Updated on January 12, 2021

Math 3607: Homework 1

Due: 11:59PM, Tuesday, January 19, 2021

Do the following problems from the textbook. The notation *Problem 2.1–6* indicates Problem 6 at the end of Section 2.1.

1. Do Problem 2.1–6
2. Do Problem 2.1–8: From part (a), do only (i), (iii), and (vi). Then do (b) and (c)
3. Do Problem 2.1–14(a)
4. Write a script which asks for a temperature in Fahrenheit, converts the temperature into Celsius, and prints it. *(Modified from Problem 2.2–5)*
5. An *oblate spheroid* such as the Earth is obtained by revolving an ellipse about its minor axis as shown in the figure. The Earth's equatorial radius is about 20km longer than its polar

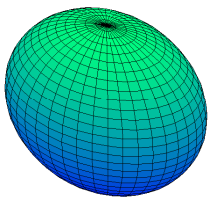


Figure 1: An oblate spheroid. Image created using MATLAB.

radius. The surface area of an oblate spheroid is given by

$$A(r_1, r_2) = 2\pi \left(r_1^2 + \frac{r_2^2}{\sin(\gamma)} \log\left(\frac{\cos(\gamma)}{1 - \sin(\gamma)} \right) \right),$$

where r_1 is the equatorial radius, r_2 is the polar radius, and

$$\gamma = \arccos\left(\frac{r_2}{r_1} \right).$$

We assume $r_2 < r_1$. Write a script that inputs the equatorial and polar radii and displays both $A(r_1, r_2)$ and the approximation $4\pi((r_1 + r_2)/2)^2$. Apply the script to the Earth data $(r_1, r_2) = (6378.137, 6356.752)$. Use `format long g` to display enough digits.

TOTAL: 20 points

Homework 1 (Solution)

Math 3607

Tae Eun Kim

Table of Contents


```
format long g
```

Problem 1 (2.1--6)

(a) One can use the fact that $\pi \text{ rad} = 180^\circ$ for conversion. For example, define conversion factors

```
deg_to_rad = pi/180;
rad_to_deg = 180/pi;
```

and then use them as

```
27.9 * deg_to_rad
```

```
ans =
         0.486946861306418
```

```
13 * rad_to_deg
```

```
ans =
           744.84513367007
```

Tip. MATLAB provides `deg2rad` and `rad2deg` functions.

```
deg2rad(27.9)
```

```
ans =
         0.486946861306418
```

```
rad2deg(13)
```

```
ans =
           744.84513367007
```

(b) Using `sind` function,

```
sind(27)
```

```
ans =
```

```
0.453990499739547
```

(c) Since the input is in radians, use `tan`:

```
tan(pi/5)
```

```
ans =
         0.726542528005361
```

(d)

```
1 / ( sind(20) + cosd(20) )
```

```
ans =
         0.780206008709879
```

(e) Note that the outputs of $\arcsin(1/3)$ and $\arctan(1/3)$ represent angles measured radians. If required, one can put them in degrees as shown above.

Radian outputs are computed by

```
asin(1/3)
```

```
ans =
         0.339836909454122
```

```
atan(1/3)
```

```
ans =
         0.321750554396642
```

while degree outs are obtained by `-d` variants of the corresponding functions

```
asind(1/3)
```

```
ans =
          19.4712206344907
```

```
atand(1/3)
```

```
ans =
           18.434948822922
```

One can quickly confirm that it is indeed correct by

```
asin(1/3) - deg2rad( asind(1/3) )
```

```
ans =
     0
```

```
atan(1/3) - deg2rad( atand(1/3) )
```

```
ans =
     0
```

(f) By the Pythagorean theorem, we know that the length of the base of a right triangle is given by

$$(\text{base}) = \sqrt{(\text{hyp})^2 - (\text{hgt})^2}$$

given the lengths of hypothenuse and height. In this problem, they are 145 and 144, respectively, and so the length of the base is

```
sqrt( 145^2 - 144^2 )
```

```
ans =
    17
```

Problem 2 (2.1--8)

(a) I will just type up my solutions instead of hand-writing.

(i) Taking the natural log of both sides and simplifying, we obtain $x = 3\ln(51)/\ln(5)$. Keep in mind that the natural logarithmic function is calculated by `log` in MATLAB. No function is named as `ln`.

```
x = 3 * log(51) / log(5)
```

```
x =
          7.32894186662572
```

(ii) Let $\theta = x + 2$ and $\theta_0 = \arcsin(-0.99) \in [-\pi/2, \pi/2]$.

```
theta0 = asin(-0.99)
```

```
theta0 =
         -1.42925685347047
```

It is clear that θ_0 is *a* solution of $\sin(\theta) = -0.99$. Using the symmetry of the sine curve about $\theta = \pi/2$, we deduce that $\theta_1 = \pi/2 + (\pi/2 - \theta_0) = \pi - \theta_0$ is another solution.

```
theta1 = pi - theta0
```

```
theta1 =
          4.57084950706026
```

θ_0 and θ_1 are all the roots within the 2π-periodic interval $[-\pi/2, 3\pi/2]$. Confirm:

```
sin(theta0)
```

```
ans =
                   -0.99
```

```
sin(theta1)
```

```
ans =
                    -0.99
```

Finally, using the 2π-periodicity of the sine function, we conclude that all solutions are written as

$$\theta = \theta_0 + 2\pi m \text{ or } \theta = \theta_1 + 2\pi n \text{ where } m, n \text{ are integers;}$$

or

$$\theta = \theta_0 + 2m\pi \text{ or } \theta = -\theta_0 + (2m+1)\pi \text{ where } m, n \text{ are integers.}$$

(iii) The (real) root of the given cubic equation is given by $x_0 = \sqrt[3]{\pi^2 - 5}$.

```
x0 = nthroot(pi^2 - 5, 3)  % or  x = (pi^2 - 5)^(1/3)
```

```
x0 =
         1.69497993102987
```

Advanced Note. The equation has two additional roots $z = x_0 e^{\pm 2\pi i/3}$ which are complex.

```
% complex roots
z1 = x0 * exp(2i*pi/3)
```

```
z1 =
        -0.847489965514937 +      1.46789567917667i
```

```
z2 = x0 * exp(-2i*pi/3)
```

```
z2 =
        -0.847489965514937 -      1.46789567917667i
```

```
% confirm that they solve the equation
z1^3 + 5 - pi^2
```

```
ans =
                        0 -   3.5527136788005e-15i
```

```
z2^3 + 5 - pi^2
```

```
ans =
                        0 +   3.5527136788005e-15i
```

We were asked to solve only for real *x*, so the calculation of `x0` is sufficient, but I still decided to write this down for any interested or advanced audience.

(iv) High school algebra exercise yields $x = (1/\sin(20^\circ) - 1)^2$:

```
( 1/sind(20) - 1 )^2
```

```
ans =
```

```
3.70102337008686
```

(vi) Let $t = x^2$. Then the given equation become $t^2 + t - 5 = 0$ whose roots are $t = (-1 \pm \sqrt{21})/2$. Since $t = x^2$, we obtain that

$$x = \pm\sqrt{(-1 + \sqrt{21})/2} \text{ (real) and } x = \pm i\sqrt{(1 + \sqrt{21})/2} \text{ (purely imaginary).}$$

Since we are only interested in real roots, we take the first two.

```
t0 = (-1 + sqrt(21))/2;
x1 = sqrt(t0)
```

```
x1 =
         1.33839002068826
```

```
x2 = -x1
```

```
x2 =
        -1.33839002068826
```

```
%% complex roots
% t1 = (-1 - sqrt(21))/2;
% x3 = sqrt(t1)
% x4 = -x3
```

Problem 3 (2.1--14)

The pendulum of a clock is supposed to make a single swing in exactly 1 second, *i.e.*, its period T is 1 second.

```
T = 1;
```

In a day, the pendulum makes 86,400 swings, because there are that many seconds in 24 hours:

$$24\,\text{hr} \times \frac{60\,\text{min}}{1\,\text{hr}} \times \frac{60\,\text{sec}}{1\,\text{min}} = 86,400\,\text{sec.}$$

So a clock running 5 minutes fast per day swings 86,400 times in 24 hr - 5 min = 86,100 seconds, which in turn implies that its period T' is smaller than 1

$$T' = \frac{86,100}{86,400}.$$

```
Tprime = 86100/86400;
```

Denoting the length of the correct pendulum by L and the incorrect one by L', we can express their periods using Equation (15.25) as

$$T = 2\pi\sqrt{\frac{L}{g}}\left(1+\frac{1}{16}\theta_0^2+\cdots\right) \text{ and } T' = 2\pi\sqrt{\frac{L'}{g}}\left(1+\frac{1}{16}\theta_0^2+\cdots\right).$$

Note that the maximum angle of oscillation is θ_0 for both cases. The problem requires that only the length of the pendulum can be changed.

