Final Coursework Assessment Matlab
Alfred Sydney Brown ab16230

Question 1: Root finding

(a)

Newton-Raphson Method. NB, this was inspired by answer 2(a) in worksheet 3 on blackboard.

```
function [rt, it] = NetwonRap(fun,dfun,intvl, tol, itmax)
%
% NEWTONRAP Uses the Newton-Raphson iteration method to solve the scalar
% equation f(x) = 0
%
% Inputs:   fun = function handel to function whose roots should be found
%           dfun = function handel to the derivative function
%           intvl = the interval [a,b] that contains a root
%           tol = Absolute error tolerance with which to find the root;
%                 Note the iteration terminates when successive iterates are
%                 within the tolerance
%           itmax = maximum number of iterations
%
%
% Usage: rt = NetwonRap(fun,dfun,intvl, tol) returns the approximation to
%        the root, if there is convergence in the absolute error
%        [rt, it] = NetwonRap(fun,dfun,intvl, tol) returns the approximation
%        to the root, and the number of iterations taken for convergence



p = inputParser; %Error checking for inputed data - found in command 'doc
    validateattributes'
addRequired(p,'tol', @(x)validateattributes(tol, {'numeric'},{'nonempty','
    positive'},4))
addRequired(p,'itmax', @(x)validateattributes(itmax, {'numeric'},{'nonempty'
    ,'positive'},5))
addRequired(p,'fun', @(x)validateattributes(fun, {'function_handle'},{'
    nonempty'},1))
addRequired(p,'dfun', @(x)validateattributes(dfun, {'function_handle'},{'
    nonempty'},2))
addRequired(p,'intvl', @(x)validateattributes(intvl, {'numeric'},{'nonempty'
    , 'vector','numel',2},3))
parse(p,tol,itmax,fun,dfun, intvl)

if intvl(1) > intvl(2) %Error checking interval
    fprintf('\nPlease check the interval for sign convention and size error.
        For [a b], a < b.\n\n');
    return
end


fprintf('Iterations         Approx. to root       Absolute Error
    Relative Error\n');
X = linspace(intvl(1), intvl(2), 3);
x0 = X(2); %Finding initial guess to root
```

```
39
40 it = 1;
41 x1 = x0 - fun(x0)./dfun(x0);
42
43 fprintf(' %3d           %20.14f %20.14f%20.14f\n', it, x1, abs(x1-x0),abs((
       x1 - x0)/x0));
44 while abs(x1 - x0) > tol && it<itmax
45     it = it + 1;
46     x0 = x1;
47     x1 = x0 - fun(x0)./dfun(x0);
48     if ~(x1 > intvl(1) && x1 < intvl(2)) %Checking x1 is not outside the
          interval
49         rt = [];
50         fprintf('\nThere exists no root within this interval, please extend
              the interval\n');
51         return
52     else
53     fprintf(' %3d           %20.14f %20.14f%20.14f\n', it, x1, abs(x1-x0),
          abs((x1 - x0)/x0));
54     end
55 end
56
57
58
59 if abs(x1 - x0) > tol %Checking limit of convergence
60     rt = [];
61     fprintf('\nNo convergence below tolerance ater %d steps. Please increase
           maximum number of iterations.\n', it);
62 else
63     rt = x1;
64     fprintf('\nConvergence to root at %f after %d steps\n\n',rt,it);
65 end
66
67 if rt ~= []
68     fprintf('Final absolute error is %g\n\n', abs(x1 - x0));
69 end
```

Secant Method - Note the function for the following iterative method was found under wikipedia at *https* : *//en.wikipedia.org/wiki/Secant_method* [accessed 30.Oct.17]

```
1  function [rt,it] = secant(fun,intvl, tol, itmax)
2  %
3  % SECANT Uses the secant iteration method to solve the scalar
4  % equation f(x) = 0
5  %
6  % Inputs:   fun = function handel to function whose roots should be found
7  %           intvl = the interval [a,b] that contains a root
8  %           tol = Absolute error tolerance with which to find the root;
9  %                 Note the iteration terminates when successive iterates are
10 %                 within the tolerance
11 %           itmax = maximum number of iterations
12 %
13 %
14 % Usage: rt = secant(fun,dfun,intvl, tol) returns the approximation to
15 %             the root, if there is convergence in the absolute error
```

```
%        [rt, it] = secant(fun,dfun,intvl, tol) returns the approximation to
%             the root, and the number of iterations taken for convergence

p = inputParser; %Error checking for inputed data - found in command 'doc
    validateattributes'
addRequired(p,'tol', @(x)validateattributes(tol, {'numeric'},{'nonempty','
    positive'},4))
addRequired(p,'itmax', @(x)validateattributes(itmax, {'numeric'},{'nonempty'
    ,'positive'},5))
addRequired(p,'fun', @(x)validateattributes(fun, {'function_handle'},{'
    nonempty'},1))
addRequired(p,'intvl', @(x)validateattributes(intvl, {'numeric'},{'nonempty'
    , 'vector','numel',2},3))
parse(p,tol,itmax,fun, intvl)

if intvl(1) > intvl(2) %Error checking interval
    fprintf('\nPlease check the interval for sign convention and size error.
        For [a b], a < b.\n\n');
    return
end

fprintf('Iterations          Approx. to root       Absolute Error
    Relative Error\n');

x0 = intvl(1);
x1 = intvl(2);

it = 1;
x2 = x1 - fun(x1).*(x1 - x0)./(fun(x1) - fun(x0));
fprintf(' %3d           %20.14f %20.14f%20.14f\n', it, x1, abs(x2-x1),abs((
    x2 - x1)/x1));
while abs(x2 - x1) > tol && it<itmax
    it = it + 1;
    x0 = x1;
    x1 = x2;
    x2 = x1 - fun(x1).*(x1 - x0)./(fun(x1) - fun(x0));
    if ~(x2 > intvl(1) && x2 < intvl(2)) %Checking x1 is not outside the
        interval
        rt = [];
        fprintf('\nIt looks like the interval is too large, please shorten
            the interval to be closer to the root\n');
        return
    else
    fprintf(' %3d           %20.14f %20.14f%20.14f\n', it, x1, abs(x2-x1),
        abs((x2 - x1)/x1));
    end
end

if abs(x2 - x1) > tol %Checking limit of convergence
    rt = [];
    fprintf('\nNo convergence to below tolerance ater %d steps. Please
        increase maximum number of iterations.\n', it);
else
    rt = x2;
    fprintf('\nConvergence to root at %f after %d steps\n\n',rt,it);
```

```
end

if rt ~= []
    fprintf('Final absolute error is %g\n\n', abs(x2 - x0));
end
```

Ridder's method - Note the following iterative method was taken from C.J.F. Ridders from the following IEEE document $http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=1084580$

```
function [rt,it] = Ridder(fun,intvl, tol, itmax)
%
% RIDDER Uses the Ridder's iteration method to solve the scalar
%        equation f(x) = 0
%
% Inputs:   fun = function handel to function whose roots should be found
%           intvl = the interval [a,b] that contains a root
%           tol = Absolute error tolerance with which to find the root;
%                 Note the iteration terminates when successive iterates are
%                 within the tolerance
%           itmax = maximum number of iterations
%
%
% Usage: rt = Ridder(fun,dfun,intvl, tol) returns the approximation to
%              the root, if there is convergence in the absolute error
%        [rt, it] = Ridder(fun,dfun,intvl, tol) returns the approximation to
%              the root, and the number of iterations taken for convergence

p = inputParser; %Error checking for inputed data - found in command 'doc
   validateattributes'
addRequired(p,'tol', @(x)validateattributes(tol, {'numeric'},{'nonempty','
   positive'},4))
addRequired(p,'itmax', @(x)validateattributes(itmax, {'numeric'},{'nonempty'
   ,'positive'},5))
addRequired(p,'fun', @(x)validateattributes(fun, {'function_handle'},{'
   nonempty'},1))
addRequired(p,'intvl', @(x)validateattributes(intvl, {'numeric'},{'nonempty'
   , 'vector','numel',2},3))
parse(p,tol,itmax,fun, intvl)

if intvl(1) > intvl(2) %Error checking interval
    fprintf('\nPlease check the interval for sign convention and size error.
        For [a b], a < b.\n\n');
    return
end

fprintf('Iterations           Approx. to root      Absolute Error\n');

it = 1;
x0 = intvl(1); %Setting initial conditions
x2 = intvl(2);
xm = (x2 + x0)/2;

s = sqrt((fun(xm)/fun(x0)) - fun(x2)/fun(x0));
d = abs((x2 - x0)/2);

```

```
x3 = xm + d*(fun(xm)/fun(x0))/s; %Ridders iteratives method


if fun(xm) == 0 %Check middle isn't root
   rt = xm;
   fprintf('\nConvergence to root at %f after %d step\n\n',rt,it);
   return;
end

fprintf(' %3d              %20.14f %20.14f\n', it, x3, d);
while (d > tol) && (it < itmax)
    it = it + 1;
    if fun(x3)*fun(x0)<0 %Following if-statements decide next interval
        x2 = x3;
    end
    if fun(x3)*fun(x2) < 0
        x0 = x3;
    end
    if fun(x3)*fun(xm)<0
        if (x3 < xm)
            x0 = x3;
            x2 = xm;
        else
            x0 = xm;
            x2 = x3;
        end
    end
    xm = (x2 + x0)/2;
    s = sqrt((fun(xm)/fun(x0)) - fun(x2)/fun(x0));
    if s == 0
        fprintf('error');
        return;
    end
    if (x3 > x2) || (x3 < x0)
        fprintf('boundary error');
        return;
    end

    d = abs(xm-x2);
    x3 = xm + d*(fun(xm)/fun(x0))/s;
    fprintf(' %3d              %20.14f %20.14f\n', it, x3, d);
end

if d > tol %Checking limit of convergence
    rt = [];
    fprintf('\nNo convergence to below tolerance ater %d steps. Please
       increase maximum number of iterations.\n', it);
else
    rt = x3;
    fprintf('\nConvergence to root at %f after %d steps\n\n',rt,it);
end

if rt ~= []
    fprintf('Final absolute error is %g \n', d);
end
```

(b)

Procedure:

- First I rearranged the equations into the form $f(x) = 0$.
- Secondly I plotted a graph in order to see how many roots there are.
- Thirdly by zooming in and moving the functions around, I isolated suitable intervals to be inserted into the Ridder function.
- Note, the excepted tolerance for all roots was $5e^{-15}$.
- For function iii) I took the derivative and set it equal to zero to find the root.

Table 1: Approximation to root for function i)

Interval used	Approx. to root with step numbers	Root	Step
[0, 0.0025]	1,0.00067459731486 2,0.00103306884080 3,0.00100217109500 4,0.00100007028504 5,0.00100000112827 6,0.00100000000902 7,0.00100000000004 8,0.00100000000000 9,0.00100000000000	0.00100000	9
[0.08, 0.12]	1,0.10000000000000	0.10000000	1
[0.005, 0.012]	... 25,0.00999999745788 26,0.00999999963043 27,0.00999999994856 28,0.00999999999490 29,0.00999999999975 30,0.00999999999933 31,0.01000000000000 32,0.00999999999995 33,0.01000000000000 34,0.01000000000000 35,0.01000000000000	0.01000000	35

Table 2: Approximation to root for function ii)

Interval used	Approx. to root with step numbers	Root	Step
[0.6, 1]	... 25,0.67888664636013 26,0.67888664636024 27,0.67888664636174 28,0.67888664636162 29,0.67888664636182 30,0.67888664636181	0.67888665	30

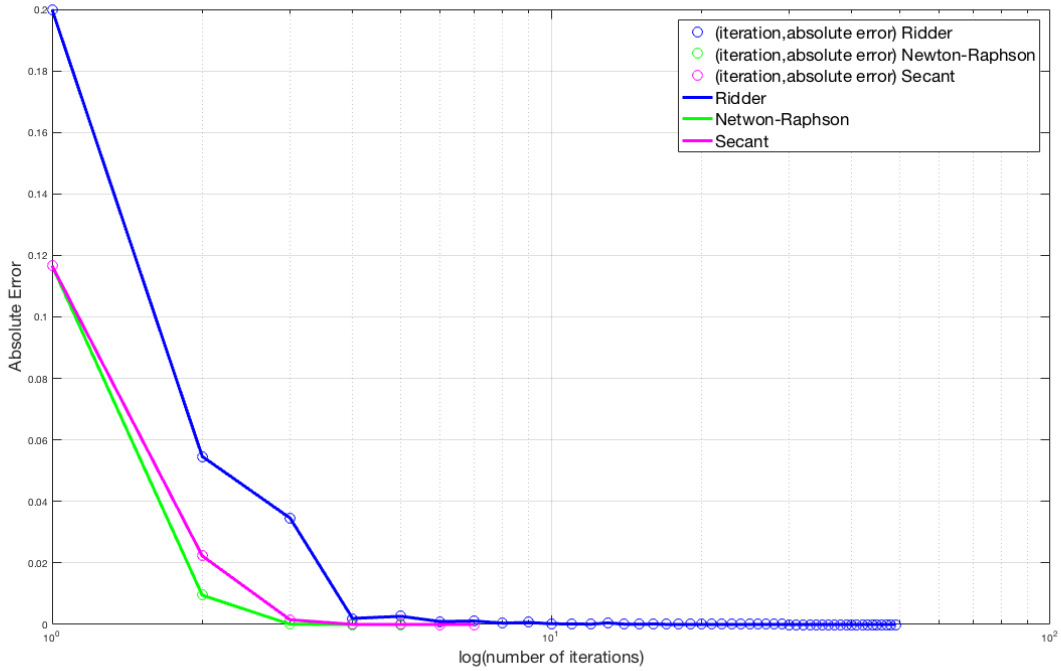
[3.1, 3.4]	... 29,3.22111843172671 30,3.22111843172658 31,3.22111843172680 32,3.22111843172677 33,3.22111843172679 34,3.22111843172679	3.22111843	34
[6.28, 6.3]	... 29,6.28690629096517 30,6.28690629096528 31,6.28690629096525 32,6.28690629096528 33,6.28690629096528	6.28690629	33

Table 3: Approximation of maximum of function iii). The derivative used to find this was: $g'(x) = -\frac{5x\left(e^{\frac{1}{x}}-1\right)-e^{\frac{1}{x}}}{x^7\left(e^{\frac{1}{x}}-1\right)^2}$. Note, the variable λ has been replaced by x.

Interval used	Approx. to root with step numbers	Root	Steps
[0.2, 0.21]	... 27,0.20140523527262 28,0.20140523527264 29,0.20140523527257 30,0.20140523527263 31,0.20140523527264 32,0.20140523527262	0.20140524	32

(c)

Figure 1: A semi-log plot for the absolute error vs the number of iterations until the root $f(x) = 0$ is reached for the Newton-Raphson, Secant and Ridder's method



Discussion of performance of the three curves seen above

If we look at the vertical grid lines from left to right, we find that the Newton-Raphson method and the Secant method have lower absolute errors than Ridder's method. It is also noticeable from the plot that the Secant and Newton-Raphson method have similar rates of convergences. In fact, it can be shown that the Newton-Raphson method converges at a quadratic rate, while the Secant method converges at a superlinear rate of the golden ratio, which is very fast, but not quite as fast as quadratic convergence (which is why the Newton-Raphson method converges quicker).[1] According to a document I found[2], Ridder's method also has a superlinear convergence rate. However, its rate of convergence is $\sqrt{2}$ which is less than the golden ratio for the secant method, which perhaps explains why it converges slower than the other two methods. Furthermore, it is apparent that the curve for Ridder's method is more 'jumpy' than the other two, and at one instance the absolute error actually increases. This may be due to the nature of the method, but I think rather that it must be a mistake in my code somewhere as it should never jump out the converging interval and hence the error should always be getting smaller.

Here is the code I used to plot the graph above.

```
function plotiterr()
%
%PLOTITERR is a simple function that quickly allows one to plot the
%absolute error vs the number of iterations for the Newton-Raphson, Secant
```

[1] This was researched under *https* : *//en.wikipedia.org/wiki/Secant*$_m$*ethod* and *https* : *//en.wikipedia.org/wiki/Secant*$_m$*ethod* and *https* : *//en.wikipedia.org/wiki/Rate*$_o f_c$*onvergence* [accessed 22. Nov 2017]

[2] http://www.it.uom.gr/teaching/linearalgebra/NumericalRecipiesInC/c9-2.pdf

```
%and Ridder methods for values generated using NewtonRap.m, Ridder.m and
%secant.m for the function f(x) = x^2 - 0.5
%
%NewtonRap
itN = 1:5;
abserrN = [0.11666666666667 0.00949612403101 0.00006375857461
    0.00000000287450 0.000000000000001];
%Ridder
itR = 1:49;
abserrR = [0.20000000000000 0.054625739351349 0.034553536280281
    0.001963684229780 0.002652880891652 0.000936477373437 0.001123794743522
    0.000424652580821 0.000672608598106 0.000217408197887 0.000187286425033
    0.000086164920593 0.000508801763576 0.000098020230939 0.000043139973123
    0.000163697790470 0.000035886883307 0.000003609299012 0.000003339888476
    0.000001471947272 0.000005649018741 0.000001233726138 0.000000117723854
    0.000000112971632 0.000000048716032 0.000000158028994 0.000000036655043
    0.000000006541954 0.000000003868910 0.000000000293794 0.000000000328142
    0.000000000129310 0.000000000248201 0.000000000072574 0.000000000042956
    0.000000000003250 0.000000000003638 0.000000000001432 0.000000000002730
    0.000000000000801 0.000000000000480 0.000000000000034 0.000000000000040
    0.000000000000015 0.000000000000027 0.000000000000008 0.000000000000006
    0.000000000000003 0.000000000000001];
%Secant
itS = 1:7;
abserrS = [0.11666666666667 0.02228464419476 0.00151428619039
    0.00002550938629 0.00000002685386 0.00000000000048 0.000000000000001];

%Plots and labels
semilogx(itR,abserrR,'bo',itN,abserrN,'go', itS, abserrS,'mo','Markersize'
    ,10), hold on
semilogx(itR,abserrR,'b',itN,abserrN,'g', itS, abserrS,'m','Linewidth',3)
xlabel('log(number of iterations)', 'Fontsize',17)
ylabel('Absolute Error','Fontsize',17)
legend({'(iteration,absolute error) Ridder','(iteration,absolute error)
    Newton-Raphson', '(iteration,absolute error) Secant', 'Ridder','Netwon-
    Raphson','Secant'},'Fontsize',17)
grid on
end
```

Question 2: Polynomial interpolation revisited

(a)

The following code was implemented to interpolate the function $f(x)$ for linearly spaced and Chebyshev points.

```
function polyinterpolation(f,a,b,N)
%
%POLYINTERPOLATION Uses the standard interpolation polynomial method to
%                  plot interpolated polynomials that approximate functions
%
%Usage:   y = polyinterpolation(f,N,xmax,xmin) returns a plot of two
%         interpolating polynomials, one of which uses Chebyshev points
%         and the other uses linearly space points, and the function that
%         is being interpolated.
```

```
%
%Inputs:  f = function handel to function that is to be interpolated
%         N = the number of data points
%         a = the maximum value of the data points to be interpolated
%         b = the minimum value of the data points to be interpolated

%Linearly spaced data points:
xl = linspace(a,b,N);
yl = f(xl); %Finding y data points
Vl = vander(xl); %Creating Vandermode matrix
cl = Vl\(yl'); %Solving for coefficents of interpolating polynomial
X = linspace(a,b,1000);
Yl = polyval(cl,X); %Evaluating the polynomial

%Chebyshev data points
for k = 1:N
    xc(k) = (a + b)/2 + 0.5*(b - a)*cos(((2*(k)-1)*pi)/(2*N));
end
yc = f(xc); %Finding y data points
Vc = vander(xc); %Creating Vandermode matrix
cc = Vc\(yc'); %Solving for coefficents of interpolating polynomial
Yc = polyval(cc,X); %Evaluating the polynomial

%Creates plot, legends and labels
figure()
plot(X,f(X),'k', 'Linewidth',3), hold on
plot(xl,yl,'ro', X,Yl,'b', 'Linewidth',3, 'Markersize',10), hold on
plot(xc,yc,'mx',X,Yc,'g','Linewidth',3, 'Markersize',10)
legend({'f(x)','Linearly spaced points','linearly spaced polynomial','
   Chebyshev points', 'Chebyshev polynomial'}, 'Fontsize',17);
xlabel('x', 'Fontsize',19);
ylabel('Functions','Fontsize', 19);
end
```

Here are some of the plots I made in order to analyse how the interpolating polynomials react to increases in N.

