2018

SEMESTER 2

ENGSCI 331 Eigen Problems Assignment

Connor McDowall
530913386
cmcd398

October 23, 2018

Contents

Listings

List of Figures

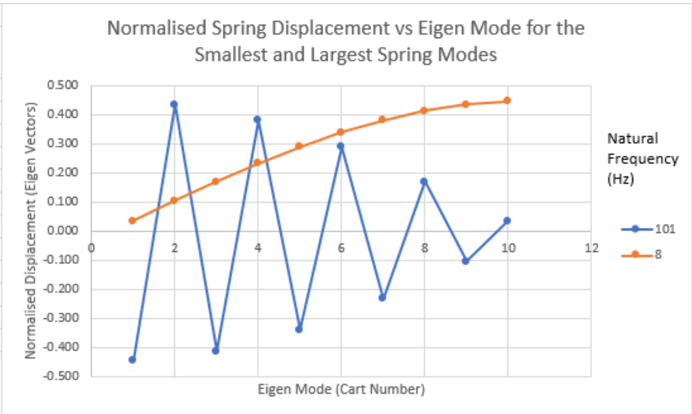
1 Report

Eigen Shift Method			Eigen Shift Normalised Eigen Vectors									
Size	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

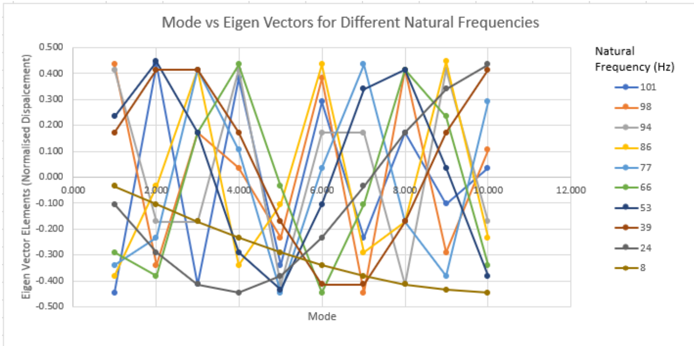
Figure 1: Eigen Shift Method Natural Frequencies and Eigen Vectors

Eigen All Method				Eigen All Normalised Eigen Vectors									
N	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



(a) Low vs High Natural Frequency (Hz)



(b) All 10 Modes

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 eachother. Using the analytical parameters, we observe this pattern in figures 3 (a) and (b).

The largest eigen value (highest natural frequency) has an ocsillating 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.

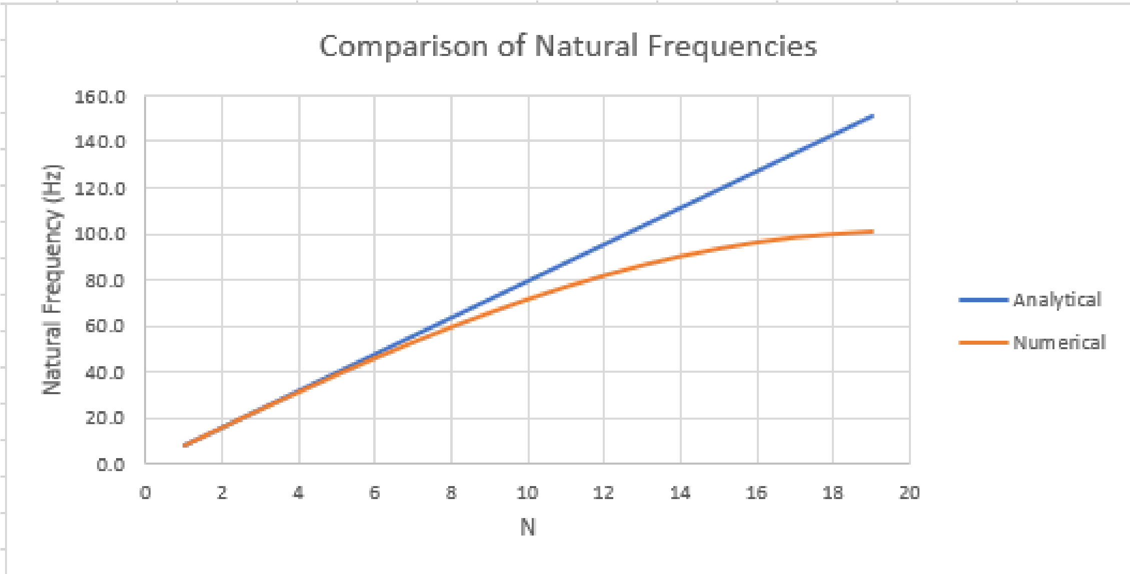


Figure 4: Numerical Solution compared against the Analytical

2.1 myEigenFunctions.h

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

        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 << ": \t";
    for (int i=0; i<length; i++)
      std::cout << vector[i] << "\t";
    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 1 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 << "Eigensmall␣Vector";
        std::cout << "\n";
        print_vector(b.vector, b.length);
        std::cout << "\n";
        std::cout << "Eigenlarge␣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 << " ";
                }
                std::cout << "\n";
        }
}
void print_vector(double *v, int n)
{
        for (int i = 0; i < n; i++)
        {
                std::cout << v[i];
                std::cout << " ";
        }
}
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;
```

```cpp
        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];
```

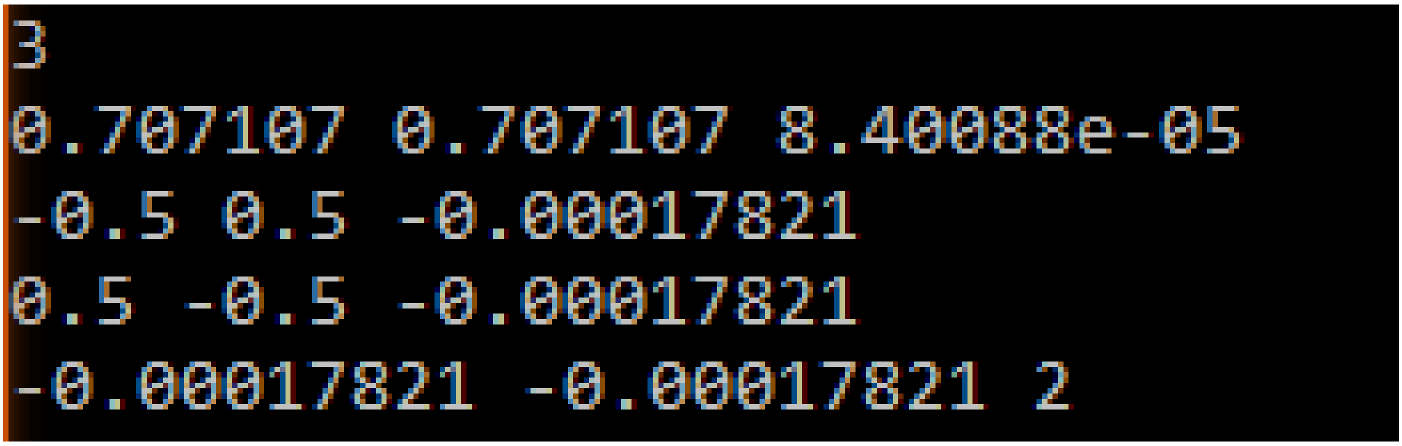
```
B = new double*[n];
for (int i = 0; i < n; i++)
        B[i] = new double[n];

