MATH 376: Numerical Analysis

Final Project

Quan Vu

December 13, 2017

Chapter 1

Introduction

1.1 Structure of the book

This book is intended to consolidate the knowledge that I've gathered through my time in the course MATH376: Numerical Analysis at Colgate University. I dedicate a chapter before each project to explain some of the numerical methods employed in the project itself.

1.2 Acknowledgements

I would like to take this opportunity to thank professor Seo Gunog for organizing and teaching this class. All the knowledge that I have gathered in this book is due to her teaching and guidance inside and outside class. Being a computer science and mathematics double major, I was able to find a very meaningful connection between the two subjects through this course, and altogether it has been a wonderful journey.

I would also like to take this opportunity to thank my fellow classmates, Kevin Wibisono and Ruoyu (Tony) Guo, for always being there to support me throughout this course.

Chapter 2

Root finding methods - Bisection

2.1 Method explanation

Imagine that you have a dictionary, and you are trying to find a definition of a word. What would you likely do? I myself would open the dictionary, and look at the page that I opened. If the words in those page come before the word that I'm looking for, I try to find a page between that current page and the end. If not, I do the opposite and find a page between the beginning and the current page. I repeat the process until I find the desired page/word.

The bisection method works by employing similar intuition. The assumption is that we have a function whose root we want to determine, and we know for a fact that the function changes sign around this root. Then by using 2 initial guesses, acting as the beginning and the end of the dictionary, we look for a potential root in the middle. If this is the root that we want, then we just return from the process. Otherwise, we update the boundary of our "dictionary", and repeat the process until we find a satisfiable answer.

2.2 Method formalization

Step 1: Rewrite the equation in the form $f(x) = 0$

Step 2: Plot the graph of the function to determine the rough estimation of the desired root.

Step 3: Choose initial guesses a and b around the root, such that the function changes sign around this interval, i.e $f(a) \times f(b) < 0$

Step 4: Find the estimated root, $x_i = \frac{a+b}{2}$

Step 5: If $f(x_i) = 0$, we're done, and we can return. Otherwise depending on the sign of $f(x_i)$, we update a and b accordingly. If $f(x_i)$ has the same sign as $f(a)$, then $a = x_i$, else $b = x_i$). Return to step 4.

2.3 Error discussion

We assume that our next estimation is the real solution. From such assumption, it's easy to see that we are dividing the error in half by every time we are making a new estimation. The upper bound for the bisection method is $\frac{b-a}{2^(n+1)}$ where n is the number of iteration.

Chapter 3

Greenhouse gases and pH

3.1 Abstract

This chapter aims at examining the relationship between the steady rise in atmospheric levels of several greenhouse gases and the pH of rainwater within the corresponding areas. In particular, this project looks at the annual levels of atmospheric carbon dioxide (CO_2) from the year 1959 to 2016 in Mauna Loa, Hawaii. By computing the pH of water using the given data, it can be shown that the pH of rainwater has decreased from 5.63 to 5.58 over the years, and the trend shows that the pH level will likely drop lower.

3.2 Introduction

3.2.1 Background information

It is well documented that the atmospheric levels of several greenhouse gases have been increasing over the past 57 years. It is also know that within areas with generally low human activities, carbon dioxide is the primary determinant of the pH of rainwater.

3.2.2 Problem description

This project aims at using the existing data regarding the levels of atmospheric CO_2 around Mauna Loa to calculate the pH of rainwater in the same region over the years. This can be done with the help of five equations governing the chemistry of rainwater:

$$K_1 = \frac{10^6[H^+][HCO_3^-]}{K_H CO_2} \tag{1}$$

$$K_2 = \frac{[H^+][CO_3^{-2}]}{[HCO_3^-]} \tag{2}$$

$$K_\omega = [H^+][OH^-] \tag{3}$$

$$c_T = \frac{K_H CO_2}{10^6} + [HCO_3^-] + [CO_3^{-2}] \tag{4}$$

$$0 = [HCO_3^-] + 2[CO_3^{-2}] + [OH^-] - [H^+] \tag{5}$$

where K_H is Henry's constant, K_1, K_2 and K_ω are equilibrium coefficients. The five unknowns are $c_T =$ total inorganic carbon, $HCO_3^- =$ bicarbonate, $[CO_3^{-2}] =$ carbonate, $[H^+] =$ hydrogen ion, $[OH^-] =$ hydroxyl ion. One major assumption with this approach is that we fix CO_2 as the sole factor contributing to the pH of rainwater. In reality, many other greenhouse gases can also contribute to the fluctuation of pH.

3.2.3 Outline

Given the values $K_H = 10^{-1.46}$, $K_1 = 10^{-6.3}$, $K_2 = 10^{-10.3}$, $K_\omega = 10^{-14}$ and the annual CO_2, we can reduce equation (5) to be one that is in terms of $[H^+]$. We can compute $[H^+]$ and calculate the pH of rainwater using the equation:

$$pH = -log_{10}[H^+] \tag{6}$$

3.3 Numerical method

Firstly we convert the given equations so that the unknowns in (5) can be expressed in terms of $[H^+]$.
From (1):

$$[HCO_3^-] = \frac{K_H K_1 CO_2}{10^6[H^+]} \tag{1a}$$

From (2) and from (1a):

$$[CO_3^{-2}] = \frac{K_2[HCO_3^-]}{[H^+]} = \frac{K_H K_1 K_2 CO_2}{10^6[H^+]^2} \tag{2a}$$

From (3):

$$[OH^-] = \frac{K_\omega}{[H^+]} \tag{3a}$$

From (4), (2a) and (3a):

$$c_T = \frac{K_H CO_2}{10^6} + \frac{K_H K_1 CO_2}{10^6[H^+]} + \frac{K_H K_1 K_2 CO_2}{10^6[H^+]^2} \tag{4a}$$

