E7 Final Exam

NAME : _____

SID : _____

SECTION : **1** or **2** (please circle your lecture section)

LAB :

#11: TuTh 8-10	#12: TuTh 10-12	#13: TuTh 12-2	#14: TuTh 2-4
#15: TuTh 4-6	#16: MW 8-10	#17: MW 10-12	#18: MW 2-4
#19: MW 4-6	#20: TuTh 10-12*	#21: MW 3-5 *	#22: TuTh 4-6*

(please circle your lab section) * in Wheeler

Problem	Points	Points
1	20	
2	10	
3	13	
4	12	
5	10	
6	10	
7	12	
8	9	
9	18	
10	18	
11	18	

Problem	Points	Points
12	10	
13	10	
14	10	
TOTAL	**180**	

Carefully read and follow these instructions:

1. **The last pages contain the syntax of some useful functions.**
2. Write your name on the top right corner of each page.
3. Start answering the exam only when instructed to do so.
4. Record your answers only in the spaces provided.
5. You may not ask questions during the exam.
6. You may not use any electronic devices.
7. You may use three 8.5×11 sheets (6 pages) of handwritten notes.
8. Count the number of pages before the start of the exam.
   There should be **31 pages**.

1. For each code-block, follow the instructions given.

   (a) Write the value of `V` after the code has executed. Write **Error** if the code will not execute.

begin code

```
clear
A = 3.7;
V = [2, 4, 6; 1, 5, 3];
V(V<A) = 0;
```

end code

   (b) Write the value of `V1`, `V2` and `V3` after the code has executed. Write **Error** if the code will not execute.

begin code

```
clear
S(1).Name = 'UCB';  S(1).Pop = 35700;
S(2).Name = 'UMich';  S(2).Pop = 43400;
S(3).Name = 'GTech';  S(3).Pop = 21500;
V1 = [S.Pop];
V2 = [S.Name];
V3 = size({S.Name})
```

end code

   (c) Write the value of `M` after the code has executed. Write **Error** if the code will not execute.

begin code

```
clear
M = [];
for k=3:-1:0
   M = [M  repmat(k,[1 k])];
end
```

end code

(d) Consider function `E7f1`, contained in file `E7f1.m`, listed below.

begin code

```
function T = E7f1(T,N,k)
for i=1:numel(T)
   T(i).(N{k}) = T(i).(N{k})(end:-1:1);
end
```

end code

Write the value of `V` after the code below has executed. Write **Error** if the code will not execute.

begin code

```
>> clear
>> S(1).Name = 'UCB';  S(1).Pop = 35700;
>> S(2).Name = 'UMich';  S(2).Pop = 43400;
>> S(3).Name = 'GTech';  S(3).Pop = 21500;
>> field1 = 'Pop';
>> test = S(1).(field1)
test =
35700
>> S = E7f1(S,fieldnames(S),1);
>> V = S(3).Name;
```

end code

(e) What is your best guess of the value of `N` after executing the code below?

begin code

```
clear
A = rand(1,1000);
N = numel(find(A>0.8));
```

end code

2. Consider a linear system of two equations with two unknowns

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}$$

John's code to solve for x given values for a and b is

begin code

```
function [x] = solveByJohn(a11,a12,a21,a22,b1,b2)
x = [a11, a12; a21, a22]\[b1;b2];
```

end code

Esme's code to solve for x is direct, eliminating one variable, solving for the remaining, and then repeating the process.

begin code

```
function [x1,x2] = solveByEsme(a11,a12,a21,a22,b1,b2)
x1 = (a22*b1-a12*b2)/(a22*a11-a12*a21);
x2 = (a21*b1 - a11*b2)/(a21*a12 - a11*a22);
x = [x1;x2];
```

end code

Their manager writes a testing program to check the correctness of the programs by comparing their answers. It is shown below:

begin code

```
nTrials = 10000;
nEqual = 0;
for i=1:nTrials
   a11 = randn; a12 = randn; a21 = randn; a22 = randn;
   b1 = randn; b2 = randn;
   xJ = solveByJohn(a11,a12,a21,a22,b1,b2);
   xE = solveByEsme(a11,a12,a21,a22,b1,b2);
   nEqual = nEqual + isequal(xJ,xE);
end
disp(['Fraction match = ' num2str(nEqual/nTrials)]);
```

end code

Unfortunately, the test appears to illustrate some unreliable results, as the fraction of tests that match is somewhat low. Their manager concludes that "one of you should be fired, since at least one of these functions is wrong."

Who should be fired: John, Esme or the manager? Why?