// ----------------------------------------------------------------------
//
// 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
//
// ------------------------------------------------------------------------------------------
```

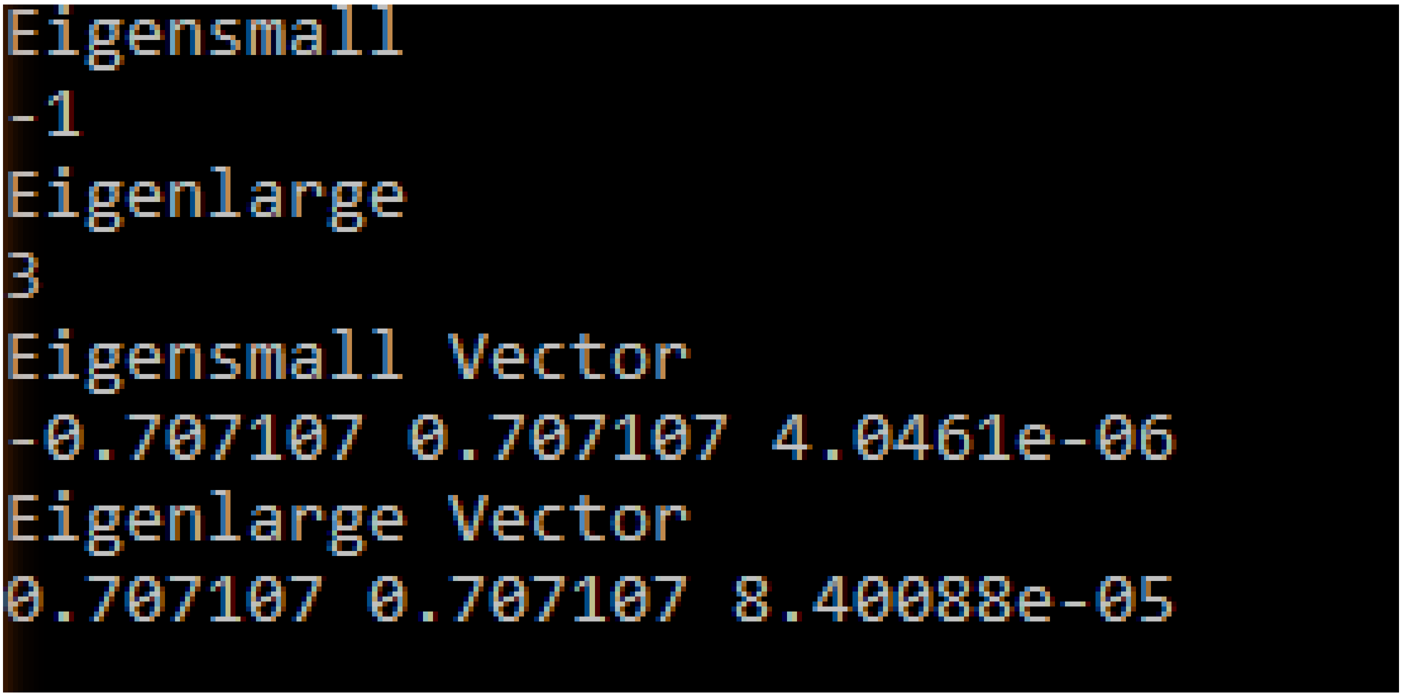
```
    // Use eigenshift to get the natural frequencies
    // and eigenvectors for the lowest and highest
    // calculated spring nodes
    eigenshift(A, x, length, 1e-8);
    // Reset the matrix
    for (int i = 0; i < length; i++)
    {
        for (int j = 0; j < length; j++)
        {
            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



```
Eigensmall
-1
Eigenlarge
3
Eigensmall Vector
-0.707107 0.707107 4.0461e-06
Eigenlarge 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

Listings

List of Figures

1 Part 1

1.1 Visualisations

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

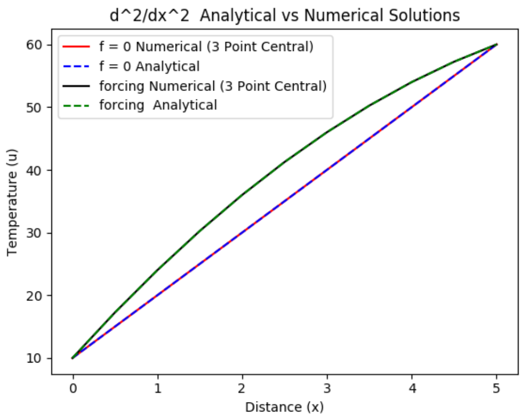


Figure 1: 1D $\frac{d^2u}{dx^2} = 0$: Numerical vs Analytical Comparison

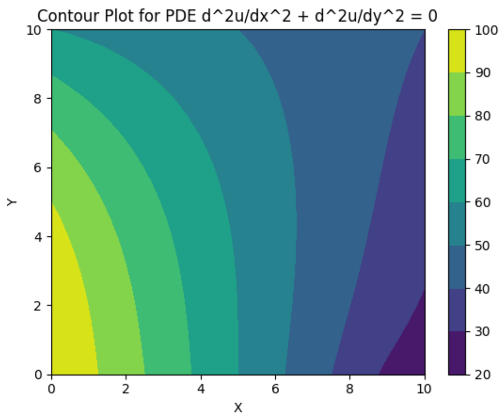


Figure 2: Contour plot for $\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = 0$

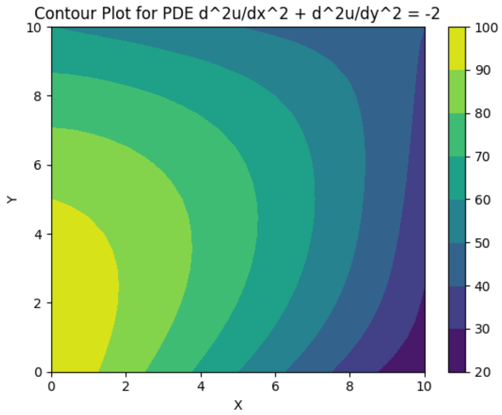


Figure 3: Contour plot for $\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = -2$

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 intepolation uses linear intepolation 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

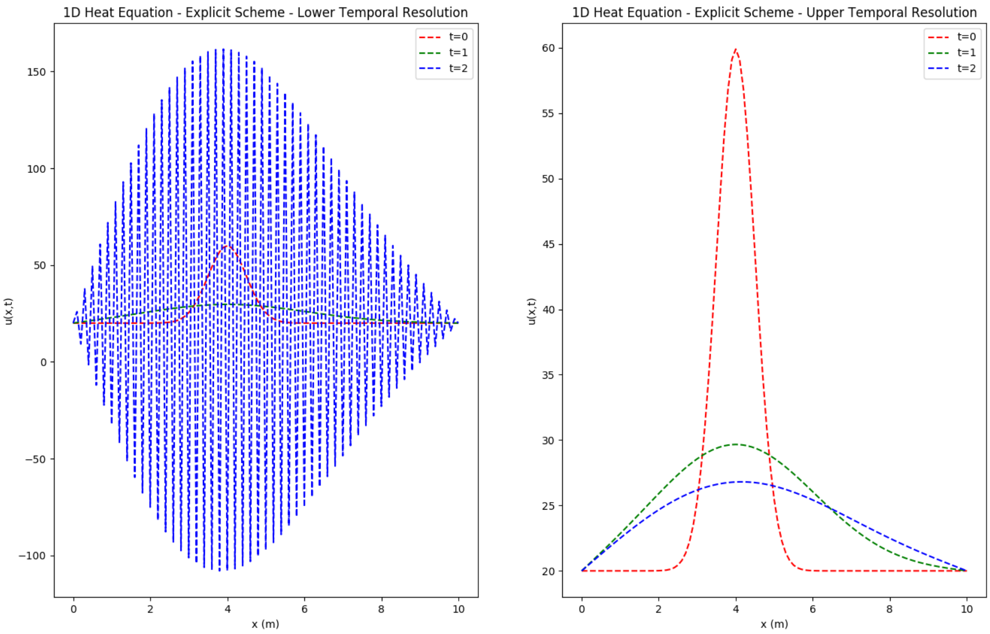


Figure 4: Explicit Scheme Upper and Lower Temporal Resolution Comparison for $\frac{\delta^2 u}{\delta x^2} - \alpha \frac{\delta u}{\delta t} = 0$

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

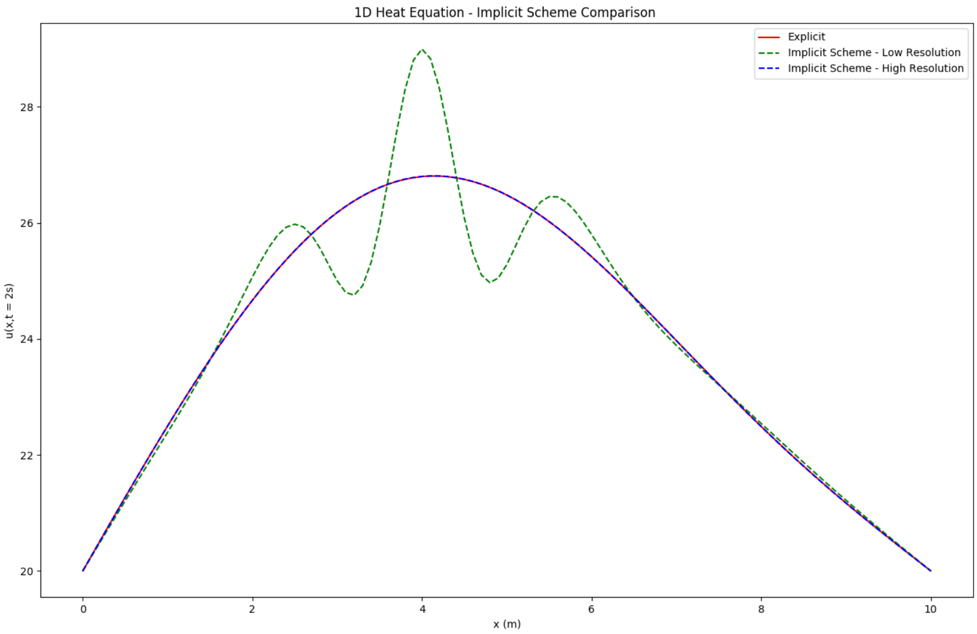


Figure 5: Scheme comparison for lower and upper resolutions for $\frac{\delta^2 u}{\delta x^2} - \alpha \frac{\delta u}{\delta t} = 0$

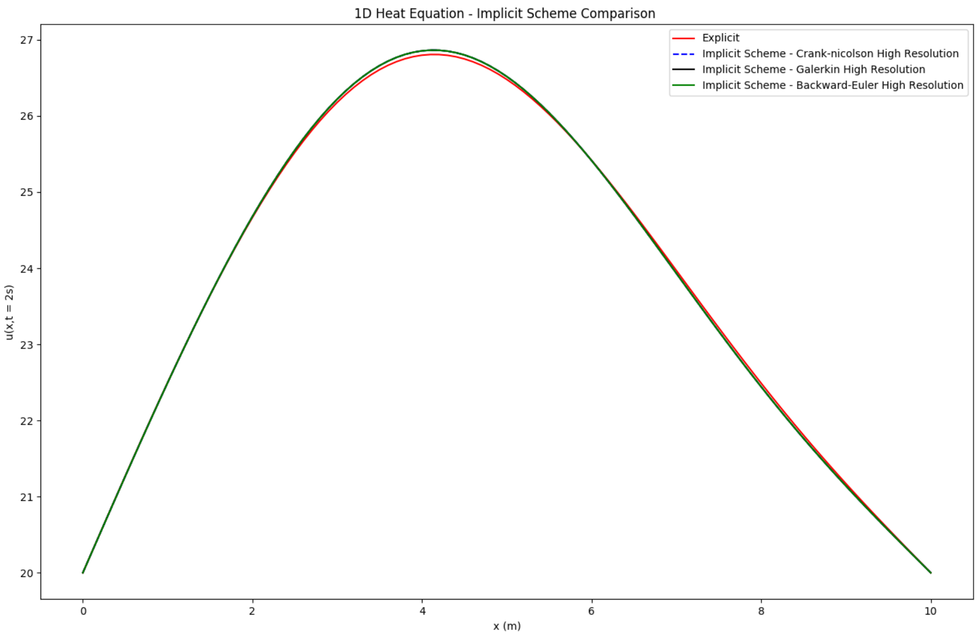


Figure 6: Scheme comparison for upper resolutions frt $\frac{\delta^2 u}{\delta x^2} - \alpha \frac{\delta u}{\delta t} = 0$

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 strenous as a system of simulataneous 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 accross all implicit methods (6). It is reasonable to assume, as shown with the explicit and imlpicit methods being apporximately 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

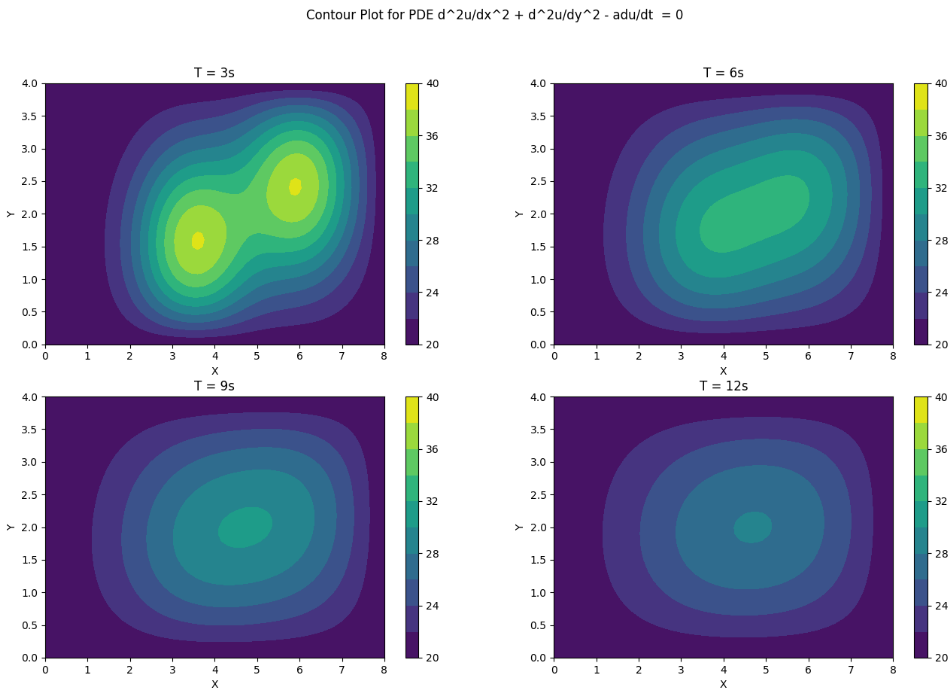


Figure 7: PDE for $\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} - \alpha \frac{\delta u}{\delta t} = 0$ where $\alpha = 10$

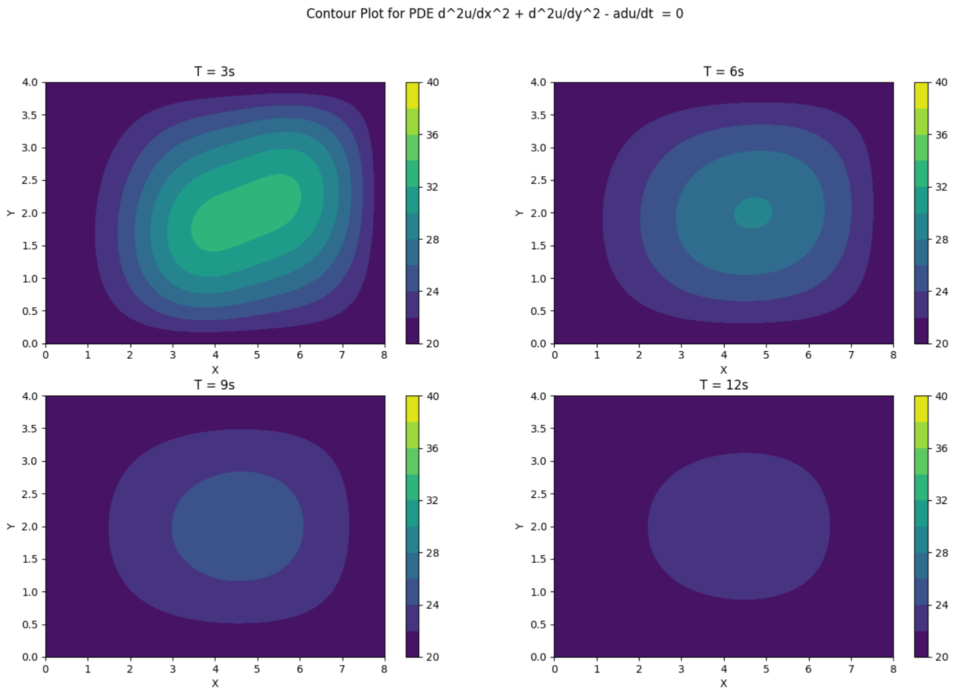


Figure 8: PDE for $\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} - \alpha \frac{\delta u}{\delta t} = 0$ where $\alpha = 5$

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

Listings

List of Figures

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 that 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.

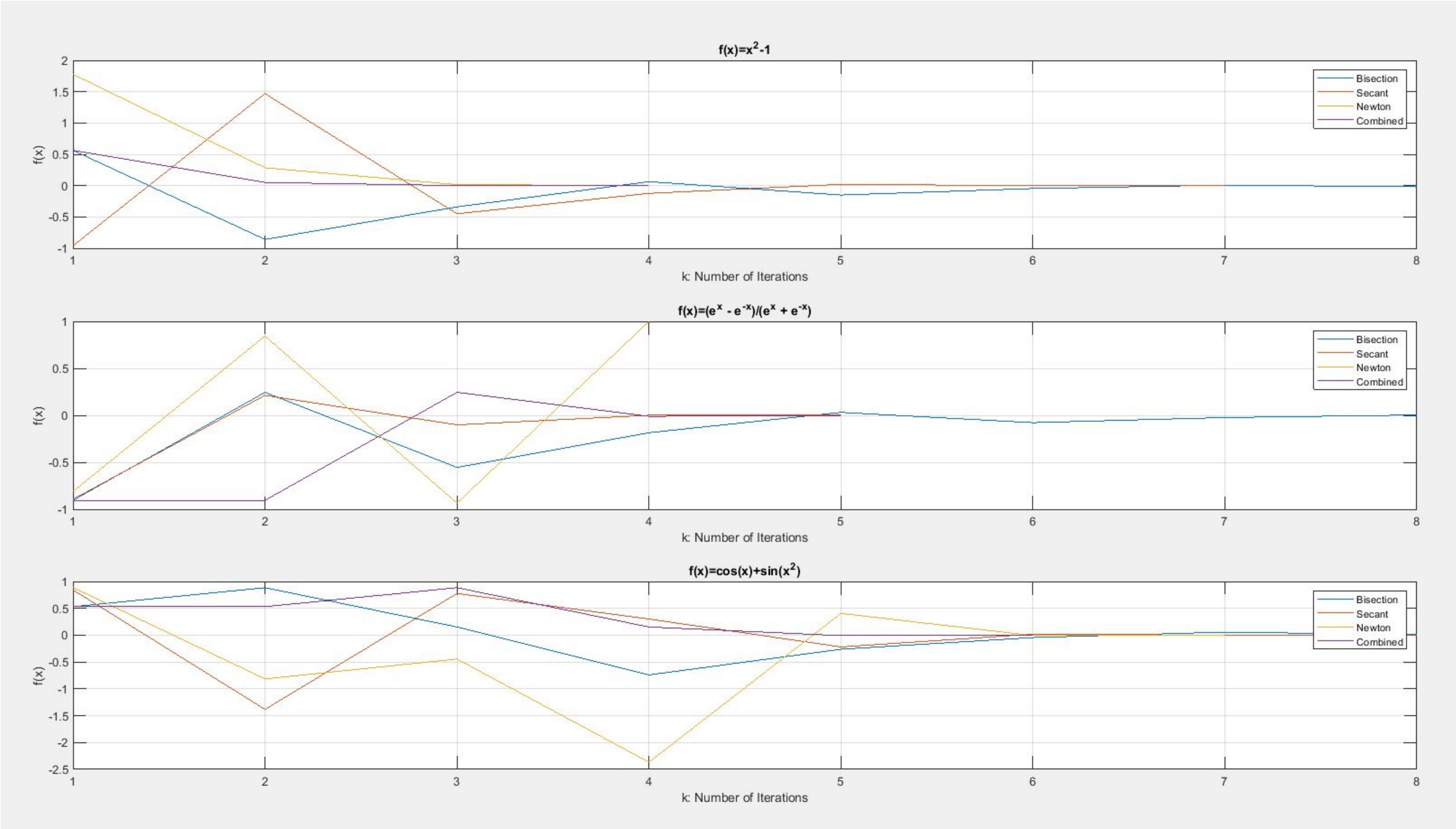


Figure 1: NLE Function Comparison

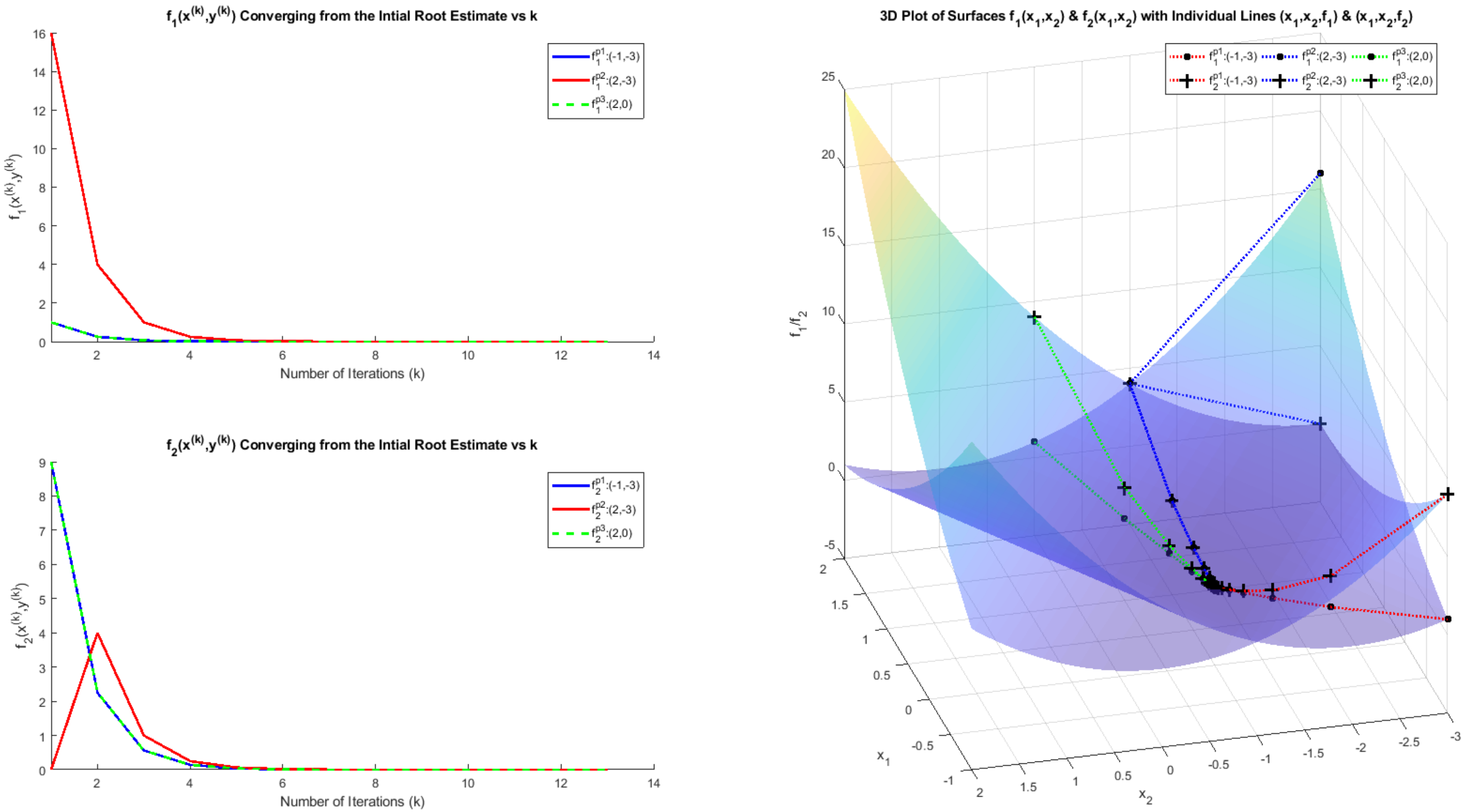


Figure 2: NLE Function Comparison

3 Code: Non Linear Equations

3.1 NLE Functions

Listing 1: Bisection

