

2018

SEMESTER 2

ENGSCI 331 Eigen Problems Assignment

Connor McDowall
530913386
cmcd398

October 23, 2018

Contents

1	Report	2
2	Appendices	3
2.1	myEigenFunctions.h	3
2.2	myEigenFunctions.cpp	5
2.3	myEigenMain.cpp	14
2.4	Screenshots from Testing and Output	20

Listings

myEigenFunctions.h	3
myEigenFunctions.cpp	5
myEigenMain.cpp	14

List of Figures

1	Eigen Shift Method Natural Frequencies and Eigen Vectors	2
2	Eigen All Method Natural Frequencies and Eigen Vectors	2
3	Comparisons to describe the specific patterns of motion	2
4	Numerical Solution compared against the Analytical	3
5	Power Method and Deflation Test	20
6	Eigen Shift Test	20
7	Eigen All Test	21
8	Eigen Shift Output	21
9	Eigen All Output	22

1 Report

Size	Eigen Shift Method		Eigen Shift Normalised Eigen Vectors									
	Eigen Values (Numerical)	Natural Frequencies (Numerical)	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
Smallest	-2489.2	7.941	0.035	0.104	0.171	0.233	0.290	0.340	0.381	0.413	0.435	0.446
Largest	-401867.0	100.893	-0.446	0.435	-0.413	0.381	-0.340	0.290	-0.233	0.171	-0.104	0.035

Figure 1: Eigen Shift Method Natural Frequencies and Eigen Vectors

N	Eigen All Method			Eigen All Normalised Eigen Vectors									
	Natural Frequency (Analytical)	Eigen Values (Numerical)	Natural Frequency (Numerical)	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
19	151.0	-401867	100.893	-0.446	0.435	-0.413	0.381	-0.340	0.290	-0.233	0.171	-0.104	0.035
17	135.1	-382320	98.409	0.434	-0.340	0.170	0.036	-0.234	0.382	-0.446	0.413	-0.290	0.104
15	119.2	-345140	93.501	0.413	-0.171	-0.171	0.413	-0.413	0.171	0.172	-0.413	0.413	-0.171
13	103.3	-293965	86.291	-0.381	-0.035	0.413	-0.340	-0.105	0.435	-0.290	-0.171	0.446	-0.234
11	87.4	-233806	76.957	-0.340	-0.234	0.413	0.105	-0.446	0.035	0.435	-0.171	-0.381	0.290
9	71.5	-170551	65.728	-0.290	-0.381	0.171	0.435	-0.035	-0.446	-0.104	0.413	0.234	-0.340
7	55.6	-110391	52.879	0.234	0.446	0.171	-0.291	-0.435	-0.104	0.340	0.413	0.035	-0.381
5	39.7	-59216.6	38.729	0.171	0.413	0.413	0.171	-0.171	-0.413	-0.413	-0.171	0.171	0.413
3	23.8	-22036.1	23.626	-0.104	-0.290	-0.413	-0.446	-0.381	-0.234	-0.035	0.171	0.340	0.435
1	7.9	-2489.15	7.940	-0.035	-0.104	-0.171	-0.234	-0.290	-0.340	-0.381	-0.413	-0.435	-0.446

Figure 2: Eigen All Method Natural Frequencies and Eigen Vectors

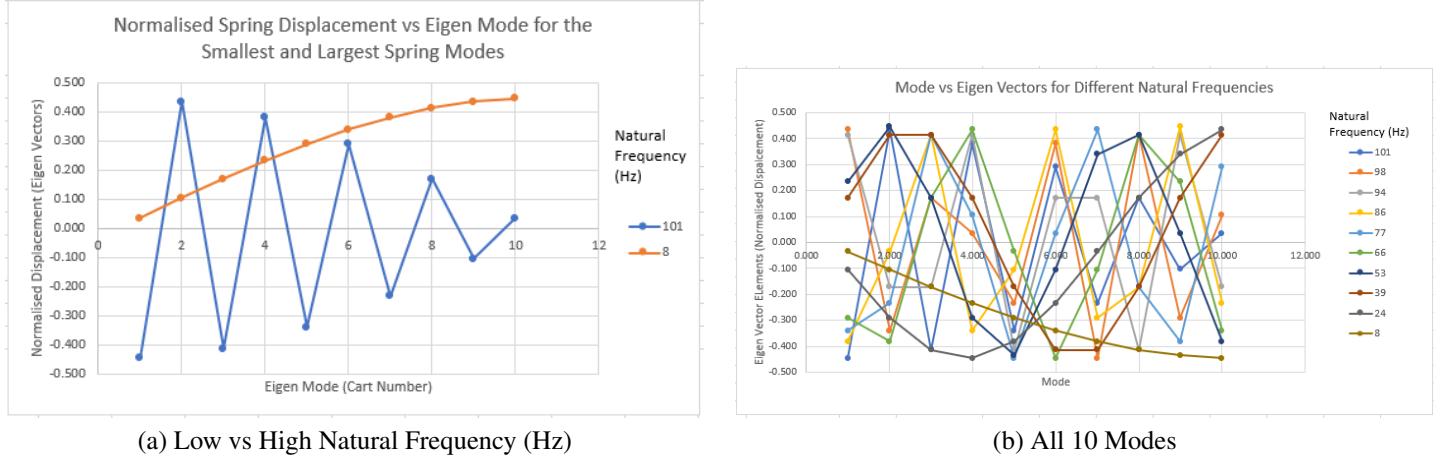


Figure 3: Comparisons to describe the specific patterns of motion

The carts model an eigenmode, a natural vibration of a system where all parts of the system move at the same frequency, moving sinusoidally with amplitudes changing proportionally to each other. Using the analytical parameters, we observe this pattern in figures 3 (a) and (b).

The largest eigen value (highest natural frequency) has an oscillating pattern across the 10 coil spring system. The associated eigen vector has both positive and negative elements in the vector. The varying signs of the eigen vector's elements show the carts are moving in different directions with the coils extending or compressing depending on the sign. This oscillating movement and pattern of the system is expected at high frequencies. Surging does not occur in the high spring system. Resonant behaviour is not observed with the surging frequencies much higher than the engine vibration frequencies.

The smallest eigen value has spring modes all moving in the same direction, as shown by the upward trend on figure 3 (a). All carts are moving in the same direction with coils all extending. The frequency is not large enough to cause the spring coils and carts to move in different directions. We observe resonant behaviour. We expect this from the low frequency and the given parameter combination. The extent the system oscillates decreases for each mode as the natural frequency decreases until all carts move in the same direction. This observed in figure 3 (b), leading to surging behaviour.

2 Appendices

All functions were tested using the test functions in the myEigenFunctions file. The output was printed and compared to the analytical solutions in the notes. The eigen vectors in the notes are not normalised but the outputs from my functions are. I converted the eigen vectors in the notes to a normalised form by hand. See 2.4 for screenshots of the test and implementation output.

After the assignment reduction, I assumed we no longer had to use a function to construct the A matrix as this was not specified in the new handout. The A matrix is constructed manually using the parameters specified in the original handout. I also assumed we didn't need to have a matrix constructed based on user inputs.

The signs of the eigen vector's elements inform the direction you are looking at the eigen vector. The smallest eigen value's vector is the same for both the Eigen All and Eigen Shift methods. This is the case as the signs for eigen all's vector elements are the exact inverse for eigen shift's vector elements. Both methods find the same eigen vector but look at it from different directions.

There is a difference in the analytical and numerical methods. This is attributable to numerical inaccuracies in working with large numbers at high frequencies. Small differences in large numbers make big impacts, as seen with higher frequencies becoming increasingly inaccurate. See figure 4 for the visualisation.

Figures 3 (a) and (b) are plotted using straight lines as the normalised eigen vectors and natural frequencies are discrete values.

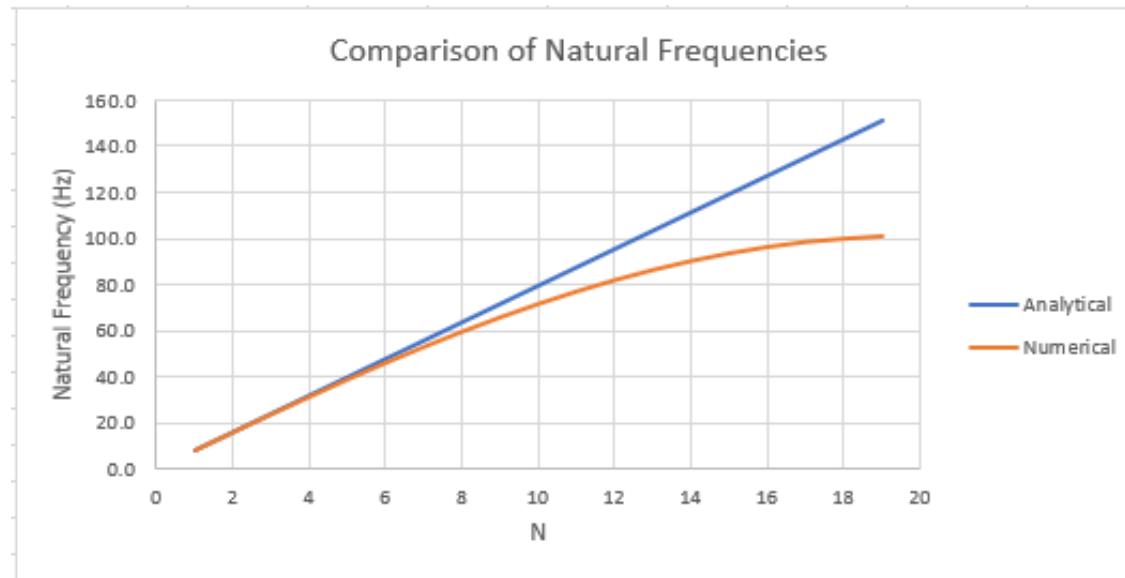


Figure 4: Numerical Solution compared against the Analytical

2.1 myEigenFunctions.h

```
/* ****myEigenFunctions.h****  
myEigenFunctions.h
```

This file is where you'll put the header information about the functions you write. This is just

the argument list (as it is in the first line of your function declarations in `myEigenFunctions.cpp`) followed by a semi-colon. Two example functions have been done for you.

```
=====
#include <iostream>
#include <math.h>

// Note that these two functions have the same name, but different argument lists. This is called
// "function overloading" and is allowed by C++ compilers.

double DotProduct(double *A, double *B, int n);

double* DotProduct(double **A, double *v, int n);

double DotProduct(double **A, double **B, int n, int m);

void powermethodanddeflatetest();

struct Eigenpair {
    double value; //Eigenvalue
    double *vector; //Eigenvector
    int length; //Length of eigenvector
    void normalize() {
        // Set eigenvalue to norm of vector and normalize eigenvector
        value = sqrt(DotProduct(vector, vector, length));
        for (int i=0; i<length; i++)
            vector[i]/= value;
    }; //

    void print() {
        std::cout << value << "\n";
        for (int i=0; i<length; i++)
            std::cout << vector[i] << "\n";
        std::cout << "\n";
    }
    // Constructor
    // Attribute value is set to 0.0 and attribute vector to an array of doubles with lenght n
    Eigenpair(const int n) : value(0.0), length(n), vector(new double[n]) {} //Constructor
};
```

```

Eigenpair power_method(double **A, double *v, int n, double tol);

void deflate(double **A, Eigenpair eigenpair);

void print_matrix(double **A, int n, int m);

void print_vector(double *v, int n);

Eigenpair eigenshift(double **A, double *v, int n, double tol);

void eigenall(double **A, double *v, int n, int tol);

void eigenalltest();

void eigenshifttest();

```

2.2 myEigenFunctions.cpp

```
*****
```

This file is where you'll put the source information for the functions you write. Make sure it's included in your project (shift-alt-A) and that the functions you add are declared in the header file, myEigenFunctions.h. You can add any extra comments or checks that could be relevant to these functions as you need to.

```
=====
#include "myEigenFunctions.h"

double DotProduct(double *A, double *B, int n)
{
    //
    //      This is a function that takes two vectors A and B of identical length (n) and
    //      calculates and returns their dot product.
    //

    double dot = 0.0;

    for (int i = 0; i < n; i++) {
        dot += A[i] * B[i];
    }
    return dot;
}
double DotProduct(double **A, double **B, int n, int m)
```

```

{
    //
    // This is a function that takes two matrices A and B of identical dimensions (n*m) and
    // returns their dot product.
    //

    double dot = 0.0;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            dot += A[i][j] * B[i][j];
        }
    }
    return dot;
}

double* DotProduct(double **A, double *v, int n)
{
    //
    // This is a function that takes a nxn A matrix and n dimensional B vector and
    // stores the product A.V at the original location of v
    //
    double *result = new double[n]; //point to the result vector

    for (int i = 0; i < n; i++)
    {
        result[i] = 0.0; // Initialize ith element of result v
        for (int j = 0; j < n; j++)
        {
            result[i] += A[i][j] * v[j];
        }
    }
    return result;
}
// Write the power_method
Eigenpair power_method(double **A, double *v, int n, double tol)
{
    // This function computes the largest eigenvalue and the corresponding normalised eigen vector
    // Do all the initial set up
    Eigenpair eigenpair(n);
    double *vector_hat = new double[n];
    double value_hat;
}
```

```

int istore;

// Setting the initial eigenpair values as those from the inputs
for (int i = 0; i < n; i++)
{
    eigenpair.vector[i] = v[i];
}
// Normalise the original eigen value estimate
eigenpair.normalize();

do {
    //Set the initial eigen value for convergence
    value_hat = eigenpair.value;
    eigenpair.vector = DotProduct(A, eigenpair.vector, eigenpair.length);

    // Find the index of the largest element
    double vstore = 0;
    for (int i = 0; i < n; i++)
    {
        if (abs(eigenpair.vector[i]) > vstore)
        {
            istore = i;
            vstore = (abs(eigenpair.vector[i]));
        }
    }

    // Set eigenvalue to the norm of the vector and normalise the vector
    eigenpair.normalize();

    // Condition to break the loop
} while (abs(eigenpair.value - value_hat) / abs(eigenpair.value) > tol);

// If condition to assess if eigen value is going in the opposite direction.
if (DotProduct(A, eigenpair.vector, n)[istore] / eigenpair.vector[istore] < 0)
{
    eigenpair.value = -1 * eigenpair.value;
}
// Convert the eigen vector back to an unnormalised form
// eigenpair.vector = eigenpair.vector*
return eigenpair;
}

// deflate method

```

```

void deflate(double **A, Eigenpair eigenpair)
{
    // This function removes the largest eigenvalue from a matrix so the power method
    // can find the next largest value.
    // Input A matrix, normalised eigenvector and largest eigen value
    //
    {
        // Eigen vector already normalised
        //Apply the deflation method in a for loop
        //Create a new matrix
        double **C;
        C = new double*[eigenpair.length];
        for (int i = 0; i < eigenpair.length; i++)
        {
            C[i] = new double[eigenpair.length];
        }

        // Calculate the C matrix values as you go
        //Create a double matrix
        for (int i = 0; i < eigenpair.length; i++)
        {
            for (int j = 0; j < eigenpair.length; j++)
            {
                C[i][j] = eigenpair.value*eigenpair.vector[i] * eigenpair.vector[j];
            }
        }
        // Perform the calculation
        for (int i = 0; i < eigenpair.length; i++)
        {
            for (int j = 0; j < eigenpair.length; j++)
            {
                A[i][j] = A[i][j] - C[i][j];
            }
        }
    }

    // eigen-shift (Insert the function here).
    Eigenpair eigenshift(double **A, double *v, int n, double tol)
    {
        // Use the power method to find the largest eigenvalue
        Eigenpair a = power_method(A, v, n, tol);
        // Store the largest eigen value
        double eigenlarge = a.value;
        // Create the identity matrix
    }
}

```

```

double **I;
I = new double*[a.length];
for (int i = 0; i < a.length; i++)
{
    I[i] = new double[a.length];
}

// Assign a value of zero to the entire matrix
for (int i = 0; i < a.length; i++)
{
    for (int j = 0; j < a.length; j++)
    {
        I[i][j] = 0;
    }
}
// Assign values of I the trace elements
for (int i = 0; i < a.length; i++)
{
    I[i][i] = eigenlarge;
}

// Use a loop to Perform the shift
for (int i = 0; i < a.length; i++)
{
    for (int j = 0; j < a.length; j++)
    {
        A[i][j] = A[i][j] - I[i][j];
    }
}
// Use the power method again to shift the matrix
Eigenpair b = power_method(A, v, n, tol);

// Shift the eigen value to return the smallest
double eigensmall = b.value + eigenlarge;

// Print Eigen Small outputs
std::cout << "Eigensmall";
std::cout << "\n";
std::cout << eigensmall;
std::cout << "\n";
std::cout << "Eigenlarge";
std::cout << "\n";
std::cout << eigenlarge;

```

```

    std :: cout << "\n";
    // Print Eigen Large outputs
    std :: cout << "Eigen small - Vector";
    std :: cout << "\n";
    print_vector(b . vector , b . length );
    std :: cout << "\n";
    std :: cout << "Eigen large - Vector";
    std :: cout << "\n";
    print_vector(a . vector , a . length );
    std :: cout << "\n";

    // This assumes eigen values are all the correct sign
    return a , b ;
}

// eigen_all ( Assumes all functions given will be symmetric )
void eigenall(double **A, double *v, int n, int tol)
{
    // Use the power method and deflation in an iterative scheme
    for (int i = 0; i < n; i++)
    {
        // Use the iterative scheme
        // Test the function
        Eigenpair test = power_method(A, v, n, 1e-8);
        std :: cout << "Eigenvalue";
        std :: cout << "\n";
        std :: cout << test . value;
        std :: cout << "\n";
        std :: cout << "Eigen - Vector - After - Power - Method";
        std :: cout << "\n";
        print_vector(test . vector , test . length );
        std :: cout << "\n";

        // Print the matrix before deflating
        std :: cout << "Matrix - Before - Deflating";
        std :: cout << "\n";
        print_matrix(A, test . length , test . length );
        std :: cout << "\n";

        // Test the deflate method
        deflate(A, test);

        // Print the output matrix after deflating
    }
}