4. Consider the following second order ODE

$$\ddot{x}(t) + 2x(t) = 3t \qquad x(0) = 1, \qquad \dot{x}(0) = 1;$$

(a) This second order ODE can be written as a pair of coupled first order ODEs. Defining $y_1 = x$ and $y_2 = \dot{x}$, write below a corresponding equivalent pair of coupled first order ODEs and initial conditions:

$$\dot{y}_1 \quad = \qquad\qquad\qquad y_1(0) =$$

$$\dot{y}_2 \quad = \qquad\qquad\qquad y_2(0) =$$

(b) Use the Euler integration method with a step-size $h = 1$ to integrate the pair of coupled first order ODEs and fill out the empty entries in the table below.

t_k	$y_1(t_k)$	$y_2(t_k)$
0	1	1
1		
2		
3	-1	

5. The contents of a file `myfuncE13.m` are shown below.

begin code

```
function [y1,y2] = myfuncE13(a,b,c)
y1 = LOCALf1(b,a);
y2 = @(x) c(LOCALf2(a*x, b+x));

function H = LOCALf1(x,y)
H = x - y^2;

function K = LOCALf2(z,q)
K = [z+q, z-q];
```

end code

The following is executed in command window

begin code

```
>> [A,B] = myfuncE13(2,4,@sqrt);
>> C = B(4);
```

end code

(a) After executing the code, what is the value of `A`?

(b) After executing the code, what is the value of `C`?

6. Use induction to prove that $13^n - 7^n$ is always divisible by 6 for any integer $n \geq 1$.

7. **Horner's** method is an algorithm to evaluate the value of a polynomial function. Consider a n'th degree polynomial

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

where $a_n, \cdots, a_0$ are real numbers, Rewrite p equivalently as

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

Note that this has the appearance of an $(n-1)$'th degree polynomial (although the coefficient associated with the x^{n-1} term is itself dependent on x).

To evaluate $p(x_0)$ (for a specific value x_0), we can define the $(n-1)$'th degree polynomial

$$q(x) \quad = \quad (a_n x_0 + a_{n-1})x^{n-1} + a_{n-2}x^{n-2} + \cdots a_1 x + a_o$$

Note that the degree of $q(x)$ is less than the degree of $p(x)$ and moreover, $p(x_0) = q(x_0)$. Repeating this procedure recursively, we can define a sequence of k^{th} order polynomials $P_k(x)$ for $k = n, n-1, \cdots, 0$,

$$\begin{aligned}
P_n(x) &= \underbrace{a_n}_{b_n} x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots a_1 x + a_o \\
P_{n-1}(x) &= \underbrace{(b_n x_0 + a_{n-1})}_{b_{n-1}} x^{n-1} + a_{n-2}x^{n-2} + \cdots a_1 x + a_o \\
P_{n-2}(x) &= \underbrace{(b_{n-1} x_0 + a_{n-2})}_{b_{n-2}} x^{n-2} + \cdots a_1 x + a_o \\
&\vdots \\
P_0(x) &= \underbrace{(b_1 x_0 + a_0)}_{b_0}
\end{aligned}$$

which all satisfy $P_n(x_0) = p(x_0)$.

The **recursive** function `Horner` shown below is designed to define the $P_k(x)$'s polynomials recursively, in order to finally evaluate $p(x_0) = P_0(x_0) = b_0$. Input and output arguments of `Horner` are:

- `P` $= [\mathtt{a_n}, \mathtt{a_{n-1}}, \cdots, \mathtt{a_o}]$ is the $1 \times (\mathtt{n}+1)$ array containing the coefficients of $p(x)$.
- `x0` is a specific value of x.
- `pval` is $p(\mathtt{x0})$.

Complete the **3 incomplete** lines of code in the next page.

begin code

```
function Pval = Horner(P, x0)

if
    ____________________________________________________;

    Pval = P;

else

    P  =  [
             _______________________________________________;


    Pval = Horner(              );
                    -------------
end
```