```
% Nonlinear equation root finding by the bisection method.
% Inputs
% f       : nonlinear function
% xl, xr  : initial root bracket
% nmax    : maximum number of iterations performed
% tol     : numerical tolerance used to check for root
% Outputs
% x       : one-dimensional array containing estimates of root

% Hint 1:
% Iterate until either a root has been found or maximum number of
    iterations has been reached

% Hint 2:
% Check for root each iteration, making use of tol

% Hint 3:
% Update the bracket each iteration

function x = Bisection(f, xl, xr, nmax, tol)
    % Initial array of root estimates, iteration and variable
       stores.
    x = [xl,xr];
    xbrac = x;
    n = 1;
    % Set iterative loop for the function
    while n < nmax + 1
        % Calculate the new root
        xnew = xbrac(1) + ((xbrac(2) - xbrac(1))/2);
        % Append the root estimate to the array
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) <= tol
           return
        % Calculate sign on the function with the new root
        % and find the new root bracket.
        elseif (f(xnew)*f(xbrac(1)))> 0
            % Reset the bracket with new LHS
            xbrac = [xnew,xbrac(2)];
        elseif (f(xnew)*f(xbrac(2)))> 0
            % Reset the bracket with new RHS
            xbrac = [xbrac(1),xnew];
        end
        % Increase iteration counter
        n = n + 1;
    end
```

```
    % Warn the user the maximum number of iterations have been
       performed
    disp('The maximum number of iterations have been performed
       without satisfying the required root finding condition');

end
```

Listing 2: Secant

```
% Nonlinear equation root finding by the secant method.
% Inputs
% f       : nonlinear function
% x0, x1  : initial root bracket
% nmax    : maximum number of iterations performed
% tol     : numerical tolerance used to check for root
% Outputs
% x       : one-dimensional array containing estimates of root

function x = Secant(f, x0, x1, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    x = [x0,x1];
    n = 1;
    k = 2;
    % Set iterative loop for the function
    while n <= nmax
        % Calculate the new root
        xnew = x(k) - (f(x(k))*(x(k)-x(k-1))/(f(x(k))- f(x(k-1)))
           );
        % Append the root estimate to the array
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) < tol
           return
        end
        % Increase iteration counter and k value
        n = n + 1;
        k = k + 1;
    end
    % Warn the user the maximum number of iterations have been
       performed
    disp('The maximum number of iterations have been performed
       without satisfying the required root finding condition');
```

Listing 3: Regula Falsi

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