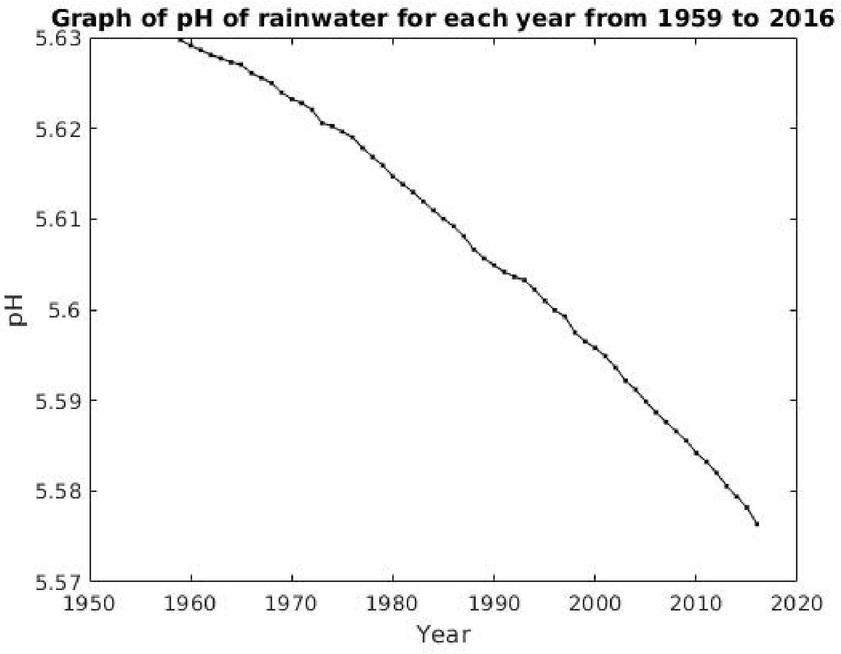
From 5, and from (1a), (2a), (3a):

$$0 = \frac{K_H K_1 CO_2 + 10^6 K_\omega}{10^6[H^+]} + \frac{2K_H K_1 K_2 CO_2}{10^6[H^+]^2}] - [H^+] \tag{5a}$$

By using the bisection method on the above equation, we can find $[H^+]$.
A short snippet of Matlab code is attached in the appendix for this section to show how the bisection method finds the root of the function.

3.4 Results

Using the methodology discussed above, we obtain the following results of the pH levels in rainwater in Mauna Loa, from 1959 to 2016:



For detailed results, consult the csv file in the project directory.

3.5 Discussion

From the calculations taken, the pH in the year 1959 was 5.63, and the pH calculated in 2016 was 5.58. While the decline seems small, we must keep in mind that pH is calculated by taking $log_{10}[H^{+}]$. This means that the concentration of Hydrogen ions in rainwater has increased over the year, and the trend does not seem to be stopping. In fact this trend can be modeled using the polyfit function in Matlab to give:

$$pH = -5.523 \times 10^{-6} t^{2} + 0.02101t - 14.33$$

where t denotes the year. If the trend continues, the pH of rainwater will drop below the threshold of 5.0, creating acid rain.

An interesting observation is that pH seems to drop with the rise of CO_2. Such investigation can be carried out in further researches to find a direct relationship between these two. The total inorganic carbon was also not discussed within the scope of this project, but at the same time it might be interesting to look into this. Presumably, with the rise in CO_2, c_T should rise as well, showing that there is an abundance in inorganic carbon intake.

3.6 Bibliography

Data for annual atmospheric levels of CO_2 are taken from:
https://www.esrl.noaa.gov/gmd/ccgg/trends/data.html

3.7 Appendix for this chapter

3.7.1 Code for the bisection method

```
function [xm] = bisectM(fun, xleft, xright, n, TOL)
    a = xleft;
    b = xright;
    fa = feval(fun, a);
    fb = feval(fun, b);

    fprintf('n \t approximation \n');

    for i = 1:n
        xm(i) = (a + b)./2;
        fm = feval(fun, xm(i));

        fprintf('%d \t %12.10f \n', i - 1, xm(i))

        if (sign(fm) == sign(fa))
            a = xm(i);
            fa = fm;
        else
            b = xm(i);
            fb = fm;
        end

        if (i >= 2)
            absE = abs(xm(i) - xm(i-1));
            if absE < TOL
                break;
            end
        end
    end
end
```

3.7.2 Code for processing data

```
clear all;
close all;
clc;

format long;

% File path is hard-coded. Please put the data file in
    the same folder
% for this to work
fid = fopen('MaunaLoa.dat','r');
data = textscan(fid, '%f%f%f', 'HeaderLines', 22);
year = data{1};
mean = data{2};
unc = data{3};
NO_OF_YEAR = length(year);

pH = zeros([1 NO_OF_YEAR]);

% Defining constants to use
KH = 10^(-1.46);
K1 = 10^(-6.3);
K2 = 10^(-10.3);
Ko = 10^(-14);

% Since pH lies between 2 and 12, [H+] lies between 1e-12
     and 1e-2
upperBound = 1e-2;
lowerBound = 1e-12;

% Tolerance of 1e^-10 should be enough. This is TOL for [
    H+].
% Having 10 correct decimal places for [H+] ensures at
    least 2 correct for pH
% (as we are taking log10, a small change in [H+] leads
    to an even smaller
% change in pH). We are taking 1e-10 since our search
    space is inherently
% small, and taking 1e-2 would terminate the search
    prematurely
TOL = 1e-10;

% Loop through the year, extract current level of CO2,
    and calculate pH
for i = 1:NO_OF_YEAR
    currentYear = year(i);
    CO2 = mean(i);
    f = @(x) (KH * K1 * CO2 + 10^6 * Ko) ./ (x * 10^6) +
        (2 * KH * K1 * K2 * CO2) ./ (x.^2 * 10^6) - x;
    pH(i) = -log10(bisectM(f, lowerBound, upperBound, 300,
        TOL));
end

pH(1)
pH(NO_OF_YEAR)
x = linspace(1959, 2016, NO_OF_YEAR);
fid = fopen('output.csv', 'wt');
fprintf(fid, '%s,%s\n', 'Year','pH');
fclose(fid);
output = horzcat(x(:), pH(:));
dlmwrite('output.csv', output, 'delimiter', ',','-append'
    );
polyfit(x, pH, 2)
plot(x, pH, 'k.-');
xlabel('Year');
ylabel('pH');
title('Graph of pH of rainwater for each year from 1959
    to 2016');

function result = bisectM(fun, xleft, xright, n, TOL)
    a = xleft;
    b = xright;
    fa = feval(fun, a);
    fb = feval(fun, b);

    %fprintf('n \t approximation \n');

    for i = 1:n
        xm(i) = (a + b)./2;
        result = xm(i);
        fm = feval(fun, xm(i));

        %fprintf('%d \t %12.12f \n', i - 1, xm(i))

        if (sign(fm) == sign(fa))
            a = xm(i);
            fa = fm;
        else
            b = xm(i);
            fb = fm;
        end

        if (i >= 2)
```

```
            absE = abs(xm(i) - xm(i-1));
            if absE < TOL
                return;
            end
        end
    end
end
```

Chapter 4

More root finding methods

4.1 Introduction

The bisection method has its merits in the sense that we are always guaranteed to find an answer to the problem, given that our assumptions are correct. The downside is that it can be slow to converge, and that if the function does not change signs around the root, this method is not employable.
In this section, we explore a few other root finding techniques, which are the method of false position, Newton's method and the method of fixed point iteration.

4.2 Method of false position

4.2.1 Method explanation

The method of false position fixes the shortcoming of the bisection method. The method starts out with the same assumption as the bisection method. We need 2 initial guesses and also we assume that the function changes sign around the roots. The method estimates the root by assuming that the function is the straight line connecting the two points evaluated at the estimations, and find the root of that straight line. Depending on the sign of the function at the new estimation, we update the endpoints accordingly.