Taking the ratio of the two equations and cancelling terms, we find that

$$\frac{T}{T'} = \sqrt{\frac{L}{L'}}, \text{ which implies that } L = L'\left(\frac{T}{T'}\right)^2.$$

That is, the length of the pendulum arm needs to by lengthened by $(T/T')^2$:

```
(T/Tprime)^2
```

```
ans =
          1.00698078160473
```

Problem 4. (Temperature Conversion)

Let F and C denote temperatures in (degree) Fahrenheit and Celsius, respectively. It is known that

$$C = \frac{5}{9}(F-32).$$

(The factor of $5/9 = 100/180$ is due to the fact that the temperature between the freezing point and the boiling point of water is divded into 180 degrees in Farrenheit and 100 degrees in Celsius; the subtraction of 32 is to match the representations for the freezing point -- $32^\circ F = 0^\circ C$. The two are linearly related.)

I will present the script as a non-executable code block.

```
F = input('Degrees in Fahrenheit: ');
C = (5/9)*(F-32);
fprintf('Fahrenheit: %6.2f\n', F)
fprintf('Celsius:    %6.2f\n', C)
```

Problem 5. (Oblate Spheroid)

I will present the requested script as a single code block, so that the document is self-contained. Instead of using `input` function, I will directly provide the geo-data of the Earth.

```
% r1 = input('Enter equatorial radius (r1): ');
% r2 = input('Enter polar radius (r2 < r1): ');
r1 = 6378.137;
r2 = 6356.752;
gamma = acos(r2/r1);
A_exact = 2*pi*( r1^2 + ...
```

```
    r2^2/sin(gamma) * log( cos(gamma)/(1-sin(gamma)) ));
A_approx = 4*pi*( (r1+r2)/2 )^2;
disp('The surface area of the given spheroid:')
```

```
The surface area of the given spheroid:
```

```
disp(['Exact:  ', num2str(A_exact)])
```

```
Exact:  510065604.9442
```

```
disp(['Approx: ', num2str(A_approx)])
```

```
Approx: 509495321.6397
```

Remarks.

1. Note that the line where `A_exact` is calculated is continued to the next line by using `...`. This is useful when you want to type in a lengthy code over multiple lines for enhanced readability.
2. When `disp` or `fprintf` is called within a Live Script, it prints out outputs and disrupts the code block. I personally find this annoying and would prefer all outputs to be printed after the end of the gray box. A workaround, for now, is to write an external script and call it in here as I demonstrated in class. Soon, we will learn about MATLAB *functions* and they can help us fix this nuiance to some degree.

Updated on January 20, 2021

Math 3607: Homework 2

Due: 11:59PM, Monday, January 25, 2021

1. A year is a *leap year* if it is a multiple of 4, except for years divisible by 100 but not by 400. In simpler terms, a non-century year is a leap year if it is divisible by 4; a century year is a leap year if it is divisible by 400.

   For example,

   - Last year (2020) was a leap year. (non-century year; divisible by 4)
   - 1900 was not a leap year. (century year; not divisible by 400)
   - 2000 was a leap year. (century year; divisible by 400)

   Write a script which determines whether a given year is a leap year or not.

2. Recall that Cartesian coordinates (x, y, z) in $\mathbb{R}^3$ are related to spherical coordinates (ρ, ϕ, θ) by

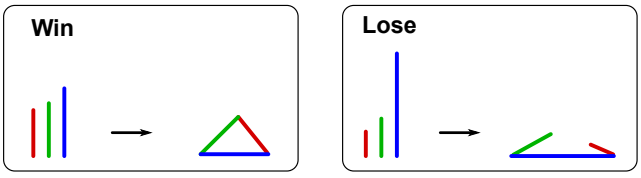
   $$x = \rho \sin\phi \cos\theta, \quad y = \rho \sin\phi \sin\theta, \quad z = \rho \cos\phi,$$

   where $\phi \in [0, \pi]$ and $\theta \in [0, 2\pi)$. Write a script which takes Cartesian coordinates as inputs and converts them to spherical ones.

3. Do **LM** [1] 5.4–1(b,d,f,h,j).

   Use *Code Example* in Live Script to format answers as non-executable code. Follow the instruction found in **Note** below the problem.

4. (Exercise 2, Lecture 4) In the game of 3-Stick, you pick three sticks each having a random length between 0 and 1. You win if you can form a triangle using three sticks; otherwise, you lose.

   Write a script simulating one million games and estimating the probability of winning a game.

[1] Reference Keys:

- **LM**: *Learning MATLAB, Problem Solving, and Numerical Analysis Through Examples* (Overman)
- **NCM**: *Numerical Computing with MATLB* (Moler)
- **FNC**: *Fundamentals of Numerical Computation* (Driscoll and Braun)

5. Each of the following sequences converges to π:

$$a_n = \frac{6}{\sqrt{3}} \sum_{k=0}^{n} \frac{(-1)^k}{3^k(2k+1)},$$
$$b_n = 16 \sum_{k=0}^{n} \frac{(-1)^k}{5^{2k+1}(2k+1)} - 4 \sum_{k=0}^{n} \frac{(-1)^k}{239^{2k+1}(2k+1)}.$$

Write a single script that prints $a_0, \ldots, a_{n_a}$, where n_a is the smallest integer so that $|a_{n_a} - \pi| \leq 10^{-6}$ and prints $b_0, \ldots, b_{n_b}$, where n_b is the smallest integer so that $|b_{n_b} - \pi| \leq 10^{-6}$.

Homework 2 (Solution)

Math 3607

Tae Eun Kim

Table of Contents


```
format long g
```

Problem 1 (Leap Year)

Solution 1

Consider the pseudocode provided in Lecture 3:

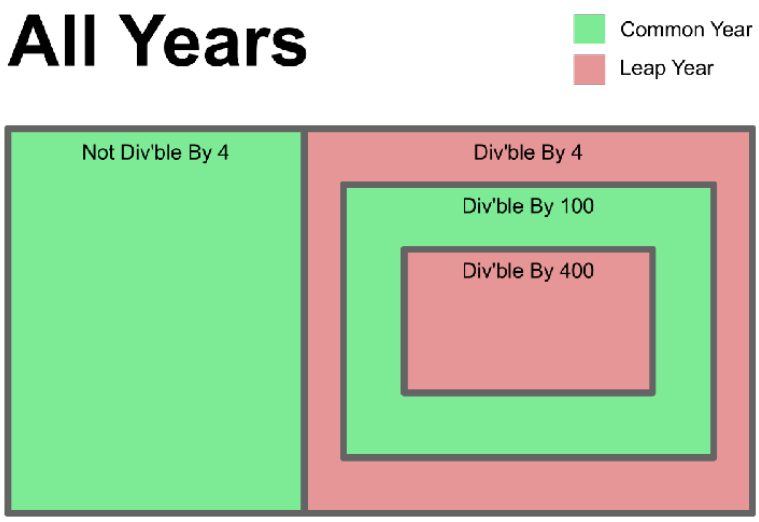
```
if [YEAR] is not divisible by 4
    it is a common year
elseif [YEAR] is not divisible by 100
    it is a leap year
elseif [YEAR] is not divisible by 400
    it is a common year
else
    it is a leap year
end
```

Below is a direct translation of the pseudocode:

```
year = 2021; % this year is not a leap year.
% or  year = input('Enter year: ');
msgCommon = 'Given year is a common year.';
msgLeap = 'Given year is a leap year.';

if mod(year, 4) ~= 0
    disp(msgCommon)
elseif mod(year, 100) ~= 0
    disp(msgLeap)
elseif mod(year, 400) ~= 0
    disp(msgCommon)
else
    disp(msgLeap)
end
```

```
Given year is a common year.
```

Analysis. The key reason why the code above work is due to the fact that branches of `if`-statement are visited sequentially from top to bottom and only the first branch whose condition evaluates to "true" is executed. One of the best ways to leverage this mechanism is to gradually narrow down the scope of the conditions if possible. See the venn diagram below which schematically partitions all years into common years (light green) and leap years (light red) according to their numerical characterization.