```
% x       : one-dimensional array containing estimates of root

function x = Regulafalsi(f, xl, xr, nmax, tol)
 % Initial array of root estimates, iteration and variable stores.
    x = [xl,xr];
    xbrac = x;
    n = 1;
    % Set iterative loop for the function
    while n <= nmax
        % Calculate the new root
        xnew = xbrac(2) - (f(xbrac(2))*(xbrac(2)-xbrac(1))/(f(xbrac(2))- f(xbrac(1))));
        % Append the root estimate to the array
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) < tol
            return
        % Calculate sign on the function with the new root
        % and find the new root bracket.
        elseif f(xnew)*f(xbrac(1))>0
            % Reset the bracket with new LHS
            xbrac = [xnew,xbrac(2)];
        elseif f(xnew)*f(xbrac(2))>0
            % Reset the bracket with new RHS
            xbrac = [xbrac(1),xnew];
        end
        % Increase iteration counter
        n = n + 1;
    end
    % Warn the user the maximum number of iterations have been performed
    disp('The maximum number of iterations have been performed without satisfying the required root finding condition');
```

Listing 4: Newton

```
% Nonlinear equation root finding by Newton's method
% Inputs
% f       : nonlinear function
% x0      : initial root estimate
% h       : step size for central difference formula
% nmax    : maximum number of iterations performed
% tol     : numerical tolerance used to check for root
% Outputs
% x       : one-dimensional array containing estimates of root

function x = Newton(f, x0, h, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    x = x0;
    n = 1;
    k = 1;
```

```
    % Set iterative loop for the function
    while n < nmax
        % Calculate the new root
        xnew = x(k) - (f(x(k)))/ ((f(x(k)+h) - f(x(k)-h))/(2*h));
        % Append the root estimate to the array
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) < tol
           return
        end
        % Increase iteration counter and k value
        n = n + 1;
        k = k + 1;
    end
    % Warn the user the maximum number of iterations have been
       performed
    disp('The maximum number of iterations have been performed
       without satisfying the required root finding condition');
```

Listing 5: Combined

```
% Nonlinear equation root finding by the combined binsection/
   Newton's method
% Inputs
% f       : nonlinear function
% xl, xr  : initial root bracket
% h       : step size for central difference formula
% nmax    : maximum number of iterations performed
% tol     : numerical tolerance used to check for root
% Outputs
% x       : one-dimensional array containing estimates of root

function x = Combined(f, xl, xr, h, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    x = [xl,xr];
    xbrac = x;
    n = 1;
    % Calculate the starting estimate
    xstart = xbrac(1) + (xbrac(2) - xbrac(1))/2;
    % Append the starting value
    x = [x,xstart];
    k = 3;
    % Set iterative loop for the function
    while n < nmax
        % Use newton method to calculate the new root estimate
        xnew = x(k) - (f(x(k)))/ ((f(x(k)+h) - f(x(k)-h))/(2*h));
        % Use if condition to check inside the bracket
        if (xnew < xbrac(1)) || (xnew > xbrac(2))
            % Use the bisection method to get a better estimate
            xnew = xbrac(1) + (xbrac(2) - xbrac(1))/2;
        end
        % Append the root estimate to the array
```

```
        x = [x,xnew];
        % Terminate function if at derivitive point
        if abs(f(xnew)) < tol
           return
        % Calculate sign on the function with the new root
        % and find the new root bracket.
        elseif f(xnew)*f(xbrac(1))>0
            % Reset the bracket with new LHS
            xbrac = [xnew,xbrac(2)];
        elseif f(xnew)*f(xbrac(2))>0
            % Reset the bracket with new RHS
            xbrac = [xbrac(1),xnew];
        end
        % Increase iteration counter and k count by one.
        n = n + 1;
        k = k + 1;
    end
    % Warn the user the maximum number of iterations have been
       performed
    disp('The maximum number of iterations have been performed
       without satisfying the required root finding condition');
```

Listing 6: Task 1

```
%% Task 1 - Bisection, Secant, Regula Falsi and Newton's Methods
% You do NOT need to modify this script

% clear workspace
clear
clc

% Initialisation
f = @(x) 2*x.^2-8*x+4; % function to evaluate
tol = 1.0e-4;           % tolerance for asserts
h = 1.0e-4;             % step size for numerical estimate of
   gradient
x0 = 0.0;               % initial interval left
x1 = 2.0;               % initial interval right
nmax = 50;              % maximum number of iterations

% Bisection method
xb = Bisection(f, x0, x1, nmax, tol);
assert(abs(f(xb(end))) < tol)
disp(['Bisection converged to root at x = ' num2str(xb(end))]);

% Secant method
xs = Secant(f, x0, x1, nmax, tol);
assert(abs(f(xs(end))) < tol)
disp(['Secant converged to root at x = ' num2str(xs(end))]);

% Regula Falsi method and verification
xrf = Regulafalsi(f, x0, x1, nmax, tol);
```

```
assert(abs(f(xrf(end))) < tol)
disp(['Regula Falsi converged to root at x = ' num2str(xrf(end))]);

% Newton's method and verification
xn = Newton(f, x0, h, nmax, tol);
assert(abs(f(xn(end))) < tol)
disp(['Newton converged to root at x = ' num2str(xn(end))]);

% Combined Bisection/Newton's method and verification
xc = Combined(f, x0, x1, h, nmax, tol);
assert(abs(f(xc(end))) < tol)
disp(['Combined Bisection/Newton converged to root at x = ' num2str(xc(end))]);
```

3.2 NLE Plotting

Listing 7: Task 2

```
%% Task 2 - Iterative Algorithm Comparison

% clear workspace
clear
clc

% Initialisation
tol = 1.0e-4;  % tolerance for asserts
h = 1.0e-4;    % step size for numerical estimate of gradient
nmax = 20;     % maximum number of iterations

% functions to test algorithms on
f1 = @(x) x.^2 - 1;                              % function 1
f2 = @(x) (exp(x)-exp(-x))./(exp(x)+exp(-x)); % function 2
f3 = @(x) cos(x)+sin(x.*x)-0.5;                  % function 3

% initial root estimates for each function
% column 1: x0 for bisection, secant, regula falsi and combined methods
% column 2: x1 for bisection, secant, regula falsi and combined methods
% column 3: x0 for Newton's method
xint1 = ([-3.0,0.5,-3.0]);
xint2 = ([-5.,2.,1.1]);
xint3 = ([-2.0,1.5,-0.40]);

% function titles for plots
title1 = 'f(x)=x^2-1';
title2 = 'f(x)=(e^x - e^{-x})/(e^x + e^{-x})';
title3 = 'f(x)=cos(x)+sin(x^2)';

```

```
% set disp_func = false when you don't need to produce plot of functions
disp_func = false;
if disp_func
        x = linspace(-5.,5.,1000);
    figure(1), clf
    subplot(3,1,1)
    plot(x,f1(x))
    grid on, xlabel('x'), ylabel('f(x)'), title(title1)
    subplot(3,1,2)
    plot(x,f2(x))
    grid on, xlabel('x'), ylabel('f(x)'), title(title2)
    subplot(3,1,3)
    plot(x,f3(x))
    grid on, xlabel('x'), ylabel('f(x)'), title(title3)
end


%% find one root for each function using bisection, secant, newton's and combined methods
% Function 1
xB1 = Bisection(f1, xint1(1), xint1(2), nmax, tol);
xS1 = Secant(f1, xint1(1), xint1(2), nmax, tol);
xR1 = Regulafalsi(f1, xint1(1), xint1(2), nmax, tol);
xN1 = Newton(f1, xint1(3), h, nmax, tol);
xC1 = Combined(f1, xint1(1), xint1(2), h, nmax, tol);

% Function 2
xB2 = Bisection(f2, xint2(1), xint2(2), nmax, tol);
xS2 = Secant(f2, xint2(1), xint2(2), nmax, tol);
xR2 = Regulafalsi(f2, xint2(1), xint2(2), nmax, tol);
xN2 = Newton(f2, xint2(3), h, nmax, tol);
xC2 = Combined(f2, xint2(1), xint2(2), h, nmax, tol);

% Function 3
xB3 = Bisection(f3, xint3(1), xint3(2), nmax, tol);
xS3 = Secant(f3, xint3(1), xint3(2), nmax, tol);
xR3 = Regulafalsi(f3, xint3(1), xint3(2), nmax, tol);
xN3 = Newton(f3, xint3(3), h, nmax, tol);
xC3 = Combined(f3, xint3(1), xint3(2), h, nmax, tol);

%% individual plot for each function of f(x^k) vs k for each method
% i.e. each of the three plots (one per function) will have four lines, one for each method called.
figure(1), clf

% create top plot for function 1
% The intial root estimates have been excluded from the iterations.
subplot(3,1,1)
```

```
plot(1:length(xB1)-2,f1(xB1(3:length(xB1))))
hold on
plot(1:length(xS1)-2,f1(xS1(3:length(xS1))))
hold on
plot(1:length(xN1)-1,f1(xN1(2:length(xN1))))
hold on
plot(1:length(xC1)-2,f1(xC1(3:length(xC1))))
legend('Bisection','Secant','Newton','Combined')
xlim([1,8]);
grid on, xlabel('k: Number of Iterations'), ylabel('f(x)'), title
    (title1)

% create middle plot for function 2
subplot(3,1,2)
plot(1:length(xB2)-2,f2(xB2(3:length(xB2))))
hold on
plot(1:length(xS2)-2,f2(xS2(3:length(xS2))))
hold on
plot(1:length(xN2)-1,f2(xN2(2:length(xN2))))
hold on
plot(1:length(xC2)-2,f2(xC2(3:length(xC2))))
legend('Bisection','Secant','Newton','Combined')
xlim([1,8]);
grid on, xlabel('k: Number of Iterations'), ylabel('f(x)'), title
    (title2)

% create bottom plot for function 3
subplot(3,1,3)
plot(1:length(xB3)-2,f3(xB3(3:length(xB3))))
hold on
plot(1:length(xS3)-2,f3(xS3(3:length(xS3))))
hold on
plot(1:length(xN3)-1,f3(xN3(2:length(xN3))))
hold on
plot(1:length(xC3)-2,f3(xC3(3:length(xC3))))
legend('Bisection','Secant','Newton','Combined')
grid on, xlabel('k: Number of Iterations'), ylabel('f(x)'), title
    (title3)
xlim([1,8]);

% Save the plot
savefig('Task2Plot')
```

4 Systems of Non Linear Equations

4.1 Newton Two Variable

Listing 8: Newton Two Variables

```
% Nonlinear equation root finding in two dimensions using Newton's Method.
% Inputs
% func   : array of function handles for system of nonlinear equations
% x0     : vector of initial root estimates for each independent variable
% h      : step size for numerical estimate of partial derivatives
% nmax   : maximum number of iterations performed
% tol    : numerical tolerance used to check for root
% Outputs
% x      : two-dimensional array (two-row matrix) containing estimates of root

% Hint 1:
% Include the initial root estimate as the first column of x

% Hint 2:
% Use MATLAB in-built functionality for solving the matrix equation for vector of updates, delta

% Hint 3:
% Check for root each iteration, continuing until the maximum number of iterations has been reached

function x = Newton2Var(func, x0, h, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    % Initialise a storage array
    xstore = transpose(x0);
    x = xstore; % Vector of the most recent root variables
    % Set iterative loop for the function
    for i = 1:nmax
        % Calculate the function variables from the most recent iteration.
        f1 = func{1}(x);
        f2 = func{2}(x);

        %Calculate the derivatives for each of the points for the jacobian
        f1x1 = (func{1}(x + [h;0])- func{1}(x - [h;0]))/(2*h);
        f1x2 = (func{1}(x + [0;h])- func{1}(x - [0;h]))/(2*h);
        f2x1 = (func{2}(x + [h;0])- func{2}(x - [h;0]))/(2*h);
        f2x2 = (func{2}(x + [0;h])- func{2}(x - [0;h]))/(2*h);

        % Set Jacobian and f
        f = [f1;f2];
        J = [f1x1,f1x2;f2x1,f2x2];

