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

Here is what function should be like:

\[
[x\_solution, x\_iterations, f\_iterations] = \text{solver}(\text{func\_name})
\]
\[
[x\_solution, x\_iterations, f\_iterations] = \text{solver}(\text{func\_name}, x\_guess)
\]
\[
[x\_solution, x\_iterations, f\_iterations] = \text{solver}(\text{func\_name}, x\_guess, \text{OPTIONS})
\]

A few example calls are shown:

```matlab
[x_f] = \text{solver}('x-sin(x)');
[x_f] = \text{solver}(@(x) sin(x));  \% where func is a function handle
[x_f, x_it, f_it] = \text{solver}(@(x) sin(x));
[x_f, x_it, f_it] = \text{solver}(@(x) sin(x), 0);
[x_f, x_it, f_it] = \text{solver}(@(x) sin(x), -2, 'method', 'Bi-Section');
[x_f, x_it, f_it] = \text{solver}(@(x) sin(x), -2, 'Method', 'bi-section');
[x_f, x_it, f_it] = \text{solver}(@(x) sin(x), -2, 'plot', 'off');
[x_f, x_it, f_it] = \text{solver}(@(x) sin(x), -2, 'method', 'Bi-Section', 'plot', 'off');
[x_f, x_it, f_it] = \text{solver}(@(x) sin(x), -2, 'plot', 'off', 'method', 'Bi-Section');
[x_f, x_it, f_it] = \text{solver}('x-sin(x)', 2, 'method', 'Newton');
[x_f, x_it, f_it] = \text{solver}(@(x) sin(x), 2, 'method', 'Newton', 'f\_derivative', @d_func);
[x_f, x_it, f_it] = \text{solver}('x-sin(x)', 2, 'max\_iteration', 10, 'method', 'Newton');
[x_f, x_it, f_it] = \text{solver}(@(x) sin(x), 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 first two roots where \( x > 20 \).

e) Repeat part c and find the first 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:

\[
\begin{align*}
    x \to +\infty &\Rightarrow f(x) \equiv e^x \sin x \\
    x \to -\infty &\Rightarrow f(x) \equiv e^{-x} \cos^2 2x
\end{align*}
\]

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

\[
    x \to +\infty \Rightarrow 0 \equiv e^x \sin x \Rightarrow x \equiv k\pi, \text{ where } k \to +\infty
\]

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 close 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.

\[ x \to -\infty \Rightarrow 0 \equiv e^{-x} \cos^2 2x, \quad \Rightarrow x_c \equiv \frac{k\pi}{2} + \frac{\pi}{4}, \quad \text{provided } \sin x_c < 0 \text{ where } k \to -\infty \]

\[ x_c = 2k\pi - \frac{\pi}{4}, \quad x_c = 2k\pi - \frac{3\pi}{4} \quad k \to -\infty \]

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

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. 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 \( \text{eps}(1) \) 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.
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.
2.29: Numerical Fluid Mechanics

Solution of Problem Set 2

\[ f(x) = e^x \sin x + e^{-2x} \cos 2x \] root using initial guess of \( x_0 = 3.1, 3.2 \)

\[ f(x) = e^x \sin x + e^{-2x} \cos 2x \] root using initial guess of \( x_0 = -0.7, -0.5 \)
2.29: Numerical Fluid Mechanics

Solution of Problem Set 2

\[ f(x) = e^x \sin x + e^{-x} \cos^2 2x \] root using initial guess of \( x_0 = -1.1, -0.9 \)

\[ f(x) = e^x \sin x + e^{-x} \cos^2 2x \] root using initial guess of \( x_0 = -2.35, -2.2 \)
Here are the solutions by Newton’s method. Note that approximate solutions are still very good, albeit $x$ is not very large ($\pi \equiv 3.14$, $-\frac{\pi}{4} = -0.79$, $-\frac{3\pi}{4} = -2.36$).

Final Solution: $f(x) = -2.3938516328855954 \times 10^{-16}$ with NEWTON method and 7 iterations
Final Solution: $f(x) = -5.689853e-15$ with NEWTON method and 7 iterations
Final Solution: $f(x) = -5.689853e-15$ with NEWTON method and 7 iterations

Since $\frac{20}{\pi} \equiv 6.37$, 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 $|x| \rightarrow \infty$. 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.
2.29: Numerical Fluid Mechanics

Solution of Problem Set 2

\[ f(x) = e^x \sin x + e^{-3x} \cos 2x \text{ root using initial guess of } x_0 = 21.8911, 22.0911 \]

\[ f(x) = e^x \sin x + e^{-3x} \cos 2x \text{ root using initial guess of } x_0 = 25.0327, 25.2327 \]
\[ \frac{20}{2\pi} = 3.18, \quad -(2\pi \times 3 + \frac{\pi}{4}) \equiv -19.63, \quad -(2\pi \times 4 + \frac{\pi}{4}) \equiv -25.91, \quad -(2\pi \times 3 + \frac{3\pi}{4}) \equiv -21.21 \]

Consequently, the roots are expected to be around \(-(2\pi \times 3 + \frac{3\pi}{4}) \equiv -21.21\). This is almost a double root and the pair of initial guesses is constructed with \pm 0.2\) offset from this middle point. To force the scheme to find the right or left root \(-(6\pi + \frac{3\pi}{4}) < x, \quad or \quad -(6\pi + \frac{3\pi}{4}) > x\) 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 \(-f(-(6\pi + \frac{3\pi}{4})) \equiv -4.3 \times 10^{-10}\) while \(-f(-(6\pi + \frac{3\pi}{4}) \pm 0.2) \equiv 2.5 \times 10^8\); 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.
2.29: Numerical Fluid Mechanics

Solution of Problem Set 2

\[ f(x) = e^x \sin x + e^{-x} \cos^2 2x \text{ root using initial guess of } x_0 = -21.2058, -21.0058 \]

- Newton
- Secant
- Modified False-Position
- False-Position
- Bi-Section

\[ \log_{10}(e_{\text{rel}}(x_n)) \text{ (relative error of } x) \]

\[ n \text{ (Iteration Number)} \]

\[ x \]

\[ -21.2 \]
\[ -21.22 \]
\[ -21.24 \]
\[ -21.26 \]
\[ -21.28 \]
\[ -21.30 \]
\[ -21.32 \]

\[ 2 \]
\[ 4 \]
\[ 6 \]
\[ 8 \]
\[ 10 \]
\[ 12 \]
\[ 14 \]
\[ 16 \]
\[ 18 \]
\[ 20 \]
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.
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.
Textbook problem 5.9

![Graph of the function](image1)

\[ x^2 \text{abs}(\cos(x^5))-5 \]

\[
\begin{array}{ccccccc}
 x & x_1 & x_2 & x_3 & f_1 & f_2 & f_3 & x_{\text{rel error}} \\
 0 & -3.6000000 & -3.6000000 & -3.6000000 & -3.7189453 & -1.88192e-06 & -1.66835e-06 & -1.58169e-06 & -7.55757e-08 \\
\end{array}
\]

Final Solution: \( f(x) = -3.7431469, x = 1.11021e-02 \) with \texttt{false-position} method and 2 iterations

Textbook problem 5.15

![Graph of the function](image2)

\[ 1 - Q^2 / g (3+y) / 3 / y^2 / z^2 \]

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
2.29: Numerical Fluid Mechanics Solution of Problem Set 2

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.