Figure 2: Plot of the Chebyshev and linearly spaced interpolating polynomials for N = 5, and $f(x)$

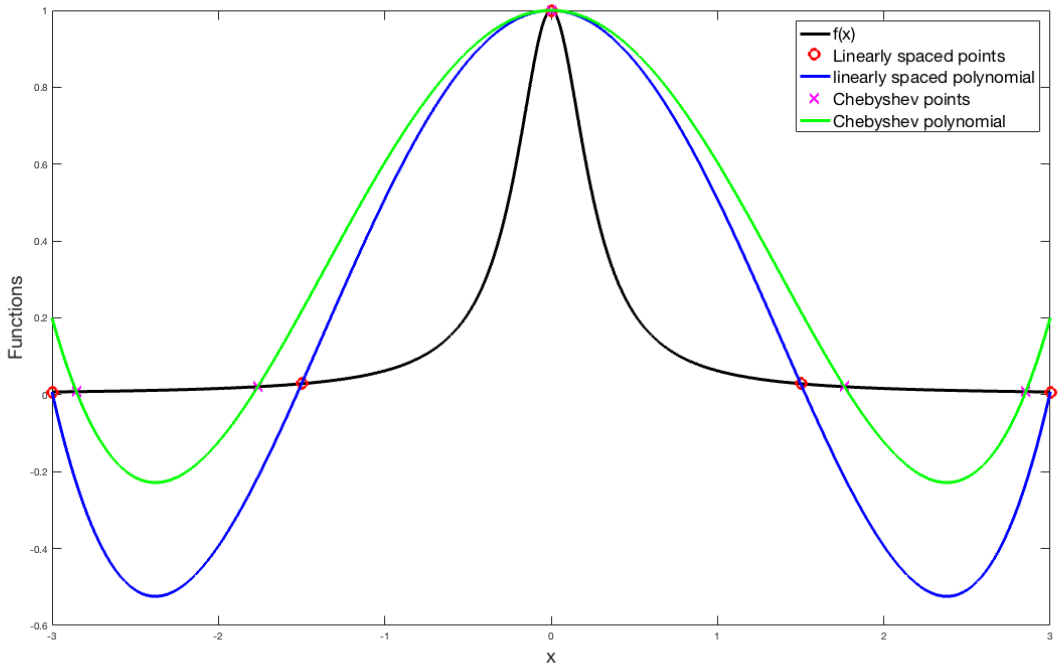


Figure 3: Plot of the Chebyshev and linearly spaced interpolating polynomials for N = 7, and $f(x)$

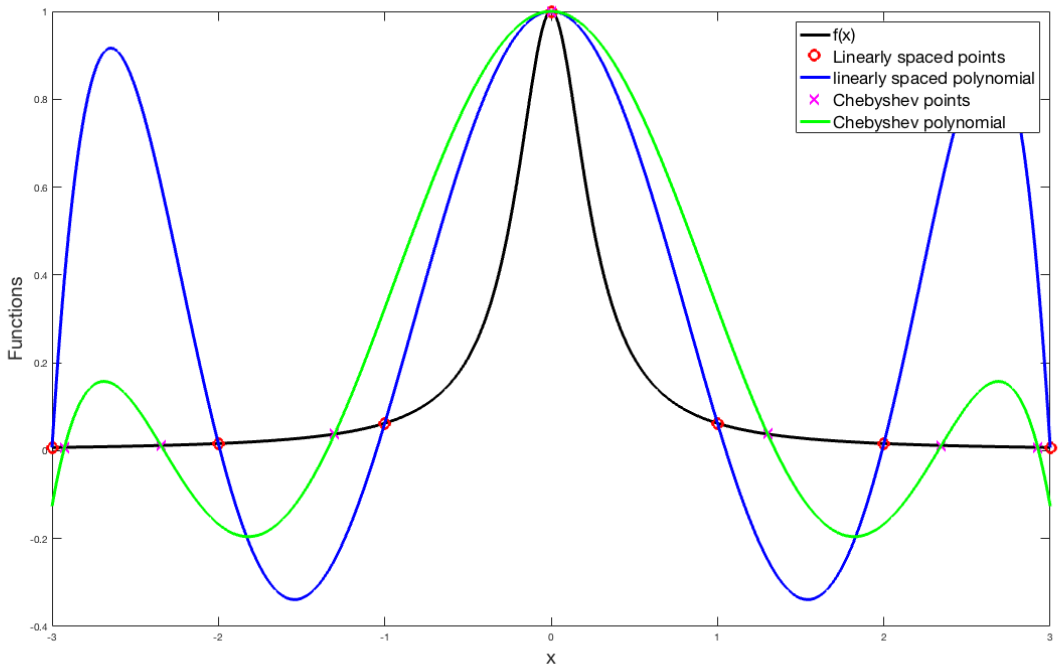


Figure 4: Plot of the Chebyshev and linearly spaced interpolating polynomials for N = 17, and $f(x)$

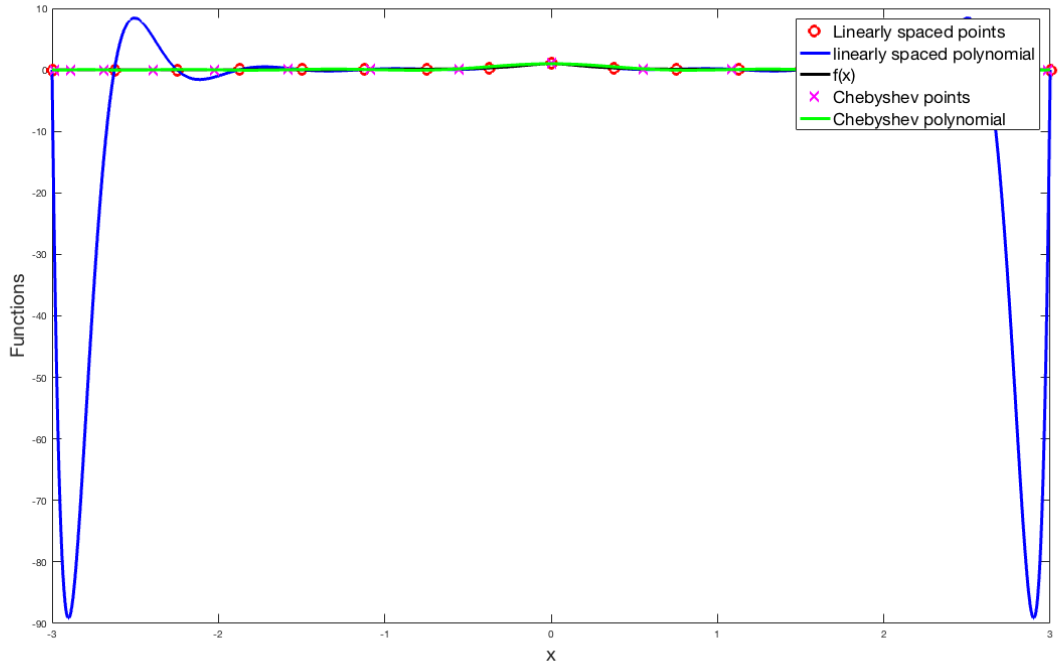
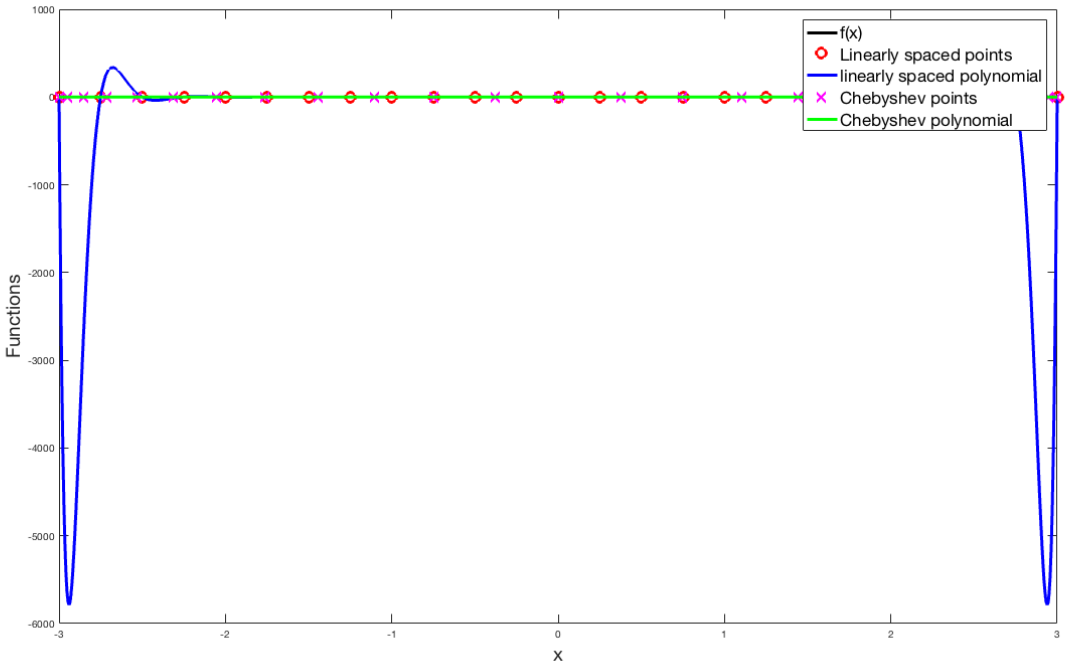


Figure 5: Plot of the Chebyshev and linearly spaced interpolating polynomials for N = 25, and $f(x)$



As we can see, as N increases, the linearly spaced polynomial has large divergences from the true function $f(x)$. At first it seems as if the Chebyshev and linearly spaced polynomials behave similarly, but as N increases the

Chebyshev polynomials becomes increasingly closer to the true function, whereas the linearly spaced polynomial begins to have larger divergences. In order to see this better, the following plots show just the Chebyshev polynomial for large N.

Figure 6: Plot of the Chebyshev interpolating polynomial for N = 25, and $f(x)$

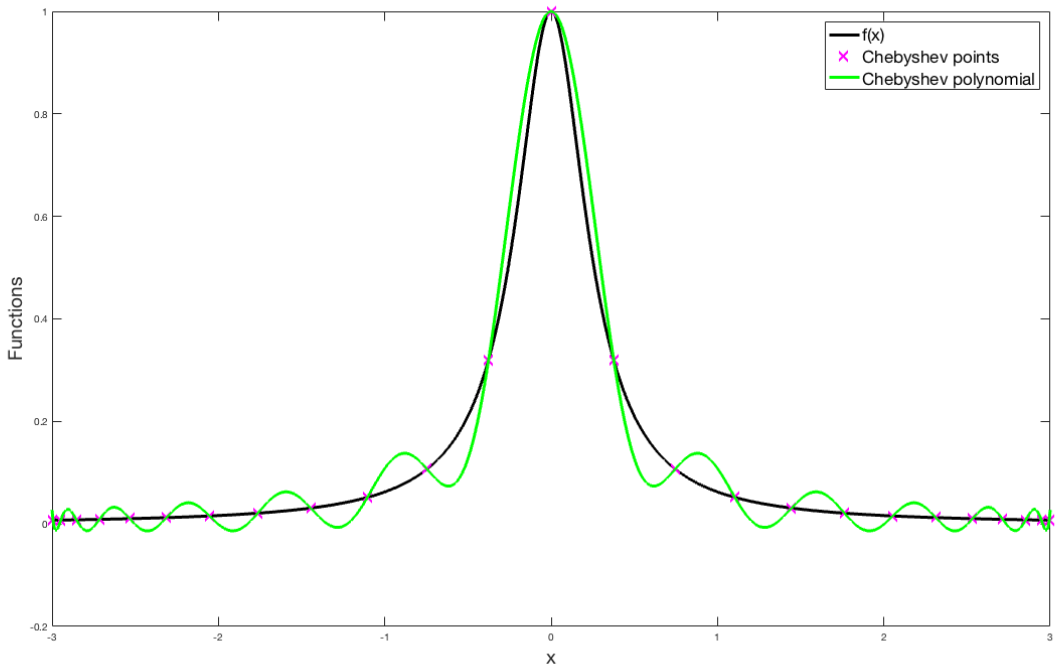
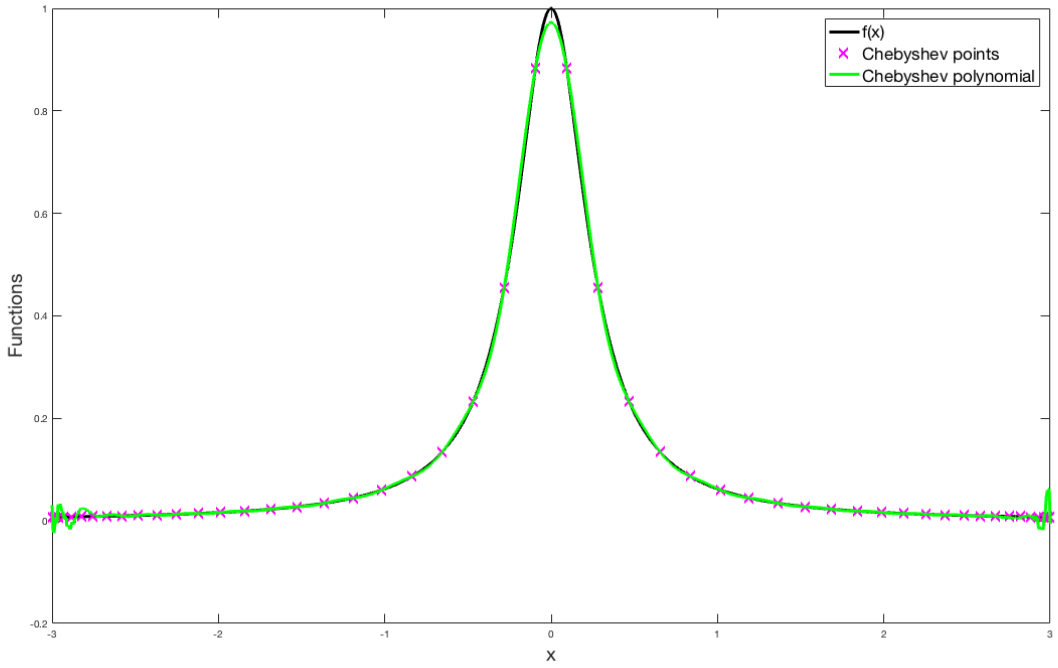


Figure 7: Plot of the Chebyshev interpolating polynomial for N = 50, and $f(x)$



Interestingly, if you increase N above around $N = 50$, the Chebyshev polynomials also start to have large divergences from f(x).
I surmise that the main difference between the two interpolating polynomials is that the Chebyshev polynomial diverges from the true function far slower (for increasing N) as the linearly spaced polynomial and is therefore a much more useful polynomial for approximating functions.

(b)

In order to show the equivalence of equation (1) and equation (2), I will start with equation (1) and end at equation (2). The key step in this derivation is shown by equation (4) below, where the derivative of the Lagrangian with respect to some $x = x_j$ is equivalent to the reciprocal of the weight w_i. Without this, this derivation would not be possible.

$$p(x) = \sum_{i=1}^{N} y_i L_i(x), \text{ where } L_i(x) = \frac{\prod_{j=1, j\neq i}^{N} (x - x_j)}{\prod_{j=1, j\neq i}^{N} (x_i - x_j)} \tag{1}$$

We can rewrite the numerator of the Lagrangian, $L_i(x)$, as,

$$L(x) = \frac{(x - x_0)(x - x_1)...(x - x_n)}{x - x_i} \tag{2}$$

Or better as,

$$l(x) = (x - x_0)(x - x_1)...(x - x_n) \tag{3}$$

divided by $(x - x_i)$. Now, since,

$$L'(x_i) = \frac{\mathrm{d}L(x)}{\mathrm{d}x}\Big|_{x=x_i} = \prod_{j=1, j\neq i}^{N} (x_i - x_j) = \frac{1}{w_i} \tag{4}$$

We can re-write $L_i(x)$ as,

$$L_i(x) = L(x)w_i = \frac{l(x)w_i}{x - x_i} \tag{5}$$

Thus, $p(x)$ from equation (1) becomes,

$$p(x) = l(x) \sum_{i=1}^{N} \frac{w_i}{x - x_i} y_i \tag{6}$$

Let us now consider the interpolation of a constant function $h(x) = 1$, which means that,

$$h(x) = l(x) \sum_{i=1}^{N} \frac{w_i}{x - x_i} \tag{7}$$

Since, $h(x) = 1$ and therefore $\frac{p(x)}{h(x)} = p(x)$, we arrive at the barycentric formula,

$$p(x) = \frac{\sum_{i=1}^{N} \frac{w_i}{(x - x_i)} y_i}{\sum_{i=1}^{N} \frac{w_i}{(x - x_i)}} \tag{8}$$

And thus, it has been shown that equation (1) is mathematically equivalent to equation (8) - the barycentric formula. The information gathered in order to do this equivalence derivation was found under *https* : *//en.wikipedia.org/wiki/Lagrange_polynomial* (accessed: Sun 5.Nov.2017) and *https* : *//people.maths.ox.ac.uk/trefethen/barycentric.pdf* (accessed: Sun 5.Nov.2017).

