MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF MECHANICAL ENGINEERING
CAMBRIDGE, MASSACHUSETTS 02139

2.29 NUMERICAL FLUID MECHANICS— SPRING 2007

Problem Set 2

Posted 02/25/07, due Thursday 4 p.m. 03/8/07, Focused on Lecture 4 to 7

Problem 2.1 (6% of final grade): Advance your programming skills and review root finding methods

Review MATLAB help about:

- Function handle
- eval
- nargin
- varargin
- cell: as a data type
- switch: as flow control command
- fprintf
- lower

Here we want to develop a script as a generalized one dimensional solver. Later you can use it for next problems. The function that you write should provide the maximum ease of use, as well as the maximum amount of flexibility and adjustment. To that end and to develop a user friendly program:

- The function should have default values for everything so that the user can run it with minimum number of inputs.
- The input function (to be solved) should be either a function handle or a string (like '3*x^3-5*x+1').
- The program should have a nice command line output or plot displaying the gradual progress of solution.
- The user should be able to adjust/provide the below options, if necessary. Note that user should not need to memorize any order for them and option names should not be case sensitive:

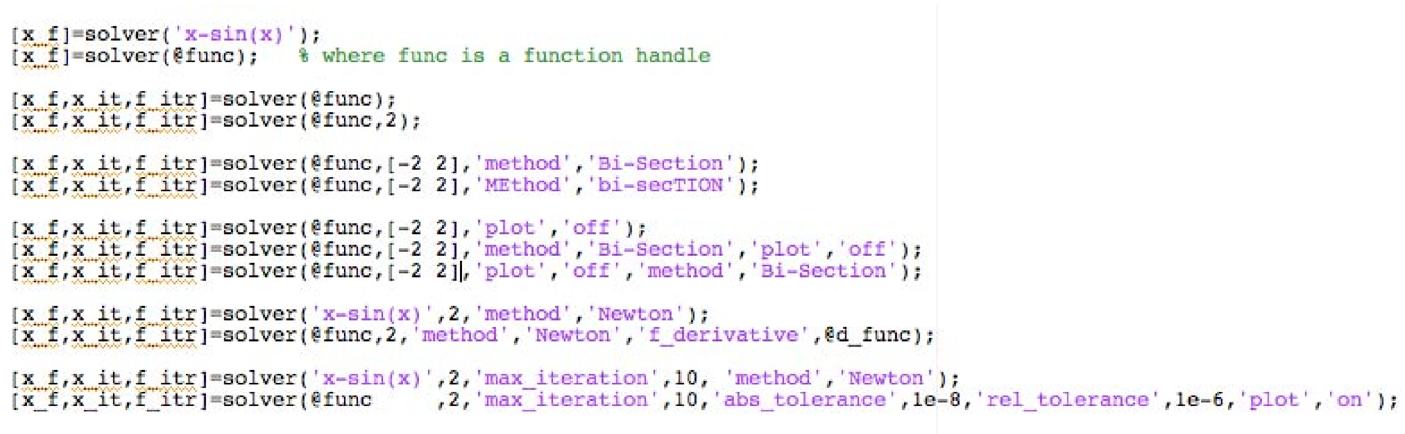
  a) Method: Newton, Secant, Bi-Section, False-Position, Modified False-Position
  b) Initial guess: it can be two numbers for methods like Bi-Section
  c) Derivative of f (note that you can compute the derivative if user provides you with a string as solution equation)
  d) Absolute tolerance on x or f

e) Relative error on x
f) Maximum number of iterations
g) Plot

Here is what function should be like:

[x_solution, x_iterations, f_iterations] = solver(func_name)
[x_solution, x_iterations, f_iterations] = solver(func_name, x_guess)
[x_solution, x_iterations, f_iterations] = solver(func_name, x_guess, OPTIONS)

A few example calls are shown:



```
[x_f]=solver('x-sin(x)');
[x_f]=solver(@func);    % where func is a function handle

[x_f,x_it,f_itr]=solver(@func);
[x_f,x_it,f_itr]=solver(@func,2);

[x_f,x_it,f_itr]=solver(@func,[-2 2],'method','Bi-Section');
[x_f,x_it,f_itr]=solver(@func,[-2 2],'MEthod','bi-secTION');

[x_f,x_it,f_itr]=solver(@func,[-2 2],'plot','off');
[x_f,x_it,f_itr]=solver(@func,[-2 2],'method','Bi-Section','plot','off');
[x_f,x_it,f_itr]=solver(@func,[-2 2],'plot','off','method','Bi-Section');

[x_f,x_it,f_itr]=solver('x-sin(x)',2,'method','Newton');
[x_f,x_it,f_itr]=solver(@func,2,'method','Newton','f_derivative',@d_func);

[x_f,x_it,f_itr]=solver('x-sin(x)',2,'max_iteration',10, 'method','Newton');
[x_f,x_it,f_itr]=solver(@func     ,2,'max_iteration',10,'abs_tolerance',1e-8,'rel_tolerance',1e-6,'plot','on');
```

After writing the program you have to UPLOAD IT ON THE COURSE WEBSITE and PRINT IT AS WELL. That's all you have to do for this problem. This will replace the MATLAB workshop assignment about MATLAB programming.

Solution:

Look at the attached MATLAB file "solver.m".

Problem 2.2 (10 Points): Examine your root finding script

We are interested to find the roots of:

$$f(x) = e^x \sin x + e^{-x} \cos^2 2x$$

a) How many roots does this equation have?

b) Can you approximate analytically some roots of this equation and characterize their type? Discuss.

c) Find all the roots where $|x| < 4$. Use above program and examine all the methods for any root. For each root use "plotyy" command and plot two things at the same figure:
   - x versus number of iterations
   - Relative error of x (with respect to the most trusted solution) versus number of iterations (use logarithmic scale if needed)

d) Repeat part c and find the fist two roots where $x > 20$.

e) Repeat part c and find the fist two roots where $x < -20$.

Solution:

a) Infinite number of roots, due to infinite number of oscillations generated by trigonometric function multiplied by an exponential function (see next part).

b) The function can be approximated by:

$$x \to +\infty \Rightarrow f(x) \cong e^x \sin x$$
$$x \to -\infty \Rightarrow f(x) \cong e^{-x} \cos^2 2x$$

The roots for $|x| \to \infty$ can be approximated accordingly. For $x \to +\infty$, the value of function changes its sign so we have:

$$x \to +\infty \Rightarrow 0 \cong e^{x_r} \sin x_r \qquad \Rightarrow x_r \cong k\pi, \; where \; k \to +\infty$$

On the other hand the situation of $x \to -\infty$ is a bit tricky. Here due to $\cos^2 2x$, the function is mostly positive (as long as governed by $e^{-x} \cos^2 2x$). In other words, the roots will like to be double roots. However, if the other term (even if exponentially small) is positive, then indeed we do not have a root. In that case the function gets very close to x axis but does not touch it. Here there are a few problems:

- Due to zero slope, convergence will be slow for gradient methods (e.g. Newton's method).
- Even the bracketing methods will have troubles, locating initial guess. Because the other small negative term will make the two roots very close together.

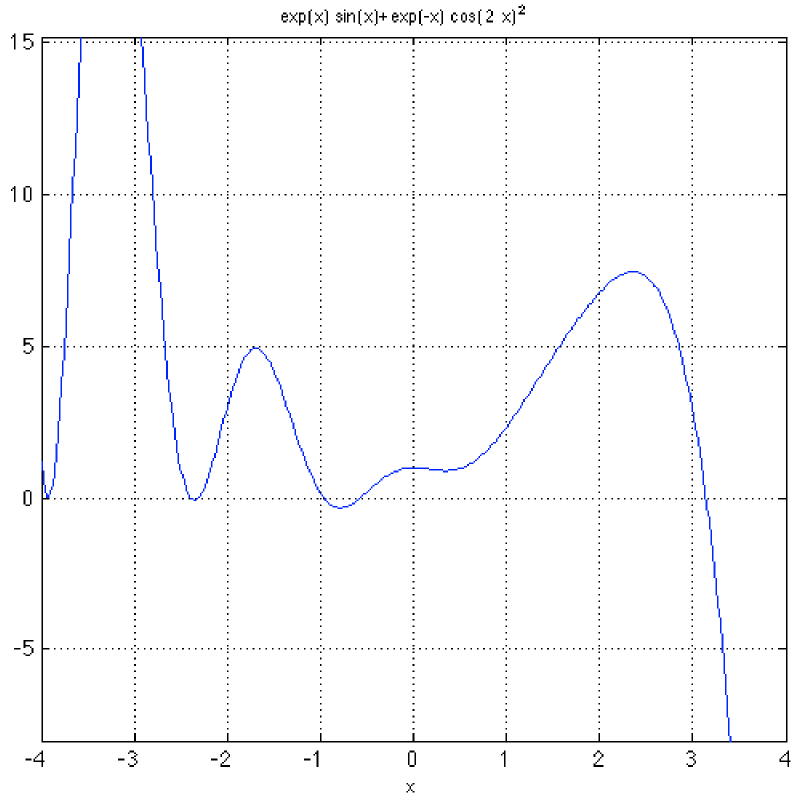
- Practically as long as we get very closed to x axis we have found a root. However, we are mathematically fooled!

Fortunately, in this particular case we know the structure of our functions and we can distinguish the two cases in advance.

$$x \to -\infty \Rightarrow 0 \cong e^{-x_r}\cos^2 2x_r \quad \Rightarrow x_r \cong \frac{k\pi}{2}+\frac{\pi}{4},\ \textit{provided } \sin x_r < 0 \textit{ where } k \to -\infty$$

$$x_r \cong 2k\pi - \frac{\pi}{4}, x_r \cong 2k\pi - \frac{3\pi}{4} \qquad k \to -\infty$$

c) The function is plotted in the $|x| \le 4$. We can zoom in and distinguish 5 roots in the given region. Note that we do not have a root around $2\pi - \frac{3\pi}{4} \cong 3.92$.



The attached program C2p29_PSET2_2.m is used for this problem. The program calls the solver with different methods and initial guesses. In all cases the maximum number of iterations is set to 20 and a relative and absolute error less or equal to 10^{-16} and 10^{-20} are chosen[1]. In particular the output of solver for the root around 3.2 is shown on the next page.

[1] Remember problem set 1 and note that eps(1) is about $2.2*10^{-16}$. As a result when a solution converges, either the consecutive solutions become equal or they fluctuate with a relative error of order 10^{-16}. This can be seen in the plots of next page.

```
k        x_o            x_n            f_o            f_n            df_o           x_rel_error
----------------------------------------------------------------------------------------------
0        +3.10000000    +3.14546479    +9.67744e-01   -4.69069e-02   -2.12856e+01   +1.44541e+00%
1        +3.14546479    +3.14345718    -4.69069e-02   -9.41699e-05   -2.33646e+01   +6.38661e-02%
2        +3.14345718    +3.14345314    -9.41699e-05   -3.82114e-10   -2.32708e+01   +1.28734e-04%
3        +3.14345314    +3.14345314    -3.82114e-10   +4.93355e-15   -2.32707e+01   +5.22376e-10%
4        +3.14345314    +3.14345314    +4.93355e-15   +4.93355e-15   -2.32707e+01   +0.00000e+00%

Final Solution: f(x= +3.1434531358588473)=+4.933554e-15 with NEWTON method and 4 iterations
```

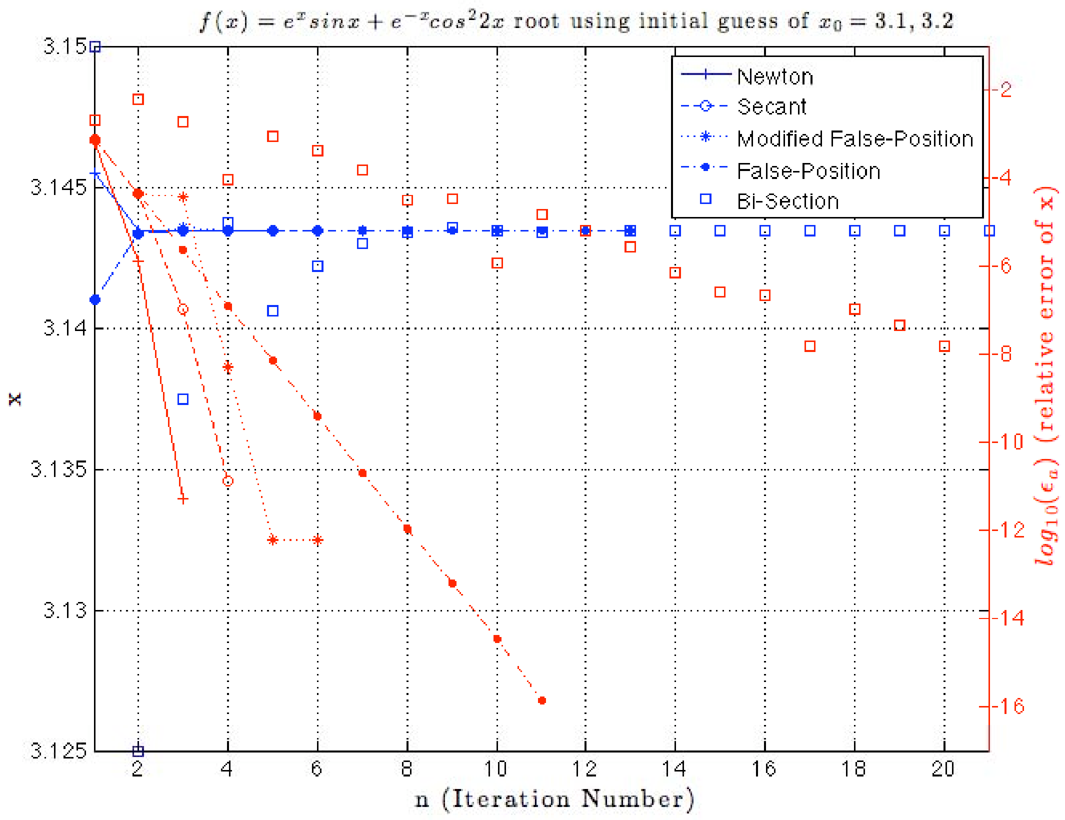
```
k        x_0            x_1            x_2            f_0            f_1            f_2            df_x_1         x_rel_error
----------------------------------------------------------------------------------------------------------------------------
0        +3.10000000    +3.20000000    +3.14101303    +9.67744e-01   -1.39186e+00   +5.66440e-02   -2.35960e+01   +1.30573e+00%
1        +3.20000000    +3.14101303    +3.14331973    -1.39186e+00   +5.66440e-02   +3.10398e-03   -2.45563e+01   +7.33843e-02%
2        +3.14101303    +3.14331973    +3.14345346    +5.66440e-02   +3.10398e-03   -7.60900e-06   -2.32107e+01   +4.25427e-03%
3        +3.14331973    +3.14345346    +3.14345314    +3.10398e-03   -7.60900e-06   +1.01793e-09   -2.32675e+01   +1.04033e-05%
4        +3.14345346    +3.14345314    +3.14345314    -7.60900e-06   +1.01793e-09   +4.93355e-15   -2.32707e+01   +1.39155e-09%
5        +3.14345314    +3.14345314    +3.14345314    +1.01793e-09   +4.93355e-15   +4.93355e-15   -2.32707e+01   +0.00000e+00%

Final Solution: f(x= +3.1434531358588473)=+4.933554e-15 with SECANT method and 5 iterations
```

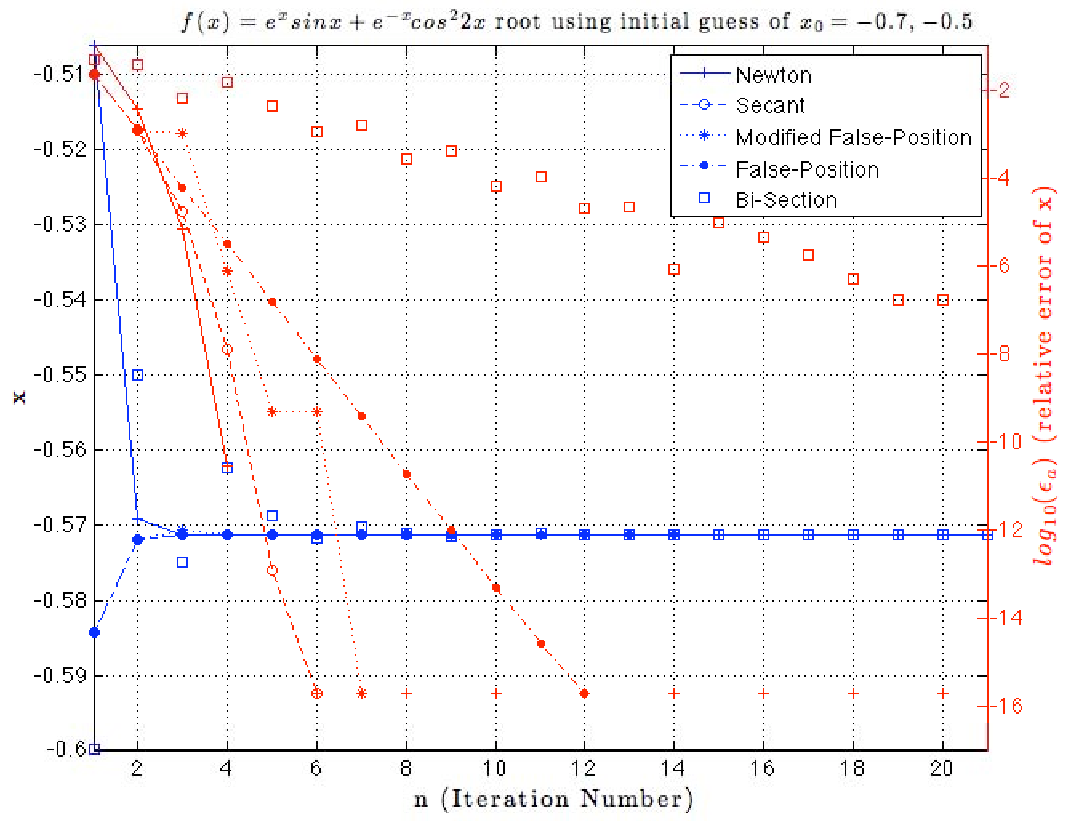
```
k        x_l            x_u            x_n            f_l            f_u            f_n            x_rel_error
-------------------------------------------------------------------------------------------------------------
0        +3.20000000    +3.10000000    +3.14101303    -1.39186e+00   +9.67744e-01   +5.66440e-02   +1.30573e+00%
1        +3.20000000    +3.14101303    +3.14331973    -1.39186e+00   +5.66440e-02   +3.10398e-03   +7.33843e-02%
2        +3.20000000    +3.14331973    +3.14357141    -6.95928e-01   +3.10398e-03   -2.75276e-03   +8.00628e-03%
3        +3.14357141    +3.14331973    +3.14345312    -2.75276e-03   +3.10398e-03   +3.68190e-07   +3.76321e-03%
4        +3.14357141    +3.14345312    +3.14345314    -2.75276e-03   +3.68190e-07   +4.36671e-11   +5.03274e-07%
5        +3.14357141    +3.14345314    +3.14345314    -1.37638e-03   +4.36671e-11   -4.36573e-11   +1.19377e-10%
6        +3.14345314    +3.14345314    +3.14345314    -4.36573e-11   +4.36671e-11   +4.93355e-15   +5.96884e-11%
7        +3.14345314    +3.14345314    +3.14345314    -4.36573e-11   +4.93355e-15   +4.93355e-15   +0.00000e+00%

Final Solution: f(x= +3.1434531358588473)=+4.933554e-15 with MODIFIED FALSE-POSITION method and 7 iterations
```

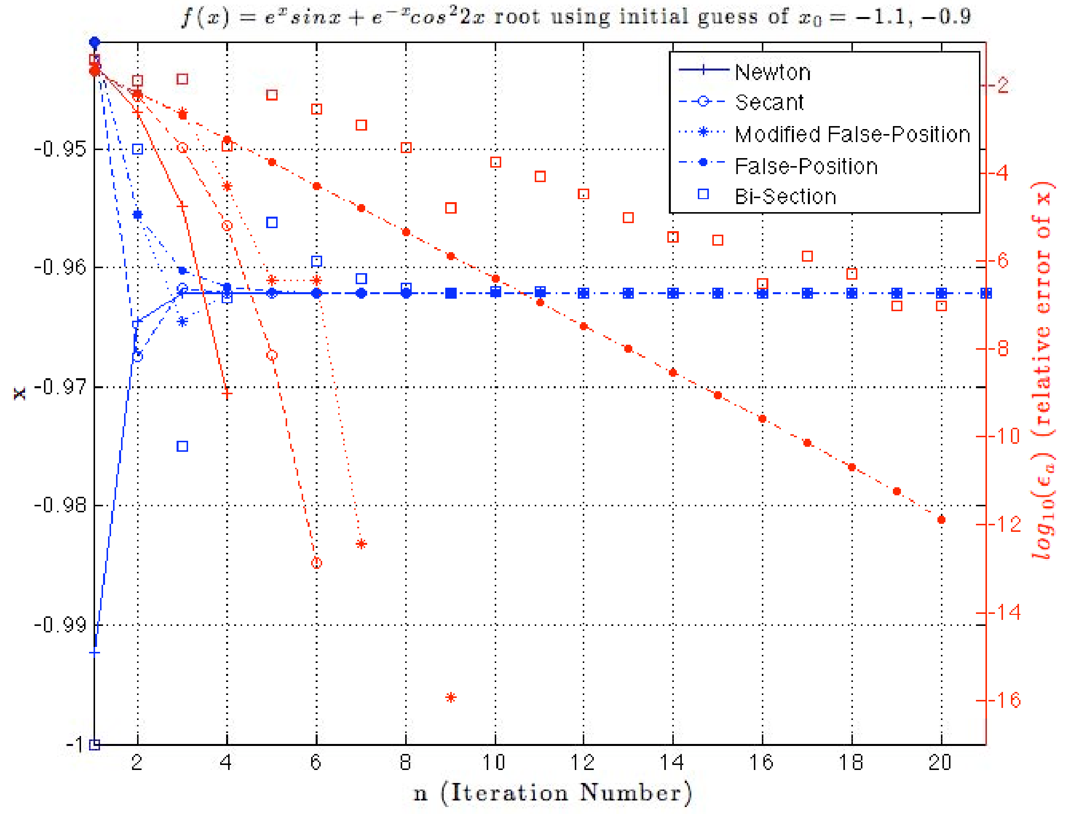
```
k        x_l            x_u            x_n            f_l            f_u            f_n            x_rel_error
-------------------------------------------------------------------------------------------------------------
0        +3.20000000    +3.10000000    +3.14101303    -1.39186e+00   +9.67744e-01   +5.66440e-02   +1.30573e+00%
1        +3.20000000    +3.14101303    +3.14331973    -1.39186e+00   +5.66440e-02   +3.10398e-03   +7.33843e-02%
2        +3.20000000    +3.14331973    +3.14344585    -1.39186e+00   +3.10398e-03   +1.69467e-04   +4.01221e-03%
3        +3.20000000    +3.14344585    +3.14345274    -1.39186e+00   +1.69467e-04   +9.25045e-06   +2.19026e-04%
4        +3.20000000    +3.14345274    +3.14345311    -1.39186e+00   +9.25045e-06   +5.04936e-07   +1.19556e-05%
5        +3.20000000    +3.14345311    +3.14345313    -1.39186e+00   +5.04936e-07   +2.75619e-08   +6.52594e-07%
6        +3.20000000    +3.14345313    +3.14345314    -1.39186e+00   +2.75619e-08   +1.50446e-09   +3.56218e-08%
7        +3.20000000    +3.14345314    +3.14345314    -1.39186e+00   +1.50446e-09   +8.21209e-11   +1.94442e-09%
8        +3.20000000    +3.14345314    +3.14345314    -1.39186e+00   +8.21209e-11   +4.47965e-12   +1.06139e-10%
9        +3.20000000    +3.14345314    +3.14345314    -1.39186e+00   +4.47965e-12   +2.42618e-13   +5.79225e-12%
10       +3.20000000    +3.14345314    +3.14345314    -1.39186e+00   +2.42618e-13   +1.52656e-14   +3.10804e-13%
11       +3.20000000    +3.14345314    +3.14345314    -1.39186e+00   +1.52656e-14   +4.93355e-15   +1.41274e-14%
12       +3.20000000    +3.14345314    +3.14345314    -1.39186e+00   +4.93355e-15   +4.93355e-15   +0.00000e+00%

Final Solution: f(x= +3.1434531358588473)=+4.933554e-15 with FALSE-POSITION method and 12 iterations
```