```matlab
>> Q=20;
>> g99.8;
>> d = @(y)1-Q/(y^2*exp(3-y^2/2));
>> subplot(1,2,1)
>> fplot(d,[-6.5,6.5]);
>> xrel solver([1.5 1.5], 'method', 'false-position', 'reltolerance', .01, 'maxiteration', 10);

<table>
<thead>
<tr>
<th>k</th>
<th>x_l</th>
<th>x_u</th>
<th>x_n</th>
<th>f_l</th>
<th>f_u</th>
<th>f_n</th>
<th>x_rel_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>1</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>2</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>3</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>4</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>5</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>6</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>7</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>8</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>9</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>10</td>
<td>6.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
<td>2.50000000</td>
<td>-2.50000000</td>
</tr>
</tbody>
</table>

Final Solutions: \(x = 2.059016051668592\) with FALSE-POSITION method and 10 iterations

```matlab
>> x_w => solver([1.0 1.0], 'method', 'false-position', 'reltolerance', .01, 'maxiteration', 10);

<table>
<thead>
<tr>
<th>k</th>
<th>x_l</th>
<th>x_u</th>
<th>x_n</th>
<th>f_l</th>
<th>f_u</th>
<th>f_n</th>
<th>x_rel_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
</tr>
<tr>
<td>1</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
</tr>
<tr>
<td>2</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
</tr>
<tr>
<td>3</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
</tr>
<tr>
<td>4</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
</tr>
<tr>
<td>5</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
</tr>
<tr>
<td>6</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
</tr>
<tr>
<td>7</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
<td>1.50000000</td>
</tr>
</tbody>
</table>

Final Solutions: \(x = 1.507612500000000\) 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 \(40^\circ C\) and by the end of the \(n\)th iteration it will be at most \(40^\circ C/2^n\). Now we have to solve \(40^\circ C/2^n \leq 0.05^\circ C\). The minimum \(n\) to satisfy this inequality is \(n=10\) (which results in absolute error less than \(0.039^\circ C\)) and we at least need 10 iterations.

