Total Points: / 100

Submission Instruction: Please submit this homework on Canvas in a single pdf format. The filename should be "HWXX_FullName_RedID.pdf" (ex. HW04_JamesGault_12345678.pdf).
Please copy your Matlab code in the given box. Adjust the box size as needed.
Please also submit all your m files separately. **Don't zip them**.

Full Name: Evan Peres

Red ID: 129964468

Email address: eperes2481@sdsu.edu

Root Finding (Write your code in the box.)

1. Write a function with header [R, E] = myBisection(f, a, b, tol) where f is a function handle, a and b are scalars such that a < b, and tol is a strictly positive scalar value. The function should return an array, R, where R(i) is the estimation of the root of f defined by (a + b)/2 for the i-th iteration of the bisection method. Remember to include the initial estimate. The function should also return an array, E, where E(i) is the value of |f(R(i))| for the i-th iteration of the bisection method. The function should terminate when E(i) < tol. You may assume that sign(f(a)) ≠sign(f(b)).

Clarification: The input a and b constitute the first iteration of bisection, and therefore R and E should never be empty. (/ 15)

Test Cases:

```
>> f = @(x) x.ˆ2 − 2;
>> [R, E] = myBisection(f, 0, 2, 1e−1)
R =
        1.0     1.5000 1.2500 1.3750 1.4375
E =
        1.0000 0.2500 0.4375 0.1094 0.0664

>> f = @(x) sin(x) − cos(x);
>> [R, E] = myBisection(f, 0, 2, 1e−2)
R =
        1.0000    0.5000    0.7500    0.8750    0.8125   0.7813
E =
        0.3012    0.3982    0.0501    0.1265    0.0383    0.0059
```

```
function [R, E] = myBisection(f, a, b, tol)

R = [];
E = [];
while true
   c = (a + b) / 2;
   R(end+1) = c;
```

```
    E(end+1) = abs(f(c));
    if E(end) < tol
        break
    end
end
end
```

2. Write a function with header [R, E] = myNewton(f, df, x0, tol) where f is a function handle, df is a function handle to the derivative of f, x0 is an initial estimation of the root, and tol is a strictly positive scalar. The function should return an array, R, where R(i) is the Newton-Raphson estimation of the root of f for the i-th iteration. Remember to include the initial estimate. The function should also return an array, E, where E(i) is the value of |f(R(i))| for the i-th iteration of the Newton-Raphson method. The function should terminate when E(i) < tol. You may assume that the derivative of f will not hit 0 during any iteration for any of the test cases given. (/ 15)

Test Cases:

```
>>f = @(x) xˆ2 − 2;
>> df = @(x) 2*x;
>> [R, E] = myNewton(f, df, 1, 1e−5)
R=
        1.000000000000000     1.500000000000000     1.416666666666667     1.414215686274510
E =
        1.000000000000000     0.250000000000000     0.006944444444445     0.000006007304883

>>f = @(x) sin(x) − cos(x);
>> df = @(x) cos(x) + sin(x);
>> [R, E] = myNewton(f, df, 1, 1e−5)

R=
        1.000000000000000     0.782041901539138     0.785398175999702
E =
        0.301168678939757     0.004746462127804     0.000000017822278
```

```
function [R, E] = myNewton(f, df, x0, tol)
R = x0;
E = abs(f(x0));
while E(end) >= tol
    x0 = x0 - f(x0)/df(x0);
    R(end+1) = x0;
    E(end+1) = abs(f(x0));
end
end
```

Total Points: / 100

Curve Fitting (Write your code in the box.)

