

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.²²
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 recommended; 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 GPU²³

²²In 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.

²³Graphics 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

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

and

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

Now that we have a way (albeit pretty arcane) for getting matrices into our program from a file²⁴ 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 of linear equations (and non-linear systems in the next chapter).

²⁴The 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,

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

Could be represented by set of vectors and matrices²⁵

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

and the linear system then written as

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

²⁵Usually 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.

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

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.

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

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

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

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

Listing 26. R code demonstrating scalar multiplication.

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

Figure 55 is an example using the earlier **A** matrix and multiplying it by the scalar value of 2.0.

The screenshot shows the RStudio interface. The top panel displays the code for scalar multiplication:

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

The bottom panel shows the R console output:

```
Console ~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/5-LinearSystems/RScripts/ReadMatrixNew.R'
> source('~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/5-LinearSystems/RScripts/ReadMatrixNew.R')
 [,1] [,2] [,3]
[1,] 12    7    3
[2,] 4     5    6
[3,] 7     8    9
 [,1] [,2] [,3] [,4]
[1,] 5     8    1    2
[2,] 6     7    3    0
[3,] 4     5    9    1
 [,1] [,2] [,3]
[1,] 24   14   6
[2,] 8    10   12
[3,] 14   16   18
> |
```

The console output shows the original matrix **Amat** and the resulting matrix **twoA**, which is the original matrix multiplied by 2.0.

Figure 55. 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

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

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

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

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 56 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.

The screenshot shows the RStudio interface. The top panel displays the code in `ReadMatrixNew.R`. The code initializes variables `HowManyRowsA`, `HowManyColumnsB`, and `HowManyRowsB` from vectors `yy` and `zz`. It then builds matrices `Amat` and `Bmat` by populating them with elements from `yy` and `zz` respectively. Afterward, it prints `Amat`, creates matrix `twoA` as twice `Amat`, prints `twoA`, creates matrix `threeA` as the sum of `Amat` and `twoA`, and prints `threeA`. The bottom panel shows the R Console output, which includes the source command, the printed matrices `Amat` and `twoA`, and the resulting matrix `threeA`.

```

14 HowManyRowsA <- length(yy$V1)
15 HowManyColumnsB <- length(zz)
16 HowManyRowsB <- length(zz$V1)
17 ##### Build A and B Matrices #####
18 Amat <- matrix(0,nrow = HowManyRowsA, ncol = HowManyColumnsB)
19 Bmat <- matrix(0,nrow = HowManyRowsB, ncol = HowManyColumnsB)
20 for (i in 1:HowManyRowsA){
21   for(j in 1:(HowManyColumnsA)){
22     Amat[i,j] <- yy[i,j]
23   }
24 }
25 rm(yy) # deallocate yy and just work with matrix and vectors
26 for (i in 1:HowManyRowsB){
27   for(j in 1:(HowManyColumnsB)){
28     Bmat[i,j] <- zz[i,j]
29   }
30 }
31 rm(zz) # deallocate zz and just work with matrix and vectors
32 ##### Echo Input #####
33 print(Amat)
34 #print(Bmat)
35 twoA <- 2 * Amat
36 print(twoA)
37 threeA <- Amat+twoA
38 print(threeA)
39
38:13  Echo Input  R Script
Console ~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PChinR-LectureNotes/5-LinearSystems/RScripts/ReadMatrixNew.R'
> source('~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PChinR-LectureNotes/5-LinearSystems/RScripts/ReadMatrixNew.R')
 [,1] [,2] [,3]
[1,] 12    7    3
[2,] 4     5    6
[3,] 7     8    9
 [,1] [,2] [,3]
[1,] 24   14   6
[2,] 8    10   12
[3,] 14   16   18
 [,1] [,2] [,3]
[1,] 36   21   9
[2,] 12   15   18
[3,] 21   24   27
>

```

Figure 56. 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 important²⁶.

To obtain **C = A 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

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

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).

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

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

²⁶Matrix multiplication is not transitive; $\mathbf{AB} \neq \mathbf{BA}$.

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

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

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

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

Making the substitutions results in :

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

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 57 is a script that multiplies the two matrices above and prints the result.²⁷

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, i -th element (diagonal) which is equal to 1:

²⁷Internal 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.

The screenshot shows the RStudio interface. The top panel displays the code in 'ReadMatrixNew.R'. The code performs matrix operations, including copying values from 'yy' to 'Amat' and 'zz' to 'Bmat', and then calculating 'twoA' and 'threeA' as linear combinations of 'Amat'. It also prints matrices 'A' and 'B' and their product 'C'. The bottom panel, titled 'Console', shows the resulting matrix multiplication:

```

[1,] -29 17
[2,] 5 10
> source('~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/5-LinearSystems/RScripts/ReadMatrixNew.R')
[,1] [,2] [,3]
[1,] 1 5 7
[2,] 2 9 3
[,1] [,2]
[1,] 3 -2
[2,] -2 1
[3,] 1 1
[,1] [,2]
[1,] 0 10
[2,] -9 8
>

```

Figure 57. Matrix multiplication example.

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

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

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \quad (48)$$

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 matrix²⁸. 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

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

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

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

Figure 58 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.²⁹

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:

²⁸The 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

²⁹Why do you think this is so, when above we stated that multiplication is intransitive?

The screenshot shows the RStudio interface. The top panel is the code editor with the file 'ReadMatrixNew.R' open. The code contains R code for matrix operations, including loops to copy elements from one matrix to another, deallocate memory, and perform matrix multiplication. It also includes sections for printing matrices and defining new cases. The bottom panel is the 'Console' window, which displays the results of running the script. The output shows two matrices, A and B, and their product C. Matrix A is [[2, 4, 3, -3], [-9, 8]] and Matrix B is [[1/6, 2/9, 1/6, -1/9], [1/6, 2/9, 1/6, -1/9]]. The product C is [[1, 0], [0, 1]], indicating that B is the inverse of A.

```

for(i in 1:HowManyRowsA){
  for(j in 1:(HowManyColumnsA)){
    Amat[i,j] <- yy[i,j]
  }
}
rm(yy) # deallocate yy 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)
#twoA <- 2 * Amat
#print(twoA)
#threeA <- Amat+twoA
#print(threeA)
#####
# New Cases #####
A <- matrix(c(2,4,3,-3),nrow=2,ncol=2)
B <- matrix(c(1/6,2/9,1/6,-1/9),nrow=2,ncol=2)
print(A)
print(B)
C <- A %*% B
print(C)

```

```

[,1] [,2]
[1,]    0   10
[2,]   -9    8
> source('~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/5-LinearSystems/RScripts/ReadMatrixNew.R')
[,1] [,2]
[1,]    2    3
[2,]    4   -3
[,1]      [,2]
[1,] 0.1666667  0.1666667
[2,] 0.2222222 -0.1111111
[,1] [,2]
[1,]    1    0
[2,]    0    1
>

```

Figure 58. 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).

$$\mathbf{A}|\mathbf{I} = \left(\begin{array}{cc|cc} 2 & 3 & 1 & 0 \\ 4 & -3 & 0 & 1 \end{array} \right) \quad (51)$$

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.

$$\mathbf{A}|\mathbf{I} = \left(\begin{array}{cc|cc} 2/2 & 3/2 & 1/2 & 0 \\ 4 & -3 & 0 & 1 \end{array} \right) = \left(\begin{array}{cc|cc} 1 & 3/2 & 1/2 & 0 \\ 4 & -3 & 0 & 1 \end{array} \right) \quad (52)$$

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.

$$\mathbf{A}|\mathbf{I} = \left(\begin{array}{cc|cc} 1 & 3/2 & 1/2 & 0 \\ 4 - 4(1) & -3 - 4(3/2) & 0 - 4(1/2) & 1 - 4(0) \end{array} \right) = \left(\begin{array}{cc|cc} 1 & 3/2 & 1/2 & 0 \\ 0 & -9 & -2 & 1 \end{array} \right) \quad (53)$$

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

$$\mathbf{A}|\mathbf{I} = \left(\begin{array}{cc|cc} 1 & 3/2 & 1/2 & 0 \\ 0 & -9/-9 & -2/-9 & 1/-9 \end{array} \right) = \left(\begin{array}{cc|cc} 1 & 3/2 & 1/2 & 0 \\ 0 & 1 & 2/9 & -1/9 \end{array} \right) \quad (54)$$

4. For all rows above and below this current row, replace row_j with $row_j - a_{j,2} * row_2$. 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.

$$\mathbf{A}|\mathbf{I} = \quad (55)$$

$$\left(\begin{array}{cc|cc} 1 & 3/2 - (3/2)(1) & 1/2 - (3/2)(2/9) & 0 - (3/2)(-1/9) \\ 0 & 1 & 2/9 & -1/9 \end{array} \right) = \quad (56)$$

$$\left(\begin{array}{cc|cc} 1 & 0 & 1/6 & 1/6 \\ 0 & 1 & 2/9 & -1/9 \end{array} \right) \quad (57)$$

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.³⁰

Figure 59 is a screen capture of using `solve(...)` to find the inverse of **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

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b} \quad (58)$$

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 **A** and the right hand side **b**, and then supply these two matrices to the solve routine (e.g. `x <- solve(A,b)`).