4.2.2 Method formalization

Step 1: Rewrite the equation in the for $f(x) = 0$

Step 2: Plot the graph of the function to determine the rough estimation of the desired root.

Step 3: Choose initial guesses a and b around the root, such that the function changes sign around this interval, i.e $f(a) \times f(b) < 0$

Step 4: Find the estimate of the root, determined by

$$x_i = a - f(a)\frac{b-a}{f(b)-f(a)}$$

Step 5: If $f(x_i) = 0$, we're done, and we can return. Otherwise depending on the sign of $f(x_i)$, we update a and b accordingly. If $f(x_i)$ has the same sign as $f(a)$, then $a = x_i$, else $b = x_i$). Return to step 4.

4.2.3 Method discussion

The method of false position assures convergence if the function changes signs around the desired root. It may converge faster than the bisection method, since it assumes takes the value of the functions at the estimations into account, instead of just mindlessly assumes that the midpoint is a good estimation for the root.
One problem with the false position method is that it may happen that all of the approximations lie on one side of the root, and therefore it converges very slowly for functions with significant curvature.

4.3 Newton's method

4.3.1 Method explanation

Newton's method is the first method that we encounter that does not rely on having 2 estimations. By employing only one estimation at the beginning, we can iteratively find a better estimate for the solution. The method assumes that the function is linear, and as such the root of the tangent function at our estimate would give us the value of the actual root of the function. The value gets updated iteratively until the error is within tolerance.

4.3.2 Method formalization

Step 1: Plot the graph of the function f.

Step 2: Start with an initial guess of a desired root, x_0, which is chosen from the graph of f

Step 3: Draw the tangent line to the curve $y = f(x)$ at the point $(x_0), f(x_0)$

Step 4: Find the x-intercept of the tangent line and label it x_1

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Step 5: Repeat steps 3 and 4 by replacing the old estimation with the new estimation, until the error is within acceptance criteria

4.3.3 Method discussion

We have the following theorem;

a If x_R is a simple root, convergence is quadratic (or second order), and $|E_{n+1}| \approx \frac{|f''(x_R)|}{2|f'(x_R)|}|E_n|^2$

b If x_R is a multiple root of order k, convergence is linear and $|E_{n+1}| \approx \frac{k-1}{k}|E_n|$

4.4 Fixed point iteration method

4.4.1 Method explanation

The premise of this method based solely on whether we can represent the original equation $f(x)$ in the form $g(x) = x$. Then by employing an initial guess, we iteratively compute the value of g at the guess, and that would give us an update on the estimation for the value of the root.

4.4.2 Method formalization

Step 1: Find an iteration function $g(x)$ by rewriting $f(x)$ in the form $x = g(x)$

Step 2: Choose an initial approximation for the root, x_0

Step 3: Using the iteration formula, obtain $x_1 = g(x_0)$

Step 4: Repeat step 4 until we find a solution that false within tolerance

4.4.3 Method discussion

The method of fixed point iteration's beauty lies in its simplicity. However at the same time, the method is not guaranteed to converge. It is a possibility that the iteration function was chosen badly, and as such the approximations diverge away from the desired root.

We have the following theorem:

Let $g \in C[a, b]$ be such that $g(x) \in [a, b] \forall x \in [a, b]$. Suppose that in addition, that g' exists on (a, b).Then the iterations are convergent on the interval $[a, b]$ if the interval contains a root and if $|g'(x)| < 1 \forall x \in [a, b]$

Chapter 5

Fluid flows in pipes and tubes

5.1 Abstract

This chapter aims at examining the flow of fluid through pipes and tubes. The project computes the dimensionless *friction factor* in turbulent flows. By computing this factor using the bisection method, the false position method, Newton's method and lastly the fixed point iteration method, this paper draws out comparisons between how effective these methods are and what are the constraints associated with them.

5.2 Introduction

5.2.1 Background information

Determining the friction factor is of great relevance to many field of engineering and science. Some of these include the flow of liquid and gases through pipelines and cooling systems. Scientists are interested in topics ranging from the flow in blood vessels to nutrient transmission through a plant's vascular system.

5.2.2 Problem description

In turbulent flows, the *Colebrook equation* provides a means to calculate the friction factor using the equation

$$0 = \frac{1}{\sqrt{f}} + 2.0\log_{10}\left(\frac{\varepsilon}{3.7D} + \frac{2.51}{Re\sqrt{f}}\right)$$

where ε is the roughness (m), D is the diameter (m), and Re is the Reynold's number, as calculated by

$$Re = \frac{\rho V D}{\mu}$$

where ρ is the fluid's density (kg/m^3), V is the velocity (m/s), and μ is the dynamic viscosity $(N.s/m^2)$

5.2.3 Outline

By computing Reynold's number using the given values, we can substitute it back into the equation and using the numerical methods discussed, we can find the value of the friction factor. The given values are $\rho = 1.23kg/m^3$, $\mu = 1.79 \times 10^{-5}N.s/m^2$, $D = 0.005m$, $V = 40m/s$, and $\varepsilon = 0.0015mm$

5.3 Numerical methods

The numerical methods involved in the calculation of the friction factor are the bisection method, false position method, Newton's method, and fixed-point iteration method. Consult the attached Matlab project file to see the details of the calculations

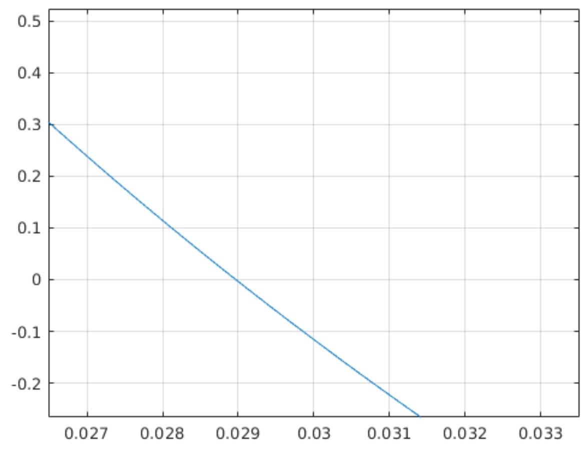
5.4 Results

5.4.1 Reynold's number

Using Matlab, we are able to determine Reynold's number $= 1.374 \times 10^4$

5.4.2 Plotting graph and estimation

Using Matlab to plot the graph, the friction factor seems to be somewhere around 0.029



5.4.3 Bisection method and False position method

For both of these methods, we approximate the value of the friction factor by using a = 0.008 and b = 0.08, with a tolerance of 10^{-8}

Bisection method

The method took 26 iterations to calculate the answer. The estimations converge to 0.02896781

False position method

The method took 31 iterations to calculate the answer. The estimations converge to 0.02896782

Discussion

We see that both method converges to the solution, however the bisection method is faster in this case.

