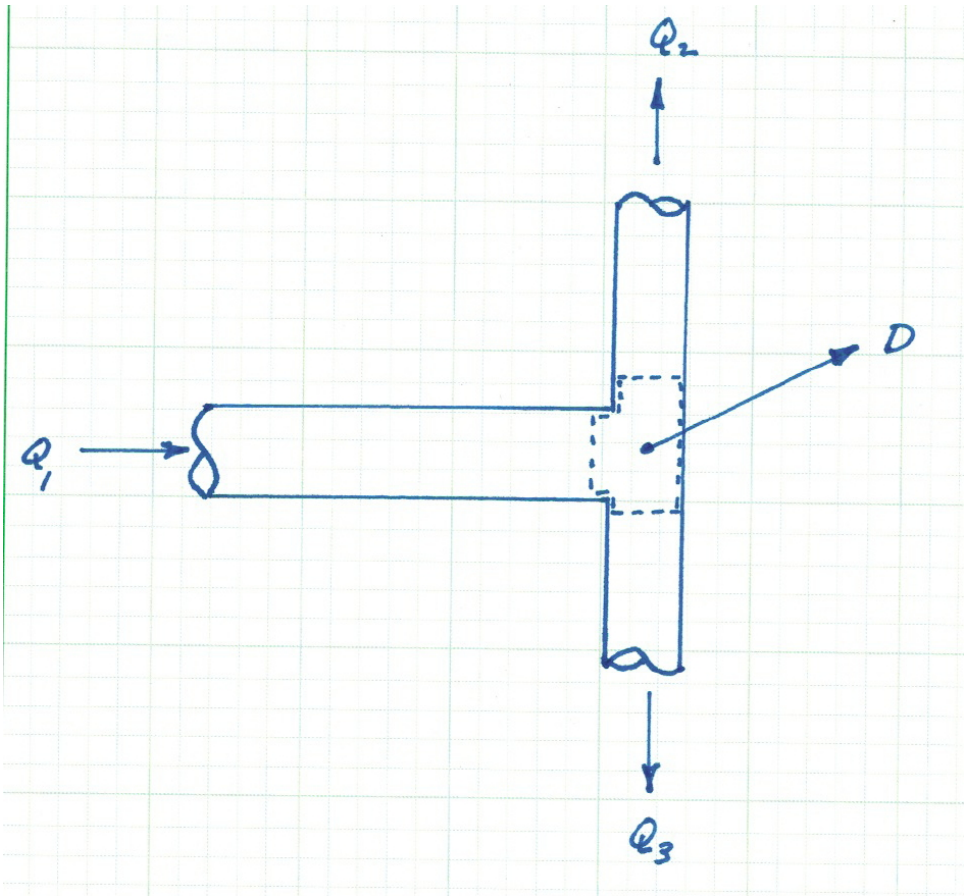

Figure 74. Continuity of mass (discharge) across a node (junction).

that mean velocities apply). If a substantial demand is located between nodes, then an additional node is established at the demand location.

8.1.2 Energy Loss (along a link)

Equation 101 is the one-dimensional steady flow form of the energy equation typically applied for pressurized conduit hydraulics.

$$\frac{p_1}{\rho g} + \alpha_1 \frac{V_1^2}{2g} + z_1 + h_p = \frac{p_2}{\rho g} + \alpha_2 \frac{V_2^2}{2g} + z_2 + h_t + h_l \tag{101}$$

where $\frac{p}{\rho g}$ is the pressure head at a location, $\alpha \frac{V^2}{2g}$ is the velocity head at a location, z is the elevation, h_p is the added head from a pump, h_t is the added head extracted by a turbine, and h_l is the head loss between sections 1 and 2. Figure 75 is a sketch that illustrates the various components in Equation 101.

In network analysis this energy equation is applied to a link that joins two nodes. Pumps and turbines would be treated as separate components (links) and their hy-

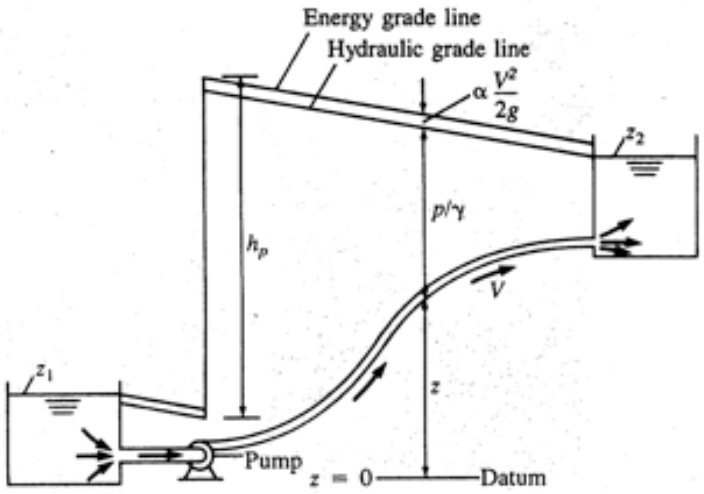

Figure 5-1 Definition sketch for terms in the energy equation

Figure 75. Definition sketch for energy equation.

draulic behavior must be supplied using their respective pump/turbine curves.

8.1.3 Velocity Head

The velocity in $\alpha\frac{V^2}{2g}$ is the mean section velocity and is the ratio of discharge to flow area. The kinetic energy correction coefficient is

$$\alpha = \frac{\int_A u^3 dA}{V^3 A} \tag{102}$$

where u is the point velocity in the cross section (usually measured relative to the centerline or the pipe wall; axial symmetry is assumed). Generally values of α are 2.0 if the flow is laminar, and approach unity (1.0) for turbulent flow. In most water distribution systems the flow is usually turbulent so α is assumed to be unity and the velocity head is simply $\frac{V^2}{2g}$.

8.1.4 Added Head — Pumps

The head supplied by a pump is related to the mechanical power supplied to the flow. Equation 103 is the relationship of mechanical power to added pump head.

$$\eta P = Q\rho g h_p \tag{103}$$

where the power supplied to the motor is P and the "wire-to-water" efficiency is η.

If the relationship is re-written in terms of added head[33] the pump curve is

$$h_p = \frac{\eta P}{Q\rho g} \tag{104}$$

This relationship illustrates that as discharge increases (for a fixed power) the added head decreases. Power scales at about the cube of discharge, so pump curves for computational application typically have a mathematical structure like

$$h_p = H_{\text{shutoff}} - K_{\text{pump}} Q^{\text{exponent}} \tag{105}$$

8.1.5 Extracted Head — Turbines

The head recovered by a turbine is also an "added head" but appears on the loss side of the equation. Equation 106 is the power that can be recovered by a turbine (again using the concept of "water-to-wire" efficiency is

$$P = \eta Q \rho g h_t \tag{106}$$

8.2 Pipe Head Loss Models

The Darcy-Weisbach, Chezy, Manning, and Hazen-Williams formulas are relationships between physical pipe characteristics, flow parameters, and head loss. The Darcy-Weisbach formula is the most consistent with the energy equation formulation being derivable (in structural form) from elementary principles.

$$h_{L_f} = f \frac{L}{D} \frac{V^2}{2g} \tag{107}$$

where h_{L_f} is the head loss from pipe friction, f is a dimensionless friction factor, L is the pipe length, D is the pipe characteristic diameter, V is the mean section velocity, and g is the gravitational acceleration.

The friction factor, f, is a function of Reynolds number Re_D and the roughness ratio $\frac{k_s}{D}$.

$$f = \sigma(Re_D, \frac{k_s}{D}) \tag{108}$$

The structure of σ is determined experimentally. Over the last century the structure is generally accepted to be one of the following depending on flow conditions and pipe properties

1. Laminar flow (Eqn 2.36, pg. 17 Chin (2006)) :

$$f = \frac{64}{Re_D} \tag{109}$$

[33] A negative head loss!

2. Hydraulically Smooth Pipes(Eqn 2.34 pg. 16 Chin (2006)):

$$\frac{1}{\sqrt{f}} = -2log_{10}(\frac{2.51}{Re_d\sqrt{f}}) \tag{110}$$

3. Hydraulically Rough Pipes(Eqn 2.34 pg. 16 Chin (2006)):

$$\frac{1}{\sqrt{f}} = -2log_{10}(\frac{\frac{k_e}{D}}{3.7}) \tag{111}$$

4. Transitional Pipes (Colebrook-White Formula)(Eqn 2.35 pg. 17 Chin (2006)):

$$\frac{1}{\sqrt{f}} = -2log_{10}(\frac{\frac{k_e}{D}}{3.7} + \frac{2.51}{Re_d\sqrt{f}}) \tag{112}$$

5. Transitional Pipes (Jain Formula)(Eqn 2.39 pg. 19 Chin (2006)):

$$f = \frac{0.25}{[log_{10}(\frac{\frac{k_e}{D}}{3.7} + \frac{5.74}{Re_d^{0.9}})]^2} \tag{113}$$

8.3 Pipe Networks Solution Methods

Several methods are used to produce solutions (estimates of discharge, head loss, and pressure) in a network. An early one, that only involves analysis of loops is the Hardy-Cross method. A later one, more efficient, is a Newton-Raphson method that uses node equations to balance discharges and demands, and loop equations to balance head losses. However, a rather ingenious method exists developed by Haman and Brameller (1971), where the flow distribution and head values are determined simultaneously. The task here is to outline the Haman and Brameller (1971) method on the problem below – first some necessary definitions and analysis.

The fundamental procedure is:

1. Continuity is written at nodes (node equations).
2. Energy loss (gain) is written along links (pipe equations).
3. The entire set of equations is solved simultaneously.

8.4 Network Analysis

Figure 76 is a sketch of the problem that will be used. The network supply is the fixed-grade node in the upper left hand corner of the drawing. The remaining nodes (N1 – N4) have demands specified as the purple outflow arrows. The pipes are labeled

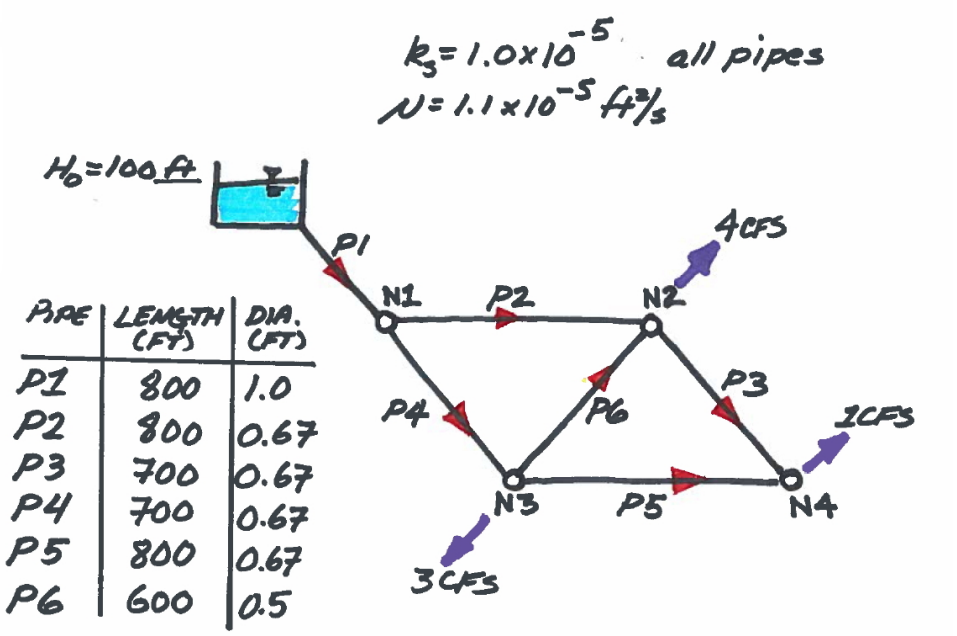

Figure 76. Pipe network for illustrative example with supply and demands identified. Pipe dimensions and diameters are also depicted..

(P1 – P6), and the red arrows indicate a positive flow direction, that is, if the flow is in the indicated direction, the numerical value of flow (or velocity) in that link would be a positive number.

Define the flows in each pipe and the total head at each node as Q_i and H_i where the subscript indicates the particular component identification. Expressed as a vector, these unknowns are:

$$[Q_1,\ Q_2,\ Q_3,\ Q_4,\ Q_5,\ Q_6,\ H_1,\ H_2,\ H_3,\ H_4] \ = \ \mathbf{x}$$

If we analyze continuity for each node we will have 4 equations (corresponding to each node) for continunity, for instance for Node N2 the equation is

$$Q_2 \quad -Q_3 \qquad\qquad Q_6 \qquad\qquad = \ 4$$

Similarily if we define head loss in any pipe as $\Delta H_i = f\frac{8L_i}{\pi^2 g D_i^5}|Q_i|Q_i$ or $\Delta H_i = L_i Q_i$, where $L_i = f\frac{8L_i}{\pi^2 g D_i^5}|Q_i|$, then we have 6 equations (corresponding to each pipe) for energy, for instance for Pipe (P2) the equation is[34]

$$-L_2 Q_2 \qquad\qquad H_1 \quad -H_2 \qquad = \ 0$$

[34]The seemingly awkward way of writing the equations will become apparent shortly!

If we now write all the node equations then all the pipe equations we could construct the following coefficient matrix below:[35]

$$\begin{array}{cccccccccc}
1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
-L_1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & -L_2 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & -L_3 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & -L_4 & 0 & 0 & 1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & -L_5 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & -L_6 & 0 & -1 & 1 & 0
\end{array}$$

Declare the name of this matrix $\mathbf{A(x)}$, where $\mathbf{x}$ denotes the unknown vector of Q augmented by H as above. Next consider the right-hand-side at the correct solution (as of yet still unknown!) as

$$[0, \quad 4, \quad 3, \quad 1, \quad -100, \quad 0, \quad 0, \quad 0, \quad 0, \quad 0] = \mathbf{b}$$

So if the coefficient matrix is correct then the following system would result:

$$\mathbf{A}(\mathbf{x}) \cdot \mathbf{x} = \mathbf{b}$$

which would look like

$$\left(\begin{array}{cccccccccc}
1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
-L_1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & -L_2 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & -L_3 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & -L_4 & 0 & 0 & 1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & -L_5 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & -L_6 & 0 & -1 & 1 & 0
\end{array}\right)
\begin{pmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \\ Q_5 \\ Q_6 \\ H_1 \\ H_2 \\ H_3 \\ H_4 \end{pmatrix}
=
\left(\begin{array}{c} 0 \\ 4 \\ 3 \\ 1 \\ \hline -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array}\right) \tag{114}$$

Observe, the system is non-linear because the coefficient matrix depends on the current values of Q_i for the L_i terms. However, the system is full-rank (rows == columns) so it is a candidate for Newton-Raphson.

[35] The horizontal lines divide the node and the pipe equations. The upper partition are the node equations in Q and H, the lower partition are the pipe equations in Q and H

Further observe that the upper partition from column 6 and smaller is simply the node-arc incidence matrix, and the lower partition for the same columns only contains L_i terms on its diagonal, the remainder is zero. Next observe that the partition associated with heads in the node equations is the zero-matrix.

Lastly (and this is important!) the lower right partition is the transpose of the node-arc incidence matrix subjected to scalar multiplication of -1. The importance is that all the information needed to find a solution is contained in the node-arc incidence matrix and the right-hand-side – the engineer does not need to identify closed loops (nor does the computer need to find closed loops).

The trade-off is a much larger system of equations, however solving large systems is far easier that searching a directed graph to identify closed loops, furthermore we obtain the heads as part of the solution process.

9 Pipelines Network Analysis

The prior chapter introduced the non-linear system that results from the analysis of the pipeline network. This chapter continues the effort and produces a workable **R** script that can compute flows and heads given just the node-arc incidence matrix, and pipe properties.

Recall from the prior chapter the non-linear system to be solved is

$$\mathbf{A}(\mathbf{x}) \cdot \mathbf{x} = \mathbf{b}$$

which would look like

$$\left(\begin{array}{cccccccccc} 1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline -L_1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -L_2 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -L_3 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -L_4 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -L_5 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -L_6 & 0 & -1 & 1 & 0 \end{array}\right) \left(\begin{array}{c} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \\ Q_5 \\ Q_6 \\ H_1 \\ H_2 \\ H_3 \\ H_4 \end{array}\right) = \left(\begin{array}{c} 0 \\ 4 \\ 3 \\ 1 \\ \hline -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array}\right) \tag{115}$$

The system is non-linear because the coefficient matrix depends on the current values of Q_i for the L_i terms. The upper partition from column 6 and smaller is simply the node-arc incidence matrix, and the lower partition for the same columns only contains L_i terms on its diagonal, the remainder is zero. Next observe that the partition associated with heads in the node equations is the zero-matrix. The lower right partition is the transpose of the node-arc incidence matrix subjected to scalar multiplication of -1. So using the Newton-Raphson approach discussed earlier we develop a script in **R** that produces estimates of discharge and total head in the system depicted in Figure 76.

9.1 Script Structure

The script will need to accomplish several tasks including reading the node-arc incidence matrix supplied as the file in Figure 77 and convert the strings into numeric values. The script will also need some support functions defined before constructing the matrix.

The rows of the input file are:

```
4
6
1.00 0.67 0.67 0.67 0.67 0.5
800 800 700 700 800 600
0.00001 0.00001 0.00001 0.00001 0.00001 0.00001
0.000011
1 1 1 1 1 1
1 -1 0 -1 0 0
0 1 -1 0 0 1
0 0 0 1 -1 -1
0 0 1 0 1 0
0 4 3 1 -100 0 0 0 0 0
```

Figure 77. Input file for the problem.

1. The node count.
2. The pipe count.
3. Pipe diameters, in feet.
4. Pipe lengths, in feet.
5. Pipe roughness heights, in feet.
6. Kinematic viscosity in feet^2/second.
7. Initial guess of flow rates (unbalanced OK, non-zero vital!)
8. The next four rows are the node-arc incidence matrix.
9. The last row is the demand (and fixed-grade node total head) vector.

9.1.1 Support Functions

The Reynolds number will need to be calculated for each pipe at each iteration of the solution, so a Reynolds number function will be useful. For circular pipes, the following equation should work,

$$Re(Q) = \frac{8L}{\mu\pi D}|Q| \tag{116}$$

The Jain equation (Jain, 1976) that directly computes friction factor from Reynolds number, diameter, and roughness is

$$f(k_s, D, Re) = \frac{0.25}{[log(\frac{k_s}{3.7D} + \frac{5.74}{Re^{0.9}})]^2} \tag{117}$$

Once you have the Reynolds number for a pipe, and the friction factor, then the head loss factor that will be used in the coefficient matrix (and the Jacobian) is

$$L_i = f\frac{8L_i}{\pi^2 g D_i^5}|Q_i| \tag{118}$$

These three support functions are coded in **R** as shown in Listing 32.

Listing 32. R Code to compute Reynolds numbers and friction factors

.

```
#################################################################
##############  Forward Define Support Functions   #################
#################################################################
# Jain Friction Factor Function -- Tested OK 23SEP16
friction_factor <- function(roughness,diameter,reynolds){
  temp1 <- roughness/(3.7*diameter);
  temp2 <- 5.74/(reynolds^(0.9));
  temp3 <- log10(temp1+temp2);
  temp3 <- temp3^2;
  friction_factor <- 0.25/temp3;
  return(friction_factor)
}
# Velocity Function
velocity <- function(diameter,discharge){
  velocity <- discharge/(0.25*pi*diameter^2)
  return(velocity)
}
# Reynolds Number Function
reynolds_number <- function(velocity,diameter,mu){
  reynolds_number <- abs(velocity)*diameter/mu
  return(reynolds_number)
}
# Geometric factor function
k_factor <- function(howlong,diameter,gravity){
  k_factor <- (16*howlong)/(2.0*gravity*pi^2*diameter^5)
  return(k_factor)
}
```

9.1.2 Augmented and Jacobian Matrices

The $\mathbf{A(x)}$ is built using the node-arc incidence matrix (which does not change), and the current values of L_i. You will also need to build the Jacobian of $\mathbf{A(x)}$ to implement the update as-per Newton-Raphson.

A brief review; at the solution we can write

$$[\mathbf{A}(\mathbf{x})] \cdot \mathbf{x} - \mathbf{b} = \mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{119}$$

Lets assume we are not at the solution, so we need a way to update the current value of $\mathbf{x}$. Recall from Newton's method (for univariate cases) that the update formula is

$$x_{k+1} = x_k - (\frac{df}{dx}\mid_{x_k})^{-1} f(x_k) \tag{120}$$

The Jacobian will play the role of the derivative, and $\mathbf{x}$ is now a vector (instead of a single variable). Division is not defined for matrices, but the multiplicative inverse is (the inverse matrix), and plays the role of division. Hence, the extension to the pipeline case is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{f}(\mathbf{x}_k) \tag{121}$$

where $\mathbf{J}(\mathbf{x}_k)$ is the Jacobian of the coefficient matrix $\mathbf{A}$ evaluated at $\mathbf{x}_k$. Although a bit cluttered, here is the formula for a single update step, with the matrix, demand vector, and the solution vector in their proper places.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}(\mathbf{x}_k)]^{-1}\{[\mathbf{A}(\mathbf{x}_k)] \cdot \mathbf{x}_k - \mathbf{b}\} \tag{122}$$

As a practical matter we actually never invert the Jacobian[36], instead we solve the related Linear system of

$$[\mathbf{J}(\mathbf{x}_k)] \cdot \Delta\mathbf{x} = \{[\mathbf{A}(\mathbf{x}_k)] \cdot \mathbf{x}_k - \mathbf{b}\} \tag{123}$$

for $\Delta\mathbf{x}$, then perform the update as $\mathbf{x}_{k+1} = \mathbf{x}_k$ - $\Delta\mathbf{x}$

The Jacobian of the pipeline model is a matrix with the following properties:

1. The partition of the matrix that corresponds to the node formulas (upper left partition) is identical to the original coefficient matrix — it will be comprised of 0 or $\pm$ 1 in the same pattern at the equivalent partition of the $\mathbf{A}$ matrix.
2. The partition of the matrix that corresponds to the pipe head loss terms (lower left partition), will consist of values that are twice the values of the coefficients in the original coefficient matrix (at any supplied value of $\mathbf{x}_k$.
3. The partition of the matrix that corresponds to the head terms (lower right partition), will consist of values that are identical to the original matrix.
4. The partition of the matrix that corresponds to the head coefficients in the node equations (upper right partition) will also remain unchanged.

You will want to take advantage of problem structure to build the Jacobian (you could just finite-difference the coefficient matrix to approximate the partial derivatives, but that is terribly inefficient if you already know the structure).

9.1.3 Stopping Criteria, and Solution Report

You will need some way to stop the process – the three most obvious (borrowed from Newton's method) are:

1. Approaching the correct solution (e.g. $[\mathbf{A}(\mathbf{x})] \cdot \mathbf{x} - \mathbf{b} = \mathbf{f}(\mathbf{x}) = \mathbf{0}$).

[36] Inverting the matrix every step is computationally inefficient, and unnecessary. As an example, solving the system in this case would at worst take 10 row operations each step, but nearly 100 row operations to invert at each step – to accomplish the same result, generate an update. Now imagine when there are hundreds of nodes and pipes!

2. Update vector is not changing (e.g. $\mathbf{x}_{k+1} = \mathbf{x}_k$), so either have an answer, or the algorithm is stuck.

3. You have done a lot of iterations (say 100).

Listing 33 is a code fragment to find the flow distribution and heads for the example problem. Not listed is the forward defined functions already listed above – these should be placed into the script in the location shown (or directly sourced into the code in **R**).

Listing 33. R Code to Implement Pipe Network Solution
This fragment reads the data file and converts it into numeric values and reports back the values.

```
# Steady Flow in a Pipe Network Using Hybrid Method (and Newton-Raphson) based on
# Haman YM, Brameller A. Hybrid method for the solution of piping networks. Proc IEEE
    1971;118(11):1607?12.
#
# Clear all existing objects
 rm(list=ls())
################################################################
##############Forward Define Support Functions Go Here ##########
################################################################
# Read Input Data Stream from File
zz <- file("PipeNetwork.txt", "r") # Open a connection named zz to file named PipeNetwork.
    txt
nodeCount <- as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
pipeCount <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
diameter <- (readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
distance <- (readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
roughness <- (readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
viscosity <- (readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
flowguess <- (readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
nodearcs <- (readLines(zz, n = nodeCount, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
rhs_true <- (readLines(zz, n = pipeCount+nodeCount, ok = TRUE, warn = TRUE,encoding = "
    unknown", skipNul = FALSE))
close(zz) # Close connection zz
#
# Convert Input Stream into Numeric Structures
diameter <-as.numeric(unlist(strsplit(diameter,split=" ")))
distance <-as.numeric(unlist(strsplit(distance,split=" ")))
roughness <-as.numeric(unlist(strsplit(roughness,split=" ")))
viscosity <-as.numeric(unlist(strsplit(viscosity,split=" ")))
flowguess <-as.numeric(unlist(strsplit(flowguess,split=" ")))
nodearcs <-as.numeric(unlist(strsplit(nodearcs,split=" ")))
rhs_true <-as.numeric(unlist(strsplit(rhs_true,split=" ")))
# convert nodearcs a matrix
# We will need to augment this matrix for the actual solution -- so after augmentation will
     deallocate the memory
nodearcs <-matrix(nodearcs,nrow=nodeCount,ncol=pipeCount,byrow = TRUE)
# echo input
message("Node Count = ",nodeCount)
message("Pipe Count = ",pipeCount)
message("Pipe Lengths = "); distance
message("Pipe Diameters = "); diameter
message("Pipe Roughness = "); roughness
message("Fluid Viscosity = ",viscosity)
message("Initial Guess = "); flowguess
message("Node-Arc-Incidence Matrix = "); nodearcs
#
```

Listing 34 is a code fragment to construct the coefficient matrix structure for the non-changing part and allocate variables for the Newton-Raphson method.