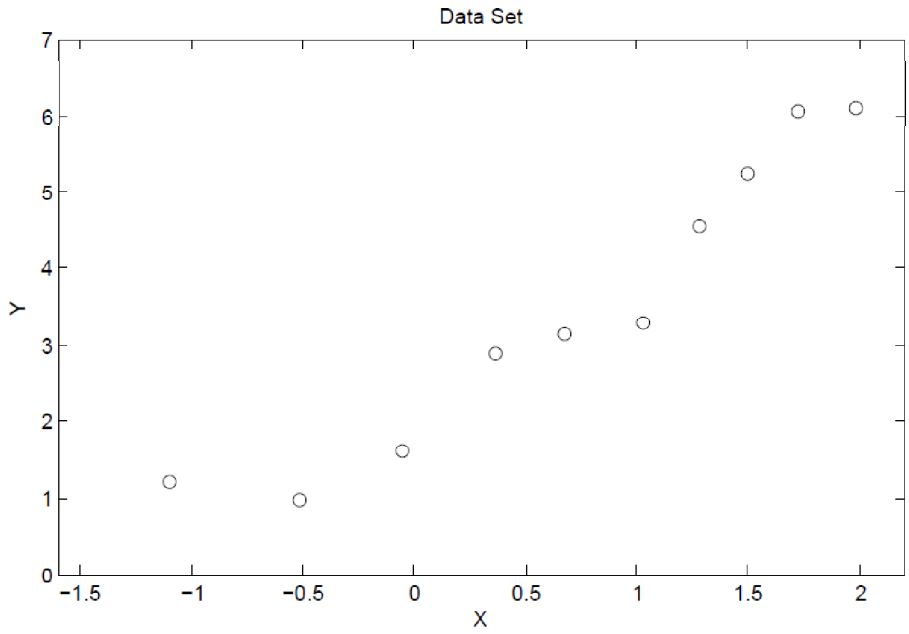
end code

8. (a) This code block produces the figure as shown below

begin code

```
clf;  clear
load e7Data;
plot(xData, yData, 'o');  hold on
```

end code



The following additional code modifies the figure. In the figure above, make a sketch of the modification after both code blocks are executed.

begin code

```
A = [ones(numel(xData),1) xData];
b = yData;
c = A\b;
xVec = linspace(min(xData),max(xData),100);
plot(xVec, c(1)+c(2)*xVec, 'k');
hold off
```

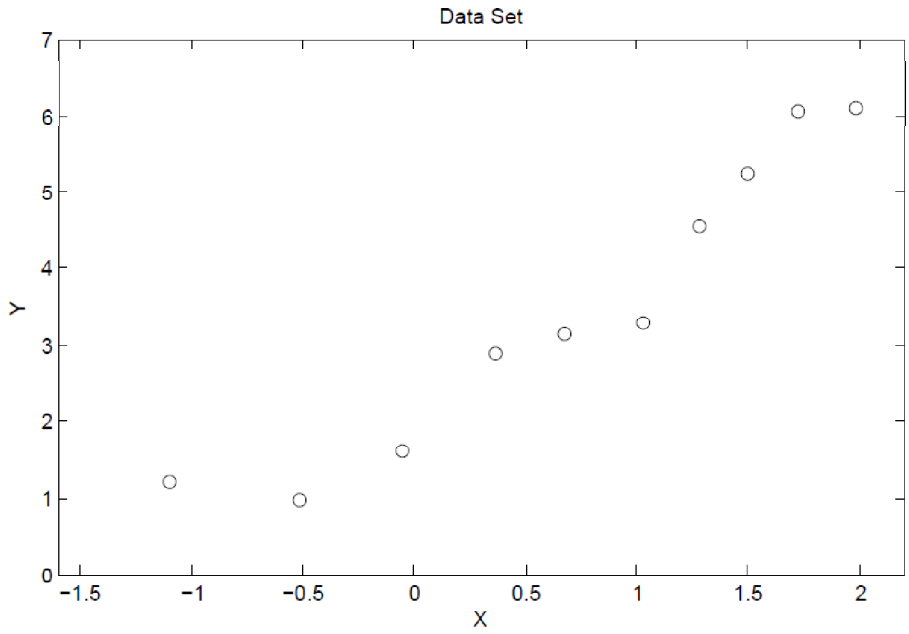
end code

(b) This code produces the figure as shown below

begin code

```
clf;  clear
load e7Data;
plot(xData, yData, 'o');  hold on
```

end code



The following additional code modifies the figure. In the figure above, make a sketch of the modification after both code blocks are executed.

begin code

```
A = [ones(numel(xData),1) xData xData.^2];
b = yData;
c = A\b;
xVec = linspace(min(xData),max(xData),100);
plot(xVec, c(1)+c(2)*xVec+c(3)*xVec.^2, 'k');
hold off
```

