COSC2500 Final Exam

Ryan White
s4499039

7th of November 2020

Required

R1

The numerically calculated derivative for $x^5 - 4x + 3$ at $x = 1$ was found to be

Step Size	Derivative	Error
1×10^{-3}	1.000010000000939	1.000000093931419e-05
1×10^{-5}	1.000000001027956	1.027956386678852e-09
1×10^{-7}	0.999999998363421	-1.636578872421524e-09

Table 1: Numerically Calculated Derivatives for $x^5 - 4x + 3$ at $x = 1$

This data contradicts what one might expect, because as the step size gets smaller, the error eventually gets larger. This, however, is just another example as what was shown in Assignment 1, where, for the central difference method, the minimum error occurs around a step size of $h = 1 \times 10^{-5}$. At step sizes both smaller and larger than this, the error begins to rise again. This data was calculated using the Matlab Code:

```
format long
x = 1;
for h = [1e-3, 1e-5, 1e-7]
    deriv = (((x + h)^5 - 4*(x + h) + 3) - ((x - h)^5 - 4*(x - h) + 3)) / (2 * h)
    err = deriv - 1
end
```

R2

Using successive parabolic interpolation, the x value at the maximum for $f(x) = x^3 e^{-x^2}$ was found to be $x = 1.2247$, with a function value of $f(1.2247) = 0.4099$. This code used initial guesses of maximum value $x = 1$ on the domain $x = 0 \to 2$, and 10 steps. The code used was

```
func = @(x) x^3 * exp(-1 * x^2);
x = spi(func, 0, 2, 1, 10)
y = func(x)


function y = spi(f, r, s, t, k)
    % Input: inline function f, initial guesses r, s, t, steps k
    % Output: approximate minimum x
    % Source: Sauer, 3rd Edition
    spi_count = 0;
    x(1) = r; x(2) = s; x(3) = t;
    fr = f(r); fs = f(s); ft = f(t);
    for i = 4:k + 3
        spi_count = spi_count + 1;
        x(i) = (r + s) / 2 - (fs - fr) * (t - r) * (t - s) / (2 * ((s - r) * (ft - fs) - (fs - fr)
            * (t - s)));
        t = s; s = r; r = x(i);
        ft = fs; fs = fr; fr = f(r);     %single function evaluation
    end
    y = x(end);
end
```

R3

The solved 8×8 Hilbert Matrix using Gaussian Elimination yielded the results

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} -8 \\ 504 \\ -7.56 \times 10^3 \\ 4.62 \times 10^4 \\ -1.386 \times 10^5 \\ 2.1622 \times 10^5 \\ -1.6817 \times 10^5 \\ 5.148 \times 10^4 \end{bmatrix}$$

With condition number $1.525757548992251 \times 10^{10}$, and so there is a loss of accuracy of approximately 10 digits (and so in double precision, there are about 6 correct digits in the solution). The code used was

```
sol = gausselim(hilb(8), ones(8, 1));
cond(hilb(8))
function x = gausselim(A,b)
    %Gaussian Elimination
    %Input: matrices A and b, where A are the multiples of the unknowns,
    %and b the solutions
    %Output: Matrix x of solutions
    %Source: Huskie, K, Jan, (1)
    [row, ~] = size(A);
    n = row;
    x = zeros(size(b));
    for k = 1:n-1
      for i = k+1:n
         xMultiplier = A(i,k) / A(k,k);
         for j=k+1:n
            A(i,j) = A(i,j) - xMultiplier * A(k,j);
         end
         b(i, :) = b(i, :) - xMultiplier * b(k, :);
      end
        % backsubstitution:
        x(n, :) = b(n, :) / A(n,n);
        for i = n-1:-1:1
            summation = b(i, :);
            for j = i+1:n
                summation = summation - A(i,j) * x(j, :);
            end
            x(i, :) = summation / A(i,i);
        end
    end
end
```

---

with the reference:
Huskie, K, Jan, 2018. *Matlab Function: Solving Ax=B using Gaussian Elimination where b is a n x m matrix not necessarily a n x 1 matrix.* Matlab Answers, Accessed 19/09/2020. https://au.mathworks.com/matlabcentral/answers/404921-matlab-function-solving-ax-busing-gaussian-elimination-where-b-is-a-n-x-m-matrix-not-necessarily-a

R4

Direct methods of solving matrices a subject to n^3 time dependence and quickly become impractical as the number of terms grows by orders of magnitude. In order to help reduce the time necessary for computation, iterative methods (with time dependence of n^2) can be used. While this still isn't ideal for higher orders of magnitude of matrix dimensions, it offers a considerable time reduction in calculation. Iterative methods, however, don't always converge to a solution, although different iterative functions offer various suitability criteria. A further time reduction is seen involving computation with sparse matrices. Sparse matrices also save far more memory than normal matrices.