Listing 34. R Code to Implement Pipe Network Solution
This fragment constructs the initial **A(x)** matrix and allocates variables used in the iteration loop.

```
# create the augmented matrix
headCount <- nodeCount
flowCount <- pipeCount
augmentedRowCount <- nodeCount+pipeCount
augmentedColCount <- flowCount+headCount
augmentedMat <- matrix(0,nrow=augmentedRowCount,ncol=augmentedColCount,byrow = TRUE)
#
augmentedMat
# build upper left partition of matrix -- this partition is constants from node-arc matrix
for (i in 1:nodeCount){
  for (j in 1:flowCount){
    augmentedMat[i,j] <- nodearcs[i,j]
  }
}
augmentedMat
# build lower right partition of matrix -- this partition is -1*transpose(node-arc) matrix
istart <- nodeCount+1
iend <- nodeCount+pipeCount
jstart <- flowCount+1
jend <- flowCount+headCount
for (i in istart:iend ){
  for(j in jstart:jend ){
    augmentedMat[i,j] <- -1*nodearcs[j-jstart+1,i-istart+1]
  }
}
augmentedMat
# here it should be safe to delete the nodearc matrix
rm(nodearcs)
# Need some vorking vectors
 HowMany <- 50
 tolerance1 <- 1e-24
 tolerance2 <- 1e-24
 velocity_pipe <-numeric(0)
 reynolds <- numeric(0)
 friction <- numeric(0)
 geometry <- numeric(0)
 lossfactor <- numeric(0)
 jacbMatrix <- matrix(0,nrow=augmentedRowCount,ncol=augmentedColCount,byrow = TRUE)
 gq <- numeric(0)
 solvecguess <- numeric(length=augmentedRowCount)
 solvecnew <- numeric(length=augmentedRowCount)
 solvecguess[1:flowCount] <- flowguess[1:flowCount]

 # compute geometry factors (only need once, goes outside iteration loop)
 for (i in 1:pipeCount)
 {
   geometry[i] <- k_factor(distance[i],diameter[i],32.2)
 }
 geometry
```

Listing 35 is the code fragment that implements the iteration loop of the Newton-Raphson method. Within each iteration, the support functions are repeatedly used to construct the changing part of the coefficient and Jacobian matrices, solving the resulting linear system, performing the vector update, and testing for stopping.

Listing 35. R Code to Implement Pipe Network Solution
This fragment executes the iteration loop where the Newton-Raphson method and updates are implemented.

```
 # going to wrap below into an interation loop -- forst a single instance
for (iteration in 1:HowMany){
#################### BEGIN ITERATION OUTER LOOP ############################
 # compute current velocity
 for (i in 1:pipeCount)
 {
   velocity_pipe[i]<-velocity(diameter[i],flowguess[i])
 }
 # compute current reynolds
 for (i in 1:pipeCount)
 {
   reynolds[i]<-reynolds_number(velocity_pipe[i],diameter[i],viscosity)
 }
 # compute current friction factors
 for (i in 1:pipeCount)
 {
   friction[i]<-friction_factor(roughness[i],diameter[i],reynolds[i])
 }
 # compute current loss factor
 for (i in 1:pipeCount)
 {
   lossfactor[i] <- friction[i]*geometry[i]*abs(flowguess[i])
 }
 # build the function matrix
 # operate on the lower left partition of the matrix
 istart <- nodeCount+1
 iend <- nodeCount+pipeCount
 jstart <- 1
 jend <- flowCount
 for (i in istart:iend ){
   for(j in jstart:jend ){
     if ((i-istart+1) == j)  augmentedMat[i,j] <- -1*lossfactor[j]
   }
 }
 # now build the current jacobian
 # slick trick -- we will copy the current function matrix, then modify the lower left
     partition
 jacbMatrix <- augmentedMat
 # build the function matrix
 # operate on the lower left partition of the matrix
 istart <- nodeCount+1
 iend <- nodeCount+pipeCount
 jstart <- 1
 jend <- flowCount
 for (i in istart:iend ){
   for(j in jstart:jend ){
     if ((i-istart+1) == j)  jacbMatrix[i,j] <- 2*jacbMatrix[i,j]
   }
 }
# now build the gq() vector
gq <- augmentedMat %*% solvecguess - rhs_true
gq
dq <- solve(jacbMatrix,gq)
# update the solution vector
solvecnew <- solvecguess - dq
solvecnew
# # now test for stopping
test <- abs(solvecnew - solvecguess)
if( t(test) %*% test < tolerance1){
  message("Update not changing -- exit loop and report current update")
  message("Iteration count = ",iteration)
  solvecguess <- solvecnew
  flowguess[1:flowCount] <- solvecguess[1:flowCount]
  break
}
test <- abs(gq)
if( t(test) %*% test < tolerance2 ){
  message("G(Q) close to zero -- exit loop and report current update")
  message("Iteration count = ",iteration)
  solvecguess <- solvecnew
  flowguess[1:flowCount] <- solvecguess[1:flowCount]
  break
}
solvecguess <- solvecnew
flowguess[1:flowCount] <- solvecguess[1:flowCount]
#################### END OF ITERATION OUTER LOOP ##############################
}
message("Current Results")
print(cbind(solvecguess,gq,dq))
print(cbind(friction,diameter,distance,velocity_pipe))
```

Figure 78 is a screen capture of the script running the example problem. The first column in the output is the solution vector. The first 6 rows are the flows in pipes P1-P6. The remaining 4 rows are the heads at nodes N1-N4.

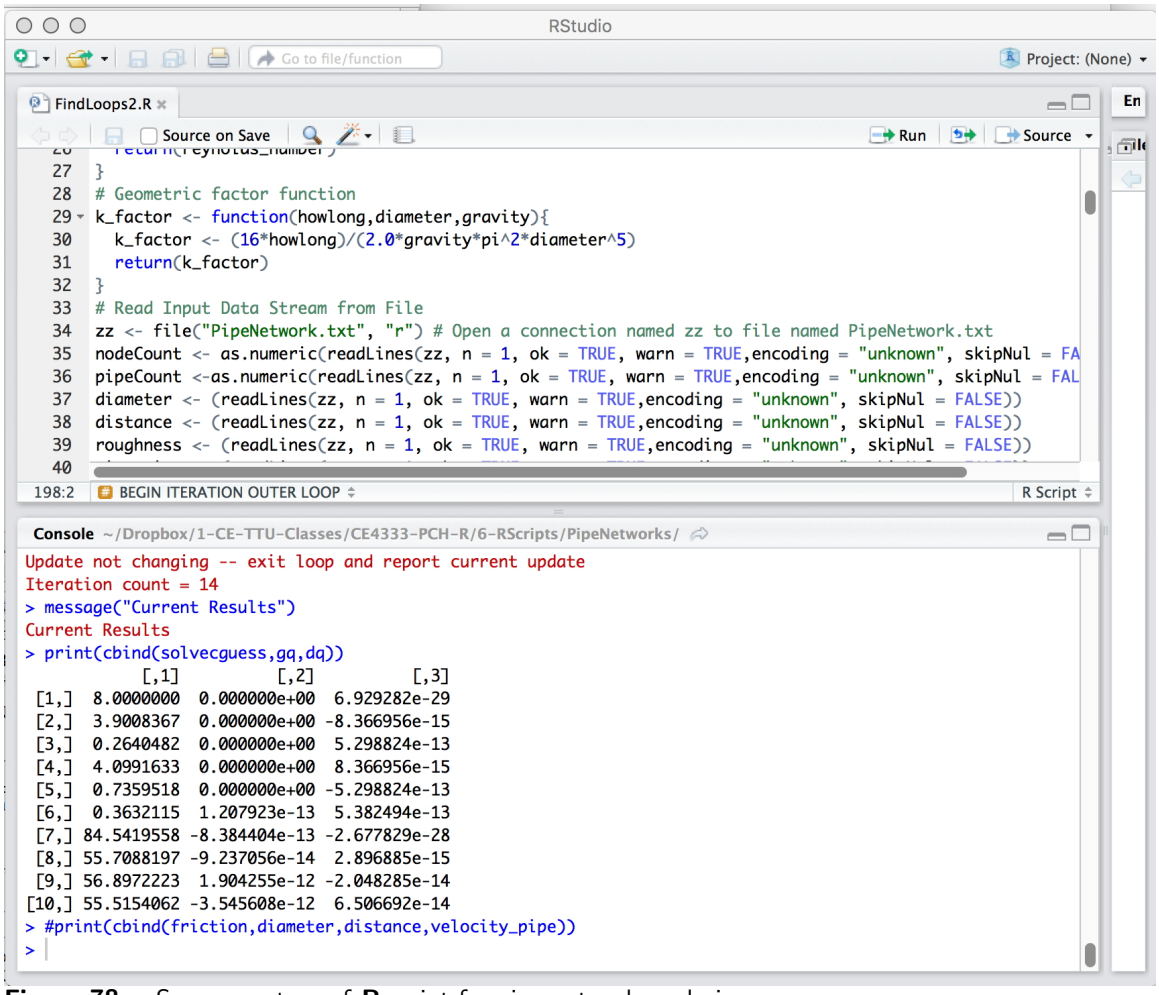

Figure 78. Screen capture of **R** script for pipe network analysis.

9.2 Exercises

1. Figure 79 is a five-pipe network with a water supply source at Node 1, and demands at Nodes 1-5. Table 7 is a listing of the node and pipe data.

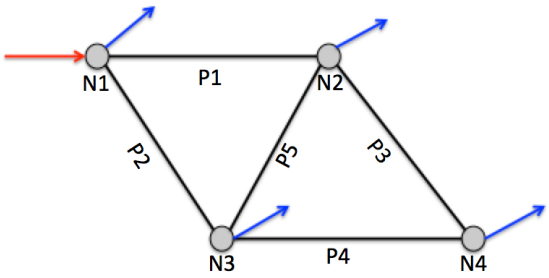

Figure 79. Layout of Simple Network.

Table 7. Node and Pipe Data.

Pipe ID	Diameter (inches)	Length (feet)	Roughness (feet)
P1	8	800	0.00001
P2	8	700	0.00001
P3	8	700	0.00001
P4	8	800	0.00001
P5	6	600	0.00001
Node ID	**Demand (CFS)**	**Elevation (feet)**	**Head (feet)**
N1	2.0	0.0	100
N2	4.0	0.0	?
N3	3.0	0.0	?
N4	1.0	0.0	?

Code the script, build an input file, and determine the flow distribution In your solution you are to supply

(a) An analysis showing the development of the node-arc incidence matrix based on the flow directions in Figure 79,

(b) The input file you constructed to provide the simulation values to your script, and

(c) A screen capture (or output file) showing the results.

2. Code the script and determine the flow distribution in Figures 80 and 81. Assume Node N1 has a total head of 300 feet.

   In your solution you are to supply

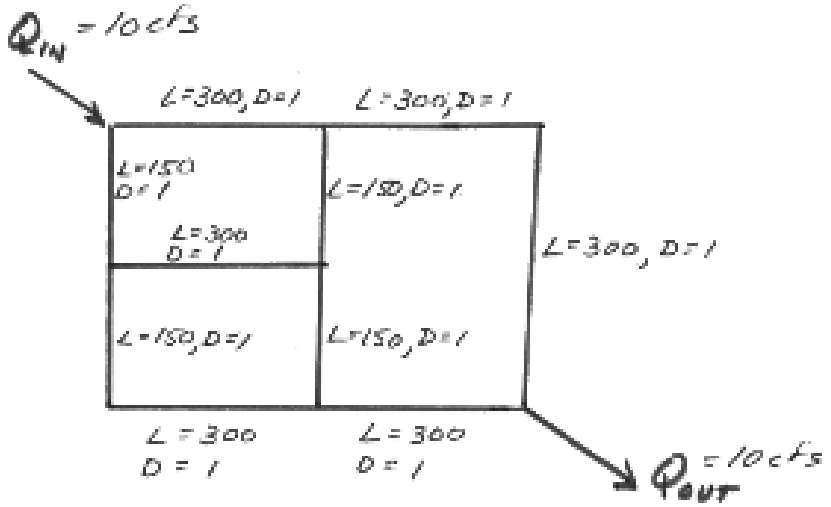

Figure 80. Pipe network for illustrative example with supply and demands identified. Pipe lengths (in feet) and diameters (in feet) are also depicted..

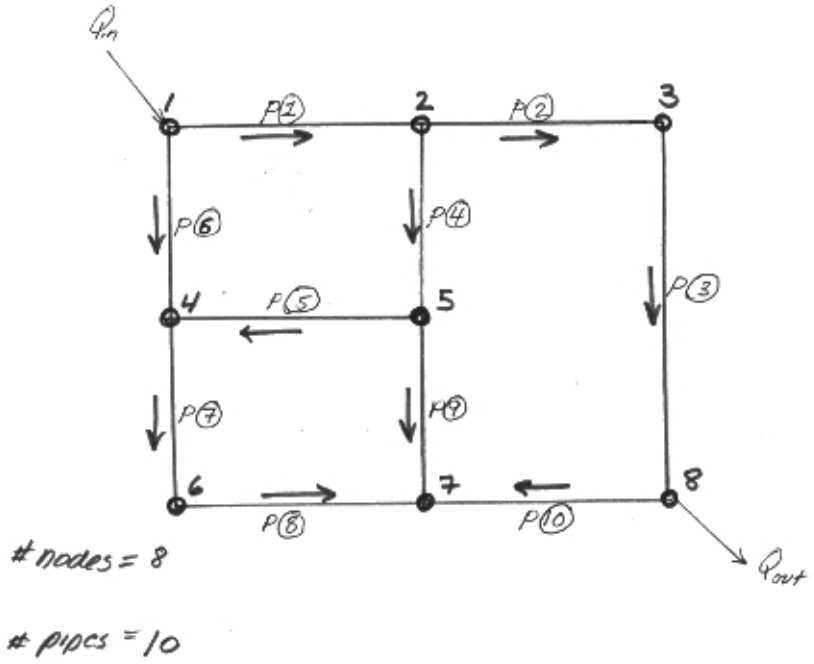

Figure 81. Pipe network for illustrative example with pipes and nodes labeled..

(a) An analysis showing the development of the node-arc incidence matrix based on the flow directions in Figure 81,

(b) The input file you constructed to provide the simulation values to your script, and

(c) A screen capture (or output file) showing the results.

3. Modify the script to include node elevation information to compute pressures. Assume all nodes are at elevation 200 feet.

4. (Advanced) Assume the supply Node N1 in the problem in Figure 76 has total head of 100 feet, as shown. Assume all the other nodes are at elevation 200 feet. Modify the algorithm and script to replace Pipe 1 with a pump with the following pump curve.

$$h_p = H_{\text{shutoff}} - K_{pump} Q^2 \tag{124}$$

10 Pumps and Valves

11 Pipeline Transients — Water Hammer

11.1 Head Loss Models

11.2 Finite-Difference Methods

11.3 Characteristics for Boundary Conditions

11.4 Exercises

12 Open Channel Flow

12.1 Uniform Flow

12.1.1 Hydraulic Elements — Depth-Area; Depth-Topwidth; Depth-Perimeter Functions

12.2 Steady Gradually Varied Flow

Need a brief theory

12.2.1 Finite Difference Methods — Fixed Step Method

The fixed-step refers to fixed changes in depth for which we solve to find the variable spatial steps. The method is a very simple method for computing water surface profiles in prismatic channels. A prismatic channel is a channel of uniform cross sectional geometry[37] with constant bed (topographic) slope.

In such channels with smooth (non-jump) steady flow the continunity and momentum equations are

$$Q = AV \tag{125}$$

where Q is volumetric discharge, A is cross sectional flow area, and V is the mean section velocity.

and

$$\frac{V}{g}\frac{dV}{dx} + \frac{dh}{dx} = S_o - S_f \tag{126}$$

where h is the flow depth (above the bottom), and x is horizontal the distance along the channel.

For the variable step method, the momentum equation is rewritten as a difference equation (after application of calculus to gather terms) then rearranged to solve for the spatial step dimension [38].

[37] Channel geometry is same at any section, thus rectangular, trapezoidal, and circular channels if the characteristic width dimension is constant would be prisimatic.

[38] The equation here is written moving upstream, direction matters for indexing. Thus position $i+1$ is assumed upstream of position i in this essay. This directional convention is not generally true in numerical methods and analysts need to use care when developing their own tools or using other tools. A clever analyst need not rewrite code, but simple interchange of upstream and downstream depths will handle both backwater and front-water curves.

$$\frac{\frac{V_{i+1}^2}{2g} - \frac{V_i^2}{2g}}{\Delta x} + \frac{h_{i+1} - h_i}{\Delta x} = S_o - \bar{S}_f \tag{127}$$

where $\bar{S}_f$ is the average slope of the energy grade line between two sections (along a reach of length Δx, the unknown value).

Rearrangement to isolate Δx produces an explicit update equation that can be evaluated to fond the different values of Δx associated with different flow depths. The plot of the accumulated spatial changes versus the sum of the flow depth and bottom elevation is the water surface profile.

$$\frac{(h_{i+1} + \frac{V_{i+1}^2}{2g}) - (h_i + \frac{V_i^2}{2g})}{S_o - \bar{S}_f} = \Delta x \tag{128}$$

The distance between two sections with known discharges is computed using the equation, all the terms on the left hand side are known values. The mean energy gradient ($\bar{S}_f$) is computed from the mean of the velocity, depth, area, and hydraulic radius for the two sections.

The friction slope can be computed using Manning's, Chezy, or the Darcy-Weisbach friction equations adapted for non-circular, free-surface conduits.

12.2.2 Coding the algorithm in R

The algorithm described can be implemented in R, `FORTRAN`, and Excel with about equal ease. Here the method is illustrated in **R** to illustrate the tool as a programming environment[39] ; the same problem in `FORTRAN` and Excel is left as an exercise.

First we build a set of utility functions, these will be used later in the `backwater` function:

1. Flow area given channel depth and width (it assumes rectangular channels):

```
# Depth-Area Function
area<-function(depth,width){
area<-depth*width;
return(area)
}
```

2. Wetted perimeter given channel depth and width (it also assumes rectangular channels):

```
# Wetted perimeter function for rectangular channel
perimeter<-function(depth,width){
```

[39]The **R** code in this essay is broken into parts and some unnecessary line feeds are included to fit the page.

```
perimeter<-2*depth+width;
return(perimeter)
}
```

3. Hydraulic radius (ratio of the above results), this is a generic function, it does not need to know the flow geometry:

```
# Hydraulic radius function
radius<-function(area,perimeter){
radius<-area/perimeter;
return(radius)
}
```

4. The friction slope given Manning's n, discharge, hydraulic radius, and flow area. Notice that this function implicitly assumes SI units (the 1.49 constant in U.S. Customary units is not present).

```
# Friction slope function
slope_f<-function(discharge,mannings_n,area,radius){
slope_f<-(discharge^2)*(mannings_n^2)/( (radius^(4/3))*(area^2) );
return(slope_f)
}
```

5. Average value of two arguments, probably unnecessary, but has some value when getting things to work.

```
# 2-point average
avg2point<-function(x1,x2){
avg2point<-0.5*(x1+x2);
return(avg2point)
}
```

The semi-colon in the functions is probably unnecessary, but has value because it forces the expression to its left to be evaluated and helps prevent ambiguous[40] code. Also notice the use of the scope { } delimiter, the delimiter is required, however there is no requirement to line feed — I simply find it easier to read my own code in this fashion (and count delimiters).

At this point, we would have 5 useful, testable functions (and we should test before the next step.

The next step is the `backwater` function. This function computes the space steps, changes in depth, etc. as per the algorithm. The function is a `FORTRAN` port, so it is not the best use of **R**, but it illustrates count controlled repetition (for loops), array indexing, and use of the utility functions to make the code readable as well as ensure that the parts work before the whole program is assembled. This concept is really crucial, if you can build a tool of parts that are known to work, it helps keep logic

[40]To the human operator; the computer will either accept the code or throw an error.

errors contained to known locations.

```
# Backwater curve function
backwater<-function(begin_depth,end_depth,how_many,discharge,width,mannings_n,slope){
#
## Example function call will be shown in class and on movie.
#
# Other functions must exist otherwise will spawn errors
#
# Prepare space for vectors
 depth<-numeric(0)
 bse<-numeric(0)
 wse<-numeric(0)
# change in depth for finding spatial steps
 delta_depth<-(begin_depth-end_depth)/(how_many)
# assign downstream value
 depth[1]<-begin_depth
# depth values to evaluate
 for (i in 2:how_many){depth[i]<-depth[1]-i*delta_depth}
#velocity for each depth
 velocity<-discharge/area(depth,width)
# numeric vector for space steps (destination space)
 deltax<-numeric(0)
# next for loop is very FORTRANesque!
for (i in 1:(how_many-1)){
#compute average depth in reach
    depth_bar<-avg2point(depth[i],depth[i+1]);
#compute average area in reach
    area_bar<-area(depth_bar,width);
#compute average wetted perimeter
    perimeter_bar<-perimeter(depth_bar,width);
 #compute average hydraulic radius
    radius_bar<-radius(area_bar,perimeter_bar);
 #compute friction slope
    friction<-slope_f(discharge,mannings_n,area_bar,radius_bar)
 # compute change in distance for each change in depth
    deltax[i]<-( (depth[i+1]+(velocity[i+1]^2)/(2*9.8)
                      - (depth[i] + (velocity[i]^2)/(2*9.8)) )  /  (slope-friction);
}
# space for computing cumulative distances
 distance<-numeric(0);
 distance[1]<-0;
 bse[1]<-0; # bottom elevation at origin
 for (i in 2:(how_many)){
distance[i]<-distance[i-1]+deltax[i-1]; # spatial distances
```

```
bse[i]<-bse[i-1]-deltax[i-1]*slope; # bottom elevations
}
 wse<-bse+depth # water surface elevations
# output
 z<-cbind(distance,depth,bse,wse) # bind output into 4 columns
return(z)
#
}
```

12.2.3 Example 1 — Backwater curve

Figure 82 is a backwater curve[41] for a rectangular channelwith discharge over a weir (on the right hand side — not depicted). The channel width is 5 meters, bottom slope 0.001, Manning's $n = 0.02$ and discharge $Q = 55.4\frac{m^3}{sec}$.

Using the `backwater` function and some plot calls in **R** we can duplicate the figure (assuming the figure is essentially correct).

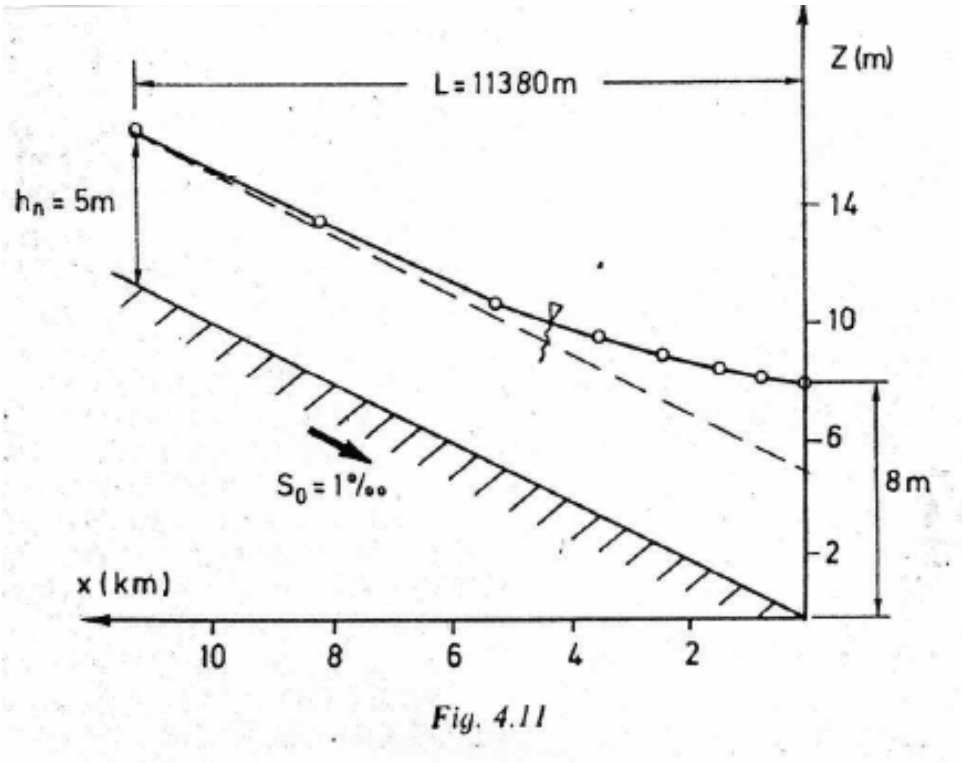

Figure 82. Example backwater curve.

```
> # here is the function call
> backwater(begin_depth=8,end_depth=5,how_many=31,
```

[41] Page 85. Koutitas, C.G. (1983). Elements of Computational Hydraulics. Pentech Press, London 138p. ISBN 0-7273-0503-4

```
    + discharge=55.4,width=5,mannings_n=0.02,slope=0.001)
         distance    depth        bse       wse
 [1,]      0.0000 8.000000  0.0000000  8.000000
 [2,]   -283.3504 7.806452  0.2833504  8.089802
 [3,]   -428.0112 7.709677  0.4280112  8.137689
 [4,]   -574.8516 7.612903  0.5748516  8.187755
 [5,]   -724.0433 7.516129  0.7240433  8.240172
 [6,]   -875.7785 7.419355  0.8757785  8.295133
... Many Rows ...
[29,]  -7186.0943 5.193548  7.1860943 12.379643
[30,]  -8389.5396 5.096774  8.3895396 13.486314
[31,] -11393.2010 5.000000 11.3932010 16.393201
>
```

Figure 83 is the same situation computed and plotted using the script in this essay[42].