³⁰If 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 identity matrix columns (as was done above).

The screenshot shows the RStudio interface. The top bar displays 'RStudio'. Below it, the 'File' menu has 'Source on Save' checked. The 'Run' button is highlighted. The left sidebar shows two open files: 'JacobiUC.R' and 'ReadMatrixNew.R'. The 'JacobiUC.R' file contains the following R code:

```

20 for (i in 1:HowManyRowsA){
21   for(j in 1:(HowManyColumnsA)){
22     Amat[i,j] <- yy[i,j]
23   }
24 }
25 rm(yy) # deallocate zz and just work with matrix and vectors
26 for (i in 1:HowManyRowsB){
27   for(j in 1:(HowManyColumnsB)){
28     Bmat[i,j] <- zz[i,j]
29   }
30 }
31 rm(zz) # deallocate zz and just work with matrix and vectors
32 ##### Echo Input #####
33 #print(Amat)
34 #print(Bmat)
35 #twoA <- 2 * Amat
36 #print(twoA)
37 #threeA <- Amat+twoA
38 #print(threeA)
39 ##### New Cases #####
40 A <- matrix(c(2,4,3,-3),nrow=2,ncol=2)
41 B <- matrix(c(1/6,2/9,1/6,-1/9),nrow=2,ncol=2)
42 print(A)
43 print(B)
44 C <- A %*% B
45 #print(C)
46 A_inv <- solve(A)
47 print(A_inv)
48

```

The 'Console' tab at the bottom shows the following output:

```

Console ~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/5-LinearSystems/RScripts/
> source '~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/5-LinearSystems/RScripts/ReadMatrixNew.R'
 [,1] [,2]
[1,]    2    3
[2,]    4   -3
 [,1]      [,2]
[1,] 0.1666667  0.1666667
[2,] 0.2222222 -0.1111111
 [,1]      [,2]
[1,] 0.1666667  0.1666667
[2,] 0.2222222 -0.1111111
>

```

Figure 59. The matrix inversion script showing results of a run and various input and output..

5.4 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(...)`.³¹ 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:

$$\begin{aligned} 8x_1 + 1x_2 - 1x_3 &= 8 \\ 1x_1 - 7x_2 + 2x_3 &= -4 \\ 2x_1 + 1x_2 + 9x_3 &= 12 \end{aligned} \tag{59}$$

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):

$$\begin{aligned} x_1 &= 1.000 & -0.125x_2 & 0.125x_3 \\ x_2 &= 0.571 & 0.143x_1 & 0.286x_3 \\ x_3 &= 1.333 & -0.222x_1 & -0.111x_2 \end{aligned} \tag{60}$$

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 4.

Table 4. 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:

³¹The **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 $\mathbf{x}_{\text{guess}}$.
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, $\mathbf{residual} = \mathbf{A} \cdot \mathbf{x}_{\text{guess}} - \mathbf{b}$.
4. Divide this error vector by the diagonal weights $\mathbf{update} = \mathbf{residual}/\mathbf{W}$
5. Update the solution vector $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{guess}} - \mathbf{update}$
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 diagonal 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 60 is a screen capture of the script in Listing 27 applied to the example problem.

The screenshot shows the RStudio interface with two tabs open: 'JacobiUC.R' and 'ReadMatrixNew.R'. The 'JacobiUC.R' tab contains the R script code for the Jacobi iteration method. The 'Console' tab shows the execution of the script, displaying the matrix A, the right-hand side vector Bvec, and the solution vector x. The output indicates that the sum squared error vector is small (1.7061092645239e-13), the number of iterations is 15, and the coefficient matrix A is:

```

Coefficient Matrix :
 [,1] [,2] [,3]
[1,] 8 1 -1
[2,] 1 -7 2
[3,] 2 1 9

```

The solution vector x is:

```

Solution Vector :
 [,1]
[1,] 1
[2,] 1
[3,] 1

```

The right-hand side vector Bvec is:

```

Right-Hand Side Vector :
 Bvec
[1,] 8
[2,] -4
[3,] 12

```

Figure 60. Jacobi Iteration applied to Example 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.

$$\begin{aligned} x^2 + y^2 &= 4 \\ e^x + y &= 1 \end{aligned} \tag{61}$$

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

$$\mathbf{A} = \begin{pmatrix} x_k & + y_k \\ 0 & + 1 \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 4 \\ 1 - e^{x_k} \end{pmatrix} \tag{62}$$

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 61 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.

The screenshot shows the RStudio interface with the QuasiLinear.R script open in the editor. The script contains the R code from Listing 28. In the bottom right corner of the editor window, there is a message: "Updates Not Installed Some updates could not be automatically." Below the editor is the R Console window, which shows the execution of the script. The output in the console includes:

```
Initial Guess
[1] -1.83 0.8
Original Equations - x_guess
3.9889 : should be 4
0.96041356775173 : should be 1
[1] [2]
[1,] -1.820072 0.8367488
[2,] 0.000000 1.0000000
[1] 4.000000 0.8379858
Current Guess
[1] -1.8124652 0.8379858
Original Equations - x_new
3.98725021975519 : should be 4
1.00123705001415 : should be 1
```

Figure 61. Quasi-linear, started near a solution, converges (sort-of) to the solution.

Figure 62 is a screen capture of the algorithm started near a solution, that fails to converge — it actually diverged.

```

QuasiLinear.R x
Source on Save | Run | Source | Addins | Updates Not
Some updates automatically.

1 # R script to solve non-linear example by quasi-linearization
2 Amat <- matrix(0, nrow=2, ncol=2)
3 Brhs <- numeric(0)
4 x_guess <- c(1, -1.7323)
5 maxiter <- 20
6 message("Initial Guess"); print(x_guess); message("Original Equations - x_guess ")
7 message( x_guess[1]^2 + x_guess[2]^2, " : should be 4 ")
8 message( exp(x_guess[1]) + x_guess[2], " : should be 1 ")
9 # Construct the current quasi-linear model
10 for (iter in 1:maxiter){
11   Amat[1,1] <- x_guess[1]; Amat[1,2] <- x_guess[2];
12   Amat[2,1] <- 0 ; Amat[2,2] <- 1;
13   Brhs[1] <- 4
14   Brhs[2] <- 1-exp(x_guess[1])
15   # Solve for the new guess
16   x_new <- solve(Amat,Brhs)
17   # Update
18   x_guess <- x_new
19 }
20 print(Amat); print(Brhs);
21 message("Current Guess"); print(x_new)
22 message("Original Equations - x_new")
23 message( x_new[1]^2 + x_new[2]^2, " : should be 4 ")
24 message( exp(x_new[1]) + x_new[2], " : should be 1 ")
4:24 (Top Level) : R Script

Console ~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/6-NonLinearSystems/RScripts/
> source("~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/6-NonLinearSystems/RScripts/QuasiLinear.R")
Initial Guess
[1] 1.0000 -1.7323
Original Equations - x_guess
4.00086329 : should be 4
0.985981828459045 : should be 1
[1,] -9.189108 0.3293203
[2,] 0.0000000 1.0000000
[1] 4.0000000 0.9998979
Current Guess
[1] -0.3994635 0.9998979
Original Equations - x_new
1.1593668337585 : should be 4
1.67057760019411 : should be 1
>

```

Figure 62. 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

$$\mathbf{f}(\mathbf{x}) = \begin{aligned} f_1(x, y) &= x^2 + y^2 - 4 \\ f_2(x, y) &= e^x + y - 1 \end{aligned} \quad (63)$$

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:

$$\mathbf{F}(\mathbf{x}) = \mathbf{f}(\mathbf{x})^T \cdot \mathbf{f}(\mathbf{x}) \quad (64)$$

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 recommended 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 63 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 64 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 65 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).