Solution 2

Alternately, we can utilize the alternate description of leap years given in the homework prompt, that is,

"In simpler terms, a non-century year is a leap year if it is divisible by 4; a century year is a leap year if it is divisible by 400."

and write

```
year = 2021;
if mod(year, 100) ~= 0                  % non-century year
    if mod(year, 4) == 0                % rule for leap year
        disp(msgLeap)
    else
        disp(msgCommon)
    end
else                                    % century year
    if mod(year, 400) == 0              % rule for leap year
        disp(msgLeap)
    else
        disp(msgCommon)
    end
end
```

```
Given year is a common year.
```

In words, this is an example in which different rules are used for century years and non-century years.

Solution 3

You could piece together multiple conditions characterizing leap years using logical operators.

```
year = 2021;
if mod(year,4)~=0 || ( mod(year,100)==0 && mod(year,400)~=0 )
    disp(msgCommon)
else
    disp(msgLeap)
end
```

```
Given year is a common year.
```

Problem 2 (Coordinate Converter)

Given the relationship between the sets of coordinates, one can derive (or recall from calculus) that

$$\begin{aligned} \rho &= \sqrt{x^2+y^2+z^2} \\ \tan\theta &= \frac{y}{x} \\ \cos\phi &= \frac{z}{\sqrt{x^2+y^2+z^2}} \end{aligned}$$

Note that the last equation, when inverted, yields

$$\phi = \arccos\frac{z}{\sqrt{x^2+y^2+z^2}} = \arccos\frac{z}{\rho},$$

and since the range of $\arccos$ is precisely $[0,\pi]$, it can be easily coded using the built-in `acos` function. However, since we are require to find $\theta \in [0,2\pi)$, adjustments are called for. Below is the breakdown of possible scenarios according to the quadrants to which (x,y) belongs. Suppose for now that not both x and y are zero.

- [Q1] If $x \geq 0$ and $y \geq 0$, then $\theta = \arctan(y/x)$.
- [Q2] If $x < 0$ and $y > 0$, then $\theta = \arctan(y/x) + \pi$.
- [Q3] If $x < 0$ and $y \leq 0$, then $\theta = \arctan(y/x) + \pi$.
- [Q4] If $x \geq 0$ and $y < 0$, then $\theta = \arctan(y/x) + 2\pi$.

In the edge case of $x = y = 0$, that is, if the point (x,y,z) lies on the z-axis:

- $\rho = \sqrt{x^2+y^2+z^2} = |z|$;
- $\phi = 0$ or π depending on the sign of z;
- θ can be any number.

We will handle this special case by printing out a separate message.

Let's code it up.

```
x = -1; y = 0; z = 1;
% x = input('Enter x: ');
% y = input('Enter y: ');
% z = input('Enter z: ');
rho = sqrt(x^2 + y^2 + z^2);
phi = acos(z/rho);
if x < 0
    theta = atan(y/x) + pi;
else
    if y >= 0
        theta = atan(y/x);
    else
        theta = atan(y/x) + 2*pi;
    end
end
disp(' ')
```

```
fprintf('rho:    %10.4f\n', rho)
```

```
rho:       1.4142
```

```
fprintf('phi:    %10.4f\n', phi)
```

```
phi:       0.7854
```

```
if x == 0 && y == 0
    fprintf('The point is on the z-axis and theta can take any value.')
else
    fprintf('theta:  %10.4f\n', theta)
end
```

```
theta:     3.1416
```

Problem 3 (LM 5.4--1)

(b) As instructed in the note to the problem, you need not put anything inside the loop. Besides, this is where "*Code Example*" environment comes in handy.

```
for j = 1:1000
  ....
end
```

Instructors' rant. Each semester, I always find some students who make their response runnable and prints out all thousand lines in the exported pdf. Please don't.

(d) In the lecture on the conditional statements, we briefly encounter a family of functions whose names begin with `is*`. One of them was `isprime`:

```
isprime(3)
```

```
ans = logical

   1
```

```
isprime(4)
```

```
ans = logical

   0
```

We can use it in conjunction with `find` to list prime numbers within a given range. For example, we can find all primes that are ≤ 100 in one statement by

```
find(isprime(1:100))
```

```
ans = 1×25
     2     3     5     7    11    13    17    19    23    29    31    37    41 ···
```

A slightly more verbose version is

```
find(isprime(1:100)==1)
```

```
ans = 1×25
     2     3     5     7    11    13    17    19    23    29    31    37    41 ···
```

Therefore, we can write

Answer.

```
for j = find(isprime(1:100))
  .....
end
```

Question. How would you modify this code so that you execute a `for` loop where the loop index *j* is 1 through 1000, but not including prime numbers?

Note 1. Finding all primes between two arbitrary integers requires more than one line, but not more than two. For example, to find all prime numbers between 10 and 20:

```
n = 10:20;
n( find(isprime(n)) )
```

```
ans = 1×4
    11    13    17    19
```

The inner expression `find(isprime(n))` yields the indices of elements of `n` that are prime numbers. We feed it back to `n` to extract them. Even though it works without an issue, MATLAB highlights and underlines

"find", suggesting that we use logical indexing in lieu of FIND for potential improvement of performance. Following the suggestion, we can trim it off to

```
n = 10:20;
n( isprime(n) )
```

```
ans = 1×4
    11    13    17    19
```

Note 2. MATLAB has a handy function that prints out all prime numbers no greater than a given number; it is called `primes`.

```
primes(10)
```

```
ans = 1×4
     2     3     5     7
```

Thus we can write down the answer even more succintly as

```
for j = primes(1000)
  .....
end
```

(f) Recall the syntax of `for` loop:

```
for j = <vector>
  <statements>
end
```

So something like the following would do:

```
for j = 2.^[0:20]
  .....
end
```

(h)

```
while x >= 0
  .....
end
```

(j)

```
while x ~= NaN
  .....
end
```

Problem 4 (Game of 3-Stick)

We can use `rand` to simulate picking three sticks with random lengths between 0 and 1.

```
a = rand();
b = rand();
c = rand();
```

For this triple to be sides of a triangle, the length of **any** one must not exceed the sum of the other two, that is, all of the following must be true:

- `a < b + c`
- `b < a + c`
- `c < a + b`

Let's simulate a single game.

```
a = rand();
b = rand();
c = rand();
if (a<b+c) && (b<a+c) && (c<a+b)
    disp('You won!')
else
    disp('You lost.')
end
```

```
You lost.
```

To simulate multiple games, you will embed the code above within a `for`-loop. In addition, we will keep track of the number of games won rather than displaying win/loss messages.

```
nrWins = 0;
nrGames = 20;
for j = 1:nrGames
    a = rand(); b = rand(); c = rand();
    if (a<b+c) && (b<a+c) && (c<a+b)
        nrWins = nrWins + 1;
    end
end
nrWins
```

```
nrWins =
    14
```

```
nrGames
```

```
nrGames =
    20
```

Note the deletion of the else-block since we need not do anything in this context.

Final Version. Now that we know that the script above works with a modest number of games, let's increase it to one million as requested by the problem. In addition, let's format outputs nicely using `fprintf` or `disp`.

```
nrWins = 0;
nrGames = 1e6;   % one million games
```

```
for j = 1:nrGames
    a = rand();
    b = rand();
    c = rand();
    if (a<b+c) && (b<a+c) && (c<a+b)
        nrWins = nrWins + 1;
    end
end
pWin = nrWins/nrGames; % prob. of winning a game
fprintf('Games won:     %8d\n', nrWins)
```

```
Games won:       499668
```

```
fprintf('Games played:  %8d\n', nrGames)
```

```
Games played:   1000000
```

```
fprintf('Winning ratio: %8.4f\n', pWin)
```

```
Winning ratio:   0.4997
```

Version Using Arrays.

The problem was assigned when we did not know about arrays. Now that we do and since Exam 1 is fast approaching, let's try to work this out using arrays.