5.4.4 Newton's method

We can calculate the derivative of the function quite easily:

$$h'(f) = \frac{-1}{2f\sqrt{f}} + 2.0\left(\frac{log_{10}e \times \frac{-2.51}{2Ref\sqrt{f}}}{\frac{\varepsilon}{3.7D} + \frac{2.51}{Re\sqrt{f}}}\right)$$

This is then used for the calculation for Newton's method in the Matlab file provided in the appendix.

With initial guess 0.008

Took 6 iteration to calculate the answer. The estimations converge to 0.02896781

With initial guess 0.08

The approximations do not converge. They fluctuate and eventually go to infinity

Discussion

We see that Newton's method converges extremely fast if we choose the correct initial guess. This is due to the nature of the function around the root.

5.4.5 Using fzero

We use the built-in fzero function with options = optimset('Display','iter', 'TolX', 1e-8).This function searches for a point near the guess where the sign of the function changes

With initial guess 0.008

The method does not converge to the solution. Complex function value encountered during search

With initial guess 0.08

The method manages to find an interval [0.0288, 0.116204] where the function changes sign. It then continues to search for the root by evaluating possible values in this interval, and eventually got to f = 0.0289678

Discussion

As opposed to Newton's method, fzero was able to find the root with initial guess 0.08, but was unable to do so with initial guess 0.008.

5.4.6 Fixed Point Iteration

Iteration function

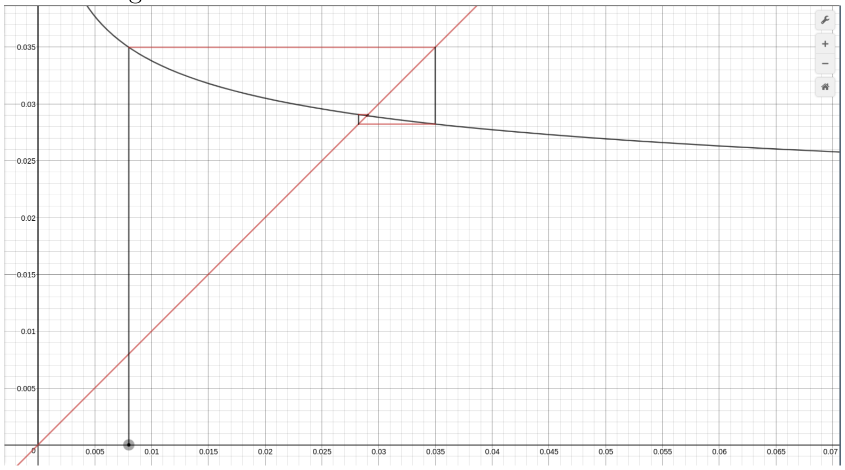
The fixed point method relies on whether we can find an iteration function $g(x)$ such that $f(x) = x - g(x)$ and $|g(xR)| < 1$ where xR is the root of the function. By repeatedly finding x for which $x = g(x)$, a reasonable estimate for the root is found. The code illustrating this method is attached in the appendix.

One of the possible iteration functions is:

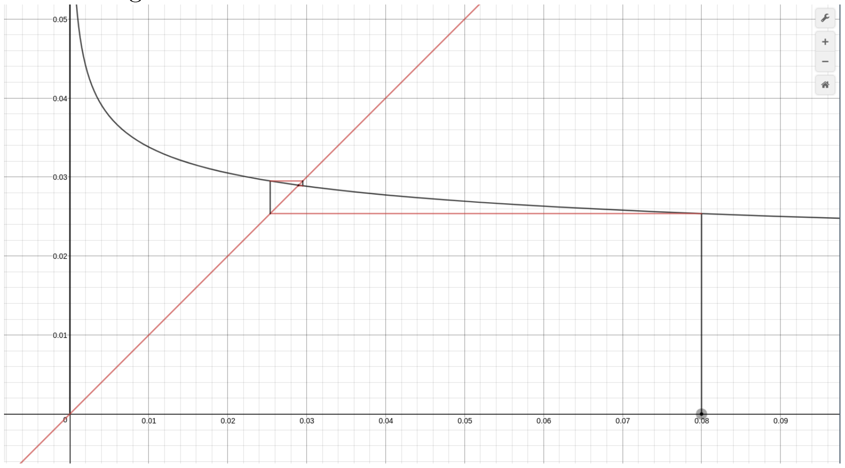
$$g(f) = \left(\frac{1}{2.0 \log_{10} \left(\frac{\varepsilon}{3.7D} + \frac{2.51}{Re\sqrt{f}} \right)} \right)^2$$

Cobweb Diagrams

For initial guess f = 0.008:



For initial guess f = 0.08:



5.4.7

Discussion With both initial guesses at 0.008 and 0.08, the method was able to find the approximation in 9 iterations. This is because $|g'(x)| < 1$ at the root and therefore guarantees convergence.

5.5 Consolidation of smaller discussions

We see that both the bisection method and the false position method were able to converge to the solution no matter what. This is unsurprising, since we know that the two methods are actually quite similar in nature. The bisection method converges faster, suggesting that there is some significant curvature to the equation that we are looking at.

Newton's method converges extremely fast if we choose the correct initial guess, however it falls apart equally quickly if we don't. Further discussion can revolve around what are the conditions for Newton's method to guarantee convergence, and how it depends on certain functions and their respective behaviors around their roots.

The method of fixed point iteration guarantees convergence due to the nature of the iteration function around the root. Further experimentation suggests that no matter what initial guess is used, the method will always find its way back to the root, further confirming our theorem stated in the previous chapter. Further discussion can involve finding a different iteration scheme to see if such scheme also guarantees convergence, and if it doesn't, why.