(c)

We have seen that equation (1) and equation (2) from the assignment are equivalent to each other, so I will only attempt to show that equation (1) is equivalent to the standard interpolating curve explained in the lecture.

The interpolating polynomial is a polynomial that is constructed by constraints of data points. It can be expressed as:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

Where the coefficients can be found by setting up the constraints into the matrix equation $\mathbf{M}\underline{a} = \underline{y}$, where $\mathbf{M}$ is also known as the Vandermonde matrix, like so:

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \dots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \dots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

where $(x_i, y_i), i = 0, ..., n$ are the given data points and $a_i, i = 0, ..., n$ are the coefficients. In order to solve this matrix, we need to find the inverse of the Vandermonde matrix and multiply it by the y column vector:

$$\begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \dots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \dots & x_n & 1 \end{bmatrix}^{-1} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

As it very beautifully turns out to be the case, the inverse of the Vandermonde matrix is [3]:

$$\begin{bmatrix} L_{0,0} & L_{1,0} & L_{2,0} & \cdots & L_{n,0} \\ L_{0,1} & L_{1,1} & L_{2,1} & \cdots & L_{n,1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{0,n} & L_{1,n} & L_{2,n} & \cdots & L_{n,n} \end{bmatrix}$$

where $L_{i,j}, i, j = 0, ..., n$ are the coefficients of the Lagrange polynomial. This can then be multiplied out with the y column vector and the coefficients a_0 to a_n can be found, showing indeed that $p(x) = \sum_{i=1}^{N} y_i L_i(x)$.

I myself wanted to check whether the mathematics above is correct and it is truly the case that the Lagrangian interpolation and standard form of $p(x)$ are equivalent, so I used Matlab to invert a 2 by 2 Vandermonde matrix and found p(x):

```
>> syms x0 x1
>> M = [x0 1;x1 1];
>> M^-1
ans =
[   1/(x0 - x1), -1/(x0 - x1)]
[ -x1/(x0 - x1), x0/(x0 - x1)]
```

$$\begin{aligned} \mathbf{M}^{-1} &= \frac{1}{x_0 - x_1} \begin{bmatrix} 1 & -1 \\ -x_1 & x_0 \end{bmatrix} \\ \Rightarrow \underline{a} &= \mathbf{M}^{-1} \underline{y} \\ \rightarrow a_0 &= \frac{y_0 - y_1}{x_0 - x_1} \\ \rightarrow a1 &= \frac{-x_1 y_0 + x_0 y_1}{x_0 - x_1} \end{aligned}$$

Plugging this into $p(x)$

$$\begin{aligned} p(x) &= a_1 x + a_0 \\ p(x) &= \frac{-x_1 y_0 + x_0 y_1}{x_0 - x_1} x + \frac{y_0 - y_1}{x_0 - x_1} \\ p(x) &= \frac{x - x_1}{x_0 - x_1} y_0 + \frac{x - x_0}{x_1 - x_0} \end{aligned}$$

Which is indeed, $p(x) = \sum_{i=2}^{1} y_i L_i(x)$

(d)

The following function was created to quickly generate the three Lagrangian polynomials, setting $x_1 = 1, x_2 = 2, x_3 = 4$.

```
function y = lagrangepoly(x1,x2,x3)

x = linspace(x1 - 2, x3 + 2, 100);
L1 = ((x - x2)./(x1 - x2)).*((x - x3)./(x1-x3));
L2 = ((x - x1)./(x2 - x1)).*((x - x3)./(x2-x3));
```

[3] This was found under *https : //math.stackexchange.com/questions/747357/how − to − obtain − lagrange − interpolation − formula − from − vandermondes − determinant* [accessed 17. No. 2017]

```
L3 = ((x - x1)./(x3 - x1)).*((x - x2)./(x3-x2));

plot(x,L1,'k',x,L2,'b',x,L3,'g','LineWidth',3);
legend({'L_1(x)','L_2(x)','L_3(x)'},'Fontsize', 17);
```

Figure 8: A plot of the three Lagrange Polynomials L_1, L_2 and L_3.

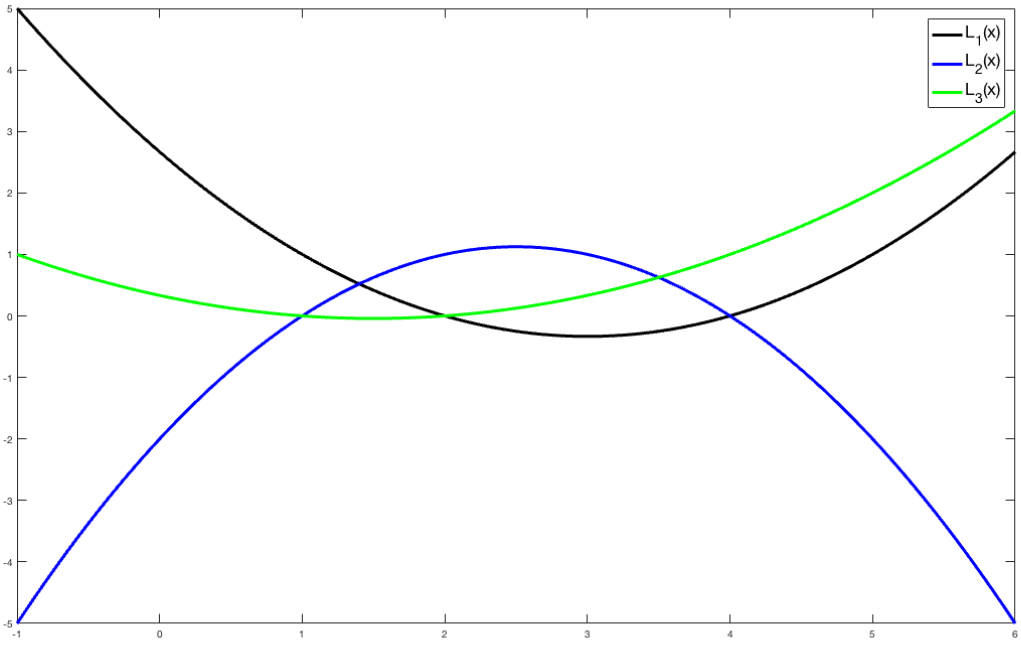


Table 4: Values of the above polynomials for $x = 1$, $x = 2$, $x = 4$

	x = 1	x = 2	x = 4
$L_1(x)$	1	0	0
$L_2(x)$	0	1	0
$L_3(x)$	0	0	1

(e)

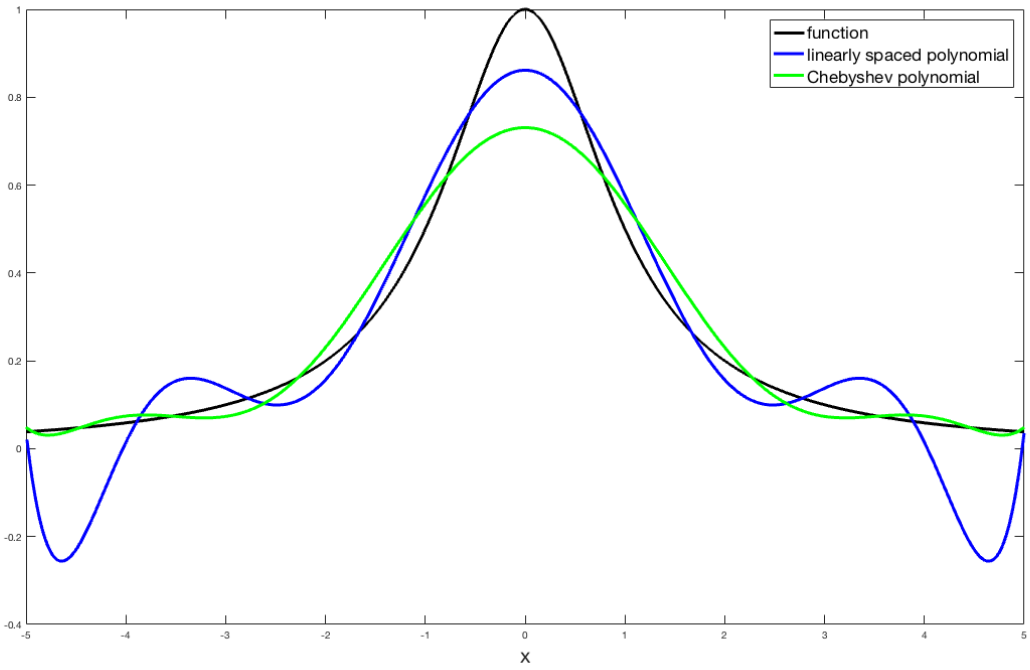
Code for polynomial interpolation using the *barycentric formula*

```
function barycentric(fun, x1,x2,N)
%
%BARYCENTRIC This function computes and plots a barycentric
%            polynomial interpolation of a funtion 'fun' between
%            an interval of upper limit x2 and lower limit x1 for
%            N data points using linearly spaced points and Chebyshev
%            points
%
%Inputs:     fun = function handel to function that should be interpolated
%            x1 = lower limit of interpolating window
%            x2 = upper limit of interpolating window
```

```
%             N = the number of data points of function 'fun' to be taken
%             into consideration for the interpolation
%
%Usage:       call barycentric(fun, x1,x2,N) in commandline using inputs

%Setting up the linearly spaced data points
X = linspace(x1,x2,N);
y = fun(X);
%Setting up Chebyshev x data points
for k = 1:N
    Xcv(k) = (x1 + x2)/2 + 0.5*(x2 - x1)*cos(((2*(k)-1)*pi)/(2*N));
end
ycv = feval(fun,Xcv);

%Setting up the barycentric formula
pTl = 0; % Upper sum for linearly spaced data points
pLl = 0; % Lower sum for linearly spaced data points
pTcv = 0; %Upper sum for Chebyshev data points
pLcv = 0; %Lower sum for Chebyshev data points

%Finding the polynomial interpolating polynomials
syms x
for i = 1:N
    wl(i) = ((-1)^(i-1))*nchoosek(N-1,i-1); %linearly spaced weights
    pTl = pTl + (wl(i)*y(i))./(x - X(i)); %Numerator of formula
    pLl = pLl + wl(i)./(x - X(i)); %Denominator of formula
    wcv(i) = ((-1)^(i-1))*sin(((2*i - 1)*pi)/(2*N)); %Chebyshev weights
    pTcv = pTcv + (wcv(i)*ycv(i))./(x - Xcv(i));%Numerator of formula
    pLcv = pLcv + wcv(i)./(x - Xcv(i));%Denominator of formula
end
pl = pTl/pLl; %p(x) for linearly spaced data points
pcv = pTcv/pLcv; %p(x) for Chebyshev data points

%Creating the plots
figure()
X = linspace(x1,x2,1000);
plot(X, feval(fun,X),'k','Linewidth',3), hold on
fplot(pl, [x1 x2], 'b', 'Linewidth',3), hold on
fplot(pcv,[x1 x2],'g','Linewidth',3)
legend({'function','linearly spaced polynomial','Chebyshev polynomial'},'
    Fontsize',17)
xlabel('x','Fontsiz',20)
end
```

Example of function call and output:

```
>> barycentric(@(x) 1./(1+x.^2), -5,5,10)
```



(f)

Using the code from part (e) I re-edited it so it can calculate the Lagrange interpolating polynomials p(x) for g(x) using the linearly-spaced and Chebyshev methods. As an example, the following is the interpolating polynomial it computed for the linearly spaced data points from $-1 : 1/8 : 1$.

```
p(x)_linearly_spaced = (1/(2*(x - 1)) + 1/(2*(x + 1)) + 1456/(x - 1/2) +
    1456/(x + 1/2) + 8286954461393137/(1099511627776*(x - 1/4)) +
    8286954461393137/(1099511627776*(x + 1/4)) + 384/(5*(x - 3/4)) + 384/(5*(
    x + 3/4)) - 11264/(x - 1/8) - 11264/(x + 1/8) -
    8421114371727023/(2199023255552*(x - 3/8)) -
    8421114371727023/(2199023255552*(x + 3/8)) - 35840/(89*(x - 5/8)) -
    35840/(89*(x + 5/8)) - 1024/(113*(x - 7/8)) - 1024/(113*(x + 7/8)) +
    12870/x)/(1/(x - 1) + 1/(x + 1) + 1820/(x - 1/2) + 1820/(x + 1/2) +
    8008/(x - 1/4) + 8008/(x + 1/4) + 120/(x - 3/4) + 120/(x + 3/4) - 11440/(
    x - 1/8) - 11440/(x + 1/8) - 4368/(x - 3/8) - 4368/(x + 3/8) - 560/(x -
    5/8) - 560/(x + 5/8) - 16/(x - 7/8) - 16/(x + 7/8) + 12870/x)
```

Here are the plots: I show two plots, because for the first plot one cannot distinguish between the different polynomials because they are too accurate.

Figure 9: Plot showing the data points and interpolating polynomials for the Chebyshev and linearly spaced method and g(x).

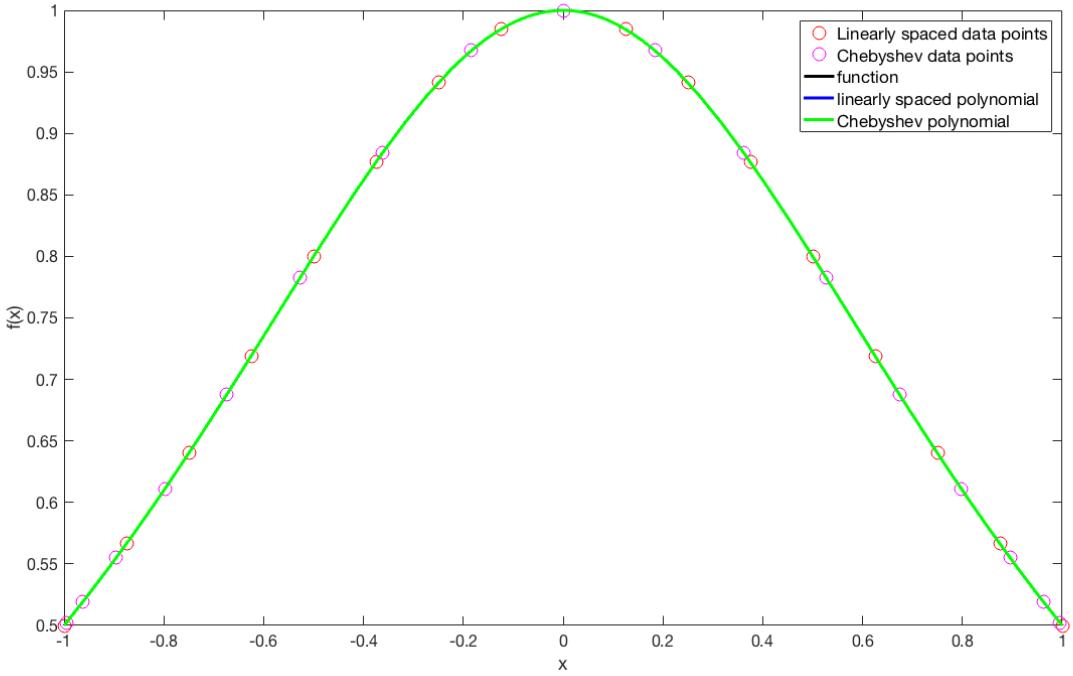
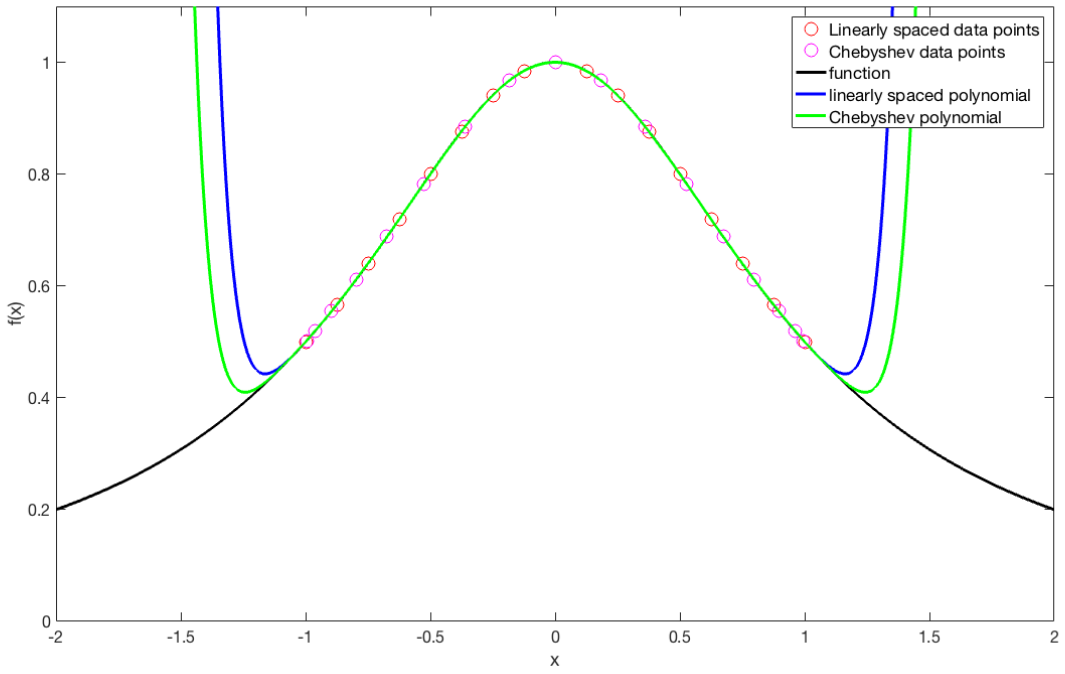


Figure 10: Plot showing the data points and interpolating polynomials for the Chebyshev and linearly spaced method and g(x) zoomed out so that one can distinguish between the different polynomials.