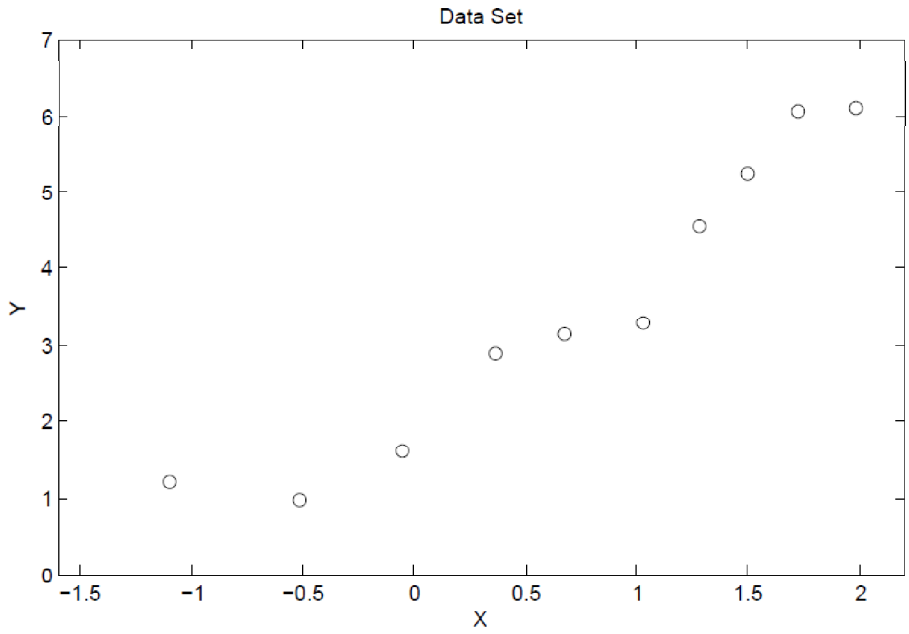
end code

(c) This code produces the figure as shown below

begin code

```
clf;  clear
load e7Data;
plot(xData, yData, 'o');  hold on
```

end code



The following additional code modifies the figure. In the figure above, make a sketch of the modification after both code blocks are executed.

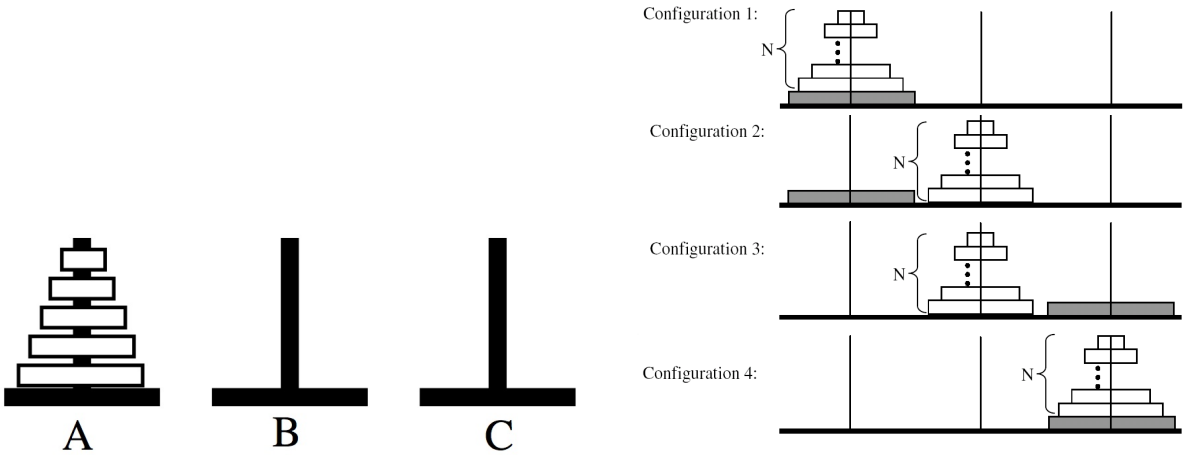
begin code

```
xVec = linspace(min(xData),max(xData),100);
ySpline = interp1(xData,yData,xVec,'spline');
plot(xVec, ySpline, 'k');
hold off
```

end code

9. Figure 1(a) shows the classic Towers of Hanoi puzzle consisting of three pegs and a number of disks of varying sizes. The objective of the puzzle is to move the ordered stack of disks from the starting peg (peg A in Fig. 1(a)) in to another peg while obeying the following rules:

   1. A move consists of taking the topmost disk from one peg and transferring it to another peg.
   2. You may only move **one** disk at a time.
   3. You may **not** place a disk on top of a smaller disk.