```matlab
>> error_ratio = log(error_ratio)/log(2)
>> n_max = ceil(log(error_ratio)/log(2))
>> n_max = 10
```

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).
2.29: Numerical Fluid Mechanics
Solution of Problem Set 2

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 \( \frac{dg}{dx} = \left| \frac{\cos\sqrt{x}}{\sqrt{x}} \right| \leq 1 \). Unfortunately, \( \left| \frac{dg}{dx} \right| (x = 0.5) \approx 1.007 \) and the fixed
point with this $g$ can be unstable. However, since $g(x_0 = 0.5) \equiv 1.29$ it will escape from unstable region. Calculations are done in attached C2p29_PSET2_6p1.m file.

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

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

---

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.

$$
a) \quad f \quad f'
$$

$$
b) \quad 2f \quad f'
$$

$$
c) \quad ff' \quad f'^2 - ff''
$$

---
A relative error of $10^{-16}$ is imposed. The case b and c converges within 3 iterations. However, case “a” requires about 25 iterations; 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 derivative.

Textbook problem 6.15

At the steady state $\frac{dc}{dt} = 0$ 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 $\frac{f(c + \delta) - f(c)}{\delta}$. 
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 2\textsuperscript{nd} one has a guaranteed convergence for the given interval to \( c_r \equiv 4.62 \) root.

```matlab
>> clc;
>> clear all;
>> close all;
>> g = @(x)exp(x) + cos(x);
>> f = @(x)2*cos(x) - 1;
>> x = 0:0.2:5;
>> a = 0;
>> b = 5;
>> for n = 1:10
        tol = 1e-6;
        x(n) = a + (b-a)/2;
        f(n) = f(x(n));
        if abs(f(n)) < tol,
            break;
        end
    end
>> x
```

Final Solution: \( f(x) = \frac{4.62407997972831 + 1.4794559e^{-91}}{2} \) with NESTOR method and 3 iterations.

**Solution of Problem Set 2**
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.

```matlab
>> f= @(x) tanh(x.^2-4);
>> df= @(x) (2-tanh(x.^2-9)).^2*x;
>> [x,xo,lr]=solver(f,3.1,method,'Newton',f_derivative,DF,rel_tolerance,1e-16,max_iterations,4);

<table>
<thead>
<tr>
<th>k</th>
<th>x_o</th>
<th>x_n</th>
<th>f_o</th>
<th>f_n</th>
<th>df_o</th>
<th>x_rel_error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.5953</td>
<td>7.9604</td>
<td>-1.4058</td>
<td>-3.1413</td>
<td>7.5621e-04</td>
<td>+7.1593e-01</td>
</tr>
<tr>
<td>2</td>
<td>3.6304</td>
<td>3.0030</td>
<td>-2.0008</td>
<td>0.0000</td>
<td>6.0000e-06</td>
<td>+1.1113e-06</td>
</tr>
</tbody>
</table>

x = 3.000000000000000 is the exact solution.