5.6 Appendix for this chapter

5.6.1 Code for the false position method

```
function [xm] = falsePositionM(f, x0, x1, n, TOL)
    xm(1) = x0;
    xm(2) = x1;
    fprintf('n \t approximation \t error \n');
    b = x0;
    a = x1;
    for i = 3:n
        xm(i) =  a - f(a) * (b - a) / (f(b) - f(a));
        error = Inf;
        if i > 3
            error = abs(xm(i) - xm(i-1));
        end
        relerror = error / abs(xm(i-1));
        fprintf('%d \t %12.8f \t %12.8f\n', i - 3, xm(i),
            relerror);
        if (error < TOL || f(xm(i)) == 0)
            break;
        end
        if (sign(f(xm(i))) == sign(f(a)))
            a = xm(i);
        else
            b = xm(i);
        end

    end
end
```

5.6.2 Code for Newton's method

```
function [xm] = NewtonsM(f, fder, x0, n, TOL)
    xm(1) = x0;
    fprintf('n \t approximation \t error \n');
    for i = 2:n
        xm(i) = xm(i-1) - f(xm(i-1)) / fder(xm(i-1));
        error = abs(xm(i) - xm(i-1));
        relerror = error / abs(xm(i-1));
        fprintf('%d \t %12.8f \t %12.8f \n', i - 1, xm(i)
            , relerror)
        if (error < TOL)
            break;
        end
    end
end
```

5.6.3 Code for fixed point method

```
function [xm] = fixedpointM(g, x0, n, TOL)
    xm(1) = x0;
    for i = 2:n
        xm(i) = g(xm(i-1));
        absE = abs(xm(i) - xm(i-1));
        relerror = absE / abs(xm(i-1));
        fprintf('%d \t %12.8f \t %12.8f \n', i - 1, xm(i)
            , relerror)
        if (absE < TOL)
            break;
        end
    end
end
```

5.6.4 Code for this chapter

```
clear all;
close all;
format long;

% Project 2
% Part a
    rho = 1.23;
    V = 40;
    D = 0.005;
    mu = 1.79e-5;
    % Gives Re = 1.374301675977654e+04
    Re = rho * V * D / mu;

% Part b
    eps = 0.0015e-3; % Given in millimeters
    f = @(x) 1 ./ (sqrt(x)) + 2 * log10(eps / (3.7 * D) +
        2.51 ./ (Re .* sqrt(x)));
    x = linspace(0.008, 0.08, 1000);
    plot(x, f(x));
    grid on;
    % From the estimation, it looks like the root is
       somewhere around 0.029

% Part c
    close all;
    bisectM(f, 0.008, 0.8, 50, 1e-8);
    falsePositionM(f, 0.008, 0.8, 50, 1e-8);

% Part d
    df =  @(x) (-1) ./ (2 *x * sqrt(x)) + 2 * (log10(exp
       (1)) * 1 / (eps / (3.7 * D) + 2.51 ./ (Re .* sqrt(
       x))) * (-2.51) ./ (2 * Re * x * sqrt(x)));
    NewtonsM(f, df, 0.008, 50, 1e-8);
    NewtonsM(f, df, 0.08, 50, 1e-8);

% Part e
    options = optimset('Display','iter', 'TolX', 1e-8);
    fzero(f, 0.008, options);
    fzero(f, 0.08, options);

% Part f
    % This function does not work
    g = @(x) (-1 ./ (2 * log10(eps / (3.7 * D) + 2.51 ./
       (Re .* sqrt(x))))).^2;
    fixedpointM(g, 0.008, 50, 1e-8);
    fprintf('\n');
    fixedpointM(g, 0.08, 50, 1e-8);
```

Chapter 6

Solving systems of linear equations

6.1 Introduction

So far we have explored several methods for solving for the root of a function. Now we make a little switch and look at systems of linear equation. Gaussian elimination is widely used to reduce matrices to forms that are easier to work with, thus enabling us to use backward substitution to solve for the solutions of the system. Another commonly used method is the LU factorization method, which is more efficient that traditional Gaussian elimination since it can be done independently of the constant matrix. Lastly, we explore another numerical method employed to solve for the solutions of these system, which is called the Jacobi method.

6.2 Gaussian elimination

6.2.1 Method explanation

This method is very familiar to anyone who has done some basic linear algebra. The idea is to concatenate our coefficient matrix with the constant matrix. Afterwards, we apply row reduction operations on the new matrix in order to reduce the complexity of the system, while keeping the resulting system equivalent to the system that we started out with. After we obtain a form that is easier to work with, we perform backwards substitution to get the solutions to our original system.

6.2.2 Method formalization

Step 1: Write the augmented matrix of the system of linear equations

Step 2: Use elementary row operations to rewrite the augmented matrix into row echelon form

Step 3: Write the system of linear equations corresponding to the matrix in row echelon form

Step 4: Use backwards substitution to find the solution of the new system of linear equation

6.2.3 Method discussion

The method is very straightforward, however the implementation of the method in Matlab can be quite involved. Readers are advised to look at the appendix of the next chapter, where the method is applied, to see how it is actually implemented. One pitfall of this method is that within practical usage, the coefficients and the constants are oftentimes the results of observations, and as such they can contain errors. Gaussian Elimination can magnify the amount of errors through the row reduction operations. Special care should be taken to make sure that the matrix is not ill-conditioned, because otherwise numerical results might be very different from exact results, since a small change in the coefficients/constants can change to a drastic change in the final solutions found.

6.3 LU factorization

6.3.1 Method explanation

LU factorization is another commonly used method which can solve for system of linear equations. The idea is to decompose the original coefficient matrix into its lower and upper matrices, after which we can employ backward and forward substitution to solve for the variables easily.

6.3.2 Method formalization

Given a linear system $\mathbf{Ax=b}$

Step 1: LU decomposition: Factor matrix $\mathbf{A}$ into lower $\mathbf{L}$ and upper $\mathbf{U}$ triangular matrices such that

$$\mathbf{Ax} = \mathbf{LUx} = \mathbf{b}$$

Step 2: **(i)** Set $\mathbf{y} = \mathbf{Ux}$ so that $\mathbf{Ax} = \mathbf{Ly} = \mathbf{b}$
(ii) Use forward substitution on $\mathbf{Ly} = \mathbf{b}$ to find $\mathbf{y}$

Step 3: Use back substitution on $\mathbf{Ux} = \mathbf{y}$ to find $\mathbf{x}$

6.3.3 Method discussion

This method is more efficient than the classic Gaussian elimination since it can be carried out regardless of the constant matrix. Therefore once we find a decomposition of the coefficient matrix, we can apply it to any constant matrix that we have without having to go through the process again, which we have to do in the case of the Gaussian elimination method.

6.4 Jacobi method

Jacobi method is the equivalence of the fixed point iteration method for systems of linear equations. The idea of the method is to rewrite each equation so that we have a calculation scheme for each of the variables. Then we employ an initial guess and plug that into the scheme to obtain new approximations for the solutions. We keep on doing this until we find an answer that falls within tolerance.

6.4.1 Method formalization

Step 1: Rewrite each equation to find an iteration scheme

Step 2: Guess initial valuse

Step 3: Iterate the Jacobi scheme

Step 4: Check for convergence, repeat from step 3 if error is not within tolerance

6.4.2 Method discussion

The method of Jacobi has its advantages. Compared to Gaussian Elimination, it's much easier to code in Matlab. As such, to solve linear systems involving several hundreds of variables, particularly if many of the coefficients are zeroes, then iterative methods such as this are much more preferable over Gaussian Elimination.

One caveat is that for this method to converge, the original matrix must be strictly diagonally dominant, meaning that the absolute value of the current diagonal coefficient must be at least the sum of the absolute values of all the other elements on the same row. This might require some extra work and care even before we begin to do the operations.