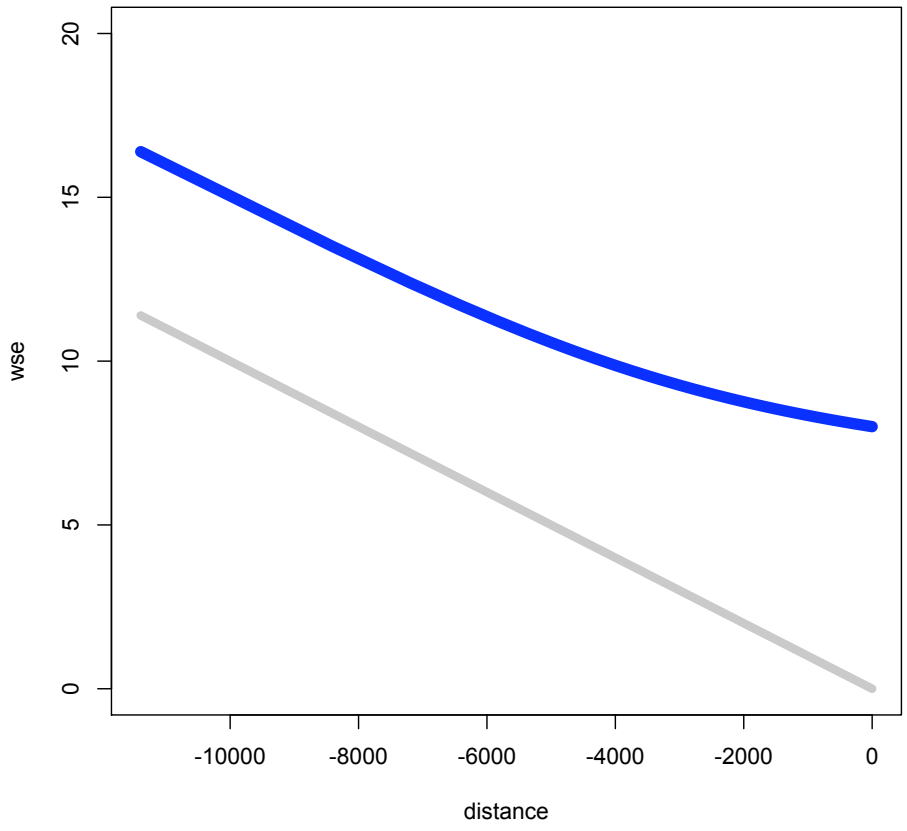

Figure 83. Backwater curve computed and plotted using **R**.

[42]With some additions that will be demonstrated in class!

12.2.4 Example 2 — Front-water curve

Figure 84 is another illustrative case. Here the water discharges into a horizontal channel at a rate of $1\frac{m^3}{sec}$ per meter width. Assuming Manning's $n \approx 0.01$ we wish to compute the profile downstream of the gate and determine if it will extend to the sharp edge[43].

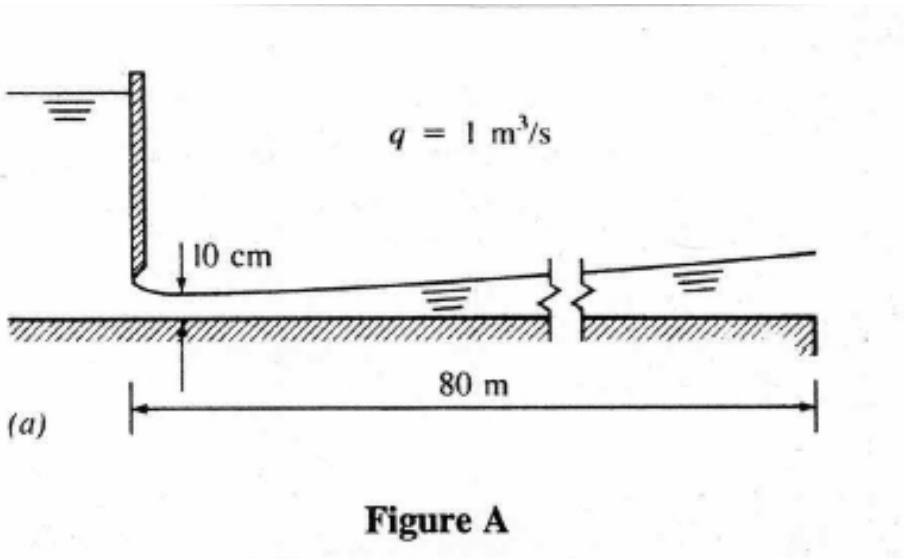

Figure 84. example caption.

We would need to know the critical depth for the section ($\approx 0.47 meters$), then compute the profile moving from the gate downstream (a frontwater curve with respect to the gate).

With the `backwater` function, all we really need to do is change the function arguments because it is already built for rectangular channels.

```
> # The function --- note the changes in parameter values!
> backwater(begin_depth=0.1,end_depth=0.47,how_many=20,
   + discharge=1,width=1,mannings_n=0.01,slope=0.000)
      distance  depth bse    wse
 [1,]  0.00000 0.1000   0 0.1000
... Rows ...
[13,] 79.00486 0.3405   0 0.3405
[14,] 82.82243 0.3590   0 0.3590
... Rows ...
```

Observe that the distance is now incrementing forward (by choice of begin and end depths). Figure 85 is the situation computed and plotted using the script in this essay.

[43] Obviously the profile will change a lot near the edge, but the question is will the profile continue to rise as depicted if the edge were further away?

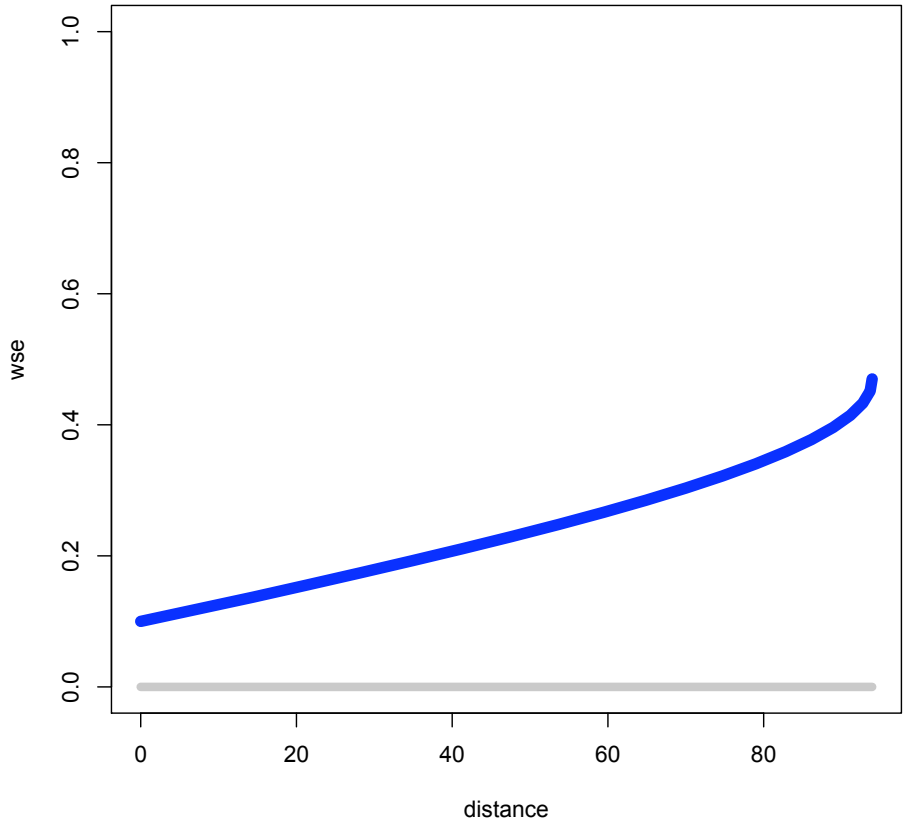

Figure 85. Frontwater curve computed and plotted using **R**.

Summary

This essay presented the variable step method to compute water surface profiles as a way to illustrate practical programming in **R**. Examples of a backwater and front-water curve were presented, as well as the actual code required to make the computations. This essay does not show how to make the plots — that information will be presented in class.

12.2.5 Finite Difference Methods — Fixed Space Method

1. Programming manual for: R version 2.8.0 (2008) The R Foundation for Statistical Computing ISBN 3-900051-07-0

2. Pages 79-115 from: Jobson, H.E. and Froelich, D. E. (1988). Basic Principles of Open-Channel Flow. U.S. Geologic Survey Open-File Report, OFR 88-707.

   [http://cleveland1.ce.ttu.edu/teaching/ce_5362/ce_5362_9.9/USGS_OFR_88-707.pdf]

12.3 Exercises

These exercises require you to modify the code. Be sure you save the code in a source file (will be demonstrated in class), so you can modify and save code with different function names. Also save a backup — **R** is not particularly kind in this respect (saving files where you can find them) so think ahead a little bit.

These exercises also assume you remember some concepts from your open channel flow class — there should be solved problems from that class you can use to test your work.

Finally, these exercises are intentionally pedagogical and you should not really use these codes for professional work unless you really understand their limitations.

1. Modify the **R** code for U.S. Customary units. Use the modified function to compute the water surface profile for a wide rectangular channel with Manning's $n = 0.022$, bottom slope $S_o = 0.0048$, and discharge per unit width of $\frac{Q}{W} = 50\frac{ft^3}{sec}$. Determine how far along the channel $x = L$ does it take for the flow depth to rise from a value of $y = 3.0ft$ to $y = 4.0ft$. Is the 4-ft depth position upstream or downstream of the 3-ft depth position? Plot the water surface profile.[44].

2. Modify the `area` and `perimeter` functions as well as the `backwater` functions for a trapezoidal channel. Apply the modified functions for a trapezoidal channel with a $[2:1]$ side slope, $3.5ft$ bottom width, 0.012 bed slope, that discharges from a reservoir at $Q = 185\frac{ft^3}{sec}$. You should assume the upstream value is at critical depth ($\approx 2.7ft$) and compute the profile to within 2% of normal depth. Plot the profile[45] .

Jaeger, C. (1957). Engineering Fluid Mechanics. St. Martin's Press. 529p.

Koutitas, C.G. (1983). Elements of Computational Hydraulics. Pentech Press, London 138p. ISBN 0-7273-0503-4

[44]Guideline: The profile should extend less than 400 feet for this problem — if your profiles are going further, there is probably something wrong with your functions or input values.

[45]Guideline: The profile should extend less than 100 feet for this problem — if your profiles are going further, there is probably something wrong with your functions or input values.

12.4 Unsteady Flow — St. Venant Equations

This essay provides a brief derivation of the St. Venant equations for one-dimensional (1-D) open channel flow. The equations were originally developed in the 1850's, so the concept is not very new. The tools have changed since that time; computational methods have greatly increased the utility of these equations.

In general, 1-D unsteady flow would be considered state-of-practice computation; every engineer would be expected to be able to make such calculations (albeit using software). 2-D computation is not routine, but within the realm of consulting practice — again using general purpose software. 3-D computation as of this writing (circa 2009) is still in the realm of state-of-art, and would not be within the capability a typical consulting firm.

12.4.1 The Computational Cell

The fundamental computational element is a computational cell or a reach. Figure 86 is a sketch of a portion of a channel. The left-most section is uphill (and upstream) of the right-most section. The section geometry is arbitrary, but is drawn to look like a channel cross section.

The length of the reach (distance between each section along the flow path) is Δx. The depth of liquid in the section is z, the width at the free surface is $B(z)$, the functional relationship established by the channel geometry. The flow into the reach on the upstream face is $Q - \partial Q/\partial x * \Delta x/2$. The flow out of the reach on the downstream face is $Q + \partial Q/\partial x * \Delta x/2$. The direction is strictly a sign convention and the development does not require flow in a single direction. The topographic slope is S_0, assumed relatively constant in each reach, but can vary between reaches.

12.4.2 Assumptions

The development of the unsteady flow equations uses several assumptions:

1. The pressure distribution at any section is hydrostatic — this assumption allows computation of pressure force as a function of depth.
2. Wavelengths are long relative to flow depth — this is called the shallow wave theory.
3. Channel slopes are small enough so that the topographic slope is roughly equal to the tangent of the angle formed by the channel bottom and the horizontal.
4. The flow is one-dimensional — this assumption implies that longitudinal dimension is large relative to cross sectional dimension. Generally river flows will meet this assumption, it fails in estuaries where the spatial dimensions (length and width) are roughly equal. Thus rivers that are hundreds of feet wide im-

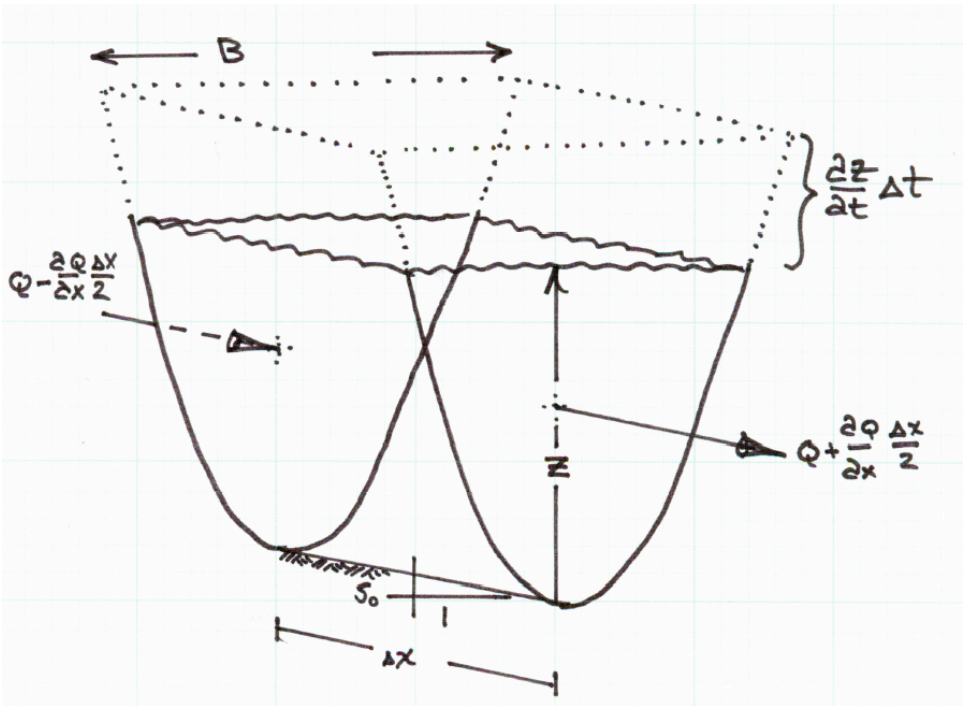

Figure 86. Reach/Computational Cell.

ply that reaches are miles long. If this assumption cannot be met, then 2-D methods are more appropriate.

5. Friction is modeled by Chezy or Manning's type empirical models. The particular friction model does not really matter, but historically these equations have used the friction slope concept as computed from one of these empirical models.

The tools that are used to build the equations are conservation of mass and linear momentum.

12.4.3 Conservation of Mass

The conservation of mass in the cell is the statement that mass entering and leaving the cell is balanced by the accumulation or lass of mass within the cell. For pedagogical clarity, this section goes through each part of a mass balance then assembles into a difference equation of interest.

Mass Entering: Mass enters from the left of the cell in our sketch. This direction only establishes a direction convention and negative flux means the arrow points in the direction opposite of that in the sketch. In the notation of the sketch mass entering in a short time interval is:

$$\dot{M}_{in} = \rho * (Q - \frac{\partial Q}{\partial x} * \frac{\Delta x}{2}) * \Delta t \tag{129}$$

where ρ is the fluid density. Notice that the mass flux is evaluated at the cell interface and not the centroid, while by convention ρ is assumed to be defined as an average cell property.

Mass Leaving: Mass leaves from the right of the cell in our sketch. In the notation of the sketch mass leaving is:

$$\dot{M}_{out} = \rho * (Q + \frac{\partial Q}{\partial x} * \frac{\Delta x}{2}) * \Delta t \tag{130}$$

Mass Accumulating: Mass accumulating within the reach is stored in the prism depicted in the sketch by the dashed lines. The product of density and prism volume is the mass added to (or removed from) storage.

The rise in water surface in a short time interval is $\frac{\partial z}{\partial t} * \Delta t$. The plan view area of the prism is $B(z) * \Delta x$. The product of these two terms is the mass added to storage, expressed as:

$$\dot{M}_{storage} = \rho * (\frac{\partial z}{\partial t} * \Delta t) * B(z) * \Delta x \tag{131}$$

Equating the accumulation to the net inflow produces

$$\rho * (\frac{\partial z}{\partial t} * \Delta t) * B(z) * \Delta x = \rho * (Q - \frac{\partial Q}{\partial x} * \frac{\Delta x}{2}) * \Delta t - \rho * (Q + \frac{\partial Q}{\partial x} * \frac{\Delta x}{2}) * \Delta t \tag{132}$$

This is the mass balance equation for the reach. If the flow is isothermal, and essentially incompressible then the density is a constant and can be removed from both sides of the equation.

$$(\frac{\partial z}{\partial t} * \Delta t) * B(z) * \Delta x = (Q - \frac{\partial Q}{\partial x} * \frac{\Delta x}{2}) * \Delta t - (Q + \frac{\partial Q}{\partial x} * \frac{\Delta x}{2}) * \Delta t \tag{133}$$

Rearranging the right hand side produces

$$(\frac{\partial z}{\partial t} * \Delta t) * B(z) * \Delta x = -\frac{\partial Q}{\partial x} * \frac{\Delta x}{2} * \Delta t - \frac{\partial Q}{\partial x} * \frac{\Delta x}{2} * \Delta t = -\frac{\partial Q}{\partial x} * \Delta x * \Delta t \tag{134}$$

Dividing both sides by $\Delta x * \Delta t$ yields

$$(\frac{\partial z}{\partial t}) * B(z) = -\frac{\partial Q}{\partial x} \tag{135}$$

This equation is the conventional representation of the conservation of mass in 1-D open channel flow. If the equation includes lateral inflow the equation is adjusted to

include this additional mass term. The usual lateral inflow is treated as a discharge per unit length added into the mass balance as expressed in Equation 160.

$$(\frac{\partial z}{\partial t}) * B(z) + \frac{\partial Q}{\partial x} = q \tag{136}$$

This last equation is one of the two equations that comprise the St. Venant equations. The other equation is developed from the conservation of linear momentum — the next section.

12.4.4 Conservation of Momentum

The conservation of momentum is the statement of the change in momentum in the reach is equal to the net momentum entering the reach plus the sum of the forces on the water in the reach. As in the mass balance, each component will be considered separately for pedagogical clarity.

Figure 87 is a sketch of the reach element under consideration, on some non-zero sloped surface.

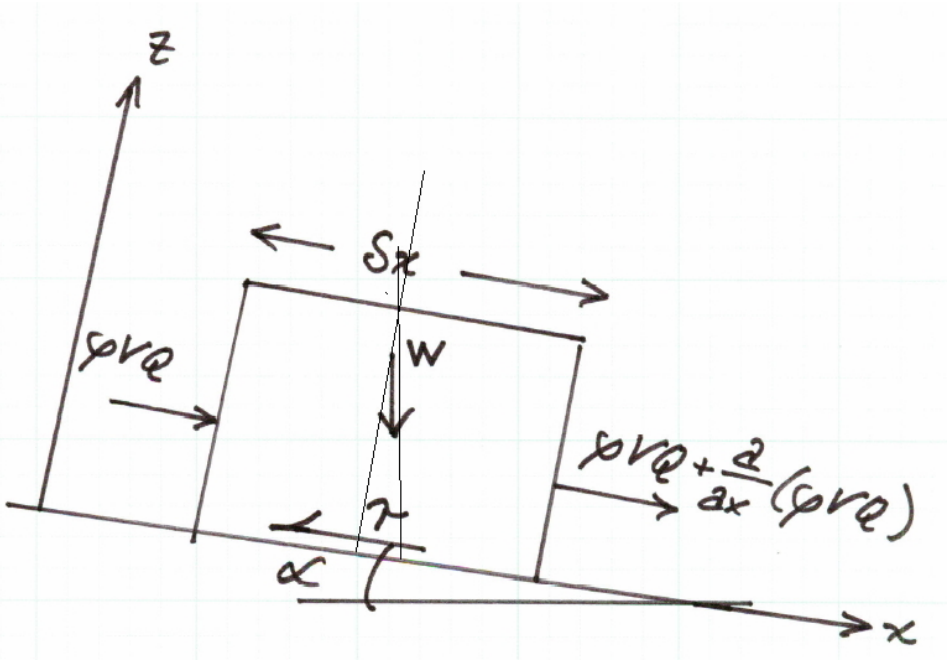

Figure 87. Equation of motion definition sketch.

Momentum Entering: Momentum entering on the left side of the sketch is

$$\rho * QV = \rho * V^2 A \tag{137}$$

Momentum Leaving: Momentum leaving on the right side of the sketch is

$$\rho * QV + \frac{\partial}{\partial x}(\rho * QV)\delta x = \rho * V^2 A + \frac{\partial}{\partial x}(\rho * \rho * V^2 A)\delta x \tag{138}$$

Momentum Accumulating: The momentum accumulating is the rate of change of linear momentum:

$$\frac{dL}{dt} = \frac{d\ (mV)}{dt} = \frac{\partial}{\partial t}(\rho * AV * \delta x) = \rho * \delta x \frac{\partial}{\partial t}(\rho * AV) \tag{139}$$

Forces on the liquid in the reach:

Gravity forces: The gravitational force on the element is the product of the mass in the element and the downslope component of acceleration.

The mass in the element is $\rho * A\delta x$

The x-component of acceleration is $g\ sin(\alpha)$, which is $\approx\ \ S_0$ for small values of α.

The resulting force of gravity is is the product of these two values:

$$F_g = \rho g * AS_0\ \delta x \tag{140}$$

Friction forces: Friction force is the product of the shear stress and the contact area. In the reach the contact area is the product of the reach length and average wetted perimeter.

$$F_{fr} = \tau * P_w * \delta x \tag{141}$$

where $P_w = A/R$, R is the hydraulic radius. A good approximation for shear stress in unsteady flow is $\tau = \rho g R S_f$. S_f is the slope of the energy grade line at some instant and is also called the friction slope. This slope can be empirically determined by a variety of models, typically Chezy's or Manning's equation is used. In either of these two models, we are using a STEADY FLOW equation of motion to mimic unsteady behavior — nothing wrong, and it is common practice, but this decision does limit the frequency response of the model (the ability to change fast — hence the shallow wave theory assumption!).

The resulting friction model is

$$F_{fr} = \rho g A S_f * \delta x \tag{142}$$

Pressure forces: [Set the equations, backfill discussion next version]

$$F_p = \int_A dF \tag{143}$$

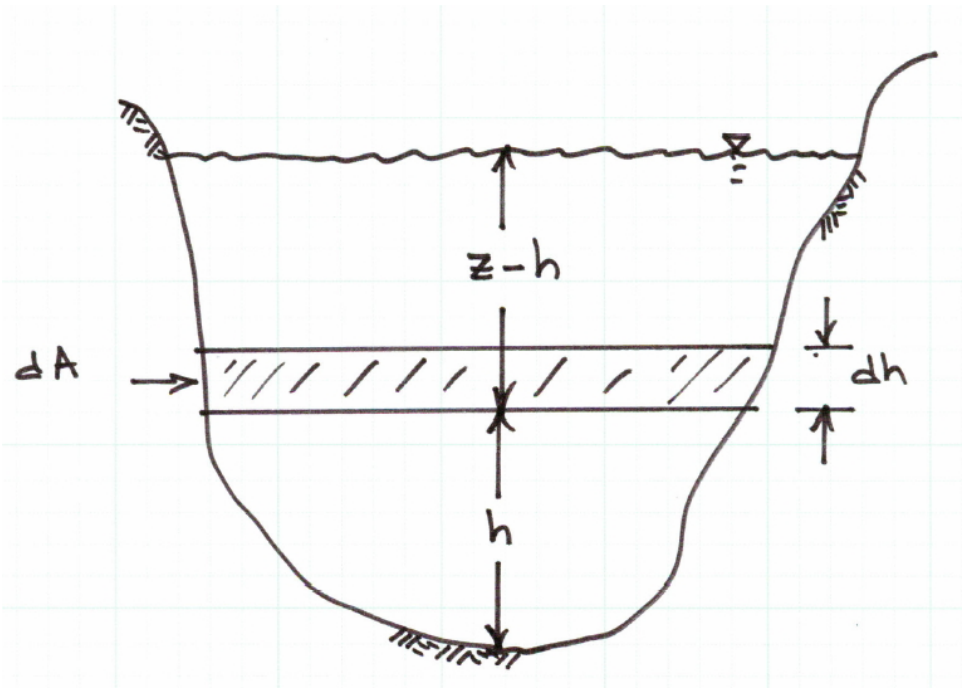

Figure 88. Pressure integral sketch.

$$dF = (z-h)\rho g \xi(h) dh \tag{144}$$

where $\xi(h)$ is the width of the panel at a given distance above the channel bottom (h) at any section.

$$F_{p\ net} = F_{p\ up} - F_{p\ down} \tag{145}$$

$$F_{p\ net} = F_p - (F_p + \frac{\partial F_p}{\partial x} * \delta x) = -\frac{\partial F_p}{\partial x} * \delta x \tag{146}$$

$$-\frac{\partial F_p}{\partial x} * \delta x = -\frac{\partial}{\partial x}[\int_0^Z \rho g (z-h)\xi(h) dh]\delta x \tag{147}$$

$$F_{p\ net} = -\rho g [\frac{\partial z}{\partial x}\int_0^Z \xi(h) dh + \int_0^Z (z-h)\xi(h)\frac{\partial \xi(h)}{\partial x} dh]\delta x \tag{148}$$

The first term integrates to the cross sectional area, the second term is the variation in pressure with position along the channel.