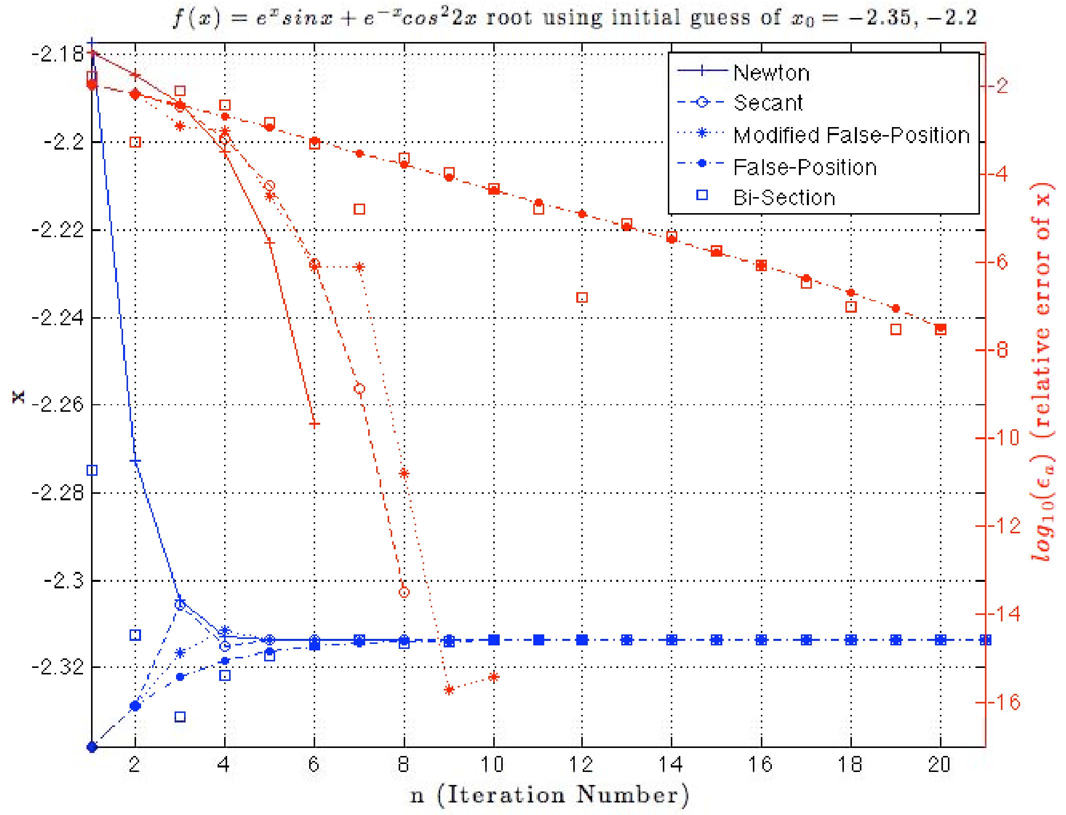
```
k        x_l            x_u            x_n            f_l            f_u            f_n            x_rel_error
-------------------------------------------------------------------------------------------------------------
0        +3.20000000    +3.10000000    +3.15000000    -1.39186e+00   +9.67744e-01   -1.53352e-01   +1.58730e+00%
1        +3.15000000    +3.10000000    +3.12500000    -1.53352e-01   +9.67744e-01   +4.21518e-01   +8.00000e-01%
2        +3.15000000    +3.12500000    +3.13750000    -1.53352e-01   +4.21518e-01   +1.37708e-01   +3.98406e-01%
3        +3.15000000    +3.13750000    +3.14375000    -1.53352e-01   +1.37708e-01   -6.91028e-03   +1.98807e-01%
4        +3.14375000    +3.13750000    +3.14062500    -6.91028e-03   +1.37708e-01   +6.56261e-02   +9.95025e-02%
5        +3.14375000    +3.14062500    +3.14218750    -6.91028e-03   +6.56261e-02   +2.94148e-02   +4.97265e-02%
6        +3.14375000    +3.14218750    +3.14296875    -6.91028e-03   +2.94148e-02   +1.12665e-02   +2.48571e-02%
7        +3.14375000    +3.14296875    +3.14335938    -6.91028e-03   +1.12665e-02   +2.18167e-03   +1.24270e-02%
8        +3.14375000    +3.14335938    +3.14355469    -6.91028e-03   +2.18167e-03   -2.36341e-03   +6.21311e-03%
9        +3.14355469    +3.14335938    +3.14345703    -2.36341e-03   +2.18167e-03   -9.06487e-05   +3.10665e-03%
10       +3.14345703    +3.14335938    +3.14340820    -9.06487e-05   +2.18167e-03   +1.04557e-03   +1.55335e-03%
11       +3.14345703    +3.14340820    +3.14343262    -9.06487e-05   +1.04557e-03   +4.77473e-04   +7.76669e-04%
12       +3.14345703    +3.14343262    +3.14344482    -9.06487e-05   +4.77473e-04   +1.93416e-04   +3.88333e-04%
13       +3.14345703    +3.14344482    +3.14345093    -9.06487e-05   +1.93416e-04   +5.13844e-05   +1.94166e-04%
14       +3.14345703    +3.14345093    +3.14345398    -9.06487e-05   +5.13844e-05   -1.96319e-05   +9.70829e-05%
15       +3.14345398    +3.14345093    +3.14345245    -1.96319e-05   +5.13844e-05   +1.58763e-05   +4.85415e-05%
16       +3.14345398    +3.14345245    +3.14345322    -1.96319e-05   +1.58763e-05   -1.87780e-06   +2.42707e-05%
17       +3.14345322    +3.14345245    +3.14345284    -1.87780e-06   +1.58763e-05   +6.99925e-06   +1.21354e-05%
18       +3.14345322    +3.14345284    +3.14345303    -1.87780e-06   +6.99925e-06   +2.56073e-06   +6.06769e-06%
19       +3.14345322    +3.14345303    +3.14345312    -1.87780e-06   +2.56073e-06   +3.41463e-07   +3.03384e-06%
20       +3.14345322    +3.14345312    +3.14345317    -1.87780e-06   +3.41463e-07   -7.68168e-07   +1.51692e-06%

Final Solution: f(x= +3.1434531688690193)=-7.681683e-07 with BI-SECTION method and 20 iterations
```

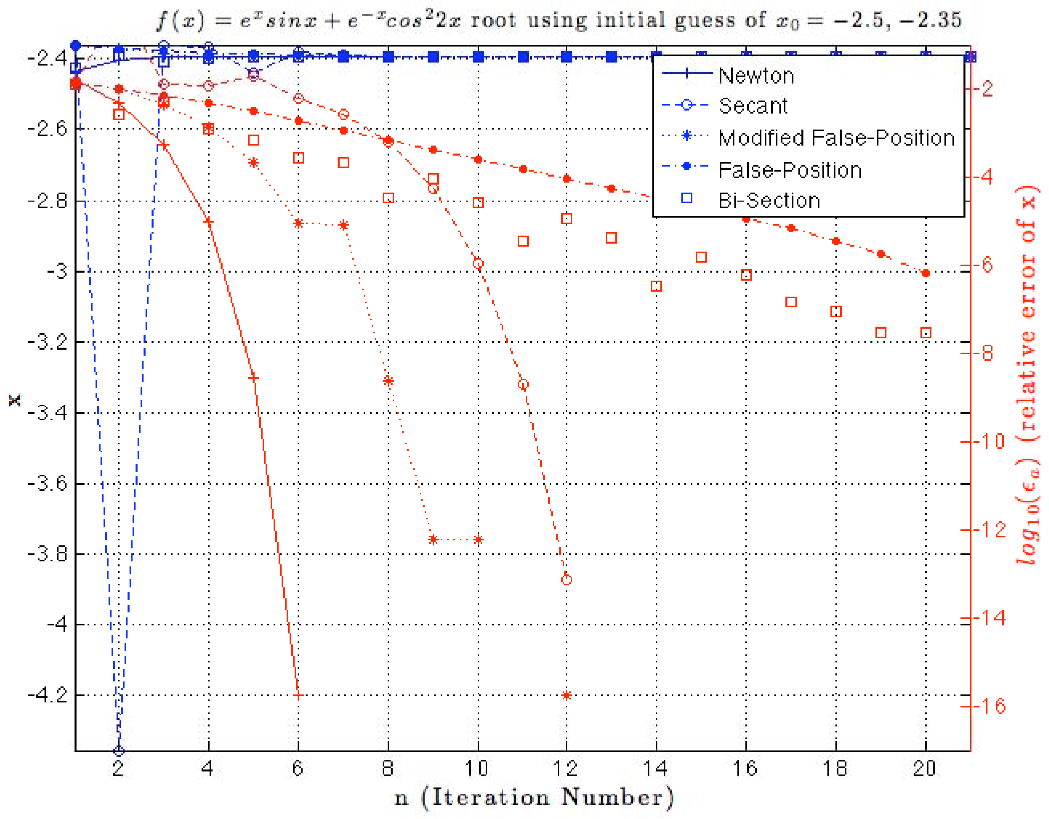
As graph shows in all cases the Newton method's has the best convergence, and usually converges within 7 iterations. After Newton's method the modified false position scheme and secant's scheme are the best. Bi-section is very slow but sometimes is better than false position. The MATLAB output in the next pages shows the solution summary by Newton's method.


$f(x) = e^x sinx + e^{-x}cos^2 2x$ root using initial guess of $x_0 = 3.1, 3.2$
Newton
Secant
Modified False-Position
False-Position
Bi-Section
x
n (Iteration Number)
$log_{10}(\epsilon_a)$ (relative error of x)
3.15
3.145
3.14
3.135
3.13
3.125
-2
-4
-6
-8
-10
-12
-14
-16
2
4
6
8
10
12
14
16
18
20


$f(x) = e^x sinx + e^{-x}cos^2 2x$ root using initial guess of $x_0 = -0.7, -0.5$
Newton
Secant
Modified False-Position
False-Position
Bi-Section
x
n (Iteration Number)
$log_{10}(\epsilon_a)$ (relative error of x)
-0.51
-0.52
-0.53
-0.54
-0.55
-0.56
-0.57
-0.58
-0.59
-0.6
-2
-4
-6
-8
-10
-12
-14
-16
2
4
6
8
10
12
14
16
18
20


$f(x) = e^x sinx + e^{-x}cos^2 2x$ root using initial guess of $x_0 = -1.1, -0.9$
Newton
Secant
Modified False-Position
False-Position
Bi-Section
x
n (Iteration Number)
$log_{10}(\epsilon_a)$ (relative error of x)
-0.95
-0.96
-0.97
-0.98
-0.99
-1
2
4
6
8
10
12
14
16
18
20
-2
-4
-6
-8
-10
-12
-14
-16

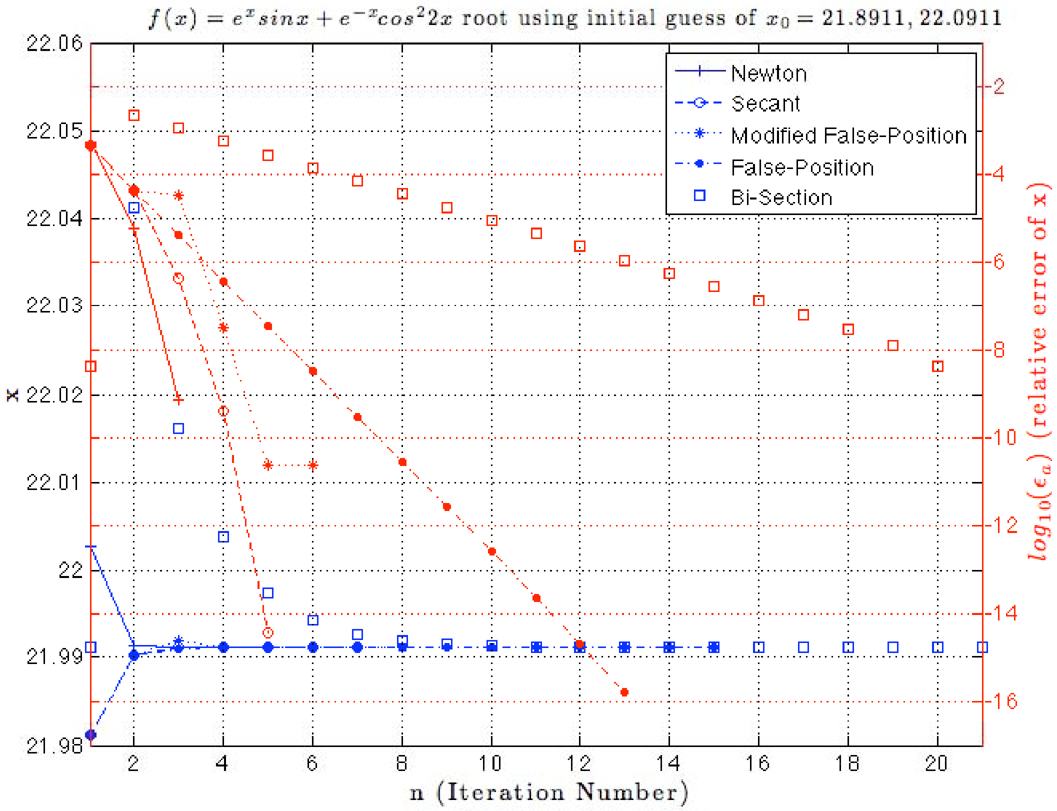

$f(x) = e^x sinx + e^{-x}cos^2 2x$ root using initial guess of $x_0 = -2.35, -2.2$
Newton
Secant
Modified False-Position
False-Position
Bi-Section
x
n (Iteration Number)
$log_{10}(\epsilon_a)$ (relative error of x)
-2.18
-2.2
-2.22
-2.24
-2.26
-2.28
-2.3
-2.32
2
4
6
8
10
12
14
16
18
20
-2
-4
-6
-8
-10
-12
-14
-16

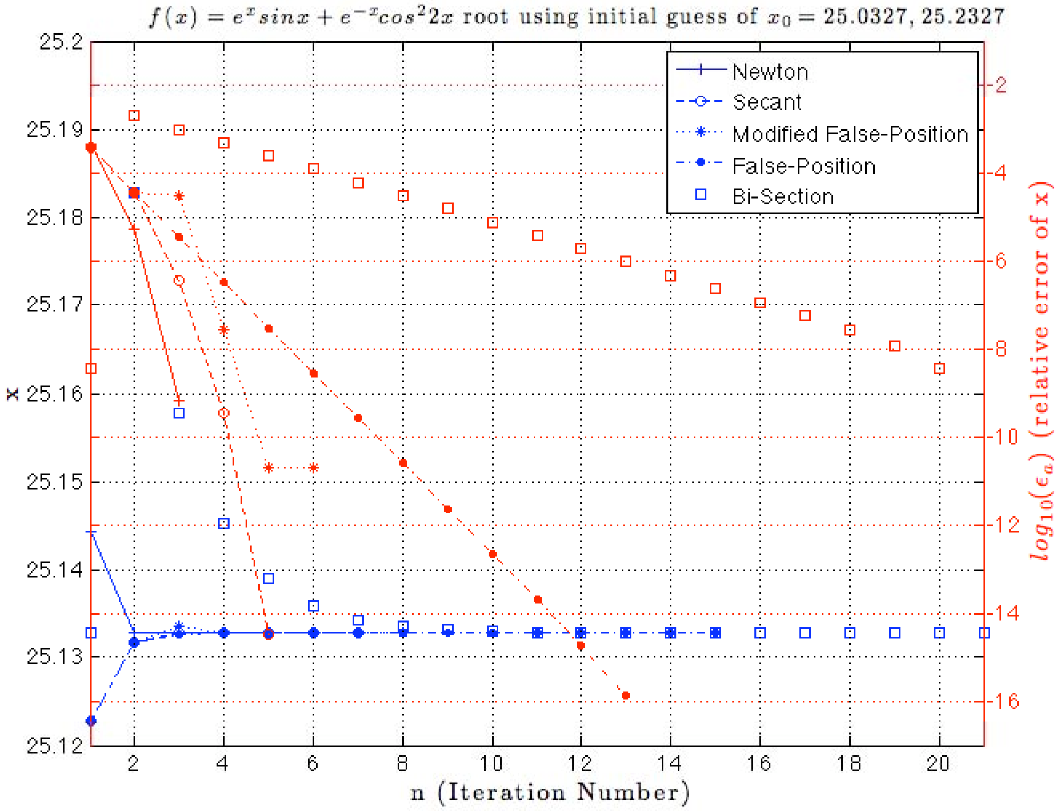


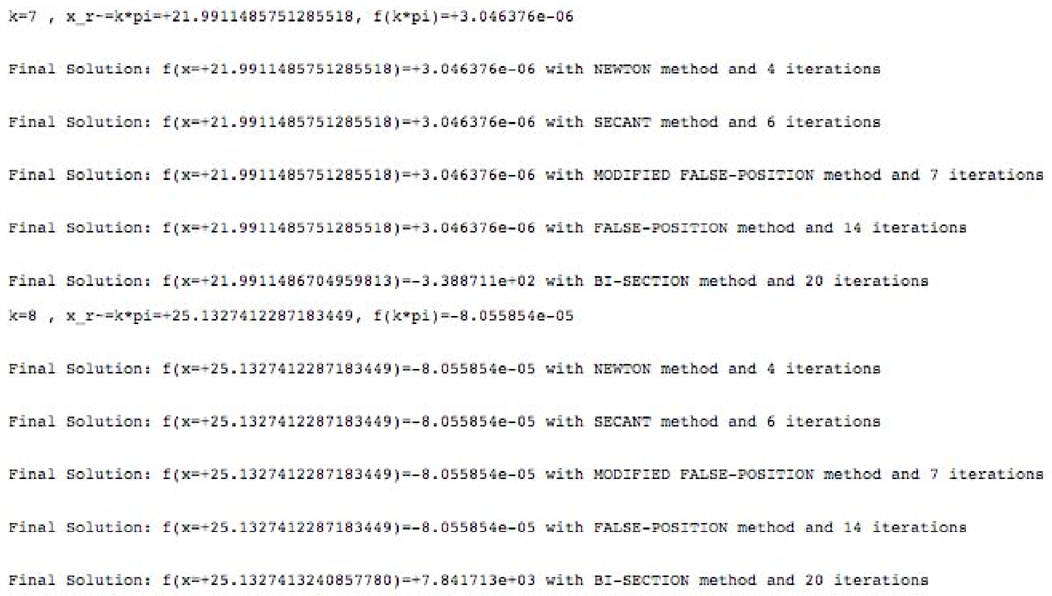
Here are the solutions by Newton'n method. Note that approximate solutions are still very good, albeit x is not very large $\left(\pi \cong 3.14,\ -\frac{\pi}{4} = -0.79,\ -\frac{3\pi}{4} = -2.36\right)$.

```
Final Solution: f(x= +3.1434531358588473)=+4.933554e-15 with NEWTON method and 4 iterations

Final Solution: f(x= -0.5712519376020334)=-1.665335e-16 with NEWTON method and 20 iterations

Final Solution: f(x= -0.9620937727804407)=-1.110223e-16 with NEWTON method and 5 iterations

Final Solution: f(x= -2.3137081787813187)=-5.689893e-16 with NEWTON method and 7 iterations

Final Solution: f(x= -2.3938636326889564)=+5.689893e-16 with NEWTON method and 7 iterations
```

d) Since $\frac{20}{\pi} \cong 6.37$, the roots are expected to be around $7\pi, 8\pi$. Initial guesses of $7\pi \pm 0.1, 8\pi \pm 0.1$ are used. The plots are shown and the program output is shown as well. Note that in the previous case the absolute value of the function is of order 10^{-16} at the approximate root, while here cannot get better than 10^{-6}. This is due to very steep behavior of function for $|x| \to \infty$. Furthermore, none of the method has been able to provide us with a better root than we approximated ahead of time. This is due to limited digits for computer representation and our very good approximate.