        % Calculate the Delta
        del = -1*linsolve(J,f);
```

```
        % Find the new xvalues
        xnew = del + x;

        % Calculate the new f values
        fnew = [func{1}(xnew);func{2}(xnew)];
        %Use condition criteria to cancel out of the list
        if (abs(fnew(1))< tol) && (abs(fnew(2))< tol)
            % Add to the storage arrays
            xstore = [xstore,xnew];
            x = xstore;
            return
        end
        xstore = [xstore,xnew];
        x = xnew;
    end
    disp('The maximum number of iterations have been performed
       without satisfying the required root finding condition');
end
```

Listing 9: Task 3

```
%% Task 3 - System of Nonlinear Equations

% clear workspace
clear all
clc

% initialisation
tol = 1.0e-6;        % numerical tolerance
h = 1.0e-4;          % step size for central difference
nmax = 50;           % maximum number of iterations
x0 = [2,0];          % initial root estimate
func = {@f1, @f2}; % array of function handles

% set func_usage to false once you know how vector/array func
   works
func_usage = true;
if func_usage
    f1_initial = func{1}(x0);
    f2_initial = func{2}(x0);
end

% 2D Newton's method and verification
disp(['Newton2Var starting at point (x0,y0) = (' num2str(x0(1)),'
   ,',num2str(x0(2)),')']);
xn = Newton2Var(func, x0, h, nmax, tol);
disp([xn(1,end),xn(2,end)]);
assert(abs(func{1}([xn(1,end),xn(2,end)])) <= tol)
assert(abs(func{2}([xn(1,end),xn(2,end)])) <= tol)
disp(['Newton2Var converged to root at (x,y) = (' num2str(xn(1,
   end)),',',num2str(xn(2,end)),') in ',num2str(length(xn)),'
```

```
    iterations']);


% Functions to be used for testing out Newton2Var
function f = f1(x)
    f = x(1)*x(1)-2*x(1)+x(2)*x(2)+2*x(2)-2*x(1)*x(2)+1;
end
function f = f2(x)
    f = x(1)*x(1)+2*x(1)+x(2)*x(2)+2*x(2)+2*x(1)*x(2)+1;
end
```

4.2 Newton Two Variable Plotting

Listing 10: Task 4

```
%% Task 4

% clear workspace
clear all
clc

% initialisation
tol = 1.0e-6;       % numerical tolerance
h = 1.0e-4;         % step size for central difference
nmax = 50;          % maximum number of iterations
x0_p1 = [-1,-3];     % initial root estimate - point 1
x0_p2 = [2,-3];     % initial root estimate - point 2
x0_p3 = [2,0];      % initial root estimate - point 3
func = {@f1, @f2}; % array of function handles

% Two function two variable Newton's method for each starting
    location
xn_p1 = Newton2Var(func, x0_p1, h, nmax, tol);
xn_p2 = Newton2Var(func, x0_p2, h, nmax, tol);
xn_p3 = Newton2Var(func, x0_p3, h, nmax, tol);

%% start figure for algorithm visualisation
figure(1), clf
% Calculate all the values needed

% Iteration count 1
[~,c1] =size(xn_p1);
k1 = 1:c1;

% Call the first function for each iteration from the first
    starting point.
for i = 1:c1
    fp11(i) = func{1}([xn_p1(1,i),xn_p1(2,i)]);
end

%Iteration Count 2
```

```
[~,c2] =size(xn_p2);
k2 = 1:c2;

% Call the first function for each iteration from the second
    starting point.
for i = 1:c2
    fp21(i) = func{1}([xn_p2(1,i),xn_p2(2,i)]);
end

% Iteration count 3
[~,c3] =size(xn_p3);
k3 = 1:c3;

% Call the first function for each iteration from the third
    starting point.
for i = 1:c3
    fp31(i) = func{1}([xn_p3(1,i),xn_p3(2,i)]);
end

% Call the second function for each iteration from the first
    starting point.
for i = 1:c1
    fp12(i) = func{2}([xn_p1(1,i),xn_p1(2,i)]);
end

% Call the first function for each iteration from the second
    starting point.
for i = 1:c2
    fp22(i) = func{2}([xn_p2(1,i),xn_p2(2,i)]);
end

% Call the first function for each iteration from the third
    starting point.
for i = 1:c3
    fp32(i) = func{2}([xn_p3(1,i),xn_p3(2,i)]);
end


% Create all the labels to plot with
f1title = 'f_{1}(x^{(k)},y^{(k)}) Converging from the Intial Root
     Estimate vs k';
f2title = 'f_{2}(x^{(k)},y^{(k)}) Converging from the Intial Root
     Estimate vs k';
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)';
f1xlabel = 'Number of Iterations (k)';
f1ylabel = 'f_{1}(x^{(k)},y^{(k)})';
f2xlabel = 'Number of Iterations (k)';
f2ylabel = 'f_{2}(x^{(k)},y^{(k)})';
surfxlabel = 'x_1';
surfylabel = 'x_2';
```

```
surfzlabel = 'f_1/f_2';

% Create axis labels
f1p1 = 'f_1^{p1}:(-1,-3)';
f1p2 = 'f_1^{p2}:(2,-3)';
f1p3 = 'f_1^{p3}:(2,0)';
f2p1 = 'f_2^{p1}:(-1,-3)';
f2p2 = 'f_2^{p2}:(2,-3)';
f2p3 = 'f_2^{p3}:(2,0)';
f1legend = {f1p1,f1p2,f1p3};
f2legend = {f2p1,f2p2,f2p3};
surflegend = {f1p1,f2p1,f1p2,f2p2,f1p3,f2p3};

% create top left plot for function 1
subplot(2,2,1)
hold on
plot(k1,fp11,'b','Linewidth',2)
hold on
plot(k2,fp21,'r','Linewidth',2)
hold on
plot(k3,fp31,'g--','Linewidth',2)
ylabel('Function 1 values')
xlabel('Number of Iterations (k)')
title(f1title)
legend(f1legend)
xlabel(f1xlabel)
ylabel(f1ylabel)
xlim([1,14]);

% create bottom left plot for function 2
subplot(2,2,3)
hold on
plot(k1,fp12,'b','Linewidth',2)
hold on
plot(k2,fp22,'r','Linewidth',2)
hold on
plot(k3,fp32,'g--','Linewidth',2)
ylabel('Function 2 values')
xlabel('Number of Iterations (k)')
title(f2title)
legend(f2legend)
xlabel(f2xlabel)
ylabel(f2ylabel)
xlim([1,14]);

% create right plot for 3d visualisation of 2d newton's method
subplot (2,2,[2 4])
% Plot both the functions
[X,Y] = meshgrid(-1:0.1:2,-3:0.1:2);
Z1 = X.*X-2.*X+Y.*Y+2.*Y-2.*X.*Y+1;
Z2 = X.*X+2.*X+Y.*Y+2.*Y+2.*X.*Y+1;
```

```
surf(X,Y,Z1);
hold on;
surf(X,Y,Z2);

% Improve plotting
alpha 0.4; % Make more transparent
rotate3d on; % Automatic switch on rotate feature
shading interp; % Change shading
view(255,25); % Specify view found by trial and error.

% Plot the lines on the surface
hold on
ob1 = plot3(xn_p1(1,:),xn_p1(2,:),fp11,':r.','Linewidth',2,'
    Markersize',20,'MarkerEdgeColor','k');
hold on
ob3 = plot3(xn_p2(1,:),xn_p2(2,:),fp21,':b.','Linewidth',2,'
    Markersize',20,'MarkerEdgeColor','k');
hold on
ob5 = plot3(xn_p3(1,:),xn_p3(2,:),fp31,':g.','Linewidth',2,'
    Markersize',20,'MarkerEdgeColor','k');
hold on
ob2 = plot3(xn_p1(1,:),xn_p1(2,:),fp12,':r+','Linewidth',2,'
    Markersize',10,'MarkerEdgeColor','k');
hold on
ob4 = plot3(xn_p2(1,:),xn_p2(2,:),fp22,':b+','Linewidth',2,'
    Markersize',10,'MarkerEdgeColor','k');
hold on
ob6 = plot3(xn_p3(1,:),xn_p3(2,:),fp32,':g+','Linewidth',2,'
    Markersize',10,'MarkerEdgeColor','k');
ob = [ob1,ob2,ob3,ob4,ob5,ob6];

% Plot labels
legend(ob,surflegend,'Location','northeast','NumColumns',3)
title(surftitle);
xlabel(surfxlabel);
ylabel(surfylabel);
zlabel(surfzlabel);
grid on;

% Functions to be used for testing out Newton2Var
function f = f1(x)
    f = x(1)*x(1)-2*x(1)+x(2)*x(2)+2*x(2)-2*x(1)*x(2)+1;
end
function f = f2(x)
    f = x(1)*x(1)+2*x(1)+x(2)*x(2)+2*x(2)+2*x(1)*x(2)+1;
end
```

4.3 Newton Multiple Variable

Listing 11: Newton Multiple Variables

```
% Nonlinear equation root finding in n dimensions using Newton's Method.
% Inputs
% n       : number of dimensions for Newton's method
% func    : array of function handles for system of nonlinear equations
% x0      : vector of initial root estimates for each independent variable
% h       : step size for numerical estimate of partial derivatives
% nmax    : maximum number of iterations performed
% tol     : numerical tolerance used to check for root
% Outputs
% x       : array (n-row matrix) containing estimates of root

% Hint 1:
% Include the initial root estimate as the first column of x

% Hint 2:
% Use MATLAB in-built functionality for solving the matrix equation for vector of updates, delta

% Hint 3:
% Check for root each iteration, continuing until the maximum number of iterations has been reached

function x = NewtonMultiVar(n, func, x0, h, nmax, tol)
% Initial array of root estimates, iteration and variable stores.
    % Initialise a storage array
    xstore = transpose(x0);
    x = xstore; % Vector of the most recent root variables
    n = 1;
    % Set iterative loop for the function
    while n < nmax
        % Calculate the current root estimate, used a nested for loop.
        for i = 1:length(func)
            f(i,1) = func{i}(x); % Vector of variables passed into the function call
        end
        % Initialise the size of the jacobian
        jacob = zeros(length(func):length(x));
        % Calculate the jacobian
        for i = 1:length(func)
            for j  = 1: length(x)
                % Create two small steps for the x values
                x(j) = x(j) + h;
                % Find the first part of the derivitive calculation.
                func1 = func{i}(x);
```