```

```
    std::cout << "Matrix_After_Deflating";
    std::cout << "\n";
    print_matrix(A, test.length, test.length);
    std::cout << "\n";
}
}

// Create a power method testong function
void powermethodanddeflatetest()
{
    // Create a matrix
    int n = 3;
    double**A;
    A = new double*[n];
    for (int i = 0; i < n; i++)
    {
        A[i] = new double[n];
    }
    //Set up the matrix
    A[0][0] = 1;
    A[0][1] = 2;
    A[0][2] = 0;
    A[1][0] = 2;
    A[1][1] = 1;
    A[1][2] = 0;
    A[2][0] = 0;
    A[2][1] = 0;
    A[2][2] = 2;

    // Set up vector
    double *v;
    v = new double[n];
    v[0] = 2;
    v[1] = 1;
    v[2] = 3;

    //Test the function
    Eigenpair test = power_method(A, v, n, 1e-8);
    std::cout << test.value;
    std::cout << "\n";
    print_vector(test.vector, test.length);
    std::cout << "\n";
```

```

// Test the deflate method
deflate(A, test);

// Print the output matrix
print_matrix(A, test.length, test.length);
}

void print_matrix(double **A, int n, int m)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            std::cout << A[i][j];
            std::cout << "\u00b7";
        }
        std::cout << "\n";
    }
}

void print_vector(double *v, int n)
{
    for (int i = 0; i < n; i++)
    {
        std::cout << v[i];
        std::cout << "\u00b7";
    }
}

void eigenalltest()
{
    // Test used the numerical soultion in the notes and printed all the outputs.
    // These were prepared and came out correct in the system.
    // Create a matrix
    int n = 3;
    double**A;
    A = new double*[n];
    for (int i = 0; i < n; i++)
    {
        A[i] = new double[n];
    }
    // Set up the matrix
    A[0][0] = 1;
    A[0][1] = 2;
    A[0][2] = 0;
    A[1][0] = 2;
    A[1][1] = 1;
}

```

```

A[1][2] = 0;
A[2][0] = 0;
A[2][1] = 0;
A[2][2] = 2;

// Set up vector
double *v;
v = new double[n];
v[0] = 1;
v[1] = 2;
v[2] = 3;

// Test the function
eigenall(A, v, n, 1e-8);
}

void eigenshifttest()
{
    int n = 3;
    double**A;
    A = new double*[n];
    for (int i = 0; i < n; i++)
    {
        A[i] = new double[n];
    }
    // Set up the matrix
    A[0][0] = 1;
    A[0][1] = 2;
    A[0][2] = 0;
    A[1][0] = 2;
    A[1][1] = 1;
    A[1][2] = 0;
    A[2][0] = 0;
    A[2][1] = 0;
    A[2][2] = 2;

    // Set up vector
    double *v;
    v = new double[n];
    v[0] = 1;
    v[1] = 2;
    v[2] = 3;

    // Test the method
}

```

```
    eigenshift(A, v, n, 1e-8);
}
```

2.3 myEigenMain.cpp

```
/* ****
```

This is a template main file for the ENGSCI331 Eigenvectors module. It demonstrates some new C++ syntax and functions, as described in the accompanying document ENGSCI331-Eigenstuff.pdf.

**** There are some examples of "bad" programming in here (bits missing etc) that you will need to find and fix, though this file should compile without errors straight away. ****

You should use this file to get you started on the assignment. You're welcome to change whatever you'd like to as you go, this is only a starting point.

```
===== */
```

```
#define _CRT_SECURE_NO_DEPRECATE
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <cmath>
```

```
// This is the header file for your functions. Usual programming practise would be to use a *.cpp
// file that has the same name (ie: myEigenFunctions.cpp) and include it as normal in your project.
// Inside this file you'll see some ideas for functions that you could use during this project. I
// suggest you plan out your code first to see what kind of functions you'll use repeatedly and then
// write them.
```

```
#include "myEigenFunctions.h"
```

```
using namespace std;
```

```
#define PI 3.14159265358979323846
```

```
int main(void)
```

```
{
```

```
    // _____
```

```
    //
```

```
    // PART 1: Initialisation
```

```
    //
```

```
    // _____
```

```
// Defining local variables to be used:  
  
// n is the dimensions of the matrix, will be square for the eigen problems  
// the option whether to read the matrix from a file or to construct from  
// user-entered values of k and m  
  
int n = 0,  
    option = 0;  
  
double *M = NULL,  
        *K = NULL;  
  
double **A = NULL;  
double **B = NULL;  
  
ifstream infile;  
ofstream outfile;  
string filename;  
  
// Test all the methods  
powermethodanddeflatetest();  
eigenalltest();  
eigenshifttest();  
  
// Prompt and read number of masses in system  
cout << "Enter the number of masses in system: ";  
cin >> n;  
cout << endl;  
  
// Allocating memory for the 1D arrays - these are the number of masses, n, long:  
M = new double[n];  
K = new double[n];  
  
// Length for iterating  
int length = 10;  
  
// Allocating memory for the 2D arrays - these have dimensions of n*n:  
A = new double*[n];  
for (int i = 0; i < n; i++)  
    A[i] = new double[n];
```

```

B = new double*[n];
for (int i = 0; i < n; i++)
    B[i] = new double[n];

// _____
// _____
// PART 2: Populating matrices from user variables OR from a file
// _____
// _____
cout << "A matrix built in the code (option 1):";
cin >> option;
cout << endl;

switch (option) {
case 1:

    // Reading in the A matrix from a file
    //
    //                                     You get to do this bit!
    //

    break;
default:
    cout << "ERROR: Option " << option << " is unrecognised. Enter 1." << endl;
    break;
}
// Set in all the values
double G = 7.929e10;
double rho = 7751;
double D = 0.005;
double R = 0.0532;
double Na = 10;

// Set all values in the matrix to 0 first
for (int i = 0; i < length; i++)
{
    for (int j = 0; j < length; j++)
    {
        A[i][j] = 0;
    }
}

```

```

// Assign the mass and k values to the values to the 1D arrays
for (int i = 0; i < length; i++)
{
    M[i] = (PI*PI*D*D*rho*R) / 2;
    K[i] = (G*D*D*D*D / (64 * R*R*R));
}
K[0] = 2 * K[0];

// Assign the values to the A matrix and solve

// First Cart
A[0][0] = (-K[0] - K[1]) / M[0];
A[0][1] = K[1] / M[0];

// Second Cart
A[1][0] = K[1] / M[1];
A[1][1] = (-K[1] - K[2]) / M[1];
A[1][2] = K[2] / M[1];

// Third Cart
A[2][1] = K[2] / M[2];
A[2][2] = (-K[2] - K[3]) / M[2];
A[2][3] = K[3] / M[2];

// Forth Cart
A[3][2] = K[3] / M[3];
A[3][3] = (-K[3] - K[4]) / M[3];
A[3][4] = K[4] / M[3];

// Fifth Cart
A[4][3] = K[4] / M[4];
A[4][4] = (-K[4] - K[5]) / M[4];
A[4][5] = K[5] / M[4];

// Sixth Cart
A[5][4] = K[5] / M[5];
A[5][5] = (-K[5] - K[6]) / M[5];
A[5][6] = K[6] / M[5];

// Seventh Cart
A[6][5] = K[6] / M[6];
A[6][6] = (-K[6] - K[7]) / M[6];
A[6][7] = K[7] / M[6];

```

```

// Eighth Cart
A[7][6] = K[7] / M[7];
A[7][7] = (-K[7] - K[8]) / M[7];
A[7][8] = K[8] / M[7];

// Ninth Cart
A[8][7] = K[8] / M[8];
A[8][8] = (-K[8] - K[9]) / M[8];
A[8][9] = K[9] / M[8];

// Tenth Cart
A[9][8] = K[9] / M[9];
A[9][9] = -K[9] / M[9];

// Define the initial guess for the eigen vector
double *x;
x = new double[length];
x[0] = 1;
x[1] = 2;
x[2] = 3;
x[3] = 4;
x[4] = 5;
x[5] = 6;
x[6] = 7;
x[7] = 8;
x[8] = 9;
x[9] = 10;

// Assign a values for a copy of the A matrix
for (int i = 0; i < length; i++)
{
    for (int j = 0; j < length; j++)
    {
        B[i][j] = A[i][j];
    }
}
// -----
// -----
//      PART 3: Solving the eigen problem
// -----

```

```
// Use eigenshift to get the natural frequencies
// and eigenvectors for the lowest and highest
// calculated spring nodes
eigenshift(A, x, length, 1e-8);
// Reset the matrix
for (int i = 0; i < length; i++)
{
    for (int j = 0; j < length; j++)
    {
        A[i][j] = B[i][j];
    }
}
// Use the eigenall to get the natural frequencies and eigen vectors for all 10 modes
eigenall(A, x, length, 1e-8);
```

```
// _____
// _____
//      PART 4: Displaying/writing the results
// _____
// _____
```

```
// Printed to the command line and copied into an
// excel spreadsheet. See the report for screenshots and tables.
```

```
// _____
// _____
//      PART 5: Housekeeping
// _____
// _____
```

```
for(int i = 0; i < n; i++) {
    delete [] A[i];
}
delete [] A;

cout << "I'm finished!"<<endl;
```

```
}
```

2.4 Screenshots from Testing and Output

```
3
0.707107 0.707107 8.40088e-05
-0.5 0.5 -0.00017821
0.5 -0.5 -0.00017821
-0.00017821 -0.00017821 2
```

Figure 5: Power Method and Deflation Test

```
Eigen small
-1
Eigen large
3
Eigen small Vector
-0.707107 0.707107 4.0461e-06
Eigen large Vector
0.707107 0.707107 8.40088e-05
```

Figure 6: Eigen Shift Test

```

Eigenvalue
3
Eigen Vector After Power Method
0.707107 0.707107 8.40088e-05
Matrix Before Deflating
1 2 0
2 1 0
0 0 2

Matrix After Deflating
-0.5 0.5 -0.00017821
0.5 -0.5 -0.00017821
-0.00017821 -0.00017821 2

Eigenvalue
2
Eigen Vector After Power Method
-6.87579e-05 -0.000109452 1
Matrix Before Deflating
-0.5 0.5 -0.00017821
0.5 -0.5 -0.00017821
-0.00017821 -0.00017821 2

Matrix After Deflating
-0.5 0.5 -4.06937e-05
0.5 -0.5 4.06937e-05
-4.06937e-05 4.06937e-05 4.13995e-09

Eigenvalue
-1
Eigen Vector After Power Method
0.707107 -0.707107 5.75496e-05
Matrix Before Deflating
-0.5 0.5 -4.06937e-05
0.5 -0.5 4.06937e-05
-4.06937e-05 4.06937e-05 4.13995e-09

Matrix After Deflating
1.32328e-09 1.32328e-09 -7.31426e-13
1.32328e-09 1.32328e-09 -1.24934e-13
-7.31426e-13 -1.24934e-13 7.45191e-09

```

Figure 7: Eigen All Test

```

Eigensmall
-2489.19
Eigenlarge
-401867
Eigensmall Vector
0.0349489 0.104013 0.170591 0.233074 0.289934 0.339752 0.381265 0.413399 0.43531 0.446414
Eigenlarge Vector
-0.446421 0.435315 -0.413402 0.381265 -0.339749 0.289928 -0.233067 0.170584 -0.104008 0.0349472

```

Figure 8: Eigen Shift Output

```

Eigenvalue
-401867

Eigen Vector After Power Method
-0.446421 0.435315 -0.413402 0.381265 -0.339749 0.289928 -0.233067 0.170584 -0.104008 0.0349472

Eigenvalue
-382320

Eigen Vector After Power Method
0.434465 -0.339547 0.170456 0.0358686 -0.23439 0.381823 -0.446066 0.413173 -0.290349 0.10435

Eigenvalue
-345140

Eigen Vector After Power Method
0.413034 -0.17091 -0.171405 0.413275 -0.412982 0.170733 0.171537 -0.413374 0.413192 -0.171121

Eigenvalue
-293965

Eigen Vector After Power Method
-0.381203 -0.0352122 0.413183 -0.339861 -0.104674 0.434939 -0.290276 -0.171358 0.445936 -0.233683

Eigenvalue
-233806

Eigen Vector After Power Method
-0.340007 -0.233716 0.413088 0.104554 -0.445813 0.0349069 0.434918 -0.171035 -0.381395 0.290453

Eigenvalue
-170551

Eigen Vector After Power Method
-0.29035 -0.381223 0.171072 0.434809 -0.0350373 -0.445852 -0.104457 0.413245 0.233749 -0.340158

Eigenvalue
-110391

Eigen Vector After Power Method
0.233701 0.445868 0.171094 -0.290514 -0.434843 -0.104314 0.340101 0.413121 0.0350452 -0.381285

Eigenvalue
-59216.6

Eigen Vector After Power Method
0.171163 0.413213 0.413184 0.171107 -0.171189 -0.413184 -0.413138 -0.171098 0.171148 0.413138

Eigenvalue
-22036.1

Eigen Vector After Power Method
-0.104405 -0.290454 -0.413183 -0.44584 -0.381307 -0.233656 -0.0350759 0.171146 0.340059 0.434845

Eigenvalue
-2489.15

Eigen Vector After Power Method
-0.0350883 -0.104401 -0.171142 -0.23367 -0.290443 -0.340065 -0.381312 -0.413171 -0.434856 -0.445834

```

Figure 9: Eigen All Output

2018

SEMESTER 2

ENGSCI 331 Finite Differences

*Connor McDowall
530913386
cmcd398*

October 3, 2018

Contents

1	Part 1	2
1.1	Visualisations	2
1.2	Questions	3
2	Part 2	4
2.1	Task 4	5
3	Part 3	6
3.1	Task 5	6
	3.1.1 Q5	7

Listings

List of Figures

1	1D $\frac{d^2u}{dx^2} = 0$: Numerical vs Analytical Comparison	2
2	Contour plot for $\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = 0$	2
3	Contour plot for $\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = -2$	3
4	Explicit Scheme Upper and Lower Temporal Resolution Comparison for $\frac{\delta^2u}{\delta x^2} - \alpha \frac{\delta u}{\delta t} = 0$	4
5	Scheme comparison for lower and upper resolutions for $\frac{\delta^2u}{\delta x^2} - \alpha \frac{\delta u}{\delta t} = 0$	5
6	Scheme comparison for upper resolutions frt $\frac{\delta^2u}{\delta x^2} - \alpha \frac{\delta u}{\delta t} = 0$	5
7	PDE for $\frac{\delta^2u}{\delta x^2} + \frac{\delta^2u}{\delta y^2} - \alpha \frac{\delta u}{\delta t} = 0$ where $\alpha = 10$	6
8	PDE for $\frac{\delta^2u}{\delta x^2} + \frac{\delta^2u}{\delta y^2} - \alpha \frac{\delta u}{\delta t} = 0$ where $\alpha = 5$	7

1 Part 1

1.1 Visualisations

See 1 for the one dimensional comparison and 2 for the two dimensional contour plot.

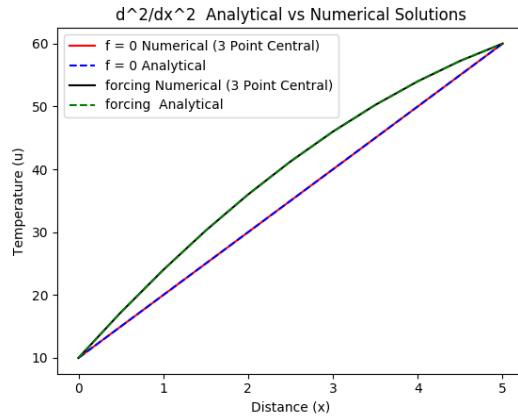


Figure 1: 1D $\frac{d^2u}{dx^2} = 0$: Numerical vs Analytical Comparison

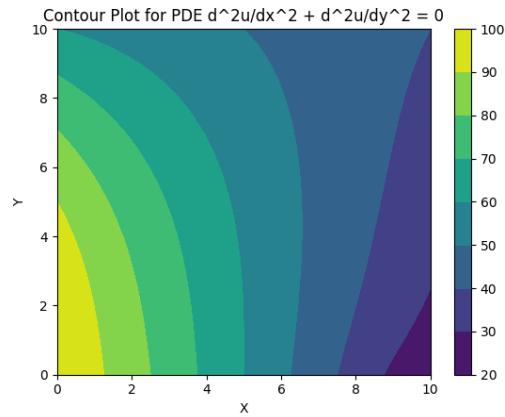


Figure 2: Contour plot for $\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = 0$

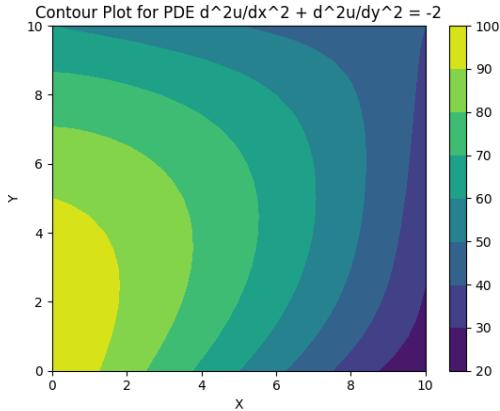


Figure 3: Contour plot for $\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = -2$

1.2 Questions

Q1A

The right hand side is an external heat source applied to the system. This is evident as the forcing terms on figure 1 with a higher temperature accross each distance value compared to the non forcing function

Q1B

Increase the number of finite points you use in the stencil, decreasing the truncation error.

Q2

2D interpolation techniques would provide a reasonable approximation. Bilinear interpolation uses linear interpolation in 2D dimensions. Both the finite difference and 2D interpolation techniques use existing points to solve unknown points. Both methods form systems of linear equations to solve unknowns via linear solvers. When considering how the boundary conditions are expressed (as analytic equations), new points are easy to estimate. Both methods may be used.