$f(x) = e^x sinx + e^{-x}cos^2 2x$ root using initial guess of $x_0 = 21.8911, 22.0911$
Newton
Secant
Modified False-Position
False-Position
Bi-Section
x
n (Iteration Number)
$log_{10}(\epsilon_a)$ (relative error of x)

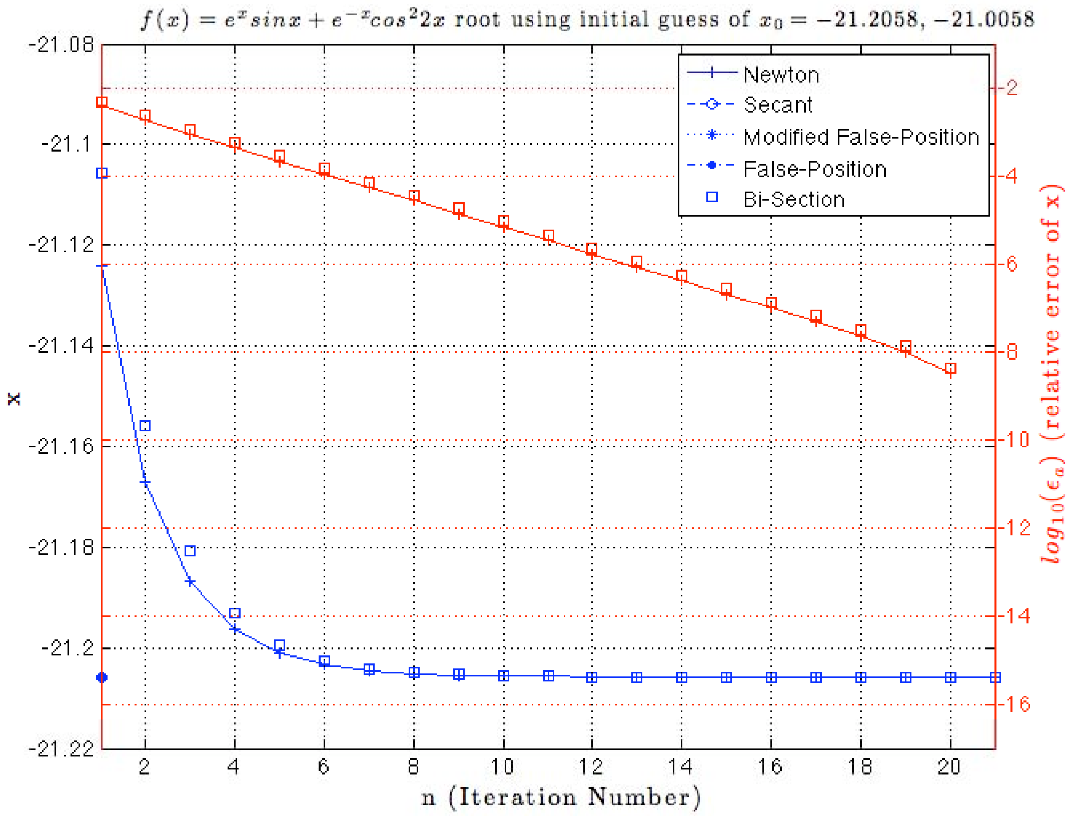

$f(x) = e^x sinx + e^{-x}cos^2 2x$ root using initial guess of $x_0 = 25.0327, 25.2327$
Newton
Secant
Modified False-Position
False-Position
Bi-Section
x
n (Iteration Number)
$log_{10}(\epsilon_a)$ (relative error of x)

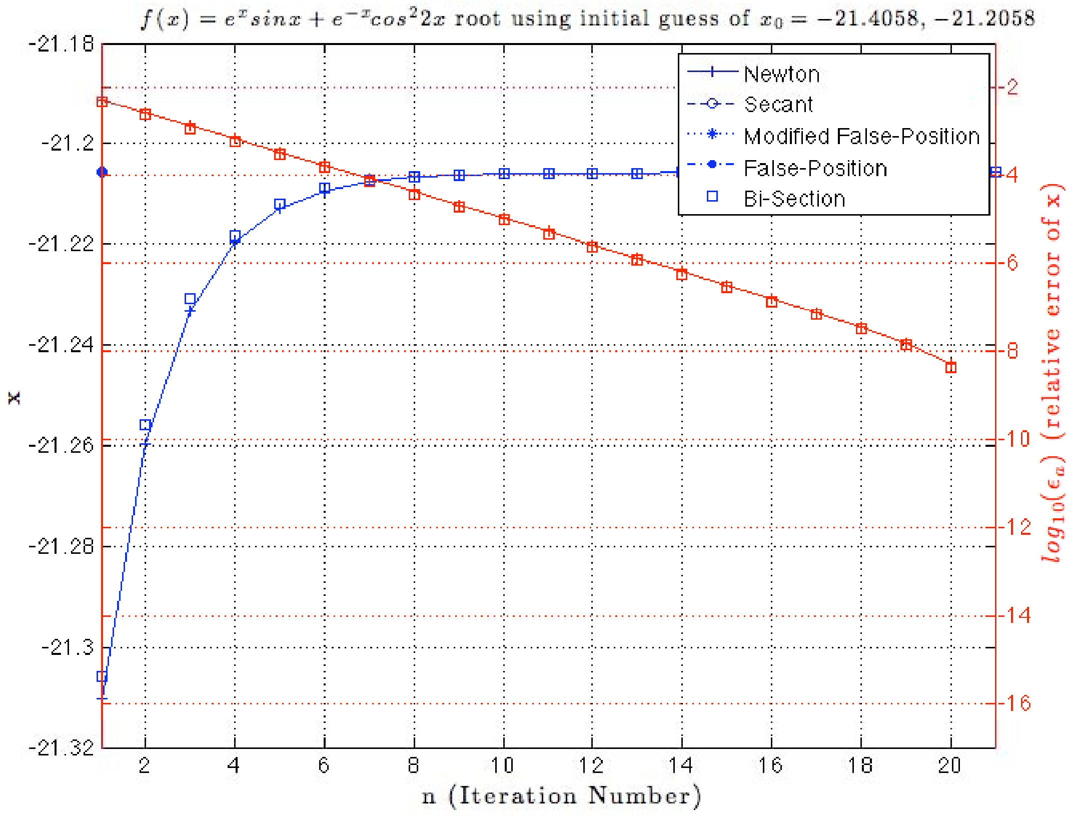


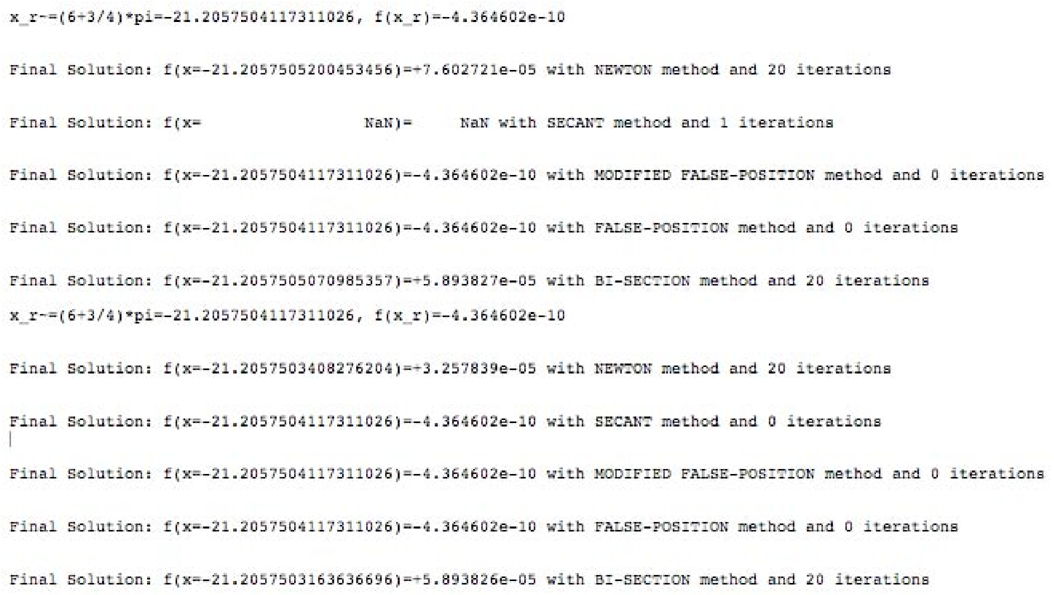
```
k=7 , x_r-=k*pi=+21.9911485751285518, f(k*pi)=+3.046376e-06

Final Solution: f(x=+21.9911485751285518)=+3.046376e-06 with NEWTON method and 4 iterations

Final Solution: f(x=+21.9911485751285518)=+3.046376e-06 with SECANT method and 6 iterations

Final Solution: f(x=+21.9911485751285518)=+3.046376e-06 with MODIFIED FALSE-POSITION method and 7 iterations

Final Solution: f(x=+21.9911485751285518)=+3.046376e-06 with FALSE-POSITION method and 14 iterations

Final Solution: f(x=+21.9911486704959813)=-3.388711e+02 with BI-SECTION method and 20 iterations
k=8 , x_r-=k*pi=+25.1327412287183449, f(k*pi)=-8.055854e-05

Final Solution: f(x=+25.1327412287183449)=-8.055854e-05 with NEWTON method and 4 iterations

Final Solution: f(x=+25.1327412287183449)=-8.055854e-05 with SECANT method and 6 iterations

Final Solution: f(x=+25.1327412287183449)=-8.055854e-05 with MODIFIED FALSE-POSITION method and 7 iterations

Final Solution: f(x=+25.1327412287183449)=-8.055854e-05 with FALSE-POSITION method and 14 iterations

Final Solution: f(x=+25.1327413240857780)=+7.841713e+03 with BI-SECTION method and 20 iterations
```

e) $\frac{20}{2\pi} = 3.18,\ -(2\pi \times 3 + \frac{\pi}{4}) \cong -19.63, -(2\pi \times 4 + \frac{\pi}{4}) \cong -25.91, -(2\pi \times 3 + \frac{3\pi}{4}) \cong -21.21$

Consequently, the roots are expected to be around $-(2\pi \times 3 + \frac{3\pi}{4}) \cong -21.21$. This is almost a double root and the pair of initial guesses is constructed with ± 0.2 offset from this middle point. To force the scheme to find the right or left root $(-(6\pi + \frac{3\pi}{4}) < x_r \ \ or \ \ -(6\pi + \frac{3\pi}{4}) > x_r)$ the offset value is used for Newton's method. Results and plots are shown on the next pages. Like part "d" no method has been able to provide us with a better approximate of solution, than our analytical approximate (the too roots are not distinguishable). The bottom line is that sometimes we have to really work out our way with analytical schemes. Indeed if did not have that good initial guess, the solution could not even converge. Finally note that $f(-(6\pi + \frac{3\pi}{4})) \cong -4.3 \times 10^{-10}$ while $f(-(6\pi + \frac{3\pi}{4}) \pm 0.2) \cong 2.5 \times 10^{8}$; consequently the secant and (modified) false position will choose $-(6\pi + \frac{3\pi}{4})$ as the right solution in a single iteration. On the other hand both Bi-section and Newton method will suffer from $-(6\pi + \frac{3\pi}{4}) \pm 0.2$ and will converge more slowly. Furthermore, recall that Newton's method has slower convergence in multiple roots and this will also contribute to our slow convergence rate.

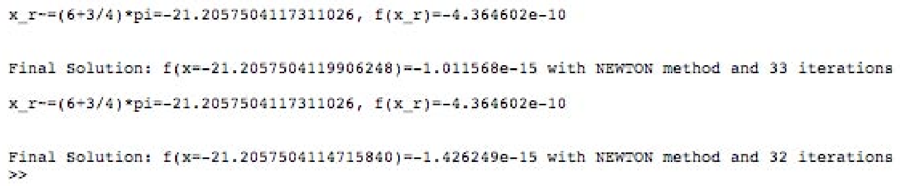

$f(x) = e^x sinx + e^{-x}cos^2 2x$ root using initial guess of $x_0 = -21.2058, -21.0058$
Newton
Secant
Modified False-Position
False-Position
Bi-Section
x
n (Iteration Number)
$log_{10}(\epsilon_a)$ (relative error of x)
-21.08
-21.1
-21.12
-21.14
-21.16
-21.18
-21.2
-21.22
-2
-4
-6
-8
-10
-12
-14
-16
2
4
6
8
10
12
14
16
18
20


$f(x) = e^x sinx + e^{-x}cos^2 2x$ root using initial guess of $x_0 = -21.4058, -21.2058$
Newton
Secant
Modified False-Position
False-Position
Bi-Section
x
n (Iteration Number)
$log_{10}(\epsilon_a)$ (relative error of x)
-21.18
-21.2
-21.22
-21.24
-21.26
-21.28
-21.3
-21.32
-2
-4
-6
-8
-10
-12
-14
-16
2
4
6
8
10
12
14
16
18
20



```
x_r~=(6+3/4)*pi=-21.2057504117311026, f(x_r)=-4.364602e-10

Final Solution: f(x=-21.2057505200453456)=+7.602721e-05 with NEWTON method and 20 iterations

Final Solution: f(x=                NaN)=     NaN with SECANT method and 1 iterations

Final Solution: f(x=-21.2057504117311026)=-4.364602e-10 with MODIFIED FALSE-POSITION method and 0 iterations

Final Solution: f(x=-21.2057504117311026)=-4.364602e-10 with FALSE-POSITION method and 0 iterations

Final Solution: f(x=-21.2057505070985357)=+5.893827e-05 with BI-SECTION method and 20 iterations
x_r~=(6+3/4)*pi=-21.2057504117311026, f(x_r)=-4.364602e-10

Final Solution: f(x=-21.2057503408276204)=+3.257839e-05 with NEWTON method and 20 iterations

Final Solution: f(x=-21.2057504117311026)=-4.364602e-10 with SECANT method and 0 iterations
|
Final Solution: f(x=-21.2057504117311026)=-4.364602e-10 with MODIFIED FALSE-POSITION method and 0 iterations

Final Solution: f(x=-21.2057504117311026)=-4.364602e-10 with FALSE-POSITION method and 0 iterations

Final Solution: f(x=-21.2057503163636696)=+5.893826e-05 with BI-SECTION method and 20 iterations
```

Note that if we allow the Newton's method to have more iterations, it will again pick up the right solutions and will distinguish the two left and right roots (which only differ in 13th digit). Here we have used the same initial guesses, but we needed 33 iterations to achieve the desired relative error.



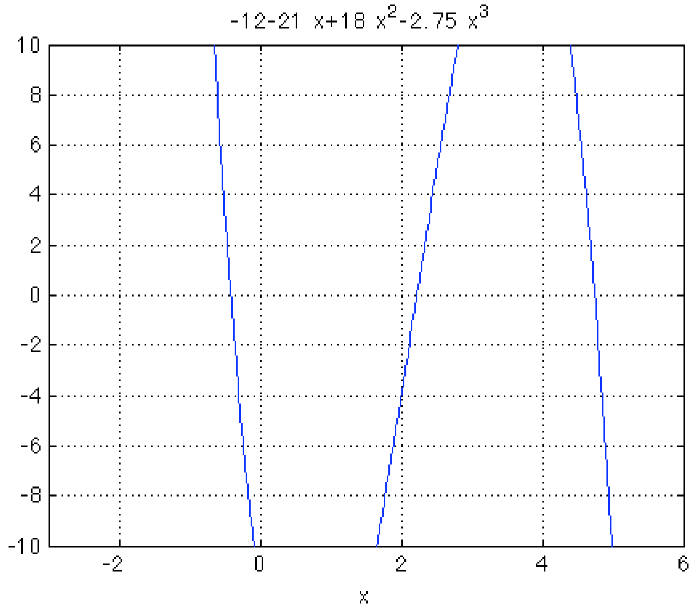
```
x_r~=(6+3/4)*pi=-21.2057504117311026, f(x_r)=-4.364602e-10

Final Solution: f(x=-21.2057504119906248)=-1.011568e-15 with NEWTON method and 33 iterations
x_r~=(6+3/4)*pi=-21.2057504117311026, f(x_r)=-4.364602e-10

Final Solution: f(x=-21.2057504114715840)=-1.426249e-15 with NEWTON method and 32 iterations
>>
```

Problem 2.3 (65 Points): Textbook problem

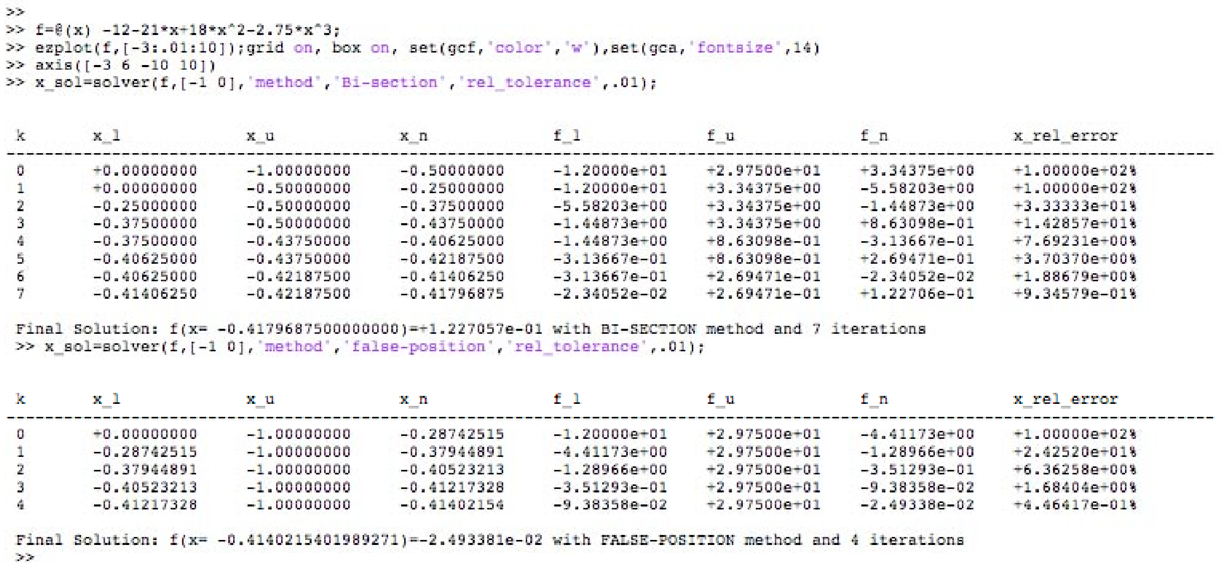
Solve the below problems from "Chapara and Canale" textbook:

- 5.4, 5.9, 5.15, 5.17 (Use your previous program if you can, then copy and paste the results)
- 6.1, 6.11, 6.15, 6.16, 6.23
- EXTRA CREDIT: 6.25 (5 Points)
- 10.6, 10.9, 10.12, 10.14

Textbook problem 5.4

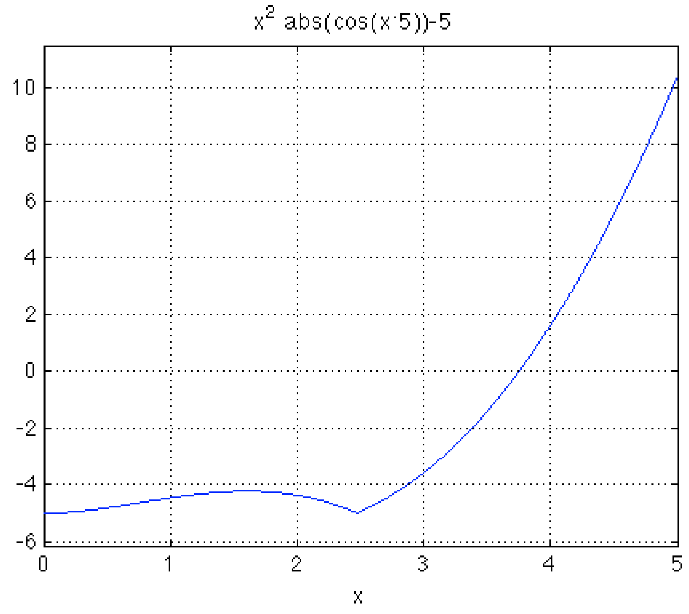


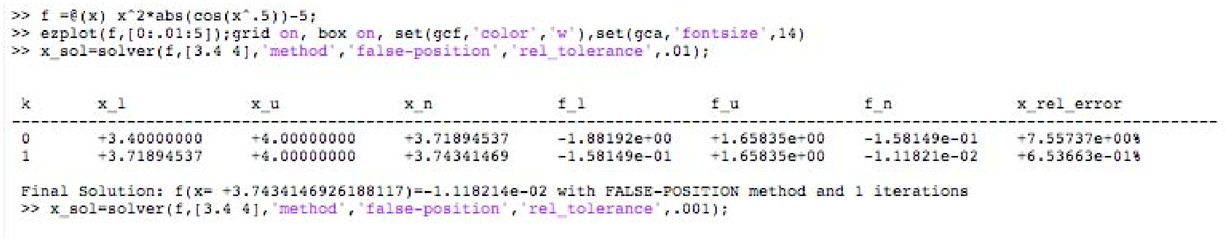
By zooming in the graph we can see that roots are approximately -0.41, 2.22 and 4.74.



```
>>
>> f=@(x) -12-21*x+18*x^2-2.75*x^3;
>> ezplot(f,[-3:.01:10]);grid on, box on, set(gcf,'color','w'),set(gca,'fontsize',14)
>> axis([-3 6 -10 10])
>> x_sol=solver(f,[-1 0],'method','Bi-section','rel_tolerance',.01);

 k       x_l         x_u         x_n         f_l          f_u          f_n          x_rel_error
---------------------------------------------------------------------------------------------------
 0       +0.00000000 -1.00000000 -0.50000000 -1.20000e+01 +2.97500e+01 +3.34375e+00 +1.00000e+02%
 1       +0.00000000 -0.50000000 -0.25000000 -1.20000e+01 +3.34375e+00 -5.58203e+00 +1.00000e+02%
 2       -0.25000000 -0.50000000 -0.37500000 -5.58203e+00 +3.34375e+00 -1.44873e+00 +3.33333e+01%
 3       -0.37500000 -0.50000000 -0.43750000 -1.44873e+00 +3.34375e+00 +8.63098e-01 +1.42857e+01%
 4       -0.37500000 -0.43750000 -0.40625000 -1.44873e+00 +8.63098e-01 -3.13667e-01 +7.69231e+00%
 5       -0.40625000 -0.43750000 -0.42187500 -3.13667e-01 +8.63098e-01 +2.69471e-01 +3.70370e+00%
 6       -0.40625000 -0.42187500 -0.41406250 -3.13667e-01 +2.69471e-01 -2.34052e-02 +1.88679e+00%
 7       -0.41406250 -0.42187500 -0.41796875 -2.34052e-02 +2.69471e-01 +1.22706e-01 +9.34579e-01%

 Final Solution: f(x= -0.4179687500000000)=+1.227057e-01 with BI-SECTION method and 7 iterations
 >> x_sol=solver(f,[-1 0],'method','false-position','rel_tolerance',.01);

 k       x_l         x_u         x_n         f_l          f_u          f_n          x_rel_error
---------------------------------------------------------------------------------------------------
 0       +0.00000000 -1.00000000 -0.28742515 -1.20000e+01 +2.97500e+01 -4.41173e+00 +1.00000e+02%
 1       -0.28742515 -1.00000000 -0.37944891 -4.41173e+00 +2.97500e+01 -1.28966e+00 +2.42520e+01%
 2       -0.37944891 -1.00000000 -0.40523213 -1.28966e+00 +2.97500e+01 -3.51293e-01 +6.36258e+00%
 3       -0.40523213 -1.00000000 -0.41217328 -3.51293e-01 +2.97500e+01 -9.38358e-02 +1.68404e+00%
 4       -0.41217328 -1.00000000 -0.41402154 -9.38358e-02 +2.97500e+01 -2.49338e-02 +4.46417e-01%

 Final Solution: f(x= -0.4140215401989271)=-2.493381e-02 with FALSE-POSITION method and 4 iterations
 >>
```