³²If 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.

```

1 # R script for system of non-linear equations using minimization
2 # WARNING -- This is not recommended for large systems
3 # forward define the functions
4 ##### f(x) #####
5 func <- function(x_vector){
6   func <- numeric(0)
7   func[1] <- x_vector[1]^2 + x_vector[2]^2 - 4
8   func[2] <- exp(x_vector[1]) + x_vector[2] - 1
9   return(func)
10 }
11 ##### F(x) #####
12 bigF <- function(x_vector){
13   vector <- numeric(0)
14   vector <- func(x_vector)
15   bigF <- t(vector) %*% vector
16   return(bigF)
17 }
18 ##### forward define some variables
19 # starting guess
20 x_guess <- c(-1.8162, 0.8374)
21 result <- nlm(bigF, x_guess)
22 message(" Estimated bigF Value : ",result$minimum)
23 message(" Estimated x_vector Value : ")
24 print(result$estimate)
25 message(" Estimated func Value : ")
26 print(func(result$estimate))
27 print(func(result$estimate))

27:29 | (Untitled) | R Script

Console ~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/6-NonLinearSystems/RScripts/
> source("~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/6-NonLinearSystems/RScripts/NLM-Method.R")
Estimated bigF Value : 5.6969192604848e-11
Estimated x_vector Value :
[1] -1.8162678 0.8373615
Estimated func Value :
[1] 2.999973e-06 -6.925991e-06
>

```

Figure 63. Solution using `nlm(...)`. Start vector (-1.8,0.8).

```

1 # R script for system of non-linear equations using minimization
2 # WARNING -- This is not recommended for large systems
3 # forward define the functions
4 ##### f(x) #####
5 func <- function(x_vector){
6   func <- numeric(0)
7   func[1] <- x_vector[1]^2 + x_vector[2]^2 - 4
8   func[2] <- exp(x_vector[1]) + x_vector[2] - 1
9   return(func)
10 }
11 ##### F(x) #####
12 bigF <- function(x_vector){
13   vector <- numeric(0)
14   vector <- func(x_vector)
15   bigF <- t(vector) %*% vector
16   return(bigF)
17 }
18 ##### forward define some variables
19 # starting guess
20 x_guess <- c(1,-1.7)
21 result <- nlm(bigF,x_guess)
22 message(" Estimated bigF Value : ",result$minimum)
23 message(" Estimated x_vector Value : ")
24 print(result$estimate)
25 message(" Estimated func Value : ")
26 print(func(result$estimate))
27 print(func(result$estimate))

21:17 | (Untitled) | R Script

Console ~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/6-NonLinearSystems/RScripts/
> source("~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/6-NonLinearSystems/RScripts/NLM-Method.R")
Estimated bigF Value : 5.12876234637262e-15
Estimated x_vector Value :
[1] 1.004169 -1.729637
Estimated func Value :
[1] -6.778064e-08 2.312027e-08
>

```

Figure 64. Solution using `nlm(...)`. Start vector (1.0,-1.7).

```

QuasiLinear.R NLM-Method.R
20 # starting guess
21 x.guess <- c(-2,-2)
22 message(" Initial Start Vector : ")
23 print(x.guess)
24 result <- nlm(bigF,x.guess)
25 message(" Estimated bigF Value : ",result$minimum)
26 message(" Estimated x_vector Value : ")
27 print(result$estimate)
21:15 (Untitled) 5

Console ~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PChinR-LectureNotes/6-NonLinearSystems/RScripts/
> source('~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PChinR-LectureNotes/6-NonLinearSystems/RScripts/NLM-Method.R')
Initial Start Vector :
[1] 1.0 -1.7
Estimated bigF Value : 5.12876234637262e-15
Estimated x_vector Value :
[1] 1.004169 -1.729637
Estimated func Value :
[1] -6.778064e-08 2.312027e-08
> source('~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PChinR-LectureNotes/6-NonLinearSystems/RScripts/NLM-Method.R')
Initial Start Vector :
[1] 2.0 -1.7
Estimated bigF Value : 5.6798320453443e-11
Estimated x_vector Value :
[1] -1.8162678 0.8373615
Estimated func Value :
[1] 3.001392e-06 -6.913029e-06
> source('~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PChinR-LectureNotes/6-NonLinearSystems/RScripts/NLM-Method.R')
Initial Start Vector :
[1] 2 2
Estimated bigF Value : 8.6677649029335e-16
Estimated x_vector Value :
[1] -1.8162641 0.8373678
Estimated func Value :
[1] 2.505832e-08 -1.545501e-08
> source('~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PChinR-LectureNotes/6-NonLinearSystems/RScripts/NLM-Method.R')
Initial Start Vector :
[1] -2 -2
Estimated bigF Value : 7.2563889410606e-15
Estimated x_vector Value :
[1] -1.8162649 0.8373678
Estimated func Value :
[1] -8.461297e-08 9.859585e-09
>

```

Figure 65. 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.

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 \mathbf{f} is a vector-valued function of a vector argument.

$$\begin{aligned}\mathbf{f}(\mathbf{x}) = f_1 &= x^2 &+ y^2 &- 4 \\ f_2 &= e^x &+ y &- 1\end{aligned}\quad (65)$$

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

$$x_{k+1} = x_k - \frac{f(x_k)}{\frac{df}{dx}|_{x_k}} \quad (66)$$

Extending to higher dimensions, the value x become the vector \mathbf{x} and the function $f()$ becomes the vector function $\mathbf{f}()$. 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 \mathbf{f} with respect to the arguments \mathbf{x} . For example for a 2-value function of 2 arguments (as our example above)

$$\frac{df}{dx}|_{x_k} \Rightarrow \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix} \quad (67)$$

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

$$\frac{1}{\frac{df}{dx}|_{x_k}} \Rightarrow \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix}^{-1} \quad (68)$$

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

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}(\mathbf{x})^{-1}|_{x_k} \cdot \mathbf{f}(\mathbf{x})|_{x_k} \quad (69)$$

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

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

$$\mathbf{x}_{k+1} - \mathbf{x}_k = -\mathbf{J}(\mathbf{x})^{-1}|_{x_k} \cdot \mathbf{f}(\mathbf{x})|_{x_k} \quad (70)$$

Next multiply both sides by the Jacobian.

$$\mathbf{J}(\mathbf{x})|_{x_k} \cdot (\mathbf{x}_{k+1} - \mathbf{x}_k) = -\mathbf{J}(\mathbf{x})|_{x_k} \cdot \mathbf{J}(\mathbf{x})^{-1}|_{x_k} \cdot \mathbf{f}(\mathbf{x})|_{x_k} \quad (71)$$

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

$$-\mathbf{J}(\mathbf{x})|_{x_k} \cdot (\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{f}(\mathbf{x})|_{x_k} \quad (72)$$

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)

$$\begin{aligned} \mathbf{f}(\mathbf{x}) = f_1 &= x^2 + y^2 - 4 \\ f_2 &= e^x + y - 1 \end{aligned} \quad (73)$$

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

$$\mathbf{J}(\mathbf{x}) => \begin{pmatrix} 2x & 2y \\ e^x & 1 \end{pmatrix} \quad (74)$$

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
}
##### 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 66 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.