The other pressure force to consider is the bank force (the pressure force exerted by the banks on the element). This force is computed using the same type of integral structure except the order is swapped.

$$F_{p\ bank} = [\int_0^Z \rho g(z-h)\frac{\partial \xi(h)}{\partial x}\delta x]dh \tag{149}$$

Now we put everything together.

$$Momentum_{in} - Momentum_{out} + \sum F = \frac{d(mV)}{dt} \tag{150}$$

Substitution of the pieces:

$$Momentum_{in} - Momentum_{out} + F_{p\ net} + F_{bank} + F_{gravity} - F_{friction} = \frac{d(mV)}{dt} \tag{151}$$

Now when the expressions for each expressions for each part

$$\begin{aligned} \rho * V^2A - \rho * V^2A - \frac{\partial}{\partial x}(\rho * V^2A)\delta x \\ -\rho g\frac{\partial z}{\partial x}\int_0^Z \xi(h)dh\delta x - [\int_0^Z \rho g(z-h)\frac{\partial \xi(h)}{\partial x}dh]\delta x \\ +[\int_0^Z \rho g(z-h)\frac{\partial \xi(h)}{\partial x}\delta x]dh \\ +\rho g * AS_0\ \delta x \\ -(\rho g R S_f * \delta x) \\ = \rho * \delta x\frac{\partial}{\partial t}(\rho * g * AV) \end{aligned} \tag{152}$$

Each row of Equation 152 is in order:

1. Net momentum entering the reach.
2. Pressure force differential at the end sections.
3. Pressure force on the channel sides.
4. Gravitational force.
5. Frictional force opposing flow.
6. Total acceleration in the reach (change in linear momentum).

Canceling terms and dividing by $\rho\delta x$ (isothermal, incompressible flow; reach has finite length) Equation 152 simplifies to

$$-\frac{\partial}{\partial x}(V^2A) - g\frac{\partial z}{\partial x}\int_0^Z \xi(h)dh + g * AS_0\ - (gRS_f*) = \frac{\partial}{\partial t}(g * AV) \tag{153}$$

The second term integral is the sectional flow area, so it simplifies to

$$-\frac{\partial}{\partial x}(V^2A) - g\frac{\partial z}{\partial x}A + gAS_0 \ - gAS_f = \frac{\partial}{\partial t}(AV) \tag{154}$$

The term with the square of mean section velocity is expanded by the chain rule, and using continunity becomes (notice the convective acceleration term from the change in area with time)

$$\frac{\partial}{\partial t}(AV) = A\frac{\partial V}{\partial t} + V\frac{\partial A}{\partial t} = A\frac{\partial V}{\partial t} - VA\frac{\partial V}{\partial x} - V^2\frac{\partial A}{\partial x} \tag{155}$$

Now expand and construct

$$-V^2\frac{\partial A}{\partial x} - 2VA\frac{\partial V}{\partial x} - gA\frac{\partial z}{\partial x} + gA(S_0 - S_f) = A\frac{\partial V}{\partial t} - VA\frac{\partial V}{\partial x} - V^2\frac{\partial A}{\partial x} \tag{156}$$

Cancel common terms and simplify

$$-VA\frac{\partial V}{\partial x} - gA\frac{\partial z}{\partial x} + gA(S_0 - S_f) = A\frac{\partial V}{\partial t} \tag{157}$$

Equation 161 is the final form of the momentum equation for practical use. It will be rearranged in the remainder of this essay to fit some other purposes, but this is the expression of momentum in the channel reach.

Divide by gA and obtain

$$-\frac{V}{g}\frac{\partial V}{\partial x} - \frac{\partial z}{\partial x} + (S_0 - S_f) = \frac{1}{g}\frac{\partial V}{\partial t} \tag{158}$$

Rearrange

$$S_f = S_0 - \frac{\partial z}{\partial x} - \frac{V}{g}\frac{\partial V}{\partial x} - \frac{1}{g}\frac{\partial V}{\partial t} \tag{159}$$

Now consider typical flow regimes.

1. The first two terms (from left to right) are uniform flow, this is an algebraic equation.

2. If the first four terms are in effect, we have gradually varied flow; an ordinary differential equation.
3. If all terms are in effect, the have the dynamic flow (shallow wave) conditions; a partial differential equation.

The pair of equations,

$$(\frac{\partial z}{\partial t}) * B(z) + \frac{\partial Q}{\partial x} = q \tag{160}$$

$$S_0 - S_f - \frac{\partial z}{\partial x} - \frac{V}{g}\frac{\partial V}{\partial x} - \frac{1}{g}\frac{\partial V}{\partial t} = 0 \tag{161}$$

are called the St. Venant equations and comprise a coupled hyperbolic differential equation system.

Solutions ((z,t) and (V,t) functions) are found by a variety of methods including finite difference, finite element, finite volume, and characteristics methods.

In this course we will study a simple finite-difference scheme to gain some familarity with building solutions, then use prepared tools (SWMM) for more practical problems.

Cunge, J.A., Holly, F.M., Verwey, A. 1980. "Practical Aspects of Computational River Hydraulics." Pittman Publishing Inc. , Boston, MA. pp. 7-50 in

12.4.5 Finite-Difference Methods — Lax Scheme

12.4.6 Method of Characteristics for Boundary Conditions

12.4.7 Rudimentary R Script

A script that implements these concepts is listed on the next several pages.

```
# hydraulic functions
# depth == flow depth
# bottom == bottom width of trapezoidal channel
# side == side slope (same value both sides) of trapezoidal channel
# bt == computed topwidth
# ar == flow area, used in fd update
# wp == wetted perimeter, used in fd update
# depth-topwidth function
bt <- function(depth,bottom,side)
 # tested 12MAR2015 TGC
```

```
{
 topwidth <- (bottom + 2.0*side*depth);
 return(topwidth);
}
# depth area function
ar <- function(depth,bottom,side)
 # tested 12MAR2015 TGC
{
 area <- (depth*(bottom+side*depth));
 return(area)
}
# depth perimeter
wp <- function(depth,bottom,side)
 # tested 12MAR2015 TGC
{
 perimeter <- (bottom+2.0*depth*sqrt(1.0+side*side));
 return(perimeter)
}
# printing functions
writenow <- function(t,dt,y,v,b0,s)
{
message("__________")
message("Time = ",t," seconds.","Time step lenght = ",dt," seconds ")
aa <- ar(y,b0,s);
qq <- aa*v
brb <- bt(y,b0,s)
ww <- wp(y,b0,s)
message("--in write now----")
print(cbind(y,qq,v,qq,brb,ww,zz))
message("-----------------")
return()
}

###### Plotting Functions ####################
plotnow<-function(t,x,y,v){
 mainlabel=c("Flow Depth at time = ",t," seconds")
plot(x,y,main=mainlabel,xlab="distance (m)",ylab="depth (m)",xlim=c(0,30000),ylim=c(0
lines(x,v,lwd=3,col="red",type="l")
}


###### Finite Difference Functions ############
finitedifference<-function(){
r <- 0.5*dt/dx;
###### LEFT BOUNDARY ##########
```

```
## MODIFY TO A REFLECTION BOUNDARY ##
## THEN SUPPLY DISCHARGE TO THE FIRST CELL AND USE CONTINUNITY TO FIND DEPTH ##
##  qq is the interpolated hydrograph flow value
#debug message("left boundary ",qq$y)
#debug message("left boundary ",y[1])
#########################################################
vp[1] <<- qq$y/ar(y[1],b0,s);
ab <- ar(y[2],b0,s);
bb <- bt(y[2],b0,s);
cb <- sqrt(g*bb/ab);
rb <- ab/wp(y[2],b0,s);
sfb <- (mn2*v[2]*v[2])/(rb^(1.333));
#debug print(cbind(ab,bb,cb,rb,sfb));
cn <- v[2] -cb*y[2]+ g*(s0-sfb)*dt;
yp[1] <<- (vp[1] - cn )/cb;
#########################################################
### gives y,v at location 1 time level k+1
#debug message("--left bndry ----")
#debug print(cbind(y,yp,v,vp,sfb,cn,rb));
######## RIGHT BOUNDARY ##############
## MODIFY TO A FREE OUTFALL
## USE A WEIR EQUATION BASED ON SOME REFERENCE DEPTH
vp[nn]<<- (y[n]-yd)*sqrt(9.8*y[n]);
## USE CHARACTERISTIC LINE TO FIND ASSOCIATED DEPTH
 aa <- ar(y[n],b0,s);
 ba <- bt(y[n],b0,s);
 ca <- sqrt(g*ba/aa);
 ra <- aa/wp(y[n],b0,s);
 sfa <- (mn2*v[n]*v[n])/(ra^(4.0/3.0));
 cp <- v[n] + ca*y[n]+g*(s0-sfa)*dt;
 yp[nn] <<- (cp - vp[nn])/ca;
#  ## fixed stage
# yp[nn] <<- yd ;
# aa <- ar(y[n],b0,s);
# ba <- bt(y[n],b0,s);
# ca <- sqrt(g*ba/aa);
# ra <- aa/wp(y[n],b0,s);
# sfa <- (mn2*v[n]*v[n])/(ra^(4.0/3.0));
# cp <- v[n] + ca*y[n]+g*(s0-sfa)*dt;
# vp[nn] <<- (cp - ca*yp[nn]); # check sign
#   message("--right bndry ----")
# print(cbind(y,yp,v,vp));

# reflection boundary, find depth along a characteristic
# vp[nn] <<-0 ;
```

```
# aa <- ar(y[n],b0,s);
# ba <- bt(y[n],b0,s);
# ca <- sqrt(g*ba/aa);
# ra <- aa/wp(y[n],b0,s);
# sfa <- (mn2*v[n]*v[n])/(ra^(4.0/3.0));
# cp <- v[n] + ca*y[n]+g*(s0-sfa)*dt;
# yp[nn] <<- (cp - vp[nn])/ca;
##print(cbind(y,yp,v,vp));
######## INTERIOR NODES AND REACHES ###############
# loop through the interior nodes
for (i in 2:n){ # begin interior node loop scope
aa <- ar(y[i-1],b0,s);
ba <- bt(y[i-1],b0,s);
pa <- wp(y[i-1],b0,s);
ra <- aa/pa;
sfa <- (mn2*v[i-1]*v[i-1])/(ra^(4.0/3.0));
ab <- ar(y[i+1],b0,s);
bb <- bt(y[i+1],b0,s);
pb <- wp(y[i+1],b0,s);
rb <- ab/pb;
sfb <- (mn2*v[i+1]*v[i+1])/(rb^(4.0/3.0));
# need averages of sf, hydraulic depth
dm <- 0.5*(aa/ba + ab/bb);
sfm <- 0.5*(sfa+sfb);
vm <- 0.5*(v[i-1]+v[i+1]);
ym <- 0.5*(y[i-1]+y[i+1]);
# update momentum
# note the double <<, this structure forces the
# value to be global and accessible
# to other functions when the script is run
vp[i] <<- vm -r*g*(y[i+1] - y[i-1]) -r*vm*(v[i+1] - v[i-1]) + g*dt*(s0-sfm);
# update depth
yp[i] <<- ym - r*dm*(v[i+1] - v[i-1]) -r*vm*(y[i+1] - y[i-1]);
} # end of interior node loop scope
} # end of function scope

###### Solution Update Function ############
update<-function(y,yp,v,vp){
y <<- yp;
v <<- vp;
return()
}
### NEED ADAPTIVE TIME STEPPING FOR THE FLOOD WAVE
###### Adaptive Time Step Functions ############
## here is where we do adaptive time stepping
```

```
bestdt<-function(y,v){
  bestdt <- dt # start with current time step
  for (i in 1:nn){
    a <- ar(y[i],b0,s);
    b <- bt(y[i],b0,s);
    c <- sqrt(g*a/b);
    dtn <- dx/abs((v[i])+c)
    # now test
    if(dtn <= bestdt){bestdt <- dtn}
  } # end loop scope
  dt <<- bestdt
} #end bestdt function
###### Solution Update Function ############
update<-function(y,yp,v,vp){
  y <<- yp;
  v <<- vp;
  return()
}
```

12.4.8 Example 1 – Flood wave in a channel

[Present the problem, and important features – then to solution listed below. Uses the library]

```
# main program for st. venant
rm(list=ls())
setwd("~/Dropbox/CE5361-2015-1/Project2/Project2-Problem2")
# clear workspace and set directory
source('~/Dropbox/CE5361-2015-1/Project2/Project2-Problem2/Project2.2.Lib.R')

###### Problem Constants #######
# these are constants that define the problem
# change for different problems
# a good habit is to assign constants to names so the
# program is readable by people in a few years
g <-9.81 # gravitational acceleration, obviously SI units
n <- 29 # number of reaches
#q0 <- 110 # initial discharge
q0 <- 1.1 # initial discharge
#yd <- 3.069 # initial flow depth in the model
yd <- 1.000 # initial flow depth in the model
#yu <- 3.069 # upstream constant depth
yu <- 1.1 # upstream constant depth
# mn <- 0.013 # Manning's n
```

```
mn <- 0.02 # Manning's n
# b0 <- 20 # bottom width
b0 <- 5 # bottom width
## modify for problem 1 s0 <- 0.0001 # longitudinal slope (along direction of flow)
s0 <- 0.0000001 # longitudinal slope (along direction of flow)
#s  <- 2.0 # side slope (passed to calls to hydraulic variables)
s  <- 0.0 # side slope (passed to calls to hydraulic variables)
l  <- 30000.0 # total length (the lenght of computational domain)
tmax <- 640000 # total simulation time in seconds
iprt <- 8 # print every iprt time steps
nn <- n+1 # how many nodes, will jack with boundaries later
mn2 <- mn*mn # Manning's n squared, will appear a lot.
a <- ar(yd,b0,s) # flow area at beginning of time
v0 <- q0/a # initial velocity
######## Here we build vectors ###############
y <- numeric(nn) # create nn elements of vector y
yp <- numeric(nn) # updates go in this vector, same length as y
v <- numeric(nn) # create nn elements of vecotr v
vp <-numeric(nn) # updates go in this vector, same length and v
ytmp <-numeric(nn)
vtmp <-numeric(nn)
y <- rep(yd,nn) # populate y with nn things, each thing has value yd
v <- rep(v0,nn) # populate v with nn things, each thing has value v0
b <- bt(yd,b0,s) # topwidth at beginning
c <- sqrt(g*a/b) # celerity at initial conditions
###
hydrograph <- numeric(1)
dummy<- read.csv(file="hydrograph.csv",header=F)
elapsedtime <- dummy$V1; # forst column of hydrograph is time
hydrograph <- dummy$V2; # second column of hydrograph is flow
print(cbind(c))
dx <- l/n # delta x, length of a reach
xx <- dx*seq(1:nn)-1000  # Spatial locations of nodes, used for plotting
zz <- 30 - s0*xx # bottom channel elevation
dt <- dx/(v0 + c) # the time step that satisfies the courant condtions
kmax <- round(tmax/dt)  # set maximum number of time steps
print(cbind(dx,dt))



### Run the simulation                  ###
k <- 0 # time counter
t <- 0.0 # elapsed time
pdf("junk2.2.plot.pdf") # graphics device for plots -- plotnow() sends data to plot
writenow(t,dt,y,v,b0,s)  # Write initial conditions
```

```
plotnow(t,xx,y,v)
####### BEGIN TIME STEPPING ########
message("kmax =", kmax)
for (itime in 1:kmax){
## put in the hydrograph here -- use approx function to interpolate
## we are after the second value qq$y in the approx function
qq <<- approx(elapsedtime,hydrograph,t)
print (qq$x);
print (qq$y);
#debug print(qq$y)
##############  NEED ADAPTIVE TIME STEPPING FOR THE HYDROGRAPH ROUTING #############
bestdt(y,v);
############## THIS IS A HACK TO GET STABILITY #################
# halve the time step
dt <- dt/16
finitedifference(); # Finite difference a single time step
#update(ytmp,yp,vtmp,vp); # Update vectors
#update(y,yp,v,vp); # Update vectors
#bestdt(yp,vp)
message("dt now = ",dt);
#finitedifference(); # Finite difference a single time step
ytmp <- (yp+ytmp)/2;
vtmp <- (vp+vtmp)/2;
update(y,yp,v,vp); # Update vectors
message(" dt is ",dt);
t <- t+dt; # Increment simulation time
k <- k+1; # Increment loop counter
if (k%%iprt == 0){writenow(t,dt,y,v,b0,s)}; # Write current conditions every iprt tim
if (k%%iprt == 0){plotnow(t,xx[seq(1,nn,1)],(y[seq(1,nn,1)]),(v[seq(1,nn,1)]))}; # Pl
# reset the time step
############ UNHACK TO KEEP ADAPTIVE TIME STEPPING WORKING ###########
dt <- 16*dt
}
dev.off() # disconnect the pdf file.
```

12.4.9 Example 2 – Long waves in a tidal-influenced channel

12.5 Quasi-2D Methods

[Use Cleveland and Botkins report to make explain – then make adaptation]

12.6 Exercises

13 Steady Water Surface Profiles

14 Unsteady Open Channel Flow

15 Using Method of Characteristics for Boundary Conditions

16 Flow in Porous Materials

Flow in porous media is a topic that appears in many branches of engineering and science, e.g., ground water hydrology, reservoir engineering, soil science, soil mechanics, and chemical engineering (filtration). The aquifer, which is the porous medium domain of the hydrologist, or the oil reservoir, which is the porous medium of the petroleum engineer are typical examples.

Figure 89 is a sketch of different aquifer classifications. A confined aquifer (pressure aquifer) is one bounded above and below by impermeable formations. In a well penetrating such an aquifer, the water level will rise above the base of the confining formation. Water levels in wells that sample a certain aquifer define an imaginary surface called the piezometric surface.

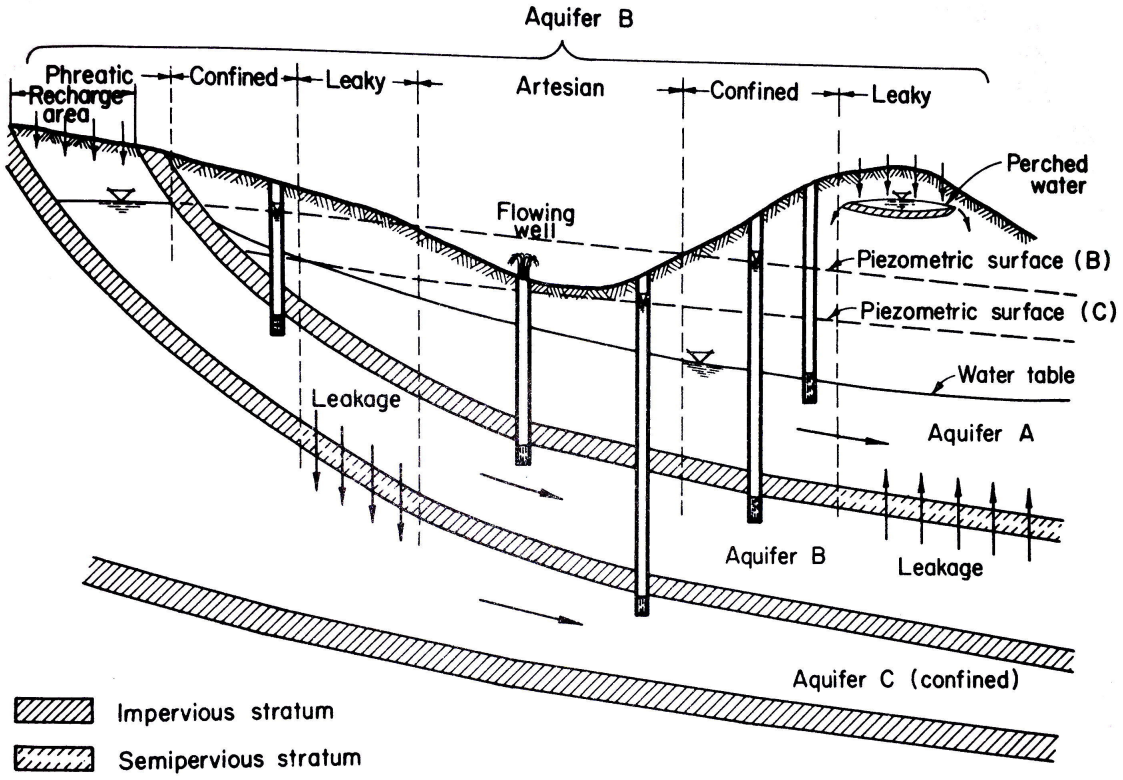

Figure 89. Aquifer classifications.

An unconfined aquifer (water table aquifer; phreatic aquifer) is one with the water table as its upper boundary. The classifications are important because the equations of motion are different in different kinds of aquifers.

16.1 Storage

Storativity of an aquifer is the relationship between changes in head within the aquifer and the quantity of water stored in the aquifer. Figure 90 is a sketch showing the storage process in a confined, and unconfined aquifer.

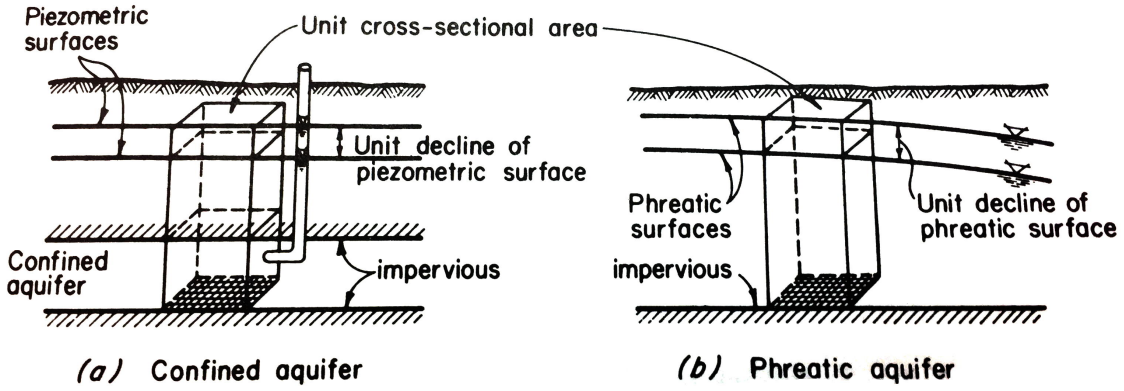

Figure 90. Illustrative sketches of definition of storage coefficient.

The mechanism of storage is different for confined and unconfined aquifers. In a confined aquifer the water is stored or released by compression and decompression of water and the solid matrix (like a sponge squeezed while wrapped in plastic wrap). In an unconfined aquifer the water is stored or released from the pore space when the water table elevation changes.

The storage coefficient (confined) or specific yield (unconfined) is the volume of water added to (or removed from) storage per unit area of aquifer per unit change in head. The usual symbols are S, and S_y.

16.2 Permeability

Permeability is the material property that relates the resistance of flow through the porous medium to the hydraulic gradient.

16.3 Head Loss Models

Darcy's law (a linear flux model) is the head loss model used for porous media flow. Equation 162 is Darcy's law expressed as a head loss model.

$$h_L = \frac{QL}{KA} \tag{162}$$

where Q is the discharge in the aquifer, L is the length in the flow direction, A is the cross sectional area of aquifer (pore space and solid phase), K is the hydraulic conductivity.[46]

A more useful (for computation) form of the head loss model, is to express it [the loss equation] as an equation of motion as in Equation 163.

$$Q = -KA\frac{\partial h}{\partial x} \tag{163}$$

where $-\frac{\partial h}{\partial x}$ is the hydraulic gradient (slope of the hydraulic grade line) in the aquifer.

Figure 91 is a diagram that illustrates the relationships expressed by Darcy's law.

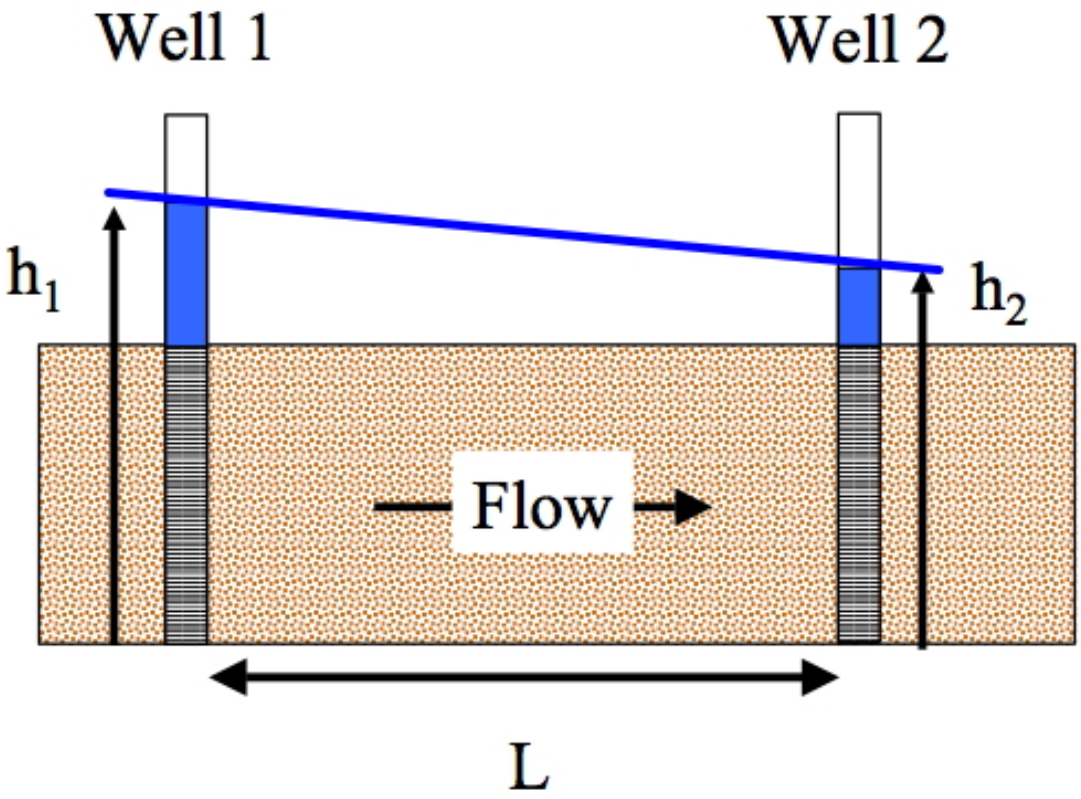

Figure 91. Schematic diagram of unidirectional flow in a generic aquifer, showing heads in two measuring wells located distance L apart..