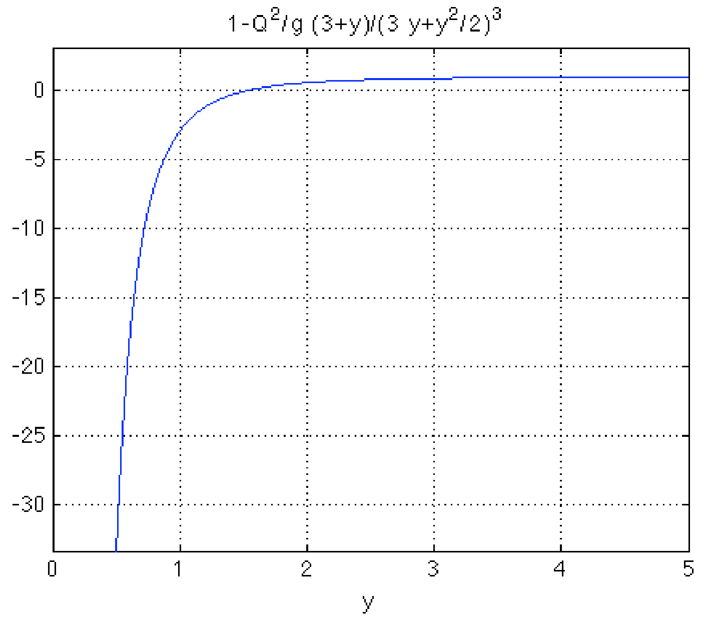
Textbook problem 5.9



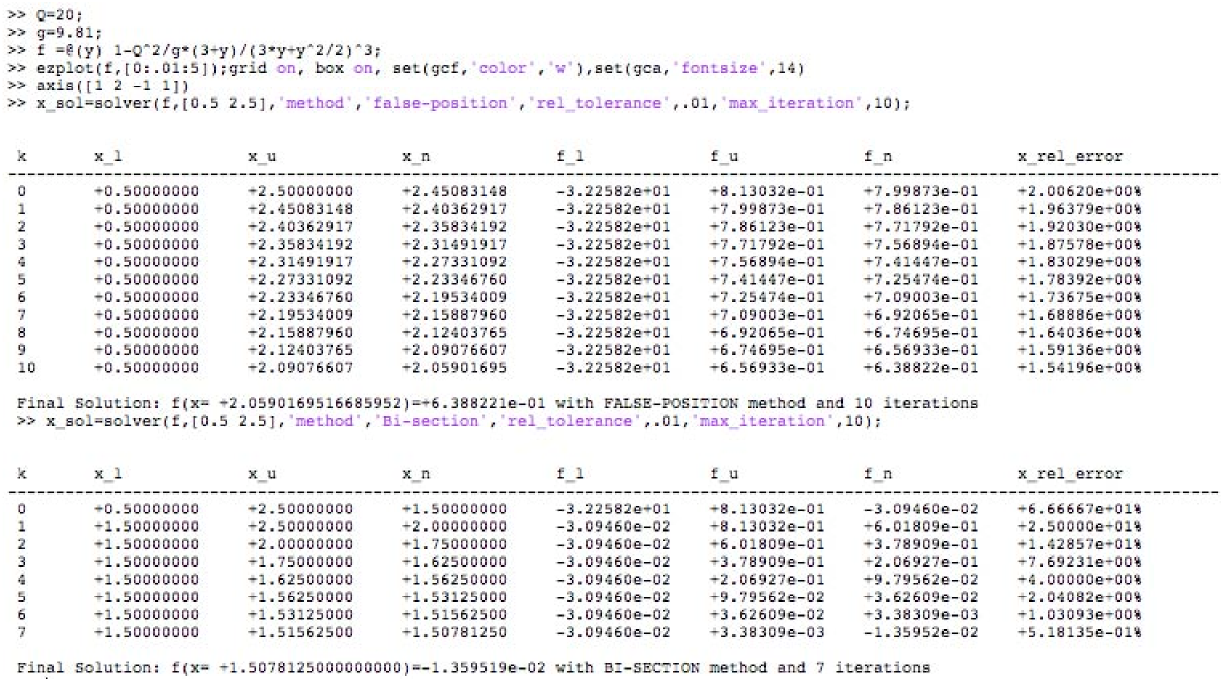


```
>> f =@(x) x^2*abs(cos(x^.5))-5;
>> ezplot(f,[0:.01:5]);grid on, box on, set(gcf,'color','w'),set(gca,'fontsize',14)
>> x_sol=solver(f,[3.4 4],'method','false-position','rel_tolerance',.01);

 k       x_l             x_u             x_n             f_l             f_u             f_n             x_rel_error
--------------------------------------------------------------------------------------------------------------------------
 0       +3.40000000     +4.00000000     +3.71894537     -1.88192e+00    +1.65835e+00    -1.58149e-01    +7.55737e+00%
 1       +3.71894537     +4.00000000     +3.74341469     -1.58149e-01    +1.65835e+00    -1.11821e-02    +6.53663e-01%

 Final Solution: f(x= +3.7434146926188117)=-1.118214e-02 with FALSE-POSITION method and 1 iterations
 >> x_sol=solver(f,[3.4 4],'method','false-position','rel_tolerance',.001);
```

Textbook problem 5.15



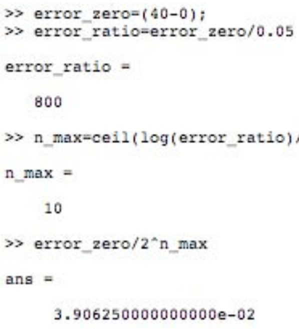
By zooming in the graph we can see that root is approximated by y=1.51 (m). Note as it can be seen the function has considerable curvature around the root. On the other hand (0.5+2.5)/2=1.5 is very close to actual root x=1.5140 and consequently Bi-section works better than false position. Furthermore, the false position method is prone to the same problem depicted in figure 5.14 of the textbook.



```
>> Q=20;
>> g=9.81;
>> f =@(y) 1-Q^2/g*(3+y)/(3*y+y^2/2)^3;
>> ezplot(f,[0:.01:5]);grid on, box on, set(gcf,'color','w'),set(gca,'fontsize',14)
>> axis([1 2 -1 1])
>> x_sol=solver(f,[0.5 2.5],'method','false-position','rel_tolerance',.01,'max_iteration',10);

 k       x_l            x_u            x_n            f_l            f_u            f_n            x_rel_error
--------------------------------------------------------------------------------------------------------------------
 0       +0.50000000    +2.50000000    +2.45083148    -3.22582e+01   +8.13032e-01   +7.99873e-01   +2.00620e+00%
 1       +0.50000000    +2.45083148    +2.40362917    -3.22582e+01   +7.99873e-01   +7.86123e-01   +1.96379e+00%
 2       +0.50000000    +2.40362917    +2.35834192    -3.22582e+01   +7.86123e-01   +7.71792e-01   +1.92030e+00%
 3       +0.50000000    +2.35834192    +2.31491917    -3.22582e+01   +7.71792e-01   +7.56894e-01   +1.87578e+00%
 4       +0.50000000    +2.31491917    +2.27331092    -3.22582e+01   +7.56894e-01   +7.41447e-01   +1.83029e+00%
 5       +0.50000000    +2.27331092    +2.23346760    -3.22582e+01   +7.41447e-01   +7.25474e-01   +1.78392e+00%
 6       +0.50000000    +2.23346760    +2.19534009    -3.22582e+01   +7.25474e-01   +7.09003e-01   +1.73675e+00%
 7       +0.50000000    +2.19534009    +2.15887960    -3.22582e+01   +7.09003e-01   +6.92065e-01   +1.68886e+00%
 8       +0.50000000    +2.15887960    +2.12403765    -3.22582e+01   +6.92065e-01   +6.74695e-01   +1.64036e+00%
 9       +0.50000000    +2.12403765    +2.09076607    -3.22582e+01   +6.74695e-01   +6.56933e-01   +1.59136e+00%
 10      +0.50000000    +2.09076607    +2.05901695    -3.22582e+01   +6.56933e-01   +6.38822e-01   +1.54196e+00%

 Final Solution: f(x= +2.0590169516685952)=+6.388221e-01 with FALSE-POSITION method and 10 iterations
 >> x_sol=solver(f,[0.5 2.5],'method','Bi-section','rel_tolerance',.01,'max_iteration',10);

 k       x_l            x_u            x_n            f_l            f_u            f_n            x_rel_error
--------------------------------------------------------------------------------------------------------------------
 0       +0.50000000    +2.50000000    +1.50000000    -3.22582e+01   +8.13032e-01   -3.09460e-02   +6.66667e+01%
 1       +1.50000000    +2.50000000    +2.00000000    -3.09460e-02   +8.13032e-01   +6.01809e-01   +2.50000e+01%
 2       +1.50000000    +2.00000000    +1.75000000    -3.09460e-02   +6.01809e-01   +3.78909e-01   +1.42857e+01%
 3       +1.50000000    +1.75000000    +1.62500000    -3.09460e-02   +3.78909e-01   +2.06927e-01   +7.69231e+00%
 4       +1.50000000    +1.62500000    +1.56250000    -3.09460e-02   +2.06927e-01   +9.79562e-02   +4.00000e+00%
 5       +1.50000000    +1.56250000    +1.53125000    -3.09460e-02   +9.79562e-02   +3.62609e-02   +2.04082e+00%
 6       +1.50000000    +1.53125000    +1.51562500    -3.09460e-02   +3.62609e-02   +3.38309e-03   +1.03093e+00%
 7       +1.50000000    +1.51562500    +1.50781250    -3.09460e-02   +3.38309e-03   -1.35952e-02   +5.18135e-01%

 Final Solution: f(x= +1.5078125000000000)=-1.359519e-02 with BI-SECTION method and 7 iterations
```

Textbook problem 5.17

a) In each iteration, the maximum absolute error is reduced by a factor of 2. The initial error is at most $40°C$ and by the end of the nth iteration it will be at most $\frac{40°C}{2^n}$. Now we have to solve $\frac{40°C}{2^n} \le 0.05°C$. The minimum n to satisfy this inequality is n=10 (which results in absolute error less than $0.039°C$) and we at least need 10 iterations.



```
>> error_zero=(40-0);
>> error_ratio=error_zero/0.05

error_ratio =

   800

>> n_max=ceil(log(error_ratio)/log(2))

n_max =

    10

>> error_zero/2^n_max

ans =

      3.906250000000000e-02
```

b) The solver.m file has been used. Slight modification are made to the file so the absolute error is only considered for x value (not the function value).

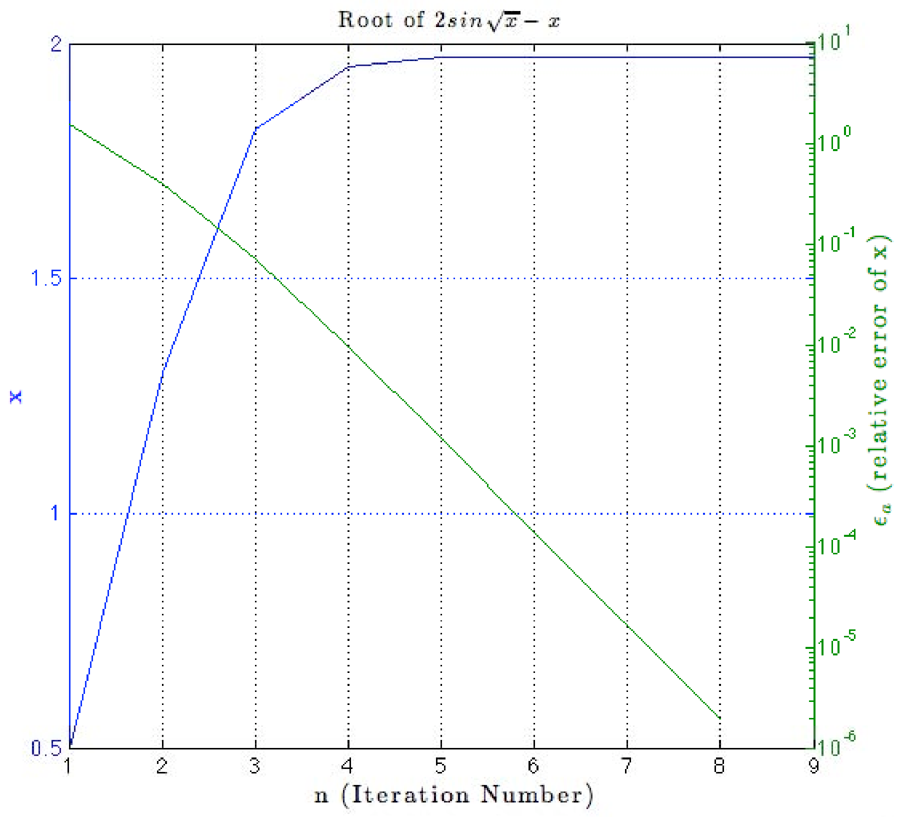
```
>> Tk=273.15;
>> f_c=@(c) -139.34411+1.575701e5/(c+Tk)-6.642308e7/(c+Tk)^2+1.243800e10/(c+Tk)^3-8.621949e11/(c+Tk)^4;
>> o_sf=8;
>> f=@(c) f_c(c)-log(o_sf);
>> C_sol=solver(f,[0 40],'method','Bi-section','abs_tolerance',0.05);

 k       x_l             x_u             x_n             f_l             f_u             f_n             x_rel_error
----------------------------------------------------------------------------------------------------------------------------
 0       +40.00000000    +0.00000000     +20.00000000    -2.21158e-01    +6.03006e-01    +1.28000e-01    +1.00000e+02%
 1       +40.00000000    +20.00000000    +30.00000000    -2.21158e-01    +1.28000e-01    -5.67296e-02    +3.33333e+01%
 2       +30.00000000    +20.00000000    +25.00000000    -5.67296e-02    +1.28000e-01    +3.24014e-02    +2.00000e+01%
 3       +30.00000000    +25.00000000    +27.50000000    -5.67296e-02    +3.24014e-02    -1.28838e-02    +9.09091e+00%
 4       +27.50000000    +25.00000000    +26.25000000    -1.28838e-02    +3.24014e-02    +9.56786e-03    +4.76190e+00%
 5       +27.50000000    +26.25000000    +26.87500000    -1.28838e-02    +9.56786e-03    -1.70433e-03    +2.32558e+00%
 6       +26.87500000    +26.25000000    +26.56250000    -1.70433e-03    +9.56786e-03    +3.92000e-03    +1.17647e+00%
 7       +26.87500000    +26.56250000    +26.71875000    -1.70433e-03    +3.92000e-03    +1.10491e-03    +5.84795e-01%
 8       +26.87500000    +26.71875000    +26.79687500    -1.70433e-03    +1.10491e-03    -3.00437e-04    +2.91545e-01%
 9       +26.79687500    +26.71875000    +26.75781250    -3.00437e-04    +1.10491e-03    +4.02057e-04    +1.45985e-01%

 Final Solution: f(x=+26.7578125000000000)=+4.020565e-04 with BI-SECTION method and 9 iterations
 >> O_c=exp(f_c(C_sol))

O_c =

     8.003217098699499e+00

>> o_sf=10;
>> f=@(c) f_c(c)-log(o_sf);
>> C_sol=solver(f,[0 40],'method','Bi-section','abs_tolerance',0.05);

 k       x_l             x_u             x_n             f_l             f_u             f_n             x_rel_error
----------------------------------------------------------------------------------------------------------------------------
 0       +40.00000000    +0.00000000     +20.00000000    -4.44301e-01    +3.79862e-01    -9.51433e-02    +1.00000e+02%
 1       +20.00000000    +0.00000000     +10.00000000    -9.51433e-02    +3.79862e-01    +1.21150e-01    +1.00000e+02%
 2       +20.00000000    +10.00000000    +15.00000000    -9.51433e-02    +1.21150e-01    +8.35087e-03    +3.33333e+01%
 3       +20.00000000    +15.00000000    +17.50000000    -9.51433e-02    +8.35087e-03    -4.44723e-02    +1.42857e+01%
 4       +17.50000000    +15.00000000    +16.25000000    -4.44723e-02    +8.35087e-03    -1.83407e-02    +7.69231e+00%
 5       +16.25000000    +15.00000000    +15.62500000    -1.83407e-02    +8.35087e-03    -5.06632e-03    +4.00000e+00%
 6       +15.62500000    +15.00000000    +15.31250000    -5.06632e-03    +8.35087e-03    +1.62426e-03    +2.04082e+00%
 7       +15.62500000    +15.31250000    +15.46875000    -5.06632e-03    +1.62426e-03    -1.72552e-03    +1.01010e+00%
 8       +15.46875000    +15.31250000    +15.39062500    -1.72552e-03    +1.62426e-03    -5.17536e-05    +5.07614e-01%
 9       +15.39062500    +15.31250000    +15.35156250    -5.17536e-05    +1.62426e-03    +7.85970e-04    +2.54453e-01%

 Final Solution: f(x=+15.3515625000000000)=+7.859701e-04 with BI-SECTION method and 9 iterations
 >> O_c=exp(f_c(C_sol))

O_c =

     1.000786279059819e+01

>> o_sf=12;
>> f=@(c) f_c(c)-log(o_sf);
>> C_sol=solver(f,[0 40],'method','Bi-section','abs_tolerance',0.05);

 k       x_l             x_u             x_n             f_l             f_u             f_n             x_rel_error
----------------------------------------------------------------------------------------------------------------------------
 0       +40.00000000    +0.00000000     +20.00000000    -6.26623e-01    +1.97541e-01    -2.77465e-01    +1.00000e+02%
 1       +20.00000000    +0.00000000     +10.00000000    -2.77465e-01    +1.97541e-01    -6.11711e-02    +1.00000e+02%
 2       +10.00000000    +0.00000000     +5.00000000     -6.11711e-02    +1.97541e-01    +6.22704e-02    +1.00000e+02%
 3       +10.00000000    +5.00000000     +7.50000000     -6.11711e-02    +6.22704e-02    -8.59350e-04    +3.33333e+01%
 4       +7.50000000     +5.00000000     +6.25000000     -8.59350e-04    +6.22704e-02    +3.03440e-02    +2.00000e+01%
 5       +7.50000000     +6.25000000     +6.87500000     -8.59350e-04    +3.03440e-02    +1.46531e-02    +9.09091e+00%
 6       +7.50000000     +6.87500000     +7.18750000     -8.59350e-04    +1.46531e-02    +6.87469e-03    +4.34783e+00%
 7       +7.50000000     +7.18750000     +7.34375000     -8.59350e-04    +6.87469e-03    +3.00215e-03    +2.12766e+00%
 8       +7.50000000     +7.34375000     +7.42187500     -8.59350e-04    +3.00215e-03    +1.07002e-03    +1.05263e+00%
 9       +7.50000000     +7.42187500     +7.46093750     -8.59350e-04    +1.07002e-03    +1.04990e-04    +5.23560e-01%

 Final Solution: f(x= +7.4609375000000000)=+1.049900e-04 with BI-SECTION method and 9 iterations
 >> O_c=exp(f_c(C_sol))

O_c =

     1.200125994656214e+01
```

Textbook problem 6.1

The root of $f(x) = 2\sin\sqrt{x} - x$ is equivalent to the fixed point of $g(x) = 2\sin\sqrt{x}$. However for the later to converge, we have to have $\left|\frac{dg}{dx}\right| = \left|\frac{\cos\sqrt{x}}{\sqrt{x}}\right| \le 1$. Unfortunately $\left|\frac{dg}{dx}(x_0 = 0.5)\right| \cong 1.007$ and the fixed point with this g can be unstable. However, since $g(x_0 = 0.5) \cong 1.29$ it will escape from unstable region. Calculations are done in attached C2p29_PSET2_6p1.m file.

$$\varepsilon_a(n) \cong A\beta^n \quad \forall n > \bar{n} \quad \Rightarrow \quad \log \varepsilon_a \cong n \log \beta + \log A \quad \forall n > \bar{n}$$

The process is linearly convergent because the log of relative error with respect to the iteration number is linear according to the below graph.



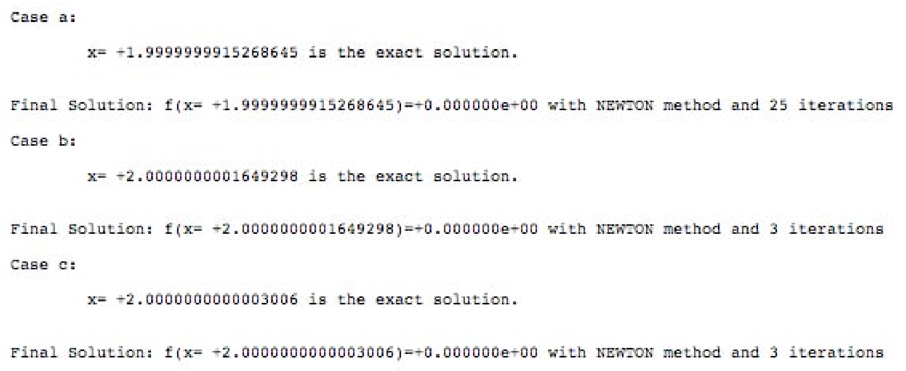
```
Final Solution: f(x= +1.9723804860817313)=+4.516657e-07 with Fixed Point method and 8 iterations
```

Textbook problem 6.11

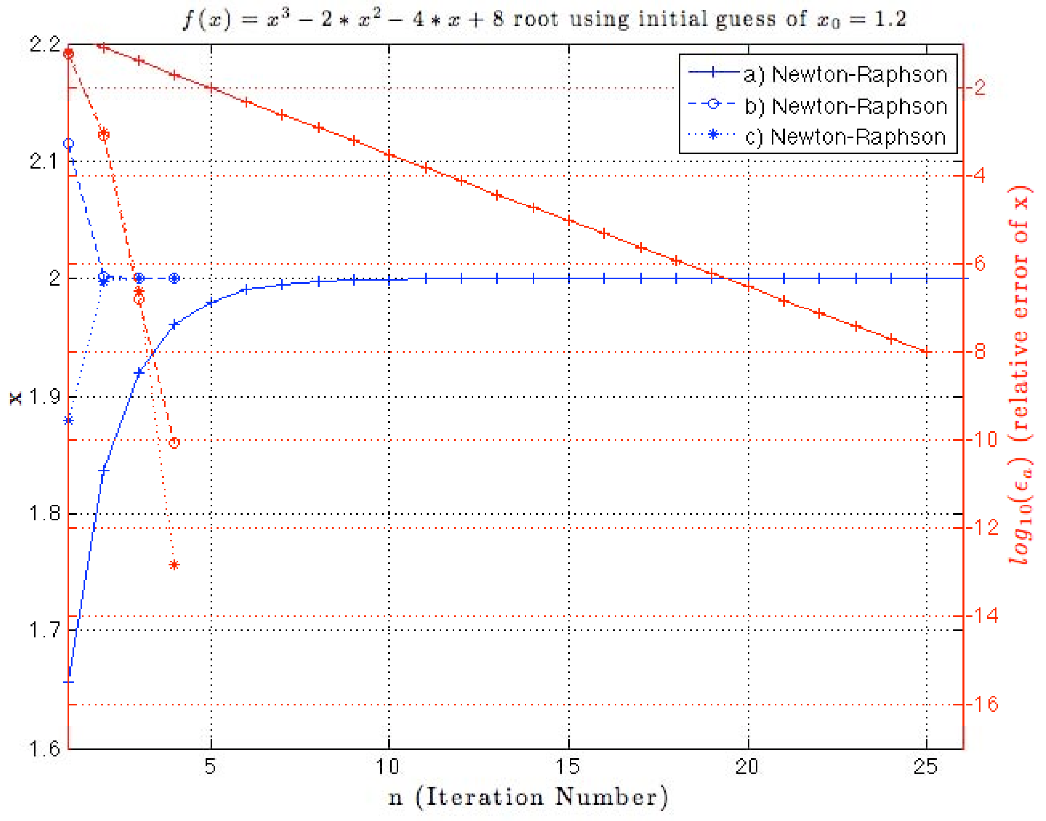
The attached file C2p29_PSET2_6p11.m solves this problem. Basically the same solver.m with Newton's method is used however we use these sets of function for this double root.

$$a) \quad f \qquad f'$$
$$b) \quad 2f \qquad f'$$
$$c) \quad ff' \quad f'^2 - ff''$$

A relative error of 10^{-16} is imposed. The case b and c converges within 3 iterations. However, case "a" requires about 25 iteration; it does not overshoot but is very slow. Case c has a slightly higher accuracy, but we should remember that it needs extra information about 2nd derivtive.