Question 3: Gaussian quadrature

(a)

The following code was implemented to calculate the coefficients of any nth-degree Legendre polynomial.

```
function [coeff] = LegendreCoeff(N)
%
%LEGENDRECOEFF This function finds the coefficients of any nth order
%              legendre polynomial. It does so by using the inbuilt matlab
%              commands legendreP, which calculates the legendre polynomials
%              from the recurrence relation described in the question, and
%              coeffs, which computes the coefficients of any polynomial.
%
%Usage:        LegendreCoeff(N) returns the cofficients from highest
%              to lowest degree of x of the nth order legendre polynomial
%
%Inputs:       N = order of of the lagrange polynomial to be evaluated

syms x; %create symbol
coeff = coeffs(legendreP(N,x), 'All'); %find all coefficients of legendre
   polynomial
end
```

(b)

The following code was implemented to find the roots and weights of the Legendre polynomials

```
function [x,w] = roots_weightsL(N)
%
%ROOTS_WEIGHTS finds the weights and roots of any nth order legendre
%              ploynomial.
%
%Usage:        roots_weightsL(N) returns only the roots of the legendre
%              polynomial
%              [x,w] = roots_weightsL(N) returns the roots and corresponding
%              weights of the legendre polynomial
%
%Inputs:       N = order of of the lagrange polynomial to be evaluated

%Using previously created function to find the coefficients
coeff = LegendreCoeff(N);

%Finding the roots using the roots command and then sorting in ascending
%order
x = sort(eval(roots(coeff)))';

%Finding weights using formula from lecture slides
w = 2*(1-(x.^2))./(((N + 1)^2).*legendreP(N+1, x).^2);

end
```

Table 5: Tabulated results for x_k and w_k for $n = 2, 3, 4$

n - order of Legendre polynomial	x_k	w_k
n = 2	-0.577350269189626	1
	0.577350269189626	1
n = 3	-0.774596669241483	0.555555555555555
	0	0.888888888888889
	0.774596669241483	0.555555555555555
n = 4	-0.861136311594053	0.347854845137454
	-0.339981043584856	0.652145154862546
	0.339981043584856	0.652145154862546
	0.861136311594053	0.347854845137454

(c)

We can use the Gaussian quadrature data points and weights for a general integral by linearly transforming it to fit the Gaussian integrals whose limits are -1 and 1. This transformation is shown in the following lines.
Aim:

$$\int_a^b f(x)dx \Rightarrow \int_{-1}^{1} f(x_g)dx_g \approx \sum_{i=1}^{n} w_i f(x_{g_i}) \tag{9}$$

where I am noting x_g as the 'Gaussian' variable whose domain is between -1 and 1.

Now, since x might have different domain as x_g, but they are both still subsets of the larger x domain over which the function is being integrated, we can assume that x is related to x_g by $x = a_1 + a_2 x_g$, where $\{a_1, a_2\}$ are constants. Thus, we can solve for the limits, like so:

$$b = a_1 + a_2(1)$$
$$a = a_1 + a_2(-1)$$

Solving for a_1 and a_2 we get,

$$a_1 = \frac{a+b}{2}$$
$$a_2 = \frac{b-a}{2}$$

Finally, to perform the conversion we also need to find dx in terms of dx_g,

$$x = \frac{a+b}{2} + \frac{b-a}{2}x_g$$
$$dx = \frac{b-a}{2}dx_g$$

Now substituting all the above into equation 9,

$$\int_a^b f(x)dx = \int_{-1}^{1} \left[f\left(\frac{a+b}{2} + \frac{b-a}{2}x_g\right)\right]\left(\frac{b-a}{2}\right)dx_g \approx \left(\frac{b-a}{2}\right)\sum_{i=1}^{n} f\left(\frac{a+b}{2} + \frac{b-a}{2}x_g\right)$$

and voilà, the transformation is complete and we can use Gaussian quadrature for any integral. The resources used to derive this were found under *https : //www.youtube.com/watch?v = yF − JlQxaZQs* [accessed 11. Nov 2017] and *https : //en.wikipedia.org/wiki/Gaussian_quadrature* [accessed 11. Nov 2017].

(d)

The following function was implemented to find a numerical approximation for any integral, between any limits, using Gaussian quadrature for $N = 2, 3, 4$ data points.

```
function I = gaussint(fun,a,b,N)
%
%GAUSSINT Uses the Gaussian quadrature method to numerically evaluate
%         definite integrals between the interval [a,b]
%
%Usage:   gaussint(fun,a,b,N) returns the approximated integral between the
%         interval [a,b] for a function 'fun' using 2,3 or 4
%         data points
%
%Inputs:  fun = function handel to function that is to be integrated
%         a = lower limit
%         b = upper limit
%         N = 2,3 or 4 (depending on the number of data points you wish to
%         use)

%Using previously implemented function (see above) to find the data points
%and weights
[xg,w] = roots_weightsL(N);

%Converting the function to be within the correct range so that Gaussian
%quadrature can be applied
gfun = fun((b+a)/2 + xg.*(b-a)/2)*((b-a)/2);

%Gaussian quadrature for N = 2,3 and 4
switch N
    case 2
        I = w(1)*gfun(1) + w(2)*gfun(2);
    case 3
        I = w(1)*gfun(1) + w(2)*gfun(2) + w(3)*gfun(3);
    case 4
        I = w(1)*gfun(1) + w(2)*gfun(2) + w(3)*gfun(3) + w(4)*gfun(4);
    otherwise
        fprintf('Sorry this function only evaluates integrals using 2,3 or 4
            data points\n')
end
end
```

(e)

The following code was implemented for Simpson's rule in order to compare the values of the integral $\frac{2}{\sqrt{\pi}} \int_0^{0.75} e^{-t^2} dt$ with the value of the integral determined by the code above using Gaussian quadrature.

```
function [int, err] = Simpson(f,min,max,n)
%
%SIMPSON Uses the composite Simpson's rule to find the definite integral
%        between an interval [min,max]
%
%Usage:  Simpson(f,min,max,n) returns the value of the integral of the
%        function f between the interval [min,max]
```

```
%         [int, err] = Simpson(f,min,max,n) returns the value of the
%         integral of the function f between the interval [min,max] and
%         gives the absolute error (here defined only for the integral
%         'erf' in the interval [0,0.75])
%Input: f = function handel
%       min = lower limit
%       max = upper limit
%       n = the number of panels (note n can only be even)
%
%Please note that the following code was inspired by the python code
%found on wikipedia under https://en.wikipedia.org/wiki/Simpson%27s_rule
%[accessed 12. Nov. 2017]

%Checking if n is odd
if mod(n,2)
    error('n must be even')
end

%Setting up step size and non-weighted Simpson functions
h = (max - min)/n;
s = f(min) + f(max);

%Adding to s all odd simpson functions
for i = 1:2:n
    s = s + 4*f(min + i*h);
end

%Adding to s all even simpson functions
for i = 2:2:(n-1)
     s = s + 2*f(min + i*h);
end

%Evaluating integral and error
int = s*(h/3);
err = abs(int - erf(0.75));
end
```

Table 6: **Comparing results for** $n = 2, 3, 4$

Method of integration used	Number of datapoints and their absolute error (all to 16 s.f)					
	n = 2	Error in n = 2	n = 3	Error in n = 3	n = 4	Error in n = 4
Gaussian Quadrature	0.7108642-66367136	0.00029136-72863792	0.71115744-0132970	0.00000180-64794552	0.7111556-31116894	0.0000000-025366215
Simpson's rule	0.7115901-84309294	0.00043455-06557788	Simpson's rule only works for even n		0.71118103-1636877	0.00002539-79833622

It is evident from the table that the Gaussian quadrature method is far more exact than the Simpson method. For $n = 4$ the error for the Gaussian method lies in the range of 10^{-9}, while for Simpson's rule it lies in the range of 10^{-4}, which is a vast difference. For $n = 2$, however, there is only a difference of around 10 in the error. I changed the Gaussian quadrature code to the following so it works for all $n \geq 2, 3, 4$ and compared how long it takes for Gaussian quadrature and the Simpson method to have approximately the same error.

```
function I = gaussint(fun,a,b,N)
%
%GAUSSINT Uses the Gaussian quadrature method to numerially evaluate
```

```
%          definite integrals between the interval [a,b]
%
%Usage:    gaussint(fun,a,b,N) returns the approximated integral between the
%          interval [a,b] for a function 'fun' using n datapoints
%
%Inputs:   fun = function handel to function that is to be integrated
%          a = lower limit
%          b = upper limit
%          N = number of data points you wish to use

%Using previously implemented function (see above) to find the data points
%and weights
[xg,w] = roots_weightsL(N);

%Converting the function to be within the correct range so that Gaussian
%quadrature can be applied
gfun = fun((b+a)/2 + xg.*(b-a)/2)*((b-a)/2);

I = 0;
for i = 1:N
     I = I + w(i)*gfun(i);
end
end
```

For an error in the range 10^{-16} the Gaussian quadrature method took $1.078s$ and needed only 10 data points. Simpson's rule on the other hand only took $0.031s$, but needed 1900 data points (panels) instead. I therefore think the main differences are that the results using the Gaussian method are more accurate, but less efficient than that of Simpson's rule for a certain number of data points.

(f)

The following code was used to plot the probability density function f for $k = 1, 2, 3, 4$

```
function chi_square()
%
%CHI_SQUARE plots the probability density function fk(x) for k = 1,2,3,4
%           using the chi-square probability distribution Xk^2
%
%Usage: Simply run chi_square()

%Setting up values for parameter k and choosing appropriate x values
k = [1 2 3 4];
max = 12;
x = linspace(0,max, max^3);

%Choosing plot colours
colour = 'rgbm';

for i = 1:length(k)
    if mod(k(i),2) %Checking if k is odd
        n = k(i)/2 - 1/2;
        %Using Gamma(n + 1/2) formula
        G = sqrt(pi)*factorial(2*n)/(2^(2*n)*factorial(n));
    else
        n = k(i)/2 - 1;
        %Using Gamma(n) formula
```

```
        G = factorial(n);
    end
    %Finding probability density function
    f = (x.^(k(i)/2 - 1).*exp(-x/2))./(2^(k(i)/2).*G);
    %Plotting functions for different k
    plot(x,f,colour(i), 'Linewidth', 3), hold on

end

%Labeling plots
axis([0 max 0 0.5])
legend({'k = 1','k = 2', 'k = 3', 'k = 4'}, 'Fontsize',17)
ylabel('f_k(x)','Fontsize',17)
xlabel('x','Fontsize',17)
end
```

Figure 11: Plot of probability density function f for $k = 1, 2, 3, 4$

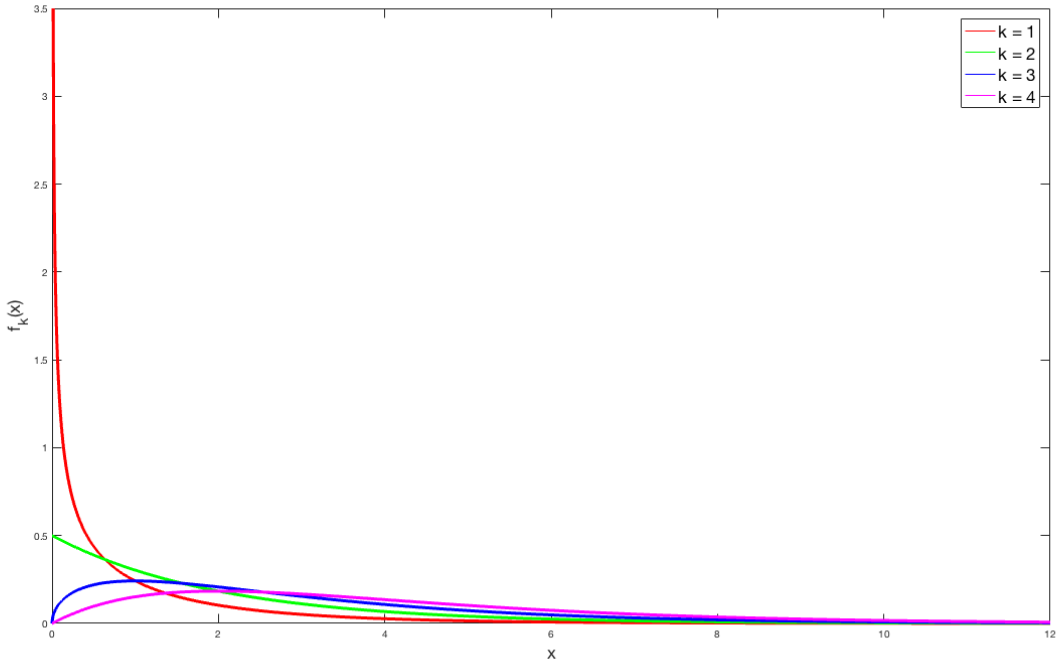
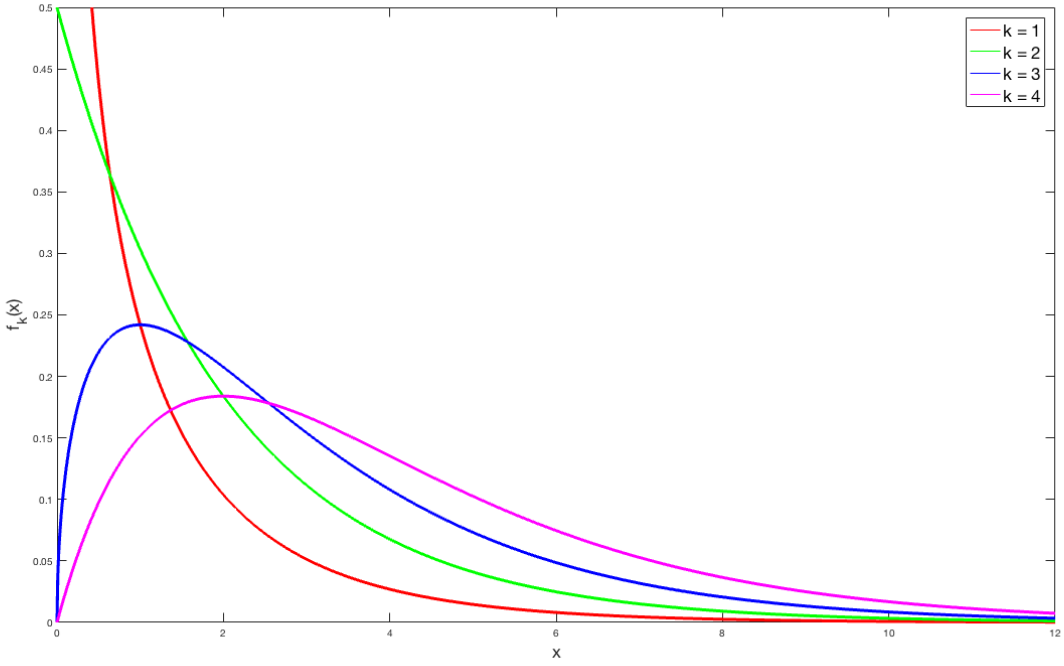


Figure 12: Zoomed-in plot of probability density function f for $k = 1, 2, 3, 4$ for more clarity



(g)

I believe the reason behind why the midpoint rule can better approximate $P(X > 3.3)$ compared to Simpson's rule and the trapezoidal rule, lies in their definitions. The midpoint rule is defined in the following way:

$$I = \int_a^b f(x)dx \approx \frac{b-a}{n} \left(f(m_1) + f(m_2) + ... + f(m_n)\right) \tag{10}$$

$$\text{where, } m_i = a + \frac{2i-1}{2n}(b-a) \tag{11}$$

Now, since $P(X > 3.3)$ is equivalent to $1 - \int_0^{3.3} p(x)dx$, we can apply the midpoint rule as above using the limits $a = 0$ and $b = 3.3$ - not forgetting that we are calculating $P(X > 3.3.)$ for $k = 1$, which means that $p(0) \to \infty$.

As we can see from equation 11, which provides the values of $f(m_i)$ in equation 10, we will never get $f(0)$, since $m_1 = 0 + \frac{2-1}{2n}(b-0) > 0$ and $\frac{b-a}{n} > 0$, which indicates that it can calculate the integral and find $P(X > 3.3)$.

For the Simpson and Trapezoidal rule, defined below respectively, we find that they are both dependent on $f(a)$ being evaluated, which means that they cannot perform the integral.

$$\text{Simpson's rule } \int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f(\frac{a+b}{2}) + f(b)\right) \tag{12}$$

$$\text{Trapezoidal Rule } \int_a^b f(x)\,dx \approx (b-a)\left[\frac{f(a)+f(b)}{2}\right] \tag{13}$$

It must also be said that all three of the methods fail if we try to approximate $P(x > 3.3)$ by evaluating the integrals from 3.3 to infinity, as they all depend on the upper limit b. Thus the only method of the three that

gives a useful result easily will be the midpoint method.

The above definitions were taken from http://www.math.pitt.edu/ sparling/23021/23022numapprox2/node4.html [accessed 15. Nov. 2017] and the lecture notes respectively.

(h)

In order to ascertain results that were suitable I did the following:

- Firstly, I checked online to find what $P(X > 3.3)$ should approximately be for $k = 1, 4$. Under *http* : *//www.statdistributions.com/chisquare/* [accessed 14. Nov 2017] I found that for $k = 1$, $P(X > 3.3) \approx 0.069$ and for $k = 4$, $P(X > 3.3) \approx 0.509$. This gave me an indication for the numbers I should be looking for.
- Secondly, I converted the code from question f to produce the function $f_k(x)$ for $k = 1, 4$ in terms of x using the *syms* function, which allowed me to use $f_k(x)$ in the command line.
- Thirdly, in the command line I then used the in-built *integral* function to evaluate the integrals from 0 to 3.3 and from 3.3 to inf to more accurately compare my results when I used Gaussian quadrature.
- Fourthly, I then applied the Gaussian quadrature method and found that I only got useful results when I computed the integrals from 0 to 3.3. When I tried the integration between the interval 3.3 to infinity it would just produce infinity or NaN as an answer. Furthermore, the more I increased the number of data-points the closer I got to the actual value of the integral, but it took more and more time to compute. The following shows the commands I used and the results I got for $k = 1$ and $k = 2$

For $k = 1$. Note, all results have been truncated to 6 s.f.

```
>> 1 - gaussint(@(x)  0.398942280401433*x.^(-1/2).*exp(-x/2), 0, 3.3, 50)
ans =

   0.081776
>> 1 - gaussint(@(x)  0.398942280401433*x.^(-1/2).*exp(-x/2), 0, 3.3, 100)
ans =

   0.075559
>> 1 - gaussint(@(x)  0.398942280401433*x.^(-1/2).*exp(-x/2), 0, 3.3, 130)

ans =

   0.074115
```

As we can see the value of the integral has changed drastically according to the number of data points I used. I could not get a result more accurate than the last shown command, because an error would occur due to the computation size being to large. Using the integral function the answer is:

```
>> 1 - integral(@(x)  0.398942280401433*x.^(-1/2).*exp(-x/2), 0, 3.3)

ans =

   0.069280
```

For $k = 4$. Note, all results have been truncated to 6 s.f.

```
>> 1 - gaussint(@(x) (x.*exp(-x./2))/4, 0, 3.3, 5)

ans =

   0.508932
>> 1 - gaussint(@(x) (x.*exp(-x./2))/4, 0, 3.3, 50)

ans =

   0.508932
```

As we can see above, not much has changed to the value of the integral when the number of data points was changed (you cannot see the change for 6 s.f.). It is also very close (or equal to 6 s.f.) to the value that the in-built *integral* command gives:

```
>> 1 - integral(@(x) (x.*exp(-x./2))/4, 0, 3.3)

ans =

   0.508932
```

In summary, I think the reason why the Gaussian quadrature method finds it so difficult to compute the integral when $k = 1$ is because $f_1(x)$ converges to infinity near 0.