2 Part 2

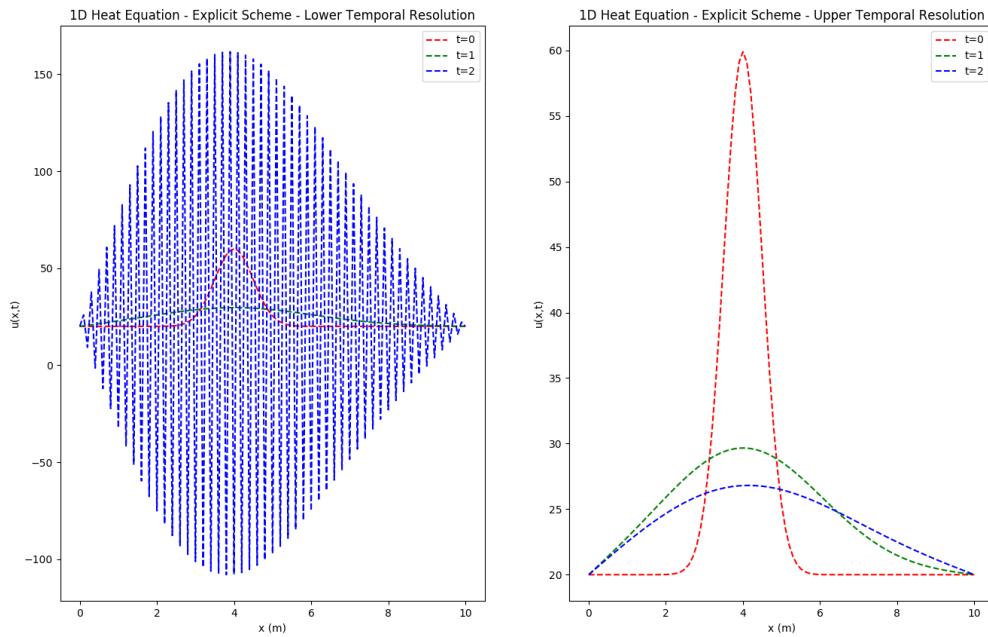


Figure 4: Explicit Scheme Upper and Lower Temporal Resolution Comparison for $\frac{\delta^2 u}{\delta x^2} - \alpha \frac{\delta u}{\delta t} = 0$

Q3

The higher resolution is numerically more accurate. The implicit method with the lower temporal solution oscillates wildly, as seen in 4 on the left. Our numerical model may have exceeded an r value ($|0.5|$) . The time step would have been too large, resulting in this numerical instability and oscillating behaviour.

2.1 Task 4

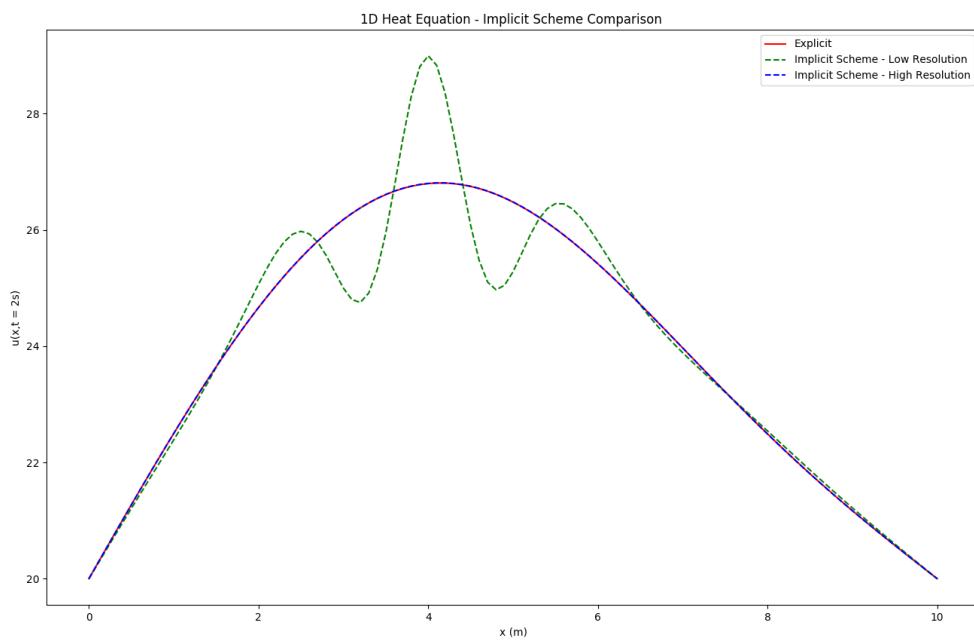


Figure 5: Scheme comparison for lower and upper resolutions for $\frac{\delta^2 u}{\delta x^2} - \alpha \frac{\delta u}{\delta t} = 0$

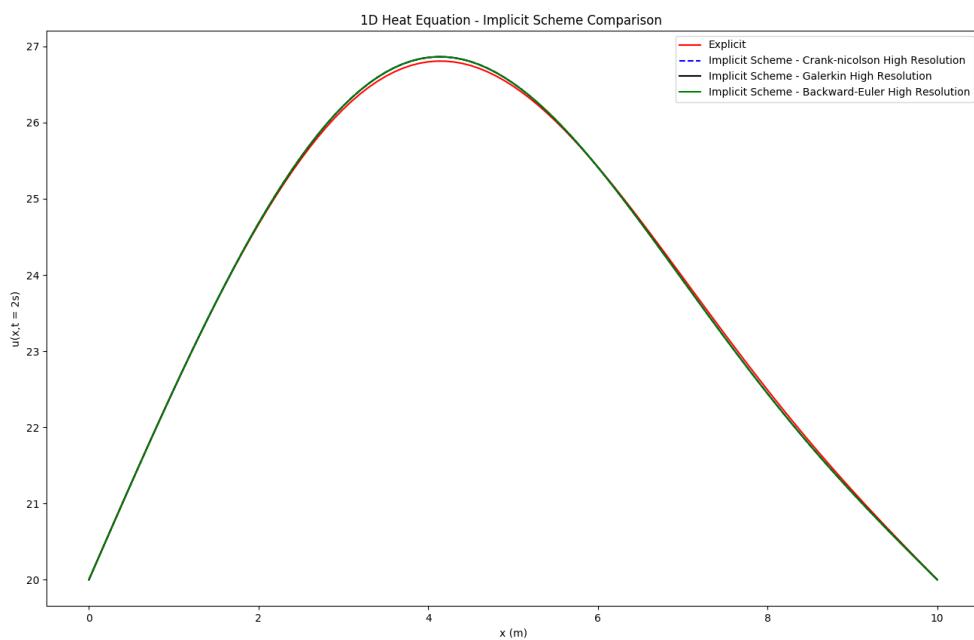


Figure 6: Scheme comparison for upper resolutions frt $\frac{\delta^2 u}{\delta x^2} - \alpha \frac{\delta u}{\delta t} = 0$

Q4a

Implicit methods are more numerically stable and are convergent but are computationally more intensive numerically than the explicit methods, therefore less efficient. The methods can use relatively large time steps and still converge. Implicit methods are computationally strenuous as a system of simultaneous equations must be solved at each time step.

Q4b

It is reasonable. As seen on 5, the r value exceeds the stability threshold for the lower temporal resolution. With the higher temporal resolution, the numerical stability threshold is met across all implicit methods (6). It is reasonable to assume, as shown with the explicit and implicit methods being approximately the same. The implicit methods should be more accurate than the explicit. If they are about the same, they are reasonable.

3 Part 3

3.1 Task 5

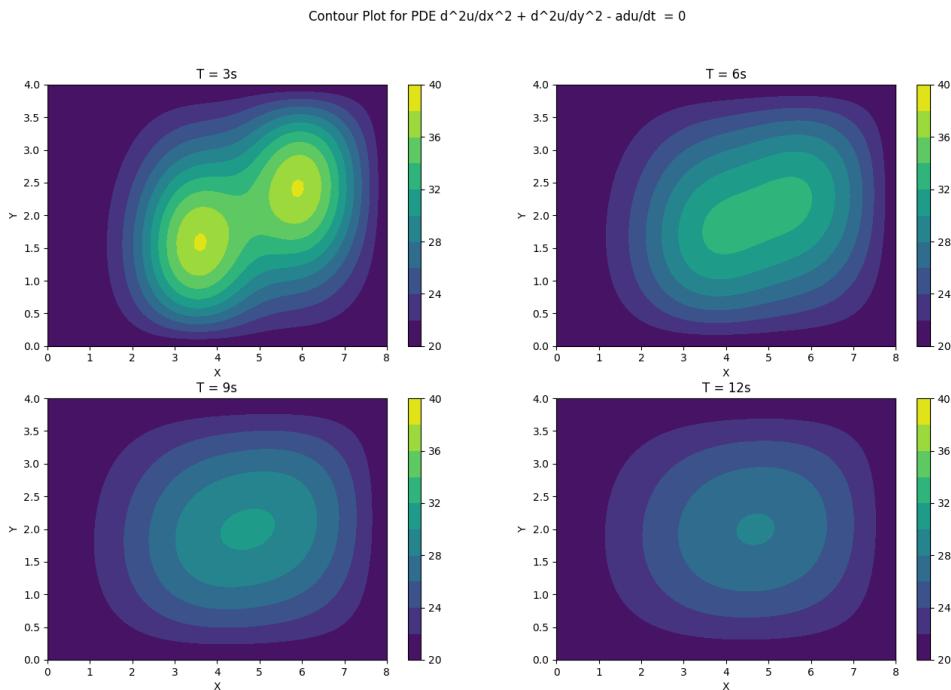


Figure 7: PDE for $\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} - \alpha \frac{\delta u}{\delta t} = 0$ where $\alpha = 10$

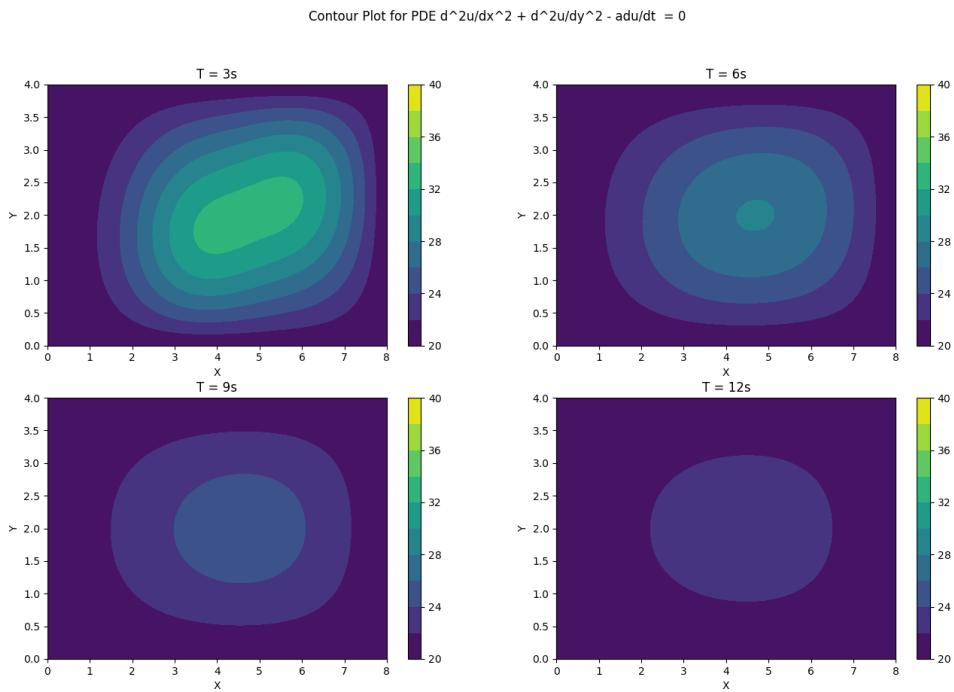


Figure 8: PDE for $\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} - \alpha \frac{\delta u}{\delta t} = 0$ where $\alpha = 5$

3.1.1 Q5

The temperature distribution diminishes over time. Initial, there are large temperarture discrepancies. As time progresses, the discrepancies decreases and the peak temperatures are lower. Contour bands are larger and cooler as time progresses.

As seen in 8, the peak temperatures are much lower with the overall surface cooler. A lower α (thermal diffusivity) decreases the rate heat is transferred from hot parts of the plate to cooler parts. For this reason, the contours are wider with less intense heat as the plate cools rather than transferring heat. This behaviour decreases the heat flow efficiency through the plate.

2018

SEMESTER 2

Lab 3: Non Linear Equations

*Connor McDowall
530913386
cmcd398*

September 4, 2018

Contents

1	Questions	2
1.1	Task 2	2
1.1.1	Question 1	2
1.1.2	Question 2	2
1.2	Task 4	2
1.2.1	Question 1	2
1.2.2	Question 2	2
2	Plots	2
3	Code: Non Linear Equations	5
3.1	NLE Functions	5
3.2	NLE Plotting	10
4	Systems of Non Linear Equations	12
4.1	Newton Two Variable	12
4.2	Newton Two Variable Plotting	15
4.3	Newton Multiple Variable	18

Listings

1	Bisection	5
2	Secant	6
3	Regula Falsi	6
4	Newton	7
5	Combined	8
6	Task 1	9
7	Task 2	10
8	Newton Two Variables	12
9	Task 3	14
10	Task 4	15
11	Newton Multiple Variables	18

List of Figures

1	NLE Function Comparison	3
2	NLE Function Comparison	4

1 Questions

1.1 Task 2

1.1.1 Question 1

The newton method finds the roots for $f(x) = x^2 - 1$ and $f(x) = \cos(x) + \sin(x^2) - 0.5$ in four and seven iterations respectively. The newton method fails to find the root before the maximum number of iterations for $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. The newton method uses the derivative to calculate the new root value. If the derivative is too small, the new root estimated is significantly greater than the current iteration. This was the case for $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ as jumped from a negative function value to a large, positive function value. The combined method uses a combination of the bisection and newton method. The newton method is used first to find a new root estimate. If this estimate falls outside a root range, the bisection method is used instead to find the new root estimate, avoiding extreme leaps. Thereafter, the root bracket is updated. The method continues to iterate until a suitable root is found. The combined method found a root in 4 iterations for $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. (I have excluded the initial root estimates from my iterations).

1.1.2 Question 2

Use parallelization. Set up array of root estimates on a plausible range of values . Next, apply the desired root finding method on each element of the array The desired root finding method is based on the method's properties. Perform the method for each element simulataneously and select the best of the root estimates found. The best root estimates will be the global minima/maxima.

1.2 Task 4

1.2.1 Question 1

The intial root estimate is either too far away from the actual root or the derivative of the function is really small or zero.

1.2.2 Question 2

Use parallelization. Set up matrix of root estimates on a plausible range of values (two to n dimensions) where each column is a different combination of starting points and each row is a different variable in the function set. Next, apply the desired root finding method on each column of the matrix. The desired root finding method is based on the properties you wish to have. Perform the method for each column simulataneously and select the best of the root estimates found. The best root estimates will be the global minima/maxima. This will work for non linear functions with two to n dimensions.

2 Plots

The plots for both Tasks 2 and 4 are on the following pages.