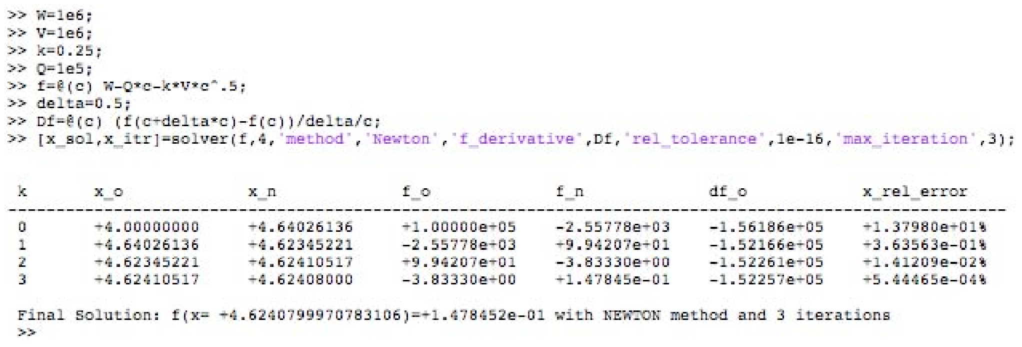


```
Case a:

        x= +1.9999999915268645 is the exact solution.

Final Solution: f(x= +1.9999999915268645)=+0.000000e+00 with NEWTON method and 25 iterations
Case b:

        x= +2.0000000001649298 is the exact solution.

Final Solution: f(x= +2.0000000001649298)=+0.000000e+00 with NEWTON method and 3 iterations
Case c:

        x= +2.0000000000003006 is the exact solution.

Final Solution: f(x= +2.0000000000003006)=+0.000000e+00 with NEWTON method and 3 iterations
```



Textbook problem 6.15

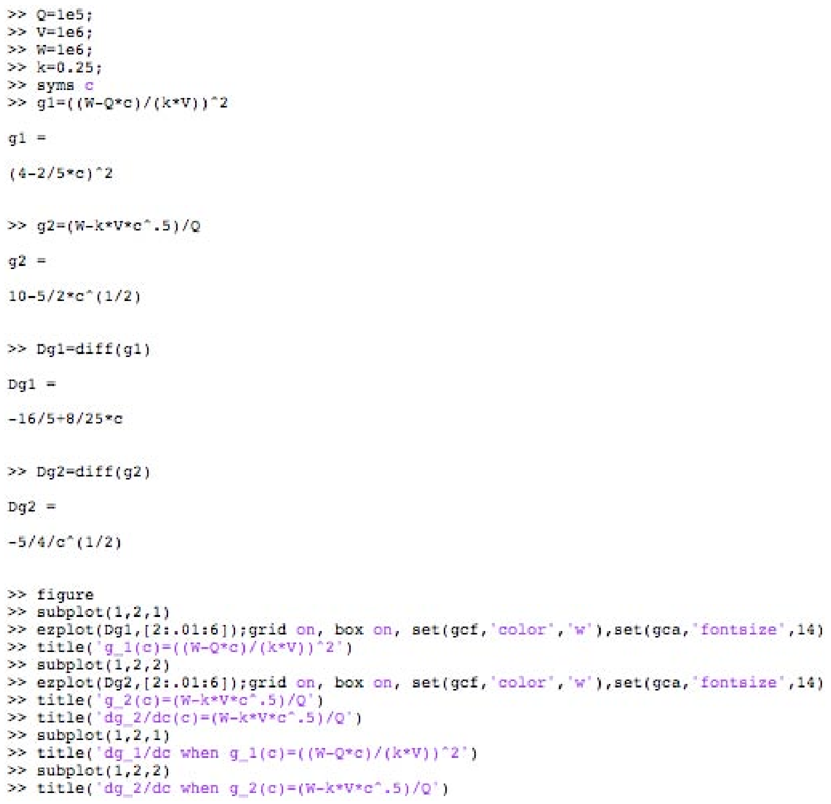
At the steady state $\dfrac{dc}{dt} = 0$ and we have $f(c) = W - Qc - KV\sqrt{c} = 0$. Again we apparently use our earlier solver.m module for the modified secant. We call it by Newton's method, but we replace the derivative with $\dfrac{f(c+\delta) - f(c)}{\delta}$.



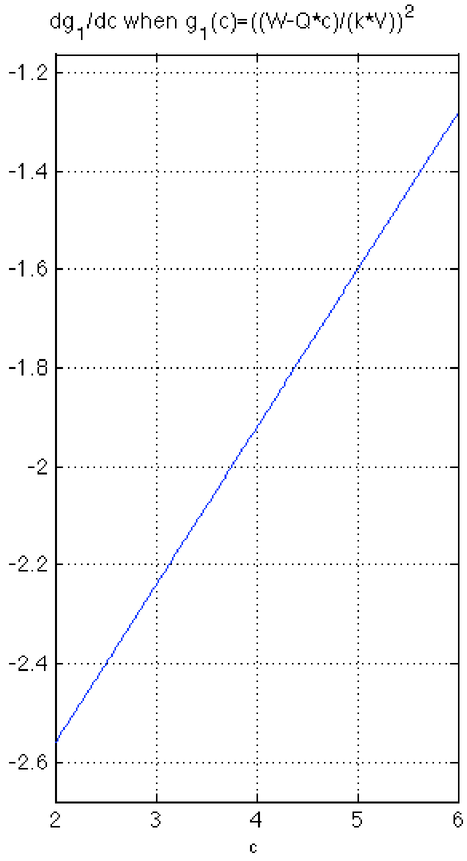
```
>> W=1e6;
>> V=1e6;
>> k=0.25;
>> Q=1e5;
>> f=@(c) W-Q*c-k*V*c^.5;
>> delta=0.5;
>> Df=@(c) (f(c+delta*c)-f(c))/delta/c;
>> [x_sol,x_itr]=solver(f,4,'method','Newton','f_derivative',Df,'rel_tolerance',1e-16,'max_iteration',3);

 k       x_o             x_n             f_o             f_n             df_o            x_rel_error
-----------------------------------------------------------------------------------------------------------
 0       +4.00000000     +4.64026136     +1.00000e+05    -2.55778e+03    -1.56186e+05    +1.37980e+01%
 1       +4.64026136     +4.62345221     -2.55778e+03    +9.94207e+01    -1.52166e+05    +3.63563e-01%
 2       +4.62345221     +4.62410517     +9.94207e+01    -3.83330e+00    -1.52261e+05    +1.41209e-02%
 3       +4.62410517     +4.62408000     -3.83330e+00    +1.47845e-01    -1.52257e+05    +5.44465e-04%

 Final Solution: f(x= +4.6240799970783106)=+1.478452e-01 with NEWTON method and 3 iterations
 >>
```

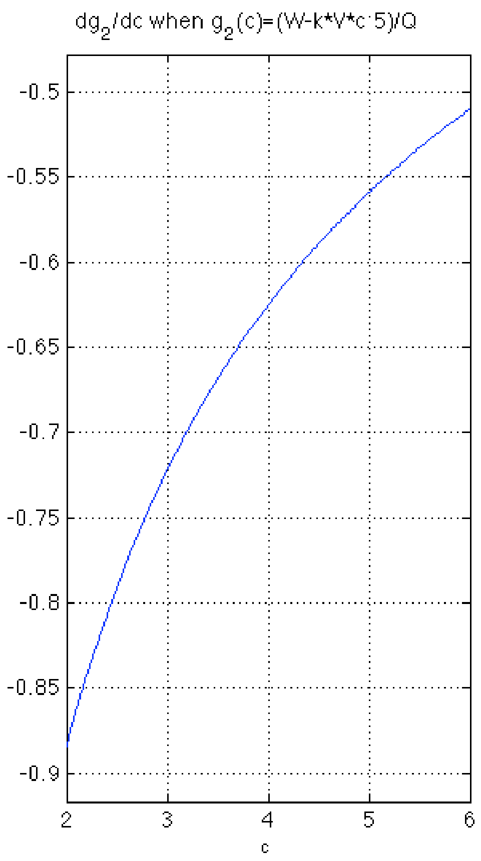
Textbook problem 6.16

We have to see which one satisfies the $\left|\frac{dg(c)}{dc}\right| \leq 1$ condition. From the graph next page we can conclude that the 2nd one has a guaranteed convergence for the given interval to $c_r \cong 4.62$ root.



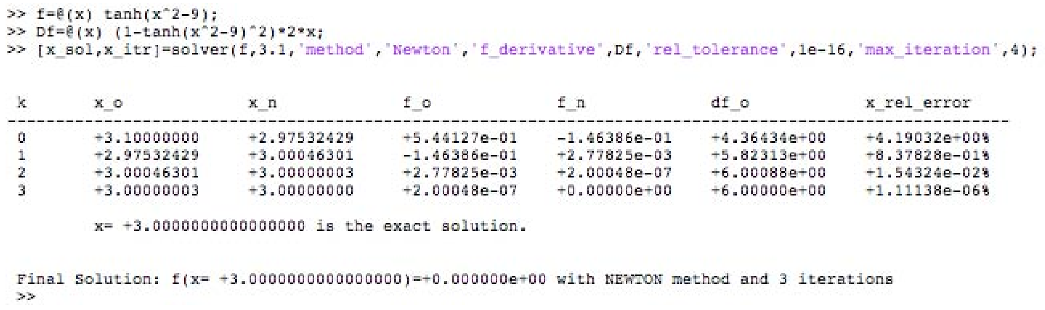
```
>> Q=1e5;
>> V=1e6;
>> W=1e6;
>> k=0.25;
>> syms c
>> g1=((W-Q*c)/(k*V))^2

g1 =

(4-2/5*c)^2


>> g2=(W-k*V*c^.5)/Q

g2 =

10-5/2*c^(1/2)


>> Dg1=diff(g1)

Dg1 =

-16/5+8/25*c


>> Dg2=diff(g2)

Dg2 =

-5/4/c^(1/2)


>> figure
>> subplot(1,2,1)
>> ezplot(Dg1,[2:.01:6]);grid on, box on, set(gcf,'color','w'),set(gca,'fontsize',14)
>> title('g_1(c)=((W-Q*c)/(k*V))^2')
>> subplot(1,2,2)
>> ezplot(Dg2,[2:.01:6]);grid on, box on, set(gcf,'color','w'),set(gca,'fontsize',14)
>> title('g_2(c)=(W-k*V*c^.5)/Q')
>> title('dg_2/dc(c)=(W-k*V*c^.5)/Q')
>> subplot(1,2,1)
>> title('dg_1/dc when g_1(c)=((W-Q*c)/(k*V))^2')
>> subplot(1,2,2)
>> title('dg_2/dc when g_2(c)=(W-k*V*c^.5)/Q')
```



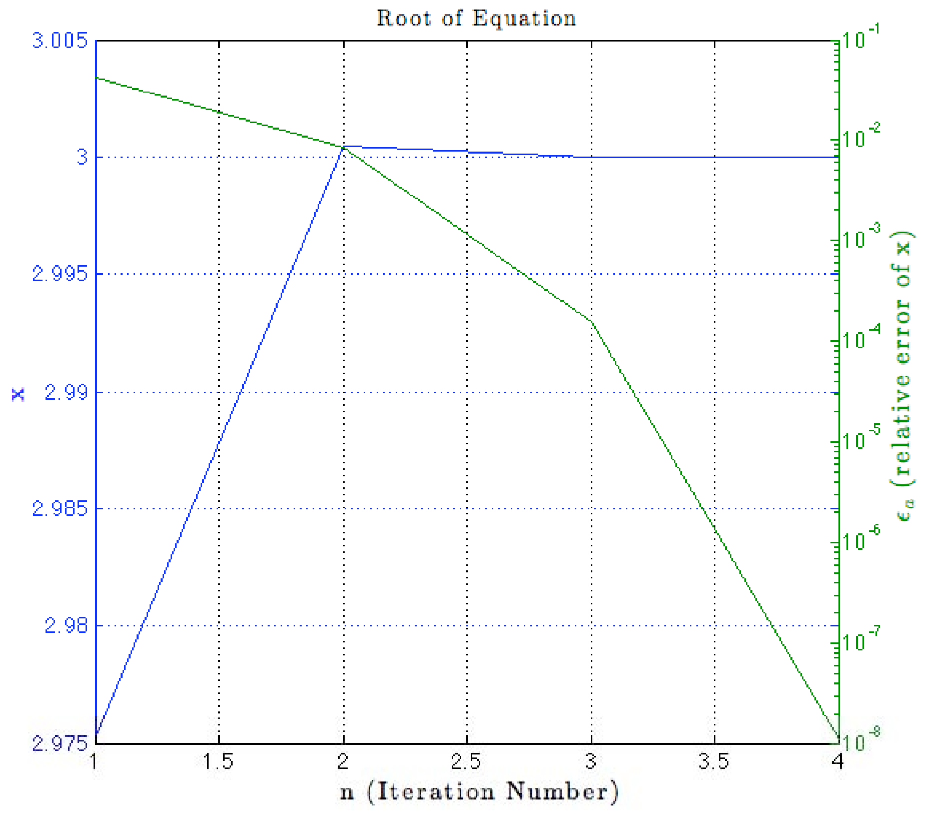


Textbook problem 6.23

Within 4 iterations it will find the exact value of root. This means that the function does not have considerable curvature around this root.



```
>> f=@(x) tanh(x^2-9);
>> Df=@(x) (1-tanh(x^2-9)^2)*2*x;
>> [x_sol,x_itr]=solver(f,3.1,'method','Newton','f_derivative',Df,'rel_tolerance',1e-16,'max_iteration',4);

 k       x_o              x_n              f_o              f_n              df_o             x_rel_error
-------------------------------------------------------------------------------------------------------------
 0       +3.10000000      +2.97532429      +5.44127e-01     -1.46386e-01     +4.36434e+00     +4.19032e+00%
 1       +2.97532429      +3.00046301      -1.46386e-01     +2.77825e-03     +5.82313e+00     +8.37828e-01%
 2       +3.00046301      +3.00000003      +2.77825e-03     +2.00048e-07     +6.00088e+00     +1.54324e-02%
 3       +3.00000003      +3.00000000      +2.00048e-07     +0.00000e+00     +6.00000e+00     +1.11138e-06%

         x= +3.0000000000000000 is the exact solution.


 Final Solution: f(x= +3.0000000000000000)=+0.000000e+00 with NEWTON method and 3 iterations
 >>
```



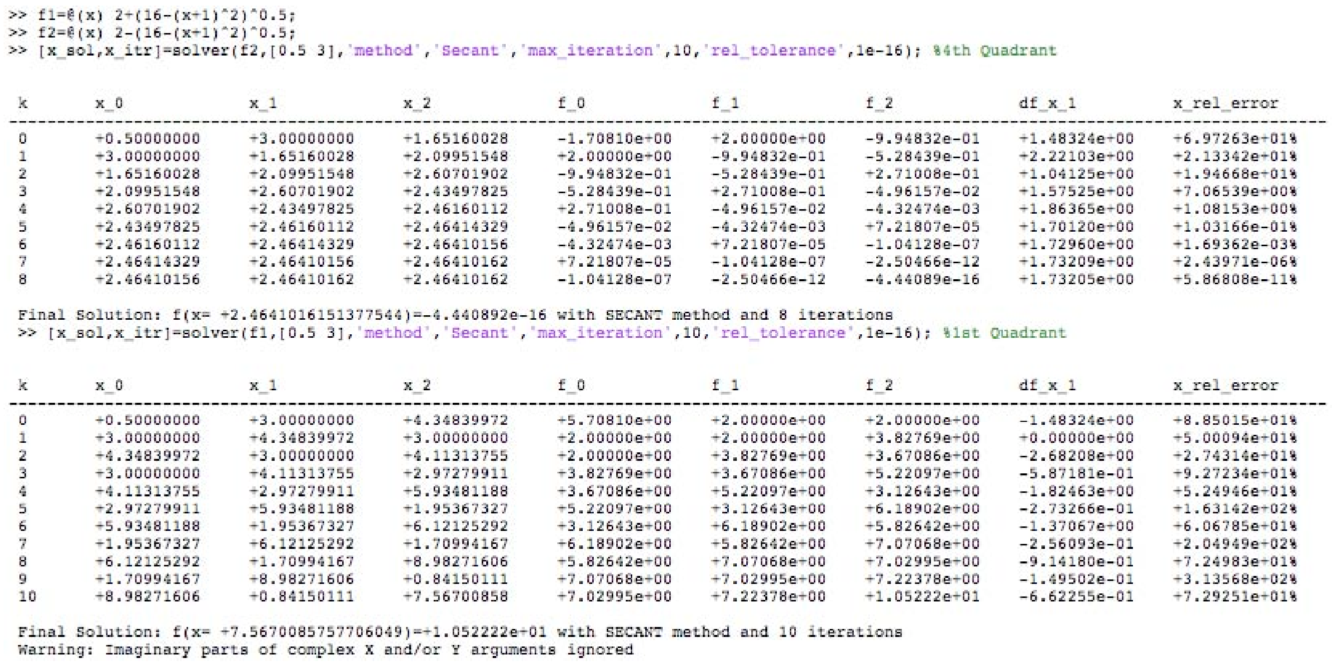
Textbook problem 6.25 (Extra Credit)

The circle radius is 4 and we have $|x+1| \le 4 \Rightarrow -5 \le x \le 3$. So note that:

$$y_1(x) = +\sqrt{16-(x+1)^2}+2\ , \quad \textit{First} \quad \textit{Quadrant}$$

$$y_2(x) = -\sqrt{16-(x+1)^2}+2\ , \quad \textit{Fourth Quadrant}$$

The results are shown on the next page. It converges for the 4th quadrant but not the 1st one. This is because in the 1st quadrant $2 \le y \le 2+4$ and we do not have a root (so it diverges toward complex numbers), while for the 4th quadrant $-4+2 \le y \le 2$ a root exists.



```
>> f1=@(x) 2+(16-(x+1)^2)^0.5;
>> f2=@(x) 2-(16-(x+1)^2)^0.5;
>> [x_sol,x_itr]=solver(f2,[0.5 3],'method','Secant','max_iteration',10,'rel_tolerance',1e-16); %4th Quadrant

 k       x_0             x_1             x_2             f_0             f_1             f_2             df_x_1          x_rel_error
----------------------------------------------------------------------------------------------------------------------------------------
 0       +0.50000000     +3.00000000     +1.65160028     -1.70810e+00    +2.00000e+00    -9.94832e-01    +1.48324e+00    +6.97263e+01%
 1       +3.00000000     +1.65160028     +2.09951548     +2.00000e+00    -9.94832e-01    -5.28439e-01    +2.22103e+00    +2.13342e+01%
 2       +1.65160028     +2.09951548     +2.60701902     -9.94832e-01    -5.28439e-01    +2.71008e-01    +1.04125e+00    +1.94668e+01%
 3       +2.09951548     +2.60701902     +2.43497825     -5.28439e-01    +2.71008e-01    -4.96157e-02    +1.57525e+00    +7.06539e+00%
 4       +2.60701902     +2.43497825     +2.46160112     +2.71008e-01    -4.96157e-02    -4.32474e-03    +1.86365e+00    +1.08153e+00%
 5       +2.43497825     +2.46160112     +2.46414329     -4.96157e-02    -4.32474e-03    +7.21807e-05    +1.70120e+00    +1.03166e-01%
 6       +2.46160112     +2.46414329     +2.46410156     -4.32474e-03    +7.21807e-05    -1.04128e-07    +1.72960e+00    +1.69362e-03%
 7       +2.46414329     +2.46410156     +2.46410162     +7.21807e-05    -1.04128e-07    -2.50466e-12    +1.73209e+00    +2.43971e-06%
 8       +2.46410156     +2.46410162     +2.46410162     -1.04128e-07    -2.50466e-12    -4.44089e-16    +1.73205e+00    +5.86808e-11%

 Final Solution: f(x= +2.4641016151377544)=-4.440892e-16 with SECANT method and 8 iterations
 >> [x_sol,x_itr]=solver(f1,[0.5 3],'method','Secant','max_iteration',10,'rel_tolerance',1e-16); %1st Quadrant

 k       x_0             x_1             x_2             f_0             f_1             f_2             df_x_1          x_rel_error
----------------------------------------------------------------------------------------------------------------------------------------
 0       +0.50000000     +3.00000000     +4.34839972     +5.70810e+00    +2.00000e+00    +2.00000e+00    -1.48324e+00    +8.85015e+01%
 1       +3.00000000     +4.34839972     +3.00000000     +2.00000e+00    +2.00000e+00    +3.82769e+00    +0.00000e+00    +5.00094e+01%
 2       +4.34839972     +3.00000000     +4.11313755     +2.00000e+00    +3.82769e+00    +3.67086e+00    -2.68208e+00    +2.74314e+01%
 3       +3.00000000     +4.11313755     +2.97279911     +3.82769e+00    +3.67086e+00    +5.22097e+00    -5.87181e-01    +9.27234e+01%
 4       +4.11313755     +2.97279911     +5.93481188     +3.67086e+00    +5.22097e+00    +3.12643e+00    -1.82463e+00    +5.24946e+01%
 5       +2.97279911     +5.93481188     +1.95367327     +5.22097e+00    +3.12643e+00    +6.18902e+00    -2.73266e-01    +1.63142e+02%
 6       +5.93481188     +1.95367327     +6.12125292     +3.12643e+00    +6.18902e+00    +5.82642e+00    -1.37067e+00    +6.06785e+01%
 7       +1.95367327     +6.12125292     +1.70994167     +6.18902e+00    +5.82642e+00    +7.07068e+00    -2.56093e-01    +2.04949e+02%
 8       +6.12125292     +1.70994167     +8.98271606     +5.82642e+00    +7.07068e+00    +7.02995e+00    -9.14180e-01    +7.24983e+01%
 9       +1.70994167     +8.98271606     +0.84150111     +7.07068e+00    +7.02995e+00    +7.22378e+00    -1.49502e-01    +3.13568e+02%
 10      +8.98271606     +0.84150111     +7.56700858     +7.02995e+00    +7.22378e+00    +1.05222e+01    -6.62255e-01    +7.29251e+01%

 Final Solution: f(x= +7.5670085757706049)=+1.052222e+01 with SECANT method and 10 iterations
 Warning: Imaginary parts of complex X and/or Y arguments ignored
```

Textbook problem 10.6

Similar to "b" we adopt 3 significant digits.

$$A=\begin{bmatrix}10 & 2 & -1\\ -3 & -6 & 2\\ 1 & 1 & 5\end{bmatrix},\ b=\begin{bmatrix}27\\ -61.5\\ 21.5\end{bmatrix}$$

$$m_{21}=\frac{-3}{10}=-0.3 \qquad A\rightarrow\begin{bmatrix}10 & 2 & -1\\ 0 & -5.4 & 1.7\\ 0 & 0.8 & 5.1\end{bmatrix}$$