Step 1. Simulate multiple games using `rand` function.

```
format short
nrGames = 20;
sticks = rand(3, nrGames)
```

```
sticks = 3×20
    0.0498    0.6049    0.8851    0.5674    0.8530    0.0030    0.9439    0.5081 ···
    0.7871    0.4683    0.8552    0.5402    0.9213    0.8968    0.1868    0.0141
    0.1397    0.6207    0.8558    0.6149    0.8929    0.9925    0.7861    0.6315
```

Each column corresponds to a single game. The matrix `sticks` holds contains 5 games.

Step 2. Generate an auxiliary matrix incorporating the "triangle" conditions.

I will create an auxiliary matrix `A` (for analysis) whose *j*th columns contain the following information:

- `A(1,j) = -sticks(1,j) + sticks(2,j) + sticks(3,j)`
- `A(2,j) =  sticks(1,j) - sticks(2,j) + sticks(3,j)`
- `A(3,j) =  sticks(1,j) + sticks(2,j) - sticks(3,j)`

In this setup, the *j*th game is won if and only if all entries of the *j*th column of `A` are positive.

```
A = zeros(size(sticks));
A(1,:) = -sticks(1,:) + sticks(2,:) + sticks(3,:);
A(2,:) =  sticks(1,:) - sticks(2,:) + sticks(3,:);
A(3,:) =  sticks(1,:) + sticks(2,:) - sticks(3,:);
```

```
A
```

```
A = 3×20
    0.8770    0.4841    0.8260    0.5876    0.9611    1.8863    0.0290    0.1375 ···
   -0.5976    0.7573    0.8857    0.6421    0.8246    0.0988    1.5431    1.1255
    0.6973    0.4526    0.8844    0.4928    0.8815   -0.0927    0.3447   -0.1093
```

Sidenote. The matrix `A` can be constructed by a single statement

```
A = [-1 1 1; 1 -1 1; 1 1 -1] * sticks
```

This is a triumphant moment in which the understanding of matrix multiplication shines in progamming! (Make sure you understand why this works!)

Now, I will replace positive entries of `A` by 1 and non-positive (that is, less than or equal to zero) entries by 0.

```
A( find(A >0) ) = 1;
A( find(A<=0) ) = 0;
A
```

```
A = 3×20
     1     1     1     1     1     1     1     1     1     1     1     1     1 ···
     0     1     1     1     1     1     1     1     1     1     0     1     1
     1     1     1     1     1     0     1     0     1     1     1     1     1
```

Step 3. Identify the games won.

Finally, generate a row vector by sum all elements of `A` along columns:

```
sumA = sum(A)
```

```
sumA = 1×20
     2     3     3     3     3     2     3     2     3     3     2     3     3 ···
```

The elements that are equal to 3 correspond to the games won!

Full script (Vectorized).

Let's put all together.

```
nrGames = 1e6;
sticks = rand(3, nrGames);
A = [-1 1 1; 1 -1 1; 1 1 -1] * sticks;
A( find(A >0) ) = 1;
A( find(A<=0) ) = 0;
sumA = sum(A);
nrWins = length(find(sumA==3));
pWin = nrWins/nrGames; % prob. of winning a game
fprintf('Games won:     %8d\n', nrWins)
```

```
Games won:       499802
```

```
fprintf('Games played:  %8d\n', nrGames)
```

```
Games played:   1000000
```

```
fprintf('Winning ratio: %8.4f\n', pWin)
```

```
Winning ratio:   0.4998
```

Problem 5 (Convergence of Sequences)

We will take the value of `pi` stored in MATLAB as the true value of π. As a prep work, suppose that `n` is given. We will concentrate first on calculating a_n and b_n. Using a `for`-loop, we can form these terms as below:

```
format long g
n = 10;
a_n = 0; b_n = 0;
for k = 0:n
    a_n = a_n + 6/sqrt(3)*(-1/3)^k / (2*k+1);
    b_n = b_n + 16/5*(-1/25)^k / (2*k+1) - 4/239*(-1/239^2)^k / (2*k+1);
end
a_n
```

```
a_n =
         3.14159330450308
```

```
b_n
```

```
b_n =
         3.14159265358979
```

As the last lines show, with only $n = 10$, they are already quite close to π.

Now that the key part of the problem has been solve, let's design an outer loop which calculates terms $a_0, a_1, \ldots$ until it is close enough to π; the given *tolerance* is 10^{-6}.

```
tol = 1e-6;
nMax = 100;     % maximum number of iterations
n = 0;
% fprintf('%2s %16s %16s\n', 'n', 'a_n', '|a_n - pi|');
% fprintf('%36s', repmat('-', 1, 36))
while n == 0 || (errAbs > tol && n <= nMax)
    a = 0;
    for k = 0:n
        a = a + (-1/3)^k / (2*k+1);
    end
    a = 6/sqrt(3)*a;   % since 6/sqrt(3) is a common factor.
    errAbs = abs(a - pi);
    fprintf('%2d %16.8f %16.8e\n', n, a, errAbs)
    n = n + 1;
end
```

```
 0       3.46410162   3.22508962e-01
```

```
 1       3.07920144   6.23912179e-02
 2       3.15618147   1.45888180e-02
 3       3.13785289   3.73976199e-03
 4       3.14260475   1.01209207e-03
 5       3.14130879   2.83868127e-04
 6       3.14167431   8.16591090e-05
 7       3.14156872   2.39376480e-05
 8       3.14159977   7.12022171e-06
 9       3.14159051   2.14265171e-06
10       3.14159330   6.50913289e-07
```

Note. Before the beginning of the `while`-loop, the variable `errAbs` was not initialized and so the loop would not be initated if it were not for `n==0` preceding `(errAbs ... )`. An alternate method that is commonly used is to begin with

```
tol = 1e-6;
nMax = 100;
errAbs = Inf;
while errAbs > tol && n <= nMax
    ....
```

You handle b_n's in the same fashion.

```
tol = 1e-6;
nMax = 100;     % maximum number of iterations
n = 0;
% fprintf('%2s %16s %16s\n', 'n', 'b_n', '|b_n - pi|');
% fprintf('%36s', repmat('-', 1, 36))
while n == 0 || (errAbs > tol && n <= nMax)
    b1 = 0; b2 = 0;
    for k = 0:n
        b1 = b1 + (-1/25)^k/(2*k+1);
        b2 = b2 + (-1/239^2)^k/(2*k+1);
    end
    b = 16/5*b1 - 4/239*b2;
    errAbs = abs(b - pi);
    fprintf('%2d %16.8f %16.8e\n', n, b, errAbs)
    n = n + 1;
end
```

```
 0       3.18326360   4.16709447e-02
 1       3.14059703   9.95624264e-04
 2       3.14162103   2.83757352e-05
 3       3.14159177   8.81407615e-07
```

The two numerical experiments provide an evidence that the sequence b_n may converge to π much quicker than a_n.

Updated on January 27, 2021

Math 3607: Homework 3

Due: 11:59PM, Monday, February 1, 2021

1. (**Guess-The-Number**) Write the following game in which a user is to guess the integer randomly generated by the computer. In the program:

   - User inputs the lower and the upper bounds of the range.
   - The program generates a random integer within the specified range and stores it in a variable.
   - Use a `while`-loop for repeated guessing.
     - If the user guessed a number larger than the generated number, print out "Your guess is too high. Try again!".
     - If the user guessed a number smaller than the generated number, print out "Your guess is too low. Try again!".
     - If the user guessed the number correctly, print out "Congratulations!" and terminate the program.

   Below is an example run of the program.

```
>> guess
   Enter the lower bound: 1
   Enter the upper bound: 100
   Guess a number: 50
   Your guess is too low. Try again!
   Guess a number: 75
   Your guess is too low. Try again!
   Guess a number: 87
   Your guess is too high. Try again!
   Guess a number: 81
   Your guess is too low. Try again!
   Guess a number: 84
   Your guess is too high. Try again!
   Guess a number: 82
   Congratulations!
```

2. (**Gap of 10**, Lecture 5) Simulate the tossing of a *biased* coin whose tails is 3 times more likely to be showing than its heads, until the gap between the number of heads and that of tails reaches 10.

3. (**Construction of Arrays**) Do the following problems [1].