3

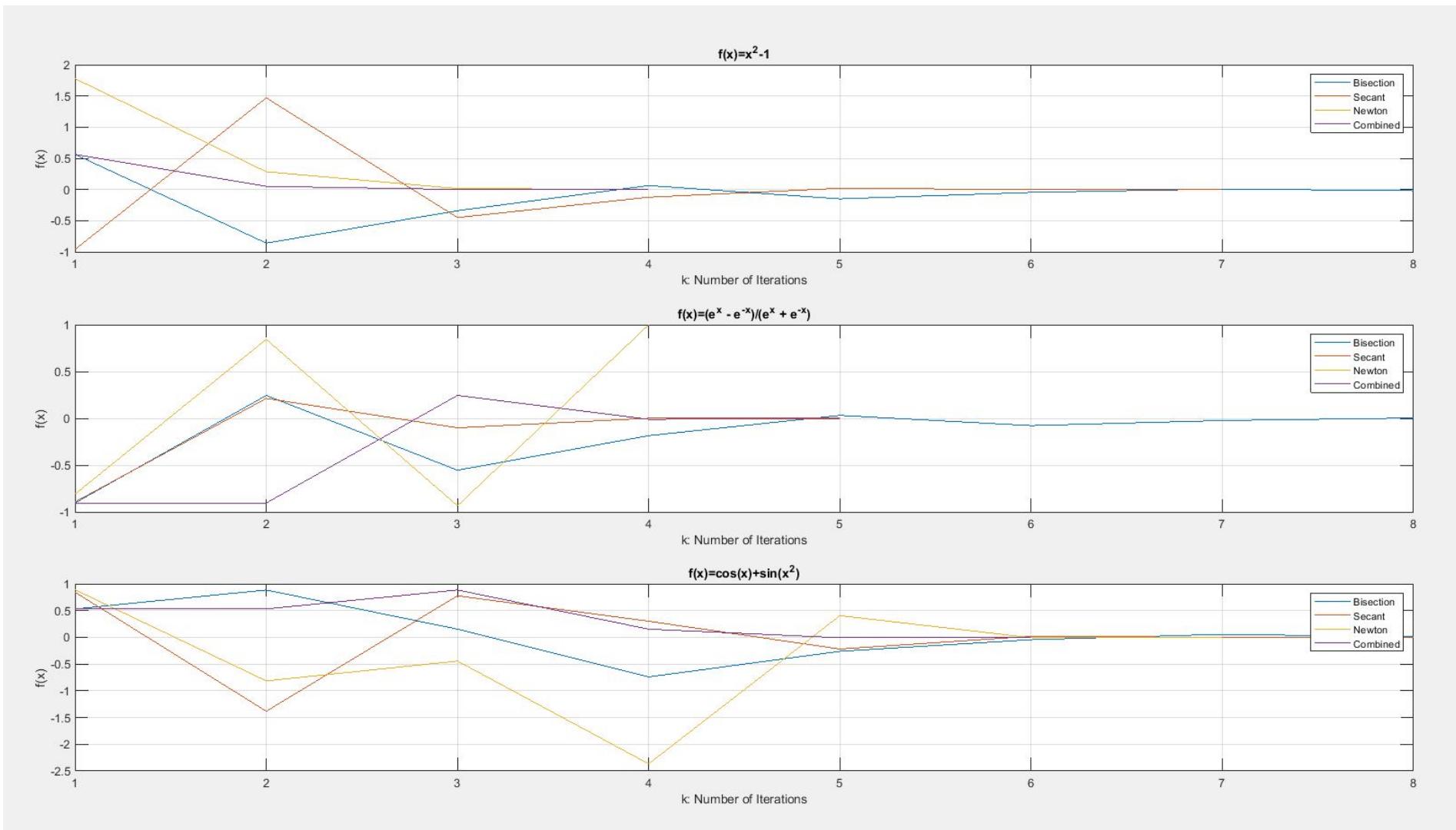


Figure 1: NLE Function Comparison

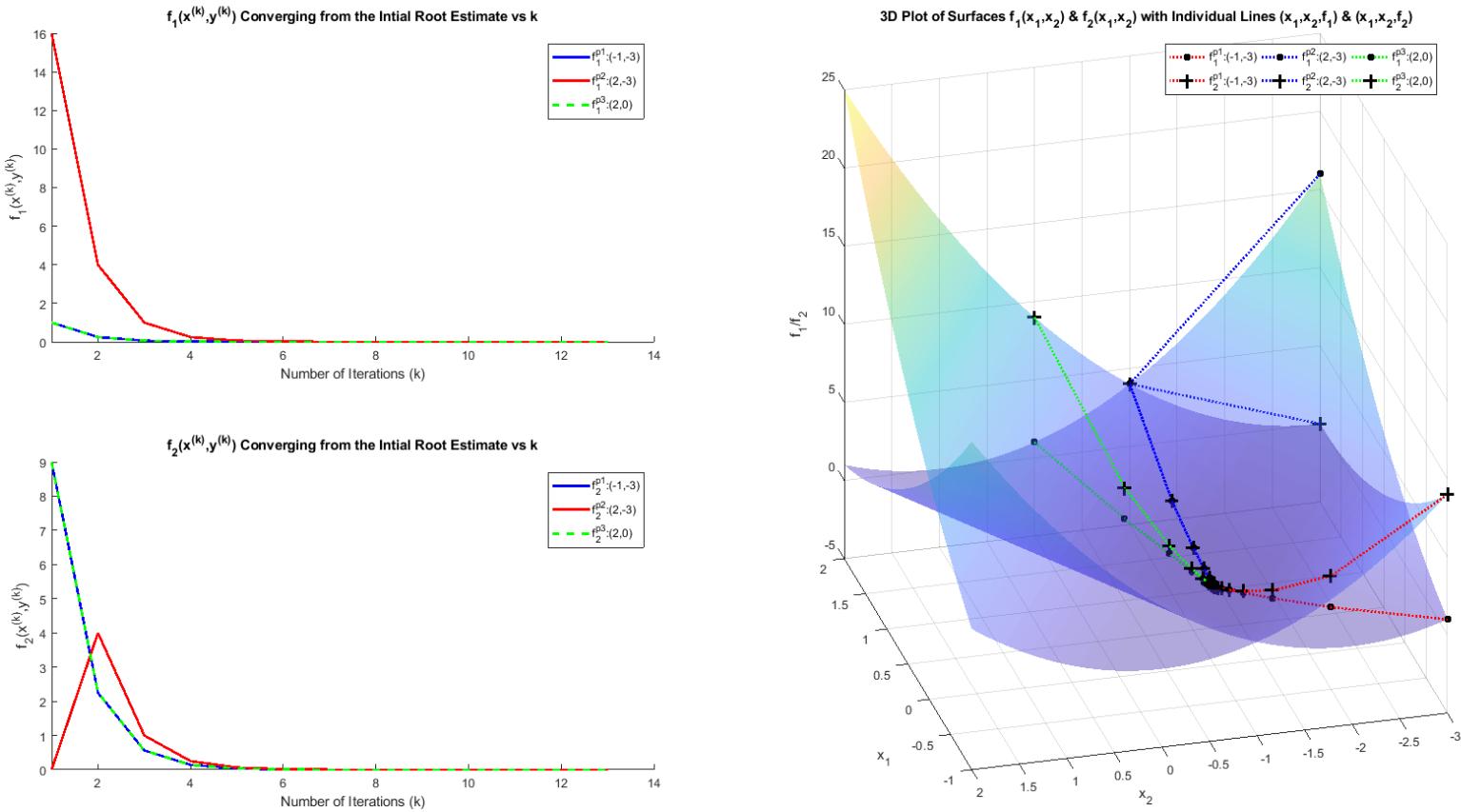


Figure 2: NLE Function Comparison

3 Code: Non Linear Equations

3.1 NLE Functions

Listing 1: Bisection

```

1 % Nonlinear equation root finding by the bisection method.
2 % Inputs
3 % f      : nonlinear function
4 % xl, xr : initial root bracket
5 % nmax   : maximum number of iterations performed
6 % tol    : numerical tolerance used to check for root
7 % Outputs
8 % x      : one-dimensional array containing estimates of root
9
10 % Hint 1:
11 % Iterate until either a root has been found or maximum number of
12 % iterations has been reached
13
14 % Hint 2:
15 % Check for root each iteration, making use of tol
16
17 % Hint 3:
18 % Update the bracket each iteration
19
20 function x = Bisection(f, xl, xr, nmax, tol)
21     % Initial array of root estimates, iteration and variable
22     % stores.
23     x = [xl,xr];
24     xbrac = x;
25     n = 1;
26     % Set iterative loop for the function
27     while n < nmax + 1
28         % Calculate the new root
29         xnew = xbrac(1) + ((xbrac(2) - xbrac(1))/2);
30         % Append the root estimate to the array
31         x = [x,xnew];
32         % Terminate function if at derivative point
33         if abs(f(xnew)) <= tol
34             return
35         % Calculate sign on the function with the new root
36         % and find the new root bracket.
37         elseif (f(xnew)*f(xbrac(1)))> 0
38             % Reset the bracket with new LHS
39             xbrac = [xnew,xbrac(2)];
40             elseif (f(xnew)*f(xbrac(2)))> 0
41                 % Reset the bracket with new RHS
42                 xbrac = [xbrac(1),xnew];
43             end
44             % Increase iteration counter
45             n = n + 1;
46     end

```

```

45 % Warn the user the maximum number of iterations have been
46 % performed
47 disp('The maximum number of iterations have been performed
48 without satisfying the required root finding condition');
end

```

Listing 2: Secant

```

1 % Nonlinear equation root finding by the secant method.
2 % Inputs
3 % f : nonlinear function
4 % x0, x1 : initial root bracket
5 % nmax : maximum number of iterations performed
6 % tol : numerical tolerance used to check for root
7 % Outputs
8 % x : one-dimensional array containing estimates of root
9
10 function x = Secant(f, x0, x1, nmax, tol)
11 % Initial array of root estimates, iteration and variable stores.
12 x = [x0, x1];
13 n = 1;
14 k = 2;
15 % Set iterative loop for the function
16 while n <= nmax
17     % Calculate the new root
18     xnew = x(k) - (f(x(k)) * (x(k) - x(k-1)) / (f(x(k)) - f(x(k-1))));
19     % Append the root estimate to the array
20     x = [x, xnew];
21     % Terminate function if at derivative point
22     if abs(f(xnew)) < tol
23         return
24     end
25     % Increase iteration counter and k value
26     n = n + 1;
27     k = k + 1;
28 end
29 % Warn the user the maximum number of iterations have been
30 % performed
31 disp('The maximum number of iterations have been performed
32 without satisfying the required root finding condition');

```

Listing 3: Regula Falsi

```

1 % Nonlinear equation root finding by the Regula falsi method.
2 % Inputs
3 % f : nonlinear function
4 % xl, xr : initial root bracket
5 % nmax : maximum number of iterations performed
6 % tol : numerical tolerance used to check for root
7 % Outputs

```

```

8 % x      : one-dimensional array containing estimates of root
9
10 function x = Regulafalsi(f, xl, xr, nmax, tol)
11 % Initial array of root estimates, iteration and variable stores
12 .
13 x = [xl,xr];
14 xbrac = x;
15 n = 1;
16 % Set iterative loop for the function
17 while n <= nmax
18     % Calculate the new root
19     xnew = xbrac(2) - (f(xbrac(2))*(xbrac(2)-xbrac(1))/(f(
20         xbrac(2))- f(xbrac(1))));
21     % Append the root estimate to the array
22     x = [x,xnew];
23     % Terminate function if at derivative point
24     if abs(f(xnew)) < tol
25         return
26     % Calculate sign on the function with the new root
27     % and find the new root bracket.
28     elseif f(xnew)*f(xbrac(1))>0
29         % Reset the bracket with new LHS
30         xbrac = [xnew,xbrac(2)];
31     elseif f(xnew)*f(xbrac(2))>0
32         % Reset the bracket with new RHS
33         xbrac = [xbrac(1),xnew];
34     end
35     % Increase iteration counter
36     n = n + 1;
37 end
38 % Warn the user the maximum number of iterations have been
39     performed
40 disp('The maximum number of iterations have been performed
41 without satisfying the required root finding condition');

```

Listing 4: Newton

```

1 % Nonlinear equation root finding by Newton's method
2 % Inputs
3 % f      : nonlinear function
4 % x0     : initial root estimate
5 % h      : step size for central difference formula
6 % nmax   : maximum number of iterations performed
7 % tol    : numerical tolerance used to check for root
8 % Outputs
9 % x      : one-dimensional array containing estimates of root
10
11 function x = Newton(f, x0, h, nmax, tol)
12 % Initial array of root estimates, iteration and variable stores.
13 x = x0;
14 n = 1;
15 k = 1;

```

```

16 % Set iterative loop for the function
17 while n < nmax
18     % Calculate the new root
19     xnew = x(k) - (f(x(k)))/ ((f(x(k)+h) - f(x(k)-h))/(2*h));
20     % Append the root estimate to the array
21     x = [x,xnew];
22     % Terminate function if at derivative point
23     if abs(f(xnew)) < tol
24         return
25     end
26     % Increase iteration counter and k value
27     n = n + 1;
28     k = k + 1;
29 end
30 % Warn the user the maximum number of iterations have been
31     performed
31 disp('The maximum number of iterations have been performed
      without satisfying the required root finding condition');

```

Listing 5: Combined

```

1 % Nonlinear equation root finding by the combined bisection/
  Newton's method
2 % Inputs
3 % f      : nonlinear function
4 % xl, xr : initial root bracket
5 % h      : step size for central difference formula
6 % nmax   : maximum number of iterations performed
7 % tol    : numerical tolerance used to check for root
8 % Outputs
9 % x      : one-dimensional array containing estimates of root
10
11 function x = Combined(f, xl, xr, h, nmax, tol)
12 % Initial array of root estimates, iteration and variable stores.
13     x = [xl,xr];
14     xbrac = x;
15     n = 1;
16     % Calculate the starting estimate
17     xstart = xbrac(1) + (xbrac(2) - xbrac(1))/2;
18     % Append the starting value
19     x = [x,xstart];
20     k = 3;
21     % Set iterative loop for the function
22     while n < nmax
23         % Use newton method to calculate the new root estimate
24         xnew = x(k) - (f(x(k)))/ ((f(x(k)+h) - f(x(k)-h))/(2*h));
25         % Use if condition to check inside the bracket
26         if (xnew < xbrac(1)) || (xnew > xbrac(2))
27             % Use the bisection method to get a better estimate
28             xnew = xbrac(1) + (xbrac(2) - xbrac(1))/2;
29         end
30         % Append the root estimate to the array

```

```

31      x = [x,xnew];
32      % Terminate function if at derivative point
33      if abs(f(xnew)) < tol
34          return
35      % Calculate sign on the function with the new root
36      % and find the new root bracket.
37      elseif f(xnew)*f(xbrac(1))>0
38          % Reset the bracket with new LHS
39          xbrac = [xnew,xbrac(2)];
40      elseif f(xnew)*f(xbrac(2))>0
41          % Reset the bracket with new RHS
42          xbrac = [xbrac(1),xnew];
43      end
44      % Increase iteration counter and k count by one.
45      n = n + 1;
46      k = k + 1;
47  end
48  % Warn the user the maximum number of iterations have been
49  % performed
50  disp('The maximum number of iterations have been performed
51  without satisfying the required root finding condition');

```

Listing 6: Task 1

```

1 %% Task 1 - Bisection, Secant, Regula Falsi and Newton's Methods
2 % You do NOT need to modify this script
3
4 % clear workspace
5 clear
6 clc
7
8 % Initialisation
9 f = @(x) 2*x.^2-8*x+4; % function to evaluate
10 tol = 1.0e-4;           % tolerance for asserts
11 h = 1.0e-4;             % step size for numerical estimate of
12 % gradient
13 x0 = 0.0;               % initial interval left
14 x1 = 2.0;               % initial interval right
15 nmax = 50;              % maximum number of iterations
16
17 % Bisection method
18 xb = Bisection(f, x0, x1, nmax, tol);
19 assert(abs(f(xb(end))) < tol)
20 disp(['Bisection converged to root at x = ' num2str(xb(end))]);
21
22 % Secant method
23 xs = Secant(f, x0, x1, nmax, tol);
24 assert(abs(f(xs(end))) < tol)
25 disp(['Secant converged to root at x = ' num2str(xs(end))]);
26
27 % Regula Falsi method and verification
28 xrf = Regulafalsi(f, x0, x1, nmax, tol);

```

```

28 assert(abs(f(xrf(end))) < tol)
29 disp(['Regula Falsi converged to root at x = ' num2str(xrf(end))])
30
31 % Newton's method and verification
32 xn = Newton(f, x0, h, nmax, tol);
33 assert(abs(f(xn(end))) < tol)
34 disp(['Newton converged to root at x = ' num2str(xn(end))]);
35
36 % Combined Bisection/Newton's method and verification
37 xc = Combined(f, x0, x1, h, nmax, tol);
38 assert(abs(f(xc(end))) < tol)
39 disp(['Combined Bisection/Newton converged to root at x = ' num2str(xc(end))]);

```

3.2 NLE Plotting

Listing 7: Task 2

```

1 %% Task 2 - Iterative Algorithm Comparison
2
3 % clear workspace
4 clear
5 clc
6
7 % Initialisation
8 tol = 1.0e-4; % tolerance for asserts
9 h = 1.0e-4; % step size for numerical estimate of gradient
10 nmax = 20; % maximum number of iterations
11
12 % functions to test algorithms on
13 f1 = @(x) x.^2 - 1; % function 1
14 f2 = @(x) (exp(x)-exp(-x))./(exp(x)+exp(-x)); % function 2
15 f3 = @(x) cos(x)+sin(x.*x)-0.5; % function 3
16
17 % initial root estimates for each function
18 % column 1: x0 for bisection, secant, regula falsi and combined
19 % methods
20 % column 2: x1 for bisection, secant, regula falsi and combined
21 % methods
22 % column 3: x0 for Newton's method
23 xint1 = ([-3.0,0.5,-3.0]);
24 xint2 = ([-5.,2.,1.1]);
25 xint3 = ([-2.0,1.5,-0.40]);
26
27 % function titles for plots
28 title1 = 'f(x)=x^2-1';
29 title2 = 'f(x)=(e^x - e^{-x})/(e^x + e^{-x})';
30 title3 = 'f(x)=cos(x)+sin(x^2)';

```

```

30 % set disp_func = false when you don't need to produce plot of
31 % functions
32 disp_func = false;
33 if disp_func
34     x = linspace(-5.,5.,1000);
35     figure(1), clf
36     subplot(3,1,1)
37     plot(x,f1(x))
38     grid on, xlabel('x'), ylabel('f(x)'), title(title1)
39     subplot(3,1,2)
40     plot(x,f2(x))
41     grid on, xlabel('x'), ylabel('f(x)'), title(title2)
42     subplot(3,1,3)
43     plot(x,f3(x))
44     grid on, xlabel('x'), ylabel('f(x)'), title(title3)
45 end
46
47 %% find one root for each function using bisection, secant,
48 % newton's and combined methods
49 % Function 1
50 xB1 = Bisection(f1, xint1(1), xint1(2), nmax, tol);
51 xS1 = Secant(f1, xint1(1), xint1(2), nmax, tol);
52 xR1 = Regulafalsi(f1, xint1(1), xint1(2), nmax, tol);
53 xN1 = Newton(f1, xint1(3), h, nmax, tol);
54 xC1 = Combined(f1, xint1(1), xint1(2), h, nmax, tol);
55
56 % Function 2
57 xB2 = Bisection(f2, xint2(1), xint2(2), nmax, tol);
58 xS2 = Secant(f2, xint2(1), xint2(2), nmax, tol);
59 xR2 = Regulafalsi(f2, xint2(1), xint2(2), nmax, tol);
60 xN2 = Newton(f2, xint2(3), h, nmax, tol);
61 xC2 = Combined(f2, xint2(1), xint2(2), h, nmax, tol);
62
63 % Function 3
64 xB3 = Bisection(f3, xint3(1), xint3(2), nmax, tol);
65 xS3 = Secant(f3, xint3(1), xint3(2), nmax, tol);
66 xR3 = Regulafalsi(f3, xint3(1), xint3(2), nmax, tol);
67 xN3 = Newton(f3, xint3(3), h, nmax, tol);
68 xC3 = Combined(f3, xint3(1), xint3(2), h, nmax, tol);
69
70 %% individual plot for each function of f(x^k) vs k for each
71 % method
72 % i.e. each of the three plots (one per function) will have four
73 % lines, one for each method called.
74 figure(1), clf
75
76 % create top plot for function 1
77 % The initial root estimates have been excluded from the
78 % iterations.
79 subplot(3,1,1)