(a) Initial Configuration

(b) Moving an $N+1$ Stack of Disks from A to C

Figure 1: Towers of Hanoi puzzle

(a) Write the **5 lines of code** of the function called `HanoiSteps`, which recursively computes the minimum number of steps required to solve the Towers of Hanoi puzzle for a stack of N disks. To do so, read the following hint:

Define $S(N)$ to be the minimum number of steps needed to move N disks from one peg to another. Now, consider $N+1$ disks: In Configuration 1 of Fig. 1(b), disk $N+1$ is shown in gray at the bottom of the initial stack. It takes $S(N)$ steps to get from Configuration 1 to Configuration 2. It takes only one step to get from Configuration 2 to Configuration 3. Finally, it takes $S(N)$ steps to get from Configuration 3 to Configuration 4. The number of steps taken to get from Configuration 1 to Configuration 4 is equal to $S(N+1)$.

begin code

```
1  function Steps = HanoiSteps(N)
2
3  ------------------------------------------------------------
4
5  ------------------------------------------------------------
6
7  ------------------------------------------------------------
8
9  ------------------------------------------------------------
10
11 ------------------------------------------------------------
```

end code

(b) Complete the missing code of the **recursive** function `Hanoi`, which will sequentially display the steps required to solve the Towers of Hanoi Puzzle. The syntax of the function `Hanoi` is

begin code
```
Hanoi(N, BPeg, IPeg, Fpeg )
```
end code

Input arguments:

- `N`: number of disks
- `BPeg`: char denoting where the stack of disks is at the beginning (e.g. `'A'` in Fig. 1(b)).
- `IPeg`: char denoting the third (extra) peg (e.g. `'B'` in Fig. 1(b)).
- `Fpeg`: char denoting the final location of the stack (e.g. `'C'` in Fig. 1(b)).

Shown below is a test case for the function `Hanoi`

begin code
```
>> Hanoi(2,'A','B','C')
Move one disk from peg A to B
Move one disk from peg A to C
Move one disk from peg B to C
```
end code

begin code
```
function Hanoi(N, BPeg, IPeg, FPeg)

if
    ------------------------------------------------------------

    disp(
    ------------------------------------------------------------

else

    ------------------------------------------------------------

    ------------------------------------------------------------

    ------------------------------------------------------------

end
```
end code

10. Assume that the following is known about a real valued function of a single real variable $F(x)$.

- The value of $F(x_o) = F_o$ is known for $x = x_o$.
- The derivative of F with respect to x, $f(x) = \frac{dF}{dx}(x)$, is known for all x. Moreover, you have access to the function `Fder` with syntax *

```
begin code
fder = Fder(x1)
end code
```

The output argument `fder` is the derivative of the function $F(x)$ at the input argument `x1`.

(a) Complete the code below for the function `Func`, which computes the value of the function $F(x_1)$ for a given value x_1. Input and output arguments are

- `x1`: specified value of x_1.
- `Fderh`: handle to function `Fder`, which evaluates the derivative $f(x) = \frac{dF(x)}{dx}$.
- `xo, Fo`: known values of x_o and $F_o = F(x_o)$.
- `F1` calculated value of $F(x_1)$.

```
begin code
function F1 = Func(x1, Fderh,  xo, Fo)

F1 =
        ------------------------------------------------------------;
end code
```

(b) Assume now that, instead of knowing the the exact value of the function $F(x)$ at a single point, its exact value is known at several points. Thus, assume that the following two arrays of size $N \times 1$ are available:

- `Xarray` $= \left[\; \mathtt{x_{o_1}} \; ; \quad \mathtt{xo_2} \; ; \quad \cdots \; ; \mathtt{x_{o_N}} \; \right]$
- `Farray` $= \left[\; \mathtt{F(x_{o_1})} \; ; \quad \mathtt{F(xo_2)} \; ; \quad \cdots \; ; \mathtt{F(x_{o_N})} \; \right]$, where $\mathtt{F(x_{o_i})}$ means the exact value of the function $F(x)$ at $x = \mathtt{x_{o_i}}$ (remember that $F(x)$ is unkown).

Complete the lines of code for the function `FuncA`, which computes the value of the function $F(x_1)$ for a given value x_1 in a computationally efficient manner, given that you have access to the arrays `Xarray` and `Farray` and the function `Fder`.

The first two input arguments and output argument of `FuncA` are the same as those of the function `Func` in part 10a. You can use less than three lines, but should not use more.

*i.e. the file `Fder.m` is stored in a directory in your `Matlab` path.

begin code

```
function F1 = FuncA(x1, Fderh, Xarray, Farray)


_____________________________________________________________;


_____________________________________________________________;

F1 =
        _____________________________________________________________;
```

end code

(c) We now want to find a zero of the function $F(x)$ (i.e. the value z such that $F(z) = 0$) using a modified Newton-Raphson algorithm. Assume that you have access to the function `Fder`, a working function `FuncA`, and the arrays `Xarray` and `Farray`.

Complete the missing or incomplete lines of code for the function `NewtonDer` in the next page, which determines the approximate location of a zero of $F(x)$.