Final solution: E(x) = 3.000000000000000e-06 with NEWTON method and 3 iterations.
```
Textbook problem 6.25 (Extra Credit)

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

\[
y_1(x) = +\sqrt{16 - (x + 1)^2} + 2, \quad \text{First Quadrant}
\]
\[
y_2(x) = -\sqrt{16 - (x + 1)^2} + 2, \quad \text{Fourth Quadrant}
\]

The results are shown on the next page. It converges for the 4\textsuperscript{th} quadrant but not the 1\textsuperscript{st} one. This is because in the 1\textsuperscript{st} quadrant \( 2 \leq y \leq 2 + 4 \) and we do not have a root (so it diverges toward complex numbers), while for the 4\textsuperscript{th} quadrant \(-4 + 2 \leq y \leq 2\) a root exists.
Textbook problem 10.6

Similar to “b” we adopt 3 significant digits.

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

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

$$m_{31} = \frac{1}{10} = +0.1 \quad m_{32} = \frac{0.8}{-5.4} = -0.148$$

$$LU = A, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ -0.3 & 1 & 0 \\ +0.1 & -0.148 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 10 & 2 & -1 \\ 0 & -5.4 & 1.7 \\ 0 & 0 & 5.35 \end{bmatrix}$$
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 4\(^{th}\) and 5\(^{th}\) digits.

\[
L = \\
\begin{pmatrix}
1.0000 & 0 & 0 \\
-0.3000 & 1.0000 & 0 \\
-0.1000 & -0.1000 & 1.0000 \\
\end{pmatrix}
\]

\[
U = \\
\begin{pmatrix}
10.0000 & 2.0000 & -1.0000 \\
0 & -5.0000 & 1.7000 \\
0 & 0 & 5.3000 \\
\end{pmatrix}
\]

\[
\text{inv}_A = \\
\begin{pmatrix}
0.1110 & 0.0260 & 0.0069 \\
-0.0089 & -0.1740 & 0.0089 \\
-0.0164 & 0.0277 & 0.1670 \\
\end{pmatrix}
\]

\[
\text{inv}_A_{\text{exact}} = \\
\begin{pmatrix}
0.1167 & 0.0581 & 0.0069 \\
-0.0088 & -0.1745 & 0.0089 \\
-0.0164 & 0.0277 & 0.1669 \\
\end{pmatrix}
\]

\[
A_{\text{inv}_A} = \\
\begin{pmatrix}
1.0000 & 0.0000 & 0 \\
0.0000 & 1.0000 & 0 \\
0.0000 & 0.0000 & 1.0000 \\
\end{pmatrix}
\]

\[
\mathbf{x}_{\text{sol}} = \\
\begin{pmatrix}
0.5000 \\
9.0000 \\
-6.0000 \\
\end{pmatrix}
\]

\[
\mathbf{x}_{\text{sol\ exact}} = \\
\begin{pmatrix}
0.5000 \\
9.0000 \\
-6.0000 \\
\end{pmatrix}
\]

\[
\text{residual} = \\
\begin{pmatrix}
0 \\
0 \\
0 \\
\end{pmatrix}
\]

**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.
2.29: Numerical Fluid Mechanics

Solution of Problem Set 2

>> A=[8 2 -10;9 1 3;15 -1 6]
A =
 8   2  -10
 9   1   3
15  -1   6
>> A_c=norm(A,'fro')
A_c =
22.8264
>> A_1=norm(A,1)
A_1 =
32
>> A_inf=norm(A, 'inf')
A_inf =
22
>> A_sc=[A(1,1)/-10;A(2,1)/-9;A(3,1)/15]
A_sc =
-0.0000   -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;6 9 16 25 36;16 25 36 49 64;25 36 49 64 81]
A =
 1   4    9   16   25
 6   9   16   25   36
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 = 3.253077e-10.

inv_A =
1.00e+15 *
0.0804  -0.0938   -0.2011  -0.3619   -0.1674
 0.0000  -0.1876   -0.2940   -0.5629    0.1876
 -0.4925   -1.2599  -0.4925    0.3217
 0.4925  -0.0405    2.0863   -0.6148
 -0.2513    0.6691   -0.0402   -0.5227    0.2547
>> inv_A_inf=norm(inv_A, 'inf')
inv_A_inf =
3.6602e+15
>> K_inf=A_inf*inv_A_inf
K_inf =
9.8438e+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 = 1.250227e-16.
K_inf =
   9.8126e+17
>> K_inf_sc=cond(A_sc, 'inf')
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 1.429738e-16.
    K_inf_sc =
   1.42974e+17
```