```

```

76 plot(1:length(xB1)-2,f1(xB1(3:length(xB1))))
77 hold on
78 plot(1:length(xS1)-2,f1(xS1(3:length(xS1))))
79 hold on
80 plot(1:length(xN1)-1,f1(xN1(2:length(xN1))))
81 hold on
82 plot(1:length(xC1)-2,f1(xC1(3:length(xC1))))
83 legend('Bisection','Secant','Newton','Combined')
84 xlim([1,8]);
85 grid on, xlabel('k: Number of Iterations'), ylabel('f(x)'), title
    (title1)
86
87 % create middle plot for function 2
88 subplot(3,1,2)
89 plot(1:length(xB2)-2,f2(xB2(3:length(xB2))))
90 hold on
91 plot(1:length(xS2)-2,f2(xS2(3:length(xS2))))
92 hold on
93 plot(1:length(xN2)-1,f2(xN2(2:length(xN2))))
94 hold on
95 plot(1:length(xC2)-2,f2(xC2(3:length(xC2))))
96 legend('Bisection','Secant','Newton','Combined')
97 xlim([1,8]);
98 grid on, xlabel('k: Number of Iterations'), ylabel('f(x)'), title
    (title2)
99
100 % create bottom plot for function 3
101 subplot(3,1,3)
102 plot(1:length(xB3)-2,f3(xB3(3:length(xB3))))
103 hold on
104 plot(1:length(xS3)-2,f3(xS3(3:length(xS3))))
105 hold on
106 plot(1:length(xN3)-1,f3(xN3(2:length(xN3))))
107 hold on
108 plot(1:length(xC3)-2,f3(xC3(3:length(xC3))))
109 legend('Bisection','Secant','Newton','Combined')
110 grid on, xlabel('k: Number of Iterations'), ylabel('f(x)'), title
    (title3)
111 xlim([1,8]);
112
113 % Save the plot
114 savefig('Task2Plot')

```

4 Systems of Non Linear Equations

4.1 Newton Two Variable

Listing 8: Newton Two Variables

```

1 % Nonlinear equation root finding in two dimensions using Newton's Method.
2 % Inputs
3 % func : array of function handles for system of nonlinear equations
4 % x0 : vector of initial root estimates for each independent variable
5 % h : step size for numerical estimate of partial derivatives
6 % nmax : maximum number of iterations performed
7 % tol : numerical tolerance used to check for root
8 % Outputs
9 % x : two-dimensional array (two-row matrix) containing estimates of root
10
11 % Hint 1:
12 % Include the initial root estimate as the first column of x
13
14 % Hint 2:
15 % Use MATLAB in-built functionality for solving the matrix equation for vector of updates, delta
16
17 % Hint 3:
18 % Check for root each iteration, continuing until the maximum number of iterations has been reached
19
20 function x = Newton2Var(func, x0, h, nmax, tol)
21 % Initial array of root estimates, iteration and variable stores.
22 % Initialise a storage array
23 xstore = transpose(x0);
24 x = xstore; % Vector of the most recent root variables
25 % Set iterative loop for the function
26 for i = 1:nmax
27     % Calculate the function variables from the most recent iteration.
28     f1 = func{1}(x);
29     f2 = func{2}(x);
30
31     %Calculate the derivatives for each of the points for the jacobian
32     f1x1 = (func{1}(x + [h;0])- func{1}(x - [h;0]))/(2*h);
33     f1x2 = (func{1}(x + [0;h])- func{1}(x - [0;h]))/(2*h);
34     f2x1 = (func{2}(x + [h;0])- func{2}(x - [h;0]))/(2*h);
35     f2x2 = (func{2}(x + [0;h])- func{2}(x - [0;h]))/(2*h);
36
37     % Set Jacobian and f
38     f = [f1;f2];
39     J = [f1x1,f1x2;f2x1,f2x2];
40
41     % Calculate the Delta
42     del = -1*linsolve(J,f);

```

```

43
44     % Find the new xvalues
45     xnew = del + x;
46
47     % Calculate the new f values
48     fnew = [func{1}(xnew);func{2}(xnew)];
49     %Use condition criteria to cancel out of the list
50     if (abs(fnew(1))< tol) && (abs(fnew(2))< tol)
51         % Add to the storage arrays
52         xstore = [xstore,xnew];
53         x = xstore;
54         return
55     end
56     xstore = [xstore,xnew];
57     x = xnew;
58 end
59 disp('The maximum number of iterations have been performed
      without satisfying the required root finding condition');
60 end

```

Listing 9: Task 3

```

1 %% Task 3 - System of Nonlinear Equations
2
3 % clear workspace
4 clear all
5 clc
6
7 % initialisation
8 tol = 1.0e-6;          % numerical tolerance
9 h = 1.0e-4;            % step size for central difference
10 nmax = 50;             % maximum number of iterations
11 x0 = [2,0];            % initial root estimate
12 func = {@f1, @f2};    % array of function handles
13
14 % set func_usage to false once you know how vector/array func
15 % works
15 func_usage = true;
16 if func_usage
17     f1_initial = func{1}(x0);
18     f2_initial = func{2}(x0);
19 end
20
21 % 2D Newton's method and verification
22 disp(['Newton2Var starting at point (x0,y0) = (' num2str(x0(1)),
23       ',',num2str(x0(2)),')']);
23 xn = Newton2Var(func, x0, h, nmax, tol);
24 disp([xn(1,end),xn(2,end)]);
25 assert(abs(func{1}([xn(1,end),xn(2,end)])) <= tol)
26 assert(abs(func{2}([xn(1,end),xn(2,end)])) <= tol)
27 disp(['Newton2Var converged to root at (x,y) = (' num2str(xn(1,
28       end)),',',num2str(xn(2,end)),') in ',num2str(length(xn)),']

```

```

    iterations']);
28
29
30 % Functions to be used for testing out Newton2Var
31 function f = f1(x)
32     f = x(1)*x(1)-2*x(1)+x(2)*x(2)+2*x(2)-2*x(1)*x(2)+1;
33 end
34 function f = f2(x)
35     f = x(1)*x(1)+2*x(1)+x(2)*x(2)+2*x(2)+2*x(1)*x(2)+1;
36 end

```

4.2 Newton Two Variable Plotting

Listing 10: Task 4

```

1 %% Task 4
2
3 % clear workspace
4 clear all
5 clc
6
7 % initialisation
8 tol = 1.0e-6;          % numerical tolerance
9 h = 1.0e-4;            % step size for central difference
10 nmax = 50;             % maximum number of iterations
11 x0_p1 = [-1,-3];      % initial root estimate - point 1
12 x0_p2 = [2,-3];       % initial root estimate - point 2
13 x0_p3 = [2,0];         % initial root estimate - point 3
14 func = {@f1, @f2};    % array of function handles
15
16 % Two function two variable Newton's method for each starting
   location
17 xn_p1 = Newton2Var(func, x0_p1, h, nmax, tol);
18 xn_p2 = Newton2Var(func, x0_p2, h, nmax, tol);
19 xn_p3 = Newton2Var(func, x0_p3, h, nmax, tol);
20
21 %% start figure for algorithm visualisation
22 figure(1), clf
23 % Calculate all the values needed
24
25 % Iteration count 1
26 [~,c1] =size(xn_p1);
27 k1 = 1:c1;
28
29 % Call the first function for each iteration from the first
   starting point.
30 for i = 1:c1
31     fp11(i) = func{1}([xn_p1(1,i),xn_p1(2,i)]);
32 end
33
34 %Iteration Count 2

```

```

35 [~,c2] =size(xn_p2);
36 k2 = 1:c2;
37
38 % Call the first function for each iteration from the second
   starting point.
39 for i = 1:c2
40     fp21(i) = func{1}([xn_p2(1,i),xn_p2(2,i)]);
41 end
42
43 % Iteration count 3
44 [~,c3] =size(xn_p3);
45 k3 = 1:c3;
46
47 % Call the first function for each iteration from the third
   starting point.
48 for i = 1:c3
49     fp31(i) = func{1}([xn_p3(1,i),xn_p3(2,i)]);
50 end
51
52 % Call the second function for each iteration from the first
   starting point.
53 for i = 1:c1
54     fp12(i) = func{2}([xn_p1(1,i),xn_p1(2,i)]);
55 end
56
57 % Call the first function for each iteration from the second
   starting point.
58 for i = 1:c2
59     fp22(i) = func{2}([xn_p2(1,i),xn_p2(2,i)]);
60 end
61
62 % Call the first function for each iteration from the third
   starting point.
63 for i = 1:c3
64     fp32(i) = func{2}([xn_p3(1,i),xn_p3(2,i)]);
65 end
66
67
68 % Create all the labels to plot with
69 f1title = 'f_{1}(x^{(k)},y^{(k)}) Converging from the Intial Root
   Estimate vs k';
70 f2title = 'f_{2}(x^{(k)},y^{(k)}) Converging from the Intial Root
   Estimate vs k';
71 surftitle = '3D Plot of Surfaces f_{1}(x_1,x_2) & f_{2}(x_1,x_
   {2}) with Individual Lines (x_1,x_2,f_1) & (x_1,x_2,f_2)';
72 f1 xlabel = 'Number of Iterations (k)';
73 f1 ylabel = 'f_{1}(x^{(k)},y^{(k)})';
74 f2 xlabel = 'Number of Iterations (k)';
75 f2 ylabel = 'f_{2}(x^{(k)},y^{(k)})';
76 surf xlabel = 'x_1';
77 surf ylabel = 'x_2';

```

```

78 surfzlabel = 'f_1/f_2';
79
80 % Create axis labels
81 f1p1 = 'f_1^{p1}:(-1,-3)';
82 f1p2 = 'f_1^{p2}:(2,-3)';
83 f1p3 = 'f_1^{p3}:(2,0)';
84 f2p1 = 'f_2^{p1}:(-1,-3)';
85 f2p2 = 'f_2^{p2}:(2,-3)';
86 f2p3 = 'f_2^{p3}:(2,0)';
87 f1legend = {f1p1,f1p2,f1p3};
88 f2legend = {f2p1,f2p2,f2p3};
89 surflegend = {f1p1,f2p1,f1p2,f2p2,f1p3,f2p3};
90
91 % create top left plot for function 1
92 subplot(2,2,1)
93 hold on
94 plot(k1,fp11,'b','Linewidth',2)
95 hold on
96 plot(k2,fp21,'r','Linewidth',2)
97 hold on
98 plot(k3,fp31,'g--','Linewidth',2)
99 ylabel('Function 1 values')
100 xlabel('Number of Iterations (k)')
101 title(f1title)
102 legend(f1legend)
103 xlabel(f1xlabel)
104 ylabel(f1ylabel)
105 xlim([1,14]);
106
107 % create bottom left plot for function 2
108 subplot(2,2,3)
109 hold on
110 plot(k1,fp12,'b','Linewidth',2)
111 hold on
112 plot(k2,fp22,'r','Linewidth',2)
113 hold on
114 plot(k3,fp32,'g--','Linewidth',2)
115 ylabel('Function 2 values')
116 xlabel('Number of Iterations (k)')
117 title(f2title)
118 legend(f2legend)
119 xlabel(f2xlabel)
120 ylabel(f2ylabel)
121 xlim([1,14]);
122
123 % create right plot for 3d visualisation of 2d newton's method
124 subplot(2,2,[2 4])
125 % Plot both the functions
126 [X,Y] = meshgrid(-1:0.1:2,-3:0.1:2);
127 Z1 = X.*X-2.*X+Y.*Y+2.*Y-2.*X.*Y+1;
128 Z2 = X.*X+2.*X+Y.*Y+2.*Y+2.*X.*Y+1;

```

```

129 surf(X,Y,Z1);
130 hold on;
131 surf(X,Y,Z2);
132
133 % Improve plotting
134 alpha 0.4; % Make more transparent
135 rotate3d on; % Automatic switch on rotate feature
136 shading interp; % Change shading
137 view(255,25); % Specify view found by trial and error.
138
139 % Plot the lines on the surface
140 hold on
141 ob1 = plot3(xn_p1(1,:),xn_p1(2,:),fp11,:r.,'Linewidth',2,
    'Markersize',20,'MarkerEdgeColor','k');
142 hold on
143 ob3 = plot3(xn_p2(1,:),xn_p2(2,:),fp21,:b.,'Linewidth',2,
    'Markersize',20,'MarkerEdgeColor','k');
144 hold on
145 ob5 = plot3(xn_p3(1,:),xn_p3(2,:),fp31,:g.,'Linewidth',2,
    'Markersize',20,'MarkerEdgeColor','k');
146 hold on
147 ob2 = plot3(xn_p1(1,:),xn_p1(2,:),fp12,:r+,'Linewidth',2,
    'Markersize',10,'MarkerEdgeColor','k');
148 hold on
149 ob4 = plot3(xn_p2(1,:),xn_p2(2,:),fp22,:b+,'Linewidth',2,
    'Markersize',10,'MarkerEdgeColor','k');
150 hold on
151 ob6 = plot3(xn_p3(1,:),xn_p3(2,:),fp32,:g+,'Linewidth',2,
    'Markersize',10,'MarkerEdgeColor','k');
152 ob = [ob1,ob2,ob3,ob4,ob5,ob6];
153
154 % Plot labels
155 legend(ob,surflegend,'Location','northeast','NumColumns',3)
156 title(surftitle);
157 xlabel(surfxlabel);
158 ylabel(surfylabel);
159 zlabel(surfzlabel);
160 grid on;
161
162 % Functions to be used for testing out Newton2Var
163 function f = f1(x)
164     f = x(1)*x(1)-2*x(1)+x(2)*x(2)+2*x(2)-2*x(1)*x(2)+1;
165 end
166 function f = f2(x)
167     f = x(1)*x(1)+2*x(1)+x(2)*x(2)+2*x(2)+2*x(1)*x(2)+1;
168 end

```

4.3 Newton Multiple Variable

Listing 11: Newton Multiple Variables

```

1 % Nonlinear equation root finding in n dimensions using Newton's
2 % Method.
3 % Inputs
4 % n : number of dimensions for Newton's method
5 % func : array of function handles for system of nonlinear
6 % equations
7 % x0 : vector of initial root estimates for each independent
8 % variable
9 % h : step size for numerical estimate of partial
10 % derivatives
11 % nmax : maximum number of iterations performed
12 % tol : numerical tolerance used to check for root
13 % Outputs
14 % x : array (n-row matrix) containing estimates of root
15
16 % Hint 1:
17 % Include the initial root estimate as the first column of x
18
19 % Hint 2:
20 % Use MATLAB in-built functionality for solving the matrix
21 % equation for vector of updates, delta
22
23 % Hint 3:
24 % Check for root each iteration, continuing until the maximum
25 % number of iterations has been reached
26
27 function x = NewtonMultiVar(n, func, x0, h, nmax, tol)
28 % Initial array of root estimates, iteration and variable stores.
29 % Initialise a storage array
30 xstore = transpose(x0);
31 x = xstore; % Vector of the most recent root variables
32 n = 1;
33 % Set iterative loop for the function
34 while n < nmax
35     % Calculate the current root estimate, used a nested for
36     % loop.
37     for i = 1:length(func)
38         f(i,1) = func{i}(x); % Vector of variables passed
39         into the function call
40     end
41     % Initialise the size of the jacobian
42     jacob = zeros(length(func):length(x));
43     % Calculate the jacobian
44     for i = 1:length(func)
45         for j = 1: length(x)
46             % Create two small steps for the x values
47             x(j) = x(j) + h;
48             % Find the first part of the derivative
49             % calculation.
50             func1 = func{i}(x);
51         end
52     end
53     % Check for convergence
54     if norm(f) < tol
55         break;
56     end
57     % Update the root estimates
58     x = x - jacob \ f;
59     % Store the current root estimates
60     xstore = transpose(x);
61     % Set the current root estimates as the previous
62     % root estimates
63     x = xstore;
64     % Increment the iteration counter
65     n = n + 1;
66 end

```

```

42      % Do the second part of the derivative
43      % calculation.
44      x(j) = x(j) - 2.*h;
45      % Find the first part of the derivative
46      % calculation.
47      func2 = func{i}(x);
48      % Calculate the jacobian
49      jacob(i,j) = ((func1 - func2)./(2.*h));
50      % Correct the x value
51      x(j) = x(j) + h;
52      end
53  end
54      % Inverse the jacobian and get del
55      del = -1.* (jacob\f);
56      % Perform the necessary exit conditions
57      % New del
58      xnew = del + x;
59      % Recalculate f with xnew values
60      for i = 1:length(func)
61          fnew(i,1) = func{i}(transpose(xnew)); % Vector of
62          % variables passed into the function call
63      end
64      % Test for both convergence and function call close to
65      % zero.
66      if (prod((abs(fnew)<= tol)) == 1) && (prod((abs(del)<=
67      tol))== 1)
68          % Add to the store arrays
69          xstore = [xstore,xnew];
70          x = xstore;
71          return
72      end
73      xstore = [xstore,xnew];
74      x = xnew;
75      % Increase iteration counter
76      n = n + 1;
77  end
78      % Warn the user the maximum number of iterations have been
79      % performed
80      disp('The maximum number of iterations have been performed
81      without satisfying the required root finding condition');
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
845
846
847
847
848
849
849
850
851
852
853
854
855
856
856
857
858
858
859
859
860
861
862
863
864
865
865
866
867
867
868
869
869
870
871
872
873
873
874
875
875
876
877
877
878
879
879
880
881
882
883
884
885
885
886
887
887
888
889
889
890
891
892
893
893
894
895
895
896
897
897
898
899
899
900
901
902
903
903
904
905
905
906
907
907
908
909
909
910
911
911
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1630
1631
1631
1632
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638