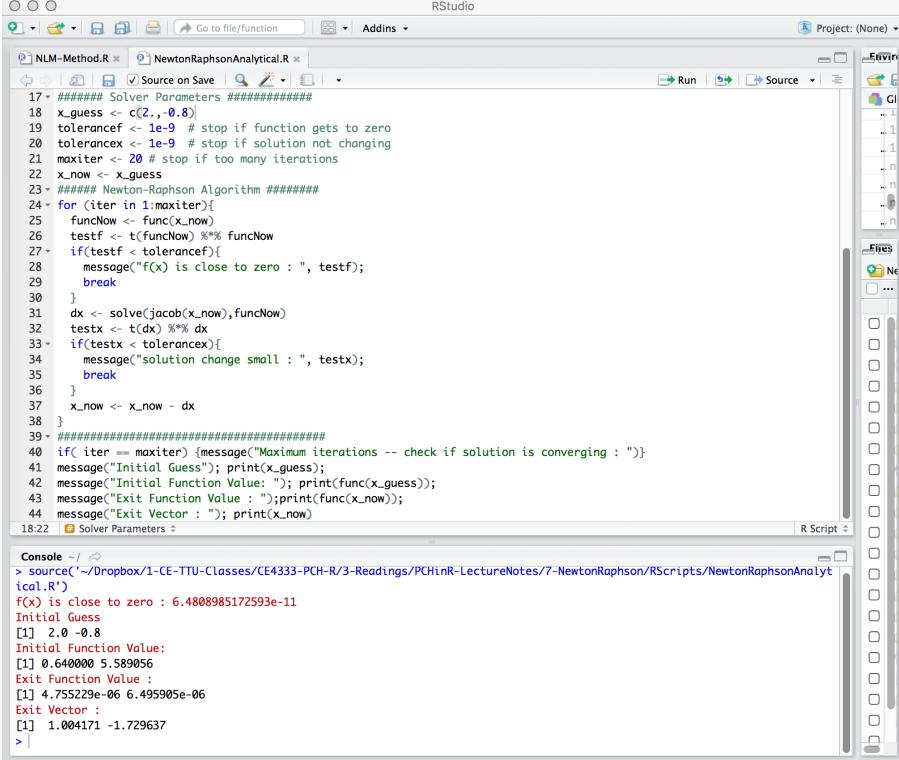
Listing 31. R code demonstrating Newton's Method calculations using finite-difference approximations to the partial derivatives.

```

# 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)

```



The screenshot shows the RStudio interface with the following details:

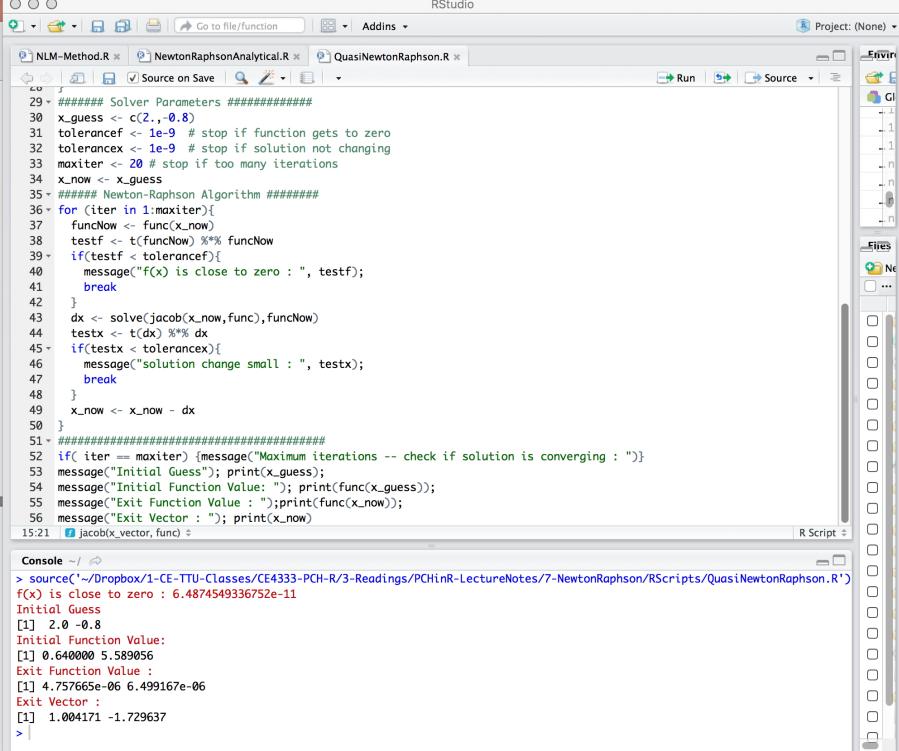
- Script Editor:** Displays the R script `NLM-Method.R` which contains the Newton-Raphson algorithm using analytical derivatives.
- Console:** Shows the execution of the script and its output. It includes initial guesses, function values, and the final exit vector.
- Environment:** Shows the current environment variables.
- Files:** Shows the project files including `NLM-Method.R`, `NewtonRaphsonAnalytical.R`, and `jac01.R`.

```

17 ###### Solver Parameters ######
18 x_guess <- c(2., -0.8)
19 tolerance <- 1e-9 # stop if function gets to zero
20 tolerancec <- 1e-9 # stop if solution not changing
21 maxiter <- 20 # stop if too many iterations
22 x_now <- x_guess
23 ##### Newton-Raphson Algorithm #####
24 for (iter in 1:maxiter){
25   funcNow <- func(x_now)
26   testf <- t(funcNow) %% funcNow
27   if(testf < tolerance){
28     message("f(x) is close to zero : ", testf);
29     break
30   }
31   dx <- solve(jacob(x_now),funcNow)
32   testx <- t(dx) %% dx
33   if(testx < tolerancec){
34     message("solution change small : ", testx);
35     break
36   }
37   x_now <- x_now - dx
38 }
39 ###### Check if converging #####
40 if( iter == maxiter) {message("Maximum iterations -- check if solution is converging : ")}
41 message("Initial Guess"); print(x_guess);
42 message("Initial Function Value: "); print(func(x_guess));
43 message("Exit Function Value : "); print(func(x_now));
44 message("Exit Vector : "); print(x_now)
45
18:22 [1] Solver Parameters

Console ~/ 
> source '~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/7-NewtonRaphson/RScripts/NewtonRaphsonAnalytical.R'
f(x) is close to zero : 6.4808985172593e-11
Initial Guess
[1] 2.0 -0.8
Initial Function Value:
[1] 0.6400000 5.589056
Exit Function Value :
[1] 4.755229e-06 6.495905e-06
Exit Vector :
[1] 1.004171 -1.729637
>

```

Figure 66. Newton-Raphson using Analytical Derivatives.


The screenshot shows the RStudio interface with the following details:

- Script Editor:** Displays the R script `NLM-Method.R` which contains the Newton-Raphson algorithm using finite-difference approximated derivatives.
- Console:** Shows the execution of the script and its output. It includes initial guesses, function values, and the final exit vector.
- Environment:** Shows the current environment variables.
- Files:** Shows the project files including `NLM-Method.R`, `NewtonRaphsonAnalytical.R`, and `QuasiNewtonRaphson.R`.

```

29 ###### Solver Parameters #####
30 x_guess <- c(2., -0.8)
31 tolerance <- 1e-9 # stop if function gets to zero
32 tolerancec <- 1e-9 # stop if solution not changing
33 maxiter <- 20 # stop if too many iterations
34 x_now <- x_guess
35 ##### Newton-Raphson Algorithm #####
36 for (iter in 1:maxiter){
37   funcNow <- func(x_now)
38   testf <- t(funcNow) %% funcNow
39   if(testf < tolerance){
40     message("f(x) is close to zero : ", testf);
41     break
42   }
43   dx <- solve(jacob(x_now,func),funcNow)
44   testx <- t(dx) %% dx
45   if(testx < tolerancec){
46     message("solution change small : ", testx);
47     break
48   }
49   x_now <- x_now - dx
50 }
51 ###### Check if converging #####
52 if( iter == maxiter) {message("Maximum iterations -- check if solution is converging : ")}
53 message("Initial Guess"); print(x_guess);
54 message("Initial Function Value: "); print(func(x_guess));
55 message("Exit Function Value : "); print(func(x_now));
56 message("Exit Vector : "); print(x_now)
15:21 [1] jacob0_vector,func

Console ~/ 
> source '~/Dropbox/1-CE-TTU-Classes/CE4333-PCH-R/3-Readings/PCHinR-LectureNotes/7-NewtonRaphson/RScripts/QuasiNewtonRaphson.R'
f(x) is close to zero : 6.4874549336752e-11
Initial Guess
[1] 2.0 -0.8
Initial Function Value:
[1] 0.6400000 5.589056
Exit Function Value :
[1] 4.757665e-06 6.499167e-06
Exit Vector :
[1] 1.004171 -1.729637
>

```

Figure 67. Newton-Raphson using Finite-Difference Approximated 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 Continuity (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 ,

$$Q = AV \quad (75)$$

where A is the cross sectional area of the pipe, and V is the mean section velocity. Typical units for discharge are 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 68 is

$$A_1 V_1 = A_2 V_2 \quad (76)$$

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

$$Q_1 - Q_2 - Q_3 - D = 0 \quad (77)$$

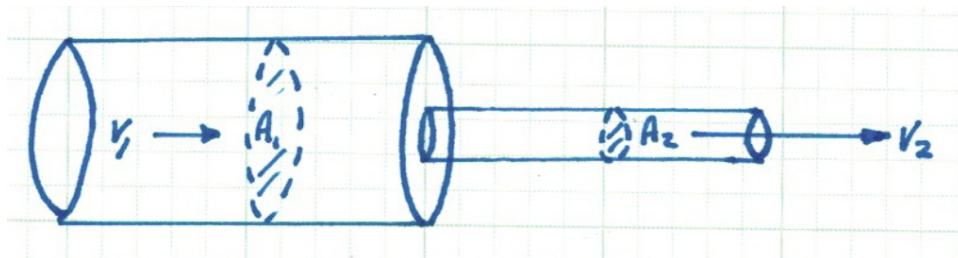


Figure 68. Continuity of mass (discharge) across a change in cross section.