[1] Reference Keys:

- **LM**: *Learning MATLAB, Problem Solving, and Numerical Analysis Through Examples* (Overman)
- **NCM**: *Numerical Computing with MATLB* (Moler)
- **FNC**: *Fundamentals of Numerical Computation* (Driscoll and Braun)

- **LM** 3.1–3(b,c,e,g)
- **LM** 3.1–4(c,e)
- **LM** 3.1–5(d,f)

Please read the notes at the beginning of Section 3.1 exercises on p. 382. In addition, pay attention to **Note** found at the end of individual problems, if there is any.

4. Do **LM** 3.1–16.

5. Do **LM** 3.2–7.

   For this problem, assume that A is already stored in MATLAB and simply provide MATLAB statements generating B, C, and D; use of the *Code Examples* environment is recommended. (Obviously, in the development stage, you may define A so that you can check your work.)

6. (**Birthday Problem**, Lecture 7) In a group of n randomly chosen people, what is the probability that everyone has a different birthday?

   (a) Find this probability by hand.

   (b) Let $n = 30$. Write a script that generates a group of n people randomly and determines if there are any matches.

   (c) Modify the script above to run a number of simulations and numerically calculate the sought-after probability. Try $1000, 10000$, and 100000 simulations. Compare the result with the analytical calculation done in the previous part.

Homework 3 (Solution)

Math 3607

Tae Eun Kim

Table of Contents

Problem 1 (Guess-The-Number)

Below is an example solution. Note the use of the infinite loop and the `break` command in one of the `if` branches.

```
nLow = input('Enter the lower bound: ');
nHigh = input('Enter the upper bound: ');
theNumber = randi([nLow nHigh]);
while true
    nGuess = input('Guess a number: ');
    if nGuess > theNumber
        disp('Your guess is too high. Try again!')
    elseif nGuess < theNumber
        disp('Your guess is too low. Try again!')
    else
        disp('Congratulations!')
        break
    end
end
```

```
Your guess is too low. Try again!
Your guess is too low. Try again!
Your guess is too low. Try again!
Your guess is too low. Try again!
Your guess is too low. Try again!
Your guess is too low. Try again!
Congratulations!
```

Problem 2 (Gap of 10)

The key point of this problem lies in simulating the tossing of a biased coin with the following probability profile: $P(H) = 1/4$ (heads) and $P(T) = 3/4$ (tails). Let's use 0 for tails and 1 for heads. I will show couple ways.

Using `randi`:

```
toss = randi([0 3]);
if toss > 0
    toss = 1;
end
toss
```

```
toss = 1
```

Using `rand`:

```
toss = floor( rand() + 0.25 )
```

```
toss = 0
```

Why does it work? The expression `rand() + 0.25` generates a number in $(0.25, 1.25)$ with equal likelihood. So numbers in the interval $(0.25, 1)$ are three times more likely to appear than those from $[1, 1.25)$. The `floor` function maps numbers in $(0.25, 1)$ to 0 and those in $[1, 1.25)$ to 1, successfully simulating the tossing of a bias coin with the given description.

It is not known in advance how many tosses are needed until the gap reached 10. So use a `while`-loop:

```
nTosses = 0;  % number of tosses
nHeads = 0;   % number of heads
nTails = 0;   % number of tails
while abs(nHeads-nTails) < 10
    toss = floor( rand() + 0.25 );
    if toss == 0
        nTails = nTails + 1;
    else
        nHeads = nHeads + 1;
    end
    nTosses = nTosses + 1;
end
fprintf('The gap reached 10 with %d tosses.\n', nTosses)
```

```
The gap reached 10 with 16 tosses.
```

Note. The code above suffices. However, if you want to repeat the simulation multiple times and to monitor how many tosses are needed till the gap reaches 10. You will need an additional (outer loop) to embrace the `while`-loop written above. For example:

```
nGames = 10; % number of games played
result = zeros(nGames, 1);
% initialize a vector to record the number of tosses
% to reach the gap of 10.

for j = 1:nGames
    nTosses = 0;  % number of tosses
    nHeads = 0;   % number of heads
```

```
    nTails = 0;   % number of tails
    while abs(nHeads-nTails) < 10
        toss = floor( rand() + 0.25 );
        if toss == 0
            nTails = nTails + 1;
        else
            nHeads = nHeads + 1;
        end
        nTosses = nTosses + 1;
    end
    %fprintf('The gap reached 10 with %d tosses.\n', nTosses)
    result(j) = nTosses;
end
```

To view how many tosses were needed for all games, simply type

```
result
```

```
result = 10×1
    16
    14
    18
    22
    48
    18
    12
    20
    16
    26
```

To find out the average number of tosses till the gap reaches 10, do

```
mean(result)
```

```
ans = 21
```

Problem 3 (Construction of Arrays)

LM 3.1--3

Following the suggestion of the textbook, I will set `n` to be a reasonable number and output results to see if my codes work properly.

```
n = 10;
```

Once I am confident of my codes, I will suppress the outputs using semicolons to keep the document concise. And you should also suppress results!

(b) Note that $b_k = (2k-1)^2$ for $k = 1, 2, \ldots, p$ where p is the number of elements of $\mathbf{b}$, which is to be determined. Since the elements are all required to be $\leq n^2$, it must be that p is the largest positive integer such that

$2p - 1 \le n$. This implies that $p = \lfloor (n+1)/2 \rfloor$ (floor function). However, as far as generating the vector on MATLAB is concerned, we can simply do

```
b = ([1:2:n].') .^ 2; % Note that b is a column vector. See below.
```

without needing to type p out explicitly. (Think about why and make sure you understand why.)

Convention. In this class, a vector, by default, is a column vector. If a row vector is desired, it will be explicitly mentioned.

Confirm. One can confirm that `a` constructed above indeed has the right number of elements:

```
length(b) == floor((n+1)/2)
```

```
ans = logical

   1
```

(c) In this part, $c_k = (2k-1)^2$ for $k = 1, 2, \ldots, q$ where q is the number of elements of $\mathbf{c}$. The requirement is that the elements are $\le n$ not n^2. Thus q is the largest integers such that $(2q-1)^2 \le n$, *i.e.,* $q = \lfloor (\sqrt{n}+1)/2 \rfloor$. Though the expression for q is messy, it can be compactly coded as

```
c = ([1:2:sqrt(n)].') .^ 2;
```

since the colon operator ensures that the last elements of `1:2:sqrt(n)` does not exceed $\sqrt{n}$.

Even though it is not required, it is fun to confirm that `c` has the right number of elements:

```
length(c) == floor((sqrt(n)+1)/2)
```

```
ans = logical

   1
```

(e) This is a simple one.

```
e = [2:n^2, 999999].';
```

It is important that you do not omit `.'` at the end since $\mathbf{e} = (2, 3, \ldots, n^2, 999999)^{\mathrm{T}}$, a column vector!

(g) All angles are in radians.