Chapter 7

Indoor Air Pollution

7.1 Abstract

This project deals with indoor air pollution in enclosed spaces such as homes, offices and work areas. We will be leveraging mathematical modeling to describe a real problems using abstract mathematical concepts. By solving for unknowns using these mathematical concepts, we relate the results back to the problem of air pollution and what we can do to improve air quality in these enclosed spaces.

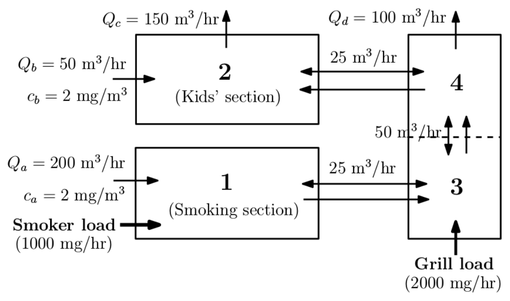
7.2 Introduction

7.2.1 Background information

In this project, we explore how to model certain real-life scenarios using systems of linear equations. The power of these linear systems will enable us to have valuable insights which we can use to make a positive impact on these scenarios.

7.2.2 Problem description

The problem is given using the figure depicted below. There is a restaurant whose serving area consists of two rooms for smokers and kids, together with one elongated room. Room 1 and 3 have sources of CO_2 coming from smokers and a faulty grill. Rooms 1 and 2 also gain CO_2 from air intakes that are positioned alongside the freeway.



7.2.3 Outline

We can model the influx and outflux of CO_2 in each room by using systems of linear equations. Since we know that the rooms are in steady states, the influx of CO_2 must equal the outflux of CO_2 for each of the rooms.

7.3 Numerical methods

In this section, we will use Gaussian elimination, LU factorization as well as the Jacobian method to solve systems of linear equations.

Let $c1, c2, c3, c4$ be the concentrations of CO_2 in rooms 1, 2, 3 and 4, respectively. Then from the diagram, we construct the steady-state equations, assuming that outflux must equal influx in the long run:

$$225 \times c1 + 25 \times c3 = -1400$$

$$175 \times c2 - 125 \times c4 = 100$$

$$225 \times c1 - 275 \times c3 + 50 \times c4 = -2000$$

$$25 \times c2 + 250 \times c3 - 275 \times c4 = 0$$

7.4 Results

7.4.1 Using Gaussian elimination

Gaussian elimination results in the following: $c1 = 8.0996 mg/m^3$, $c2 = 12.3448 mg/m^3$, $c3 = 16.8966 mg/m^3$, and $c4 = 16.4828 mg/m^3$

7.4.2 Using LU Factorization

The built-in LU factorization method in Matlab results in the exact same results:
$c1 = 8.0996 mg/m^3$, $c2 = 12.3448 mg/m^3$, $c3 = 16.8966 mg/m^3$, and $c4 = 16.4828 mg/m^3$

7.4.3 Using Jacobi method

The Jacobi method requires that the matrix must be strictly diagonally dominant in order for the method to converge. Luckily, our matrix satisfies this requirement. Using the initial guesses where all concentrations are $1mg/m^3$, the method converges after 24 iterations, with tolerance 10^{-6}. The results are $c1 = 8.0996mg/m^3$, $c2 = 12.3448mg/m^3$, $c3 = 16.8965mg/m^3$, and $c4 = 16.4827mg/m^3$

7.4.4 Discussion

Quite surprisingly, room 1, the designated smoker's room, actually has the lowest concentration of CO_2. However upon closer inspection, we see that room 1 is very well ventilated, with the outflux measured in $225 \times c1$. Room 3 has the highest concentration of CO_2, because it receives the large outflux of room 1 as parts of its influx. Even though room 3 is quite well ventilated (outflux is measured at $275 \times c3$, the heavy influx coming from the grill and from room 1 hold the concentration of CO_2 in this room at a high level.

The inverse of the coefficient matrix A is given by Matlab's inv function:

$$A^{-1} = \begin{bmatrix} 0.004996168582375 & 0.000015325670498 & -0.000551724137931 & -0.000107279693487 \\ 0.003448275862069 & 0.006206896551724 & -0.003448275862069 & -0.003448275862069 \\ 0.004965517241379 & 0.000137931034483 & -0.004965517241379 & -0.000965517241379 \\ 0.004827586206897 & 0.000689655172414 & -0.004827586206897 & -0.004827586206897 \end{bmatrix}$$

Let the constant matrix be b, where:

$$b = \begin{bmatrix} 1400 \\ 100 \\ -2000 \\ 0 \end{bmatrix}$$

then we know that the concentration of CO_2 in the second room is precisely the the dot product of the second row of A^{-1} and b^T.
As such, the smokers contribute around

$$\frac{1000}{1400} \times \frac{1400 \times 0.003448275862069}{12.3448} \times 100\% \approx 27.9\%$$

As for the grills:

$$\frac{-2000 \times -0.003448275862069}{12.3448} \times 100\% \approx 55.9\%$$

The rest, 16.2%, comes from the intake vents.
Suppose now smoking is banned, and the grill is fixed. We see that CO_2 contribution in the constant matrix would drop to 400 for the first entry, and 0 for the third. Using Matlab to solve with a new constant matrix, we obtain that the concentration is $2mg/m^3$ for every room.

Now suppose that the mixing between areas 2 and 4 is decreased to $5m^3/hr$. Then we have a new system:

$$225\times c1 + 25\times c3 = -1400$$

$$155\times c2 - 105\times c4 = 100$$

$$225\times c1 - 275\times c3 + 50\times c4 = -2000$$

$$5\times c2 + 250\times c3 - 255\times c4 = 0$$

The concentration in room 2 drops to 12.0800, which makes sense since room 2 is getting its CO_2 intake from the air circulating in the other 3 rooms. Decreasing the amount of air flowing between 2 and 4 would help reduce the concentration in room 2. However the drop is not too significant since from room 4, there is still a significant portion of volumetric airflow flowing into room 2 at $100m^3/h$.