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

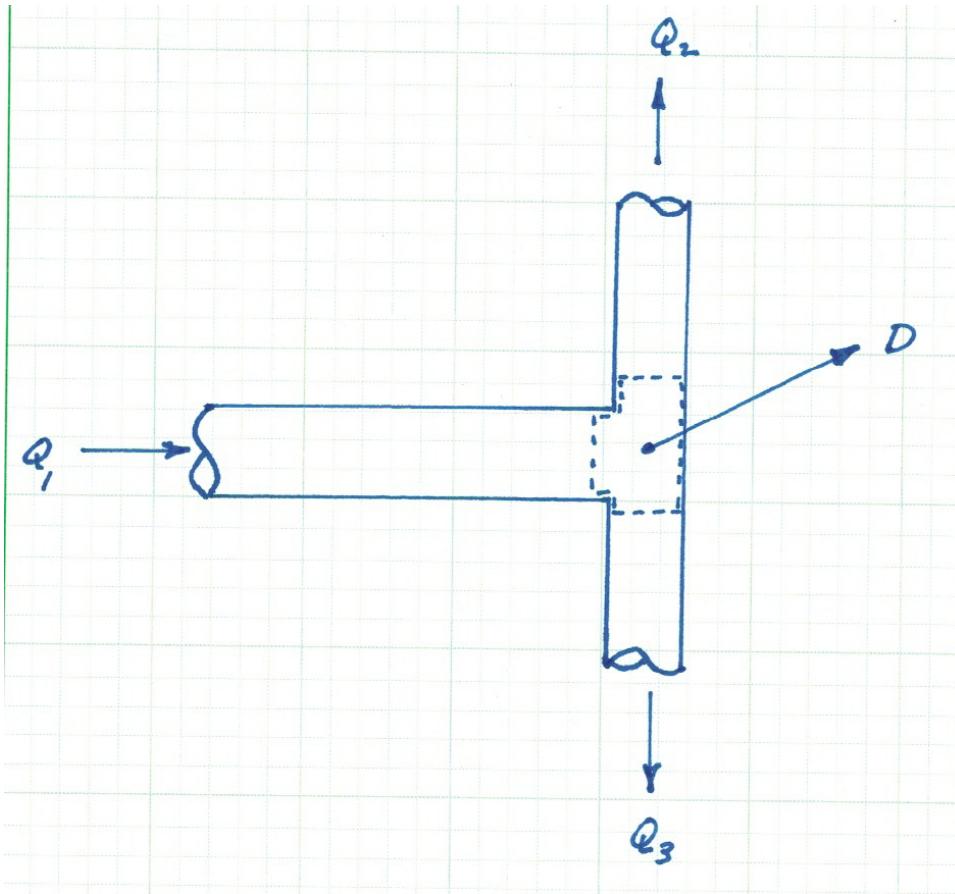


Figure 69. Continuity of mass (discharge) across a node (junction).

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.

$$\frac{p_1}{\rho g} + \alpha_1 \frac{V_1^2}{2g} + z_1 + h_p = \frac{p_2}{\rho g} + \alpha_2 \frac{V_2^2}{2g} + z_2 + h_t + h_l \quad (78)$$

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 77 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-

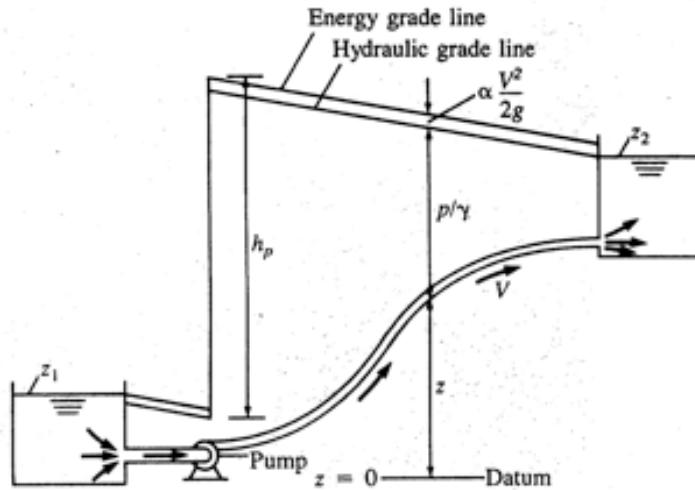


Figure 5-1 Definition sketch for terms in the energy equation

Figure 70. 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

$$\alpha = \frac{\int_A u^3 dA}{V^3 A} \quad (79)$$

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 102 is the relationship of mechanical power to added pump head.

$$\eta P = Q \rho g h_p \quad (80)$$

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 head³³ the pump curve is

$$h_p = \frac{\eta P}{Q\rho g} \quad (81)$$

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

$$h_p = H_{\text{shutoff}} - K_{\text{pump}} Q^{\text{exponent}} \quad (82)$$

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 109 is the power that can be recovered by a turbine (again using the concept of “water-to-wire” efficiency is

$$P = \eta Q \rho g h_t \quad (83)$$

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.

$$h_{L_f} = f \frac{L}{D} \frac{V^2}{2g} \quad (84)$$

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 $\frac{k_s}{D}$.

$$f = \sigma(Re_D, \frac{k_s}{D}) \quad (85)$$

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)) :

$$f = \frac{64}{Re_D} \quad (86)$$

³³A negative head loss!

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

$$\frac{1}{\sqrt{f}} = -2\log_{10}\left(\frac{2.51}{Re_d \sqrt{f}}\right) \quad (87)$$

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

$$\frac{1}{\sqrt{f}} = -2\log_{10}\left(\frac{\frac{k_e}{D}}{3.7}\right) \quad (88)$$

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

$$\frac{1}{\sqrt{f}} = -2\log_{10}\left(\frac{\frac{k_e}{D}}{3.7} + \frac{2.51}{Re_d \sqrt{f}}\right) \quad (89)$$

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

$$f = \frac{0.25}{\left[\log_{10}\left(\frac{\frac{k_e}{D}}{3.7} + \frac{5.74}{Re_d^{0.9}}\right)\right]^2} \quad (90)$$

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 71 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

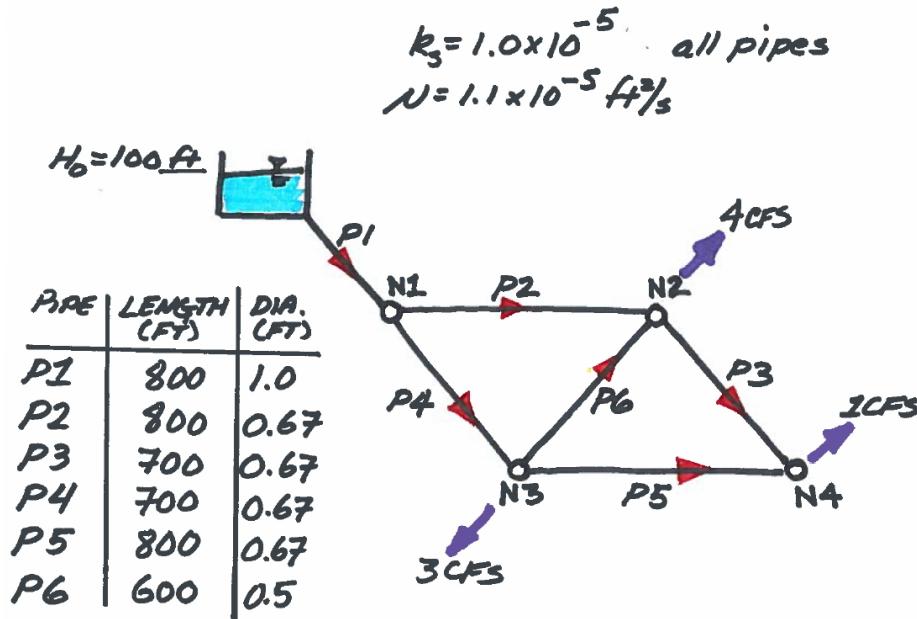


Figure 71. Pipe network for illustrative example with supply and demands identified. Pipe dimensions and diameters are also depicted..