(i)

I think the in-built *integral* command is very appropriate for calculating $P(X > 3.3)$, because it can calculate indefinite integrals as well as definite integrals. This is shown by the following:

```
>> integral(@(x) (x.*exp(-x./2))/4, 3.3, inf)

ans =

   0.508932257844998
```

is identical to

```
>> 1 - integral(@(x) (x.*exp(-x./2))/4, 0, 3.3)

ans =

   0.508932257844999
```

Here I have computed the integrals of the functions in the command line directly, but one could also call function from an m. file, like so:

```
function y = k1(x)
y = (x.*exp(-x./2))/4
end

>> integral(@(x) k1)/4, 3.3, inf)

ans =

   0.508932257844998
```

Question 4: Runga-Kutta methods

(a)

All Runge-Kutta methods can be expressed in the following form:

$$Y_1 = y_n, \tag{14}$$
$$Y_2 = y_n + ha_{2,1}f(t_n + c_1h, Y_1), \tag{15}$$
$$Y_3 = y_n + h\left[a_{3,1}f(t_n + c_1h, Y_1) + a_{3,2}f(t_n + c_2h, Y_2)\right] \tag{16}$$
$$\vdots \tag{17}$$
$$Y_s = y_n + h\left[a_{s,1}f(t_n + c_1h, Y_1) + a_{s,2}f(t_n + c_2h, Y_2) + ... + a_{s,s-1}f(t_n + c_{s-1}, Y_{s-1})\right] \tag{18}$$

where, the coefficients $\{a_{i,j}, c\}$ are given and define the particular method in question and $h = t_{n+1} - t_n$. The iterative form of the method is then as follows,

$$y_{n+1} = y_n + h\left[b_1f(t_n, Y_1) + b_2f(t_n + c_2h, Y_2) + ... + b_{s-1}f(t_n + c_{s-1}, Y_{s-1}) + b_sf(t_n + c_sh, Y_s)\right] \tag{19}$$

where $\{b_i\}$ is known as the "weight". We can formulate Equations 18 and 19 in a more a succinct manner like so,

$$Y_i = y_n + h\sum_{j=1}^{i-1} a_{i,j}f(t_n + c_jh, Yj), i = 1, ..., s \tag{20}$$

$$y_{n+1} = y_n + h\sum_{j=1}^{s} b_jf(t_n + c_j, Y_j) \tag{21}$$

Now I am in the position to explain what a Butcher tableau is. All that a Butcher tableau does is allow for a visual representation of the coefficients $\{c, b, a\}$ that define the Runge-Kutta method in question. For as shown above, the method depends entirely on these three coefficients. The Butcher tableau provides a quick means by which one can see whether the method is explicit or implicit and which order the method has and the number of computation stages it requires. The following shows what a Butcher tableau looks like for an implicit and explicit method, respectively.

$$\begin{array}{c|cccc} c_1 & a_{1,1} & a_{1,2} & \cdots & a_{1,s} \\ c_2 & a_{2,1} & a_{2,2} & \cdots & a_{2,s} \\ \vdots & \cdots & \cdots & & \cdots \\ c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{2,1} & & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array}$$

The information above was researched using the following sources: Numerical Solution of Ordinary Differential Equations by Kendall Atkinson, Weimin Han, David E. Stewart [accessed on google books 6.Nov.2017], John Butcher's tutorials presentation found online under https://www.math.auckland.ac.nz/butcher/ODE-book-2008/Tutorials/RK-methods.pdf [accessed 6.Nov.2017] and on Wikipedia under Runge-Kutta methods [accessed 6.Nov.2017].

(b)

The following code shows the m-file of the differential equation asked for in this question.

```
function dydt = df(y,t)
%DF Simple function of the differential equation in question 4b)

dydt = -(y.*t)./(1+(t.^2));

end
```

The following code was created in order to find the values of y(1) for each of the four methods asked for.

```
function [yK4_1,yK2_1,yMM_1,yRK3_8,yReal] = RK4_RK2_RK3_8()

%RK4_RK2_RK3_8 First order differential equation solver for the RK4,
%              RK2 (Huen and midpoint method) and RK3/8 Method for the
%              differential equation df.m.
%
%Usage:        RK4_RK2_RK3_8
%
%Output        RK4_RK2_RK3_8 when run will plot a graph of each of the
%              methods and give the approximation of the Runge-Kutta method
%              at y(1) of the ode df.m
%              [yK4_1,yK2_1,yMM_1,yRK3_8,yReal] = RK4_RK2_RK3_8 will give
%              the value of y(1) for each method of the ode df.m

%Number of steps
N = 10;

%Initial conditions
h = 0.1;
t4 = zeros(size(N)); y4 = zeros(size(N));
t4(1) = 0; y4(1) = 1;
n = 1;

%Real solution
t = 0:h:N;
ysol = (1+ (t.^2)).^(-0.5);
figure()
plot(t,ysol,'k','Linewidth',3)
hold on

%RK4 Method - The following code was inspired by information found
%under https://www.youtube.com/watch?v=PPwUaxTp8Uk,
%and https://uk.mathworks.com/videos/solving-odes-in-mat
%lab-3-classical-runge-kutta-ode4-117528.html

while t4 < N
    t4(n+1) = t4(n) + h;
    k1 = df(y4(n),t4(n));
    k2 = df(y4(n) + 0.5*h*k1, t4(n) + 0.5*h);
    k3 = df(y4(n) + 0.5*h*k2, t4(n) + 0.5*h);
    k4 = df(y4(n) + h*k3, t4(n) + h);
    T4 = (1/6)*(k1 + 2*k2 + 2*k3 + k4);
    y4(n+1) = y4(n) + T4*h;
    n = n+1;
end
```

```
plot(t4,y4,'b', 'Linewidth',3)
hold on

%RK2 - Heun's Method - The following code was inspired by information found
%under http://www.mymathlib.com/diffeq/runge-kutta/heuns_method.html
t2 = zeros(size(N)); y2 = zeros(size(N));
t2(1) = 0; y2(1) = 1;
n = 1;
while t2 < N
    t2(n+1) = t2(n) + h;
    k1 = df(y2(n),t2(n));
    k2 = df(y2(n) + h*k1, t2(n) + h);
    T2 = 0.5*(k1+k2);
    y2(n+1) = y2(n) + h*T2;
    n = n+1;
end
plot(t2,y2,'g','Linewidth',3)

%RK2 - Midpoint Method he following code was inspired by information found
%under https://uk.mathworks.com/videos/solving-odes-in-matlab-2-midpoint-
%method-ode2-117527.html
tMM = zeros(size(N)); yMM = zeros(size(N));
tMM(1) = 0;
yMM(1) = 1;
n = 1;

while tMM < N
      tMM(n+1) = tMM(n) + h;
      k1 = df(yMM(n), tMM(n));
      k2 = df(yMM(n) + 0.5*h, tMM(n) + 0.5*h*k1);
      yMM(n+1) = yMM(n) + h*k2;
      n = n+1;
end
plot(tMM,yMM,'r','Linewidth',3)

%RK3/8 - he following code was inspired by information found
%under http://www.mymathlib.com/diffeq/runge-kutta/runge_kutta_3_8.html
tRK3_8 = zeros(size(N));
yRK3_8 = zeros(size(N));
tRK3_8(1) = 0;
yRK3_8(1) = 1;
n = 1;

while tRK3_8 < N
    tRK3_8(n+1) = tRK3_8(n) + h;
    k1 = h*df(yRK3_8(n),tRK3_8(n));
    k2 = h*df(yRK3_8(n) +h/3, tRK3_8(n) + k1/3);
    k3 = h*df(yRK3_8(n) +2*h/3, tRK3_8(n) - k1/3 + k2);
    k4 = h*df(yRK3_8(n) +h, tRK3_8(n) + k1 - k2 + k3);
    TRK3_8 = (k1 + 3*k2 + 3*k3 + k4);
    yRK3_8(n+1) = yRK3_8(n) + TRK3_8;
    n = n+1;
end
plot(tRK3_8,yRK3_8,'k','Linewidth',3)

```

```
%Evaluates y(1) for each of the methods
yRK3_8 = yRK3_8(11);
yK4_1 = y4(11);
yK2_1 = y2(11);
yMM_1 = yMM(11);
yReal = ysol(11);
end
```

Table 7: Values of y(1) for each of the numerical methods

	Runge-Kutta method (RK4)	Midpoint-Method	Heun's Method	Kutta's 3/8-rule	True value
y(1) to 15s.f	0.707106593980605	0.711396716007072	0.707739933164713	-0.001551664381526	0.707106781186547

(c)

In order to find the global truncation error for the four methods, I re-edited the code in such a way that the function shown above in part (b) now takes the step-size h as an input and collects the errors in an array, like so:

```
function [GErrRK4,GErrH,GErrRK2,GErrRK38] = RK4_RK2_RK3_8(h)
```

The global truncation errors for each method were found by using the following commands beneath each method:

```
GErrRK4 = abs(y4(1/h + 1) - ysol)/h;
GErrH = abs(y2(1/h + 1) - ysol)/h;
GErrRK2 = abs(yMM(1/h + 1) - ysol)/h;
GErrRK38 = abs(yRK3_8(1/h + 1) - ysol)/h;
```

where *ysol* is my own analytical solution to ode at $y(1)$:

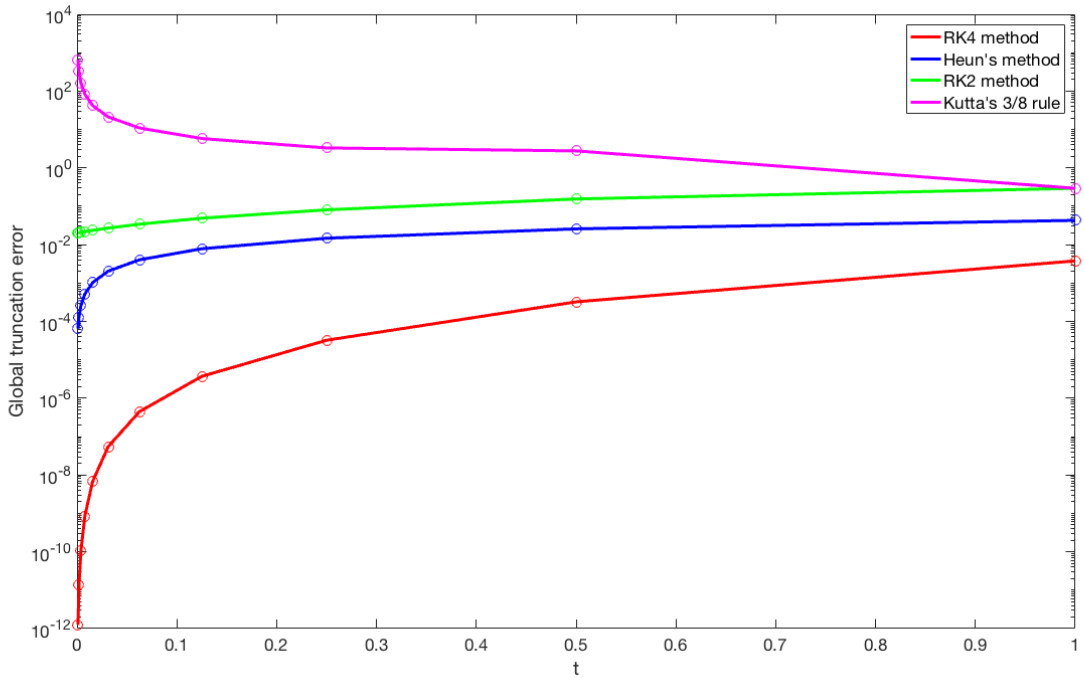
```
%Real solution
ysol = (1+ (1.^2)).^(-0.5);
```

I then created a different function that allowed me to plot the global truncation errors against various step-sizes:

```
function plotGlobalErr()
%
%PLOTGLOBALERR Plots the global tuncation error of the four methods used in
%              question 4(b) against the step-size h.
%
%Usage:        Simply run plotGlobalErr(). Note, by using the comment
%              command one can decide which error for each method to plot.
%

%Various step sizes - starting at 1 and then halving until 2^-10
h = 2.^(-(0:10));

%Creating zero arrays to help with computating speed
GErrRK4 = zeros(length(h));
GErrH = zeros(length(h));
GErrRK2 = zeros(length(h));
GErrRK38 = zeros(length(h));

%Finding the global truncation errors for various h
for j = 1:length(h)
```

```
    [GErrRK4(j),GErrH(j),GErrRK2(j),GErrRK38(j)] = RK4_RK2_RK3_8(h(j));
end

%Plotting the graph and labelling. Note I am using semi-plot because the
%errors differ largely for the different methods.
figure()
semilogy(h, GErrRK4,'ro', 'Markersize', 10), hold on
RK4 = semilogy(h, GErrRK4,'r', 'Linewidth',3); hold on
semilogy(h, GErrH, 'bo', 'Markersize', 10), hold on
H = semilogy(h, GErrH,'b', 'Linewidth',3); hold on
semilogy(h, GErrRK2,'go','Markersize', 10), hold on
RK2 = semilogy(h, GErrRK2,'g','Linewidth', 3); hold on
semilogy(h, GErrRK38,'mo','Markersize', 10), hold on
RK38 = semilogy(h, GErrRK38,'m','Linewidth', 3); hold on
xlabel('t', 'Fontsize',19)
ylabel('Global truncation error', 'Fontsize',19)
lgd = legend([RK4(1) H(1) RK2(1) RK38(1)],'RK4 method', 'Heun''s method', '
   RK2 method','Kutta''s 3/8 rule');
lgd.FontSize = 17;
end
```

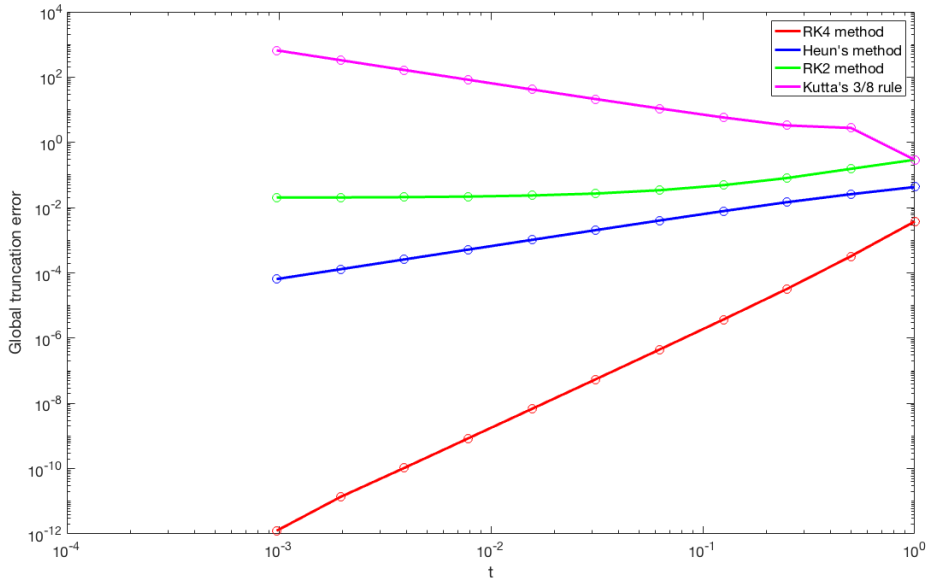
And here is the result:



Discussion

The results show that the global truncation error for the fourth-order Runga-Kutta method decreases rapidly as the step-size decreases, going from an error of around 10^{-4} to 10^{-12}. For Heun's method the global truncation error also decreases, but not at such a fast rate. The RK2 method seems to almost remain constant when the step-size decreases, and rather interestingly, the global truncation error for Kutta's 3/8 rule actually increases - (which is making me doubt whether my code for this method is correct). Below is a loglog plot showing the rates at which the global truncation error is decreasing (or increasing) in a clearer fashion. If these results are true, then it clearly indicates that the RK4 method is the most accurate method, as its error decreases the most

rapidly with decreases in step-size.



Question 5: Rock-scissor-paper

(a)

To show that $x + y + z$ is a conserved quantity, we must show that $\frac{d(x+y+z)}{dt} = 0$.
The ODE's in matrix form are:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} xy - xz \\ yz - yx \\ zx - zy \end{bmatrix} \tag{22}$$

Thus,

$$\frac{d(x+y+z)}{dt} = \frac{dx}{dt} + \frac{dy}{dt} + \frac{dz}{dt} = xy - xz + yz - yx + zx - zy = 0$$

Hence $x + y + z$ is a conserved quantity.

(b)

To show that xyz is a conserved quantity, we must show that $\frac{d(xyz)}{dt} = 0$. Since,

$$\frac{d(xyz)}{dt} = \frac{d(x)}{dt}yz + \frac{d(y)}{dt}xz + \frac{d(z)}{dt}yx \tag{23}$$

Now plugging in the values from equation 22,

$$\begin{aligned} \frac{d(xyz)}{dt} &= [xy - xz]\, yz + [yz - yx]\, xz + [zx - zy]\, yx \\ &= (xy^2z - zy^2x) + (xyz^2 - yxz^2) + (zyx^2 - yx^2z) = 0 \end{aligned}$$

Hence xyz is a conserved quantity.