Input arguments:

- `z0`: initial guess of the zero of $F(x)$.
- `Tol`: relative error tolerance value.
- `iterMax`: maximum number of iterations allowed.
- `FuncAh`: handle to function `FuncA` described in part 10b.
- `Fderh`: handle to function `Fder`, which evaluates the derivative $f(x) = \frac{dF(x)}{dx}$.
- `Xarray` $= \left[\; \mathtt{x_{o_1}}; \;\; \mathtt{xo_2}; \;\; \cdots; \mathtt{x_{o_N}} \; \right]$
- `Farray` $= \left[\; \mathtt{F(x_{o_1})}; \;\; \mathtt{F(xo_2)}; \;\; \cdots; \mathtt{F(x_{o_N})} \; \right]$.

Output arguments:

- `z`: final estimate of the root of $F(x)$.
- `Fz`:The calculated value $F(\mathtt{z})$.

The algorithm should stop when either of the following conditions are satisfied

$$\frac{|z_k - z_{k-1}|}{|z_k|} \leq \mathtt{Tol} \qquad \mathtt{or} \qquad k \geq \mathtt{iterMax}$$

with $k \geq 1$ is the k^{th} iteration of the algorithm and x_k is the zero estimate at step k. Assume that `Fder(x)` $\neq 0$ for all `x` and $z_k \neq 0$ for all k.

begin code

```
function [z, Fz] = NewtonDer(z0, Tol, iterMax, ...
                             FuncAh, Fderh, Xarray, Farray)

zold  = z0;                              % Set z(0) = z0
RelError = Tol + 1;                      % Set RelError(0) > Tol
k = 1;                                   % Set iteration counter to 1

%    Check for stopping conditions

while  (RelError > Tol) && (k < iterMax)

    Fzold =

    ____________________________________________________________;
    % Compute the new iteration of the zero z(k)

    z =

    ____________________________________________________________;
    % Update the iteration counter

    k =
    ____________________________________________________________;
    % Compute RelError(k)


    ____________________________________________________________;


    ____________________________________________________________;

end


_____________________________________________________________;
```

end code

11. After totaling the scores of the E7 class, the probability density function (PDF) of the scores $p_S(x) \geq 0$ was determined for $x_{\min} \leq x \leq x_{\max}$, with $x_{\min}$ and $x_{\max}$ being respectively the minimum and maximum scores and

$$\int_{x_{\min}}^{x_{\max}} p_S(x)dx = 1\,.$$

A plot of $p_S(x)$ with $x_{\min} = 10$ and $x_{\max} = 100$ is shown below.

(a) Assume that you have access to the Matlab function `E7ScoresPDF`, with syntax

begin code

```
ps = E7ScoresPDF(x)
```

end code

where `x` is a particular score and `ps` $= p_S($`x`$)$.

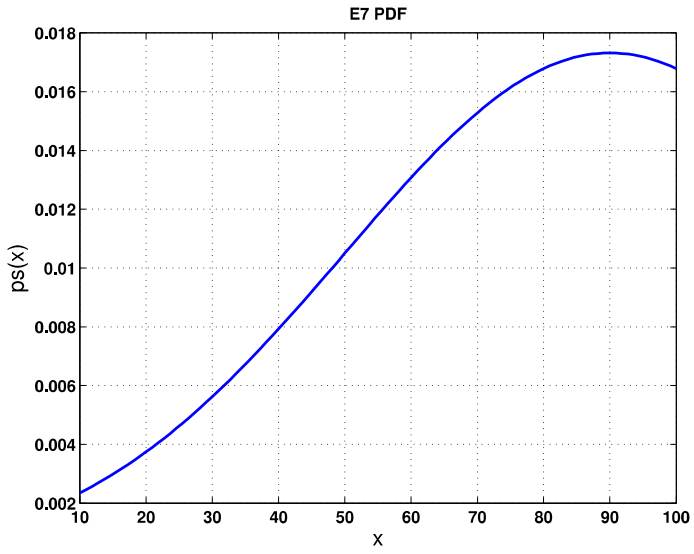


Figure 2: PDF of E7 Scores

Write a Matlab script that will produce a similar figure as Fig. 2 including annotations [†].

begin code

```
1
2     ____________________________________________________________;
3
4     ____________________________________________________________;
5
6     ____________________________________________________________;
7
8     ____________________________________________________________;
9
10    ____________________________________________________________;
11
12    ____________________________________________________________;
```

end code

(b) In addition to the information in part 11a, assume that the following variables have been defined and are accessible in your workspace.