7.5 Appendix for this chapter

7.5.1 Code for the Gaussian elimination method

```
function [x] = GE(A, b, ptol)

% Gauss elimination and back sub

% No pivoting is used

% INPUT
% A =  coefficient matrix; b = constant matrix;
% ptol = [optional], tolerance for detection of zero
    pviot
% set [Default ] ptol = 50 * eps

% OUTPUT
% x = solution vector, if solution exists

if nargin<3, ptol = 50*eps; end
[m,n] = size(A);
if m ~= n, error('Matrix A must be square'); end
Ab = [A b]; % Augmented matrix
fprintf('\n Begin GE with augmented matrix:\n');
disp(Ab);

% Elimination
for i = 1:(n-1)
    pivot = Ab(i,i);
    if abs(pivot) < ptol
        fprintf("Partial pivoting");
        for k = i+1:n
            if abs(Ab(k,i)) > ptol
               swapRow(Ab, i, k);
            end
        end
    end

    for k = (i+1):n % j = index of next row to be
        eliminated
        M = Ab(k,i)/pivot;
        Ab(k, i:(n+1)) = Ab(k, i:(n+1)) - M * Ab(i, i:(n
            +1));
    end
end

disp(Ab);

% Back substitution
x = zeros(n,1); % initialize x
x(n) = Ab(n, n+1)/Ab(n,n);

for j=(n-1):-1:1
        x(j) = (Ab(j, n+1) - Ab(j, (j+1:n)) * x(j+1:n))./
            Ab(j, j);
end
```

7.5.2 Code for the Jacobi method

```
function [x] = Jacobi(A, b, initial, n, TOL)

    if nargin<3, TOL = 50*eps; end
    [row, col] = size(A);
    if row ~= col, error('Invalid matrix'); end
    Ab = [A b]; % Augmented matrix
    fprintf('\n Augmented matrix:\n');
    disp(Ab);
    x = zeros(1, col);
    prev = zeros(1, col);
    for i = 1:col
        prev(i) = initial(i);
    end

    for i = 1:n
        for j = 1:row
```

```
                coefficient = Ab(j,j);
                current = Ab(j, col + 1);
                for k = 1:col
                    if (k == j)
                        continue;
                    end
                    current = current - prev(k) * Ab(j, k);
                end
                new_value = current / coefficient;
                x(j) = new_value;
            end
            if norm(x - prev, inf) < TOL
                break;
            end
            for j = 1:row
                prev(j) = x(j);
            end
            fprintf(i + " ");
            disp(x);
        end
```

7.5.3 Code for this chapter

```
format long;

A = [225, 0, -25, 0; 0, 175, 0, -125; 225, 0, -275, 50;
    0, 25, 250, -275]
b = [1400; 100; -2000; 0];

TOL = 10e-6;
% Using Gaussian Elimination
GE(A, b, TOL)

% Using LU Factorization
[L, U] = lu(A);
y =  L\b;
x = U\y

% Using the Jacobi method
initial = [1; 1; 1; 1]
Jacobi(A, b, initial, 30, TOL);

% Finding inverses of the matrix
disp(inv(A));

% Question e
```

```
b1 = [400; 100; 0; 0];
[L, U] = lu(A);
y =  L\b1;
x = U\y


A1 = [225, 0, -25, 0; 0, 155, 0, -105; 225, 0, -275, 50;
    0, 5, 250, -255]
GE(A1, b, TOL)
```

Chapter 8

Curve fitting - Cubic spline interpolation

8.1 Introduction

The final part of this book deals with curve fitting given different data points. We switch gear from finding solutions to given equations/systems. and now we do something that is of the opposite nature. Given the data points, we construct a curve that interpolates the data points, giving us a picture of what might have been the equation that gave rise to those data points. The specific technique that we are talking about is cubic spline interpolation.

8.2 Method explanation

By employing cubic curves to fit the intervals of the data points given, we construct a smooth curve that goes through all the data points that we have, in the hope that this curve will give us good estimations of the other points inside the interval created by the given points. Given n data points, we construct n-1 cubic functions that, put together, give us a smooth curve that can be used for further estimations.

8.3 Method formalization

Assume that we are given the N data points $(x_1, Y_1), (x_2, y_2)...(x_N, y_N)$, where the $x_i's$ are distinct and in increasing order. Then a **cubic spline** $S(x)$ through the data points is a set of cubic polynomials $S_1, S_2...S_{N-1}$ satisfying the following properties

Property 1: $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$ for $i = 1, 2, ..., N-1$

Property 2: $S'_{i-1}(x_i) = S'_i(x_i)$ for $i = 2, 3, ..., N-1$

Property 3: $S''_{i-1}(x_i) = S''_i(x_i)$ for $i = 2, 3, ..., N-1$

8.4 Method discussion

The above properties gives us $3N-5$ equations, however it is easy to see that we have $N-1$ cubic functions, each containing 3 unknowns. Therefore we would need to have $3N-3$ equations to solve for all the coefficients. The 2 extra equations usually come from endpoint conditions, There are many endpoint conditions, however the next project deals with natural endpoint conditions, meaning that the second derivatives of the first and last functions at the two outermost points evaluate to 0 for both of them.

Chapter 9

Thermocline at Platte Lake

9.1 Abstract

This project investigates the thermocline, which is the horizontal plane separating regions of different temperatures in lakes. By using data, which give the depth of water and the corresponding temperature at that depth, collected at Platte Lake, we use the method of cubic spline interpolation to find a function that gives the water temperature at any given depth. Using this function and its first and second derivatives, we are able to find the position at which the thermocline occurs, and from there we can also calculate the heat flux across this thermocline.

9.2 Introduction

9.2.1 Background information

In this section, we explore the method of cubic spline interpolation, which we will use to interpolate data points collected by experimentation. The data given, reflecting the various water depths and the corresponding water temperature, are collected from Platte Lake.

9.2.2 Problem description

The first task in this problem is to find a function that models water temperature with respect to changing water depths. After that, we need to find the position of the thermocline. The thermocline is defined to be the point of inflection for the temperature depth graph. In other words, this is the point where $\frac{d^2T}{dx^2} = 0$, where T is the temperature, measured in Celcius degrees, and x is the depth, measured in meters. Also, the thermocline is the position at which the absolute value of the first derivative of the graph is at maximum.
After finding the thermocline, we determine the heat flux across the thermocline

by using this equation:

$$J = -\alpha \times \rho \times C \times \frac{dT}{dx}$$

where J is the heat flux in $cal/(cm^2 s$, α is the eddy diffusion coefficient cm^2/s, ρ is the density of water($\approx 1g/cm^3$), and C is the specific heat of water $\approx 1cal/gC$.

9.2.3 Outline

Here the method chosen to find such a function is the method of cubic spline interpolation. Since this method introduces $3N-3$ unknowns, but we can only construct $3N-5$ linear equations using the properties discussed in class, we have to supply 2 more equations in order to find all the unknowns. The problem wants to explore natural cubic spline interpolation, meaning that if $f(x)$ is our function, then $f''(x) = 0$ at both of the outermost data points.