(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 continuity, for instance for Node N2 the equation is

$$Q_2 - Q_3 + Q_6 = 4$$

Similarly 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 is³⁴

$$-L_2 Q_2 + H_1 - H_2 = 0$$

³⁴The 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 $\mathbf{A}(\mathbf{x})$, 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

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

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

$$\left(\begin{array}{cccccccccc} 1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline -L_1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -L_2 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -L_3 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -L_4 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -L_5 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -L_6 & 0 & -1 & 1 & 0 \end{array} \right) \begin{pmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \\ Q_5 \\ Q_6 \\ H_1 \\ H_2 \\ H_3 \\ H_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 3 \\ 1 \\ -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (91)$$

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.

³⁵The 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 than 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

$$\left(\begin{array}{cccccc|ccc} 1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline -L_1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & -L_2 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -L_3 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -L_4 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & -L_5 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -L_6 & 0 & -1 & 1 \end{array} \right) \begin{pmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \\ Q_5 \\ Q_6 \\ H_1 \\ H_2 \\ H_3 \\ H_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 3 \\ 1 \\ -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (92)$$

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 71.

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 72 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 72. 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 feet²/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,

$$Re(Q) = \frac{8L}{\mu\pi D} |Q| \quad (93)$$

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

$$f(k_s, D, Re) = \frac{0.25}{[\log(\frac{k_s}{3.7D} + \frac{5.74}{Re^{0.9}})]^2} \quad (94)$$

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

$$L_i = f \frac{8L_i}{\pi^2 g D_i^5} |Q_i| \quad (95)$$

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 $\mathbf{A}(\mathbf{x})$ 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 $\mathbf{A}(\mathbf{x})$ to implement the update as-per Newton-Raphson.

A brief review; at the solution we can write

$$[\mathbf{A}(\mathbf{x})] \cdot \mathbf{x} - \mathbf{b} = \mathbf{f}(\mathbf{x}) = \mathbf{0} \quad (96)$$

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

$$x_{k+1} = x_k - \left(\frac{df}{dx} |_{x_k} \right)^{-1} f(x_k) \quad (97)$$

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{f}(\mathbf{x}_k) \quad (98)$$

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.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}(\mathbf{x}_k)]^{-1}\{[\mathbf{A}(\mathbf{x}_k)] \cdot \mathbf{x}_k - \mathbf{b}\} \quad (99)$$

As a practical matter we actually never invert the Jacobian³⁶, instead we solve the related Linear system of

$$[\mathbf{J}(\mathbf{x}_k)] \cdot \Delta\mathbf{x} = \{[\mathbf{A}(\mathbf{x}_k)] \cdot \mathbf{x}_k - \mathbf{b}\} \quad (100)$$

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

³⁶Inverting 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 $\mathbf{A}(\mathbf{x})$ 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 working 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)
solveguess <- numeric(length=augmentedRowCount)
solvencnew <- numeric(length=augmentedRowCount)
solveguess[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
  jacobMatrix <- 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) jacobMatrix[i,j] <- 2*jacobMatrix[i,j]
    }
  }
  # now build the gq() vector
  gq <- augmentedMat %*% solvecguess - rhs_true
  gq
  dq <- solve(jacobMatrix,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 73 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.

The screenshot shows the RStudio interface with the following details:

- Script Editor:** The file `FindLoops2.R` is open, showing R code for reading input data from a file named `PipeNetwork.txt` and performing iterations. The code includes functions for geometric factor calculation and printing results.
- Console:** The console output shows the following text:


```
Update not changing -- exit loop and report current update
Iteration count = 14
> message("Current Results")
Current Results
> print(cbind(solvecgguess,gq,dq))
 [,1] [,2] [,3]
[1,] 8.000000e+00 0.000000e+00 6.929282e-29
[2,] 3.9008367 0.000000e+00 -8.366956e-15
[3,] 0.2640482 0.000000e+00 5.298824e-13
[4,] 4.0991633 0.000000e+00 8.366956e-15
[5,] 0.7359518 0.000000e+00 -5.298824e-13
[6,] 0.3632115 1.207923e-13 5.382494e-13
[7,] 84.5419558 -8.384404e-13 -2.677829e-28
[8,] 55.7088197 -9.237056e-14 2.896885e-15
[9,] 56.8972223 1.904255e-12 -2.048285e-14
[10,] 55.5154062 -3.545608e-12 6.506692e-14
> #print(cbind(friction,diameter,distance,velocity_pipe))
> |
```

Figure 73. Screen capture of R script for pipe network analysis.

9.2 Exercises

1. Figure 74 is a five-pipe network with a water supply source at Node 1, and demands at Nodes 1-5. Table 5 is a listing of the node and pipe data.

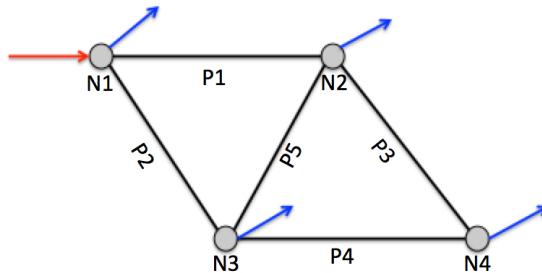


Figure 74. Layout of Simple Network.

Table 5. 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 74,
 - (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 75 and 76. Assume Node N1 has a total head of 300 feet.

In your solution you are to supply

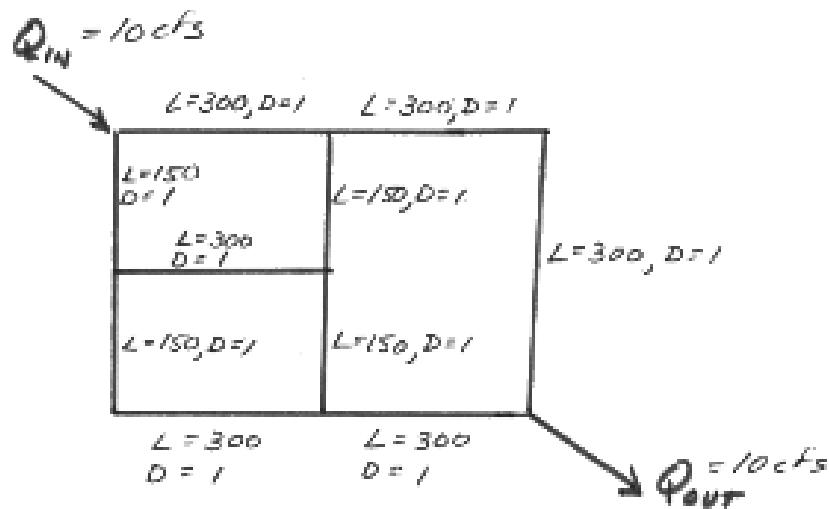


Figure 75. Pipe network for illustrative example with supply and demands identified. Pipe lengths (in feet) and diameters (in feet) are also depicted..

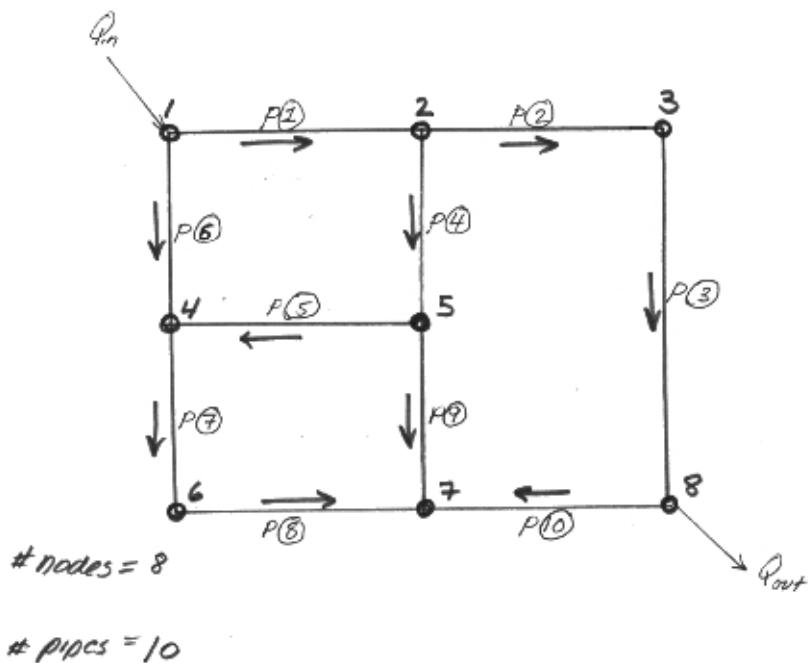


Figure 76. Pipe network for illustrative example with pipes and nodes labeled..