- `xmin`: the minimum E7 score $x_{\min}$.
- `xmax`: the maximum E7 score $x_{\max}$.
- `xhat`: the mean or expected value $E\{x\} = \int_{\texttt{xmin}}^{\texttt{xmax}} x p_S(x) dx$ of the E7 scores.
- `Probxhat`: the probability of attaining a score that is less than or equal to the mean, i.e. $\texttt{Probxhat} = \texttt{Prob}\{x \leq \texttt{xhat}\}$.

Write a 2-line Matlab script to determine the standard deviation of the E7 scores.

begin code

```
1
2
3     ____________________________________________________________;
4
5
6     ____________________________________________________________;
```

end code

[†]You **do not** have to specify the tick locations, fontsize, Fontweight, or linewidth.

(c) Several students want to know what is the lowest score that they have to obtain in order to get an A+ in E7. The professors teaching E7 have decided that a student must obtain a score that is higher than or equal to 95% of all other scores, in order to receive an A+.

Denote `xAP` to be the lowest score that will receive an A+ in E7, i.e.

$$\mathrm{Prob}\{x \leq \texttt{xAP}\} = 0.95$$

Note that $\mathrm{Prob}\{x \leq \texttt{A}\}$ is defined as

$$\mathrm{Prob}\{x \leq \texttt{A}\} = \int_{\texttt{xmin}}^{\texttt{A}} p_S(x)dx\,.$$

Using the functions, `E7ScoresPDF`, `FuncA` and `NewtonDer` in Problem 10 and all of the above information, the professors determined that `xAP` $= 97$.

Assume that you have access to the same functions (i.e. files E7ScoresPDF.m, `FuncA.m` and `NewtonDer.m` are stored in directory that is in your `Matlab path`) and all of the above information.

In the space below, write a `Matlab` script to compute `xAP` using **ONLY** the information and functions described above.

begin code

```
Tol = 1e-10; iterMax = 100; AP = 0.95;

_____________________________________________________________;

_____________________________________________________________;

_____________________________________________________________;

_____________________________________________________________;

_____________________________________________________________;
```

end code

13. **Warning:** This problem is related to but different from a homework problem.

Consider the following heat transfer problem. The temperature of a metallic bar is denoted $T(x)$. The temperature is a function of position (x), due to various heat sources, conduction and radiation. Regarding notation, $\dot{T}(x)$ denotes the 1st derivative of T with respect to x, and similarly, $\ddot{T}(x)$ represents 2nd derivative of T with respect to x. The differential equation governing the temperature is

$$\ddot{T}(x) = -\alpha\left(T_\infty - T(x)\right) - \beta\left(T_\infty^4 - T^4(x)\right) \tag{1}$$

where α, β and T_∞ are known constants. Let y denote the 2×1 vector

$$y(x) = \begin{bmatrix} T(x) \\ \dot{T}(x) \end{bmatrix}$$

(a) Convert equation (1) into a set of coupled, first-order ordinary differential equations (ODEs) in the variable y and complete the function `temperaturebar.m` shown below so that `ode45` can be used to solve this system of differential equations. Note that α, β and T_∞ are constants, and have been defined.

begin code

```
function yprime = temperaturebar(x,y)
alpha = 0.05;
beta = 2.7e-9;
Tinf = 200;

yprime1 =

yprime2 =

yprime =
```

end code

(b) Fill in the code below to numerically solve $T(x)$ on the interval $x = 0$ to $x = 10$, subject to the initial condition $T(0) = 300, \dot{T}(0) = -44$, and then to plot the solution, $T(x)$ versus x.

begin code

```

[xSol,ySol] = ode45(

plot(

```

end code

(c) Next consider a boundary-value problem (BVP) for a bar with length equal to 10. Rather than being given initial conditions (T and $\dot{T}$ at $x = 0$), the temperature is specified at one end ($x = 0$), and the slope ($\dot{T}$) is specified at the other end ($x = 10$) as

$$T(0) = 300, \quad \dot{T}(10) = -40$$

The differential equation for T is still the same, namely equation (1), but the form of the given information differs from an initial value problem (IVP). **In a Boundary-Value problem**

(BVP), the challenge is to determine the missing component ($\dot{T}(0)$) of the initial condition, so that when the differential equation is solved as an IVP, the resultant solution has the correct final value ($\dot{T}(10) = -40$). Since the differential equation, (1), is nonlinear, a numerical-based method, called *shooting*, can be used to solve the BVP. The basic idea is to solve the IVP repeatedly for different values of the unknown initial condition $\dot{T}(0)$, until the resultant solution at $x = 10$ satisfies the given boundary condition $\dot{T}(10) = -40$. This search can be automated using `fzero`. Below are 2 tasks that break the problem into manageable steps.