```

2018

SEMESTER 2

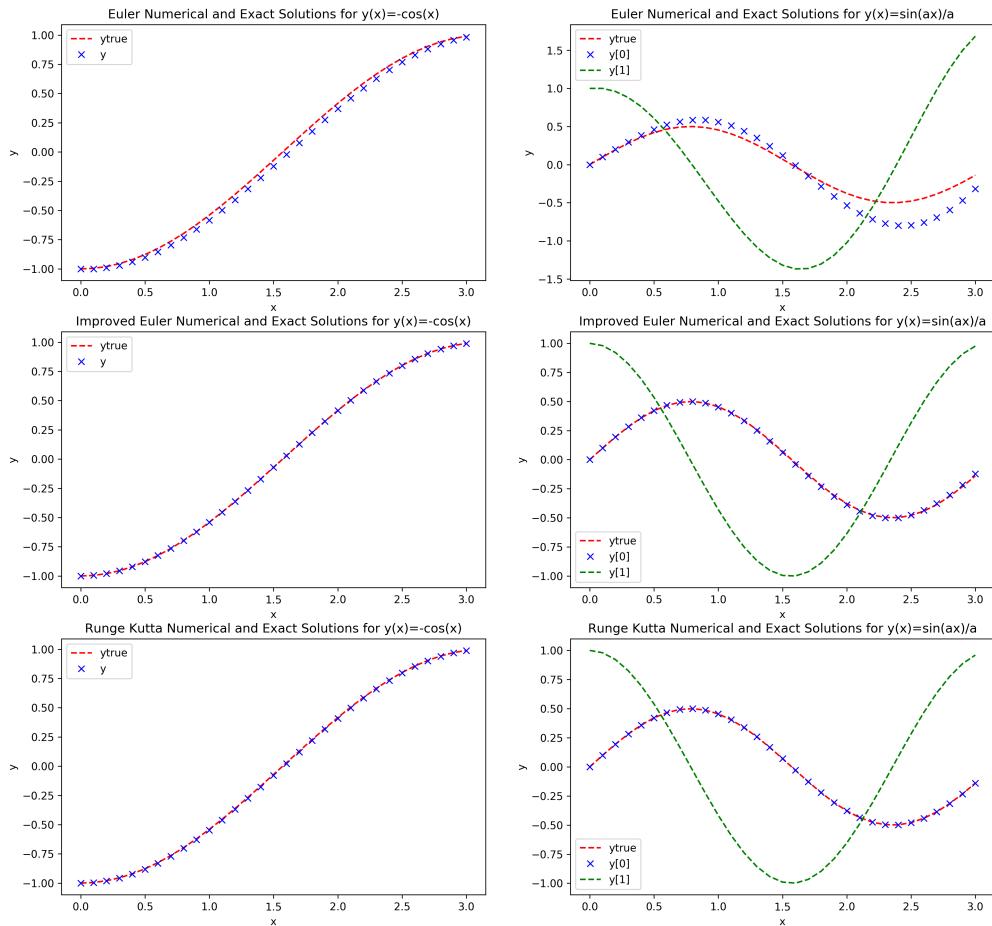
ENGCSI 331 Lab 2 ODEs

Connor McDowall
CMCD398 530913386

August 14, 2018

1 Improved Euler and Runge Kutta Solves

1.1 Hand in 1



```

def improved_euler_solve(f, x0, y0, x1, h, *args):
    ''' Compute solution to ODE using improved Euler method

    inputs
    -----
    f : callable
        derivative function which, for any input x and ya, yb, yc, ... values, returns a tuple of
        derivative values
    x0 : float
        initial value of independent variable
    y0 : a float, or a numpy array of floats
        array of initial values of solution variables (ya, yb, yc, ...)
    x1 : float
        final value of independent variable
    h : float
        step size
    *args : '*args'optional parameters
        optional parameters to pass to derivative function f()

    returns
    -----
    a list, xs, that gives each of the x values where the solution has been estimated
    a list of numpy arrays, where each array is an estimate of the solution (ya, yb, yc, ...)
    at the corresponding x value
    ...

    n = int(np.ceil((x1-x0)/h))          # number of Improved Euler steps to take
    xs = [x0+h*i for i in range(n+1)]    # x's we will evaluate function at
    ys = [y0]                            # list to store solution; we will append to this

    # iteration
    for k in range(n):
        ys.append( improved_euler_step(f, xs[k], ys[k], h, *args) )

    return xs, ys


def improved_euler_step(f, xk, yk, h, *args):
    ''' Compute a single improved Euler step.

    inputs
    -----
    f : callable
        derivative function
    xk : float
        independent variable at beginning of step
    yk : a float, or a numpy array of floats
        solution at beginning of step
    h : float
        step size
    *args : '*args' optional parameters
        optional parameters to pass to derivative function

    returns
    -----
    a float, or a numpy array of floats, giving solution at end of the step
    ...

    # Compute the improved euler solve to get the new co-ordinate point
    yeuler = yk + h*f(xk,yk,*args)
    # Compute the improved euler step
    return yk + 0.5*(h*f(xk,yk,*args) + h*f(xk+h, yeuler,*args))

```

```
# Runge Kutta Solve
def runge_kutta_solve(f, x0, y0, x1, h, *args):
    """ Compute solution to ODE using the clasical 4th order Runge Kutta method

    inputs
    -----
    f : callable
        derivative function which, for any input x and ya, yb, yc, ... values, returns a tuple of
        derivative values
    x0 : float
        initial value of independent variable
    y0 : a float, or a numpy array of floats
        array of initial values of solution variables (ya, yb, yc, ...)
    x1 : float
        final value of independent variable
    h : float
        step size
    *args : '*args' optional parameters
        optional parameters to pass to derivative function f()

    returns
    -----
    a list, xs, that gives each of the x values where the solution has been estimated
    a list of numpy arrays, where each array is an estimate of the solution (ya, yb, yc, ...)
    at the corresponding x value
    ...

n = int(np.ceil((x1-x0)/h))          # number of Runge Kutta steps to take
xs = [x0+h*i for i in range(n+1)]    # x's we will evaluate function at
ys = [y0]                            # list to store solution; we will append to this

# iteration
for k in range(n):
    ys.append( runge_kutta_step(f, xs[k], ys[k], h, *args) )

return xs, ys
```

```

# Runge Kutta
def runge_kutta_step(f, xk, yk, h, *args):
    """ Compute a single Runge Kutter step.

    inputs
    -----
    f : callable
        derivative function
    xk : float
        independent variable at beginning of step
    yk : a float, or a numpy array of floats
        solution at beginning of step
    h : float
        step size
    *args : '*args' optional parameters
        optional parameters to pass to derivative function

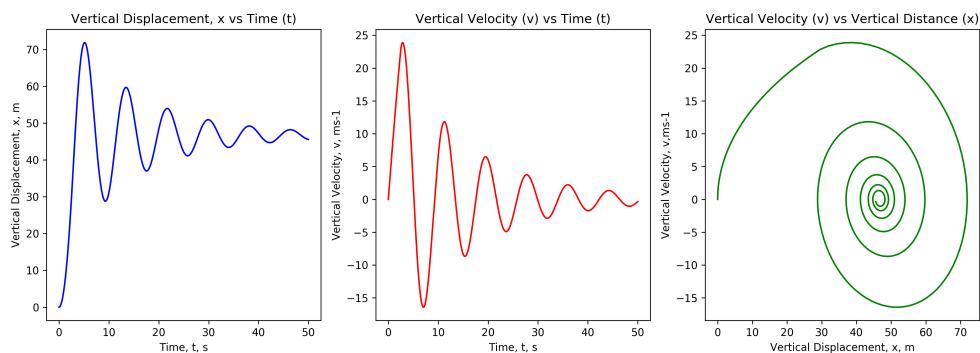
    returns
    -----
    a float, or a numpy array of floats, giving solution at end of the step
"""

    f0 = f(xk,yk,*args)
    f1 = f(xk + 0.5*h,yk + 0.5*h*f0,*args)
    f2 = f(xk + 0.5*h,yk + 0.5*h*f1,*args)
    f3 = f(xk + 0.5*h,yk + h*f2,*args)
    return yk + ((h/6)*(f0 + 2*f1 + 2*f2 + f3))

```

2 Bungee Jumper Derivative

2.1 Hand in 2



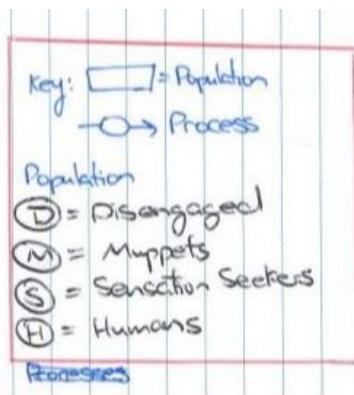
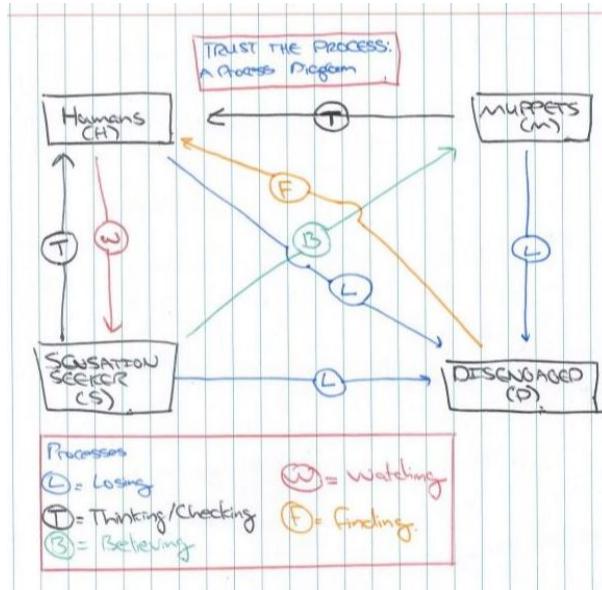
```

# Task 2
# Higher Order System: Bungy Jumper Model
def bungee_jumper_derivative(t,x,g,cd,m,k,L,gamma):
    if (x[0]<= L):
        result = np.array([x[1],g-np.sign(x[1])*(cd/m)*x[1]**2] )
        return result
    else:
        result = np.array([x[1],(g-np.sign(x[1])*(cd/m)*x[1]**2 - (k/m)*(x[0]-L) - gamma*x[1]/m)])
        return result

```

3 Fakes News

3.1 Hand in 3: Process Diagram



3.2 Hand in 4: Derivative Relationship

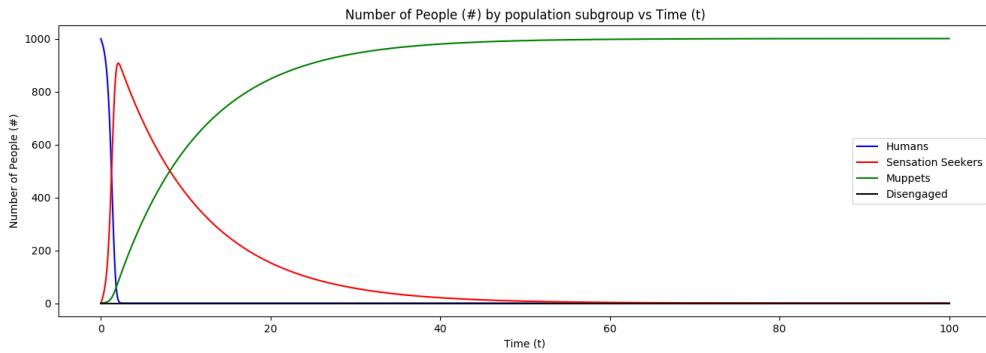
$$\frac{dH}{dt} = (S \times pt) - (H \times M \times pw) + (D \times pf) + (M \times pt) - (H \times pl) \quad (1)$$

$$\frac{dM}{dt} = (-M \times pt) - (M \times pl) + (S \times pb) \quad (2)$$

$$\frac{dS}{dt} = (-S \times pt) + (H \times M \times pw) - (S \times pl) - (S \times pb) \quad (3)$$

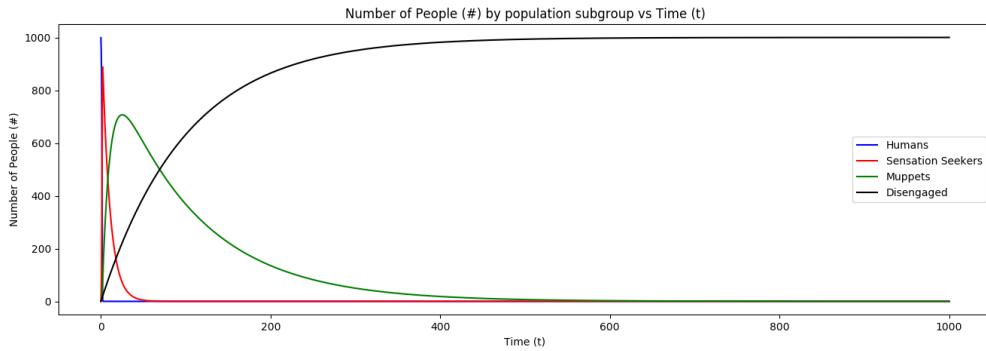
$$\frac{dD}{dt} = (S \times pl) + (H \times pl) + (M \times pl) - (D \times pf) \quad (4)$$

3.3 Hand in 5



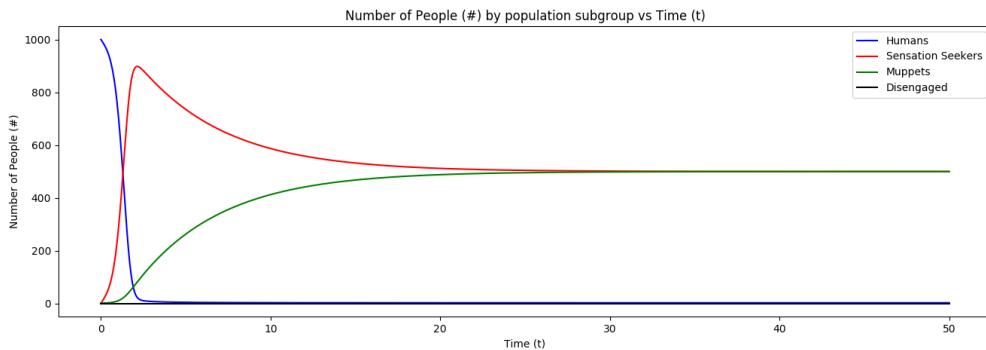
Humans will be rapidly converted to sensation seekers with sensation seekers also converted to muppets at the same time. Once there are no humans left, all sensation seekers will be convert to muppets, enough to occupy street for many years to come.

3.4 Hand in 6



Humans will be converted to sensation seekers, and then muppets. However, losing phones will cause all other parties to be disengaged, with no consuption of fake news. It will take a long time for everyone to lose their phones as the rate is quite small. This seems like a utopia as people will finally talk to eachother.

3.5 Hand in 7



With critical thinking and education, we will have an equal number of sensation seekers to muppets. A state of equilibrium. This is important as there will be a school of thought to contest ideas. Half the population won't believe the fake news but will still consume it as the rate of believing fake news will be the same as thinking about it. Critical thinking and education will continue to be very important.

4 Adaptive step-sizes: Orienteering Model

4.1 Hand in 10

```
PS C:\Users\Connor McDowall> cd 'c:\Users\Connor McDowall\Desktop\Lab 2\3e\extensions\ms-python.python-2018.7.1\pythonFiles\PythonTools\visualstude\putput' 'c:\Users\Connor McDowall\Desktop\Lab 2 331\ODETesterMain.py'  
Solving Instance 0  
Derivative Call Count=480, Tolerance=0.01, a=5, b=2, c=-0.1, d=20, e=3  
Score = 480 with 480 fn calls, maximum error of 0.000800459 & 0 penalties  
Score is 480.0
```

□

```
def adaptative_runge_kutta_solve(f, x0, y0, x1, h, tol, *args):
    ''' Compute solution to ODE using the clasical 4th order Runge Kutta method with a variable
        inputs
        -----
        f : callable
            derivative function which, for any input x and ya, yb, yc, ... values,
            returns a tuple of derivative values
        x0 : float
            initial value of independent variable
        y0 : a float, or a numpy array of floats
            array of initial values of solution variables (ya, yb, yc, ...)
        x1 : float
            final value of independent variable
        h : float
            step size
        *args : '*args' optional parameters
            optional parameters to pass to derivative function f()

        returns
        -----
        a list, xs, that gives each of the x values where the solution has been estimated
        a list of numpy arrays, where each array is an estimate of the solution (ya, yb, yc, ...)
        at the corresponding x value
    ...
    # Set up intial lists and values for the function
    xs = [x0]
    ys = [y0]
    xk = x0
    yk = y0
    hnew = h
    # iteration until back at the very end point.
    while xk < x1:
        h = min(x1 - xk, hnew)
        xs.append(xk)
        ys.append(yk)
        xk, yk, hnew = adaptive_runge_kutta_step(f, xk, yk, h, tol, *args)
        xk = xk + h
    return xs, ys
```

```

# Adaptive runge kutta step
def adaptive_runge_kutta_step(f, xk, yk, h, tol, *args):
    """ Compute a single Runge Kutter step that adapts the step size.

    inputs
    -----
    f : callable
        derivative function
    xk : float
        independent variable at beginning of step
    yk : a float, or a numpy array of floats
        solution at beginning of step
    h : float
        step size
    *args : '*args' optional parameters
        optional parameters to pass to derivative function

    returns
    -----
    a float, or a numpy array of floats, giving solution at end of the step
"""

# Calculate all the function values , 3rd and 4th Order Runge Kutta's
f0 = f(xk,yk,*args)
f1 = f(xk + 0.5*h,yk + 0.5*h*f0,*args)
f2_3rd = f(xk + h,yk - h*f0 + 2*h*f1,*args)
f2_4th = f(xk + 0.5*h,yk + 0.5*h*f1,*args)
f3 = f(xk + h,yk + h*f2_4th,*args)
third0 = yk + ((h/6)*(f0 + 4*f1 + f2_3rd))
fourth0 = yk + ((h/6)*(f0 + 2*f1 + 2*f2_4th + f3))
# Calculate the observed error
obs = abs(third0 - fourth0)
# Calculate the new h value
hnew = h*(np.abs(tol/obs)**0.2) if obs > 1e-12 else h
# 4th Order Return
return xk, fourth0, hnew

# Use my adaptive runge kutta method
def SolveODE_AdapativeStepping(f, x0, y0, x1, tol, a, b, c, d, e):
    h = 1.05
    x,y = adaptative_runge_kutta_solve(f, x0, y0, x1, h, tol, a, b, c, d, e)

    return (x,y)

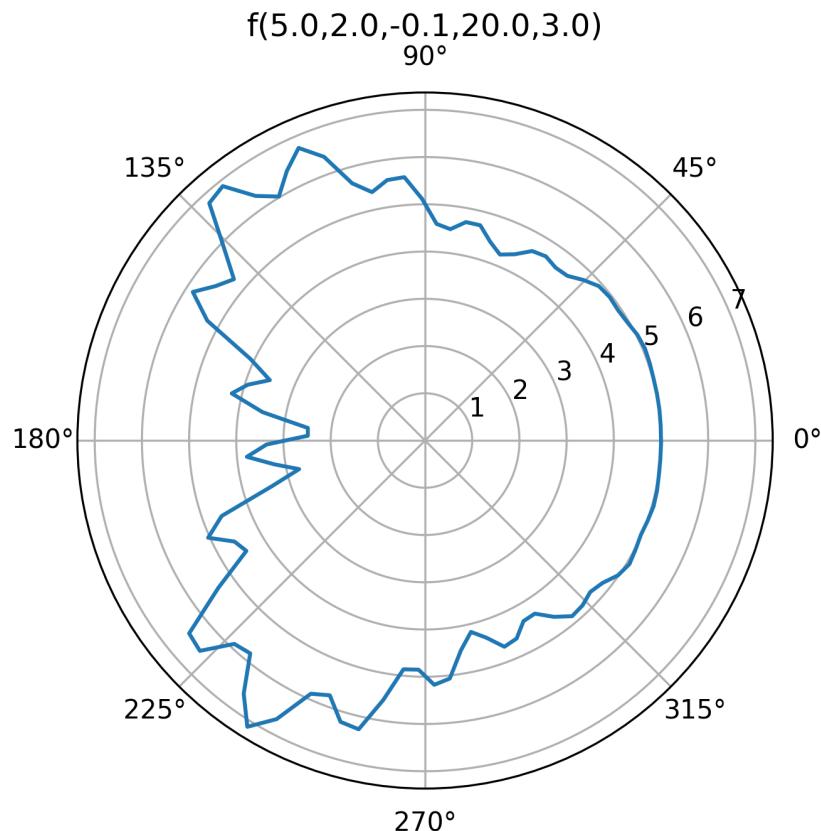
```

4.2 Hand in 11

I used error estimation using embedded runge kutta methods. In the adaptive runge kutta step function, all function evaluations are calculated for third and forth order techniques. An observed difference is calculated between the two function calls. The step sized is scaled by the absolute value of the target difference divided by observed difference, if greater than a machine

precision of 1e-12. The stepping function returns the new step size, y and x values.