```
                % Do the second part of the derivitive calculation.
                x(j) = x(j) - 2.*h;
                % Find the first part of the derivitive calculation.
                func2 = func{i}(x);
                % Calculate the jacobian
                jacob(i,j) = ((func1 - func2)./(2.*h));
                % Correct the x value
                x(j) = x(j) + h;
            end
        end
        % Inverse the jacobian and get del
        del = -1.*(jacob\f);
        % Perform the necesary exit conditions
        % New del
        xnew = del + x;
        % Recalculate f with xnew values
        for i = 1:length(func)
            fnew(i,1) = func{i}(transpose(xnew)); % Vector of variables passed into the function call
        end
        % Test for both convergence and function call close to zero.
        if (prod((abs(fnew)<= tol)) == 1) && (prod((abs(del)<= tol))== 1)
            % Add to the store arrays
            xstore = [xstore,xnew];
            x = xstore;
            return
        end
        xstore = [xstore,xnew];
        x = xnew;
        % Increase iteration counter
        n = n + 1;
    end
    % Warn the user the maximum number of iterations have been performed
    disp('The maximum number of iterations have been performed without satisfying the required root finding condition');
end
```

2018

SEMESTER 2

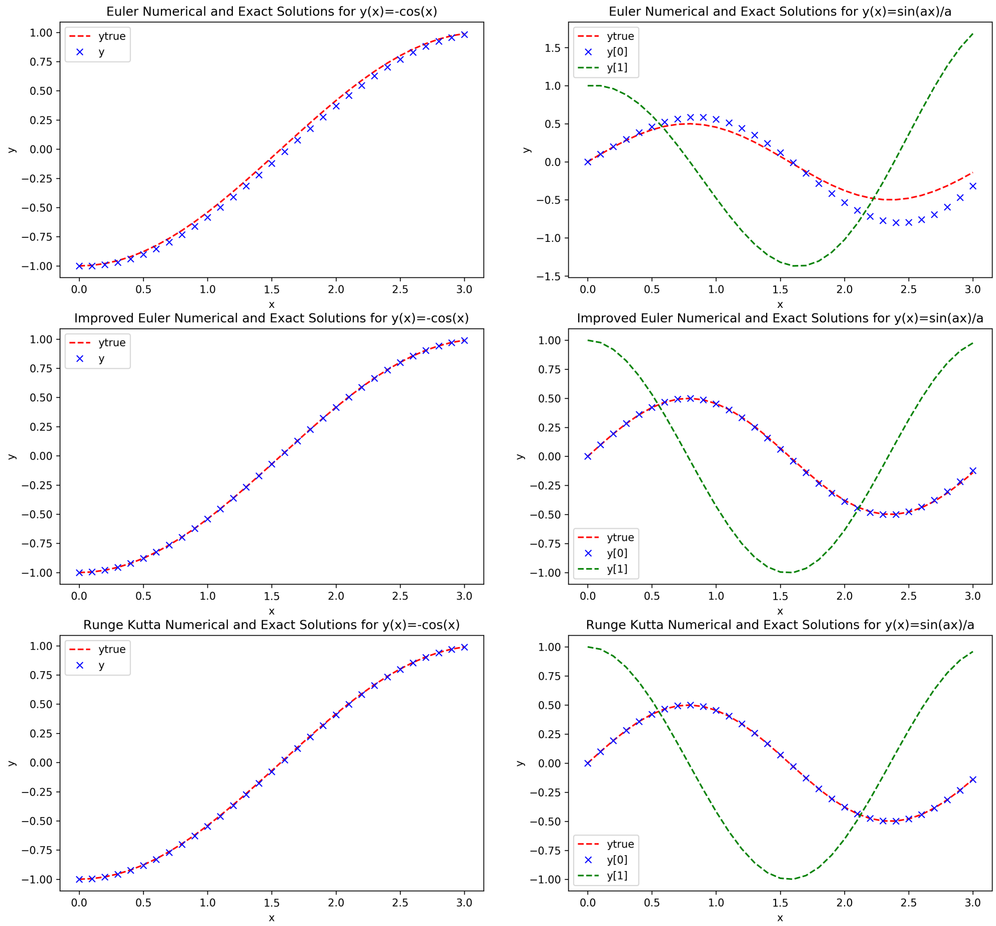
ENGCSI 331 Lab 2 ODEs

Connor McDowall
CMCD398 530913386

August 14, 2018

1 Improved Euler and Runge Kutta Solves

1.1 Hand in 1



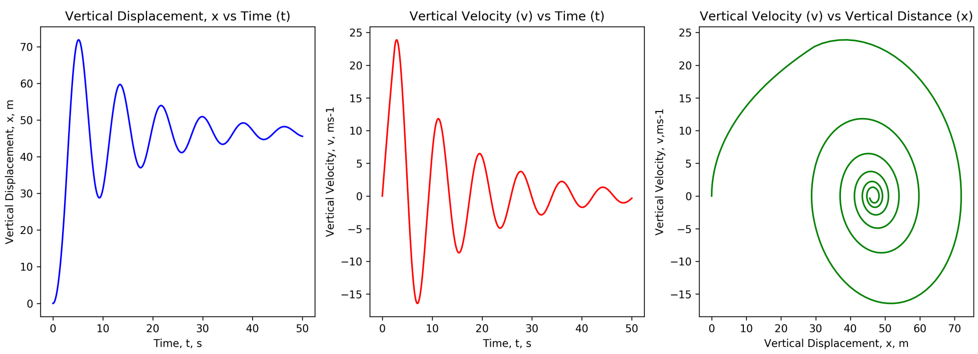
```python
def improved_euler_solve(f, x0, y0, x1, h, *args):
    ''' Compute solution to ODE using improved Euler method

        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))
```

```python
# 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

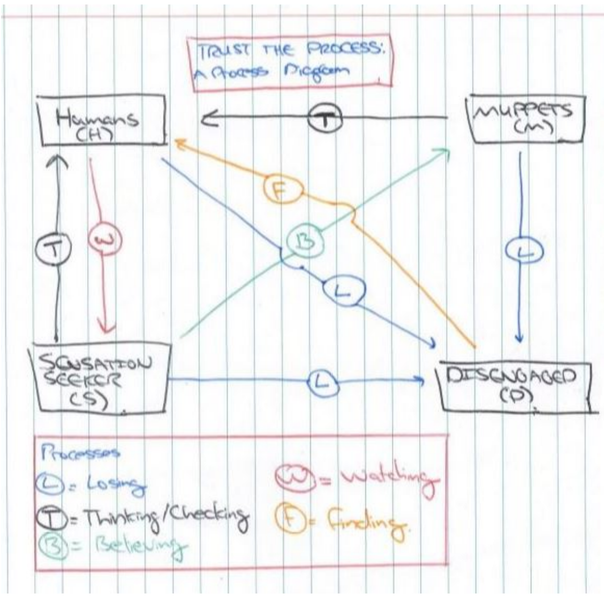
2.1 Hand in 2

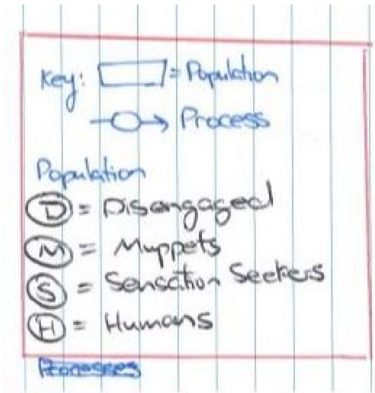



```
# Task 2
# Higher Order System: Bungy Jumper Model
def bungee_jumper_derivative(t,x,g,cd,m,k,L,gamma):
    if (x[0]<= L):
        result =  np.array([x[1],g-np.sign(x[1])*(cd/m)*x[1]**2] )
        return result
    else:
        result = np.array([x[1],(g-np.sign(x[1])*(cd/m)*x[1]**2 - (k/m)*(x[0]-L)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

3 Fakes News

3.1 Hand in 3: Process Diagram





3.2 Hand in 4: Derivative Relationship

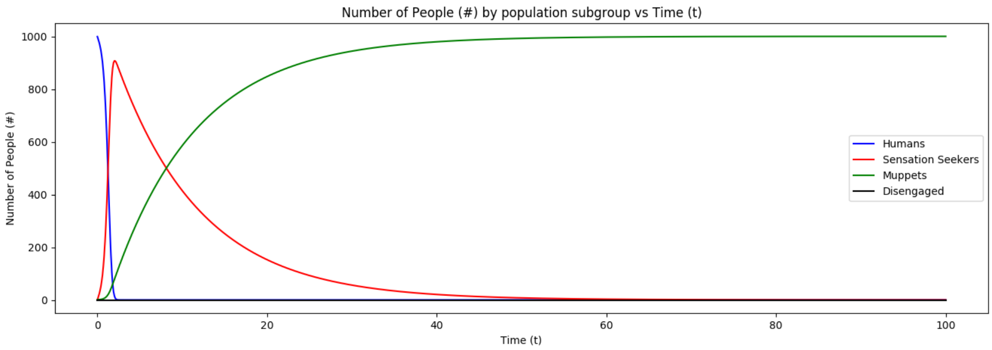
$$\frac{dH}{dt} = (S \times pt) - (H \times M \times pw) + (D \times pf) + (M \times pt) - (H \times pl) \tag{1}$$

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

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

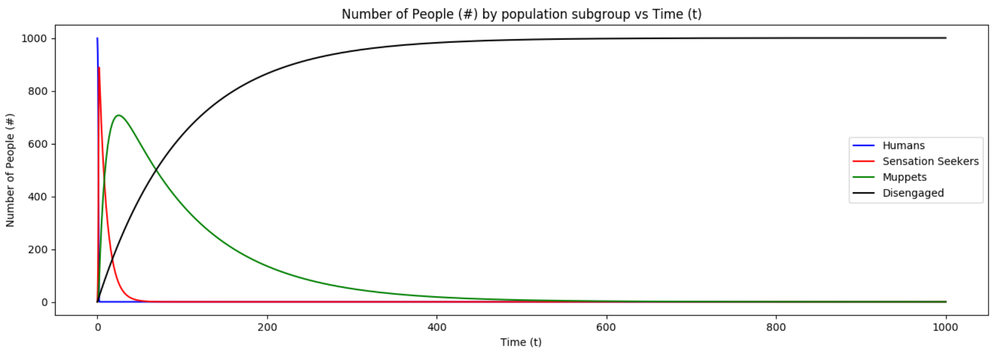
$$\frac{dD}{dt} = (S \times pl) + (H \times pl) + (M \times pl) - (D \times pf) \tag{4}$$

3.3 Hand in 5



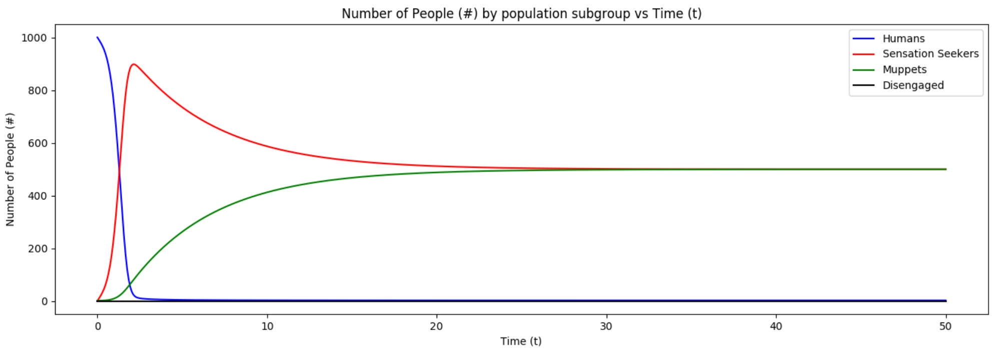
Humans will be rapidly converted to sensation seekers with sensation seekers also converted to muppets at the same time. Once there are no humans left, all sensation seekers will be convert to muppets, enough to occupy street for many years to come.

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 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 stil 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 3
e\extensions\ms-python.python-2018.7.1\pythonFiles\PythonTools\visualstuc
utput' '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 penaltie
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
```