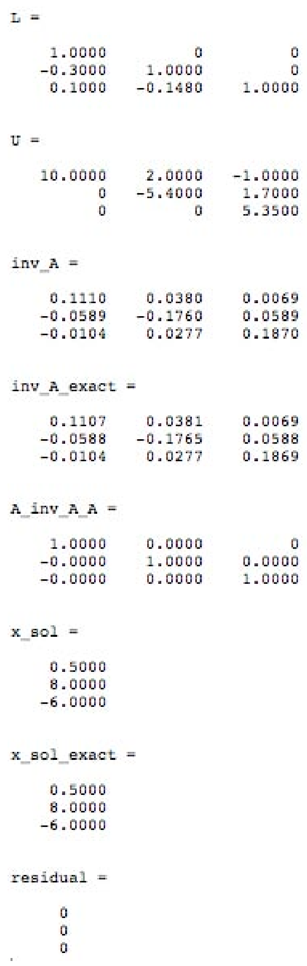
$$m_{31}=\frac{1}{10}=+0.1$$

$$m_{21}=-0.3 \qquad A\rightarrow\begin{bmatrix}10 & 2 & -1\\ 0 & -5.4 & 1.7\\ 0 & 0 & 5.35\end{bmatrix}$$

$$m_{31}=+0.1 \quad m_{32}=\frac{0.8}{-5.4}=-0.148$$

$$LU=A,\quad L=\begin{bmatrix}1 & 0 & 0\\ -0.3 & 1 & 0\\ +0.1 & -0.148 & 1\end{bmatrix},\ U=\begin{bmatrix}10 & 2 & -1\\ 0 & -5.4 & 1.7\\ 0 & 0 & 5.35\end{bmatrix}$$

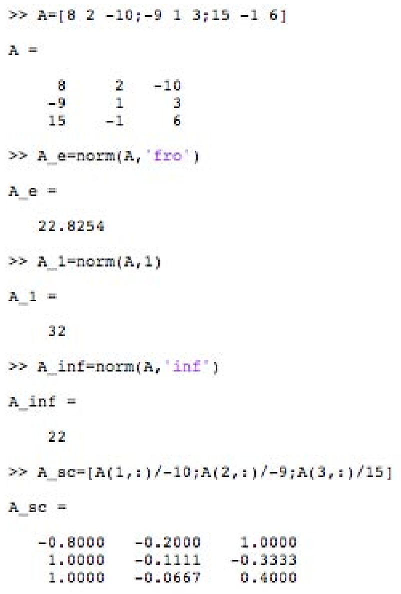
The forward and backward substitution with corresponding chopping is done in the attached C2p29_PSET2_10p6.m file. We use the LU factorization to compute A^{-1} by 3 times of LU application. Finally the solution is also computed with the same technique. Here is the program output. Note that we used only 3 digits and the final errors are on the 4th and 5th digits.



```
L =

    1.0000         0         0
   -0.3000    1.0000         0
    0.1000   -0.1480    1.0000


U =

   10.0000    2.0000   -1.0000
         0   -5.4000    1.7000
         0         0    5.3500


inv_A =

    0.1110    0.0380    0.0069
   -0.0589   -0.1760    0.0589
   -0.0104    0.0277    0.1870


inv_A_exact =

    0.1107    0.0381    0.0069
   -0.0588   -0.1765    0.0588
   -0.0104    0.0277    0.1869


A_inv_A_A =

    1.0000    0.0000         0
   -0.0000    1.0000    0.0000
   -0.0000    0.0000    1.0000


x_sol =

    0.5000
    8.0000
   -6.0000


x_sol_exact =

    0.5000
    8.0000
   -6.0000


residual =

     0
     0
     0
```

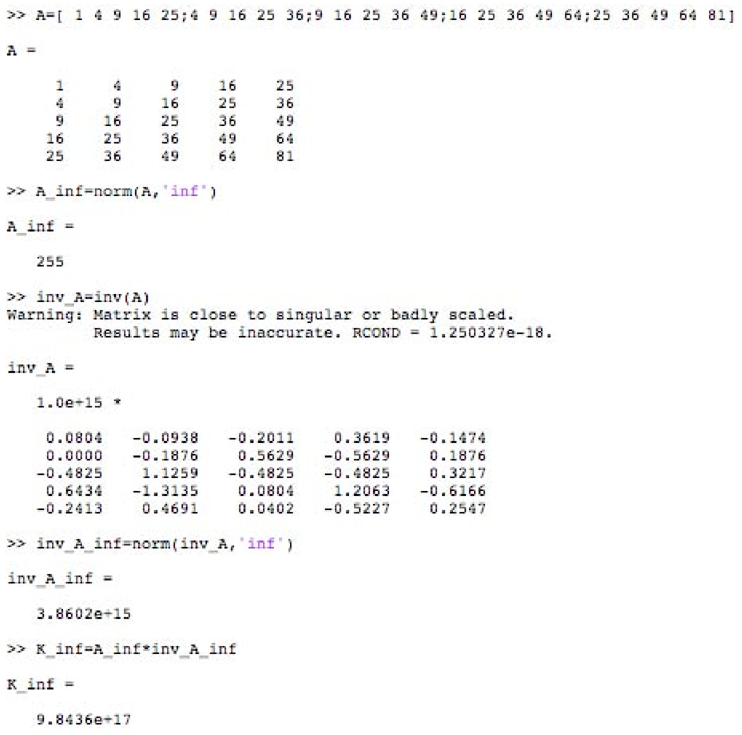
Textbook problem 10.9

Note that for the scaling we divide each row by the element whose absolute value is the maximum of that row.

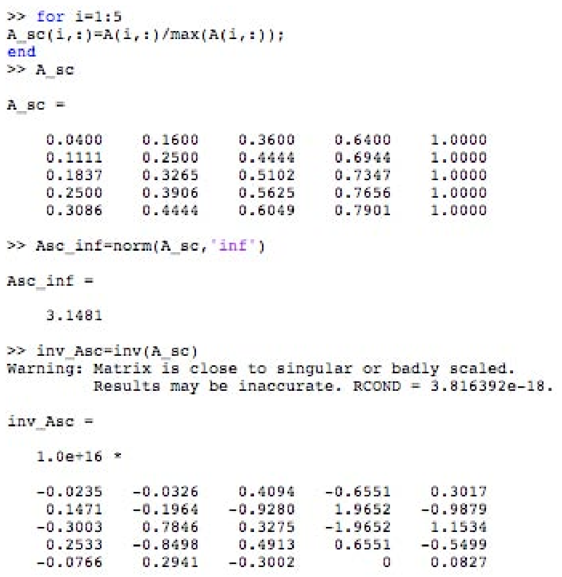


```
>> A=[8 2 -10;-9 1 3;15 -1 6]

A =

     8     2   -10
    -9     1     3
    15    -1     6

>> A_e=norm(A,'fro')

A_e =

   22.8254

>> A_1=norm(A,1)

A_1 =

    32

>> A_inf=norm(A,'inf')

A_inf =

    22

>> A_sc=[A(1,:)/-10;A(2,:)/-9;A(3,:)/15]

A_sc =

   -0.8000   -0.2000    1.0000
    1.0000   -0.1111   -0.3333
    1.0000   -0.0667    0.4000
```

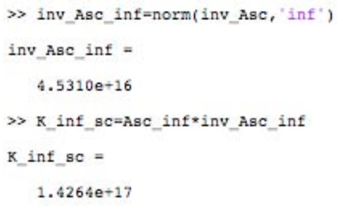
Textbook problem 10.12



```
>> A=[ 1 4 9 16 25;4 9 16 25 36;9 16 25 36 49;16 25 36 49 64;25 36 49 64 81]

A =

     1     4     9    16    25
     4     9    16    25    36
     9    16    25    36    49
    16    25    36    49    64
    25    36    49    64    81

>> A_inf=norm(A,'inf')

A_inf =

   255

>> inv_A=inv(A)
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.250327e-18.

inv_A =

   1.0e+15 *

    0.0804   -0.0938   -0.2011    0.3619   -0.1474
    0.0000   -0.1876    0.5629   -0.5629    0.1876
   -0.4825    1.1259   -0.4825   -0.4825    0.3217
    0.6434   -1.3135    0.0804    1.2063   -0.6166
   -0.2413    0.4691    0.0402   -0.5227    0.2547

>> inv_A_inf=norm(inv_A,'inf')

inv_A_inf =

   3.8602e+15

>> K_inf=A_inf*inv_A_inf

K_inf =

   9.8436e+17
```

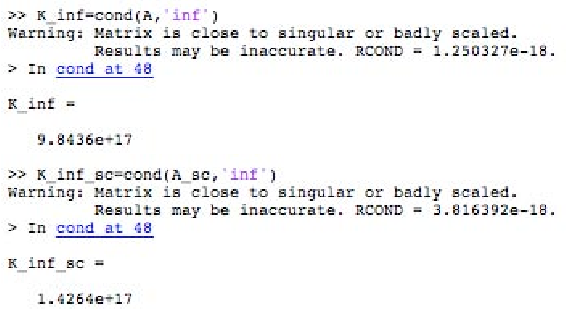


```
>> for i=1:5
A_sc(i,:)=A(i,:)/max(A(i,:));
end
>> A_sc

A_sc =

    0.0400    0.1600    0.3600    0.6400    1.0000
    0.1111    0.2500    0.4444    0.6944    1.0000
    0.1837    0.3265    0.5102    0.7347    1.0000
    0.2500    0.3906    0.5625    0.7656    1.0000
    0.3086    0.4444    0.6049    0.7901    1.0000

>> Asc_inf=norm(A_sc,'inf')

Asc_inf =

    3.1481

>> inv_Asc=inv(A_sc)
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 3.816392e-18.

inv_Asc =

   1.0e+16 *

   -0.0235   -0.0326    0.4094   -0.6551    0.3017
    0.1471   -0.1964   -0.9280    1.9652   -0.9879
   -0.3003    0.7846    0.3275   -1.9652    1.1534
    0.2533   -0.8498    0.4913    0.6551   -0.5499
   -0.0766    0.2941   -0.3002         0    0.0827
```



```
>> inv_Asc_inf=norm(inv_Asc,'inf')

inv_Asc_inf =

   4.5310e+16

>> K_inf_sc=Asc_inf*inv_Asc_inf

K_inf_sc =

   1.4264e+17
```

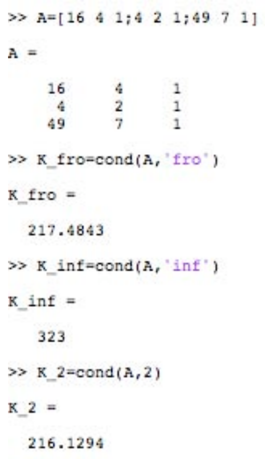
Alternatively we can use “cond” command to directly compute the matrix condition number.



```
>> K_inf=cond(A,'inf')
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.250327e-18.
> In cond at 48

K_inf =

   9.8436e+17

>> K_inf_sc=cond(A_sc,'inf')
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 3.816392e-18.
> In cond at 48

K_inf_sc =

   1.4264e+17
```

Since the condition number is about $9.8 \times 10^{17} \cong 10^{18}$ we will lose about 18 significant digits. As a result the significant digits of ordinary double and single in MATLAB can be all lost (they have about 15 and 7 significant digit). Unfortunately even the scaling does not help so much. Indeed scaling can only improve about 0.84 digits in base 10 ($\log_{10}(\frac{9.84}{1.42}) \cong 0.84$).

Textbook problem 10.14



```
>> A=[16 4 1;4 2 1;49 7 1]

A =

    16     4     1
     4     2     1
    49     7     1

>> K_fro=cond(A,'fro')

K_fro =

  217.4843

>> K_inf=cond(A,'inf')

K_inf =

   323

>> K_2=cond(A,2)

K_2 =

  216.1294
```

Problem 2.4 (15 Points): Computation cost in MATLAB

Review MATLAB help about:

- profile
- tic
- toc

Here we want to investigate the computational cost of basic operations $\{+,-,\times,\div\}$ and some functions $\{x,|x|,\sqrt{x},x^2,x^3,\sin x,\cos x,\tan x,e^x,e^{-x},\ln x\}$ in the MATLAB program. To that end we generate a huge vector/matrix of random numbers and repeat the operation on them as long as to get consistent result. Please specify your computer speed, memory and MATLAB version. Try to provide as much consistency as possible. Also clear and free your memory as much as possible. In each case examine both single and double data type and also report the cost time normalized by your CPU clock time (for example for a 2 GHz computer multiply time by 2×10^9 $1/\sec$).

a) Report assignment cost (or function y=x). Compare cost of a scalar assignment with cost of matrix assignment (normalized by the number of elements).

b) Compare cost of basic operations for scalar. Then compare $+,-$ for the matrixes and also element wise $\times,\div$. Discuss.

c) Repeat part b for specified functions.

Solution:

These results are produced by:

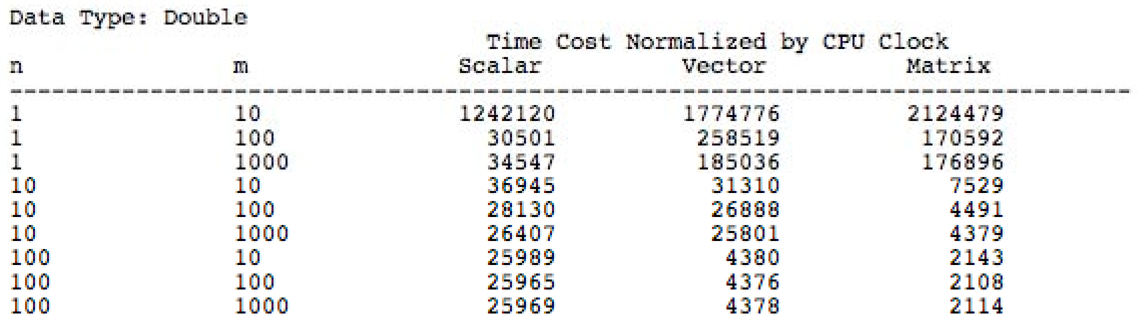
- MacBook 1.1
- Intel Core Duo 2.0GHz CPU (1 processor, 2 cores, 2Mb L2 cache)
- About 100% CPU commitment for MATLAB during runs
- 1 Gb of 667 MHz DDR2 RAM
- Up to 13.5 Gb of Virtual Memory
- MATALB version R2006b

The results might vary across different systems but the whole idea of this homework is to get an insight about computational cost. Here an experimental approach is used, while a better technique might rely on investigating the computer architecture combined with software implementations. Some people might use the flops for normalizing the time cost, but indeed there is a difference between clock and flops and here we use clock.

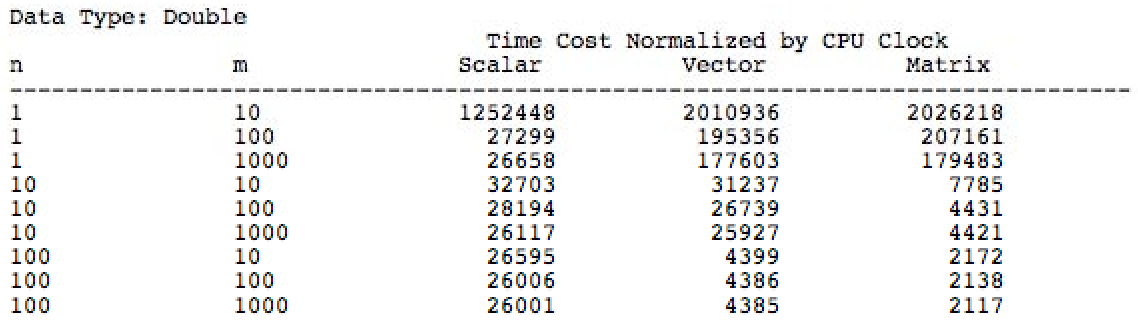
The costs are evaluated for normalized positive numbers (between 0 and 1) and they are generated by “rand” command. The corresponding file is named C2p29_PSET1_4.m and is included in the script package. The costs are normalized by the number of element and the first question is that how much are they sensitive to matrix size and computer resources at that instant. So we consider the below computational block:

```
a=rand(n,n);
b=rand(n,n);

c=a; %Assignment Cost
c=a+b;
c=a-b;
c=a.*b;
c=a./b;

c=abs(a);
c=a.^2;
c=a.^3;
c=a.^.5;
c=sqrt(a);

c=sin(a);
c=cos(a);
c=tan(a);

c=exp(a);
c=exp(-a);
```

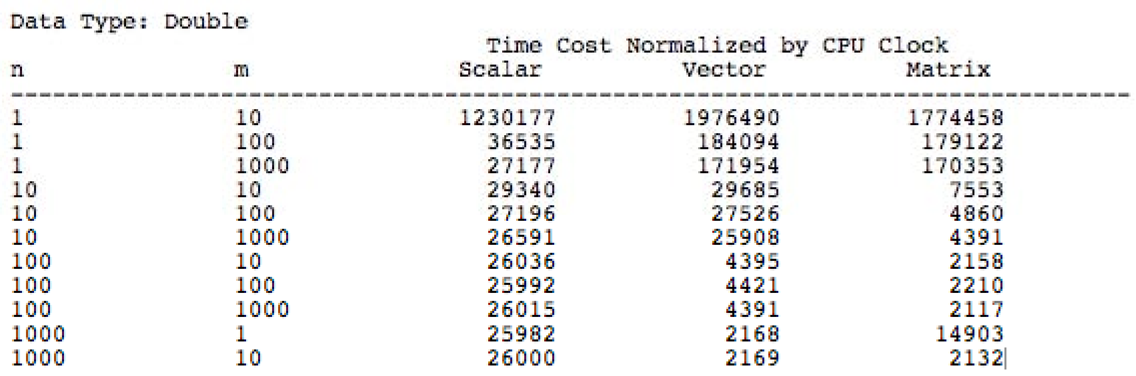
The above calculation is done for a matrix with size $a_{n\times n}$ for "m" times. Accordingly it is done for a vector with size $a_{n\times 1}$ for $m\times n$ times, as well as a scalar a for $m\times n^2$ times. In each case, "a,b and c" are allocated before going inside "times" loop; consequently our assignment cost is indeed just a "copy" cost and does not include memory release\allocation time. The time is computed with "tic, toc" command and time profiler is not used at this point because it is not as accurate as to "tic, toc" (due to time profiling overhead). The program is run for different pairs of "m" and "n" and a few sample run is shown below:



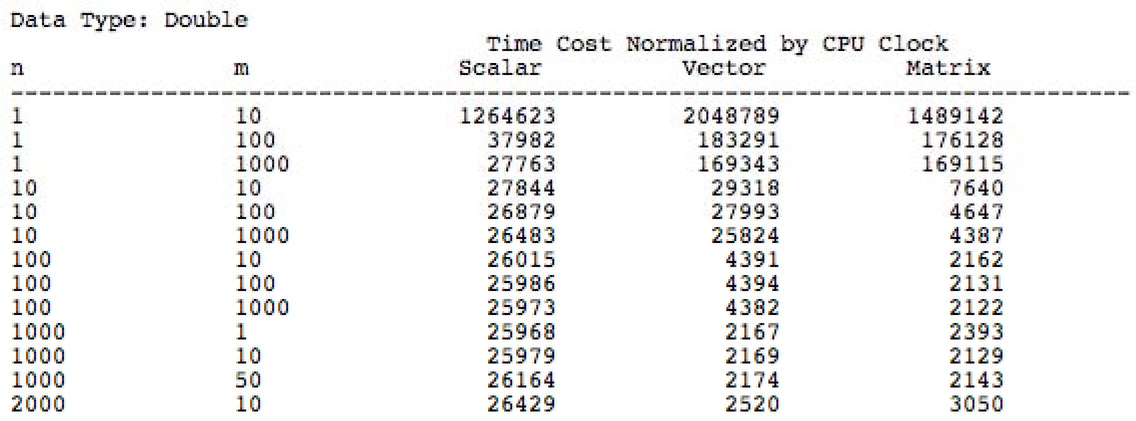
```
Data Type: Double
                                   Time Cost Normalized by CPU Clock
n               m                Scalar          Vector          Matrix
---------------------------------------------------------------------------------
1               10               1242120         1774776         2124479
1               100                30501          258519          170592
1               1000               34547          185036          176896
10              10                 36945           31310            7529
10              100                28130           26888            4491
10              1000               26407           25801            4379
100             10                 25989            4380            2143
100             100                25965            4376            2108
100             1000               25969            4378            2114
```



```
Data Type: Double
                                   Time Cost Normalized by CPU Clock
n               m                Scalar          Vector          Matrix
---------------------------------------------------------------------------------
1               10               1252448         2010936         2026218
1               100                27299          195356          207161
1               1000               26658          177603          179483
10              10                 32703           31237            7785
10              100                28194           26739            4431
10              1000               26117           25927            4421
100             10                 26595            4399            2172
100             100                26006            4386            2138
100             1000               26001            4385            2117
```



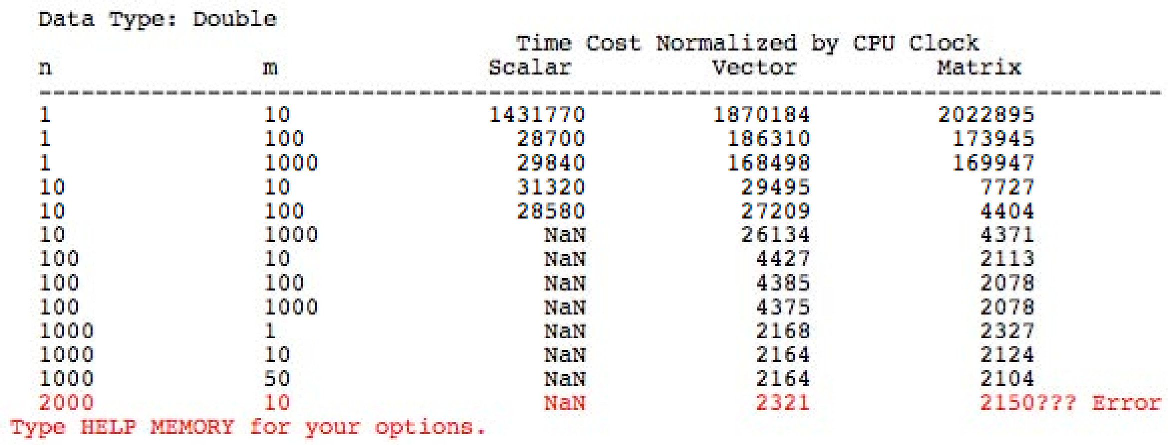
```
Data Type: Double
                                    Time Cost Normalized by CPU Clock
n               m                 Scalar          Vector          Matrix
--------------------------------------------------------------------------------
1               10                1230177         1976490         1774458
1               100                 36535          184094          179122
1               1000                27177          171954          170353
10              10                  29340           29685            7553
10              100                 27196           27526            4860
10              1000                26591           25908            4391
100             10                  26036            4395            2158
100             100                 25992            4421            2210
100             1000                26015            4391            2117
1000            1                   25982            2168           14903
1000            10                  26000            2169            2132
```



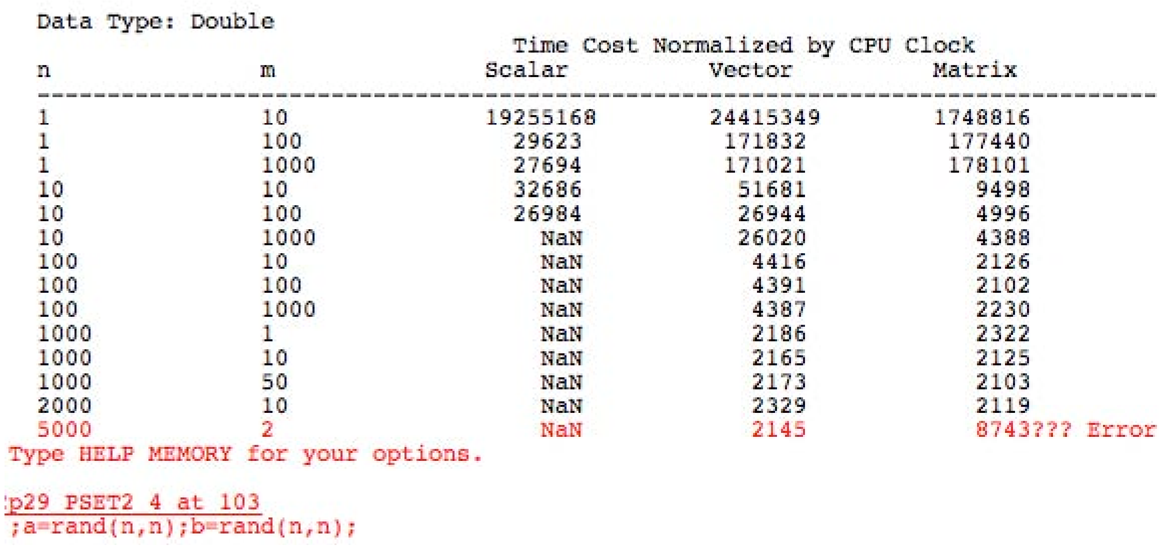
```
Data Type: Double
                                    Time Cost Normalized by CPU Clock
n               m                 Scalar          Vector          Matrix
--------------------------------------------------------------------------------
1               10                1264623         2048789         1489142
1               100                 37982          183291          176128
1               1000                27763          169343          169115
10              10                  27844           29318            7640
10              100                 26879           27993            4647
10              1000                26483           25824            4387
100             10                  26015            4391            2162
100             100                 25986            4394            2131
100             1000                25973            4382            2122
1000            1                   25968            2167            2393
1000            10                  25979            2169            2129
1000            50                  26164            2174            2143
2000            10                  26429            2520            3050
```

Mathematically a scalar and a vector or matrix with n=1 are equivalent. This is also true in MATAB, but their computation time is different. As seen above for n=1 a scalar is about 180000/30000=6 times faster.