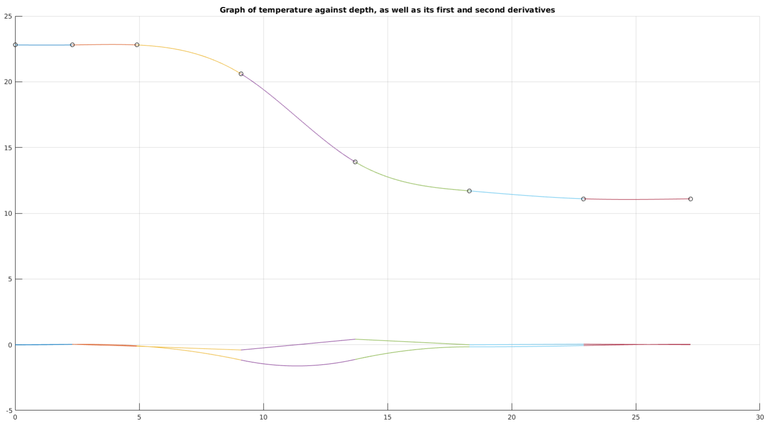
9.3 Numerical methods

In this project, we use the method of cubic spline interpolation, with natural endpoint conditions. Given the data points $(x1, y1), (x2, y2), ...(xN, yN)$, the method constructs a function that interpolates all the data points. The code to implement this is attached in the appendix.

9.4 Results

9.4.1 Graph of temperature against depth

This is the graph produced by using Matlab to construct the cubic spline that interpolates all the data points given:



9.4.2 Discussion

From the graph given, we see that the thermocline occurs in the depth between the 4th and the 5th data points, or in the purple cubic function. We proceed to find the function modeling the temperature against depth between those 2 points using the coefficients calculated from before. The, we take the second derivative of that function with respect to the depth. After that, we use the bisection method to find the root of the second derivative in this interval, to give that the thermocline is ≈ 11.3463673830. Plugging this into the equation for the heat flux, we find that the heat flux around the thermocline is $J \approx 1.614055517555187e-04$

9.5 Appendix for this chapter

9.5.1 Code for this chapter

```
clear clc;
close all;

depth = [0, 2.3, 4.9, 9.1, 13.7, 18.3, 22.9, 27.2];
temp = [22.8, 22.8, 22.8, 20.6, 13.9, 11.7, 11.1, 11.1];

an = spline_natural(depth, temp);
size(an)
hold on;
grid on;
for i=1:7
    xspace = linspace(depth(i), depth(i+1), 100);
    curx = depth(i);
    f = @(x) temp(i) + an(3 * (i - 1) + 1) .* (x - curx)
        ...
    + an(3 * (i - 1) + 2) .* (x - curx).^2 ...
    + an(3 * (i - 1) + 3) .* (x - curx).^3;
    plot(xspace, f(xspace));
end
for i=1:7
    xspace = linspace(depth(i), depth(i+1), 100);
    curx = depth(i);
    f = @(x) an(3 * (i - 1) + 1) ...
    + 2 * an(3 * (i - 1) + 2) .* (x - curx) ...
    + 3 * an(3 * (i - 1) + 3) .* (x - curx).^2;
    plot(xspace, f(xspace));
end
for i=1:7
    xspace = linspace(depth(i), depth(i+1), 100);
    curx = depth(i);
    f = @(x)2 * an(3 * (i - 1) + 2) ...
    + 6 * an(3 * (i - 1) + 3) .* (x - curx);
    plot(xspace, f(xspace));
end
title('Graph of temperature against depth, as well as its
     first and second derivatives');
plot(depth, temp, 'ko');

% The graph shows that the thermocline should lie between
     the 4th and the
% 5th data point.
f = @(x) 2 * an(3 * 3 + 2) ...
    + 6 * an(3 * 3 + 3) .* (x - depth(4));
bisectM(f, 9.1, 13.7, 30, 1e-6);

thermocline = 11.3463673830;

alpha = 0.01;
rho = 1;
C = 1;

% Function for the gradient
f = @(x) an(3 * 3 + 1) ...
    + 2 * an(3 * 3 + 2) .* (x - depth(4)) ...
    + 3 * an(3 * 3 + 3) .* (x - depth(4)).^2;

%  -1.6141 Celcius/m
gradient = f(thermocline);

% 1.614055517555187e-04
flux = - alpha * rho * C * gradient / 100
```

9.5.2 Natural Cubic Spline calculation in Matlab

```
function [result] = spline_natural(x, y)
    [k, n] = size(x);
    [l, m] = size(y);
    if n ~= m || k ~= l
        error('Wrong arguments given. Exiting');
    end
    % Matrix to hold results. bi is at entry 3 * (i-1) +
       1, ci is at 3 *
    % (i-1) + 2, di is at 3 * (i-1) + 3
    result = zeros(3 * (n - 1), 1);

    % Matrix to hold all equations
    eq = zeros(3 * (n-1), 3 * (n-1));

    % Column vector to hold the constants of the
        equations
    const = zeros(3 * (n - 1), 1);
    % constructing equations according to the first
        property
    % Use a currentInterval variable to keep track of
        what intervals we are
    % dealing with

    % Use currentEquation to keep track of which equation
         we are
    % constructing

    currentEquation = 1;
    % Applying endpoint conditions (natural)
    eq(currentEquation, 2) = 2;
    currentEquation = currentEquation + 1;
    for currentInterval=1:n-1
        % Construct equation according to the first
            property
        xdiff = x(currentInterval + 1) - x(
            currentInterval);
        ydiff = y(currentInterval + 1) - y(
            currentInterval);
        % bi
        eq(currentEquation, 3 * (currentInterval - 1) +
            1) = xdiff;
        % ci
        eq(currentEquation, 3 * (currentInterval - 1) +
            2) = xdiff.^2;
        % di
        eq(currentEquation, 3 * (currentInterval - 1) +
            3) = xdiff.^3;
        % constant
        const(currentEquation) = ydiff;
        currentEquation = currentEquation + 1;

        if currentInterval ~= n - 1
            % Construct equation according to the second
                property
            eq(currentEquation, 3 * (currentInterval - 1)
                + 1) = 1;
            eq(currentEquation, 3 * (currentInterval - 1)
                + 2) = 2 * xdiff;
            eq(currentEquation, 3 * (currentInterval - 1) + 3) = 3 * xdiff.^2;
            eq(currentEquation, 3 * (currentInterval - 1) + 4) = -1;
            currentEquation = currentEquation + 1;
            % Construct the equation according to the third property
            eq(currentEquation, 3 * (currentInterval - 1) + 2) = 2;
            eq(currentEquation, 3 * (currentInterval - 1) + 3) = 6 * xdiff;
            eq(currentEquation, 3 * (currentInterval - 1) + 5) = -2;
            currentEquation = currentEquation + 1;
        end

        if currentInterval == n - 1
            % Apply last endpoint condition (natural)
            eq(currentEquation, 3 * (currentInterval - 1) + 2) = 2;
            eq(currentEquation, 3 * (currentInterval - 1) + 3) = 6 * xdiff;
            currentEquation = currentEquation + 1;
        end
    end
    cond(eq)
    result = eq\const;
end
```