4.3 Hand in 12



```
Evaluating ODE code for Instance 0
Derivative Call Count=480, Tolerance=0.01, a=5, b=2, c=-0.1, d=20, e=3
Score = 480 with 480 fn calls, maximum error of 0.000800459 & 0 penalties.
Evaluating ODE code for Instance 1
Derivative Call Count=480, Tolerance=0.01, a=4.1, b=2, c=-0.054, d=-20, e=5.2
Score = 480 with 480 fn calls, maximum error of 0.000921546 & 0 penalties.
Evaluating ODE code for Instance 2
Derivative Call Count=480, Tolerance=0.001, a=4.1, b=2, c=-0.054, d=-20, e=5.2
Score = 480 with 480 fn calls, maximum error of 0.000921546 & 0 penalties.
Evaluating ODE code for Instance 3
Derivative Call Count=480, Tolerance=0.001, a=4.1, b=2, c=-0.2, d=20, e=3
Score = 480 with 480 fn calls, maximum error of 0.000614508 & 0 penalties.
User = cmcd398: Total Score = 1920
Result cmcd398: 1920 submitted at Tue Aug 14 20:23:42 2018 [ <Response [200]> ]
PS C:\Users\Connor McDowall\Desktop\Lab 2 331> 
```

5 Appendix

Databases Assignment

Part 1

Question 2

Directions Table

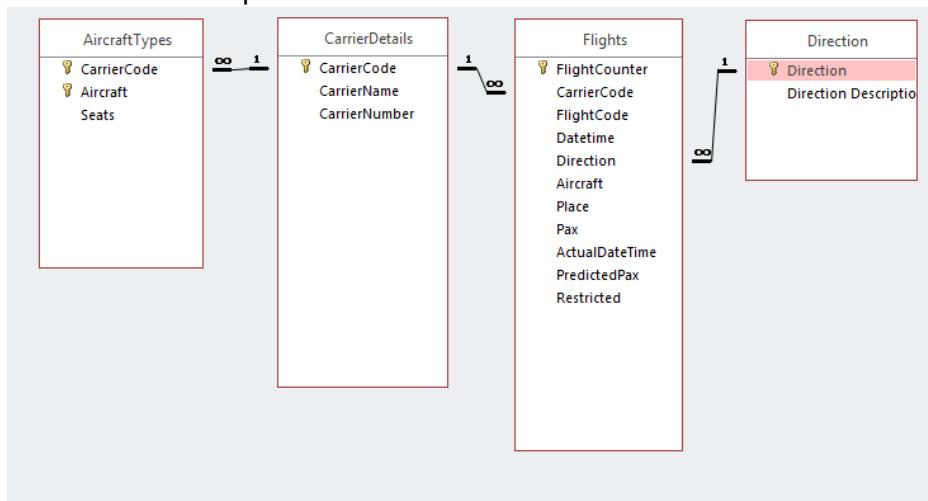
a. Design mode

Direction	Direction Description	Click to Add
A	Arrivals	
D	Departures	
*		

b. Database mode

Field Name	Data Type
Direction	Short Text
Direction Description	Short Text

c. New Relationship



Question 3

AirlineDailyTotals

d. Datasheet View

Date	CarrierCode	SumOfPax
1/02/1996		221
1/02/1996	BR	356
1/02/1996	BY	308
1/02/1996	FJ	137
1/02/1996	KE	57
1/02/1996	NZ	3104
1/02/1996	PP	295
1/02/1996	QF	716
1/02/1996	SQ	168
1/02/1996	UA	941
2/02/1996	CX	346
2/02/1996	FJ	276
2/02/1996	NZ	3135
2/02/1996	DU	257

e. SQL View

```
SELECT Int([DateTime]) AS [Date], Flights.CarrierCode, Sum(Flights.Pax) AS SumOfPax
FROM Flights
GROUP BY Int([DateTime]), Flights.CarrierCode
ORDER BY Int([DateTime]);
```

f. Design View

The screenshot shows the Microsoft Access Design View for a query. At the top, the SQL code is displayed:

```
SELECT Int([DateTime]) AS [Date], Flights.CarrierCode, Sum(Flights.Pax) AS SumOfPax
FROM Flights
GROUP BY Int([DateTime]), Flights.CarrierCode
ORDER BY Int([DateTime]);
```

Below the SQL, the 'Design View' interface is shown. It consists of two main sections: a 'Fields' list on the left and a 'Query Properties' grid on the right.

Fields List:

Flights
Aircraft
Place
Pax
ActualDateTime
PredictedPax
Restricted

The 'Pax' field is highlighted with a pink background.

Query Properties Grid:

Field:	Date : Int([DateTime])	CarrierCode	Sum
Table:	Flights	Flights	
Total:	Group By	Group By	
Sort:	Ascending		
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Criteria: or:			

Question 5

FlightSeats

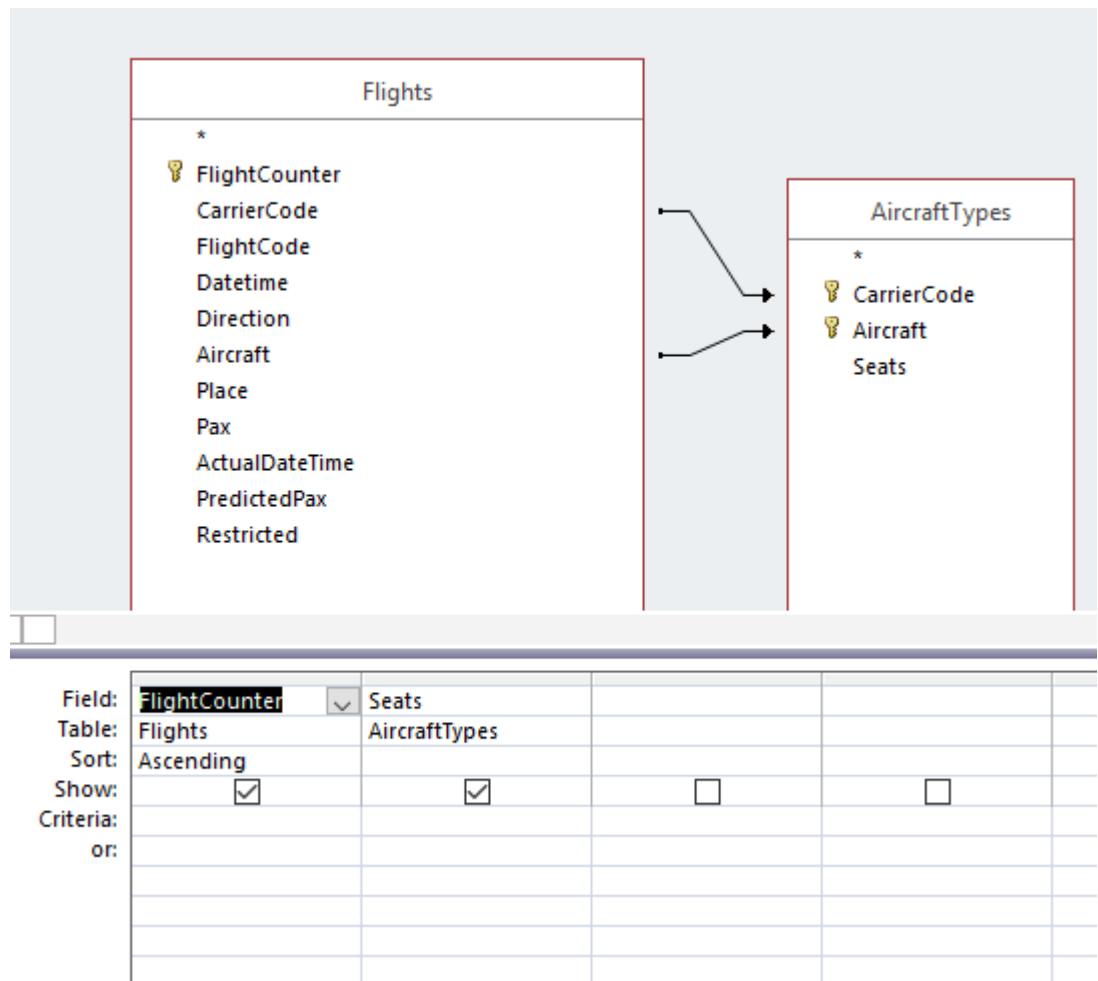
g. Database view

FlightCounter	Seats
21391	121
21392	359
21393	359
21394	250
21395	362
21396	431
21397	359
21398	394
21399	149
21400	361
21401	227
21402	431
21403	359
21404	236
21405	250
21406	136
21407	201
21408	121
21409	362

h. SQL view

```
SELECT Flights.FlightCounter, AircraftTypes.Seats  
FROM Flights LEFT JOIN AircraftTypes ON (Flights.Aircraft = AircraftTypes.Aircraft) AND (Flights.CarrierCode = AircraftTypes.CarrierCode)  
ORDER BY Flights.FlightCounter;
```

i. Design view



Question 6

FlightSeatUtilisation

j. Database view

FlightCounter	SeatUtilisati
21391	1.652892562
21392	0.5348189415
21393	0.2590529248
21394	0.124
21395	0.7679558011
21396	0.2157772622
21397	0.2590529248
21398	0.2131979695
21399	0.4697986577
21400	0.6149584488
21401	0.4713656388
21402	0.4454756381
21403	0.2646239554
21404	0.1906779661
21405	0.372
21406	0.7647058824
21407	0.7661601507

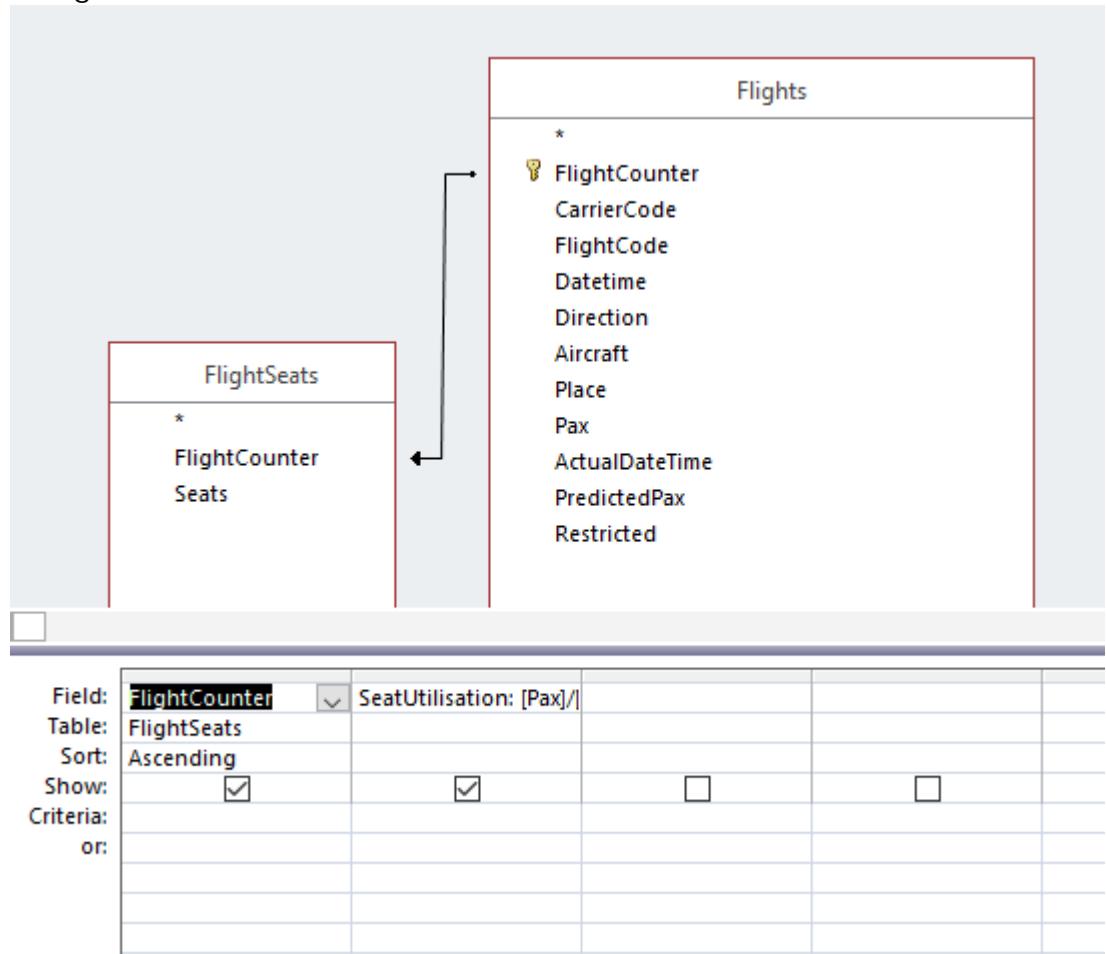
k. SQL view

```

SELECT FlightSeats.FlightCounter, [Pax]/[Seats] AS SeatUtilisation
FROM FlightSeats RIGHT JOIN Flights ON FlightSeats.FlightCounter = Flights.FlightCounter
ORDER BY FlightSeats.FlightCounter;

```

I. Design view



Question 7

CarrierUtilisation

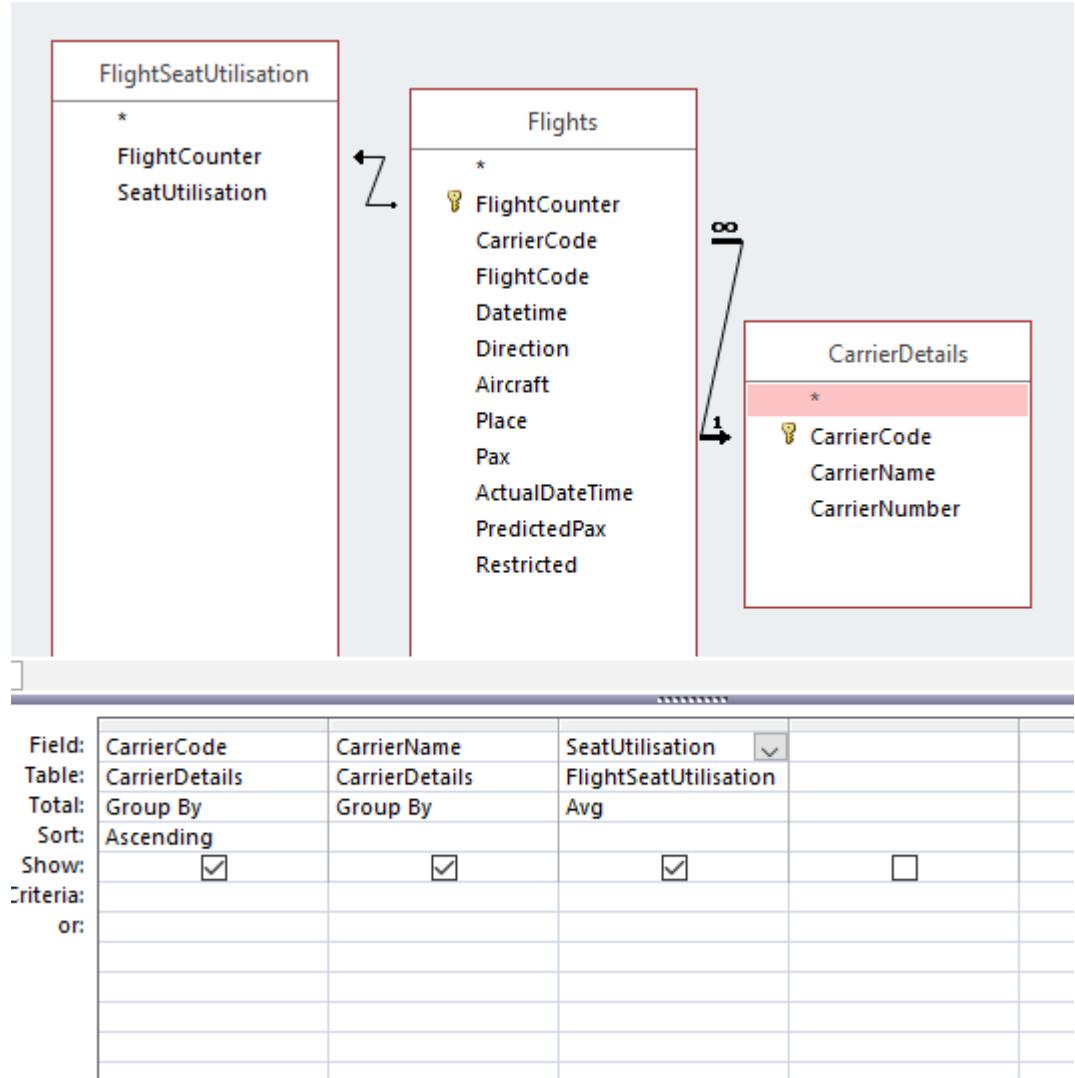
m. Database view

CarrierCode	CarrierName	AvgOfSeatU
AR	Aerolinas Argentin	0.4613801026
BR	EVA Air	0.432132964
BY	Britannia Airways	0.7661691542
CX	Cathay Pacific	0.7511061947
FJ	Air Pacific	0.3548267381
GA	Garuda	0.6057142857
IE	Solomon Is	0.2824427481
JL	Japan Airlines	0.7655172414
KE	Korean	0.4937953995
MH	Malaysian	0.4772117962
NF	Air Vanuatu	0.2415384615
NZ	Air New Zealand	0.3846097904
PH	Polynesian	0.6662946429
PP	Pacific Pandas	0.7487309645
QF	Qantas	0.4933232074
SB	Air Caledonie	0.3821950554
SJ	Freedom Airline	0.6899350649
SQ	Singapore	0.276831037
TG	Thai International	0.5612807464
UA	United Airlines	0.5662490134
WR	Royal Tongan	0.1881463803

n. SQL view

```
SELECT CarrierDetails.CarrierCode, CarrierDetails.CarrierName, Avg(FlightSeatUtilisation.SeatUtilisation) AS AvgOfSeatUtilisation
FROM CarrierDetails RIGHT JOIN (FlightSeatUtilisation RIGHT JOIN Flights ON FlightSeatUtilisation.FlightCounter =
Flights.FlightCounter) ON CarrierDetails.CarrierCode = Flights.CarrierCode
GROUP BY CarrierDetails.CarrierCode, CarrierDetails.CarrierName
ORDER BY CarrierDetails.CarrierCode
```

o. Design view



Part 2

- # Earliest year of first registration for using the minimum function


```
dataframe = pandas.read_sql_query('SELECT
MIN(FIRST_NZ_REGISTRATION_YEAR) AS EarliestYear FROM Fleet ',
connection)
dataframe
```