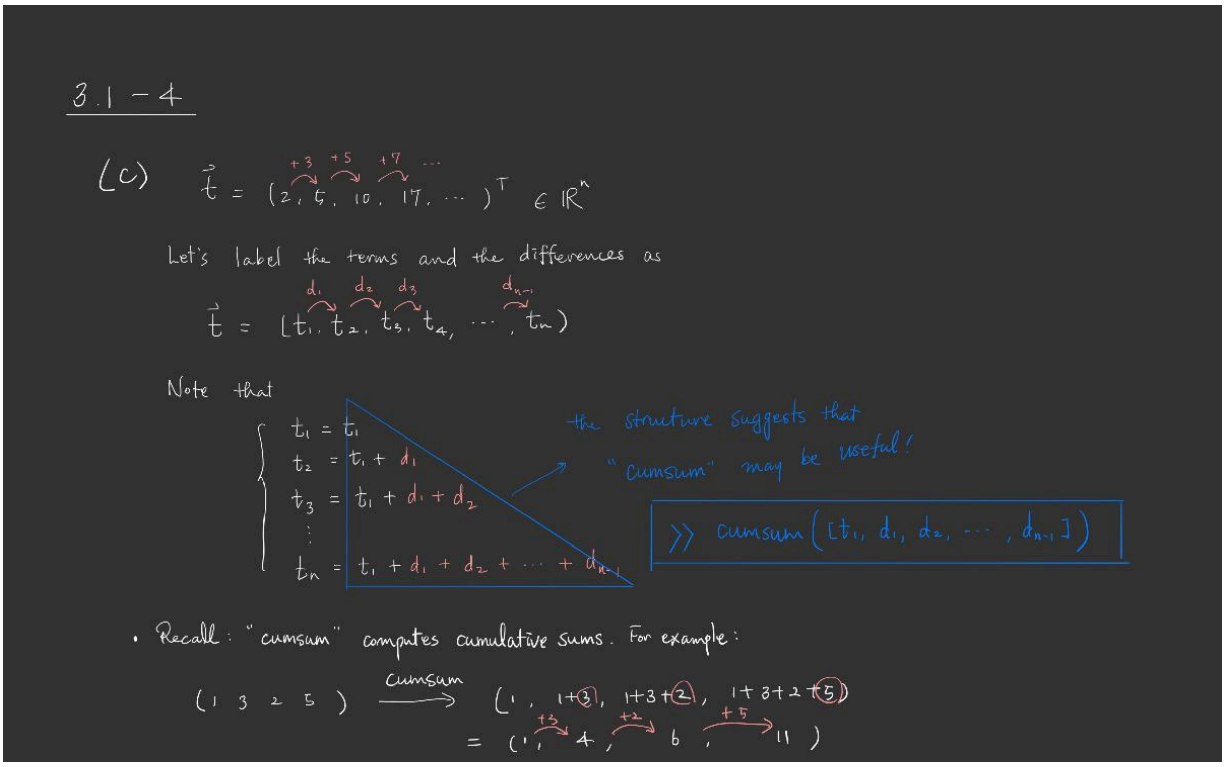
```
g = [sin(1:n), cos(n:-1:1)].'
```

```
g = 20×1
    0.8415
    0.9093
    0.1411
   -0.7568
   -0.9589
   -0.2794
    0.6570
    0.9894
    0.4121
   -0.5440
      ⋮
```

LM 3.1--4

In this solution, I will show you another trick to handle the "undetermined" n. Since we are only interested in your code and not in seeing the outputs, you can put in your answer within a Code Example block. To create one, go to **Insert** tab and click **Code Example**. Then type in your code in the box. What you write there will be formatted in a monospace font and will be syntax-highlighted, but it will not be executed. It is perfectly suitable for our purposes!

(c) I wrote by hand the following note for a student in an office hour.



(I apologize for "light on dark background" theme here, which is not very appropriate when exported to a pdf. I initially did not intend on attaching it here.) Accordingly, we can write

```
t = cumsum( [2, 3:2:2*n-1] ) .';  % or t = cumsum( [2, 3:2:2*n-1].' );
```

Note that `3:2:2*n-1` generates a row vector $(3, 5, 7, \ldots, 2n-1)$ which contains $n-1$ numbers, which ensures that `t` constructed as above will have exactly n elements.

Alternately, we can view it as follows:

- $t_1 = 1^2 + 1 = 2$
- $t_2 = 2^2 + 1 = 5$
- $t_3 = 3^2 + 1 = 10$
- $t_4 = 4^2 + 1 = 17$
- $\vdots$

In general, $t_j = j^2 + 1$, for $j \geq 1$. To see if this description agrees with the former viewpoint, note that

$$t_1 = 2 \text{ and } \Delta t_j := t_{j+1} - t_j = (j+1)^2 - j^2 = 2j + 1.$$

Confirmed! So we can generate $\mathbf{t}$ also by

```
t = ([1:n].^2 + 1)'
```

(e) Note that the elements of $\mathbf{v}$ are powers of 2:

$$2^{-1}, 2^0, 2^1, \ldots$$

To grab n such numbers starting from 2^{-1}, the last element must be 2^{n-1}. (Simply count the numbers in the list $-1, 0, 1, 2, \ldots, n-1$.) Thus, we can write

```
v = 2 .^ ([-1:n-2].');
```

LM 3.1--5

Pay attention to elementwise operations used below such as `./`, `.^`, and the use of elementary functions in conjunction with arrays.

(d)

```
d = ( 1 ./ sin(n:-1:1).^3 ).';
```

(f)

```
f = factorial(1:n+1).';
```

Problem 4 (LM 3.1--16)

(a)

```
s = A(1,:) + A(2,:)
```

(b) Since we are asked to calculate the inner product of two *row* vectors,

```
A(2,:) * A(3,:)'
```

(c) Since we are asked to calculate the outer product of two *column* vectors,

```
B = A(:,3) * A(:,7)'
```

(d) Since we are asked to calculate the outer product of a *row* vector and a *column* vector,

```
C = A(4,:)' * A(:,9)'
```

(e) The `diag` function can also take a rectangular matrix as an input:

```
d = diag(A)
```

(f) By way of vertical concatenation,

```
e = [diag(A); zeros(10,1)]
```

Problem 5 (LM 3.2--7)

As per the instruction found at the beginning of the exercises for LM 3.2, we answer this question without using loops.

(a) First make a copy of `A` and call it `B`. Then find where `B` is positive (`B` is currently identical to `A`) using `find` function and change the corresponding elements of `B` to zeros.

```
B = A;
B(find(B<0)) = 0;
```

(b) Similarly, we do

```
C = A^2 - 100;
C(find(C<0)) = 0;
```

One things to note is that A is a square matrix and the expression $A^2 - 100$ requires the kind of multiplication we do in a **linear algebra** course. It is **not** an elementwise operation!

(c) Let's use `max` function.

```
D = max( A-10, B-100 )
```

Problem 6 (Birthday Problem)

(a) Since we ignore a leap year, the probability p for the group of n people to have no match in birthdays is

$$p = \frac{365}{365} \cdot \frac{364}{365} \cdots \frac{365 - n + 1}{365} = \prod_{j=0}^{n-1} \frac{365 - j}{365}.$$

and so the probability q to have at least on birthday match is

$$q = 1 - p.$$

(b) For a single instance:

```
n = 30;
nrs =  randi(365, 1, n);
nrs = sort(nrs);
d_nrs = diff(nrs);
id_nrs = find(d_nrs == 0);
nr_matches = length(id_nrs)
```

```
nr_matches = 1
```

(c) For multiple simulations (I will just use 10000 for demo here):

```
%% simulations
nr_sims = 1e4;
Nrs = randi(365, n, nr_sims);
Nrs = sort(Nrs);
D_Nrs = diff(Nrs);

Match = zeros(size(D_Nrs));    % preallocate: Match = 0 for no match
iD_Nrs = find(D_Nrs == 0 );    %              Match = 1 for a match
Match(iD_Nrs) = 1;
matches = sum(Match);
i_matches = find(matches > 0);
ratio_num = length(i_matches)/nr_sims;
ratio_ext = 1 - prod((365 - (0:n-1))./365);
disp('--------------------------------------')
```

```
-----------------------------------
```

```
disp(['fraction of matches (numeric):  ', num2str(ratio_num, 4)])
```

```
fraction of matches (numeric):  0.7094
```

```
disp(['fraction of matches (analytic): ', num2str(ratio_ext, 4)])
```

```
fraction of matches (analytic): 0.7063
```

In this script, `ratio_num` is a numerical approximation of the probability q to have at least one match based on the result of simulations; `ratio_ext` is the exact probability computed in part (a). They appear to be in a reasonably good agreement.

Updated on February 9, 2021

Math 3607: Homework 4

Due: 11:59PM, Monday, February 15, 2021

1. (Handling large numbers and scientific notation; adapted from **LM** 5.6) A product of terms can grow or decay much faster than a sum of terms, leading to an *overflow* or an *uderflow* in a floating-point architecture. This difficulty can usually be avoided by replacing

$$P_n = \prod_{i=1}^{n} a_i \quad \text{by} \quad \log|P_n| = \sum_{i=1}^{n} \log|a_i| \qquad (\spadesuit)$$

as long as none of the terms are 0; if one or more terms are 0 the product is immediate. The result is then $P_n = f \times 10^m$ in (base-10) scientific notation where $|f| \in [1, 10)$ and f can be positive or negative, and where m is an integer.

Write a MATLAB function which calculates the product of all the elements of an input vector `a` by using ($\spadesuit$).

- The name of the function should be `logprod.m` and must output f and m.
- $f = 0$ if one of the elements of `a` is 0.
- The code must check each element to determine if it is positive, negative, or zero, and also keep track of the overall sign of the product.
- If a zero element is found, the function must exit immediately with $f = m = 0$.