Since the condition number is about \(9.8 \times 10^{17} \equiv 10^{18}\) 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 \(\log_{10}\left(\frac{9.84}{1.42}\right) \equiv 0.84\).
Textbook problem 10.14

```matlab
>> A=[16 4 2; 4 2 1; 49 7 1]
A =
16  4  2
 4  2  1
49  7  1
>> norm(A, 'fro')
norm_fro =
217.4043
>> norm(A, 'inf')
norm_inf =
323
>> norm(A, 2)
norm_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 \{+,−,×,+\} and some functions \{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 \(2 \times 10^9\) 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 ×,+. 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
- MATLAB 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 $a_{m \times 1}$ 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:

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>Scalar</th>
<th>Vector</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1242120</td>
<td>1774776</td>
<td>2124479</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>30501</td>
<td>258519</td>
<td>170592</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>34547</td>
<td>135036</td>
<td>176896</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>36946</td>
<td>31310</td>
<td>7529</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>28130</td>
<td>26888</td>
<td>4491</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>26407</td>
<td>25801</td>
<td>4379</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>25989</td>
<td>4380</td>
<td>2143</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>25965</td>
<td>4376</td>
<td>2108</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>25969</td>
<td>4378</td>
<td>2114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>Scalar</th>
<th>Vector</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1252448</td>
<td>2010936</td>
<td>2026218</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>27299</td>
<td>195356</td>
<td>207161</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>26259</td>
<td>17985</td>
<td>179485</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>32703</td>
<td>31237</td>
<td>7785</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>28194</td>
<td>26739</td>
<td>4431</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>26117</td>
<td>25927</td>
<td>4421</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>26595</td>
<td>4399</td>
<td>2172</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>26006</td>
<td>4386</td>
<td>2138</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>26001</td>
<td>4385</td>
<td>2117</td>
</tr>
</tbody>
</table>
Mathematically a scalar and a vector or matrix with \( n=1 \) are equivalent. This is also true in MATLAB, but their computation time is different. As seen above for \( n=1 \) a scalar is about \( \frac{180000}{30000} = 6 \) times faster.

When \( “n” \) becomes very big (here like 5000, which require \( 25 \times 8 \sim 200 \) 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 \( 4^{th} \) 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.

<table>
<thead>
<tr>
<th>Data Type: Double</th>
<th>Time Cost Normalized by CPU Clock</th>
<th>Scalar</th>
<th>Vector</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n ) ( m )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1230177</td>
<td>1976490</td>
<td>1774458</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>36535</td>
<td>184094</td>
<td>179122</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>27177</td>
<td>171954</td>
<td>170353</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>29340</td>
<td>29685</td>
<td>7553</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>27156</td>
<td>27286</td>
<td>4020</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>26591</td>
<td>25908</td>
<td>4391</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>26036</td>
<td>4395</td>
<td>2158</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>25952</td>
<td>4421</td>
<td>2210</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>26015</td>
<td>4391</td>
<td>2117</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>25982</td>
<td>2168</td>
<td>14903</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>26000</td>
<td>2169</td>
<td>2132</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Type: Double</th>
<th>Time Cost Normalized by CPU Clock</th>
<th>Scalar</th>
<th>Vector</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n ) ( m )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1264623</td>
<td>2048789</td>
<td>1489142</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>37962</td>
<td>183291</td>
<td>176128</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>27763</td>
<td>169343</td>
<td>169115</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>27844</td>
<td>29318</td>
<td>7640</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>26979</td>
<td>27903</td>
<td>4647</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>26483</td>
<td>25824</td>
<td>4387</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>26015</td>
<td>4391</td>
<td>2162</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>25986</td>
<td>4394</td>
<td>2131</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>25973</td>
<td>4382</td>
<td>2122</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>25968</td>
<td>2167</td>
<td>2393</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>25979</td>
<td>2169</td>
<td>2129</td>
</tr>
<tr>
<td>1000</td>
<td>50</td>
<td>26164</td>
<td>2174</td>
<td>2143</td>
</tr>
<tr>
<td>2000</td>
<td>10</td>
<td>26429</td>
<td>2520</td>
<td>3050</td>
</tr>
</tbody>
</table>

Type `HELP MEMORY` for your options.
### Data Type: Double

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>Scalar</th>
<th>Vector</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>19255.168</td>
<td>24415.168</td>
<td>17481.168</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>29623</td>
<td>211832</td>
<td>177440</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>27694</td>
<td>171021</td>
<td>178101</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>32686</td>
<td>51681</td>
<td>9498</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>26984</td>
<td>26944</td>
<td>4996</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>NaN</td>
<td>26020</td>
<td>4388</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>NaN</td>
<td>4414</td>
<td>2126</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>NaN</td>
<td>4391</td>
<td>2102</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>NaN</td>
<td>4387</td>
<td>2230</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>NaN</td>
<td>2186</td>
<td>2322</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>NaN</td>
<td>2165</td>
<td>2125</td>
</tr>
<tr>
<td>1000</td>
<td>50</td>
<td>NaN</td>
<td>2173</td>
<td>2103</td>
</tr>
<tr>
<td>2000</td>
<td>10</td>
<td>NaN</td>
<td>1329</td>
<td>814</td>
</tr>
<tr>
<td>5000</td>
<td>2</td>
<td>NaN</td>
<td>2145</td>
<td>8743?? Error</td>
</tr>
</tbody>
</table>

Type HELP MEMORY for your options.

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

### Data Type: Double

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>Scalar</th>
<th>Vector</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>287114.158</td>
<td>137051.86</td>
<td>186528.9</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>34334</td>
<td>188819</td>
<td>170120</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>26974</td>
<td>160965</td>
<td>167866</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>27004</td>
<td>29439</td>
<td>7513</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>26558</td>
<td>25955</td>
<td>4453</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>NaN</td>
<td>25926</td>
<td>4370</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>NaN</td>
<td>4410</td>
<td>2129</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>NaN</td>
<td>4388</td>
<td>2096</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>NaN</td>
<td>4384</td>
<td>2206</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>NaN</td>
<td>7257</td>
<td>7444</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>NaN</td>
<td>2164</td>
<td>2120</td>
</tr>
<tr>
<td>1000</td>
<td>50</td>
<td>NaN</td>
<td>2169</td>
<td>2107</td>
</tr>
<tr>
<td>2000</td>
<td>10</td>
<td>NaN</td>
<td>2363</td>
<td>2140</td>
</tr>
<tr>
<td>5000</td>
<td>2</td>
<td>NaN</td>
<td>2159</td>
<td>20316?? Error</td>
</tr>
</tbody>
</table>

Type HELP MEMORY for your options.

### Data Type: Double

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>Scalar</th>
<th>Vector</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>17781.94</td>
<td>20955.36</td>
<td>18294.60</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>27660</td>
<td>193902</td>
<td>174047</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>27752</td>
<td>169976</td>
<td>169784</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>27442</td>
<td>30183</td>
<td>7570</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>27099</td>
<td>27603</td>
<td>4584</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>NaN</td>
<td>26121</td>
<td>4305</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>NaN</td>
<td>4418</td>
<td>2139</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>NaN</td>
<td>4396</td>
<td>2113</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>NaN</td>
<td>4435</td>
<td>2177</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>NaN</td>
<td>2185</td>
<td>2811</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>NaN</td>
<td>2198</td>
<td>4241/</td>
</tr>
<tr>
<td>1000</td>
<td>50</td>
<td>NaN</td>
<td>2219</td>
<td>2134</td>
</tr>
<tr>
<td>2000</td>
<td>10</td>
<td>NaN</td>
<td>2499</td>
<td>2887</td>
</tr>
<tr>
<td>5000</td>
<td>2</td>
<td>NaN</td>
<td>2291</td>
<td>18754</td>
</tr>
<tr>
<td>10000</td>
<td>2</td>
<td>NaN</td>
<td>2117</td>
<td>2129 Error in =&gt;</td>
</tr>
</tbody>
</table>

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 ∝ n^2) and then for vectors (memory ∝ 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.

<table>
<thead>
<tr>
<th>tic_toc_mean</th>
<th>tic_toc_std_pcnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.00</td>
<td>40.66</td>
</tr>
<tr>
<td>3.53</td>
<td>27.01</td>
</tr>
<tr>
<td>3.42</td>
<td>19.74</td>
</tr>
<tr>
<td>3.29</td>
<td>5.62</td>
</tr>
<tr>
<td>3.55</td>
<td>19.48</td>
</tr>
<tr>
<td>3.42</td>
<td>15.10</td>
</tr>
<tr>
<td>3.33</td>
<td>8.30</td>
</tr>
<tr>
<td>3.29</td>
<td>10.38</td>
</tr>
<tr>
<td>3.37</td>
<td>10.68</td>
</tr>
</tbody>
</table>

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.

![Graph of Time cost of for loops](image)

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.
2.29: Numerical Fluid Mechanics  
Solution of Problem Set 2