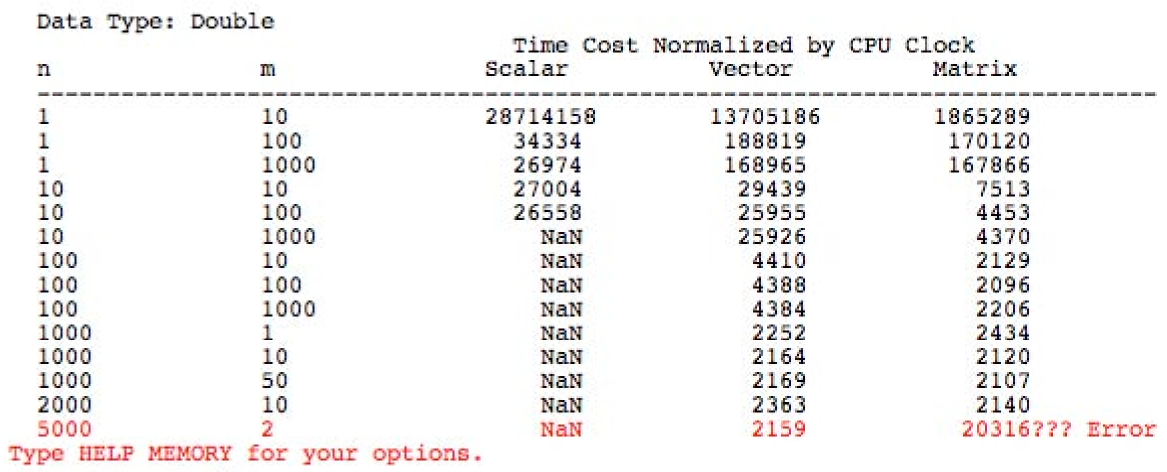
When “n” becomes very big (here like 5000, which require 25*8~200 Mb of memory), MATLAB cannot allocate memory for it and an error is produced. Also we can notice that when $m \times n^2 > 10^5$ results are rather stationary for scalars up to 4th digit. So from here forward we will not compute scalar cost for very huge pairs of “m, n”. On the other hand for small m, numbers are not consistent and for loop cost might also be inferring.



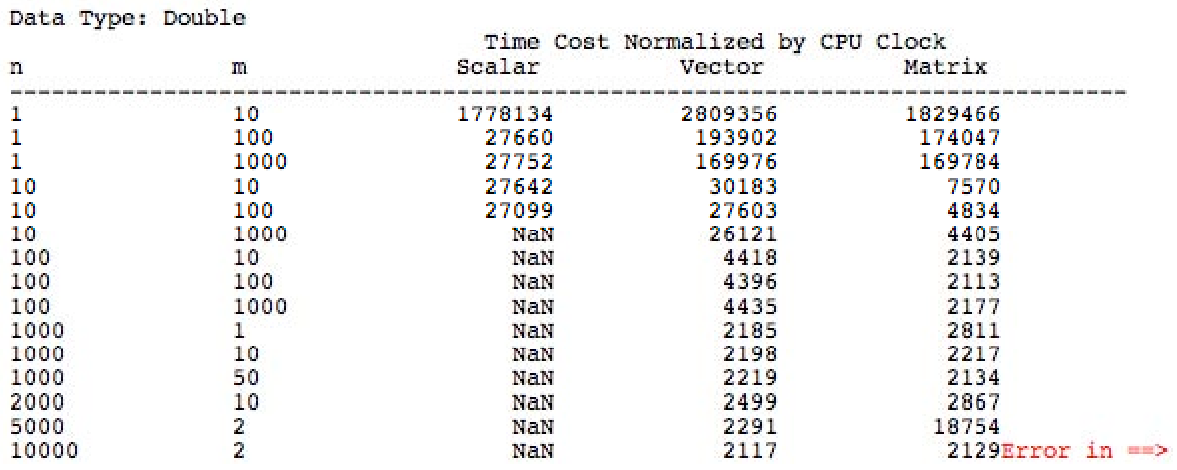
```
  Data Type: Double
                                    Time Cost Normalized by CPU Clock
  n               m                 Scalar          Vector          Matrix
  --------------------------------------------------------------------------------
  1               10                1431770         1870184         2022895
  1               100                 28700          186310          173945
  1               1000                29840          168498          169947
  10              10                  31320           29495            7727
  10              100                 28580           27209            4404
  10              1000                  NaN           26134            4371
  100             10                    NaN            4427            2113
  100             100                   NaN            4385            2078
  100             1000                  NaN            4375            2078
  1000            1                     NaN            2168            2327
  1000            10                    NaN            2164            2124
  1000            50                    NaN            2164            2104
  2000            10                    NaN            2321            2150??? Error
Type HELP MEMORY for your options.
```



```
Data Type: Double
                                   Time Cost Normalized by CPU Clock
n               m                Scalar          Vector          Matrix
-----------------------------------------------------------------------------------
1               10               19255168        24415349        1748816
1               100                 29623          171832         177440
1               1000                27694          171021         178101
10              10                  32686           51681           9498
10              100                 26984           26944           4996
10              1000                  NaN           26020           4388
100             10                    NaN            4416           2126
100             100                   NaN            4391           2102
100             1000                  NaN            4387           2230
1000            1                     NaN            2186           2322
1000            10                    NaN            2165           2125
1000            50                    NaN            2173           2103
2000            10                    NaN            2329           2119
5000            2                     NaN            2145           8743??? Error
Type HELP MEMORY for your options.

:p29 PSET2 4 at 103
;a=rand(n,n);b=rand(n,n);
```



```
Data Type: Double
                                   Time Cost Normalized by CPU Clock
n               m                Scalar          Vector          Matrix
-----------------------------------------------------------------------------------
1               10               28714158        13705186        1865289
1               100                 34334          188819         170120
1               1000                26974          168965         167866
10              10                  27004           29439           7513
10              100                 26558           25955           4453
10              1000                  NaN           25926           4370
100             10                    NaN            4410           2129
100             100                   NaN            4388           2096
100             1000                  NaN            4384           2206
1000            1                     NaN            2252           2434
1000            10                    NaN            2164           2120
1000            50                    NaN            2169           2107
2000            10                    NaN            2363           2140
5000            2                     NaN            2159          20316??? Error
Type HELP MEMORY for your options.
```



```
Data Type: Double
                                   Time Cost Normalized by CPU Clock
n               m                Scalar          Vector          Matrix
-----------------------------------------------------------------------------------
1               10                1778134         2809356        1829466
1               100                 27660          193902         174047
1               1000                27752          169976         169784
10              10                  27642           30183           7570
10              100                 27099           27603           4834
10              1000                  NaN           26121           4405
100             10                    NaN            4418           2139
100             100                   NaN            4396           2113
100             1000                  NaN            4435           2177
1000            1                     NaN            2185           2811
1000            10                    NaN            2198           2217
1000            50                    NaN            2219           2134
2000            10                    NaN            2499           2867
5000            2                     NaN            2291          18754
10000           2                     NaN            2117           2129Error in ==>
```

Interesting conclusions can be made from above runs for n:

- n=1: Scalars are about 6 times faster than both matrix or vector which cost the same.
- n=10: Matrix is about 4 times faster than both vector or scalar which cost the same.

- n=100: Matrix is about 13 times faster than scalars and about 2 times faster than vector.
- n=1000: Both Matrix and vector cost about the same and are about 13 times faster than scalars.
- n>>1000: At some point the memory limitation becomes an obstacle and the speed decrease first for matrixes (memory $\propto n^2$) and then for vectors (memory $\propto n$). At this stage the time cost is strongly dependent on available memory and CPU commitment at that instant; and that's why matrix result for n=2000 and above are strongly varying.

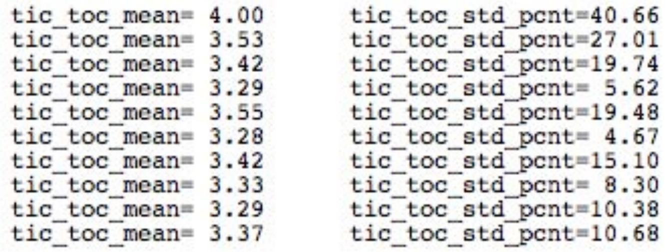
"m" is the number which determine on how much large set the results are averaged. Also interesting conclusions can be made from above runs for m.

- As it was stated earlier we have to have $m \times n^2 > 10^5$ to get rather consistent result for scalars. At this point the cost for scalar might still vary in 4th digit.
- For matrix with n=100 or higher, results are pretty constant regardless of m up to the 2nd digit.

Consequently from this point forward we set at least n=100 and m=200 for our computations. This is about the best "n" to speed up for matrixes (for our block of calculation not the individual functions\operands). However, we should remember above conclusions and specifically the fact that the cost is dependent on n and only accurate up to a few digits.

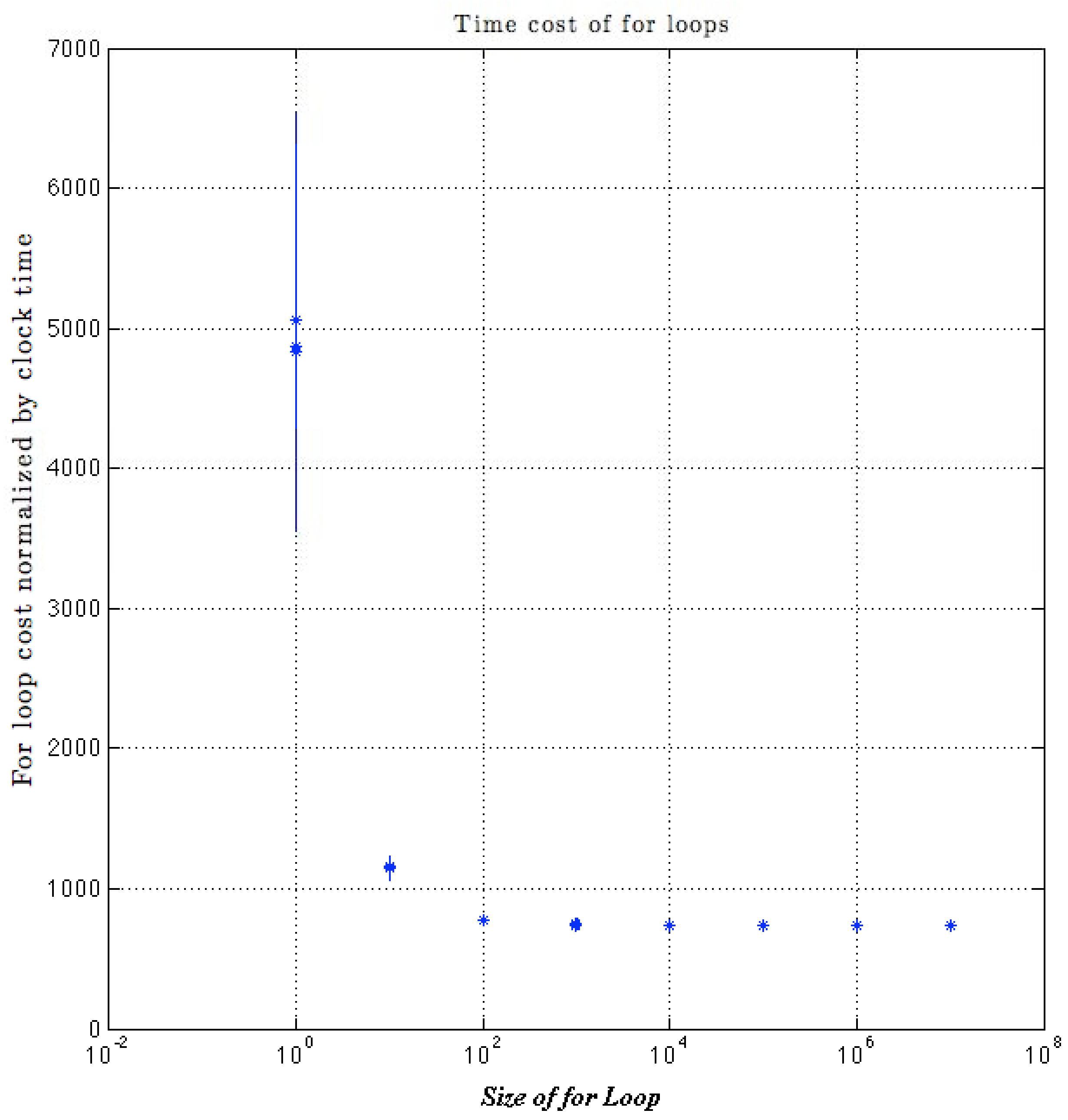
Furthermore, a careful examination shows that previous results are slightly affected by for loop cost. Part of attached program computes "tic toc" cost as well as pure "for loop" cost. The result is displayed in the graph in the next page and it is seen that "for loops" with more than 100 times of repetition there is about 740 clocks per step overhead due to the for loops. For a repetition about 10 times there is about 1200 clocks per step overhead, and finally for a repetition of 1 time the overhead is about 5000 clocks with huge variations. In the below figure, 3 averages, each averaging 10 runs, for each loop size and their corresponding error bar is shown. For the huge loop sizes the error bar (standard deviation) is negligible.

To modify the previous results for scalars when we have $m \times n^2 \geq 10^2$ we have to subtract 740 from printed clock time. For the vectors when $m \times n \geq 10^2$ we have to subtract $\frac{740}{n}$ from the results and finally for matrixes when $m \geq 10^2$ we have to subtract $\frac{740}{n^2}$. This all together can justify subtle differences for vector and matrix cost in previous pages while changing n or m.


```
tic_toc_mean= 4.00        tic_toc_std_pcnt=40.66
tic_toc_mean= 3.53        tic_toc_std_pcnt=27.01
tic_toc_mean= 3.42        tic_toc_std_pcnt=19.74
tic_toc_mean= 3.29        tic_toc_std_pcnt= 5.62
tic_toc_mean= 3.55        tic_toc_std_pcnt=19.48
tic_toc_mean= 3.28        tic_toc_std_pcnt= 4.67
tic_toc_mean= 3.42        tic_toc_std_pcnt=15.10
tic_toc_mean= 3.33        tic_toc_std_pcnt= 8.30
tic_toc_mean= 3.29        tic_toc_std_pcnt=10.38
tic_toc_mean= 3.37        tic_toc_std_pcnt=10.68
```

Also one might be concerned about the cost of "tic-toc" command. Within two for loops an average and standard deviation was computed for "tic toc" time (without command line output). The average was done on 10 averages, each averaging 100 times of "tic toc". The average was about 3.5-4.5 clocks and its standard deviation varied between about 5%-50% in different runs.

Consequently the “tic toc” time has not affected above results. A few consecutive run results in shown in the previous page.



Specific problem parts:

a)
b)
c)

MATLAB is a 4th generation language in which you do not need to specify the data type. There is a huge ease in MATLAB programming due to this, compared to 3rd generation languages like C or Fortran where you have to allocate and specify memory for each data type. There are also a lot of drawbacks: one is the time cost, which is higher. But the most serious one is the fact that MATLAB is basically an interpreter and it does not work by compiling your program in advance.

MATLAB has tried to decrease these drawbacks with smarter implementations. For example while we saw that the cost difference between a scalar and a matrix with huge "n" is about 13 times here, it could be about few hundred times in older MATLAB (before 6.5 version). MATLAB 6.5 included the JIT (Just-in-Time) technique and after that it is much less sensitive to vectorization for ordinary programs.

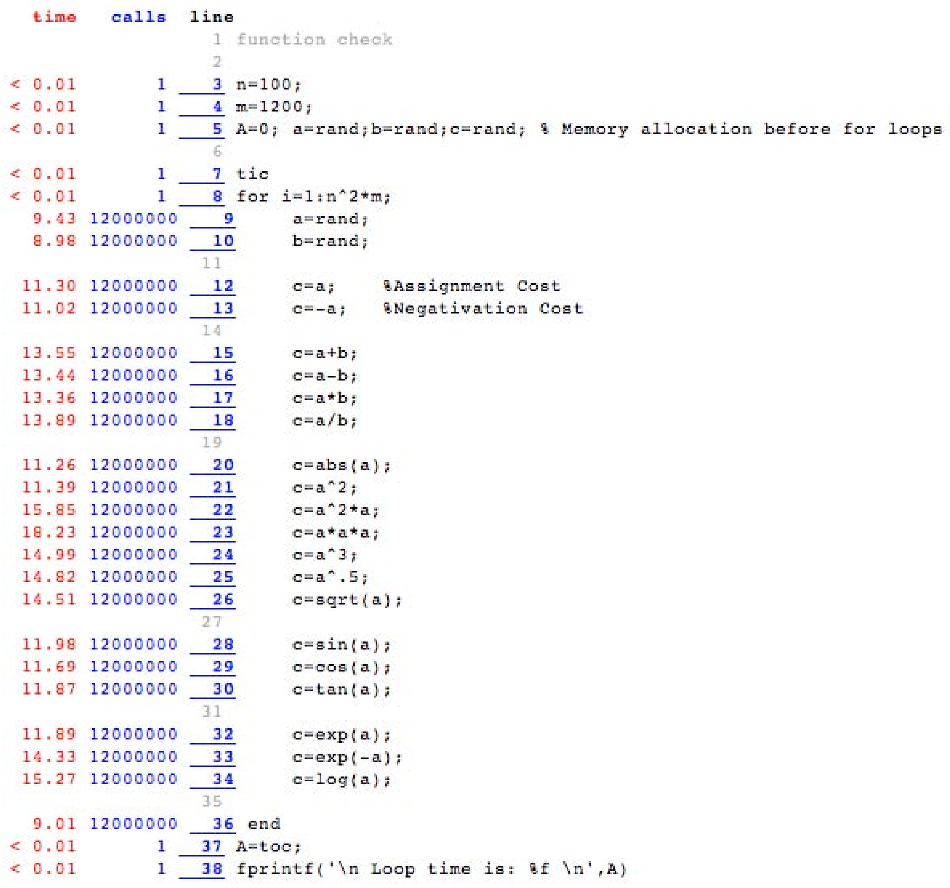
In 3rd generation languages assignment, memory allocation and memory release are totally different steps. Indeed even the assignment can mean either generating a new copy of data or just passing its pointer. On the other hand in MATLAB these all might happen at the same time and this means that there is a lot of ambiguity in the assignment cost. Furthermore smart MATLAB might notice that in y=x, y can be a pointer as long as y or x will not change. Here our numbers are updated frequently with rand and new assignments and we do not expect the assignment be a pointer one. Also for simplicity before each loop we allocate the "a,b,c" with the right size so our assignment is indeed just data copy. Also here we do not subtract the assignment cost from evaluation, but some might do that and report for example the pure time for subtraction (without the time to copy the results).

By now we know the time cost of tic-toc and for loops. One approach might be to surround a "for loop" with tic-toc and then subtract the additional overhead due to tic-toc and for loop. This is probably the best method, but it is not very convenient because we have to run the program for different task separately (and at different performance levels).

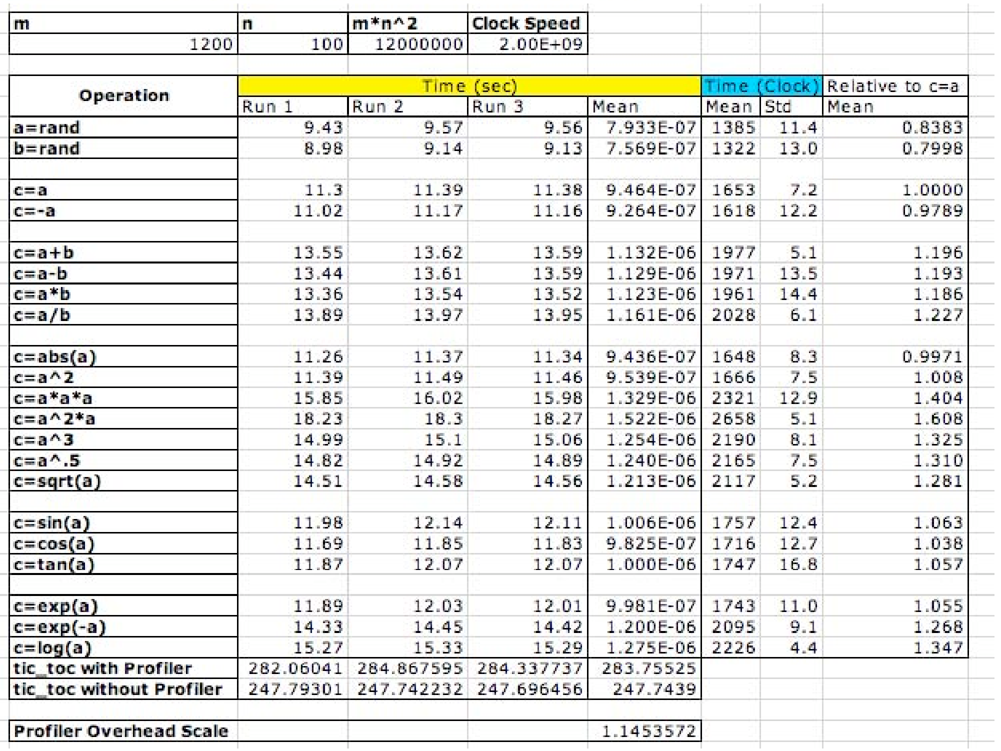
Here we adopt another simpler approach. We use time profiler. Time profiler is rather accurate, especially if we are concerned about relative time cost. In practice time profiler is used to find the bottleneck of programs. To get more accurate results:

1) We increase the "m, n" so that we get 4 significant digits from time profiler.
2) We scale the time cost of profiler for each line equally to omit the profiler time overhead. The scale value is set by a "tic-toc" time surrounding for loops and its output value with and without time profiling.
3) To have a better idea about statistical significance of results we do this for three times for all the processes.