(c)

The following code was created to solve the dynamical system of first order differential equations that describe a rock-scissors-paper game using the Euler scheme with a step-size $h = 0.02$

```
function y = S_odeEuler()
%
%S_ODEEULER is a runnable function that plots the dynamical system of
%           differential equations given by equations 3,4 and 5 in question
%           five of the assignment using the Euler scheme.
%
%Usage:     Simply run S_odeEuler()


%Setting conditions
tf = 250; %final time
h = 0.02; %step size
N = tf/h; %number of intervals from t = 0 to t = tf
t(1) = 0; %t at t = 0
x(1) = 0.5; %x at t = 0
y(1) = 0.3; %y at t = 0
z(1) = 0.2; %z at t = 0

%Generates the solutions
for n = 1:N
    t(n+1) = t(n) + h;
    x(n+1) = x(n) + h*Xf(x(n),y(n),z(n));
    y(n+1) = y(n) + h*Yf(x(n),y(n),z(n));
    z(n+1) = z(n) + h*Zf(x(n),y(n),z(n));
end

%Plotting and creating pretty labels
figure()
plot(t,x,'m',t,y, 'g',t,z,'r','Linewidth',2)
legend({'x - "rock"','y - "scissors"' ,'z - "paper"'},'Fontsize',17)
xlabel({'time'}, 'Fontsize', 17)
ylabel({'Solutions to the dynamical system of Rock-Scissors-Paper'},'
   Fontsize',17)
end

%First ode (equation 3)
function dxdt = Xf(x,y,z)
    dxdt = x*y - x*z;
end

%Second ode (equation 4)
function dydt = Yf(x,y,z)
    dydt = y*z - y*x;
end

%Third ode (equation 5)
function dzdt = Zf(x,y,z)
    dzdt = z*x - z*y;
end
```

Figure 13: A plot of the solutions x, y, z of the system of differential equations against time for a step-size of $h = 0.02$ and medium large $t = 250$

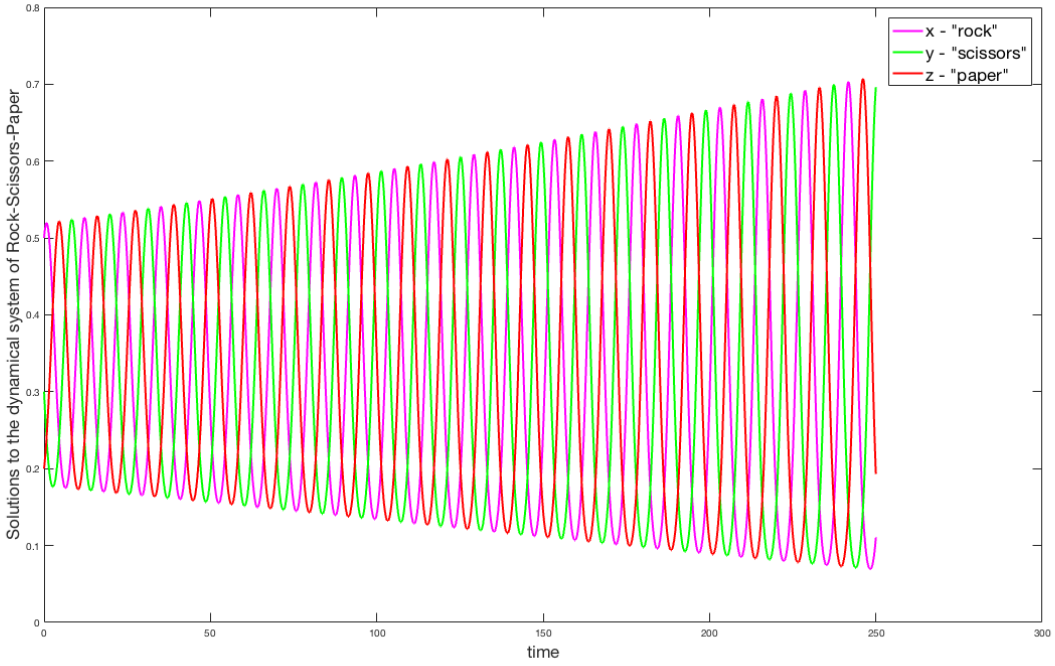
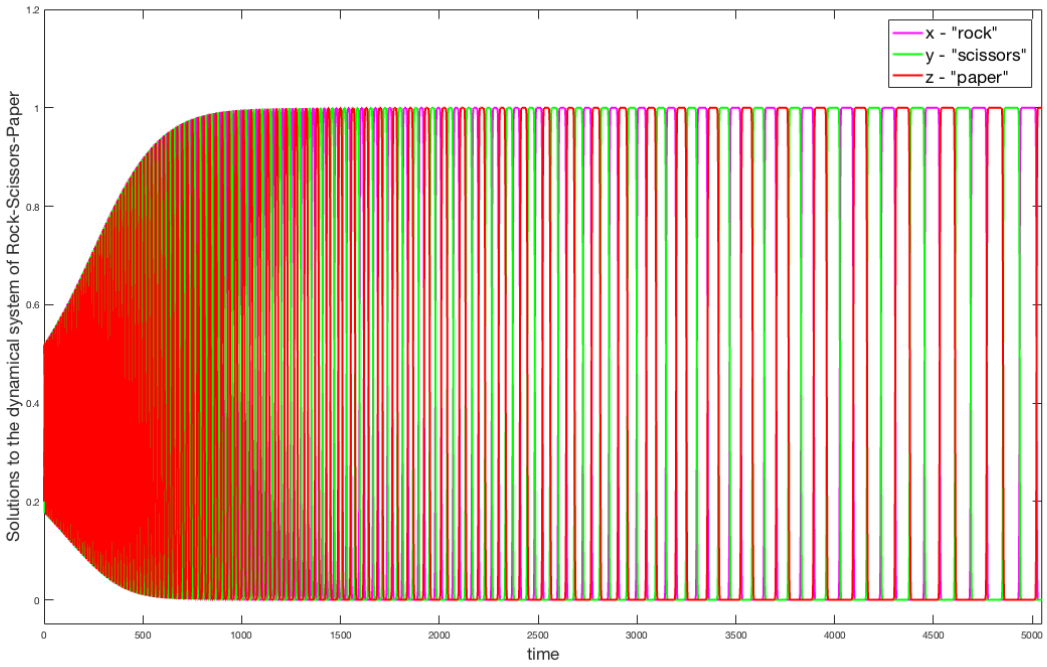


Figure 14: A plot of the solutions x, y, z of the system of differential equations against time for a step-size of $h = 0.02$ and very large $t = 5050$



Observations: We do indeed see everlasting oscillations as t increases. At first these oscillations grow at a steady rate, and then remain at at constant amplitude throughout most of the time interval. Furthermore, it seems to be the case that the oscillations begin to spread out more and more as t increases. I think this graph confirms the mathematical prediction because at first the three types of rock-scissors-paper have different strengths (rock = 0.5, scissors = 0.3, paper = 0.2), so they are constantly beating each other until they reach a sort of equilibrium state, where each of them have the same strength.

(d)

The following code is essentially the same as above, but adjusted to also be able to plot the conserved quantities $x + y + z$ and xyz.

```
function y = S_odeEuler()
%
%S_ODEEULER is a runnable function that plots the dynamical system of
%           differential equations given by equations 3,4 and 5 in question
%           five of the assignment using the Euler scheme and the
%           conserved quantities x + y + z and xyz.
%
%Usage:     Simply run S_odeEuler()


%Setting conditions
tf = 5050; %final time
h = 0.02; %step size
N = tf/h; %number of intervals from t = 0 to t = tf
t(1) = 0; %t at t = 0
x(1) = 0.5; %x at t = 0
y(1) = 0.3; %y at t = 0
z(1) = 0.2; %z at t = 0
c(1) = 1; %c is the sum of x y and z
m(1) = 0.5*0.3*0.2; %m is the multiple of xyz
%Generates the solutions
for n = 1:N
    t(n+1) = t(n) + h;
    x(n+1) = x(n) + h*Xf(x(n),y(n),z(n));
    y(n+1) = y(n) + h*Yf(x(n),y(n),z(n));
    z(n+1) = z(n) + h*Zf(x(n),y(n),z(n));
    c(n+1) = x(n) + y(n) + z(n); %Evaluating c for all x, y and z
    m(n+1) = x(n)*y(n)*z(n); %Evaluating m for all x,y and z
end

%Plotting and creating pretty labels
figure()
plot(t,x,'m',t,y, 'g',t,z,'r',t,c,'k',t,m,'y','Linewidth',2)
legend({'x - "rock"','y - "scissors"' ,'z - "paper"','x + y + z', 'xyz'},'
   Fontsize',17)
xlabel({'time'}, 'Fontsize', 17)
ylabel({'Solutions to the dynamical system of Rock-Scissors-Paper'},'
   Fontsize',17)
axis([0 tf -0.005 1.2])
end

%First ode (equation 3)
function dxdt = Xf(x,y,z)
```

```
    dxdt = x*y - x*z;
end

%Second ode (equation 4)
function dydt = Yf(x,y,z)
    dydt = y*z - y*x;
end

%Third ode (equation 5)
function dzdt = Zf(x,y,z)
    dzdt = z*x - z*y;
end
```

Figure 15: A plot of the conservative quantity $x + y + z$ for a very large t = 5050

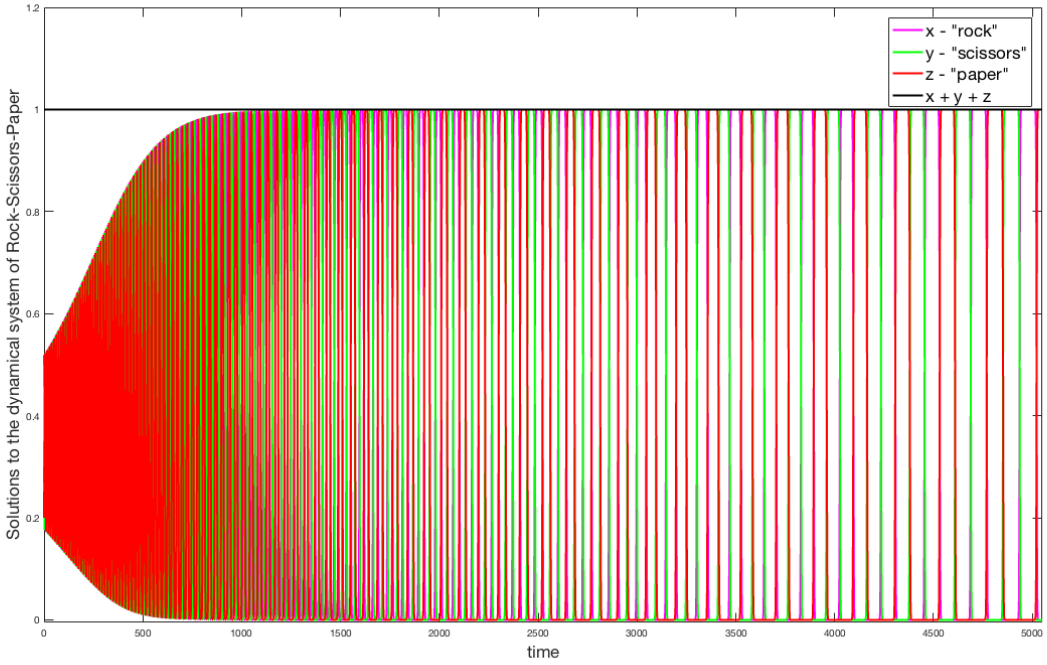


Figure 16: A plot of the conservative quantity $x + y + z$ for a very large t = 5050 without solutions to the dynamical system

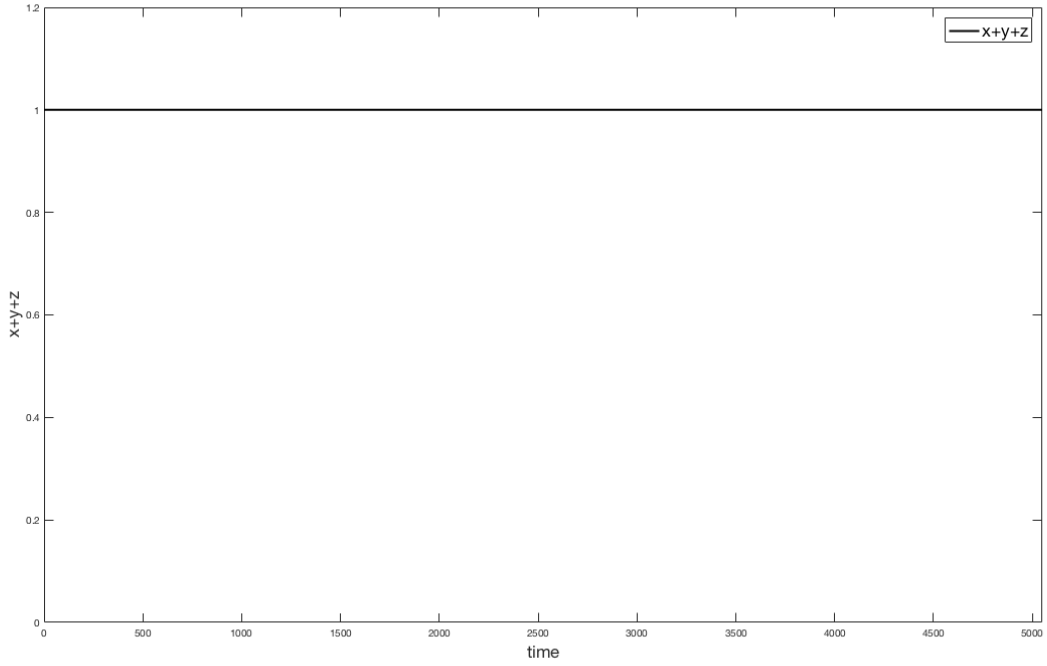


Figure 17: A plot of the conservative quantity xyz for a very large t = 5050

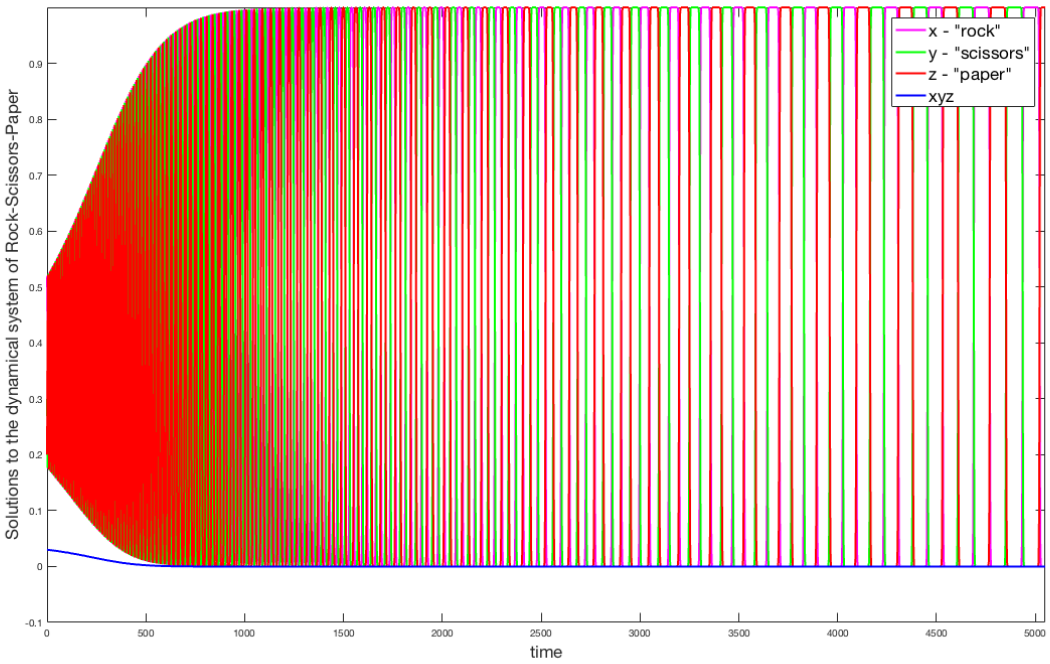
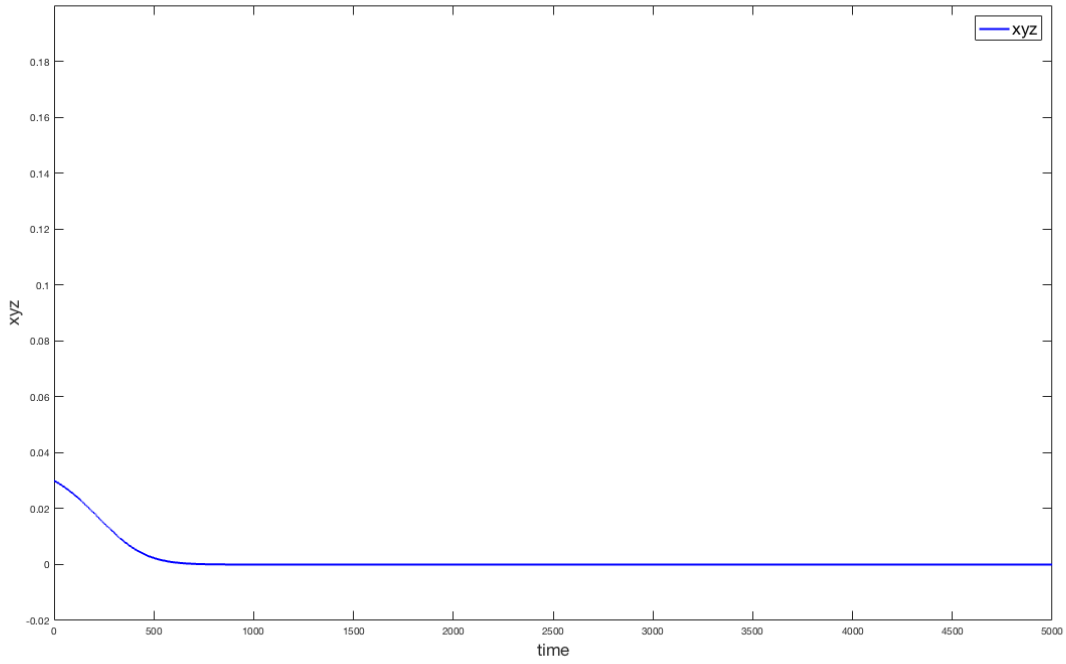


Figure 18: A plot of the conservative quantity xyz for a very large t = 5050 without solutions to the dynamical system



Observations We notice indeed from figures 15 and 16 that $x + y + z$ remains constant or conserved for all t, which is consistent with the mathematical results obtained in question (a). The figures 17 and 18, however, do not confirm that xyz is a conserved quantity, because it is changing with respect to time for $t \leq 500$. For $t \geq 500$ it remains constant. I have tried solving the system analytically, but no explicit solution exists, so I am not sure whether this is just a fault in using the Euler method or I have made an error somewhere in the code. Perhaps its because the step size is too large and the solution is consequently too inaccurate.

(e)

All the following figures were plotted using a step-size of $h = 0.002$ using the code from the previous question.