The cross-sectional flow area, A, is the product of height of the aquifer block and its width [in this case the width is into the plane of the paper]. The distance between two measurement points is L. The head at the two points is h_1 and h_2. The gradient of head, $\frac{\partial h}{\partial x}$, is $\frac{h_2-h_1}{L}$. The hydraulic gradient is $-\frac{\partial h}{\partial x}$, is $\frac{h_1-h_2}{L}$. Finally Darcy's law (for the drawing) is $Q = KA\frac{h_1-h_2}{L}$.

[46] also called the permeability

16.4 Confined Aquifer Flow

Using Figure 91 as a starting point, we can develop a computational model of flow in a confined aquifer. Let's decide that the distance L in the figure is going to be divided into a series of connected, small blocks. The flow direction in the figure will be declared the x direction, the depth into the drawing is declared the y direction, and the height of the block is declared the z direction.

Figure 94 is a diagram of one such small block.

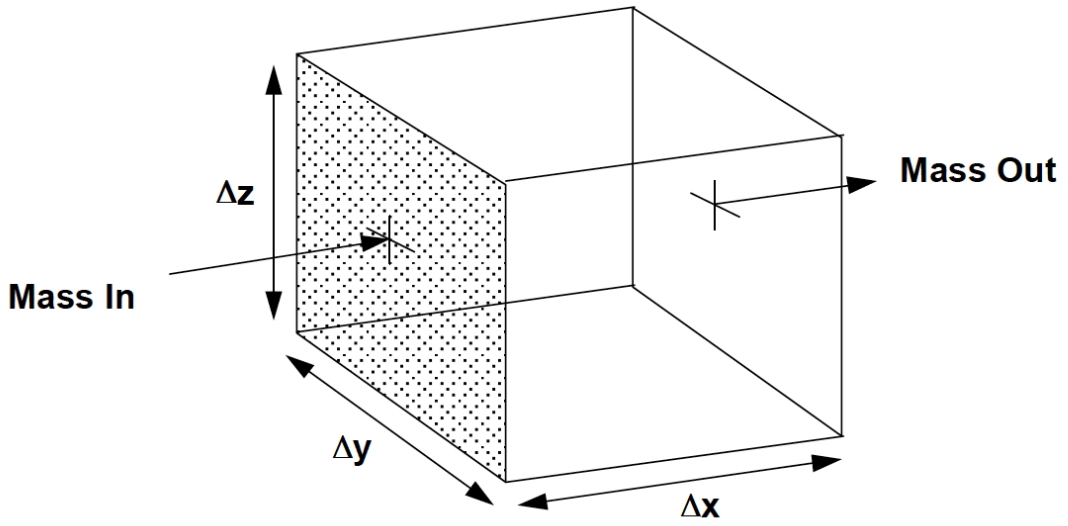

Figure 92. Single computational cell definition sketch.

Using this diagram we can now develop a set of expressions for the cell volume, solids volume in the cell, pore volume in the cell (where water actually can flow), and solids mass.

$$V_{cell} = \Delta x \times \Delta y \times \Delta z \tag{164}$$

$$V_{soild} = (1 - \omega)\Delta x \times \Delta y \times \Delta z \tag{165}$$

$$V_{pore} = \omega \Delta x \times \Delta y \times \Delta z \tag{166}$$

$$M_{solid} = \rho_s (1 - \omega)\Delta x \times \Delta y \times \Delta z \tag{167}$$

Next write a mass balance for water in the cell;

$$\frac{dM_{water}}{dt} = M_{Inflow} - M_{Outlfow} \tag{168}$$

The left side of the expression is simply the storage term, and in the context of storage coefficients and aquifer head is replaced by

$$\frac{dM_{water}}{dt} \mid_{cell} = \rho_w S_s \Delta x \Delta y \Delta z \frac{\partial h_i}{\partial t} \tag{169}$$

where h_i is the head in the $i-$th cell.

The right hand side of the expression is based on writing Darcy's law for the cell, using values in adjacent (hydraulically connected) cells.

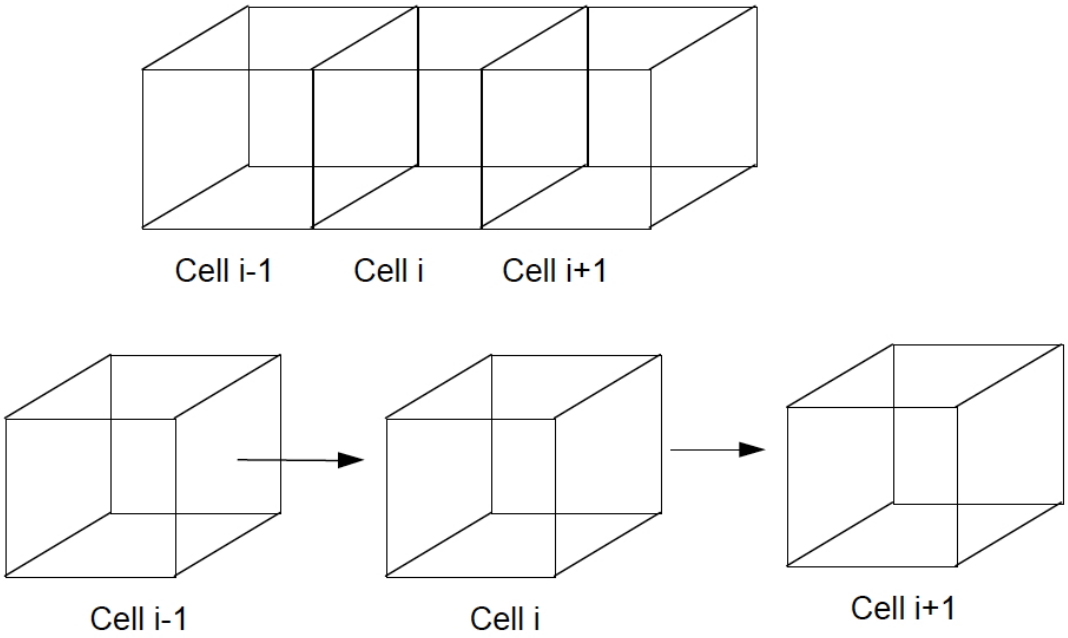

Figure 93. Multiple computational cell definition sketch.

Figure 95 is a sketch showing three such cells. The $i-$th cell is the cell of interest, the cell to the left is cell ID $i-1$, and the cell to the right is cell ID $i+1$.

We now write Darcy's law for each face of cell i, treating the head in each of the cell centers as if they were the sampling wells of Figure 91.[47]

Darcy's law for the left face is

$$M_{Inflow} = Q_{left} = \rho_w K \Delta y \Delta z \frac{h_{i-1} - h_i}{\Delta x} \tag{170}$$

[47] In the context of Figure 91, the cell face is halfway between the two wells; the cell centers are at the wells.

Similarly for the right face,

$$M_{Outflow} = Q_{right} = \rho_w K \Delta y \Delta z \frac{h_i - h_{i+1}}{\Delta x} \tag{171}$$

Now combine these together in the mass balance

$$\rho_w S_s \Delta x \Delta y \Delta z \frac{\partial h_i}{\partial t} = (\rho_w K \Delta y \Delta z \frac{h_{i-1} - h_i}{\Delta x}) - (\rho_w K \Delta y \Delta z \frac{h_i - h_{i+1}}{\Delta x}) \tag{172}$$

Next divide by the water density ρ_w,

$$S_s \Delta x \Delta y \Delta z \frac{\partial h_i}{\partial t} = (K \Delta y \Delta z \frac{h_{i-1} - h_i}{\Delta x}) - (K \Delta y \Delta z \frac{h_i - h_{i+1}}{\Delta x}) \tag{173}$$

The divide by cell width Δy,

$$S_s \Delta x \Delta z \frac{\partial h_i}{\partial t} = (K \Delta z \frac{h_{i-1} - h_i}{\Delta x}) - (K \Delta z \frac{h_i - h_{i+1}}{\Delta x}) \tag{174}$$

Rewrite the right hand side into gradient of head form

$$S_s \Delta x \Delta z \frac{\partial h_i}{\partial t} = (K \Delta z \frac{h_{i+1} - h_i}{\Delta x}) - (K \Delta z \frac{h_i - h_{i-1}}{\Delta x}) = K \Delta z \frac{\partial h}{\partial x} \mid_{i \rightarrow i+1} - K \Delta z \frac{\partial h}{\partial x} \mid_{i-1 \rightarrow i} \tag{175}$$

Divide by cell distance, Δx,

$$S_s \Delta z \frac{\partial h_i}{\partial t} = \frac{K \Delta z \frac{\partial h}{\partial x} \mid_{i \rightarrow i+1} - K \Delta z \frac{\partial h}{\partial x} \mid_{i-1 \rightarrow i}}{\Delta x} \tag{176}$$

Take limit as $\Delta x \rightarrow 0$,

$$S_s \Delta z \frac{\partial h}{\partial t} = \frac{\partial}{\partial x}(K \Delta z \frac{\partial h}{\partial x}) \tag{177}$$

Finally, stipulate that $S_s \Delta z = S$, the storage coefficient (for confined aquifer), and define the aquifer transmissivity as $K \Delta z = T$ and we have performed a back-handed way to get the partial differential equation of aquifer flow.

$$S \frac{\partial h}{\partial t} = \frac{\partial}{\partial x}(T \frac{\partial h}{\partial x}) \tag{178}$$

Ironically, the analysis actually provides an algorithm to approximate head in the aquifer at Equation 188 – which is the subject of the next Chapter.

16.5 Unconfined Aquifer Flow

17 Steady Groundwater Flow

17.1 Confined Aquifer Flow

Using Figure 91 as a starting point, we can develop a computational model of flow in a confined aquifer. Let's decide that the distance L in the figure is going to be divided into a series of connected, small blocks. The flow direction in the figure will be declared the x direction, the depth into the drawing is declared the y direction, and the height of the block is declared the z direction.

Figure 94 is a diagram of one such small block.

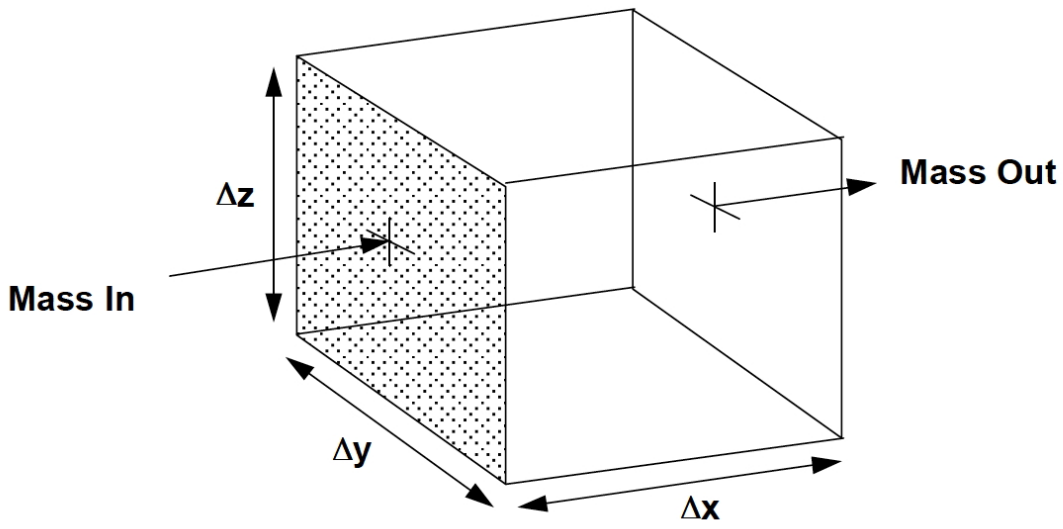

Figure 94. Single computational cell definition sketch.

Using this diagram we can now develop a set of expressions for the cell volume, solids volume in the cell, pore volume in the cell (where water actually can flow), and solids mass.

$$V_{cell} = \Delta x \times \Delta y \times \Delta z \tag{179}$$

$$V_{soild} = (1 - \omega)\Delta x \times \Delta y \times \Delta z \tag{180}$$

$$V_{pore} = \omega \Delta x \times \Delta y \times \Delta z \tag{181}$$

$$M_{solid} = \rho_s(1-\omega)\Delta x \times \Delta y \times \Delta z \tag{182}$$

Next write a mass balance for water in the cell;

$$\frac{dM_{water}}{dt} = M_{Inflow} - M_{Outlfow} \tag{183}$$

The left side of the expression is simply the storage term, and in the context of storage coefficients and aquifer head is replaced by

$$\frac{dM_{water}}{dt} \mid_{cell} = \rho_w S_s \Delta x \Delta y \Delta z \frac{\partial h_i}{\partial t} \tag{184}$$

where h_i is the head in the $i-$th cell.

The right hand side of the expression is based on writing Darcy's law for the cell, using values in adjacent (hydraulically connected) cells.

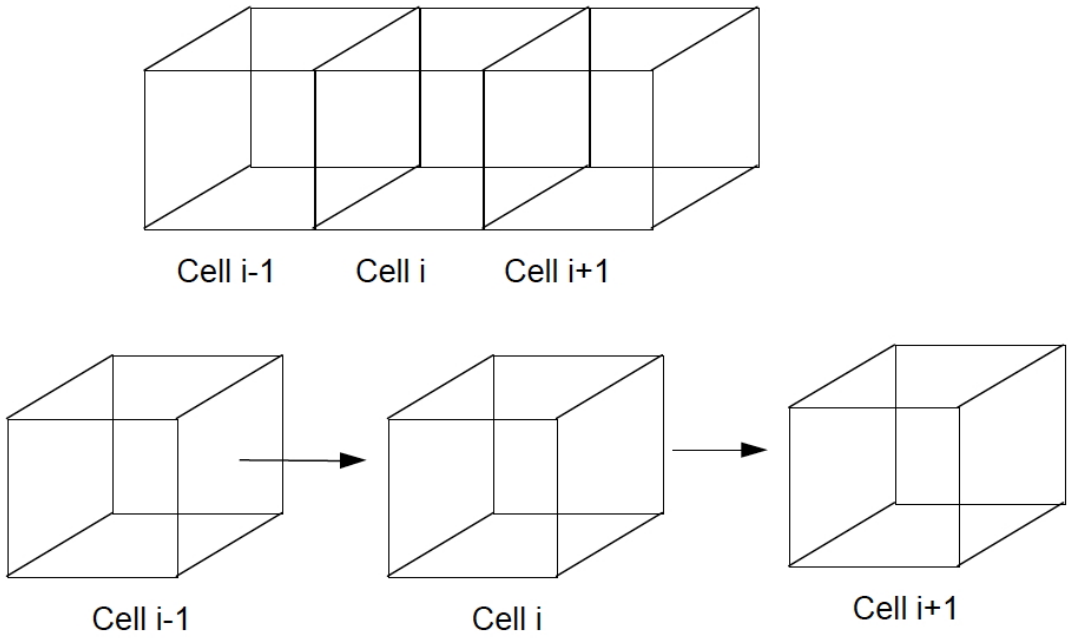

Figure 95. Multiple computational cell definition sketch.

Figure 95 is a sketch showing three such cells. The $i-$th cell is the cell of interest, the cell to the left is cell ID $i-1$, and the cell to the right is cell ID $i+1$.

We now write Darcy's law for each face of cell i, treating the head in each of the cell centers as if they were the sampling wells of Figure 91.[48]

Darcy's law for the left face is

[48] In the context of Figure 91, the cell face is halfway between the two wells; the cell centers are at the wells.

$$M_{Inflow} = Q_{left} = \rho_w K \Delta y \Delta z \frac{h_{i-1} - h_i}{\Delta x} \tag{185}$$

Similarly for the right face,

$$M_{Outflow} = Q_{right} = \rho_w K \Delta y \Delta z \frac{h_i - h_{i+1}}{\Delta x} \tag{186}$$

Now combine these together in the mass balance

$$\rho_w S_s \Delta x \Delta y \Delta z \frac{\partial h_i}{\partial t} = (\rho_w K \Delta y \Delta z \frac{h_{i-1} - h_i}{\Delta x}) - (\rho_w K \Delta y \Delta z \frac{h_i - h_{i+1}}{\Delta x}) \tag{187}$$

Next divide by the water density ρ_w,

$$S_s \Delta x \Delta y \Delta z \frac{\partial h_i}{\partial t} = (K \Delta y \Delta z \frac{h_{i-1} - h_i}{\Delta x}) - (K \Delta y \Delta z \frac{h_i - h_{i+1}}{\Delta x}) \tag{188}$$

The divide by cell width Δy,

$$S_s \Delta x \Delta z \frac{\partial h_i}{\partial t} = (K \Delta z \frac{h_{i-1} - h_i}{\Delta x}) - (K \Delta z \frac{h_i - h_{i+1}}{\Delta x}) \tag{189}$$

Rewrite the right hand side into gradient of head form

$$S_s \Delta x \Delta z \frac{\partial h_i}{\partial t} = (K \Delta z \frac{h_{i+1} - h_i}{\Delta x}) - (K \Delta z \frac{h_i - h_{i-1}}{\Delta x}) = K \Delta z \frac{\partial h}{\partial x} \mid_{i \rightarrow i+1} - K \Delta z \frac{\partial h}{\partial x} \mid_{i-1 \rightarrow i} \tag{190}$$

Divide by cell distance, Δx,

$$S_s \Delta z \frac{\partial h_i}{\partial t} = \frac{K \Delta z \frac{\partial h}{\partial x} \mid_{i \rightarrow i+1} - K \Delta z \frac{\partial h}{\partial x} \mid_{i-1 \rightarrow i}}{\Delta x} \tag{191}$$

Take limit as $\Delta x \rightarrow 0$,

$$S_s \Delta z \frac{\partial h}{\partial t} = \frac{\partial}{\partial x}(K \Delta z \frac{\partial h}{\partial x}) \tag{192}$$

Finally, stipulate that $S_s \Delta z = S$, the storage coefficient (for confined aquifer), and define the aquifer transmissivity as $K \Delta z = T$ and we have performed a back-handed way to get the partial differential equation of aquifer flow.

$$S \frac{\partial h}{\partial t} = \frac{\partial}{\partial x}(T \frac{\partial h}{\partial x}) \tag{193}$$

Ironically, the analysis actually provides an algorithm to approximate head in the aquifer at Equation 188 – which is the subject of the next section.

17.1.1 Finite-Difference Methods – 1 Spatial Dimension

Here we will use Equation 188 as a starting point for simulating aquifer behavior.

As a first model, lets consider the steady flow situation, in which case the left hand side vanishes (there is no change in storage).

$$0 = (K\Delta y\Delta z\frac{h_{i-1} - h_i}{\Delta x}) - (K\Delta y\Delta z\frac{h_i - h_{i+1}}{\Delta x}) \tag{194}$$

Next we will use the arithmetic mean values of the material properties (K) at the cell interfaces, so the difference equation becomes

$$0 = (\frac{1}{2}(K_{i-1} + K_i)\Delta y\Delta z\frac{h_{i-1} - h_i}{\Delta x}) - (\frac{1}{2}(K_i + K_{i-1})\Delta y\Delta z\frac{h_i - h_{i+1}}{\Delta x}) \tag{195}$$

Now lets group some constants:

$$\begin{array}{l} A_i = \frac{1}{2\Delta x}(K_{i-1} + K_i)\Delta y\Delta z \\ B_i = \frac{1}{2\Delta x}(K_i + K_{i+1})\Delta y\Delta z \end{array} \tag{196}$$

Now substitute into the difference equation

$$0 = A_i(h_{i-1}) - (A_i + B_i)(h_i) + B_i(h_{i+1}) \tag{197}$$

Now all that remains is to specify boundary conditions, and then implement an algorithm to solve the resulting system of algebraic equations.[49]

Some of the plausible boundary conditions are:

1. Specified head boundary (pretty easy to specify in a computer representation).
2. Zero-flux boundary (also easy to specify using the cell-centered formulation herein).
3. Specified flux boundary (using a modeling trick not too hard to specify).

These three types of conditions should handle the majority of practical situations where one would need to model an aquifer system.

Lets examine the difference equation a little bit – assume we have the correct values then

$$h_i = \frac{A_i(h_{i-1}) + B_i(h_{i+1})}{(A_i + B_i)} \tag{198}$$

which suggests a nice algorithm. We will simply apply boundary conditions, then evaluate the expression for each cell, and after we do the expression for al the cells,

[49] I have assumed that the spatial step, Δx is the same for each cell – it doesn't have to be, but relaxing that assumption complicates the specifications of the constants A and B.

we will repeat the process until the solution stops changing. Computationally, we are employing a Jacobi iteration scheme, which will work nicely for this particular problem structure. An alternative, equally valid, would be to construct the linear system of equations (in this case it will be a three-banded matrix), and apply an appropriate row reduction technique to find the solution.

Here are a few examples:

Example 1: 1D Steady Flow in a Confined Aquifer using Jacobi Iteration
First the iterative approach.

Consider the aquifer depicted in Figure 91. Suppose that the two wells are located 400 meters apart, and we wish to approximate the head distribution in the aquifer using the multiple-cell-balance model. Further suppose that the desired cell spacing is Δx =100 meters. Additionally suppose the aquifer is 100 meters wide (Δy =100 meters), and the thickness is 50 meters (Δz =50 meters). The hydraulic conductivity everywhere in the aquifer is 0.2 meters per second.

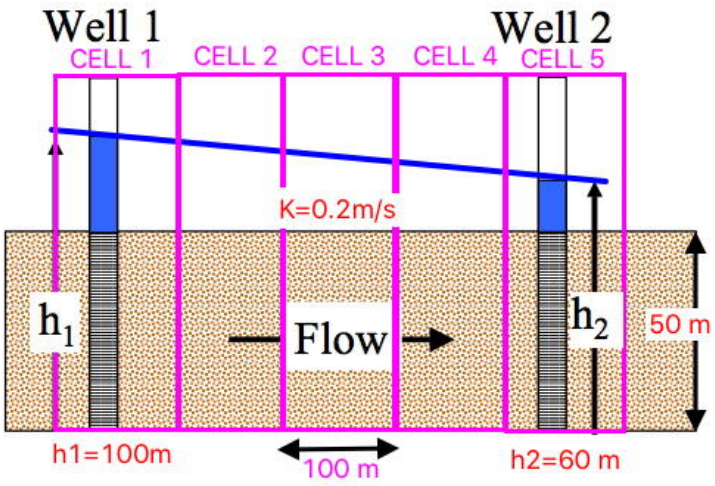

Figure 96. Schematic diagram of unidirectional flow in a generic aquifer, showing heads in two measuring wells. The diagram is annotated with the cell spacing for the example (in magenta), as well as the cell ID. The aquifer material properties are constant (K=0.2 m/s). The head in the left well (Well 1) is 100 meters. The head in the right well (Well 2) is 60 meters..

Figure 96 is the original sketch, annotated with cell spacing, material properties, and the head at the two wells. In the figure, Well 1 has a head measurement of 100 meters, whereas Well 2 has a head measurement of 60 meters. Because the flow is steady, and the material properties are spatially invariant, the EGL/HGL would be a line (the blue one in the figure) connecting the wells sloping from Well 1 to Well 2.

We will use the cell balance model to estimate the EGL/HGL in the aquifer (i.e. find the piezometric surface[50] between the two wells).

[50]Hydraulic head or piezometric head is a specific measurement of liquid pressure above a geodetic

Listing 36. R code demonstrating an Aquifer Flow Simulator for Steady Flow
This fragment of code contains ....

```
# 1D-confined-aquifer-steady-flow
# Implements Finite-Difference Porous Medium Flow using Jacobi Iteration
# Assumes boundary cells 1 and ncells are fixed head cells.
zz <- file("input1.dat", "r") # Open a connection named zz to file named input.dat
#
# read the simulation conditions
#
deltax <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltay <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltaz <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
ncells <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
tolerance <- as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
hydhead <-(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
hydcond <-(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
close(zz)
#
# split the multiple column strings into numeric components for a vector
#
hydhead <-as.numeric(unlist(strsplit(hydhead,split=" ")))
hydcond <-as.numeric(unlist(strsplit(hydcond,split=" ")))
#
# built a position array for plotting
#
distance<-numeric(ncells)
distance[1]<-deltax/2.0
for (i in 2:ncells){distance[i]<-distance[i-1]+deltax}
#
# Plot Distance vs. Head before calculations
#
plot(distance,hydhead,col="red",xlim=c(0,deltax*ncells),ylim=c(0,max(hydhead)*2.0),pch=21,
    tck=1)
lines(distance,hydhead,col="red",type="l",lwd=3)
#
# built the transmissivity arrays
#
amat<-numeric(ncells) # make an ncells long array
bmat<-numeric(ncells) # make an ncells long array
for(irow in 2:(ncells-1)){
  amat[irow]<-((hydcond[irow-1]+hydcond[irow  ])*deltay*deltaz)/(2.0*deltax)
  bmat[irow]<-((hydcond[irow  ]+hydcond[irow+1])*deltay*deltaz)/(2.0*deltax)
}
#
# #se Jacobi iteration to find a solution to the difference equations
#
headold<-hydhead # copy the head array, used to test for stopping
maxit <- 100 # set the maximum number of iterations (to prevent infinite loop)
for (iter in 1:maxit){
  for (irow in 2:(ncells-1)){
    hydhead[irow]<-(amat[irow]*hydhead[irow-1]+bmat[irow]*hydhead[irow+1])/(amat[irow]+bmat
        [irow])
  }
# test for stopping iterations
  percentdiff <- sum((hydhead-headold)^2)
  if (percentdiff < tolerance){break}
  headold<-hydhead
}
#
# Add the steady-flow solution to the graph -- both points and a line.
#
lines(distance,hydhead,col="blue",type="p")
lines(distance,hydhead,col="blue",type="l",lwd=3)
```

Listing 36 is a **R** script that reads an input file, computes the head distirbution (the "blue line") and plots the result. The script also plots the initial values of the heads used in the computation engine. Only the left most value and right most value remain unchanged as they represent the boundary conditions for the problem.