```
   time  calls  line
         1 function check
         2
        < 0.01  1  a=100;
        < 0.01  1  e=1200;
        < 0.01  1  A=0; a=rand;b=rand;c=rand; % Memory allocation before for loops
         6
        < 0.01  1    tic
        < 0.01  1    for i=1:2000;
         9.12  12000000  2    e=rand;
         9.96  12000000  10    b=rand;
        13
        11.30  12000000  12    c=a; %Assignment Cost
        11.02  12000000  13    c=a; %Negation Cost
        14
        13.55  12000000  15    c=a+b;
        13.44  12000000  16    c=a-b;
        13.36  12000000  17    c=a*b;
        13.09  12000000  18    c=a/b;
        19
        11.26  12000000  20    c=abs(a);
        11.20  12000000  21    c=abs(b);
        15.85  12000000  22    c=abs(a);
        18.22  12000000  23    c=abs(a);
        11.90  12000000  24    c=abs(b);
        15.82  12000000  25    c=abs(c);
        15.51  12000000  26    c=e^sqrt(a);
        27
        11.96  12000000  29    c=sin(a);
        11.60  12000000  30    c=sin(a);
        11.87  12000000  31    c=tan(a);
        31
        11.89  12000000  32    c=exp(a);
        14.33  12000000  33    c=exp(-a);
        15.37  12000000  34    c=exp(a);
        35
         9.01  12000000  36 end
        < 0.01  1    A=toc;
        < 0.01  1    fprintf('
\nTime (sec) 
\n\hline
m & n & m*n^2 & Clock Speed  
\hline
1200 & 100 & 12000000 & 2.00E+09  
\hline
\end{longtable}
```