R5

The plotted numerical solution for $f'(x) = xsin(x)$ from $x = 0$ to $x = \pi$, with $y(0) = 0$ is shown below

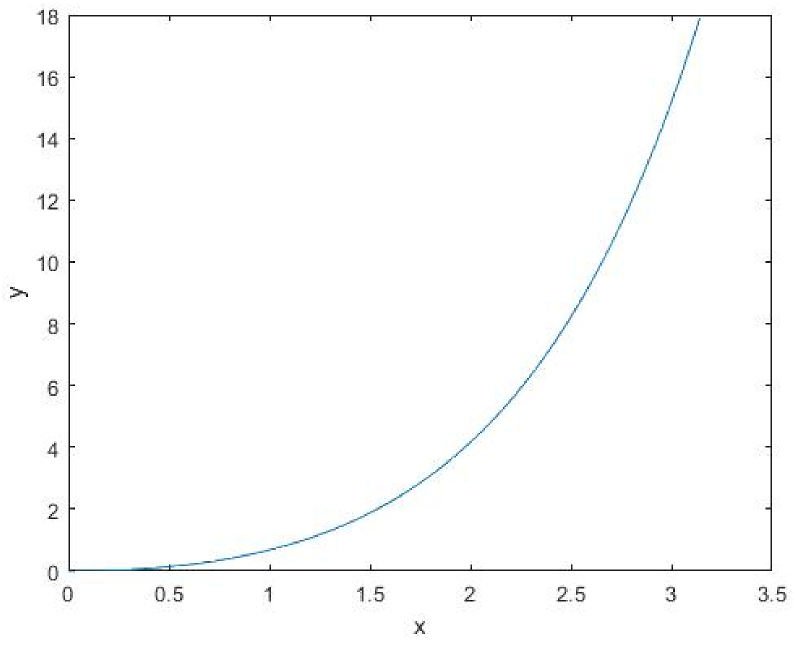

Figure 1: Plotted Solution for $f'(x) = xsin(x)$ with $y(0) = 0$

This was calculated with Euler's Method, and the code:

```
[xe, ye] = euler1([0, pi], 0, 100);
plot(xe, ye);
xlabel('x'); ylabel('y');

function [x, y] = euler1(inter, y0, n)
    x(1) = inter(1); y(1) = y0;
    h = (inter(2) - inter(1)) / n;
    for i = 1:n
        x(i + 1) = x(i) + h;
        y(i + 1) = eulerstep(x(i), y(i), h);
    end
end

function y = eulerstep(x, y, h)
    y = y + h * ydot(x, y);
end

function z = ydot(x, y)
    z = x + y;
end
```

R6

Adaptive step-size methods are characterised by minimising error across all step sizes, and use a calculated step size such that the error remains below some defined value. Stiff equations are characterised by very large (in magnitude) higher order derivatives. Thus, an adaptive step-size method computing a high-order derivative of a stiff equation yields large error, and so the step-size is reduced accordingly to keep error below some maximum, pre-determined threshold. And so a negative-feedback loop is created in that, for high order derivatives, the step-size is iteratively reduced and eventually becomes so small that it becomes computationally very expensive compute to the end of the domain of the calculation. The easiest solution in such a case is to use a lower-order adaptive-step size method (at the cost of some accuracy). Otherwise, an approximation of the stiff equation may be used (at the cost of more accuracy).

R7

For cases where uncertainty in data points is negligible or zero, fitting an exact fit, higher order polynomial to the data would be desirable (discounting computation time). In every other case, however, a least-squares fit of a lower-order polynomial is often more ideal. This method has the effect of 'averaging out' uncertainties in data, which reduces the effect of anomalous data. Least-squares fitting is also often much more computationally cheap, especially for data where a solution is not easily found. Higher order polynomials are often oscillatory, which can, in some cases, provide an un-physical trendline connecting data.