The iterative method is elementary for this case – the replication of the head array

datum. It is usually measured as a liquid surface elevation, expressed in units of length, at the entrance (or bottom) of a piezometer. In an aquifer, it can be calculated from the depth to water in a piezometric well, and given information of the piezometer's elevation and screen depth.

at each step is used to test for stopping when the solution meets some tolerance. The compute, test, update part of the script is a common structure in iterative problems, and when we modify the program for transient (time-varying) behavior it will come in handy. There is no error trapping in the example – so it is quite possible to supply an input file that would not work.

Listing 37 is the contents of the input file (`input1.dat`) that is read by the script. To model different condtions, we would change the input file contents and leave the script alone.

Listing 37. Input File for Example Problem
This fragment of code contains ....

```
100
100
 50
5
1e-12
100 100 100 100 60
0.2 0.2 0.2 0.2 0.2
```

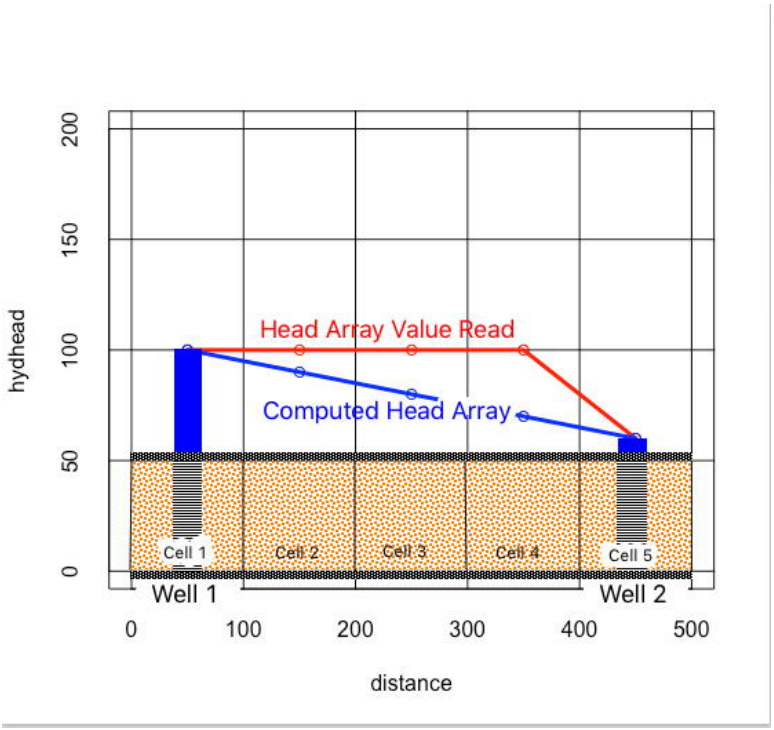

Figure 97. Plot of computed head in each cell using Jacobi interation. The head in the left well (Well 1) is 100 meters. The head in the right well (Well 2) is 60 meters. The steady flow solution is the blue line labelled "Computed Head Array"..

Finally, for this problem Figure 97 is the output graphics from the script. The red line is the value of heads supplied in the input file (these can be any values as long as the left and right values represent the conditions at the two wells); the blue line is the computed piezometric surface. The markers are the computed head for each cell.

Example 2: 1D Steady Flow in a Confined Aquifer using Gaussian Reduction The second example is the identical physical situation, but instead of iterative computations in our code, we will instead construct a simultaneous linear system ($Ax = b$) where the coefficient matrix A is constructed from the difference equations and solved simultaneously.

First we have to specify how to construct the matrices – essentially because h_1 and h_5 are known, there are only three unknowns in the 5-cell example. So we will have a linear system with 3 equations and 3 unknowns. Here is the linear system in the context of the development of the difference equations.

$$\begin{array}{ccccc} -(A_2+B_2)\times h_2 & B_2\times h_3 & 0\times h_4 & = & -A_3\times h_1 \\ A_3\times h_2 & -(A_3+B_3)\times h_3 & B_3\times h_4 & = & 0 \\ 0\times h_2 & A_4\times h_3 & -(A_4+B_4)\times h_4 & = & -B_4\times h_5 \end{array}$$

A more compact representation is

$$\begin{pmatrix} -(A_2+B_2) & B_2 & 0 \\ A_3 & -(A_3+B_3) & B_3 \\ 0 & A_4 & -(A_4+B_4) \end{pmatrix} \begin{pmatrix} h_2 \\ h_3 \\ h_4 \end{pmatrix} = \begin{pmatrix} -A_3h_1 \\ 0 \\ -B_4h_5 \end{pmatrix}$$

Recall that the values for the coefficient matrix are known, as are the values h_1 and h_5. Now we will modify the script, to use the linear system solver and dispense with the iteration entirely.

The modification(s) are to read in the values for the material and spatial properties – just use the same code. Then construct the compact matrix-vector equation by careful looping on the index values. In the 5 cell example, only the inner three cells [2-4] are part of the linear system. So we use indexing in the arithmetic to build the coefficient matrix and right-hand-side.

Then when we have the solution, we need to put the computed values back into their correct position in the head vector. Listing 38 is a code listing that performs the various tasks. The script is intentionally built to use the same input file as the prior example.

The computed results are identical (as anticipated). The next step (and the point of building such tools) is to try different spatial sizing (partly to be sure the code is generic and automatically adapts to such changes), and to apply the tool to aquifers with different material properties.

Listing 38. R code demonstrating an Aquifer Flow Simulator for Steady Flow
This version constructs coefficient matrix then solves the linear system. The program uses the same input file.

```
# 1D-confined-aquifer-steady-flow
# Implements Finite-Difference Porous Medium Flow using Gaussian reduction
# Assumes boundary cells 1 and ncells are fixed head cells.
zz <- file("input1.dat", "r") # Open a connection named zz to file named input.dat
# read the simulation conditons
deltax <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltay <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltaz <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
ncells <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
tolerance <- as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
hydhead <-(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
hydcond <-(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
close(zz)
# split the multiple column strings into numeric components for a vector
hydhead <-as.numeric(unlist(strsplit(hydhead,split=" ")))
hydcond <-as.numeric(unlist(strsplit(hydcond,split=" ")))

# built a position array for plotting
distance<-numeric(ncells)
distance[1]<-deltax/2.0
for (i in 2:ncells){distance[i]<-distance[i-1]+deltax}
plot(distance,hydhead,col="red",xlim=c(0,deltax*ncells),ylim=c(0,max(hydhead)*2.0),pch=21,
    tck=1)
lines(distance,hydhead,col="red",type="l",lwd=3)
# built the transmissivity arrays
amat<-numeric(ncells) # make an ncells long array
bmat<-numeric(ncells) # make an ncells long array
for(irow in 2:(ncells-1)){
  amat[irow]<-((hydcond[irow-1]+hydcond[irow  ])*deltay*deltaz)/(2.0*deltax)
  bmat[irow]<-((hydcond[irow  ]+hydcond[irow+1])*deltay*deltaz)/(2.0*deltax)
}
amatrix <- matrix(0,ncells-2,ncells-2)  # prefill matrix with zeros
rhs <- matrix(0,ncells-2)               # prefill vector with zeros
#####################################################################
## build matrices here -- there are array indexing things going on #
## because we only have three equations to deal with.              #
## hydhead[1] is the left boundary                                 #
## hydhead[ncells] is the right boundary                           #
##  Only build the non-zero elements using the structure of the    #
##  difference-equation stencil (mask, computational molecule ...) #
#####################################################################
## first row special
for (irow in 1:1){
  amatrix[irow,1]=-1.0*(amat[irow+1]+bmat[irow+1])
  amatrix[irow,2]=bmat[irow+1]
  rhs[irow] = -amat[irow+1]*hydhead[irow]
}
## interior rows
for (irow in 2:(ncells-3)){
  amatrix[irow,irow-1]=amat[irow+1]
  amatrix[irow,irow]= -1.0*(amat[irow+1]+bmat[irow+1])
  amatrix[irow,irow+1]= bmat[irow+1]
}
## last row special
for (irow in (ncells-2):(ncells-2)){
  amatrix[irow,ncells-3]=amat[irow+1]
  amatrix[irow,ncells-2]=-1.0*(amat[irow+1]+bmat[irow+1])
  rhs[irow] = -bmat[irow+1]*hydhead[irow+2]
}
#####################################################################
## Now solve the linear system amatrix*unkvector=rhs for unkvector #
##  then put values from unkvector into correct position hydhead   #
#####################################################################
unkvector<-solve(amatrix,rhs)
for (irow in 2:(ncells-1)){
  hydhead[irow] <- unkvector[irow-1]
}
#####################################################################
## Plot the computed values in blue                               #
#####################################################################
lines(distance,hydhead,col="blue",type="p")
lines(distance,hydhead,col="blue",type="l",lwd=3)
```

The next part of the example is what is the cell size is changed? The whole point of input files and such to to keep the script generic. So to check this situation, lets halve the space size (50 meters spacing, instead of 100 meters). Thus instead of 5 cells of 100 meters each, we now have 5 cells of 50 meters each. Listing 39 is a listing of what the input file looks like, other than twice as many initial heads and K values, essentially the same input.

Listing 39. Input File for Example Problem

.

```
50 << changed cell size
100
 50
9 << more cells
1e-12
100 100 100 100 100 100 100 60 << more cells
0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 << more cells
```

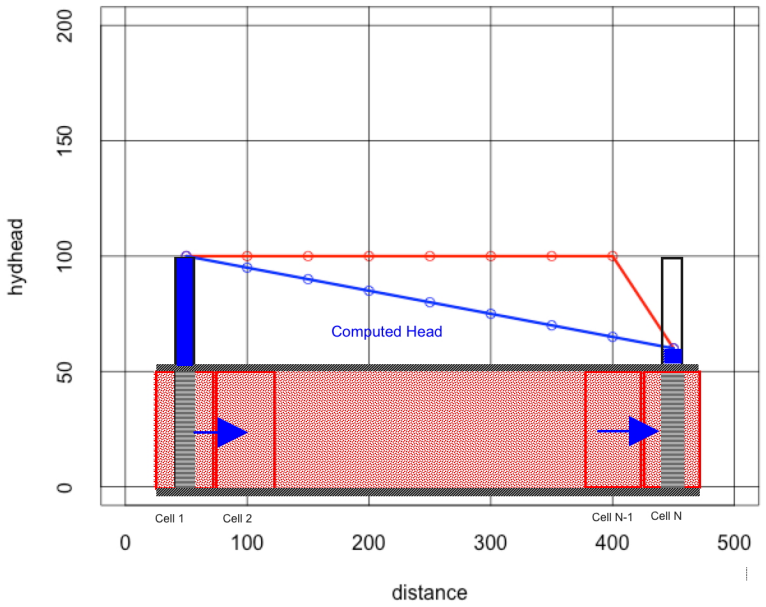

Figure 98. Plot of computed head in each cell using different cell spacing. The head in the left well (Well 1) is 100 meters. The head in the right well (Well 2) is 60 meters. The steady flow solution is the blue line labelled "Computed Head Array"..

Lastly, before we move to 2-Dimensional cases, lets examine when the material properties change. In this change, we will leave the first third of the aquifer with the same properties, but decrease hydraulic conductivity in the next third by 1/2 and the last third by 1/4 again.

Listing 40 is the input file for this case. The anticipated result is that the HGL/EGL will change slope twice (starting out shallow, and getting steeper as we move to the right).

Listing 40. Input File for Example Problem

.

```
50
100
 50
9 <<
1e-12
100 100 100 100 100 100 100 100 60 << more cells
0.2 0.2 0.2 0.1 0.1 0.1 0.05 0.05 0.05 << more cells
```

Figure 99 is the results with this different input file and indeed as anticipated there are three different slopes (it looks like curvature on the plot, but its really just three different line segments with different slopes.).

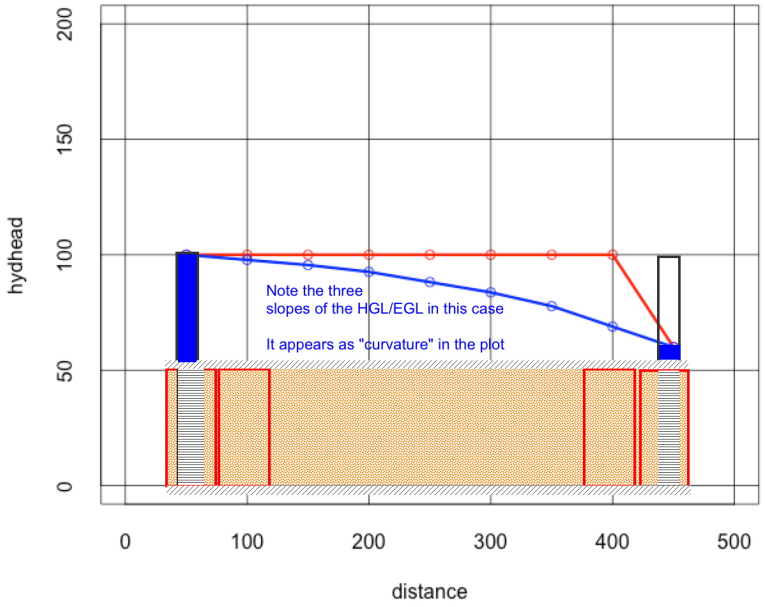

Figure 99. Plot of computed head in each cell using different cell spacing. The head in the left well (Well 1) is 100 meters. The head in the right well (Well 2) is 60 meters. The steady flow solution is the blue line. The apparent curvature is really the three different slopes anticipated as the material properties change..

17.1.2 Finite-Difference Methods – 2 Spatial Dimensions

If we perform an analysis in the same way as we did to arrive at Equation 188 except now include another direction (the y-direction) we will have an aquifer in two spatial dimensions. The governing equation becomes

$$S\frac{\partial h}{\partial t} = \frac{\partial}{\partial x}(T_x\frac{\partial h}{\partial x}) + \frac{\partial}{\partial y}(T_y\frac{\partial h}{\partial y}) \tag{199}$$

The meanings of the terms are the same, except the transmissivity terms now have subscripts to indicate they can have different values depending on direction.

Then as before we will construct the difference-equation model from a multiple-cell balance model of the aquifer at a cell of interest, then extend the equations to cover the entire model domain.

Figure 100 is a plan view schematic of a aquifer with flow to be computed in two directions (x and y). The cell indexing convention in the sketch is that rows are indexed by the letter j and columns are indexed by the letter i. This naming convention is arbitrary; in some instances it is probably preferable to reverse the convention. The schematic also shows the assumed flow directions; for column i, flows are upward in the drawing, and for row j, flows are from left to right. If indeed the opposite is true for a given set of boundary conditions and material properties, then the flows will be computed as negative numbers – hence the convention here is that "positive flow" is right and up.

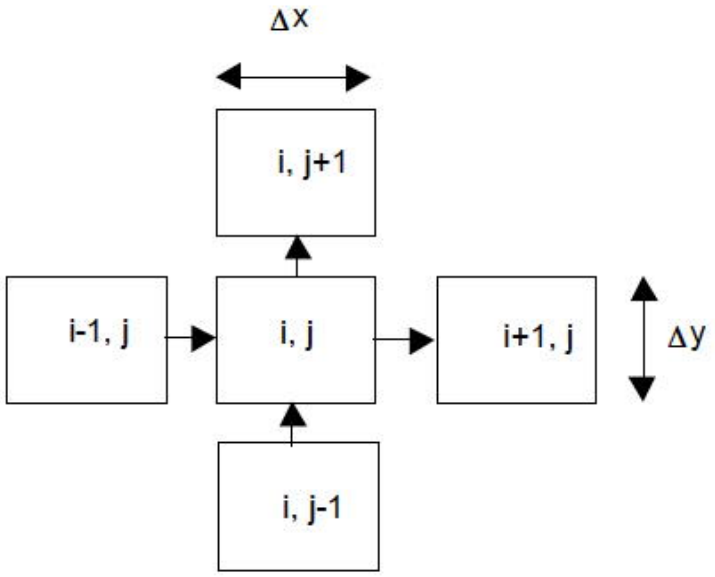

Figure 100. Plan view schematic of 2-dimensional multiple cell balance computational stencil.

As in the one-dimensional development the storage term is

$$\frac{dM_{water}}{dt} \Big|_{cell} = \rho_w S_s \Delta x \Delta y \Delta z \frac{\partial h_i}{\partial t} \tag{200}$$

where h_i is the head in the $i-$th cell.

The mass flows entering the $i-\text{th}, j-\text{th}$ cell are:

$$M_{Inflow} = Q_{left} + Q_{bottom} = \rho_w K_x \Delta y \Delta z \frac{h_{i-1,j} - h_{i,j}}{\Delta x} + \rho_w K_y \Delta x \Delta z \frac{h_{i,j-1} - h_{i,j}}{\Delta y} \tag{201}$$

The mass flows leaving the $i-\text{th}, j-\text{th}$ cell are:

$$M_{Inflow} = Q_{right} + Q_{top} = \rho_w K_x \Delta y \Delta z \frac{h_{i,j} - h_{i+1,j}}{\Delta x} + \rho_w K_y \Delta x \Delta z \frac{h_{i,j} - h_{i,j+1}}{\Delta y} \tag{202}$$

Now write the entire balance equation

$$\rho_w S_s \Delta x \Delta y \Delta z \frac{\partial h_i}{\partial t} = [\rho_w K_x \Delta y \Delta z \frac{h_{i-1,j} - h_{i,j}}{\Delta x} + \rho_w K_y \Delta x \Delta z \frac{h_{i,j-1} - h_{i,j}}{\Delta y}] - [\rho_w K_x \Delta y \Delta z \frac{h_{i,j} - h_{i+1,j}}{\Delta x} + \rho_w K_y \Delta x \Delta z \frac{h_{i,j} - h_{i,j+1}}{\Delta y}] \tag{203}$$

Next replace $S_s \Delta z$ with the storage coefficient S, and the $K_{x,y} \Delta z$ with the transmissivity $T_{x,y}$ terms, and divide by the density of the fluid and the cell plan view area $\Delta x \Delta y$ to obtain a more compact form of the difference equation.

$$S \frac{\partial h_i}{\partial t} = [\frac{1}{\Delta x} T_x \frac{h_{i-1,j} - h_{i,j}}{\Delta x} + \frac{1}{\Delta y} T_y \frac{h_{i,j-1} - h_{i,j}}{\Delta y}] - [\frac{1}{\Delta x} T_x \frac{h_{i,j} - h_{i+1,j}}{\Delta x} + \frac{1}{\Delta y} T_y \frac{h_{i,j} - h_{i,j+1}}{\Delta y}] \tag{204}$$

As in the one-dimensional case, lets again consider steady flow (we will do transient flows later on)

$$0 = [\frac{1}{\Delta x} T_x \frac{h_{i-1,j} - h_{i,j}}{\Delta x} + \frac{1}{\Delta y} T_y \frac{h_{i,j-1} - h_{i,j}}{\Delta y}] - [\frac{1}{\Delta x} T_x \frac{h_{i,j} - h_{i+1,j}}{\Delta x} + \frac{1}{\Delta y} T_y \frac{h_{i,j} - h_{i,j+1}}{\Delta y}] \tag{205}$$

Also as in the one-dimensional case, we will approximate the spatial variation of the material properties (transmissivity) as arithmetic mean values between two cells, so making the following definitions:

$$\begin{aligned} A_{i,j} &= \frac{1}{2\Delta x^2}(T_{x,(i-1,j)} + T_{x,(i,j)}) \\ B_{i,j} &= \frac{1}{2\Delta x^2}(T_{x,(i,j)} + T_{x,(i+1,j)}) \\ C_{i,j} &= \frac{1}{2\Delta y^2}(T_{y,(i,j-1)} + T_{y,(i,j)}) \\ D_{i,j} &= \frac{1}{2\Delta y^2}(T_{y,(i,j)} + T_{y,(i,j+1)}) \end{aligned} \tag{206}$$

Substitution into the difference equation yields

$$0 = A_{i,j} h_{i-1,j} + B_{i,j} h_{i+1,j} - (A_{i,j} + B_{i,j} + C_{i,j} + D_{i,j}) h_{i,j} + C_{i,j} h_{i,j-1} + D_{i,j} h_{i,j+1} \tag{207}$$

As before we can explicitly write the cell equation for $h_{i,j}$ as

$$h_{i,j} = \frac{[A_{i,j}h_{i-1,j} + B_{i,j}h_{i+1,j} + C_{i,j}h_{i,j-1} + D_{i,j}h_{i,j+1}]}{[A_{i,j} + B_{i,j} + C_{i,j} + D_{i,j}]} \tag{208}$$

This difference equation represents an approximation to the governing flow equation, the accuracy depending on the cell size. Boundary conditions are applied directly into the analogs (another name for the difference equations) at appropriate locations on the computational grid. Also as in the one-dimensional case we can generate solutions either by iteration or solving the resulting linear system.

Example 1: 2D Steady Flow in a Confined Aquifer using Jacobi Iteration As before we will start the example with a simple physical condition and use Jacobi iteration[51] to obtain a solution. We will also incorporate two kinds of boundary conditions (fixed head as before, and no-flow boundaries).

Figure 101 is a schematic of this example. The right panel of the figure shows the index naming convention. The known material properties are transmissivity (in each direction, at each cell center, and the thickness of the aquifer ($b == \Delta z$). Our task

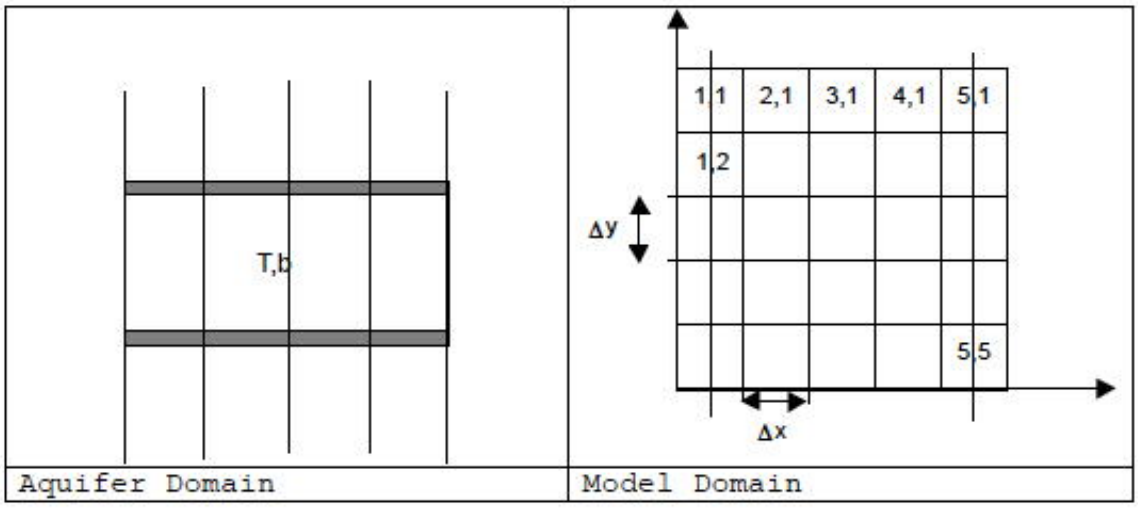

Figure 101. Schematic of 2-dimensional aquifer. The left and right boundaries are constant head boundaries, whereas the upper and lower boundaries are no-flow.

is to simulate the aquifer with the 5 x 5 model shown. The left and right boundaries will be treated as specified head boundaries. The upper and lower boundary will be treated as no flow boundaries.

The head on the left is 100 meters and the right is 60 meters (same as before). The transmissivity ($K_{x,y}\Delta z$=10) square meters per second (but to be consistent with the earlier models we will supply a value of K and Δz; keeping with the earlier

[51] Jacobi iteration for large domain problems (say 200x200) or bigger, is pretty inefficient – better iterative methods are available; however they represent clever changes to the basic iteration methods explained here, hence Jacobi is a good place to start.

examples the values are $K = 0.2$ meters per second, and $\Delta z = 50$ meters. The spatial dimensions are $\Delta x = 100$ meters and $\Delta y = 100$ meters.

Listing 41 is a listing that implements the method – in this case there are changes to the data reading (to read and build matrices), and notice how the no-flow boundary conditions are implemented.