```
\begin{longtable}{|c|c|c|c|c|c|c|}
\hline
Operation & Time (sec) & Time (Clock) & Relative to c=a & Mean & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & Mean & Stc & M
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 $a_{m \times n}$. 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Cost in Clocks</th>
<th>Single Clock=0.5 nsec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single</td>
<td>Double</td>
</tr>
<tr>
<td>a=rand, a=single(rand), b=rand(n,n), a=single(rand(n,n))</td>
<td>2157</td>
<td>1385</td>
</tr>
<tr>
<td>b=rand, b=single(rand), b=rand(n,n), b=single(rand(n,n))</td>
<td>1691</td>
<td>1623</td>
</tr>
<tr>
<td>c=a</td>
<td>2414</td>
<td>1977</td>
</tr>
<tr>
<td>c=a-b</td>
<td>2401</td>
<td>1961</td>
</tr>
<tr>
<td>c=a*b, c=a.*b</td>
<td>2063</td>
<td>1648</td>
</tr>
<tr>
<td>c=a/b, c=a./b</td>
<td>2740</td>
<td>2321</td>
</tr>
<tr>
<td>c=abs(a)</td>
<td>2010</td>
<td>1990</td>
</tr>
<tr>
<td>c=sin(a)</td>
<td>2168</td>
<td>2117</td>
</tr>
<tr>
<td>c=cos(a)</td>
<td>2138</td>
<td>1716</td>
</tr>
<tr>
<td>c=tan(a)</td>
<td>2171</td>
<td>1743</td>
</tr>
<tr>
<td>c=exp(a)</td>
<td>2273</td>
<td>2226</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Cost Relative to a=c</th>
<th>Matrix Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single</td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>Scalar</td>
<td>Double</td>
</tr>
<tr>
<td>a=rand, a=single(rand), b=rand(n,n), a=single(rand(n,n))</td>
<td>1.276</td>
<td>0.9382</td>
</tr>
<tr>
<td>b=rand, b=single(rand), b=rand(n,n), b=single(rand(n,n))</td>
<td>1.237</td>
<td>0.9798</td>
</tr>
<tr>
<td>c=a</td>
<td>1.000</td>
<td>1.0000</td>
</tr>
<tr>
<td>c=-a</td>
<td>1.210</td>
<td>0.9789</td>
</tr>
<tr>
<td>c=a+b</td>
<td>1.428</td>
<td>1.196</td>
</tr>
<tr>
<td>c=a-b</td>
<td>1.423</td>
<td>1.193</td>
</tr>
<tr>
<td>c=a.*b, c=a.*b</td>
<td>3.420</td>
<td>1.186</td>
</tr>
<tr>
<td>c=a./b, c=a./b</td>
<td>1.451</td>
<td>1.227</td>
</tr>
<tr>
<td>c=abs(a)</td>
<td>1.220</td>
<td>0.9971</td>
</tr>
<tr>
<td>c=a.*2, c=a.*2</td>
<td>1.237</td>
<td>1.006</td>
</tr>
<tr>
<td>c=a.*a, c=a.*a</td>
<td>1.620</td>
<td>1.404</td>
</tr>
<tr>
<td>c=a.*2, c=a.*2</td>
<td>1.830</td>
<td>1.608</td>
</tr>
<tr>
<td>c=a.*3, c=a.*3</td>
<td>1.547</td>
<td>1.325</td>
</tr>
<tr>
<td>c=a.*5, c=a.*5</td>
<td>3.151</td>
<td>3.130</td>
</tr>
<tr>
<td>c=sqrt(a)</td>
<td>1.282</td>
<td>1.281</td>
</tr>
<tr>
<td>c=sin(a)</td>
<td>1.280</td>
<td>1.068</td>
</tr>
<tr>
<td>c=cos(a)</td>
<td>1.264</td>
<td>1.038</td>
</tr>
<tr>
<td>c=tan(a)</td>
<td>1.279</td>
<td>1.057</td>
</tr>
<tr>
<td>c=exp(a)</td>
<td>1.284</td>
<td>1.055</td>
</tr>
<tr>
<td>c=exp(-a)</td>
<td>1.484</td>
<td>1.268</td>
</tr>
<tr>
<td>c=log(a)</td>
<td>1.344</td>
<td>1.347</td>
</tr>
</tbody>
</table>
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” 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\footnote{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.}.

- Another surprise: we have two calls to rand and the 2\textsuperscript{nd} 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 2\textsuperscript{nd} 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 \{×\} seems to be faster than both \{+,−\}, while this is not true (in the table we see that \{×\} 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 \{+,−,×,÷\} take the same time for scalars. For huge matrixes \{+,-,×\} cost the same, while \{÷\} cost about 4.4 times.
  - The square operator \{\(x^2\)\} is a special function and is slightly (15\%) faster than \{\(x×x\)\}.
  - 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 \{\(×\)\} if possible. For example, replacing \{\(x^3\)\} with \{\(x^2×x\)\} 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 \(c=\text{abs}(a)\).
  - 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 \{+,-,×\}.
  - 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.
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:

$$S_\pm(n+1) = (n+1) \times S_\pm(n) + n$$
$$M_\times(n+1) = (n+1) \times M_\times(n) + n + 1$$