3. Given the following data, find the best linear functional relationship and quadratic functional relationship using "polyfit," "polyval," and "plot" built-in functions. Write a function with header [p1, f1, p2, f2] = myCurveFitting(x, y) where p1 is the coefficients for the linear polynomial, f1 is the respective y-values for the linear polynomial, p2 is the coefficients for the quadratic polynomial, and f2 is the respective y-values of the quadratic polynomial. Plot all three fits that you got from Matlab. Your code should produce a plot of the data points (x, y) marked as red circles. The points (x, y) should be connected by a blue line. Be sure to include title, axis labels, AND a legend. Save your function as myCurveFitting.m (10)

```
x = [-3, -2, 0, 1, 3, 4, 6, 7, 8, 9];
y = [5, 7, 8, 10, 6, 2, -4, -6, -2, 1];
```

```
function [p1, f1, p2, f2] = myCurveFitting(x, y)
p1 = polyfit(x, y, 1);
f1 = polyval(p1, x);
p2 = polyfit(x, y, 2);
f2 = polyval(p2, x);
plot(x, y, 'ro-', x, f1, 'g--', x, f2, 'k-.')
title('Curve Fitting Results')
xlabel('x values')
ylabel('y values')
legend('Data points','Linear fit','Quadratic fit')
end
```

Total Points: / 100

Interpolation (Write your code in the box.)

4. Given the following data, use linear interpolation and cubic spline interpolation to obtain functions that can be used to estimate intermediate data values. Write a function with header [vq1, vq2] = myInterpolation(x, y) where vq1 is the interpolated y-values for the linear interpolation and vq2 is the interpolated y-values for the cubic spline interpolation, the sampled xq is set as xq= **-3:0.1:9** to evaluate vq1 and vq2. Plot all two interpolations you got from Matlab. Your code should produce a plot of the data points (x, y) marked as red circles. The points (x, y) should be connected by a blue line. When producing your x-values, use an increment of .1. Be sure to include the title, axis labels, AND a legend. Save your function as myInterpolation.m (10)

```
x = [-3, -2, 0, 1, 3, 4, 6, 7, 8, 9];
y = [5, 7, 8, 10, 6, 2, -4, -6, -2, 1];
```

```
function [vq1, vq2] = myInterpolation(x, y)
   xq= -3:0.1:9;
   vq1 = interp1(x,y,xq);
   vq2 = spline(x,y,xq);
   plot(x, y, 'ro-', xq, vq1, 'g--', xq, vq2, 'k-.')
   title('Linear and Cubic Spline Interpolation')
   xlabel('x values')
   ylabel('y values')
   legend('Data points','Linear interpolation','Cubic spline interpolation')
end
```

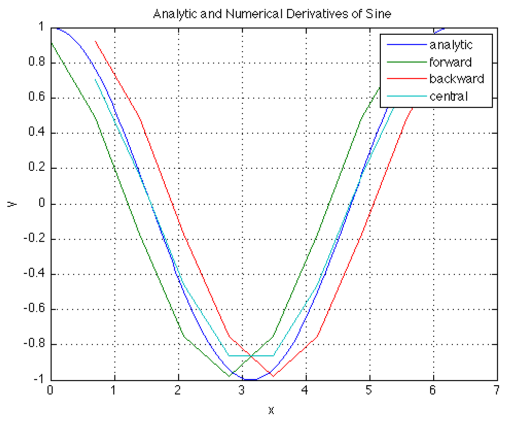
Total Points: / 100

Numerical differentiation (Write your code in the box.)

5. Write a function with header [dy, X] = myNumDiff(f, a, b, n, option) where f is handle to a function. The function myNumDiff should compute the derivative of f numerical for n evenly spaced points starting at a and ending at b according to the method defined by option. The input argument option is one of the following strings: 'forward', 'backward', 'central'. Note that for the forward and back ward method, the output argument, dy, should be 1 × (n − 1), and for the central difference method dy should be 1×(n−2). The function should also output a row vector X that is the same size as dy and denotes the x-values for which dy is valid. (/ 10)

Test Cases:

```
>> x = linspace(0, 2*pi,100);
>> f = @sin;
>> [dyf, Xf] = myNumDiff(f, 0, 2*pi, 10, 'forward');
>> [dyb, Xb] = myNumDiff(f, 0, 2*pi, 10, 'backward');
>> [dyc, Xc] = myNumDiff(f, 0, 2*pi, 10, 'central');
>> plot(x, cos(x), Xf, dyf, Xb, dyb, Xc, dyc)
>> title('Analytic and Numerical Derivatives of Sine')
 >> xlabel('x')
>> ylabel('y')
>> grid on
>> legend('analytic', 'forward', 'backward', 'central')
```



```
function [dy, X] = myNumDiff(f, a, b, n, option)
    x = linspace(a, b, n);
    h = x(2) - x(1);
    switch option
        case 'forward'
            dy = (f(x(2:end)) - f(x(1:end-1))) / h;
            X = x(1:end-1);
        case 'backward'
            dy = (f(x(2:end)) - f(x(1:end-1))) / h;
            X = x(2:end);
        case 'central'
        dy = (f(x(3:end)) - f(x(1:end-2))) / (2*h);
        X = x(2:end-1);
    end
end
```