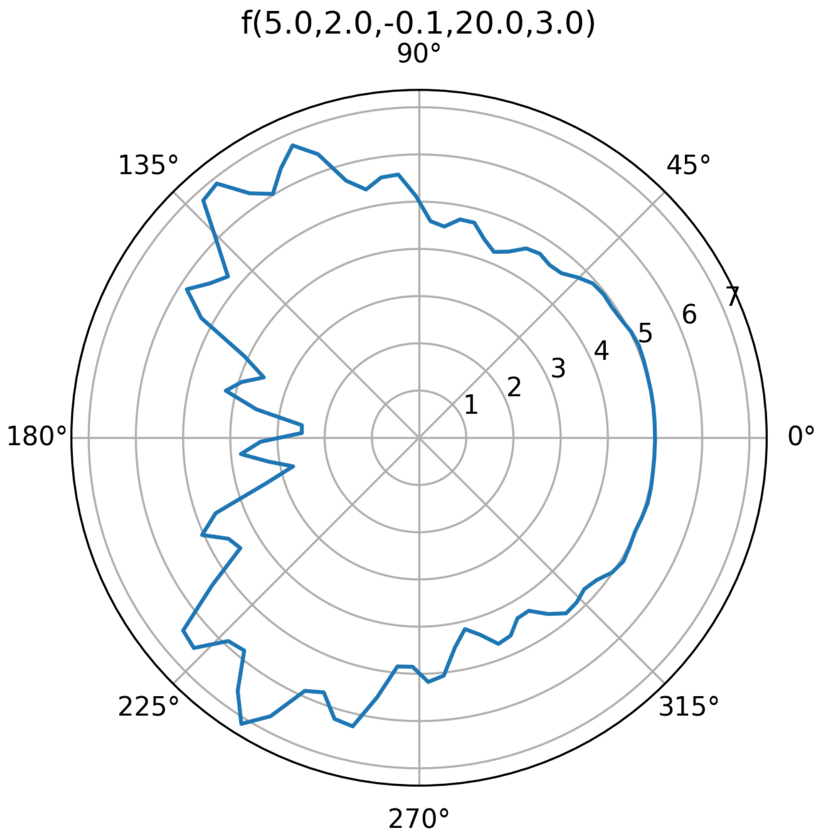
```python
# Adaptative runge kutta step
def adaptive_runge_kutta_step(f, xk, yk, h, tol, *args):
    ''' Compute a single Runge Kutter step that adapts the step size.

        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)
    thirdO = yk + ((h/6)*(f0 + 4*f1 + f2_3rd))
    fourthO = yk + ((h/6)*(f0 + 2*f1 + 2*f2_4th + f3))
    # Calculate the observed error
    obs = abs(thirdO - fourthO)
    # Calculate the new h value
    hnew = h*(np.abs(tol/obs)**0.2) if obs > 1e-12 else h
    # 4th Order Return
    return xk, fourthO, hnew
```

```python
# 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 estimatation 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

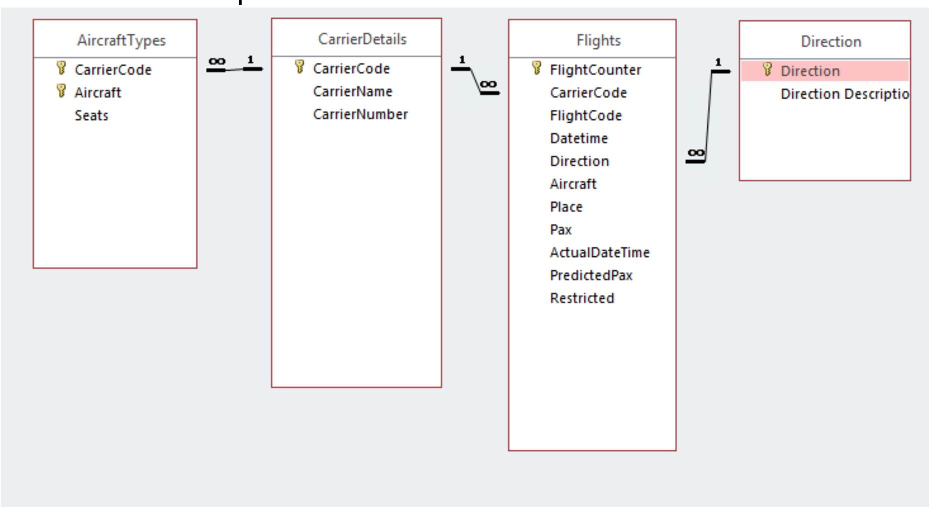
a. Design mode

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

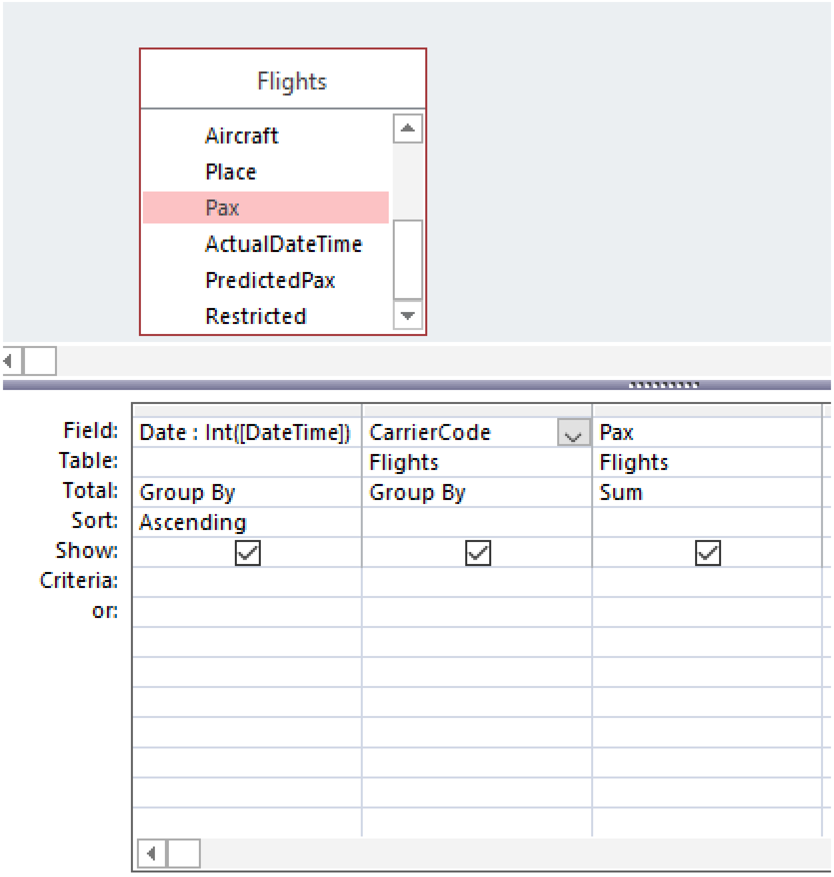
d. Datasheet View

Date	CarrierCode	SumOfPax
1/02/1996	[illegible]	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	PH	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



Question 5

FlightSeats

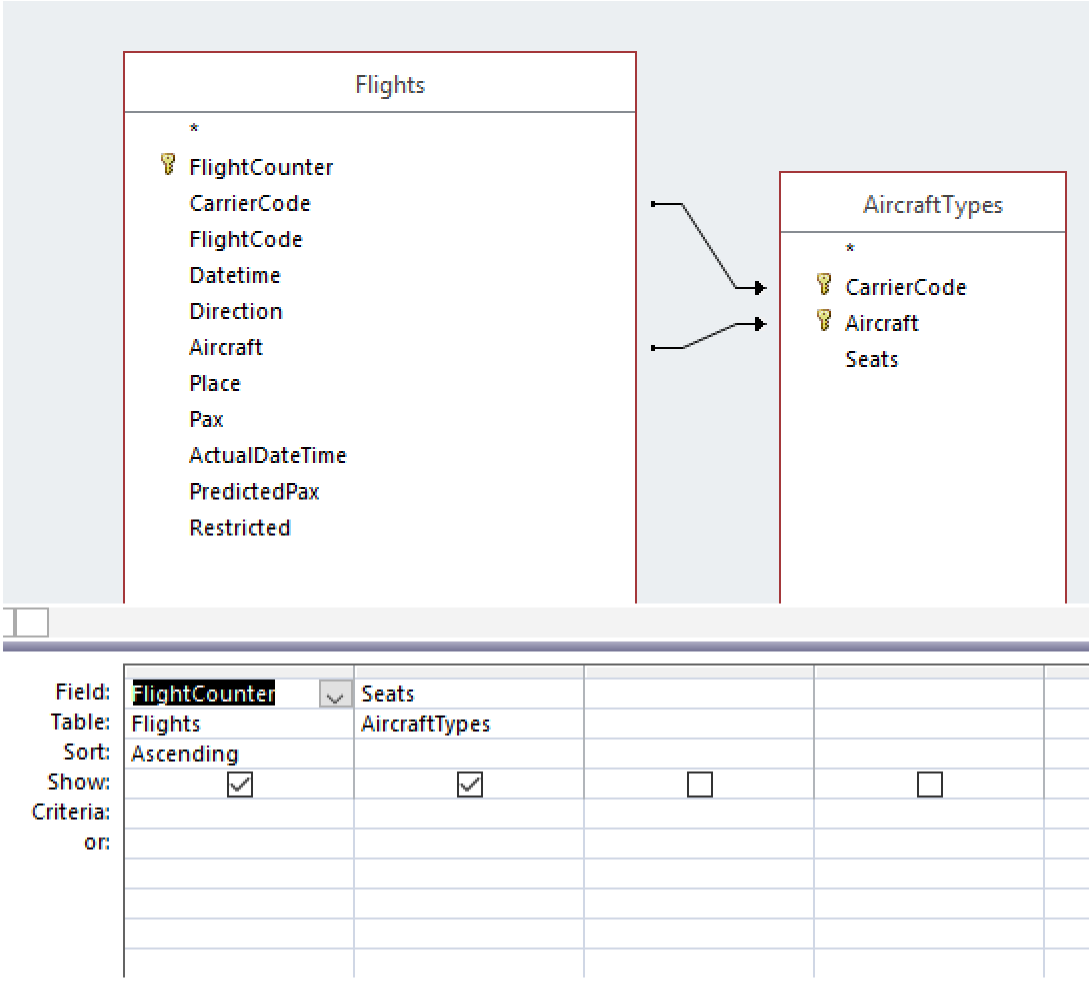
g. Database view

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

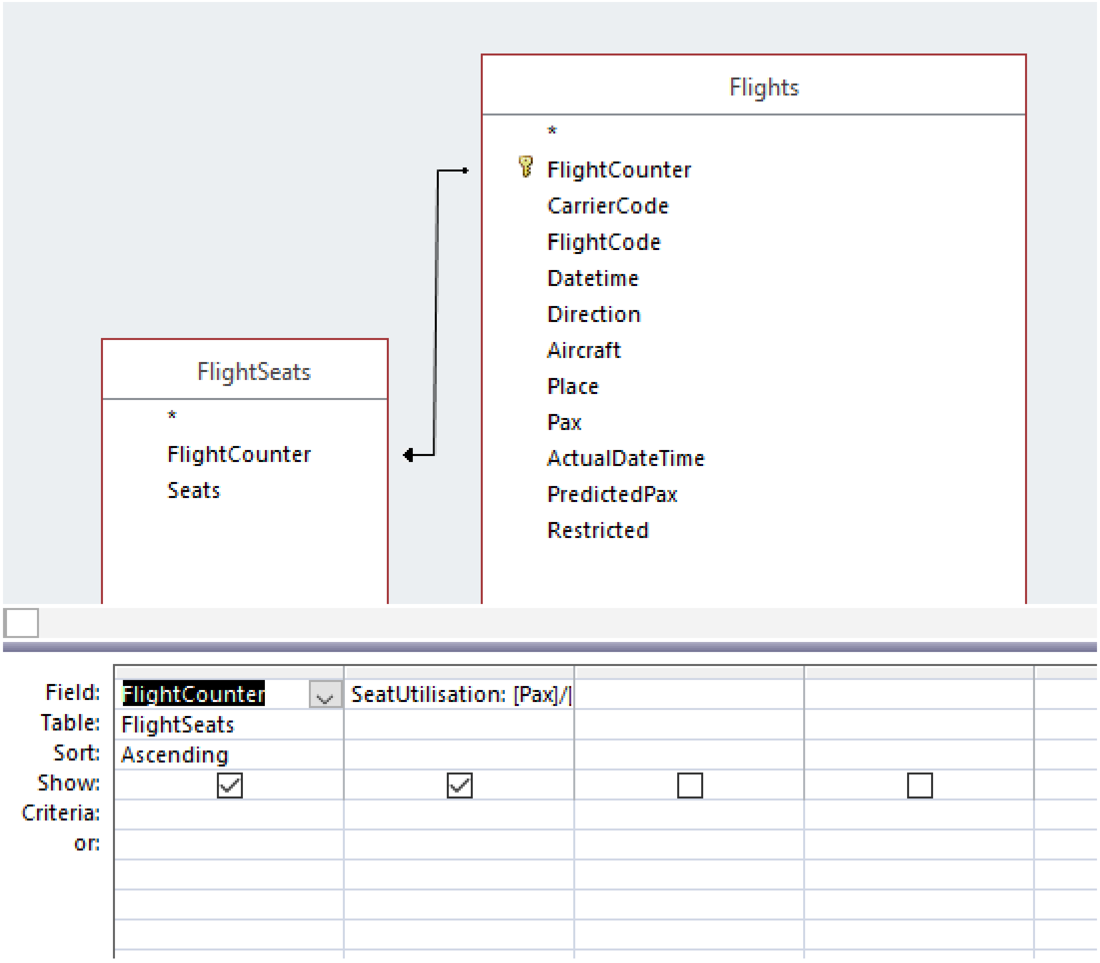
j. Database view

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	0.7661691542

k. SQL view

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

l. Design view



Question 7

CarrierUtilisation

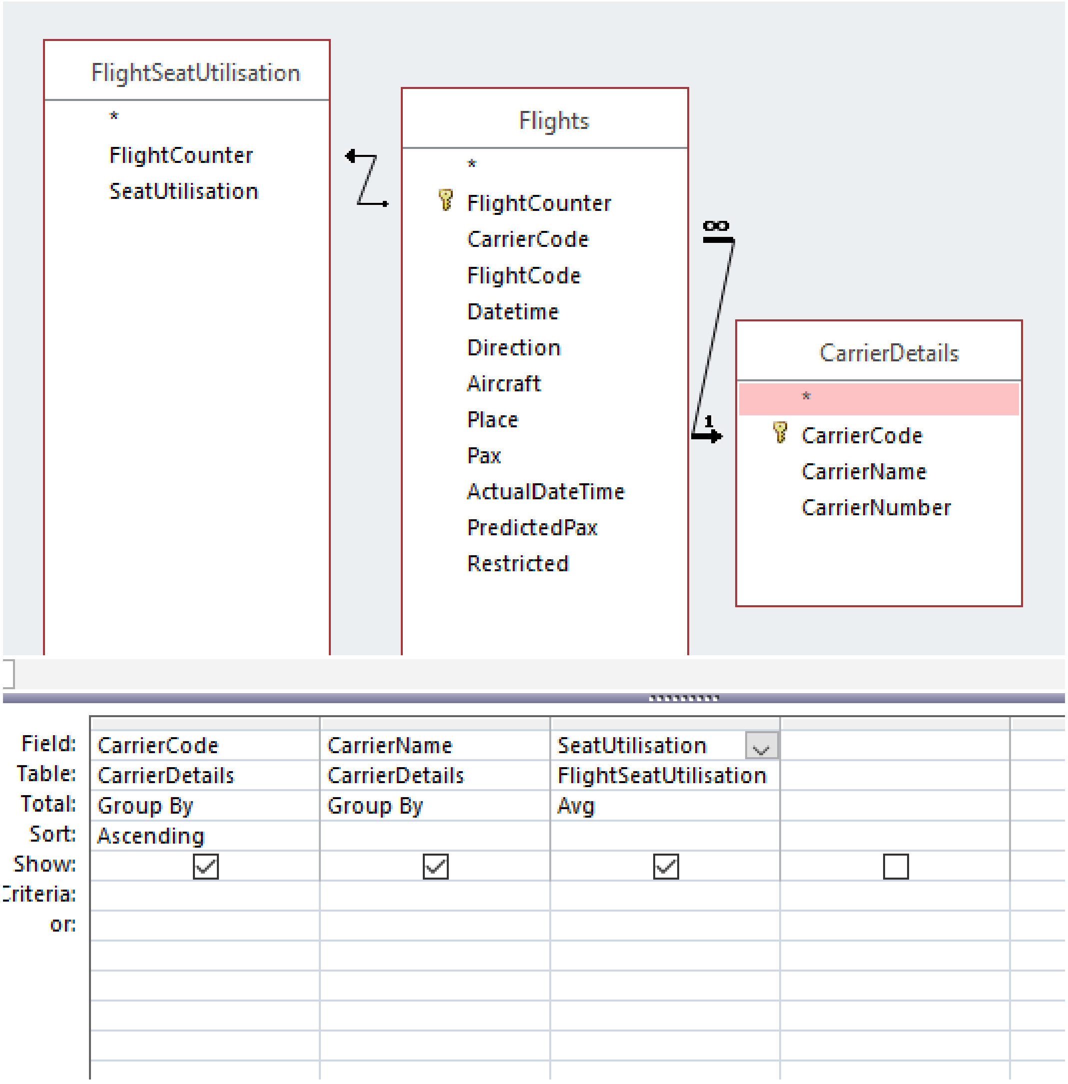
m. Database view

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



Field:	CarrierCode	CarrierName	SeatUtilisation	
Table:	CarrierDetails	CarrierDetails	FlightSeatUtilisation	
Total:	Group By	Group By	Avg	
Sort:	Ascending			
Show:	☑	☑	☑	☐
Criteria:				
or:				

Part 2

1. # 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	LOC0MOBILE	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

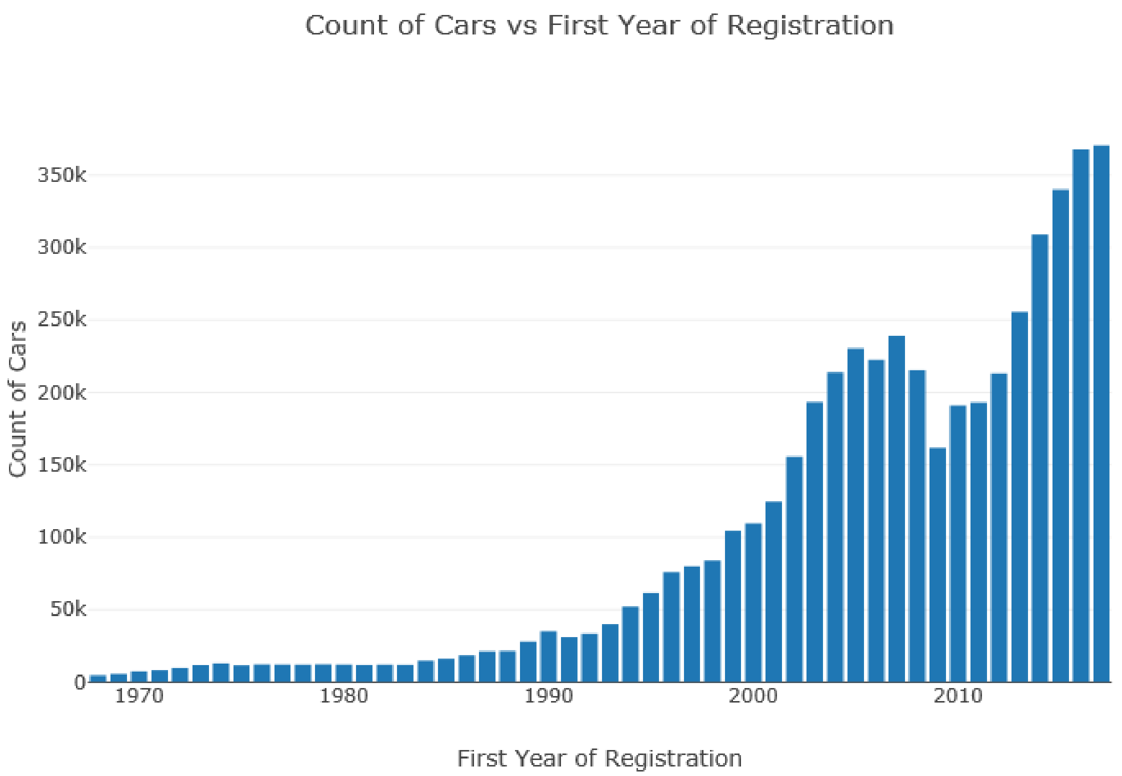
5. # How many cars are first registered in each of the most recent 50 years?