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:

$$M_\times(n) = n! + \sum_{i=3}^{n} \prod_{j=n-i+3}^{j=n} j i \geq 3$$
$$M_\times(6) = 6! + (6 \times 5 \times 4 \times 3 + 6 \times 5 \times 4 + 6 \times 5 + 6) \quad n \geq 3$$

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

$$M_\times(n) = n! + \sum_{i=3}^{n} \prod_{j=n-i+3}^{j=n} j = n! (1 + \sum_{i=2}^{n-1} \prod_{j=i+1}^{j=n} 1/j i \geq 3)$$
$$M_\times(6) = n! (1 + \sum_{i=2}^{n-1} 1/i!) = n! \sum_{i=1}^{n-1} 1/i! \quad n \geq 1$$

remember $e^x$ series $\lim_{n \to \infty} \sum_{i=0}^{n-1} 1/i! = e - 1$

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

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.

\[
M_x(n) = \frac{n^3}{3} + O(n^2) + n
\]

\[
S_x(n) = \frac{n^3}{3} + O(n)
\]

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 \( \det(B) = c \cdot \det(A) \).

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:

```matlab
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
```

39
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
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.

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m = -\frac{\text{Ab}(i,i)}{\text{Ab}(j,i)}; ) for ( k=1:n+1 ) ( \text{Ab}(j,k) = m \text{Ab}(j,k) + \text{Ab}(i,k); )</td>
<td>( \text{if} \ \text{Ab}(j,j) \neq 0 ) ( m = -\frac{\text{Ab}(j,i)}{\text{Ab}(i,i)}; ) for ( k=i+1:n+1 ) ( \text{Ab}(j,k) = \text{Ab}(j,k) + m \text{Ab}(i,k); )</td>
</tr>
<tr>
<td>( \text{Ab}(j,k) = \text{Ab}(j,k) + m \text{Ab}(i,k); ) end</td>
<td>end</td>
</tr>
<tr>
<td>( \text{Ab}(j,i) = 0 ) end</td>
<td>end</td>
</tr>
<tr>
<td>( \text{x} = \text{zeros}(n,1); ) for ( i=n:-1:1 ) ( \text{sum} = \text{Ab}(i,n+1); ) for ( j=i+1:n ) ( \text{sum} = \text{sum} - x(j) \text{Ab}(i,j); ) end</td>
<td>( \text{Ab} = [\text{A b}]; ) ( \text{n} = \text{length}(\text{A}); ) for ( i=n:-1:1 ) ( \text{sum} = \text{Ab}(i,n+1); ) if ( i &lt; n ) ( \text{sum} = \text{sum} - \text{Ab}(i,i+1:n) * x(i+1:end); ) end</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Modified (Still have to add pivoting in case ( \text{Ab}(i,i) ) becomes zero)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Ab} = [\text{A b}]; ) ( \text{n} = \text{length}(\text{A}); ) for ( i=1:n-1 ) ( \text{for} \ j=i+1:n ) ( \text{m} = -\frac{\text{Ab}(i,i)}{\text{Ab}(j,j)}; ) ( \text{for} \ k=1:n+1 ) ( \text{Ab}(j,k) = \text{m Ab}(j,k) + \text{Ab}(i,k); ) end</td>
<td>( \text{Ab} = [\text{A b}]; ) ( \text{n} = \text{length}(\text{A}); ) for ( i=1:n-1 ) ( \text{for} \ j=i+1:n ) ( \text{m} = -\frac{\text{Ab}(j,i)}{\text{Ab}(i,i)}; ) ( \text{for} \ k=i+1:n+1 ) ( \text{Ab}(j,k) = \text{Ab}(j,k) + \text{m Ab}(i,k); ) end</td>
</tr>
<tr>
<td>( \text{x} = \text{zeros}(n,1); ) for ( i=1:n-1 ) ( \text{sum} = \text{Ab}(i,n+1); ) for ( j=i+1:n ) ( \text{sum} = \text{sum} - x(j) \text{Ab}(i,j); ) end</td>
<td>( \text{x} = \text{zeros}(n,1); ) for ( i=n:-1:1 ) ( \text{sum} = \text{Ab}(i,n+1); ) if ( i &lt; n ) ( \text{sum} = \text{sum} - \text{Ab}(i,i+1:n) * x(i+1:end); ) end</td>
</tr>
</tbody>
</table>

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

\[
\text{Ab} = [\text{A b}];
\text{n} = \text{length(\text{A});}
\text{for} \ i=1:n-1
\text{Scaled_Coeff} = \text{\text{Ab}(i:n,i:)} / \max(\max(\text{\text{Ab}(i:n,i:)}),[,2], \text{abs(\text{\text{min(\text{\text{Ab}(i:n,i:)}},[,2]))})
\text{[pivot, index]} = \max(\text{Scaled_Coeff});
\]
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
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