Numerical integration (Write your code in the box.)

6. Write a function with header [I] = myNumInt(f, a, b, n, option) where I is the numerical integral of f, a function handle, computed on a grid of n evenly spaced points starting at a and ending at b. The integration method used should be one of the following strings defined by option: 'left', 'right', 'middle', 'trap'. For left, right, middle and Trapezoidal Riemann Sum. You may assume that n is odd and that f is vectorized. (15)

Test Cases:

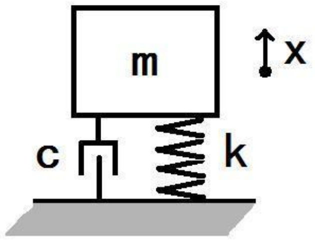
```
>> format long
>> f = @(x) x.^2;
>> I = myNumInt(f, 0, 1, 3 , 'left')
I =
  0.125000000000000
>> I = myNumInt(f, 0, 1, 3 , 'right')
I =
  0.625000000000000
>> I = myNumInt(f, 0, 1, 3 , 'middle')
I =
  0.312500000000000
>> I = myNumInt(f, 0, 1, 3 , 'trap')
I =
  0.375000000000000

>>  format long
>> f = @(x) exp(x.^2);
>> I = myNumInt(f, -1, 1, 101, 'trap')
I =
  2.925665905232575
>> I = myNumInt(f, -1, 1, 1001, 'trap')
I =
  2.925307116187722
>> I = myNumInt(f, -1, 1, 10001, 'trap')
I =
  2.925303528058127
```

```
function [I] = myNumInt(f, a, b, n, option)
x = linspace(a, b, n);
h = (b - a) / (n - 1);
switch option
```

```
    case 'left'
        I = sum(f(x(1:end-1))) * h;
    case 'right'
        I = sum(f(x(2:end))) * h;
    case 'middle'
        xm = (x(1:end-1) + x(2:end)) / 2;
        I = sum(f(xm)) * h;
    case 'trap'
        I = (f(a) + f(b) + 2*sum(f(x(2:end-1)))) * h / 2;
end
end
```

ODE (Write your code in the box.)

7. Consider the following model of a mass-spring-damper (MSD) system in one dimension. In this figure m denotes the mass of the block, c is called the damping coefficient, and k is the spring stiffness. A damper is a mechanism that dissipates energy in the system by resisting velocity. The MSD system is a simplistic model of several engineering applications such as shock observers and structural systems.



The relationship between acceleration, velocity, and displacement can be expressed by the following mass- spring-damper (MSD) differential equation:

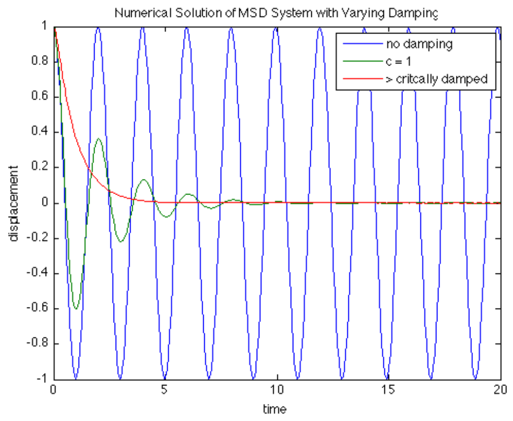
$$m\ddot{x} + c\dot{x} + kx = 0$$

which can be rewritten:

$$\ddot{x} = (-c\dot{x} - kx)/m$$

Let the state of the system be denoted by the vector S = [x; v] where x is the displacement of the mass from its resting configuration and v is its velocity. Rewrite the MSD equation as a first-order differential equation in terms of the state, S. In other words, rewrite the MSD equation as dS/dt = f(t,S).
Write a function with header [dS] = myMSD(t, S, m, c, k) where t is a scalar denoting time, S is a 2 × 1 vector denoting the state of the MSD system, and m, c, and k are the mass, damping, and stiffness coefficients of the MSD equation, respectively. (10)
Test Cases:

```
>> dS = myMSD(0, [1; −1], 10, 1, 100)
dS =
−1.0000 −9.9000
>> m = 1; k = 10;
>> [T0, S0] = ode45(@myMSD, [0 20], [1 0], [], m, 0, k);
>> [T1, S1] = ode45(@myMSD, [0 20], [1 0], [], m, 1, k);
>> [T2, S2] = ode45(@myMSD, [0 20], [1 0], [], m, 10, k);
>> plot(T0,S0(:,1), T1, S1(:,1), T2, S2(:,1))
>> title('Numerical Solution of MSD System with Varying Damping')’
 >> xlabel('time')
>> ylabel('displacement')
>> legend('no damping', 'c = 1', '> critcally damped')
```



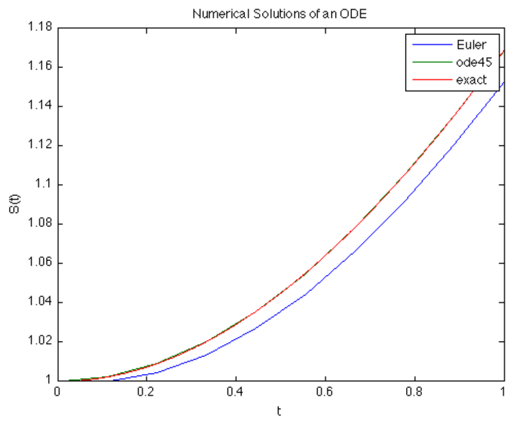
```
function [dS] = myMSD(t, S, m, c, k)
x = S(1);
v = S(2);
dx = v;
dv = (-c*v - k*x) / m;
dS = [dx; dv];
end
```

8. Write a function with header [T, S] = myForwardEuler(dS, tSpan, S0 where dS is a handle to a function, f(t,S), describing a first-order differential equation, tSpan is an array of times for which numerical solutions of the differential equation are desired, and S0 is the initial condition of the system. You may assume that the size of the state is one. The output argument, T, should be a column vector such that T(i) = tSpan(i) for all i, and S should be the integrated values of dS at times T. You should perform the integration using the Forward Euler method, i.e., $S(t_i) = S(t_{i-1}) + (t_i \cdot t_{i-1})dS(t_{i-1}, S(t_{i-1}))$.

Note: S(1) should equal S0. (15)

Test Cases:

```
>> dS = @(t, S) t*exp(−S);
>> tSpan = linspace(0,1,10);
>> S0 = 1;
>> [Teul, Seul] = myForwardEuler(dS, tSpan, S0);
>> Teul'
ans =
0 0.1111 0.2222 0.3333 0.4444 0.5556 0.6667 0.7778 0.8889 1.0000
>> Seul'
ans =
1.0000 1.0000 1.0045 1.0136 1.0270 1.0447 1.0664 1.0919 1.1209 1.1531
>> [Tode, Sode] = ode45(dS, tSpan, S0);
>> t = linspace(0,1,1000);
>> plot(Teul, Seul, Tode, Sode, t, log(exp(S0) + (t.ˆ2 − t(1)ˆ2)/2))
>> title('Numerical Solutions of an ODE')
>> xlabel('t')
>> ylabel('S(t)')
>> legend('Euler', 'ode45', 'exact')
```



```
function [T, S] = myForwardEuler(dS, tSpan, S0)
T = tSpan(:);
S = zeros(length(T),1);
S(1) = S0;
for i = 2:length(T)
   h = T(i) - T(i-1);
   S(i) = S(i-1) + h * dS(T(i-1), S(i-1));
end
end
```