i. Write a function `getTdot10.m`, with the following function declaration line

begin code
```
function tdot10 = getTdot10(P)
```
end code

where the input argument, `P`, is a value for $\dot{T}(0)$, the output argument `tdot10` is the value of $\dot{T}(10)$ of the resulting solution, subject to the initial condition

$$\begin{bmatrix} T(0) \\ \dot{T}(0) \end{bmatrix} = \begin{bmatrix} 300 \\ P \end{bmatrix}$$

Write your solution in the box below.

begin code
```
function tdot10 = getTdot10(P)
```
end code

ii. Using `getTdot10`, create an anonymous function (assigned to the variable `H` in the code block below) that has one input argument, P (the guess for $\dot{T}(0)$), such that the code below solves for the correct value of $\dot{T}(0)$ such that the condition $\dot{T}(10) = -40$ is satisfied.

begin code
```
H =

InitialGuess = -55;
Tdot0required =

```
end code

Appendix of useful functions

- **repmat** Replicate and tile an array.

  `B = repmat(A,[M N])` creates a large matrix B consisting of an M-by-N tiling of copies of A. The size of B is [size(A,1)*M, size(A,2)*N].

- **find** Find indices of nonzero elements.

  I = find(X) returns the linear indices corresponding to the nonzero entries of the array X. X may be a logical expression. Use IND2SUB(SIZE(X),I) to calculate multiple subscripts from the linear indices I.

- **min** Smallest component.

  For vectors, min(X) is the smallest element in X. For matrices, min(X) is a row vector containing the minimum element from each column.

- **lpsolver** Solves LP in standard form: min c'*x, subject to $Ax_j=b$

  [optx,lambda,status,gamma] = lpsolver(c,A,b)

- **integral** Numerically evaluate integral, adaptive Simpson quadrature. Q = integral(FUN,A,B) tries to approximate the integral of scalar-valued function FUN from A to B to within an error of 1.e-6 using recursive adaptive Simpson quadrature. FUN is a function handle. The function Y=FUN(X) should accept a vector argument X and return a vector result Y, the integrand evaluated at each element of X.

- **ode45** Solve non-stiff differential equations, medium order method.

  [TOUT,YOUT] = ode45(ODEFUN,TSPAN,Y0) with TSPAN = [T0 TFINAL] integrates the system of differential equations y' = f(t,y) from time T0 to TFINAL with initial conditions Y0. ODEFUN is a function handle. For a scalar T and a vector Y, ODEFUN(T,Y) must return a column vector corresponding to f(t,y). Each row in the solution array YOUT corresponds to a time returned in the column vector TOUT. To obtain solutions at specific times T0,T1,...,TFINAL (all increasing or all decreasing), use TSPAN = [T0 T1 ... TFINAL].

- **fzero** Single-variable nonlinear zero finding.

  X = fzero(FUN,X0) tries to find a zero of the function FUN near X0

- **rand** Uniformly distributed pseudorandom numbers. R = rand(N) returns an N-by-N matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval(0,1). rand(M,N) or rand([M,N]) returns an M-by-N matrix. rand(M,N,P,...) or rand([M,N,P,...]) returns an M-by-N-by-P-by-... array. rand returns a scalar. rand(SIZE(A)) returns an array the same size as A.

- **randn** Normally distributed pseudorandom numbers.

  R = randn(N) returns an N-by-N matrix containing pseudorandom values drawn from the standard normal distribution. randn(M,N) or randn([M,N]) returns an M-by-N matrix. randn(M,N,P,...) or randn([M,N,P,...]) returns an M-by-N-by-P-by-... array. randn returns a scalar. randn(SIZE(A)) returns an array the same size as A.

- **interp1** 1-D interpolation (table lookup)

  Vq = interp1(X,V,Xq) interpolates to find Vq, the values of the underlying function V=F(X) at the query points Xq. X must be a vector of length N. If V is a vector, then it must also have length N, and Vq is the same size as Xq. If V is an array of size [N,D1,D2,...,Dk], then the interpolation is performed for each D1-by-D2-by-...-Dk value in V(i,:,:,...,:). If Xq is a vector of length M, then Vq has size [M,D1,D2,...,Dk]. If Xq is an array of size [M1,M2,...,Mj], then Vq is of size [M1,M2,...,Mj,D1,D2,...,Dk].

  Vq = interp1(V,Xq) assumes X = 1:N, where N is LENGTH(V) for vector V or SIZE(V,1) for array V.

  Vq = interp1(X,V,Xq,METHOD) specifies alternate methods. The default is linear interpolation. Use an empty matrix [] to specify the default. Available methods are:

  'nearest' - nearest neighbor interpolation

  'linear' - linear interpolation

  'spline' - spline interpolation