2. (Tiling with spiral polygons; adapted from **LM** 6.8–34,35) Modify the function `spiralgon.m` you wrote for Exam 1 so that it now takes `n`, `m`, `d_angle`, `d_rot`, and `shift`. The additional input argument `shift` is a two-vector and it shifts the center of all the polygons to the point (`shift(1)`, `shift(2)`). Consequently, `plot` must now be changed to

```
plot(V(1,:)+shift(1), V(2,:)+shift(2), 'Color', C(i,:))
```

In addition, comment out the " `hold off` " statement before the `for` loop.

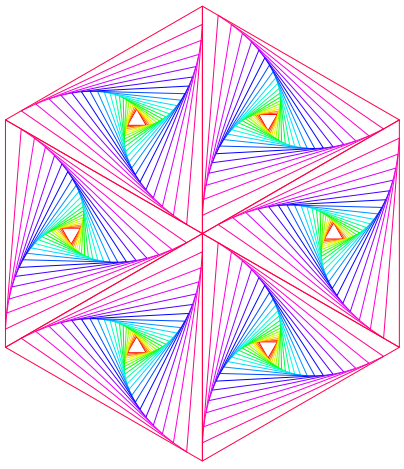


Figure 1: Tiling with spiral triangles.

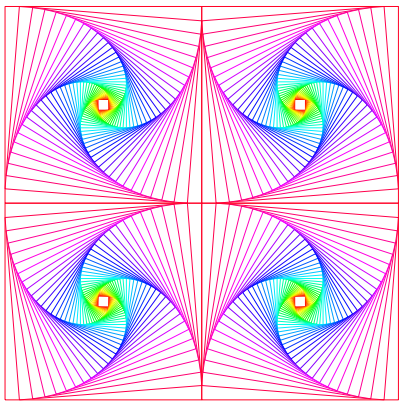


Figure 2: Tiling with spiral squares.

(a) Generate Figure 1.

- Run `spiralgon`, letting `n=3`, `m=21`, `d_angle=4.5`, `d_rot=90`, and `shift=[0 0]'` and save `V`. Its first column is the location of the leftmost vertex. Follow the call with " `hold off` " so that the image disappears. It was only executed to return `V`.
- Modify `shift` so that the next time you run this function, the leftmost vertex will be at the origin. Run `spiralgon` again and you have 1/6th of your figure.
- Now use a `for` loop and plot the next five spiral triangles. You have to add 60° to `d_rot` for each new spiral triangle and also shift `shift` by 60°.

(b) Generate Figure 2 by patching four spiral squares as in the previous part. Each of the four spiral squares is generated with `m=41` squares. Systematically determine other input arguments so as to reproduce the shape. (*Hint.* Pay attention to the direction of each spiral. Study the parameters provided in Problem 1(b) of Exam 1. They were carefully chosen to ensure that all shapes are nicely configured. Or read the instructions found in **LM** 6.8–34, 35.)

(c) Generate Figure 3. Each of the spiral hexagons is generated with `m=51`. Determine other input arguments so as to reproduce the shape.

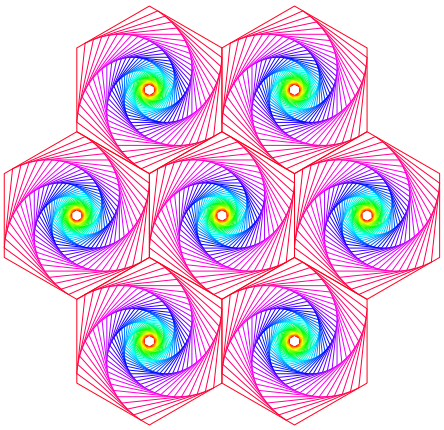


Figure 3: Tiling with spiral hexagons.

3. Do **LM** 9.3–3(a).

4. Do **LM** 9.3–10.

5. (Inverting hyperbolic cosine; **FNC** 1.3.6) The function

$$x = \cosh(t) = \frac{e^t + e^{-t}}{2}$$

can be inverted to yield a formula for $\operatorname{acosh}(x)$:

$$t = \log\left(x - \sqrt{x^2 - 1}\right) \qquad (\star)$$

where $\log(\cdot)$ denotes the natural logarithmic function $\ln(\cdot)$. In MATLAB, let `t=-4:-4:-16` and `x=cosh(t)`.

(a) Find the condition number of the problem $f(x) = \operatorname{acosh}(x)$. (You may use Equation ($\star$), or look up a formula for f' in a calculus book.) Evaluate κ_f at the entries of `x` in MATLAB.

(b) Use Equation ($\star$) on `x` to approximate `t`. Record the accuracy of the answers (by displaying absolute and/or relative errors), and explain. (Warning: Use `format long` to get enough digits or use `fprintf` with a suitable format.)

(c) An alternate formula for $\operatorname{acosh}(x)$ is

$$t = -2\log\left(\sqrt{\frac{x+1}{2}} + \sqrt{\frac{x-1}{2}}\right). \qquad (\dagger)$$

Apply Equation ($\dagger$) to `x` and record the accuracy as in part (b). Comment on your observation.

(d) Based on your experiments, which of the formulas ($\star$) and ($\dagger$) is unstable? What is the problem with that formula?

Homework 4 (Solution)

Math 3607

Tae Eun Kim

Table of Contents

Problem 1 (Log Product; LM 5.6)

Here we compute the product of all elements of $\mathbf{a} = (a_1, a_2, \dots, a_n)^{\mathrm{T}}$

$$P_n = \prod_{i=1}^{n} a_i$$

using the sum of logarithms as

$$\log |P_n| = \sum_{i=1}^{n} \log |a_i|,$$

and then expressing P_n in scientific notation $f \times 10^m$, where $|f| \in [1, 10)$ and m is an integer. How do we obtain f and m from the sum of logs? Let $\lambda = \log |P_n| = m + (\lambda - m)$, where $m = \lfloor \lambda \rfloor$ is the integer part of λ and $\lambda - m \in [0, 1)$ is its fractional part. Upon exponentiation (with base 10), we obtain

$$|P_n| = 10^{\lambda} = 10^{\lambda - m} \times 10^m.$$

Note that the first term $10^{\lambda - m}$ is in $[1, 10)$, so it must correspond to $|f|$. In addition, m is the same m that appears in the scientific notation. In short, the procedure can be summarized as follows:

1. Compute the sum of logs of absolute value of elements; call it λ. In doing so, keep track of the sign of each term. If an element is 0, the answer is 0 so set $f = 0$ and $m = 0$ and exit the program.
2. Determine the integer part m and the fractional part $\lambda - m$ of λ.
3. m is found in the previous step; $f = \pm 10^{\lambda - m}$ where the sign is determined by the number of negative elements.

Below is an example script.

```
% script m-file: logprod.m
% computes the product of elements of  a  using
% input: a
% output: f, m
n = length(a);
sgn = 1;              % sign tracker
sumLog = 0;           % sum of logs of abs vals (lambda)
for i = 1:n
    if a(i) == 0      % if an elem is zero,
        f = 0;        % the product is zero.
        m = 0;
        return
    elseif a(i) < 0
        sgn = (-1)*sgn;
    end
    sumLog = sumLog + log10(abs(a(i)));
end
m = floor(sumLog);            % integer part
f = sgn * 10^(mod(sumLog, 1)); % fractional part
```

We can do better without using any loop. (This is the style that I want you to ultimately acquire from this course.)

```
if any(a==0)                  % if any of the elements is zero
    f = 0; m = 0;
    return
end
nrNeg = length(find(a<0));    % number of negative elements
sumLog = sum(log10(abs(a)));
f = (-1)^nrNeg * 10^(mod(sumLog, 1));
m = floor(sumLog);
```

Solution: (Function m-file)

```
function [f, m] = logprod(a)
% LOGPROD computes the product of elements of  a  using
% sum of logs of its elements
% input: a
% output: f, m
if any(a==0)                  % if any of the elements is zero
    f = 0; m = 0;
    return
end
nrNeg = length(find(a<0));    % number of negative elements
sumLog = sum(log10(abs(a)));
f = (-1)^nrNeg * 10^(mod(sumLog, 1));
m = floor(sumLog);
```

Note: When asked to write a function m-file, use Code Example environment as shown above. If you need to call the function, you may include the function at the end of the live script. See the end of this file.