Additional

A9

The computational model of the baking of a cake could be made in a few ways. The first of which I'll discuss is an optimisation problem, where one might want to find the global maximum of some function of the cake's "edibility". Consider some function of edibility, $E(t)$, composed of 4 separate, time-dependent functions: density ($\rho(t)$), moisture ($M(t)$), thermal conductivity ($K(t)$), and heat capacity ($C(t)$):

$$E(t) = \frac{MKC}{\rho} \tag{1}$$

where the variable t is time spent in the oven at some fixed temperature T. Now, imagining that this function of edibility is of the shape of an inverse parabola on the domain $[0, \infty)$, that is with one maximum edibility at time t_{ideal} and one local minima at $t = 0$. Finding the point at which $dE/dt = 0$ would yield the ideal edibility of the cake, and might be easily done with the central difference method. Of course, we all love cake even if it's a little over or under cooked, so some error is reasonable and the forward or backward difference methods might work also.
The arrangement of the variables above was chosen such that the function initially has a sharp incline, reaches some maximum, and then sharply declines; this approximates the transition from undercooked, to just right, and then to overcooked, in terms of the cake's edibility. It was assumed, based on personal cake-baking experience, that the density of the cake changes much quicker initially as opposed to the other variables (due to the cake quickly rising in the presence of oven temperatures), and so as the density is reduced, the edibility quickly rises. It was briefly thought out that each of the 4 variables might follow an inverse exponential function as time in the oven is increased, but was determined that density would have the highest initial (negative) rate of change.

Another possible model would be that of a system of equations. Considering that each of the variables were assumed to follow an inverse exponential curve in the last model, one could think of the baking of a cake as a system of each of the derivatives of these variables equalling 0. That is,

$$\begin{aligned} \frac{d\rho}{dt} &\approx 0 \\ \frac{dM}{dt} &\approx 0 \\ \frac{dK}{dt} &\approx 0 \\ \frac{dC}{dt} &\approx 0 \end{aligned}$$

Solving this system numerically using matrix operations would yield the time at which each variable's change with respect to time is negligible and so suggesting that the cake is fully cooked. This model, while definitively determining when the cake is cooked, would hypothetically be quite poor at finding when the cake is perfectly cooked, and instead favouring far overcooked and dry cakes.

The final model I'll suggest is that of an initial value problem, where the function in question is the derivative of the edibility with respect to time. For example,

$$\frac{dE(t)}{dt} = -2t - 3 \tag{2}$$

with initial value $E(0) = 0$ on the domain $[0, \infty)$. One might consider the solution for this IVP to be equation 1. Symbolically solving equation 2 gives an inverse parabola with $E(0) = 0$, which helps justify the form of the equation. Solving this equation numerically could be easily done with Euler's method, for example.

A11

One of the biggest sources of error in computational mathematics is rounding error, often in single-precision calculations. One example of this, even in this very exam sheet, is the use of Gaussian Elimination to solve the 8×8 Hilbert Matrix (R3). In this example, the exact error could not be calculated, but could be estimated by using the condition number which yielded the number of accurate digits lost. For a solution on one order of magnitude (the solution of x_1 in the case of R3), 10 digits were lost, equating to an error of about $\pm 1 \times 10^{-6}$. However, for a case where the solution was on an order of magnitude of 10^5 (see the solutions to x_5 through x_7), the error is as high as 1×10^1. While the relative error remains constant, the magnitude of the error increased significantly.

A16

The Matlab code given seems to be of variable quality. For instance, the author defines the value of `n` twice on lines 47 and 49. Another small nitpick is on line 92 where they use the `&` operator as opposed to the `&&` operator, which can save time over many iterations in a `for` loop. There are also minimal comments detailing what methods they're using and what exactly they're doing, bar the few instances of them defining the variables they're using.

As for the purpose of the code, to my knowledge it seems to be modelling the diffusion of some "peo" molecules through some volume of fixed radius. On each time step, the concentration of particles at the edge of the volume is calculated, along with the pressure they exert. Using these, the change in the number of particles inside the sphere is calculated, where the change is then added to (with a minus sign, so actually subtracted from) the total number of particles. With this data, the total concentration of peo particles over time is recorded, with the maximum and minimum values recorded at each time step.