- An analysis showing the development of the node-arc incidence matrix based on the flow directions in Figure 76,
- The input file you constructed to provide the simulation values to your script, and
- 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.

10 Pumps and Valves

The addition of pumps, turbines, and valves increases some of the complexity for a network simulator. Valves and other fittings like elbows and such, that have a fixed setting are modeled as links and the resulting equations look much like pipe loss equations.

Pumps while also logically categorized as links are more complex because their head loss behavior is firstly negative – that is they add head to a flow system, and their ability to actually function is governed by their own performance curve. First we will review the modified Bernoulli equation again and then construct a prototype pump function to add to the program and simulate pump performance.

10.1 Energy Loss (along a link)

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

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

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

10.1.1 Added Head — Pumps

The head supplied by a pump is related to the mechanical power supplied to the flow. Equation 102 is the relationship of mechanical power to added pump head.

$$\eta P = Q \rho g h_p \quad (102)$$

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 head³⁷ the pump curve is

$$h_p = \frac{\eta P}{Q \rho g} \quad (103)$$

³⁷A negative head loss!

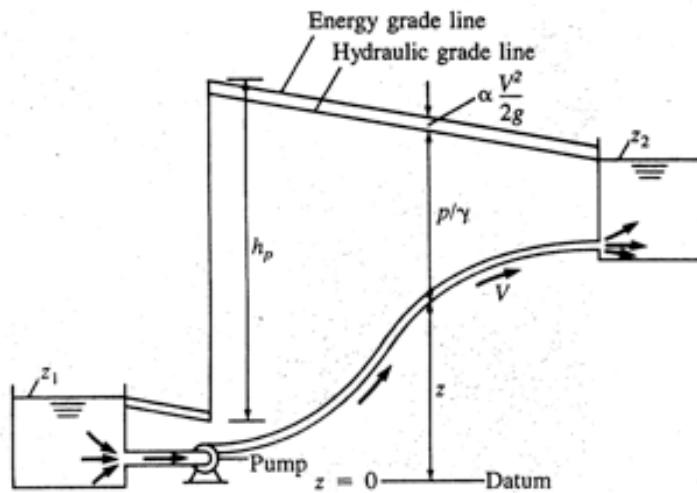


Figure 5-1 Definition sketch for terms in the energy equation

Figure 77. Definition sketch for energy equation.

Figure 78 is a typical pump curve depicting the kind of information available from a manufacturer of a pump.

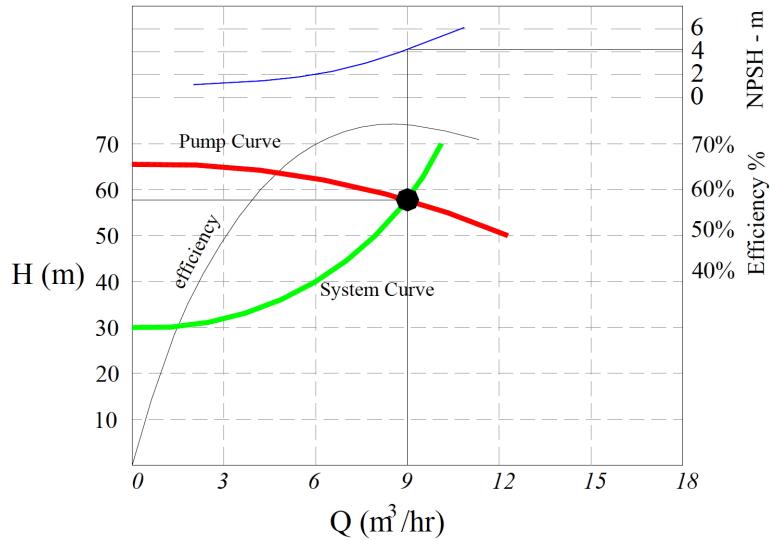


Figure 78. Pump Curve.

In introductory fluid mechanics we spend effort to match the pump curve to the system curve (head losses in our distribution system) and that match tells us how the pump-system combination should function. The pump curve relationship, as well as Equation 103, 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

$$h_p = H_{shutoff} - K_{pump} Q^{\text{exponent}} \quad (104)$$

In computational hydraulics we will need to represent the added as a head loss term (with opposite sign), and the functional form represented by Equation 104 is a good starting point. Practical (professional) programs will allow the curve to be represented in a tabular form and will use interpolation (just like our examples earlier) to specify the added head at a particular flow rate.

The next example will illustrate how to add pumps into the model.

Example 1: Pipe network with pumps

Figure 79 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 – in this example its head is set at zero. The remaining nodes (N1 – N4) have demands specified as the purple outflow arrows. The pipes are labeled (P2 – 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. The pump replaces

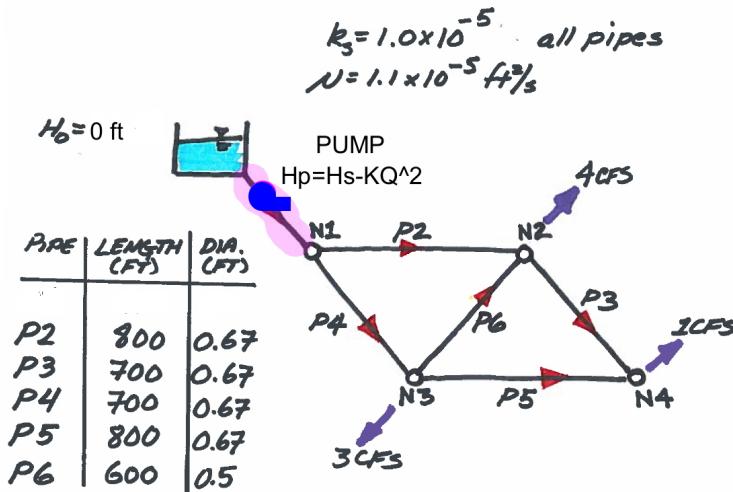


Figure 79. Pipe Network with a Pump.

pipe (P1) from the previous version of this example. We will use the observation that we really only need to identify which links are pumps, substitute in the correct added head component and then solve the system as in the earlier example.

We have to specify how the pump curve will be represented. In this example we will use a functional form.

$$h_p(Q) = H_{shutoff} - K_{pump} \times Q^n \quad (105)$$

For this example we will use the following numerical values for the pump function: $H_{shutoff} = 104.54$ feet, $K_{pump} = 0.25$ feet/ cfs^2 , and $n = 2$.

The $h_p(Q)$ is actually written as an added head factor, just like the friction factor, so we will use absolute values of flow so the term at each computational step can be placed in the augmented maxrix as if it were a head loss term; the solver will not know the difference.

The actual functional form employed is

$$h_p(Q) = [H_{shutoff}/|Q| - K_{pump} \times |Q|]Q \quad (106)$$

As before the sign of Q at the solution conveys flow direction. The program example does not trap the potential divide by zero error $H_{shutoff}/|Q|$, but one could test for zero flow, and just apply the shutoff head. Listing 36 implements the prototype function described above.

Listing 36. R Code to pump prototype function

```
.....
# Pump Curve factor function
p_factor <- function(shutoff,constant,exponent,flow){
  p_factor <- shutoff/abs(flow) - constant*abs(flow^(exponent-1))
  return(p_factor)
}
```

Next we have to read in the pump characteristics, I decided to just have pumps replace links (so I won't have to rebuild a node-arc-incidence matrix), so the pump characteristics are

1. Link ID – the index of the pipe that is replaced by a pump.
2. Shutoff head.
3. K_{pump} .
4. Exponent on the pump curve, n . Typically it will be larger than 1.0.

Listing 37 implements the reads from the input file, and builds the pump matrix.