```
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)
```



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

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

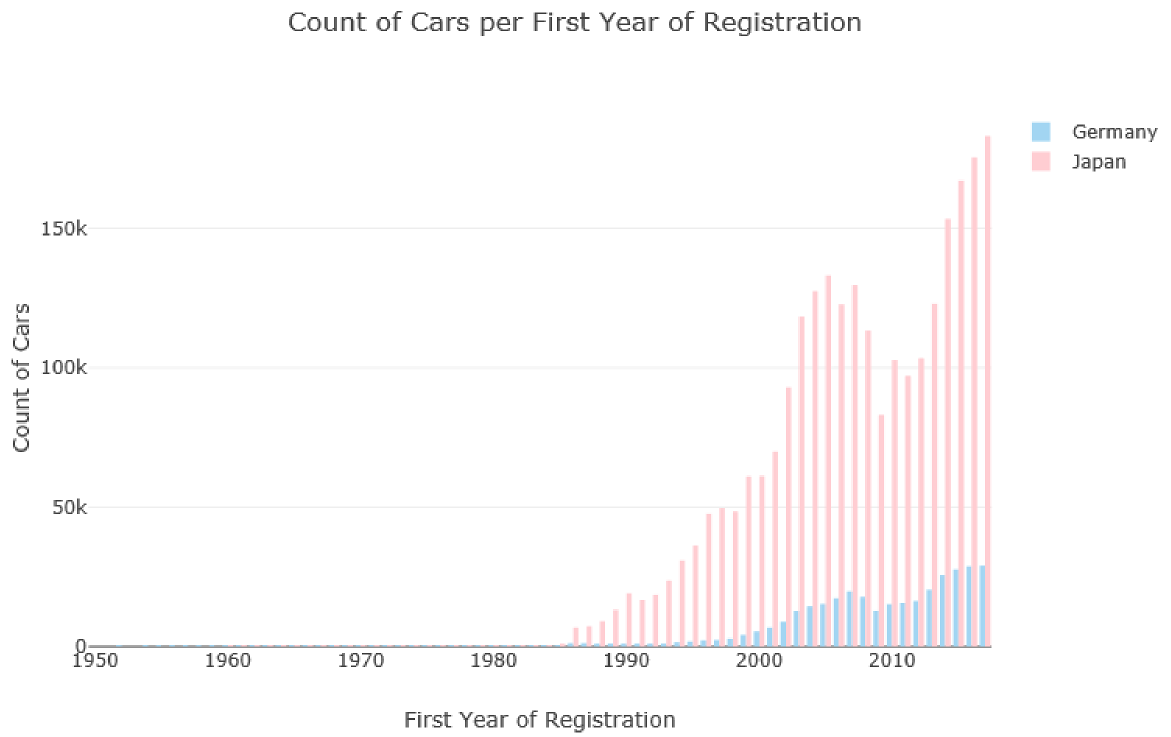
	FIRST_NZ_REGISTRATION_YEAR	JapanCars
0	2017	183069
1	2016	175471
2	2015	167140
3	2014	153308
4	2013	123021
5	2012	103495
6	2011	97141
7	2010	102768
8	2009	83153
9	2008	113423
10	2007	129588

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.GermanCars,name = 'Germany',marker=dict(color='#A2D5F2'))
trace_japan =
plotly.graph_objs.Bar(x=Jap.FIRST_NZ_REGISTRATION_YEAR,y=Jap.JapanCars,name = 'Japan',marker=dict(color='#ffcdd2'))

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)
```