Figure 19: A plot of the solutions x, y, z of the system of differential equations against time for a step-size of $h = 0.002$ and medium large $t = 500$

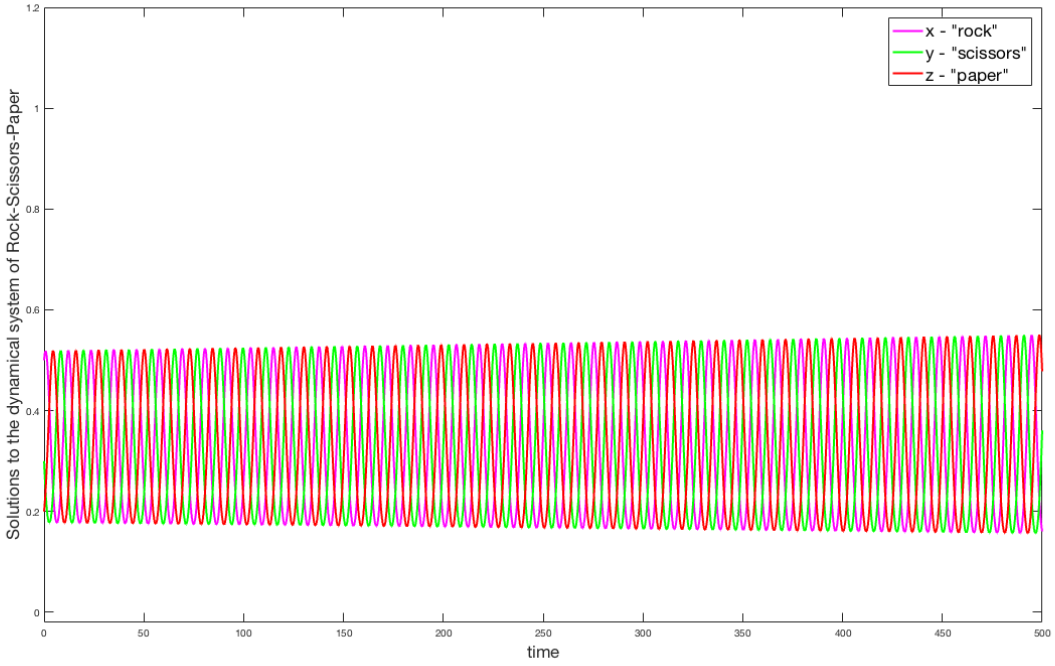


Figure 20: A plot of the solutions x, y, z of the system of differential equations against time for a step-size of $h = 0.002$ and very large $t = 5000$

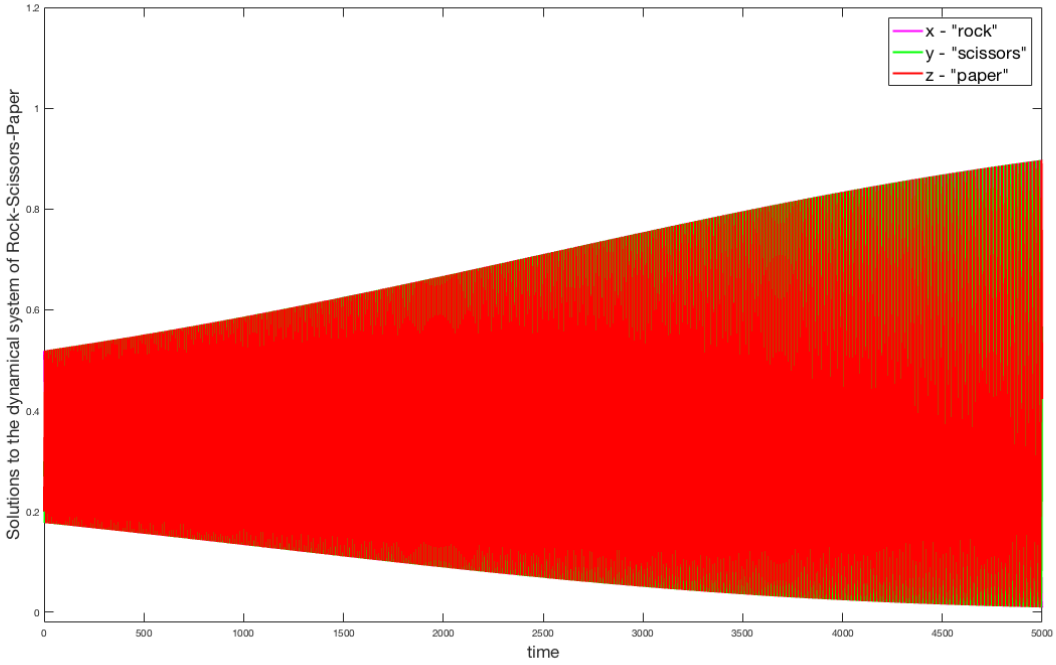


Figure 21: A plot of the conservative quantity $x + y + z$ for a very large t = 5050

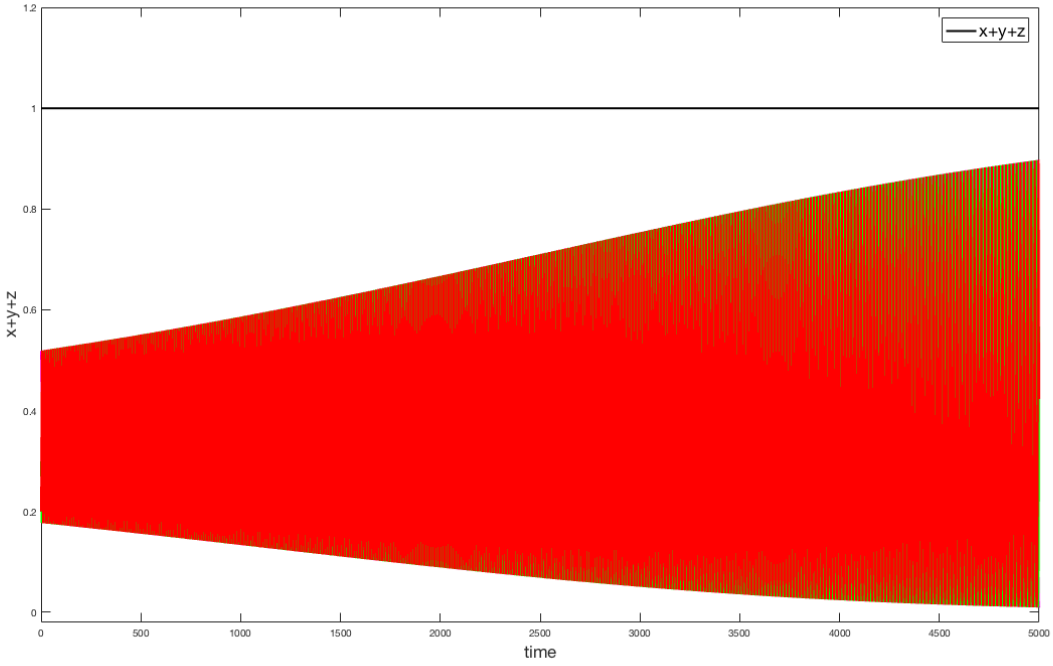


Figure 22: A plot of the conservative quantity $x + y + z$ for a very large t = 5050 without solutions to the dynamical system

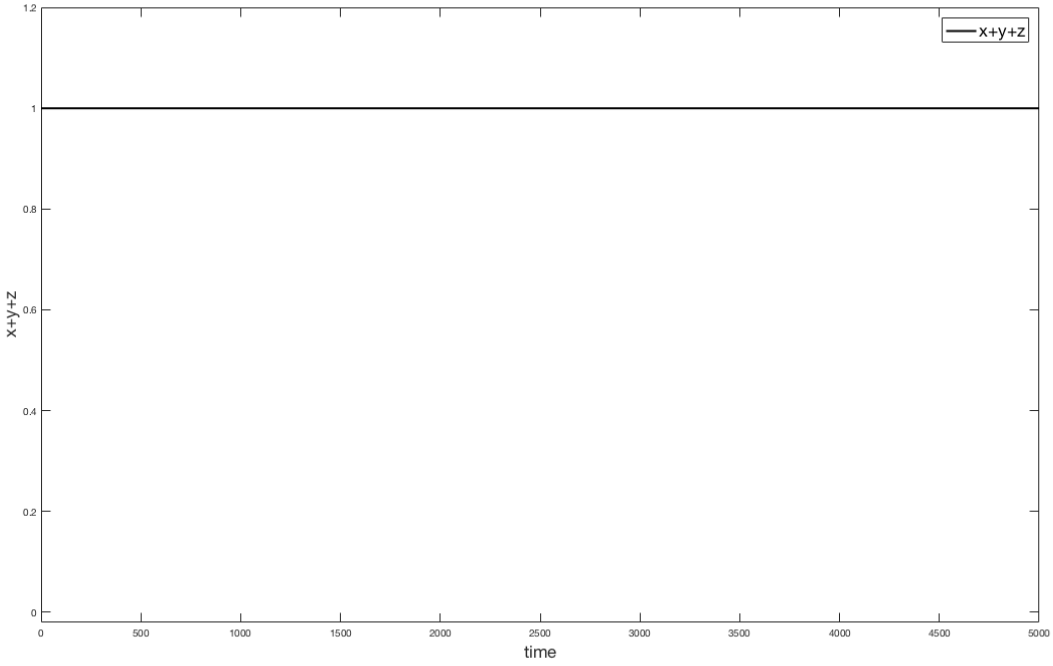


Figure 23: A plot of the conservative quantity xyz for a very large t = 5050

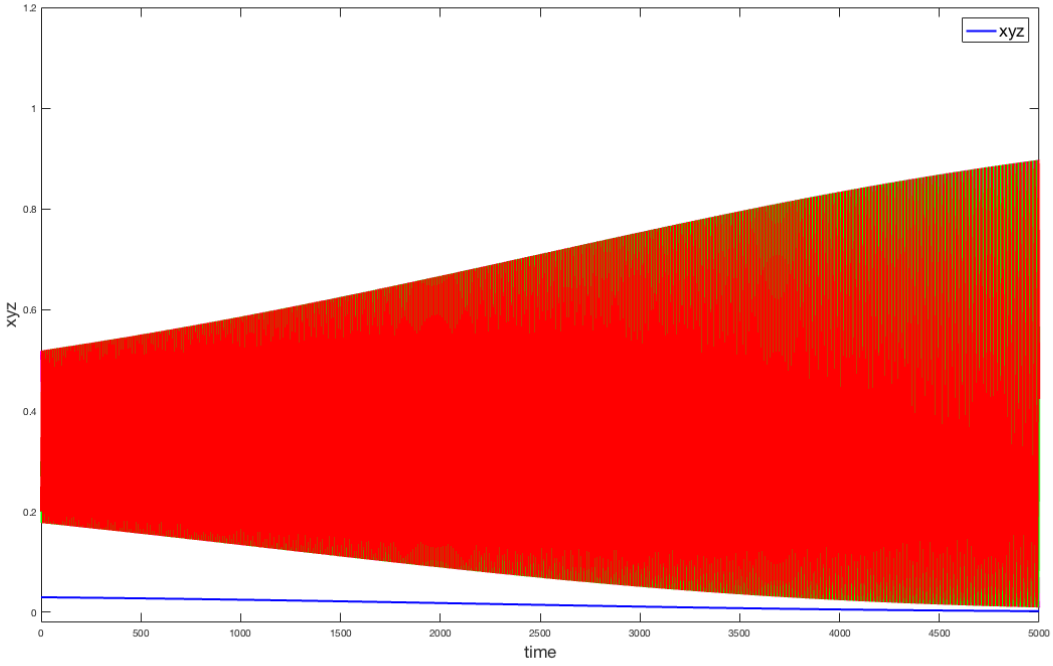
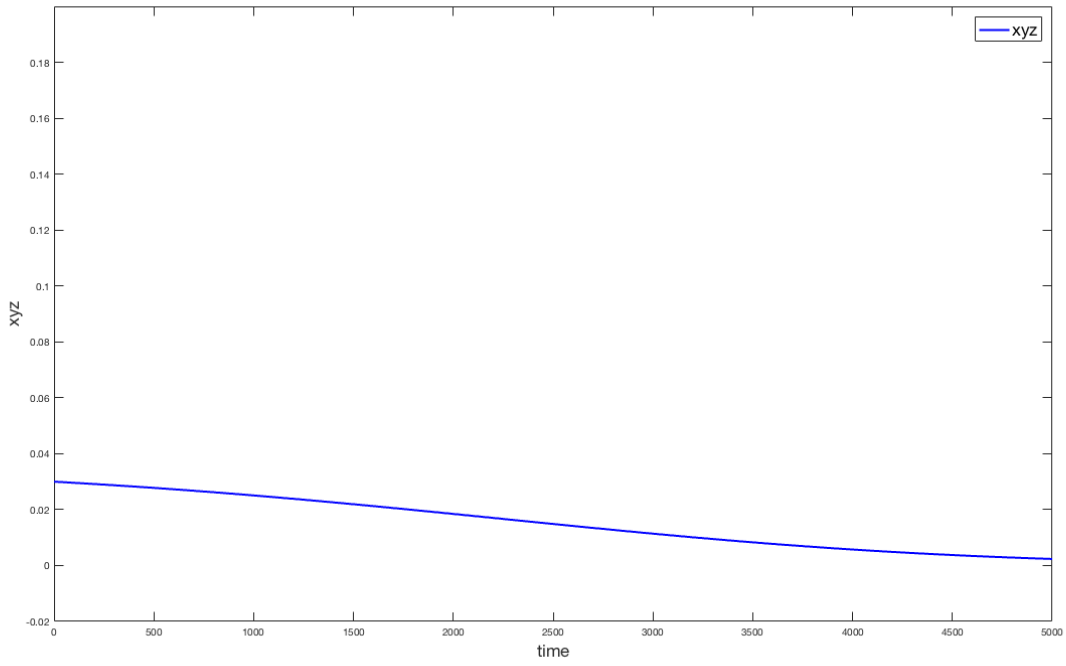


Figure 24: A plot of the conservative quantity xyz for a very large t = 5050 without solutions to the dynamical system

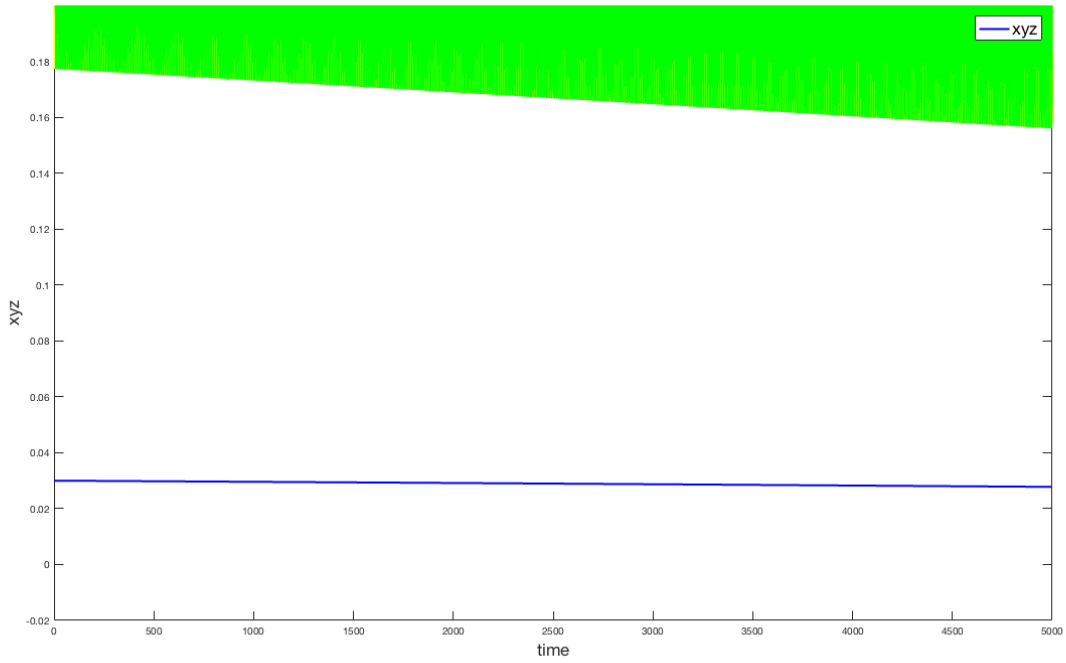


Differences in results using $h = 0.002$ and $h = 0.02$

When the step-size is decreased to $h = 0.002$ the results show a more gradual increase in the oscillations for large t and hardly any increase in oscillations for small t. Again, $x + y + z$ remains conserved and when I first plotted the quantity xyz it looked as if it too is conserved (Figure 23), but on closer analysis it does actually seem to change with time (Figure 24), which is rather odd. I think the reason for this has to with the magnitude of t, because if t is small enough, the oscillations remain at a constant size (Figure 19) and xyz is conserved.

On a side note, the smaller h is, the closer we should be getting to the true solution, so I plotted another graph for $h = 0.0002$ and indeed xyz looks more conserved. This also gives us an indication to what the true solution should look like: Indefinite oscillations of constant amplitude.