Listing 37. R Code to include pumps in a pipeline network

```
# Read Input Data Stream from File
zz <- file("PipeNetwork.txt", "r") # Open a connection named zz to file named PipeNetwork.
  txt
pumpCount <- as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE, encoding = "unknown",
  skipNul = FALSE))
nodeCount <- as.numeric(readLines(zz, n = 1, ok = TRUE, warn = TRUE, encoding = "unknown",
  skipNul = FALSE))
.....
rhs_true <- (readLines(zz, n = 1, ok = TRUE, warn = TRUE, encoding = "unknown", skipNul =
  FALSE))
pumps <- (readLines(zz, n = pumpCount, ok = TRUE, warn = TRUE, encoding = "unknown", skipNul =
  FALSE))
close(zz) # Close connection zz
.....
pumps <- as.numeric(unlist(strsplit(pumps, 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)
pumps <- matrix(pumps, nrow=pumpCount, ncol=4, byrow=TRUE)
.....
```

Next we will have to compute the added head factor at each step, just like the friction factor, and we will overwrite the pipe that the pump replaces.³⁸

Listing 38 implements the computation of the added head factor, and the pump selection factor.

Listing 38. R Code to include pumps in a pipeline network

```
.....
# compute the current pump factor
if(pumpCount > 0){
  for (i in 1:pumpCount)
    {
      addedhead[i] <- p_factor(pumps[i,2],pumps[i,3],pumps[i,4],flowguess[pumps[i,1]])
    }
}
# 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];
    if(pumpCount > 0){
      for(ipump in 1:pumpCount) {
        if(j == pumps[ipump,1]) augmentedMat[i,j] <- addedhead[ipump]
      }
    }
  }
}
# print(augmentedMat)
.....
```

The remainder of the code is unchanged. Listing 39 illustrates the changes in the input file. We have added a row to indicate how many pumps will be used as the first record in the file. The last record after the right-hand side vector is the pump characteristics; one row for each pump. The scripts also test if there are zero pumps and skip code as needed. Observe we still preserve Link #1 data because its part of the node-arc matrix, but the length and diameter of the link is irrelevant (but need to be non-zero because we compute friction factors as if there were a pipe, but never use them).

Listing 39. Input file with pumps at link#1 in a pipeline network

```
1 <== how many pumps
4
6
1.00 0.67 0.67 0.67 0.67 0.5 <== link #1 needs values as placeholders, but are not used
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 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 0 0 0 0 0 0
1 100.54 0.25 2.0 <== Pump Link ID, H\_shutoff, K\_pump, Exponent
```

Figure 80 is a screen capture of the example problem run in **R Studio**. The script produces the correct flow values, and the pump specified was intended to match the

³⁸This approach is decidedly a hack for illustration purposes. A more advanced program would probably just treat everything as a link and use a similar database build structure to determine if a link is a head loss or head add link. My reasoning is that there will be fewer pumps than pipes in any system, so overwriting a fictitious pipe is not too much trouble.

The screenshot shows the RStudio interface with the following details:

- Script Editor:** Displays the R script `PipeLineModel-Pumps.R`. The code is a loop that iterates over a pipe network, solving for flow variables and reporting results.
- Console:** Shows the execution of the script and its output. The output includes:
 - Node Count = 4
 - Pipe Count = 6
 - Pump Count = 1
 - Update not changing -- exit loop and report current update
 - Iteration count = 14
 - Current Results (displayed as a matrix of values)
 - Pump # 1 at Q = 8

Figure 80. Pipe Network with a Pump.

previous problem closely in that it produces enough head so that node N1 has nearly the same head value as the problem without a pump.

10.1.2 Fitting (Minor) Losses

In addition to head loss in the conduit, other losses are created by inlets, outlets, transitions, and other connections in the system. In fact such losses can be used to measure discharge (think of the orifice plate in the fluids laboratory). The fittings create additional turbulence that generates heat and produces the head loss.

Equation 107 is the typical loss model

$$h_{minor} = K \frac{V^2}{2g} \quad (107)$$

where K is called a minor loss coefficient, and is tabulated (e.g. Table 6) for various kinds of fittings.

The use is straightforward, and multiple fittings are summed in the loss term in the energy equation. In practical computation, these losses make the most sense when

Table 6. Minor Loss Coefficients for Different Fittings.

Fitting Type	<i>K</i>
Tee, Flanged, Line Flow	0.2
Tee, Threaded, Line Flow	0.9
Tee, Flanged, Branched Flow	1.0
Tee, Threaded , Branch Flow	2.0
Union, Threaded	0.08
Elbow, Flanged Regular 90°	0.3
Elbow, Threaded Regular 90°	1.5
Elbow, Threaded Regular 45°	0.4
Elbow, Flanged Long Radius 90°	0.2
Elbow, Threaded Long Radius 90°	0.7
Elbow, Flanged Long Radius 45°	0.2
Return Bend, Flanged 180°	0.2
Return Bend, Threaded 180°	1.5
Globe Valve, Fully Open	10
Angle Valve, Fully Open	2
Gate Valve, Fully Open	0.15
Gate Valve, 1/4 Closed	0.26
Gate Valve, 1/2 Closed	2.1
Gate Valve, 3/4 Closed	17
Swing Check Valve, Forward Flow	2
Ball Valve, Fully Open	0.05
Ball Valve, 1/3 Closed	5.5
Ball Valve, 2/3 Closed	200
Diaphragm Valve, Open	2.3
Diaphragm Valve, Half Open	4.3
Diaphragm Valve, 1/4 Open	21
Water meter	7

associated with a particular pipe. If we rewrite the loss equation

$$h_{minor} = \frac{K}{2g} \frac{16Q^2}{\pi^2 D^4} \quad (108)$$

we see that these terms can be added to a pipe either as an additional loss term and placed in the augmented matrix in the same way as the other loss term.

10.1.3 Extracted Head — Turbines

The head recovered by a turbine is also an “added head” but appears on the loss side of the equation. Equation 109 is the power that can be recovered by a turbine (again using the concept of “water-to-wire” efficiency is

$$P = \eta Q \rho g h_t \quad (109)$$

An approach similar to pumps would be employed — the effort in all these cases is to represent the hydraulic components as a loss factor so the non-linear solver we have already built can be used.

11 Pipeline Transients — Water Hammer

Unsteady flow in closed conduits is important for estimating the forces involved from a sudden change in discharge from a pump failing (or starting), or the closing (or opening) of a valve. The flow variation will create a pressure wave traveling along the pipe.³⁹ The computational goal is to estimate the magnitude and timing of these extreme pressures to evaluate the safety of the conduit, or design a pump shutdown (or startup) or valve operation protocol to control these extreme pressures to some acceptable magnitude.

11.1 Analysis

If the conduit walls are treated as elastic material, and the liquid is compressible the velocity of a wave along the conduit is

$$c = \sqrt{\frac{1}{\rho(\frac{1}{E_f} + \frac{D}{E_c \cdot e})}} \quad (110)$$

The value c is called the celerity, E_f is the fluid modulus of elasticity, D is the conduit diameter, E_c is the pipe material modulus of elasticity, e is the conduit wall thickness. Typical values for water are $E_f = 2.2 \times 10^9$ Pa and for steel $E_c = 160 \times 10^9$ Pa.

The pipe is approximated as a series of steel rings (all in line, negligible Poisson ratio) with uniform internal pressure in each ring (but can be different for adjacent rings). The relative pressure head inside the pipe at any given ring is related to the pipe diameter as

$$H = \frac{2c^2}{g} \times \frac{\frac{d}{2} - \frac{d_0}{2}}{\frac{d}{2}} \quad (111)$$

where $\frac{d}{2}$ is the radius under pressure head H and $\frac{d_0}{2}$ is the radius when pressure head is zero gage.⁴⁰

The continuity equation is written for each ring, and a force balance is written between each ring.⁴¹

³⁹These waves will travel in alternating directions as they find boundaries as each end of the disturbed discharge conduit – in some sense the pipe becomes a resonant chamber. Over time the magnitude of the waves will decrease as friction dissipates the energy. During these transients high and low pressures are applied to the pipe walls and fixtures and can conceivably damage them.

⁴⁰1 atmosphere absolute. It is usually easier to work with gage pressure in practice.

⁴¹The time to drain problem is mildly similar the tanks represent a ring where the change in depth (head) is related to outflow velocity; the outflow velocity is related to force at the outlet – in that case the pressure force of the water above the outlet; Bernoulli's equation for that case simplified the work considerably. Here we will have a series of tanks (rings) that communicate pressure head and velocity between them. We will arrive at a staggered spatial and temporal grid.