Listing 41. R code demonstrating an Aquifer Flow Simulator for 2D Steady Flow. This code fragment implements the Jacobi iteration. A subsequent listing shows the contour plot syntax. In the example the two fragments are joined and run as a single source file

.

```
# Implements 2D-Confined Aquifer Steady Flow Finite-Difference using Jacobi Iteration
# Assumes no-flow boundary row=1,and nrows.  Assumes fixed head boundary col=1, and ncols
# Head boundary conditions are entered in input file
zz <- file("input1.dat", "r") # Open a connection named zz to file named input.dat
# read the simulation conditons
deltax <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltay <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltaz <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
nrows <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
ncols <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
tolerance <- as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
hydhead <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
hydcondx <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
hydcondy <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
close(zz)
# split the multiple column strings into numeric components for a vector
hydhead <-as.numeric(unlist(strsplit(hydhead,split=" ")))
hydcondx <-as.numeric(unlist(strsplit(hydcondx,split=" ")))
hydcondy <-as.numeric(unlist(strsplit(hydcondy,split=" ")))
# convert the numeric vectors into matrices for easier indexing
hydhead <-matrix(hydhead,nrow=nrows,ncol=ncols,byrow = TRUE)
hydcondx <-matrix(hydcondx,nrow=nrows,ncol=ncols,byrow = TRUE)
hydcondy <-matrix(hydcondy,nrow=nrows,ncol=ncols,byrow = TRUE)
# built the transmissivity arrays
amat<-matrix(0,nrows,ncols)
bmat<-matrix(0,nrows,ncols)
cmat<-matrix(0,nrows,ncols)
dmat<-matrix(0,nrows,ncols)
for(irow in 2:(nrows-1)){
  for(jcol in 2:(ncols-1)){
amat[irow,jcol]<-((hydcondx[irow-1,jcol  ]+hydcondx[irow  ,jcol  ])*deltaz)/(2.0*deltax^2)
bmat[irow,jcol]<-((hydcondx[irow  ,jcol  ]+hydcondx[irow+1,jcol  ])*deltaz)/(2.0*deltax^2)
cmat[irow,jcol]<-((hydcondy[irow  ,jcol-1]+hydcondy[irow  ,jcol  ])*deltaz)/(2.0*deltay^2)
dmat[irow,jcol]<-((hydcondy[irow  ,jcol  ]+hydcondy[irow  ,jcol+1])*deltaz)/(2.0*deltay^2)
  }
}
headold<-hydhead # copy the head array, used to test for stopping
maxit <- 100 # set the maximum number of iterations (to prevent infinite loop)
for (iter in 1:maxit){
  # first and last row special == no flow boundaries
  for(jcol in 1:ncols){
    hydhead[1,jcol]=hydhead[2,jcol]
    hydhead[nrows,jcol]=hydhead[nrows-1,jcol]
  }
  for (irow in 2:(nrows-1)){
    for (jcol in 2:(nrows-1)){
      hydhead[irow,jcol] =
        (amat[irow,jcol]*hydhead[irow-1,jcol  ] +
         bmat[irow,jcol]*hydhead[irow+1,jcol  ] +
         cmat[irow,jcol]*hydhead[irow  ,jcol-1] +
         dmat[irow,jcol]*hydhead[irow  ,jcol+1]  )/
        (amat[irow,jcol]+bmat[irow,jcol]+cmat[irow,jcol]+dmat[irow,jcol])
    }
  }
# test for stopping iterations
  percentdiff <- sum((hydhead-headold)^2)
  if (percentdiff < tolerance){
    message("Exit iterations because tolerance met")
    break}
  headold<-hydhead  #update the current head vector
}
```

Instead of line plots, the built-in contouring algorithm in **R** is used to render the output plot(s). Listing 42 is the script that generates a contour plot. The rows are actually plotted in the vertical, and columns in the horizontal, so the plot is rotated relative to the definition sketch.[52]

Listing 42. R code demonstrating building a contour plot from the computed head distribution

```
#################################################################
###    built position arrays for contour plotting          ###
#################################################################
distancex<-numeric(nrows)
distancey<-numeric(ncols)
distancex[1]<-50
distancey[1]<-50
for (irow in 2:nrows){distancex[irow]<-distancex[irow-1]+deltax}
for (jcol in 2:ncols){distancey[jcol]<-distancey[jcol-1]+deltay}
#################################################################
###    contour plot of head -- note axes are rotated       ###
#################################################################
contour(distancex,distancey,hydhead,
        plot.title = title(main = "Steady 2D Aquifer Model (Head in Meters)",
                           xlab = "Meters (Y axis) ====>>",
                           ylab = "Meters (X axis) ====>>"))
```

Listing 43 is a listing of the input file. The only major change from the one-dimensional examples is the entire head and transmissivity arrays are supplied.

Listing 43. Input File for Example Problem

```
100
100
 50
5
5
1e-12
100 100 100 100 60
100 100 100 100 60
100 100 100 100 60
100 100 100 100 60
100 100 100 100 60
0.2 0.2 0.2 0.2 0.2
0.2 0.2 0.2 0.2 0.2
0.2 0.2 0.2 0.2 0.2
0.2 0.2 0.2 0.2 0.2
0.2 0.2 0.2 0.2 0.2
```

Figure 102 is a screen capture of the **R** output (using the R studio IDE). The plot is on the right side of the screen image. Figure 103 is a screen capture of just the plot, rotated, and with the boundaries identified (no-flow top and bottom; fixed head left and right).

Inspection of the script shows that there are still some parts of the script that could use generalization (namely the graphics portion, and a more sophisticated approach to boundary conditions), but otherwise we have the beginnings of a useful tool.

Example 3: 2D Stream Function using Jacobi Iteration In steady aquifer flow, the flow is irrotational (or at least can be modeled as such). There exists an orthogonal function called the stream function (it is the function that exists in the

[52] This kind of plotting is a hold-over from line-printer days, where the long axis would be oriented to the vertical so that it could print to its heart's content and still fit on the paper. Older tractor-feed line-printers could print 135 characters wide, and as long as the paper roll held out. The paper was called green-bar; each perforated sheet was 11x17 and the sheets were connected. Essentially the paper length was functionally infinite, but the width was fixes at 17 inches.

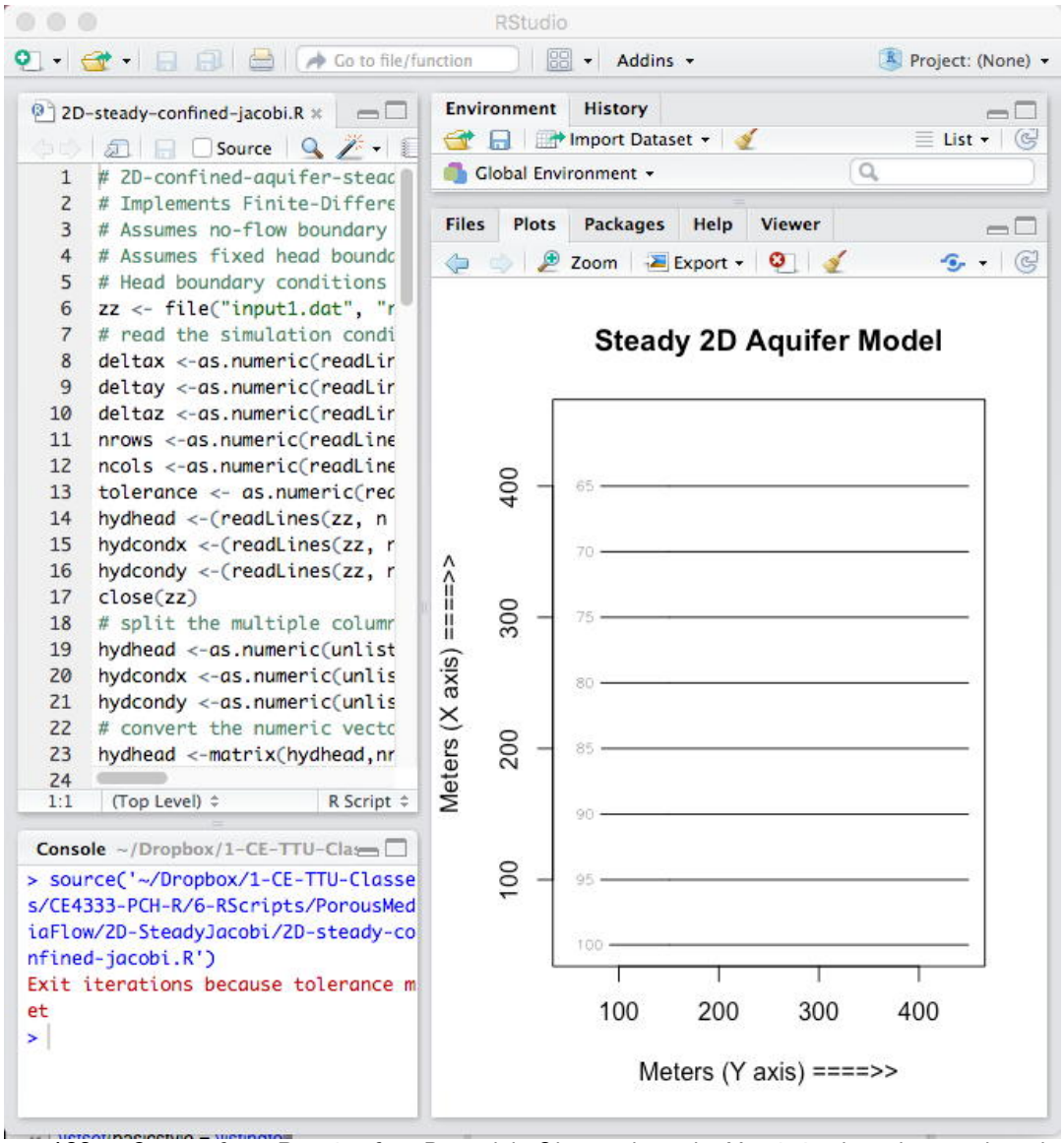

Figure 102. Output from **R** script for 2D model. Observe how the X-axis is plotted upward, and the Y-axis is plotter left to right. The plotting is because of how ROWS and COLUMNS were indexed in the script. Rather than alter the script, I find it easier to rotate the problem for practical application..

flow field when vorticity is zero). A really good explaination of stream functions (and streamlines) appears on pages 381–398 in Zheng and Bennett (1995).

This function can be used to generate plots of streamlines for the same system. The principal changes are the material properties representation and the boundary conditions.

The stream function Ψ as a partial differential equation is

$$0 = \frac{\partial}{\partial x}\left(\frac{1}{T_y}\frac{\partial \Psi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{1}{T_x}\frac{\partial \Psi}{\partial y}\right) \tag{209}$$

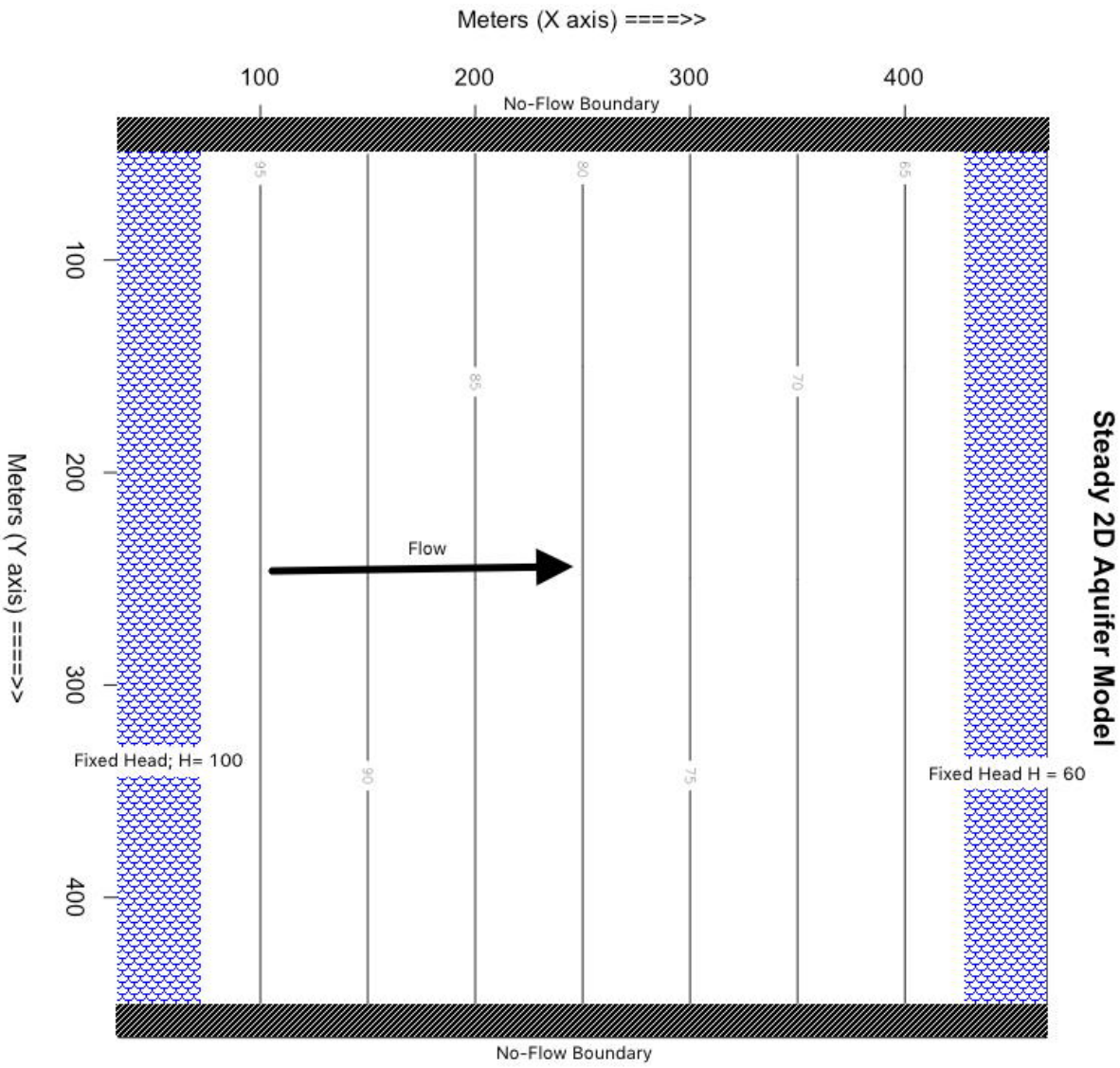

Figure 103. Output from **R** script for 2D model, rotated and annotated to fit the original problem statement.

Observe how the material property is inverted and changes directional identity, otherwise the equation is structurally identical to the groundwater flow equation (for steady flow).

The difference equation is also almost the same

$$0 = [\frac{1}{\Delta x}\frac{1}{T_y}\frac{\Psi_{i-1,j}-\Psi_{i,j}}{\Delta x} + \frac{1}{\Delta y}\frac{1}{T_x}\frac{\Psi_{i,j-1}-\Psi_{i,j}}{\Delta y}]- \\ [\frac{1}{\Delta x}\frac{1}{T_y}\frac{\Psi_{i,j}-\Psi_{i+1,j}}{\Delta x} + \frac{1}{\Delta y}\frac{1}{T_x}\frac{\Psi_{i,j}-\Psi_{i,j+1}}{\Delta y}] \tag{210}$$

The substitutions are

$$\begin{aligned} A_{i,j} &= \tfrac{1}{2\Delta x^2}(T^{-1}_{y,(i-1,j)} + T^{-1}_{y,(i,j)}) \\ B_{i,j} &= \tfrac{1}{2\Delta x^2}(T^{-1}_{y,(i,j)} + T^{-1}_{y,(i+1,j)}) \\ C_{i,j} &= \tfrac{1}{2\Delta y^2}(T^{-1}_{x,(i,j-1)} + T^{-1}_{x,(i,j)}) \\ D_{i,j} &= \tfrac{1}{2\Delta y^2}(T^{-1}_{x,(i,j)} + T^{-1}_{x,(i,j+1)}) \end{aligned} \tag{211}$$

Substitution into the difference equation yields

$$0 = A_{i,j}\Psi_{i-1,j} + B_{i,j}\Psi_{i+1,j} - (A_{i,j} + B_{i,j} + C_{i,j} + D_{i,j})\Psi_{i,j} + C_{i,j}\Psi_{i,j-1} + D_{i,j}\Psi_{i,j+1} \tag{212}$$

As before we can explicitly write the cell equation for $h_{i,j}$ as

$$\Psi_{i,j} = \frac{[A_{i,j}\Psi_{i-1,j} + B_{i,j}\Psi_{i+1,j} + C_{i,j}\Psi_{i,j-1} + D_{i,j}\Psi_{i,j+1}]}{[A_{i,j} + B_{i,j} + C_{i,j} + D_{i,j}]} \tag{213}$$

So at this point we could literally use the same script, however boundary conditions also "invert." A no-flow head-domain boundary is a constant value stream function-domain boundary. A constant value head-domain boundary is a zero-gradient stream function-domain boundary.

So we could use the same code, but probably are better off building a separate code (it can read the same input file), and use it to generate stream functions. The prior example code is modified to generate the stream function for its case (and plot the stream function contours), which if overlaid on the head contours produces a flow net.

The modified code literally changes the names of the head array to stream function, modifies how the material properties (A, B, C, D) are constructed, and modified how the boundary conditions are incorporated. Listing 44 is a listing that implements the method – notice how the no-flow boundary conditions are implemented.

In this example the value of the stream function is arbitrarily set to range from 0 to 100. One useful interpretation of stream function values is that their differences indicate the flow fraction (of total flow) that flows between the streamlines (contour lines of constant stream function value).

Listing 44. R code demonstrating an Stream Function Simulator for 2D Steady Flow. This code fragment implements the Jacobi iteration. A subsequent listing shows the contour plot syntax. In the example the two fragments are joined and run as a single source file

.

```
# 2D-streamfunction
# Implements Finite-Difference Stream Function using Jacobi Iteration
# Assumes no-flow boundary row=1,and nrows ==> constant stream function
# Assumes fixed head boundary col=1, and ncols ==> no-flux stream function -- stream
    function runs from 0 to 100
zz <- file("input1.dat", "r") # Open a connection named zz to file named input.dat
# read the simulation conditons
deltax <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltay <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltaz <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
nrows <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
ncols <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
tolerance <- as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
streamfn <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
hydcondx <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
hydcondy <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
close(zz)
# split the multiple column strings into numeric components for a vector
streamfn <-as.numeric(unlist(strsplit(streamfn,split=" ")))
hydcondx <-as.numeric(unlist(strsplit(hydcondx,split=" ")))
hydcondy <-as.numeric(unlist(strsplit(hydcondy,split=" ")))
# convert the numeric vectors into matrices for easier indexing
streamfn <-matrix(streamfn,nrow=nrows,ncol=ncols,byrow = TRUE)
hydcondx <-matrix(hydcondx,nrow=nrows,ncol=ncols,byrow = TRUE)
hydcondy <-matrix(hydcondy,nrow=nrows,ncol=ncols,byrow = TRUE)
# built the transmissivity arrays
amat<-matrix(0,nrows,ncols)
bmat<-matrix(0,nrows,ncols)
cmat<-matrix(0,nrows,ncols)
dmat<-matrix(0,nrows,ncols)
for(irow in 2:(nrows-1)){
  for(jcol in 2:(ncols-1)){
    amat[irow,jcol]<-((1.0/hydcondy[irow-1,jcol  ]+1.0/hydcondy[irow  ,jcol  ])*deltaz)
        /(2.0*deltax^2)
    bmat[irow,jcol]<-((1.0/hydcondy[irow  ,jcol  ]+1.0/hydcondy[irow+1,jcol  ])*deltaz)
        /(2.0*deltax^2)
    cmat[irow,jcol]<-((1.0/hydcondx[irow  ,jcol-1]+1.0/hydcondx[irow  ,jcol  ])*deltaz)
        /(2.0*deltay^2)
    dmat[irow,jcol]<-((1.0/hydcondx[irow  ,jcol  ]+1.0/hydcondx[irow  ,jcol+1])*deltaz)
        /(2.0*deltay^2)
  }
}
# begin the calculations
streamold<-streamfn # copy the head array, used to test for stopping
maxit <- 100 # set the maximum number of iterations (to prevent infinite loop)
for (iter in 1:maxit){
  # first and last row special == no flow boundaries in head, fixed value streamfunction
  for(jcol in 1:ncols){
    streamfn[1,jcol]=0.0
    streamfn[nrows,jcol]=100.0
  }
  for (irow in 2:(nrows-1)){
  # first and last columns special == fixed head boundaries, no-gradient stream function
    streamfn[irow,1]=streamfn[irow,2]
    streamfn[irow,nrows]=streamfn[irow,nrows-1]
    for (jcol in 2:(nrows-1)){
      streamfn[irow,jcol] =
        (amat[irow,jcol]*streamfn[irow-1,jcol  ] +
         bmat[irow,jcol]*streamfn[irow+1,jcol  ] +
         cmat[irow,jcol]*streamfn[irow  ,jcol-1] +
         dmat[irow,jcol]*streamfn[irow  ,jcol+1]  )/
        (amat[irow,jcol]+bmat[irow,jcol]+cmat[irow,jcol]+dmat[irow,jcol])
    }
  }
# test for stopping iterations
  percentdiff <- sum((streamfn-streamold)^2)
  if (percentdiff < tolerance){
    message("Exit iterations because tolerance met")
    break}
  streamold<-streamfn  #update the current head vector
}
}
```

Listing 45. R code demonstrating an Stream Function Simulator for 2D Steady Flow. This code fragment implements the contour plotting

.

```
# echo contents for debugging
# streamfn
# streamold
#############################################################
###    built position arrays for contour plotting          ###
#############################################################
distancex<-numeric(nrows)
distancey<-numeric(ncols)
distancex[1]<-50
distancey[1]<-50
for (irow in 2:nrows){distancex[irow]<-distancex[irow-1]+deltax}
for (jcol in 2:ncols){distancey[jcol]<-distancey[jcol-1]+deltay}
#############################################################
###    contour plot of head -- note axes are rotated       ###
#############################################################
contour(distancex,distancey,streamfn,
        plot.title = title(main = "Stream Function Model",
                           xlab = "Meters (Y axis) ====>>",
                           ylab = "Meters (X axis) ====>>"))
}
```

Figure 104 is the stream function result. Compare it to Figure 102 and it should be clear that the "lines" are at right angles to each other –that is the stream function is orthogonal to the head function (which is anticipated, because of the nature of the relationship between stream function and head functions; the orthogonality is a consequence of the flow satisfying the the Cauchy-Riemann conditions.).

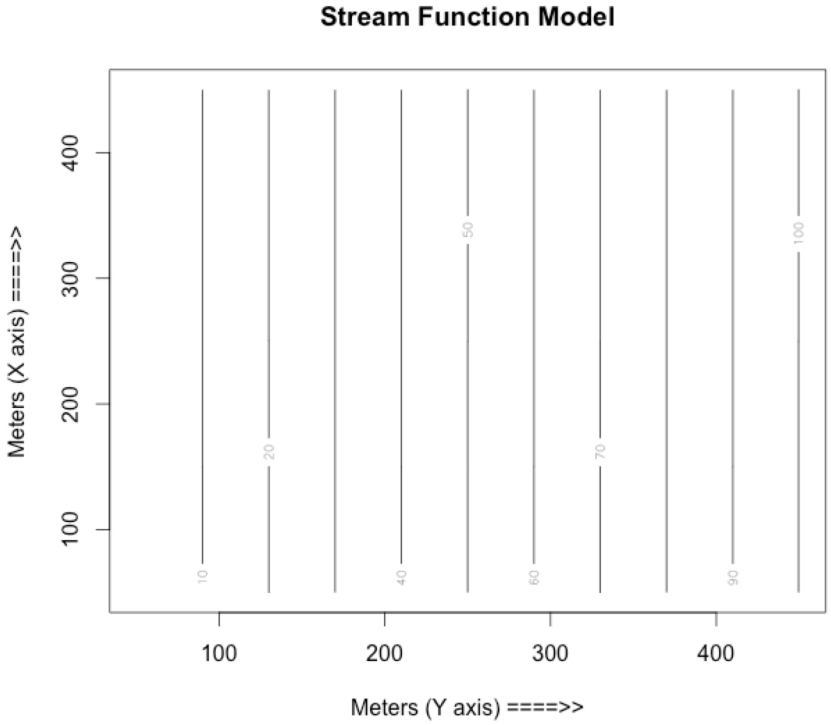

Figure 104. Output from **R** script for stream function model..

To complete the example, and prepare for the next example, we will modify the script one more time to:

1. Read in the material property values, head values, etc. (no change).

2. Compute the head distribution.
3. Compute the stream function distribution. (merge the two scripts)
4. Plot the head and stream function on the same graph – use different colors. Also rotate the plots so axes agree with the problem statement.

Figure 105 is the result of these modifications.