EarliestYear

0	1899
---	------

2. # Make, model and vehicle year of all the cars with a vehicle year earlier than 1900?

```
dataframe = pandas.read_sql_query('SELECT MAKE, MODEL,  
VEHICLE_YEAR FROM Fleet WHERE VEHICLE_YEAR < 1900 ', connection)  
dataframe
```

	MAKE	MODEL	VEHICLE_YEAR
0	FACTORY BUILT	STANLEY STEAMER	1899
1	FACTORY BUILT	AVELING & PORTER	1894
2	VETERAN	RANSOMES SIMS &	1899
3	CARAVAN	CARAVAN	1897
4	FACTORY BUILT	FOWLER	1892
5	MOBILE MACHINE	HILL&MOORE CHUKWAGON	1890
6	MCLAREN	DCC	1892
7	LOCOMOBILE	02	1899
8	YAMAHA	RAZZ	1898
9	NISSAN	PH02	1898
10	TRACTOR	FOWLER ENGINE	1898
11	DE DION-BOUTON	L 68	1898
12	FACTORY BUILT	BURRELL TRACTION ENG	1899
13	VETERAN	LOCOMOBILE	1899
14	CUSTOMBUILT	FOWLER	1896

3. # What are the 10 most popular (by count) car makes, and the counts of these?

```
dataframe = pandas.read_sql_query('SELECT MAKE, COUNT(MAKE) AS  
Count FROM Fleet GROUP BY MAKE ORDER BY Count DESC LIMIT 10',  
connection)  
dataframe
```

	MAKE	Count
0	TOYOTA	967765
1	NISSAN	491082
2	TRAILER	465880
3	MAZDA	347232
4	FORD	334040
5	HONDA	289657
6	MITSUBISHI	266473
7	HOLDEN	236895
8	SUZUKI	165627
9	SUBARU	132182

4. # What are the 20 most popular (by count) car models (where each (make, model) tuple counts as a different model), and the counts of these?

```
dataframe = pandas.read_sql_query('SELECT MAKE,MODEL,
COUNT(MODEL) AS Count FROM Fleet GROUP BY MAKE,MODEL ORDER BY
Count DESC LIMIT 20', connection)
dataframe
```

	MAKE	MODEL	Count
0	TOYOTA	COROLLA	170589
1	TOYOTA	HILUX	125273
2	HOLDEN	COMMODORE	86761
3	TOYOTA	HIACE	84895
4	SUZUKI	SWIFT	73171
5	FORD	FALCON	66504
6	TOYOTA	RAV4	62660
7	SUBARU	LEGACY	61038
8	TOYOTA	LANDCRUISER	49277
9	FORD	RANGER	49024
10	NISSAN	NAVARA	47799
11	TOYOTA	CAMRY	45312
12	TRAILER	HOMEBUILT	45102
13	HONDA	CIVIC	43851
14	MAZDA	DEMIO	43786
15	TRAILER	LOCAL	42656
16	VOLKSWAGEN	GOLF	42210
17	HONDA	ACCORD	42040
18	NISSAN	TIIDA	41343
19	MITSUBISHI	LANCER	40174

5. # How many cars are first registered in each of the most recent 50 years?

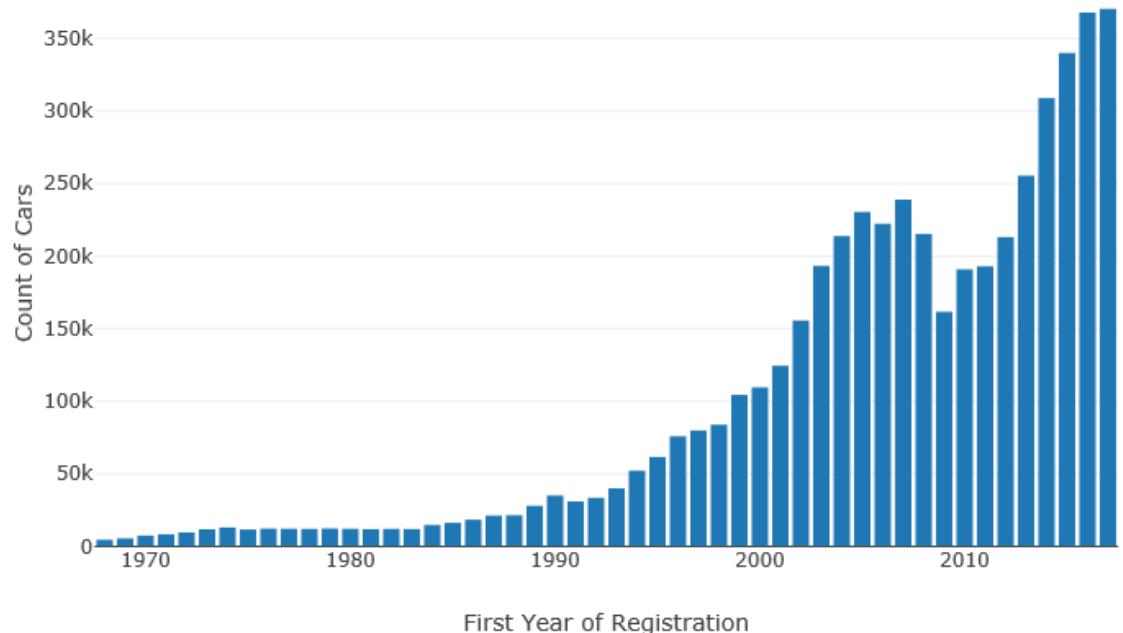
```
dataframe = pandas.read_sql_query('SELECT
FIRST_NZ_REGISTRATION_YEAR,
COUNT(FIRST_NZ_REGISTRATION_YEAR) AS Count FROM Fleet WHERE
FIRST_NZ_REGISTRATION_YEAR <> "" GROUP BY
FIRST_NZ_REGISTRATION_YEAR ORDER BY
FIRST_NZ_REGISTRATION_YEAR DESC LIMIT 50 ', connection)
dataframe
```

	FIRST_NZ_REGISTRATION_YEAR	Count
0	2017	370431
1	2016	367856
2	2015	340006
3	2014	308933
4	2013	255464
5	2012	213171
6	2011	193042
7	2010	190921
8	2009	161670
9	2008	215344
10	2007	238951
11	2006	222354
12	2005	230420
13	2004	213841
14	2003	193299

6. # Generate a plot with the previous answers

```
dataframeNew = pandas.read_sql_query('SELECT
FIRST_NZ_REGISTRATION_YEAR,
COUNT(FIRST_NZ_REGISTRATION_YEAR) AS Count FROM Fleet WHERE
FIRST_NZ_REGISTRATION_YEAR >= 1968 GROUP BY
FIRST_NZ_REGISTRATION_YEAR ORDER BY
FIRST_NZ_REGISTRATION_YEAR ASC ', connection)
trace =
plotly.graph_objs.Bar(x=dataframeNew.FIRST_NZ_REGISTRATION_YEAR,
y=dataframeNew.Count)
layout = plotly.graph_objs.Layout(title="Count of Cars vs First Year of
Registration",
xaxis=dict(title='First Year of Registration'),
yaxis=dict(title='Count of '))
fig = plotly.graph_objs.Figure(data=[trace], layout=layout)
plotly.offline.iplot(fig)
```

Count of Cars vs First Year of Registration



7. # How many Toyota cars were first registered in each year from 1950 onwards?

```
Toyota = pandas.read_sql_query('SELECT  
FIRST_NZ_REGISTRATION_YEAR, COUNT(MAKE) AS Toyotas FROM Fleet  
WHERE MAKE = "TOYOTA" AND FIRST_NZ_REGISTRATION_YEAR <> ""  
AND FIRST_NZ_REGISTRATION_YEAR > 1949 GROUP BY  
FIRST_NZ_REGISTRATION_YEAR ORDER BY  
FIRST_NZ_REGISTRATION_YEAR DESC', connection)  
Toyota  
# No registered toyotas in New Zealand before 1966.
```

FIRST_NZ_REGISTRATION_YEAR	Toyotas				
0	2017	72768	36	1981	794
1	2016	68532	37	1980	603
2	2015	61919	38	1979	490
3	2014	57738	39	1978	425
4	2013	49253	40	1977	333
5	2012	41409	41	1976	341
6	2011	38230	42	1975	278
7	2010	39824	43	1974	356
8	2009	31288	44	1973	220
9	2008	42825	45	1972	205
10	2007	49991	46	1971	131
11	2006	46622	47	1970	91
12	2005	50686	48	1969	28
13	2004	46482	49	1968	5
14	2003	44167	50	1967	8
15	2002	33583	51	1966	1
16	2001	24656			

8. # How many cars from Japan (ORIGINAL_COUNTRY="JAPAN") were first registered in each year from 1950 onwards?

```
Jap = pandas.read_sql_query('SELECT FIRST_NZ_REGISTRATION_YEAR,
COUNT(ORIGINAL_COUNTRY) AS JapanCars FROM Fleet WHERE
ORIGINAL_COUNTRY = "JAPAN" AND FIRST_NZ_REGISTRATION_YEAR
<> "" AND FIRST_NZ_REGISTRATION_YEAR > 1949 GROUP BY
FIRST_NZ_REGISTRATION_YEAR ORDER BY
FIRST_NZ_REGISTRATION_YEAR DESC', connection)
Jap
```

FIRST_NZ_REGISTRATION_YEAR	JapanCars
0	183069
1	175471
2	167140
3	153308
4	123021
5	103495
6	97141
7	102768
8	83153
9	113423
10	129588

9. # How many cars from Germany (ORIGINAL_COUNTRY="GERMANY") were first registered in each year from 1950 onwards?

```

Ger = pandas.read_sql_query('SELECT FIRST_NZ_REGISTRATION_YEAR,
COUNT(ORIGINAL_COUNTRY) AS GermanCars FROM Fleet WHERE
ORIGINAL_COUNTRY = "GERMANY" AND FIRST_NZ_REGISTRATION_YEAR
<> "" AND FIRST_NZ_REGISTRATION_YEAR > 1949 GROUP BY
FIRST_NZ_REGISTRATION_YEAR ORDER BY
FIRST_NZ_REGISTRATION_YEAR DESC', connection)
Ger

```

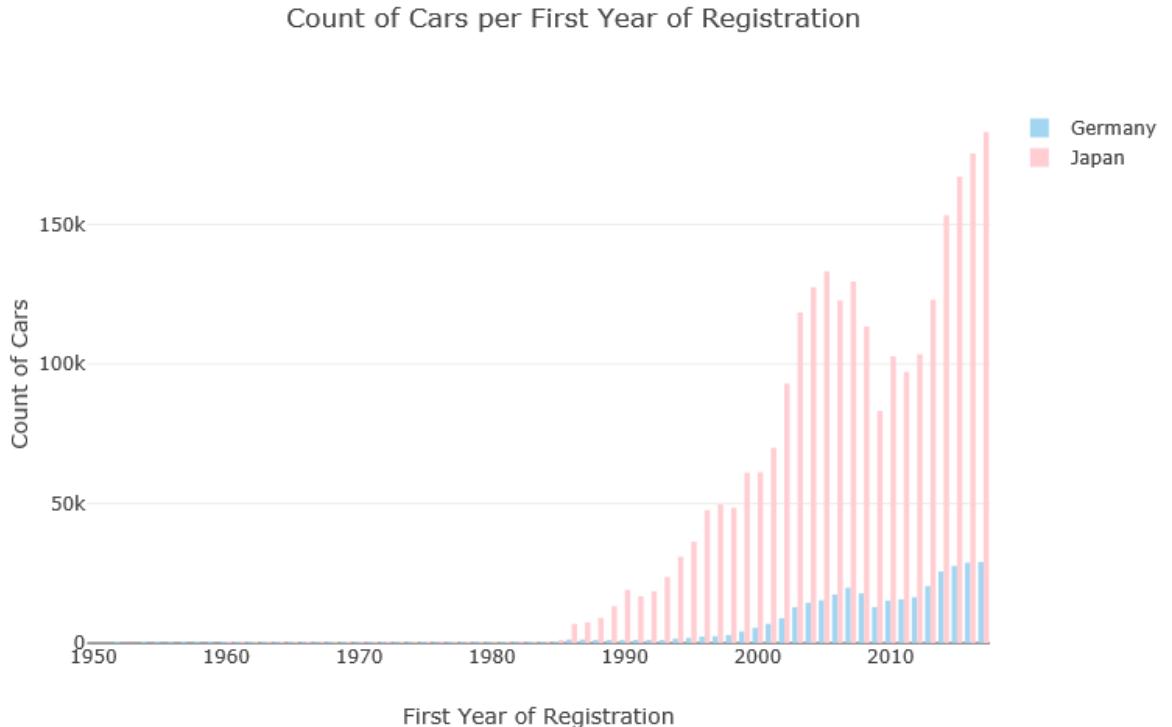
	FIRST_NZ_REGISTRATION_YEAR	GermanCars
0	2017	29059
1	2016	28784
2	2015	27621
3	2014	25684
4	2013	20431
5	2012	16422
6	2011	15674
7	2010	15216
8	2009	12896
9	2008	17825
10	2007	19826
11	2006	17412

```

10.# 10. Generate a labelled bar plot (with a legend) showing this first-
registered data for Japan and Germany
trace_germany =
plotly.graph_objs.Bar(x=Ger.FIRST_NZ_REGISTRATION_YEAR,y=Ger.Ger-
manCars,name = 'Germany',marker=dict(color='#A2D5F2'))
trace_japan =
plotly.graph_objs.Bar(x=Jap.FIRST_NZ_REGISTRATION_YEAR,y=Jap.Japa-
nCars,name = 'Japan',marker=dict(color='#ffcd2'))

layout = plotly.graph_objs.Layout(title="Count of Cars per First Year of
Registration",
        xaxis=dict(title='First Year of Registration'),
        yaxis=dict(title='Count of Cars'))
fig = plotly.graph_objs.Figure(data=[trace_germany,trace_japan],
layout=layout)
plotly.offline.iplot(fig)

```



11. # Create the scatter plot for the hybrids

```

# Find the list of all motive powers,
motive = pandas.read_sql_query('SELECT MOTIVE_POWER FROM Fleet
WHERE MOTIVE_POWER <> "DIESEL" AND MOTIVE_POWER <> "PETROL"
AND MOTIVE_POWER <> "" AND MOTIVE_POWER <> "CNG" AND
MOTIVE_POWER <> "LPG" AND MOTIVE_POWER <> "OTHER" GROUP BY
MOTIVE_POWER', connection)

# Initialise trace storage vector
traces = []
# For loop to run to create traces to append to a list of traces
for power in motive.MOTIVE_POWER:
    energy = pandas.read_sql_query('SELECT
FIRST_NZ_REGISTRATION_YEAR, COUNT(MOTIVE_POWER) AS Count
FROM Fleet WHERE MOTIVE_POWER = "{}" AND
FIRST_NZ_REGISTRATION_YEAR >=2000 GROUP BY
FIRST_NZ_REGISTRATION_YEAR ORDER BY
FIRST_NZ_REGISTRATION_YEAR ASC'.format(power), connection)
    # Create the plotting option
    energy_plot =
plotly.graph_objs.Scatter(x=energy.FIRST_NZ_REGISTRATION_YEAR,
                           y=energy.Count,
                           name = power)
    #Append to the traces for plotting
    traces.append(energy_plot);

layout = plotly.graph_objs.Layout(title="Number of Electric and Hybrid
Cars vs First Year of Registration",

```

```

xaxis=dict(title='First year of Registration'),
yaxis=dict(title='Number of Cars')

# Plot them all
fig = plotly.graph_objs.Figure(data=traces, layout=layout)
plotly.offline.iplot(fig)

```