Figure 25: An additional plot of the conservative quantity xyz for a very large t = 5000 with solutions to the dynamical system in green for an even smaller step-size of $h = 0.0002$, which shows better that xyz is actually a conserved quantity



(f)

The following code was implemented to solve the system of differential equations using ode45 instead of the Euler scheme as above.

```
function Sode45()
%
%SODE45  is runnable function that solves the dynamical system of
%        differential equations presentented by equations 3,4 and 5 in the
%        coursework assignment using ode45
%
%Usage:  Simply run Sode45(). If you want to run the conservative
%        quantities xyz and x + y + z too, or any sort of permutation of
%        conservative quantity and solutions, comment and un-comment as
%        neccessary

%Set max time
```

```
tmax = 50000;

%Setting up system of differential equations. This method was inspired by
%https://www3.nd.edu/~nancy/Math20750/Demos/3dplots/dim3system.html
%[accessed 10. Nov. 2017]
f = @(t,x) [x(1)*x(2)-x(1)*x(3);x(2)*x(3)-x(2)*x(1);x(3)*x(1)-x(3)*x(2)];

%Solving the system of odes using ode45
[t,x] = ode45(f,[0 tmax],[0.5 0.3 0.2]);

%Plotting the solutions and labels
plot(t,x(:,1), 'm', t, x(:,2),'g', t, x(:,3),'r'), hold on
plot(t,(x(:,2) + x(:,1)+ x(:,3)), 'k'), hold on %x + y + z
plot(t,(x(:,2) .* x(:,1) .* x(:,3))) %xyz
xlabel({'time'}, 'Fontsize', 19)
ylabel({'Solution to the system of ODEs using ode45'}, 'FontSize', 19)

end
```

Figure 26: Solution of the system of differential equations using ode45 up until $t = 500$

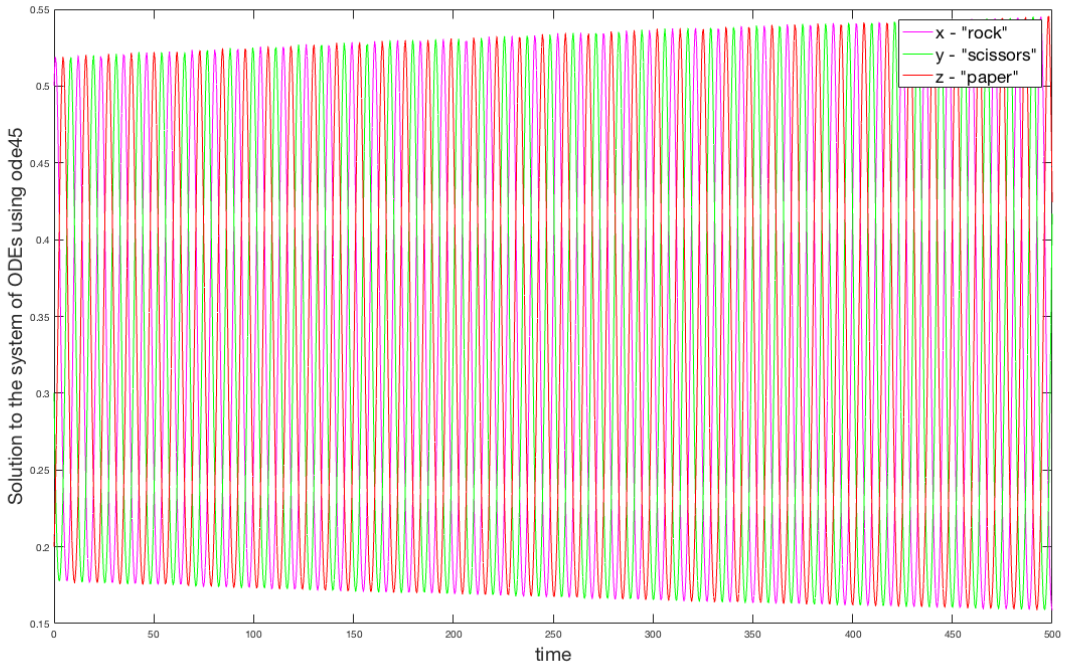


Figure 27: Solution of the system of differential equations using ode45 up until $t = 5000$

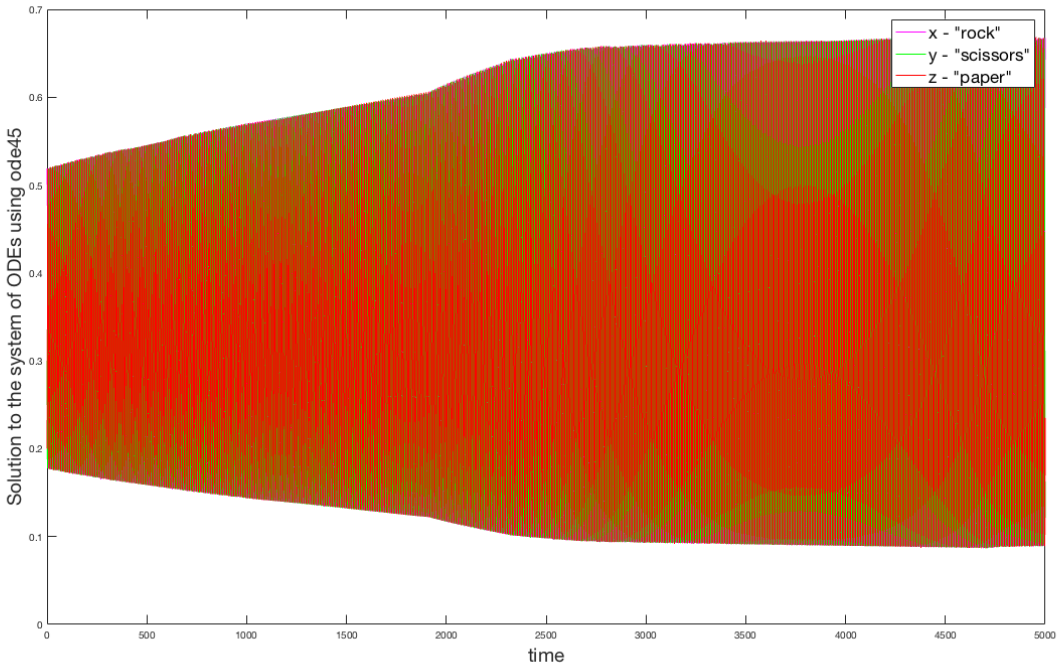


Figure 28: Solution of the system of differential equations using ode45 up until $t = 5000$ and conserved quantity x + y + z

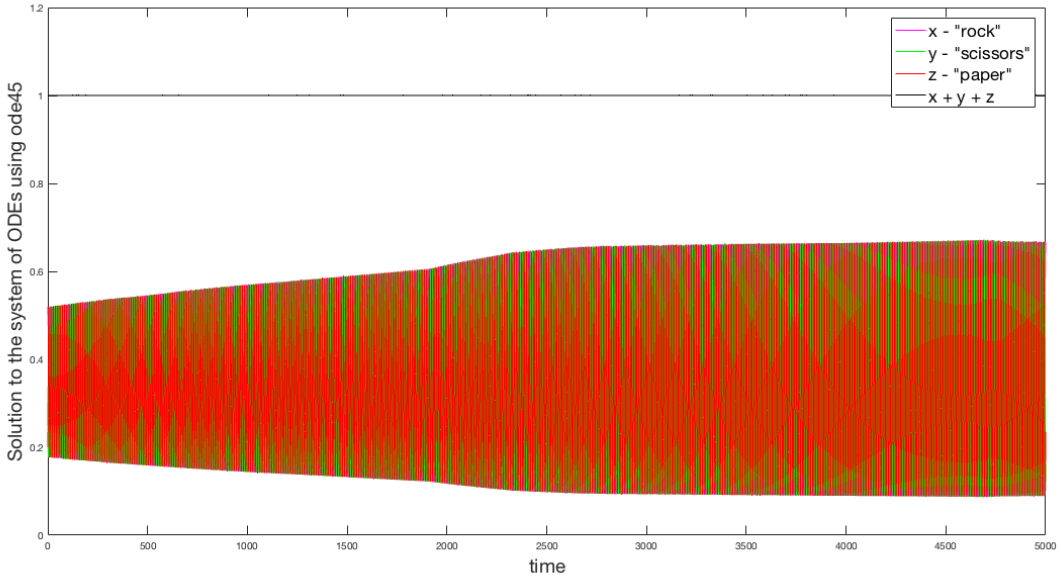


Figure 29: Conserved quantity $x + y + z$ up until $t = 5000$

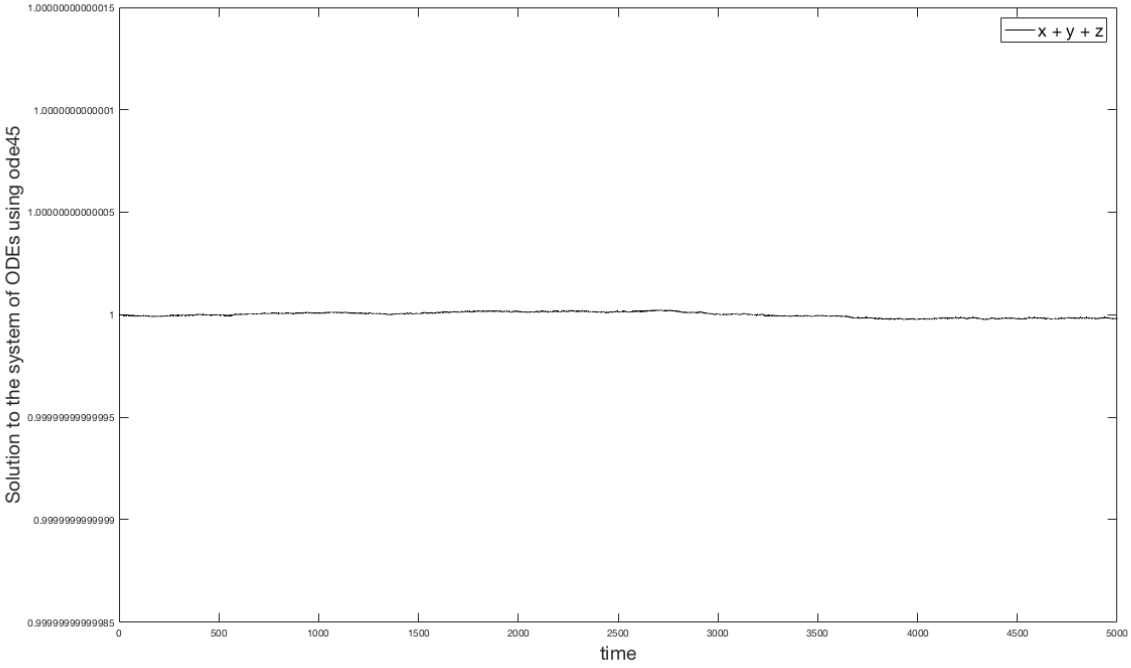


Figure 30: Conserved quantity x + y + z up until $t = 50000$

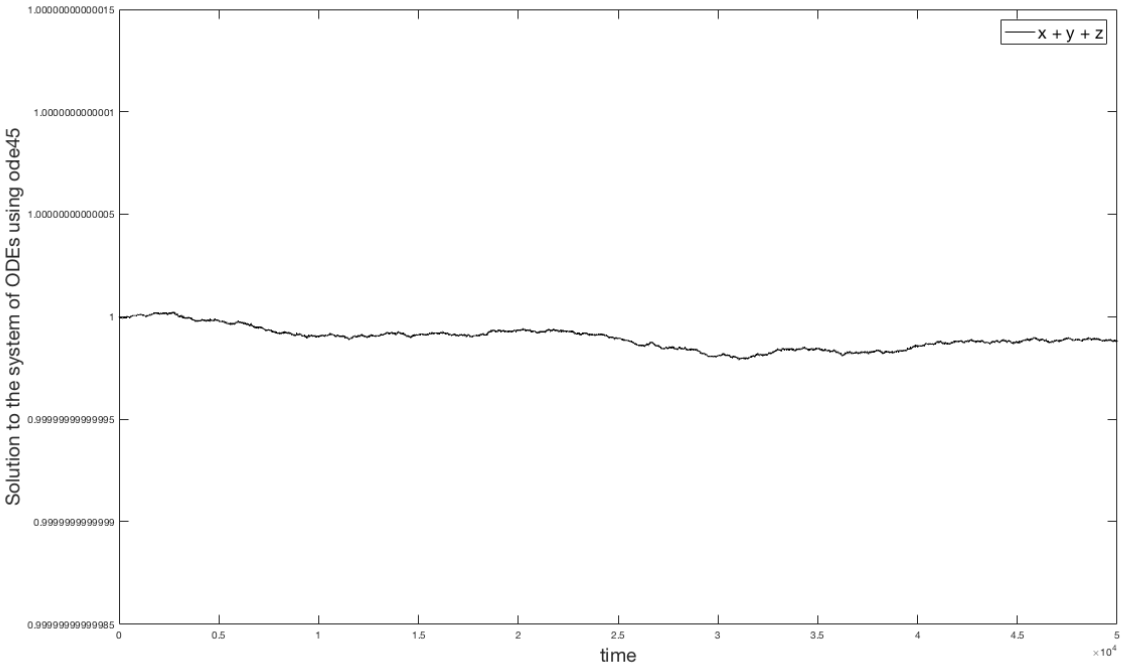


Figure 31: Solution of the system of differential equations using ode45 up until $t = 5000$ and conserved quantity xyz

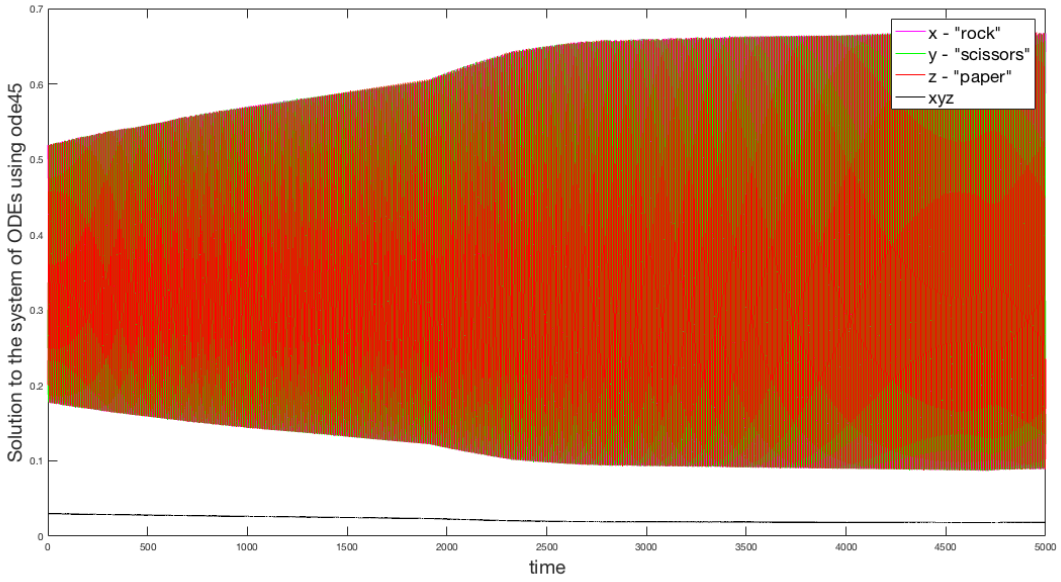


Figure 32: Conserved quantity xyz up until $t = 5000$

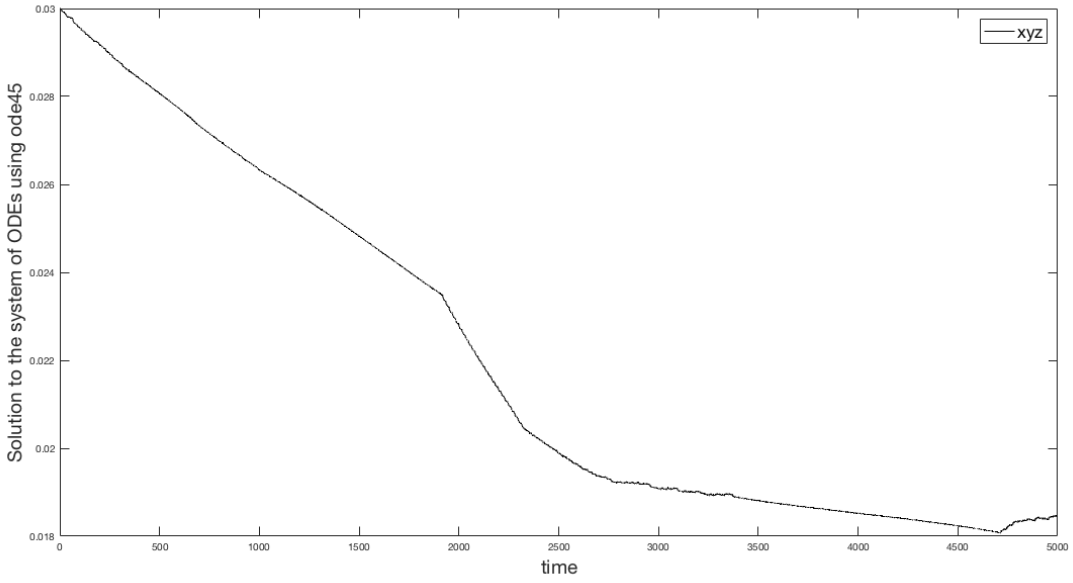
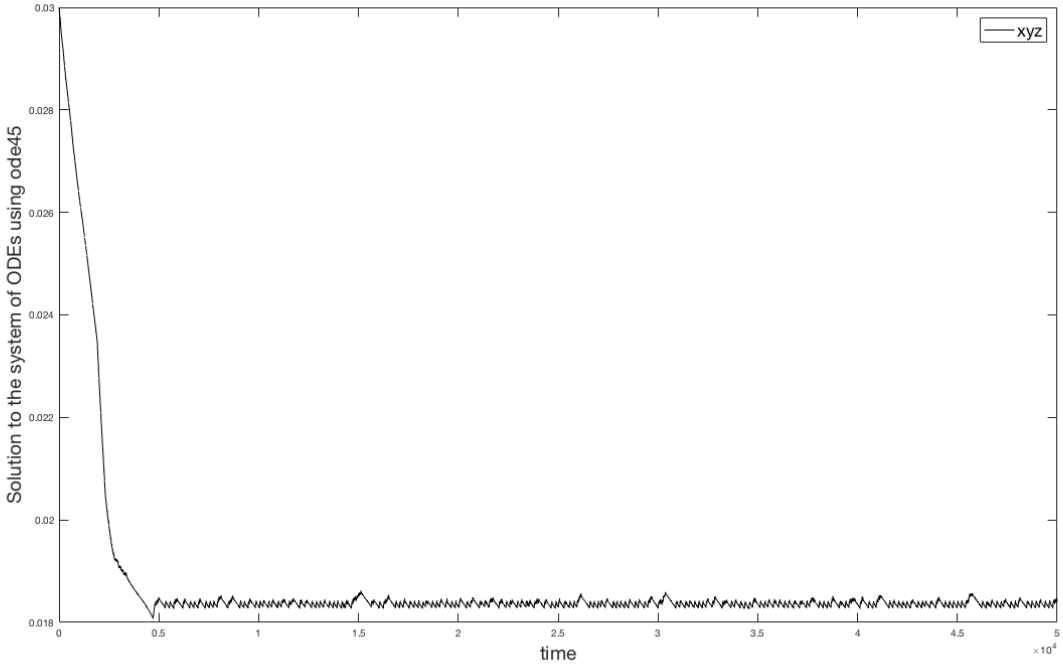


Figure 33: Conserved quantity xyz up until $t = 50000$



Observations

Ode45 performs well for up to around $t \approx 500$, but after that a spurious oscillation takes place and the oscillations jump to a higher amplitude. The conserved quantity $x + y + z$, looks as if it is conserved (Figure 28), but under closer analysis we see that it behaves very oddly for $t = [0, 5000]$ and $t = [0, 50000]$ (see Figure 29 and Figure 30). It is no loner a smooth line, like it is using the Euler method, but instead very unsteady and jumpy. This indicates that the altitudes of the oscillation are constantly changing slightly, showing ode45's incapability to deal with this system of ode's, because it produces spurious oscillations. The same is true for the conserved quantity xyz. Due to these spurious changes in amplitude of the oscillations, we see very unsteady results (Figure 32, 33) for different values of t, which is very different to the results obtained using the Euler method, where the lines were smooth.

END

Figure 34: I had to add this last image, because it looks so awesome. It shows a 3D plot of the system of ode's and I think one can really see that no single type is stronger than the others and that a continuous game will always result in going around a triangular loop.