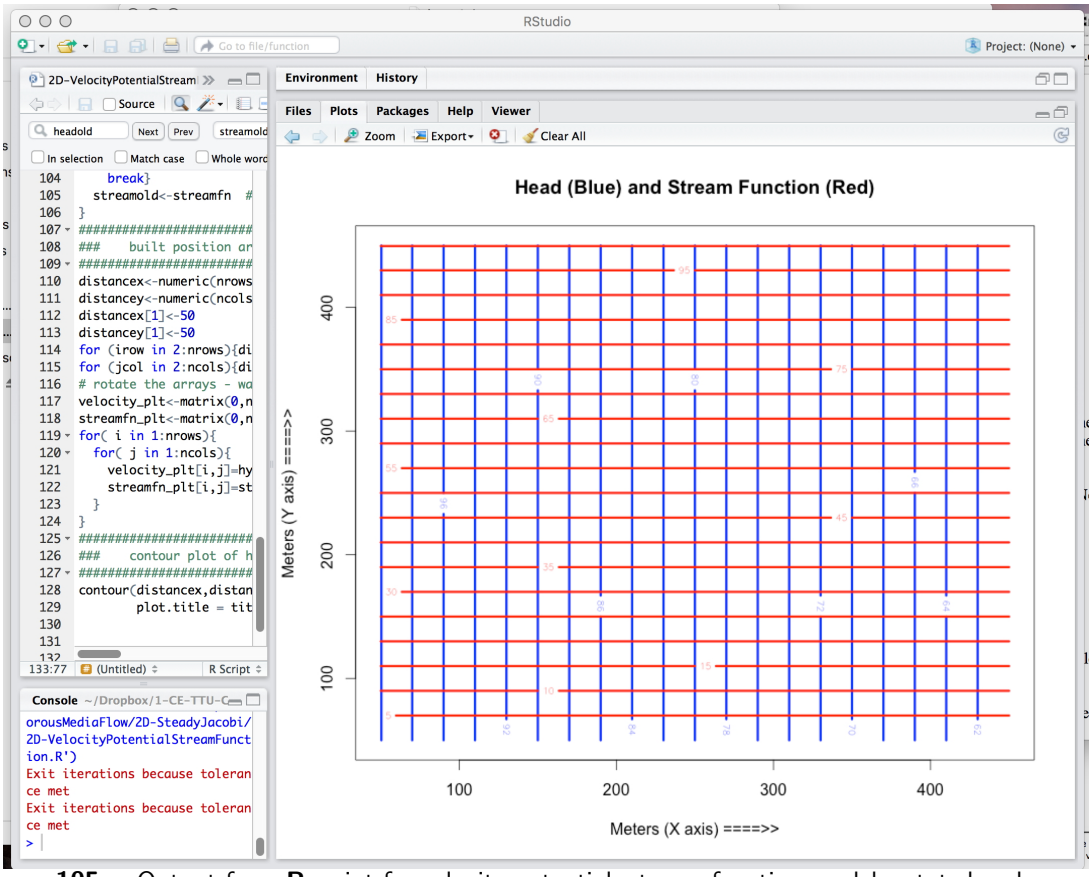

Figure 105. Output from **R** script for velocity-potential, stream function model, rotated and annotated to fit the original problem statement.

Listing 46 is a script fragment that implements the velocity-potential (here the same thing as the head equations) portion of the computations. The data file is the same, the only difference is the head values are copied to another vector (the stream functions) for use in the next fragment.

Listing 47 is a script fragment that implements the stream-function calculations. The A, B, C, D matrices are re-initialized and re-used. The stream function values are computed using the same computation engine (code is repeated – generally poor practice; done here to illustrate the re-use).

Listing 48 is the script fragment that rotates the results and plots the flow net.

Listing 46. R code demonstrating an Velocity Potential (Aquifer Head) Simulator for 2D Steady Flow. This code fragment implements the contour plotting

.

```
# 2D Velocity Potential Stream Function Model
# hydhead is velocity potential
# streamfm is stream function
zz <- file("input1.dat", "r") # Open a connection named zz to file named input.dat
# read the simulation conditons
deltax <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltay <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
deltaz <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
nrows <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
ncols <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
tolerance <- as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
    skipNul = FALSE))
hydhead <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
    FALSE))
hydcondx <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
hydcondy <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
close(zz)
# split the multiple column strings into numeric components for a vector
hydhead <-as.numeric(unlist(strsplit(hydhead,split=" ")))
hydcondx <-as.numeric(unlist(strsplit(hydcondx,split=" ")))
hydcondy <-as.numeric(unlist(strsplit(hydcondy,split=" ")))
# convert the numeric vectors into matrices for easier indexing
hydhead <-matrix(hydhead,nrow=nrows,ncol=ncols,byrow = TRUE)
hydcondx <-matrix(hydcondx,nrow=nrows,ncol=ncols,byrow = TRUE)
hydcondy <-matrix(hydcondy,nrow=nrows,ncol=ncols,byrow = TRUE)
# copy the hydhead array into streamfn for later use
streamfn <- hydhead
# here we perform the velocity potential calculations
# built the transmissivity arrays
amat<-matrix(0,nrows,ncols)
bmat<-matrix(0,nrows,ncols)
cmat<-matrix(0,nrows,ncols)
dmat<-matrix(0,nrows,ncols)
for(irow in 2:(nrows-1)){
  for(jcol in 2:(ncols-1)){
    amat[irow,jcol]<-((hydcondx[irow-1,jcol  ]+hydcondx[irow  ,jcol  ])*deltaz)/(2.0*deltax
        ^2)
    bmat[irow,jcol]<-((hydcondx[irow  ,jcol  ]+hydcondx[irow+1,jcol  ])*deltaz)/(2.0*deltax
        ^2)
    cmat[irow,jcol]<-((hydcondy[irow  ,jcol-1]+hydcondy[irow  ,jcol  ])*deltaz)/(2.0*deltay
        ^2)
    dmat[irow,jcol]<-((hydcondy[irow  ,jcol  ]+hydcondy[irow  ,jcol+1])*deltaz)/(2.0*deltay
        ^2)
  }
}
# veloity potential
headold<-hydhead # copy the head array, used to test for stopping
maxit <- 100 # set the maximum number of iterations (to prevent infinite loop)
for (iter in 1:maxit){
  # first and last row special == no flow boundaries
  for(jcol in 1:ncols){
    hydhead[1,jcol]=hydhead[2,jcol]
    hydhead[nrows,jcol]=hydhead[nrows-1,jcol]
  }
  for (irow in 2:(nrows-1)){
    for (jcol in 2:(nrows-1)){
      hydhead[irow,jcol] =
        (amat[irow,jcol]*hydhead[irow-1,jcol  ] +
           bmat[irow,jcol]*hydhead[irow+1,jcol  ] +
           cmat[irow,jcol]*hydhead[irow  ,jcol-1] +
           dmat[irow,jcol]*hydhead[irow  ,jcol+1]  )/
        (amat[irow,jcol]+bmat[irow,jcol]+cmat[irow,jcol]+dmat[irow,jcol])
    }
  }
  # test for stopping iterations
  percentdiff <- sum((hydhead-headold)^2)
  if (percentdiff < tolerance){
    message("Exit iterations because tolerance met")
    break}
  headold<-hydhead  #update the current head vector
}
}
```

Listing 47. R code demonstrating an Stream Function Simulator for 2D Steady Flow. This code fragment implements the stream function

.

```
# built the streamfn transmissivity arrays  -- notice reuse of the a,b,c,d arrays
amat<-matrix(0,nrows,ncols)
bmat<-matrix(0,nrows,ncols)
cmat<-matrix(0,nrows,ncols)
dmat<-matrix(0,nrows,ncols)
for(irow in 2:(nrows-1)){
  for(jcol in 2:(ncols-1)){
    amat[irow,jcol]<-((1.0/hydcondy[irow-1,jcol  ]+1.0/hydcondy[irow  ,jcol  ])*deltaz)
         /(2.0*deltax^2)
    bmat[irow,jcol]<-((1.0/hydcondy[irow  ,jcol  ]+1.0/hydcondy[irow+1,jcol  ])*deltaz)
         /(2.0*deltax^2)
    cmat[irow,jcol]<-((1.0/hydcondx[irow  ,jcol-1]+1.0/hydcondx[irow  ,jcol  ])*deltaz)
         /(2.0*deltay^2)
    dmat[irow,jcol]<-((1.0/hydcondx[irow  ,jcol  ]+1.0/hydcondx[irow  ,jcol+1])*deltaz)
         /(2.0*deltay^2)
  }
}
streamold<-streamfn # copy the head array, used to test for stopping
maxit <- 100 # set the maximum number of iterations (to prevent infinite loop)
for (iter in 1:maxit){
  # first and last row special == no flow boundaries in head, fixed value streamfunction
  for(jcol in 1:ncols){
    streamfn[1,jcol]=0.0
    streamfn[nrows,jcol]=100.0
  }
  for (irow in 2:(nrows-1)){
  # first and last columns special == fixed head boundaries, no-gradient stream function
    streamfn[irow,1]=streamfn[irow,2]
    streamfn[irow,nrows]=streamfn[irow,nrows-1]
    for (jcol in 2:(nrows-1)){
      streamfn[irow,jcol] =
        (amat[irow,jcol]*streamfn[irow-1,jcol  ] +
         bmat[irow,jcol]*streamfn[irow+1,jcol  ] +
         cmat[irow,jcol]*streamfn[irow  ,jcol-1] +
         dmat[irow,jcol]*streamfn[irow  ,jcol+1]  )/
        (amat[irow,jcol]+bmat[irow,jcol]+cmat[irow,jcol]+dmat[irow,jcol])
    }
  }
# test for stopping iterations
  percentdiff <- sum((streamfn-streamold)^2)
  if (percentdiff < tolerance){
    message("Exit iterations because tolerance met")
    break}
  streamold<-streamfn  #update the current head vector
}
}
```

Listing 48. R code demonstrating an Velocity Potential – Stream Function Simulator for 2D Steady Flow. This code fragment implements the contour plotting

.

```
################################################################
###    built position arrays for contour plotting          ###
################################################################
distancex<-numeric(nrows)
distancey<-numeric(ncols)
distancex[1]<-50
distancey[1]<-50
for (irow in 2:nrows){distancex[irow]<-distancex[irow-1]+deltax}
for (jcol in 2:ncols){distancey[jcol]<-distancey[jcol-1]+deltay}
# rotate the arrays - wasting memory.  Fix for homework.
velocity_plt<-matrix(0,nrows,ncols)
streamfn_plt<-matrix(0,nrows,ncols)
for( i in 1:nrows){
  for( j in 1:ncols){
    velocity_plt[i,j]=hydhead[j,i]
    streamfn_plt[i,j]=streamfn[j,i]
  }
}
################################################################
###    contour plot of head -- note axes are rotated       ###
################################################################
contour(distancex,distancey,velocity_plt,
        plot.title = title(main = "Head (Blue) and Stream Function (Red)",
                           xlab = "Meters (X axis) ====>>",
                           ylab = "Meters (Y axis) ====>>"),
        col="blue",lwd=3,xlim=c(50,450),ylim=c(50,450),nlevels=20)
contour(distancey,distancex,streamfn_plt,add=TRUE,col="red",lwd=3,nlevels=20)
}
```

Example 3: 2D Flow Net in a Confined Aquifer using Jacobi Iteration with Low Permeability Inclusion

Figure 106 is a schematic of a different situation that now only requires us to change the contents of the data file, and re-run the scripts unchanged. Some additional modifications have been added, mostly messages to the user because of anticipated long run times. The data file is changed a bit and two lines are added to help with the plotting – they represent the axis labels used in the contour plots. The boundary conditions are still directly coded into the algorithm, and that would be the next modification to the code - general boundary condition information.[53]

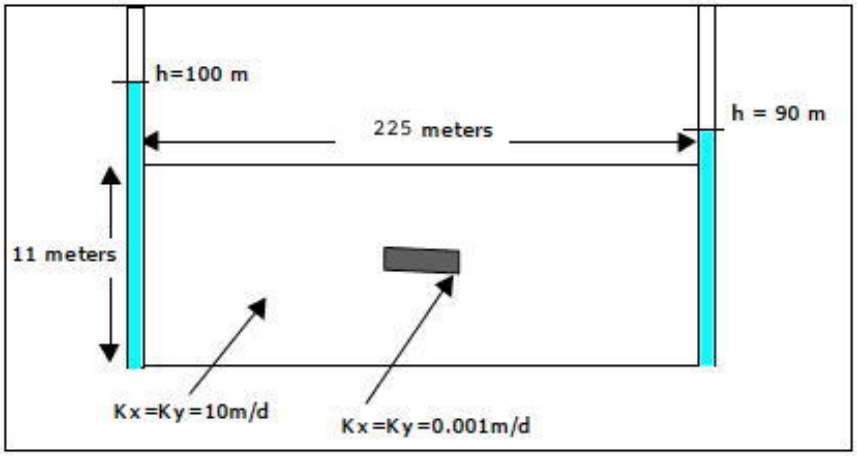

Figure 106. Schematic of vertical slice in an aquifer with low permeability inclusion. Values are indicated on the schematic. Example illustrates how to use the scripts to generate flow nets for the vertical slice .

The following pages contain the code fragments (listings) for the velocity potential, the stream function, and the contour plotting. As above, these listings are combined into a single file (the fragmentation herein is to fit the printed page layout) and then run as a script.

Listing 49 is the listing for the velocity potential calculations.

Listing 50 is the listing for the stream function calculations.

Listing 51 is the is the listing for the plotting calculations.

Listing 52 is the input file for the problem. The file in this case is named `input2.dat`. In addition the generalized boundary conditons, the reader should consider making the program prompt the user for the file name, so that the program is somewhat deployable.

[53] That modification is left for homework.

Listing 49. Velocity Potential Script.

```
# 2D Velocity Potential Stream Function Model
# hydhead is velocity potential; streamfm is stream function
rm(list=ls())  # deallocate (clear) memory
zz <- file("input2.dat", "r") # Open a connection named zz to file named input.dat to read
     input conditions
deltax <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
     skipNul = FALSE))
deltay <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
     skipNul = FALSE))
deltaz <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
     skipNul = FALSE))
nrows <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
     skipNul = FALSE))
ncols <-as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
     skipNul = FALSE))
tolerance <- as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown",
     skipNul = FALSE))
distancex <- (readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
distancey <- (readLines(zz, n = 1, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
hydhead <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
     FALSE))
hydcondx <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
      FALSE))
hydcondy <-(readLines(zz, n = nrows, ok = TRUE, warn = TRUE,encoding = "unknown", skipNul =
      FALSE))
close(zz)
# split the multiple column strings into numeric components for a vector
distancex <-as.numeric(unlist(strsplit(distancex,split=" ")))
distancey <-as.numeric(unlist(strsplit(distancey,split=" ")))
hydhead <-as.numeric(unlist(strsplit(hydhead,split=" ")))
hydcondx <-as.numeric(unlist(strsplit(hydcondx,split=" ")))
hydcondy <-as.numeric(unlist(strsplit(hydcondy,split=" ")))
# convert the numeric vectors into matrices for easier indexing
hydhead <-matrix(hydhead,nrow=nrows,ncol=ncols,byrow = TRUE)
hydcondx <-matrix(hydcondx,nrow=nrows,ncol=ncols,byrow = TRUE)
hydcondy <-matrix(hydcondy,nrow=nrows,ncol=ncols,byrow = TRUE)
# built the transmissivity arrays
amat<-matrix(0,nrows,ncols)
bmat<-matrix(0,nrows,ncols)
cmat<-matrix(0,nrows,ncols)
dmat<-matrix(0,nrows,ncols)
for(irow in 2:(nrows-1)){
  for(jcol in 2:(ncols-1)){
    amat[irow,jcol]<-((hydcondx[irow-1,jcol  ]+hydcondx[irow  ,jcol  ])*deltaz)/(2.0*deltax
        ^2)
    bmat[irow,jcol]<-((hydcondx[irow  ,jcol  ]+hydcondx[irow+1,jcol  ])*deltaz)/(2.0*deltax
        ^2)
    cmat[irow,jcol]<-((hydcondy[irow  ,jcol-1]+hydcondy[irow  ,jcol  ])*deltaz)/(2.0*deltay
        ^2)
    dmat[irow,jcol]<-((hydcondy[irow  ,jcol  ]+hydcondy[irow  ,jcol+1])*deltaz)/(2.0*deltay
        ^2)
  }
}
headold<-hydhead      # copy the head array, used to test for stopping
tolflag<-FALSE
maxit <- 500000 # set the maximum number of iterations (to prevent infinite loop)
for (iter in 1:maxit){
  # first and last row special == no flow boundaries
  for(jcol in 1:ncols){
    hydhead[1,jcol]<-hydhead[2,jcol]
    hydhead[nrows,jcol]<-hydhead[nrows-1,jcol]
  }
  for (irow in 2:(nrows-1)){
    for (jcol in 2:(ncols-1)){
      hydhead[irow,jcol] <-
        (amat[irow,jcol]*hydhead[irow-1,jcol  ] +
           bmat[irow,jcol]*hydhead[irow+1,jcol  ] +
           cmat[irow,jcol]*hydhead[irow  ,jcol-1] +
           dmat[irow,jcol]*hydhead[irow  ,jcol+1]  )/
        (amat[irow,jcol]+bmat[irow,jcol]+cmat[irow,jcol]+dmat[irow,jcol])
    }
  }
  # test for stopping iterations
  percentdiff <- sum((hydhead-headold)^2)
  if (percentdiff < tolerance){
    message("Exit iterations in velocity potential because tolerance met");
    message("Iterations =", iter);
    tolflag <- TRUE
    break}
 headold<-hydhead  #update the current head vector
 if( iter %% 5000 == 0){message("Calculating in Potential Function ",iter," of ",maxit, "
      iterations")}
}
if (tolflag == FALSE){message("Exit iterations in potential function at max iterations")}
}
```

Listing 50. Stream Function Script.

```
streamfn<-matrix(1.0,nrows,ncols)
for (i in 1:nrows){
  for(j in 1:ncols){
    streamfn[i,j]=as.numeric(i)/as.numeric(nrows)
  }
}
streamfn
# built the streamfn transmissivity arrays
amats<-matrix(0,nrows,ncols)
bmats<-matrix(0,nrows,ncols)
cmats<-matrix(0,nrows,ncols)
dmats<-matrix(0,nrows,ncols)
for(irow in 2:(nrows-1)){
  for(jcol in 2:(ncols-1)){
    amats[irow,jcol]<-((1.0/hydcondy[irow-1,jcol  ]+1.0/hydcondy[irow  ,jcol  ])*deltaz)
        /(2.0*deltax^2)
    bmats[irow,jcol]<-((1.0/hydcondy[irow  ,jcol  ]+1.0/hydcondy[irow+1,jcol  ])*deltaz)
        /(2.0*deltax^2)
    cmats[irow,jcol]<-((1.0/hydcondx[irow  ,jcol-1]+1.0/hydcondx[irow  ,jcol  ])*deltaz)
        /(2.0*deltay^2)
    dmats[irow,jcol]<-((1.0/hydcondx[irow  ,jcol  ]+1.0/hydcondx[irow  ,jcol+1])*deltaz)
        /(2.0*deltay^2)
  }
}
streamold<-streamfn # copy the head array, used to test for stopping
tolflag<-FALSE
maxit <- maxit/10 # set the maximum number of iterations (to prevent infinite loop)
for (iter in 1:maxit){
  # first and last row special == no flow boundaries in head, fixed value streamfunction
  for(jcol in 1:ncols){
    streamfn[1,jcol]=0.0
    streamfn[nrows,jcol]=1.0
  }
  for (irow in 2:(nrows-1)){
    # first and last columns special == fixed head boundaries, no-gradient stream function
    streamfn[irow,1]=streamfn[irow,2]
    streamfn[irow,ncols]=streamfn[irow,ncols-1]
    for (jcol in 2:(ncols-1)){
      streamfn[irow,jcol] =
        (amats[irow,jcol]*streamfn[irow-1,jcol  ] +
           bmats[irow,jcol]*streamfn[irow+1,jcol  ] +
           cmats[irow,jcol]*streamfn[irow  ,jcol-1] +
           dmats[irow,jcol]*streamfn[irow  ,jcol+1]  )/
        (amats[irow,jcol]+bmats[irow,jcol]+cmats[irow,jcol]+dmats[irow,jcol])
    }
  }
  # test for stopping iterations
  percentdiff <- sum((streamfn-streamold)^2)
  if (percentdiff < tolerance)
    {
    message("Exit iterations in stream function because tolerance met");
    message("Iterations =", iter);
    tolflag <- TRUE;
    break
    }
  streamold<-streamfn  #update the current head vector
  if( iter %% 5000 == 0){message("Calculating in Stream Function ",iter," of ",maxit, "
      iterations")}
}
if (tolflag == FALSE){message("Exit iterations in stream function at max iterations")}
}
```

Listing 51. Contour plotting script.

```
##############################################################################
###    rotate arrays for contour plotting -- position values read in input  ###
##############################################################################
velocity_plt<-matrix(0,ncols,nrows)
streamfn_plt<-matrix(0,ncols,nrows)
for( i in 1:nrows){
  for( j in 1:ncols){
    velocity_plt[j,i]=hydhead[i,j]
    streamfn_plt[j,i]=streamfn[i,j]
  }
}
#################################################################
###    contour plot of head -- note axes are rotated       ###
#################################################################
contour(distancey,distancex,velocity_plt,
        plot.title = title(main = "Head (Blue) and Stream Function (Red)",
                           xlab = "Meters (X axis) ====>>",
                           ylab = "Meters (Y axis) ====>>"),
        col="blue",lwd=3,nlevels=20)
contour(distancey,distancex,streamfn_plt,add=TRUE,col="red",lwd=3,nlevels=20)
}
```

Listing 52. Input file for 2D vertical slice confined aquifer with low permeability inclusion.

```
1
10
1
13
23
1e-16
0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5 10.5 11.5 12.5
5 15 25 35 45 55 65 75 85 95 105 115 125 135 145 155 165 175 185 195 205 215 225
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100 90
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 0.0001 0.0001 0.0001 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 0.0001 0.0001 0.0001 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 0.0001 0.0001 0.0001 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 0.0001 0.0001 0.0001 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 0.0001 0.0001 0.0001 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 0.0001 0.0001 0.0001 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
}
```

Lastly, the result of running the script on the input file is shown in figure 107

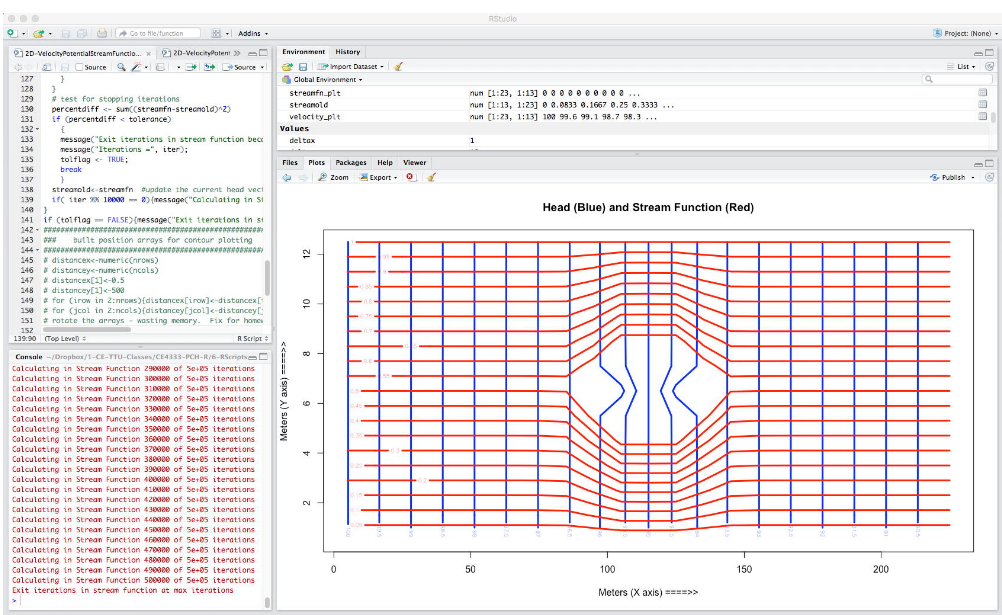

Figure 107. Vertical slice in an aquifer with low permeability inclusion using Jacobi iteration scripts.

[Modifications to handle generalized boundary conditions]

17.2 Unconfined Aquifer Flow

17.2.1 Finite-Difference Methods

17.3 Exercises

18 Unsteady Groundwater Flow

18.1 Confined Aquifers

18.1.1 Explicit Formulation

18.1.2 Implicit Formulation

18.2 Unconfined Aquifers

18.2.1 Explicit Formulation

18.2.2 Implicit Formulation

19 Flow Nets

20 Heat and Mass Transport

20.1 Concentration

20.2 Tracer Hypothesis

20.3 Advection (Convection) Process

20.4 Diffusion Process

20.5 Dispersion Process

20.6 Reaction Processes

20.6.1 Linear Equilibrum Isotherms and Concept of Retardation

20.7 Advection, Dispersion, Retardation, Decay Mathematics

21 Analytical Solutions to ADR Equations

21.1 Impulse Solution

21.2 Ogata-Banks Solution

21.3 2D-Injection (Hunt) Solution

21.4 2D-, and 3D- Domenico Robbins Solutions

22 Numerical Solutions to ADR Equations

22.1 Explicit Methods

22.1.1 Centered Differences

22.1.2 Upwind Formulation(s)

22.2 Crank-Nicholson Methods

References

Christian, B. and Griffiths, T. (2016). Algorithms to Live By: The Computer Science of Human Decisions Henry Holt and Co.. Kindle Edition.

Zheng, C. and Bennett, G. D. (1995). *Applied Contaminant Transport Modeling*. Van Nostrand Reinhold.

Press, W.H., Flannery, B.P., Teukolsky, S.A., Vettering, W. T. 1986, Numerical Recipes:*The Art of Scientific Computing*. Cambridge University Press, London. 818p.

Gill, P.E., Murray, W, Wright, M. H., 1981. Practical Optimization. Academic Press, San Diego. 401p.

Wikipedia discussion of Newton's method. `http://en.wikipedia.org/wiki/Newton's_method`.

Wikipedia discussion of matrix inversion. `http://en.wikipedia.org/wiki/Invertible_matrix`.

Christian, B. and Griffiths, T. (2016). Algorithms to Live By: The Computer Science of Human Decisions Henry Holt and Co.. Kindle Edition.

Chin, D. A. (2006). *Water-Resources Engineering*. Prentice Hall.

Haman YM, Brameller A. (1971) Hybrid method for the solution of piping networks. Proc IEEE 1971;118(11):1607-12.