The next page shows the scalar costs for a single run with time profiler. 3 times of "time profiler" run and 3 times of command line run has been used to generate the table shown in the next pages. The table shows time normalized by number of elements in seconds unit, clock unit and also relative to a=c time cost. Also the standard deviation is calculated in clock unit to monitor the significance of results.



```
 time     calls   line
                     1 function check
                     2
< 0.01        1      3 n=100;
< 0.01        1      4 m=1200;
< 0.01        1      5 A=0; a=rand;b=rand;c=rand; % Memory allocation before for loops
                     6
< 0.01        1      7 tic
< 0.01        1      8 for i=1:n^2*m;
  9.43 12000000      9     a=rand;
  8.98 12000000     10     b=rand;
                    11
 11.30 12000000     12     c=a;     %Assignment Cost
 11.02 12000000     13     c=-a;    %Negativation Cost
                    14
 13.55 12000000     15     c=a+b;
 13.44 12000000     16     c=a-b;
 13.36 12000000     17     c=a*b;
 13.89 12000000     18     c=a/b;
                    19
 11.26 12000000     20     c=abs(a);
 11.39 12000000     21     c=a^2;
 15.85 12000000     22     c=a^2*a;
 18.23 12000000     23     c=a*a*a;
 14.99 12000000     24     c=a^3;
 14.82 12000000     25     c=a^.5;
 14.51 12000000     26     c=sqrt(a);
                    27
 11.98 12000000     28     c=sin(a);
 11.69 12000000     29     c=cos(a);
 11.87 12000000     30     c=tan(a);
                    31
 11.89 12000000     32     c=exp(a);
 14.33 12000000     33     c=exp(-a);
 15.27 12000000     34     c=log(a);
                    35
  9.01 12000000     36 end
< 0.01        1     37 A=toc;
< 0.01        1     38 fprintf('\n Loop time is: %f \n',A)
```



m	n	m*n^2	Clock Speed
1200	100	12000000	2.00E+09

Operation	Time (sec)				Time (Clock)		Relative to c=a
	Run 1	Run 2	Run 3	Mean	Mean	Std	Mean
a=rand	9.43	9.57	9.56	7.933E-07	1385	11.4	0.8383
b=rand	8.98	9.14	9.13	7.569E-07	1322	13.0	0.7998
c=a	11.3	11.39	11.38	9.464E-07	1653	7.2	1.0000
c=-a	11.02	11.17	11.16	9.264E-07	1618	12.2	0.9789
c=a+b	13.55	13.62	13.59	1.132E-06	1977	5.1	1.196
c=a-b	13.44	13.61	13.59	1.129E-06	1971	13.5	1.193
c=a*b	13.36	13.54	13.52	1.123E-06	1961	14.4	1.186
c=a/b	13.89	13.97	13.95	1.161E-06	2028	6.1	1.227
c=abs(a)	11.26	11.37	11.34	9.436E-07	1648	8.3	0.9971
c=a^2	11.39	11.49	11.46	9.539E-07	1666	7.5	1.008
c=a*a*a	15.85	16.02	15.98	1.329E-06	2321	12.9	1.404
c=a^2*a	18.23	18.3	18.27	1.522E-06	2658	5.1	1.608
c=a^3	14.99	15.1	15.06	1.254E-06	2190	8.1	1.325
c=a^.5	14.82	14.92	14.89	1.240E-06	2165	7.5	1.310
c=sqrt(a)	14.51	14.58	14.56	1.213E-06	2117	5.2	1.281
c=sin(a)	11.98	12.14	12.11	1.006E-06	1757	12.4	1.063
c=cos(a)	11.69	11.85	11.83	9.825E-07	1716	12.7	1.038
c=tan(a)	11.87	12.07	12.07	1.000E-06	1747	16.8	1.057
c=exp(a)	11.89	12.03	12.01	9.981E-07	1743	11.0	1.055
c=exp(-a)	14.33	14.45	14.42	1.200E-06	2095	9.1	1.268
c=log(a)	15.27	15.33	15.29	1.275E-06	2226	4.4	1.347
tic_toc with Profiler	282.06041	284.867595	284.337737	283.75525			
tic_toc without Profiler	247.79301	247.742232	247.696456	247.7439			
Profiler Overhead Scale				1.1453572			

Then I have modified the code so that a,b,c are singles instead of default format of double (replacing for example 'a=rand;' with 'a=single(rand);'). And finally the same is done for matrixes with size $a_{n\times n}$. This all together needs 24 runs (6 runs for each pair of single\double and scalar\matrix) of the attached "check.m" file with slight modifications. The results are stored in C2p29_PSET2_4.xls EXCEL file and the file is attached (similar to previous page table). The next table shows the final results. Note that I have chosen m=1200 and n=100 for scalars and m=25000 and n=100 for matrixes to get reasonable run times for both matrix\scalars.

Operation	Time Cost in Clocks (each Clock= 0.5 nsec)			
	Single	Double	Single	Double
	Scalar		Matrix (100*100)	
a=rand, a=single(rand), a=rand(n,n), a=single(rand(n,n))	2157	1385	130	85
b=rand, b=single(rand), b=rand(n,n), b=single(rand(n,n))	2092	1322	123	81
c=a	1691	1653	4.6	4.8
c=-a	2047	1618	32	30
c=a+b	2414	1977	9.63	9.57
c=a-b	2406	1971	9.23	9.15
c=a*b, c=a.*b	2401	1961	9.71	9.12
c=a/b, c=a./b	2454	2028	50.1	42.2
c=abs(a)	2063	1648	26.5	22.5
c=a^2, c=a.^2	2092	1666	8.76	8.22
c=a*a*a, c=a.*a.*a	2740	2321	17.1	16.0
c=a^2*a, c=a.^2*a	3094	2658	18.1	15.0
c=a^3, c=a.^3	2616	2190	538	534
c=a^.5, c=a.^.5	2224	2165	203	197
c=sqrt(a)	2168	2117	95.1	91.8
c=sin(a)	2164	1757	110.1	102.1
c=cos(a)	2138	1716	128.5	123.0
c=tan(a)	2163	1747	175.8	170.5
c=exp(a)	2171	1743	165.3	162.8
c=exp(-a)	2510	2095	171.7	167.5
c=log(a)	2273	2226	230.1	212.5

Operation	Time Cost Relative to a=c				Matrix Speed Up	
	Single	Double	Single	Double	Single	Double
	Scalar		Matrix (100*100)		Scalar to Matrix(100*100)	
a=rand, a=single(rand), a=rand(n,n), a=single(rand(n,n))	1.276	0.8383	28.47	17.77	16.56	16.28
b=rand, b=single(rand), b=rand(n,n), b=single(rand(n,n))	1.237	0.7998	26.96	16.99	16.95	16.25
c=a	1.000	1.0000	1.000	1.000	369.4	345.2
c=-a	1.210	0.9789	7.080	6.200	63.16	54.51
c=a+b	1.428	1.196	2.103	2.000	250.8	206.5
c=a-b	1.423	1.193	2.017	1.911	260.6	215.5
c=a*b, c=a.*b	1.420	1.186	2.121	1.906	247.4	214.9
c=a/b, c=a./b	1.451	1.227	10.95	8.817	48.95	48.05
c=abs(a)	1.220	0.9971	5.799	4.700	77.75	73.24
c=a^2, c=a.^2	1.237	1.008	1.914	1.717	238.8	202.7
c=a*a*a, c=a.*a.*a	1.620	1.404	3.736	3.339	160.3	145.2
c=a^2*a, c=a.^2*a	1.830	1.608	3.966	3.128	170.5	177.5
c=a^3, c=a.^3	1.547	1.325	117.5	111.6	4.866	4.101
c=a^.5, c=a.^.5	1.315	1.310	44.40	41.13	10.95	10.99
c=sqrt(a)	1.282	1.281	20.78	19.17	22.80	23.07
c=sin(a)	1.280	1.063	24.06	21.32	19.65	17.22
c=cos(a)	1.264	1.038	28.07	25.70	16.64	13.95
c=tan(a)	1.279	1.057	38.42	35.62	12.30	10.24
c=exp(a)	1.284	1.055	36.11	34.01	13.13	10.71
c=exp(-a)	1.484	1.268	37.52	34.99	14.61	12.51
c=log(a)	1.344	1.347	50.28	44.40	9.880	10.474

A lot of interesting conclusion can be made from above tables. First the general ones:

- All the operations\functions are faster for doubles with respect to singles, for both matrixes and scalars. That's first because MATLAB is optimized for doubles, and secondly probably because some operations are done on doubles and then converted to singles. The only exception is a 10% difference for c=a between single and double matrixes; and that's because "c=a" is only a memory copy which is faster for smaller data. Consequently the single data does not sound appropriate "FOR MATLAB" [2] and we will not discuss it anymore.
- 2nd table in the previous page shows the time cost relative to c=a. Note that for double scalars this ratio varies from 0.84 to 1.83 (for rand function and c=a^2*a), while for double matrix the number varies from 1 to 111.6 (for c=a and c=a^3). This means that operational cost is magnified for matrixes; and for scalars the timing is mostly due to interpretation and data handling. For example for doubles the {+,-,*} are only about 2% faster than {/}, while for matrixes there are about 4.4 times faster. For scalars we do not care about operation\function type, but we just have to have smaller number of them.
- We can see that the relative time cost can be quite different for an operation\function between scalars and matrixes. This is shown in the table on the previous page. The speed up varies from 4.1 to 345 (for c=a.^3 and c=a). Consequently the speed up "13" we saw earlier was just an average.
- Probably the most interesting fact is that the operational cost is not solely dependent on mathematical complexity. Indeed a few very high-speed transistors makes CPU capable of computing any of those operations\functions very quickly for scalars (almost a clock when data are in CPU registers and the results will be in registers too). That's another reason why scalar cost are so close together. However, when it comes to huge data (like huge matrixes) the parallelism plays the key role for speed up. The parallel calculation relies on the CPU arithmetic capacity for each basic operation\function. The arithmetic capacity is based on usual needs of a computer. For example typical use of the computer needs a lot of $\{+,-,\times,\sqrt{x},x^2,\sin x\}$ and they are very fast (there are a lot of rooms for those calculations). On the other hand x^n where $n \neq 0.5, 2$ is not a popular function and a typical computer does not have a considerable capacity for that operation. As a result, amazingly x^3 is slower than any other operations\functions above for matrixes (including $\{\sqrt{x}, \sin x, \cos x, \tan x, e^x, e^{-x}, \ln x\}$). In those situations, we have to replace our function with popular ones. Here using $x^3 = x^2 \times x$ or $x^3 = x \times x \times x$ speeds up our calculation for about 35 times. Consequently we always prefer to compute x^n by recursive method for series. Also notice that due to different

[2] Still this CPU manual recommends singles for arithmetic, as much as possible instead of double. So the hardware is still better for singles but software preferences pushes for doubles.

computational capacity for different operations\functions, the speed up due to their vectorization differs and varies by changing the size of array[3].

- Another surprise: we have two calls to rand and the 2nd one is consistently faster. This is because the system still has that function in its memory and when it notices that it is going to be called again it keeps that in the memory. Consequently the 2nd run is faster. This is also true for other operands\functions, especially when they are similar or they need some similar temporary memory or variables. For example $\{+,-\}$ take the same time, but since we first do the summation, the subtraction seems to be faster. This is just an artifact and similarly $\{\times\}$ seems to be faster than both $\{+,-\}$, while this is not true (in the table we see that $\{\times\}$ is about 5% faster than $\{+\}$). This can be verified by changing the order of these operations (as I have done). Also it can be checked that c=a.^2.*a and c=a.*a.*a take the same time.
- Some specific results and speed ups are reported here. Note that they can be specific to the array size (here 100*100):
    - Operations $\{+,-,\times,\div\}$ take the same time for scalars. For huge matrixes $\{+,-,\times\}$ cost the same, while $\{\div\}$ cost about 4.4 times.
    - The square operator $\{x^2\}$ is a special function and is slightly (15%) faster than $\{x\times x\}$.
    - The square root can be evaluated faster by using "sqrt" function. Indeed sqrt is at least 2 times faster than equivalent x.^0.5 operation.
    - As stated earlier $\{x^n\}$ is a very slow operation if $n \neq 0.5, 2$. In those cases it is better to use $\{\times\}$ if possible. For example, replacing $\{x^3\}$ with $\{x^2\times x\}$ can accelerate our computation up to 35 times.
    - Apparently c=-a is about 3 times slower than c=b-a. This shows bad implementation of MATLAB. Indeed it is even 1.32 more costly than c=abs(a).
    - Among the trigonometric function, the $\{\sin x\}$ is about 1.21 times faster than $\{\cos x\}$ and about 1.67 times faster than $\{\tan x\}$. Still $\{\sin x\}$ is as costly as "sqrt(x)" and both cost about 10 times $\{+,-,\times\}$.
    - After $\{x^n\}$, the functions $\{\ln x\}$ is the most costly function. $\{\ln x\}$ costs about 1.3 times of $\{e^x\}$, while $\{e^x\}$ and $\{\tan x\}$ cost about the same.

- Finally again remember that cost of scalar operations\functions is all about the same in MATLAB. Consequently in general, for a faster scalar calculation you have to have smaller number of operations\functions (regardless of their type). On the other hand, for a faster matrix calculations, the functions\operations should be selected with respect to your arithmetic capacity; as a rule of thumb select the popular ones.

[3] Before we reach the arithmetic capacity, the cost per element is almost inversely proportional to the number of arrays element and the total time is almost constant. In these regime the whole data is processed in one pass. As long as we exceed the capacity we need extra passes to process all the data.

Problem 2.5 (15 Points): Computational cost in other language (EXTRA CREDIT)

Repeat previous problem for a basic language (like C++ or Fortan), but only on the scalars.

Problem 2.6 (10 Points): Computational cost of determinant evaluation

a) We want to have a recursive formula for computational cost of determinant evaluation by expansion of minors. Assume that for a matrix with size n the number of multiplication and summation\subtraction is $S_{\pm}(n),\, M_{\times}(n)$. Now compute $S_{\pm}(n+1),\, M_{\times}(n+1)$ by a recursive formula.

b) Now notice that $S_{\pm}(1)=0,\, M_{\times}(1)=0$ and try to develop a closed formula (or an order estimation) for $S_{\pm}(n),\, M_{\times}(n)$.

c) Study the textbook Box 9.1. Ignore the pivoting and compute or estimate the computational cost of determinant evaluation by Gauss elimination.

d) Compare "b" and "c" and discuss.

Solution:

a) For a matrix with size "n+1", we have to compute "n+1" minors, multiply them by matrix elements and then sum\subtract "n+1" numbers:

$$\begin{aligned} S_{\pm}\ (n+1) &= (n+1)\times S_{\pm}\ (n)+n \\ M_{\times}(n+1) &= (n+1)\times M_{\times}(n)+n+1 \end{aligned}$$

b) Note that $S_{\pm}(2)=1,\, M_{\times}(2)=2$. Consequently both $S_{\pm}(n),\, M_{\times}(n)$ will grow faster than $n!$. For example:

$$\begin{aligned} M_{\times}(n) &= n!+\sum_{i=3}^{n}\prod\nolimits_{j=n-i+3}^{j=n} j \qquad n\geq 3 \\ M_{\times}(6) &= 6!+(6\times5\times4\times3+6\times5\times4+6\times5+6) \qquad n\geq 3 \end{aligned}$$

Also by a little extension of summation notation (so that $if\ \ b<a \Rightarrow 0=\sum_{i=a}^{b} m(i)$):

$$M_{\times}(n) = n!+\sum_{i=3}^{n}\prod\nolimits_{j=n-i+3}^{j=n} j = n!\,(1+\sum_{i=2}^{n-1}\prod\nolimits_{j=1}^{j=i}\frac{1}{j}) \qquad n\geq 3$$

$$M_{\times}(n) = n!\,(1+\sum_{i=2}^{n-1}\frac{1}{i!}) = n!\sum_{i=1}^{n-1}\frac{1}{i!} \qquad n\geq 1$$

$$remember\ e^{x}\ series \Rightarrow \lim_{n\to\infty}\sum_{i=1}^{n-1}\frac{1}{i!}=e-1$$

$$\lim_{n\to\infty} M_{\times}(n) = n!\,(e-1)$$

Also by induction we can prove that: $S_{\pm}(n)=n!-1$.

c) After doing the gauss elimination, we are left with a triangular matrix and the determinant of such a matrix is the product of its diagonal elements. So we need "n" extra multiplications beside regular Guass elimination.

$$M_\times(n) = \frac{n^3}{3} + O(n^2) + n$$

$$S_\pm(n) = \frac{n^3}{3} + O(n)$$

Please note that:

i. If B results from A by exchanging two rows or columns, then det(A)=-det(B).
ii. If B results from A by multiplying one row or column with the number c, then det(B)=c*det(A) .
iii. If B results from A by adding a multiple of one row to another row, or a multiple of one column to another column, then det(A)=det(B).

Consequently the determinant of the initial matrix and the standard Gaussian eliminated one can only differ by a sign. They have the same sign if we have an even number of pivoting and opposite sign if we have an odd number of pivoting.

d) The cost (flops: or total number of floating point operations) of determinant evaluation by Gaussian elimination is of order $\frac{2n^3}{3}$. On the other hand, the cost of minors expansion is at least of order $n!e \approx 2.7n!$. Consequently the Gaussian elimination is strongly preferred and indeed the huge cost of minor expansion renders it as an impractical method for large matrixes.

Problem 2.7 (10 Points): Correct and effective implementation of numerical algorithms

For any numerical code we are concerned with three issues:

1. Generalization (scope of algorithm) and exceptions
2. Numerical stability
3. Effective and fast implementation

Here we have a very simple MATLAB code for solving a linear system with Gauss elimination and we are interested to investigate these issues for below code:

```
function [x]=gauss(A,b) % A: n*n matrix, b: n*1 vector where Ax=b

Ab=[A b];
n=length(A);
for i=1:n-1
```

```
        for j=i+1:n
            m=-Ab(i,i)/Ab(j,i);
            for k=1:n+1
                Ab(j,k)=m*Ab(j,k)+Ab(i,k);
            end
        end
    end
    x=zeros(n,1);
    for i=n:-1:1
        sum=Ab(i,n+1);
        for j=i+1:n

            sum=sum-x(j)*Ab(i,j);
        end
        x(i)=sum/Ab(i,i);
    end
```

a) Is that mathematically correct? Does it work for all matrixes or some matrixes exist which it fails to solve.

b) Is that numerically stable? Briefly discuss possible improvements.

c) Is that written effectively? Can we increase its speed? Is that operating with the minimum number of operations? More specifically about MATLAB, can the code be vectorized?

d) (EXTRA CREDIT: 10 Points) Improve the above code according to your answer in previous part (print the code).

Solution:

a) We can examine the code and we will see that it produces the right solution. However, does it work for all matrixes? The answer is "No". If we look at the code every line sounds safe unless these two lines:

```
1)  m=-Ab(i,i)/Ab(j,i);
2)  x(i)=sum/Ab(i,i);
```

Both lines can be problematic when the denominator is zero. For 1) we do not need a row summation (note that above "m" is inverse of standard definition of "m" in Gaussian elimination). In 2) A(i,i) is zero, and reminds the case when we have to do a pivoting. This codes lacks the pivoting and more importantly lacks to check the inversibility of the matrix; which means if it is at all possible to find a nonzero element by pivoting.

b) We have to do the pivoting, when the diagonal element of the active row becomes zero. To improve the numerical stability:

i. Do partial pivoting so that the pivot element has the highest absolute value among all the below elements in its own row. Even better, do the scaled partial pivoting, so that the absolute value of the pivot element divided by the maximum absolute coefficients in its own row is the largest.

ii. Do full pivoting by exchanging variable orders and rows appropriately.

iii. Scale the equations or unknowns for equilibration.

c) To increase the speed we have to choose the right schemes. Even after that we have to be careful to simplify it as much as possible so our code is as simple as possible. To that end when we know the value of something, we should not compute it. In particular if A(i,j) is zero then we do not need to compute the “m”. Beside that in the most internal loop we are varying “k” from 1 to n+1, while elements before j will remain zero.

Original Code

```
m=-Ab(i,i)/Ab(j,i);
        for k=1:n+1
            Ab(j,k)=m*Ab(j,k)+Ab(i,k);
        end
```

Modified

```
if Ab(j,i)~=0
            m=-Ab(j,i)/Ab(i,i);
            Ab(j,i)=0
            for k=i+1:n+1
              Ab(j,k)=Ab(j,k)+m*Ab(i,k);
            end
end
```

Also regarding the computer or MATLAB implementation we have to vectorize our code as much as possible. In the MATLAB this is equivalent to replace “for” loops with equivalent matrix or vector versions. So we can modify the code:

Original Code

```
Ab=[A b];
n=length(A);
for i=1:n-1
    for j=i+1:n
        m=-Ab(i,i)/Ab(j,i);
        for k=1:n+1
            Ab(j,k)=m*Ab(j,k)+Ab(i,k);
        end
    end
end
x=zeros(n,1);
for i=n:-1:1
    sum=Ab(i,n+1);
    for j=i+1:n

        sum=sum-x(j)*Ab(i,j);
    end
    x(i)=sum/Ab(i,i);
end
```

Modified (Still have to add pivoting in case Ab(i,i) becomes zero)

```
Ab=[A b];
n=length(A);
for i=1:n-1
    for j=i+1:n
        if Ab(j,i)~=0
            m=-Ab(j,i)/Ab(i,i);
            Ab(j,i:end)=Ab(j,i:end)+m*Ab(i,i:end);
        end
    end
end
x=zeros(n,1);
for i=n:-1:1
    sum=Ab(i,n+1)
    if i<n
        sum=sum-Ab(i,i+1:n)*x(i+1:end);
    end
    x(i)=sum/Ab(i,i)
end
```

d) A modified code with scaled full pivoting is shown below. The related file is attached by C2p29_PSET1_7.m name.

```
Ab=[A b];
n=length(A);
for i=1:n-1
    Scaled_Coeff=Ab(i:n,i)./max(max(Ab(i:n,i:n),[],2),abs(min(Ab(i:n,i:n),[],2)));
    [pivot,index]=max(Scaled_Coeff);
    if pivot==0
        error('Matrix is singular');
    elseif index~=1
        temp               =Ab(i         ,i:end);
        Ab(i        ,i:end)=Ab(i-1+index,i:end);
        Ab(i-1+index,i:end)=temp;
    end
    for j=i+1:n
        if Ab(j,i)~=0
            m=-Ab(j,i)/Ab(i,i);
            Ab(j,i:end)=Ab(j,i:end)+m*Ab(i,i:end);
        end
    end
end
x=zeros(n,1);
for i=n:-1:1
    sum=Ab(i,n+1);
    if i<n
        sum=sum-Ab(i,i+1:n)*x(i+1:end);
    end
    x(i)=sum/Ab(i,i);
end
```