Explore. What is the largest factorial computable in MATLAB?

```
n = 1;
while factorial(n) < inf
    n = n + 1;
end
n
```

```
n =
   171
```

So $170!$ is the largest factorial computable and $n!$, with $n \geq 171$, overflows to infinity in MATLAB.

```
factorial(170)
```

```
ans =
     7.25741561530799e+306
```

```
factorial(171)
```

```
ans =
   Inf
```

(Recall that `realmax` is the largest double-precision floating-point number.)

```
realmax
```

```
ans =
     1.79769313486232e+308
```

Question: How, then, would you go about calculating $200!$ using MATLAB?

Problem 2 (Tiling with Sprial Polygons)

We have been working with this shapes several times and I hope by now you understand the underlying geometry as well as all programming techniques involved. See at the end of the current file for the code listing.

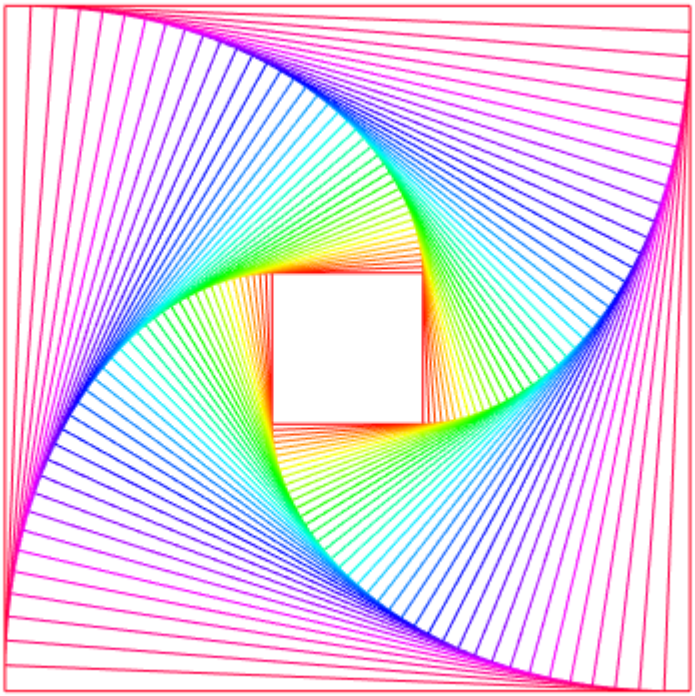
Illustration: Roles of Parameters

To make sure everyone understands what is really going on, take a look at the following examples:

```
m = 4;
rotate = 45;
shift = [0 0];
```

Example 1:

```
clf; spiralgon(m, 41, 2.25, rotate, shift);
```

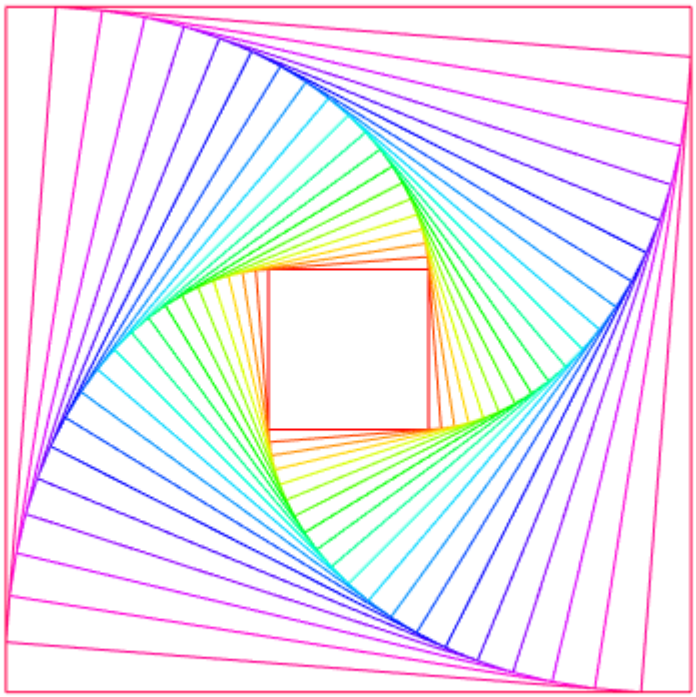


Since the first one is drawn as a square with horizontal and vertical sides and since 40 additional squares are drawn with each rotated by $2.25°$, the very last one is rotated by $40 \times 2.25° = 90°$, and so the last one also have horizontal and vertical sides.

If you understand the previous paragraph, you will see why `n` and `d_angle` values were given as they were.

Example 2:

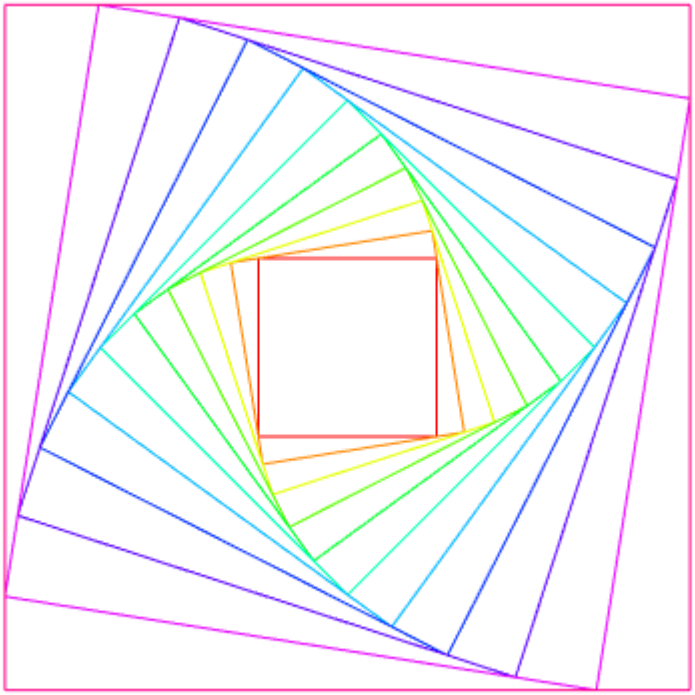
```
clf; spiralgon(m, 21, 4.5, rotate, shift);
```



Here, $20 \times 4.5^{\circ} = 90^{\circ}$. Once again, the outermost square has horizontal and veritcal sides, just as the innermost one.

Example 3:

```
clf; spiralgon(m, 11, 9, rotate, shift);
```



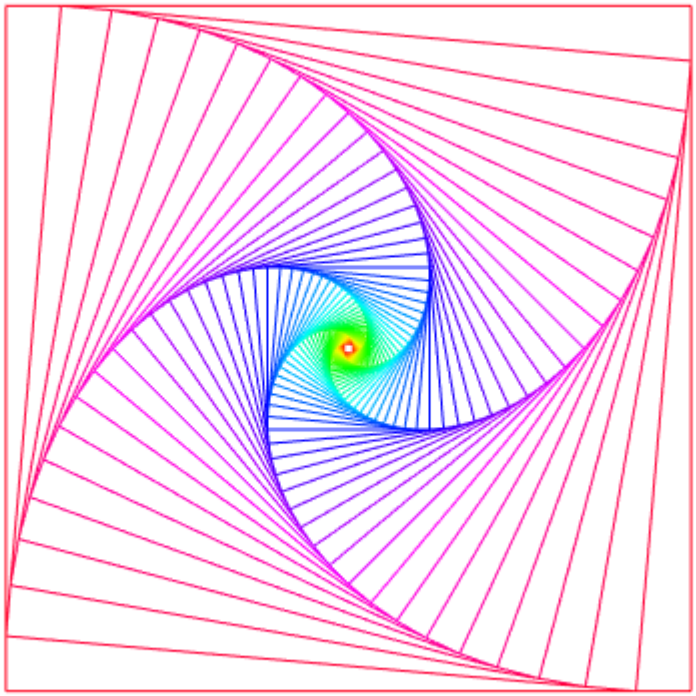
Again, $10 \times 9^\circ = 90^\circ$.

Observation. As you can see, as long as `n` and `d_angle` values are chosen so that

$$(\texttt{n} - 1) \times \texttt{d\_angle}$$

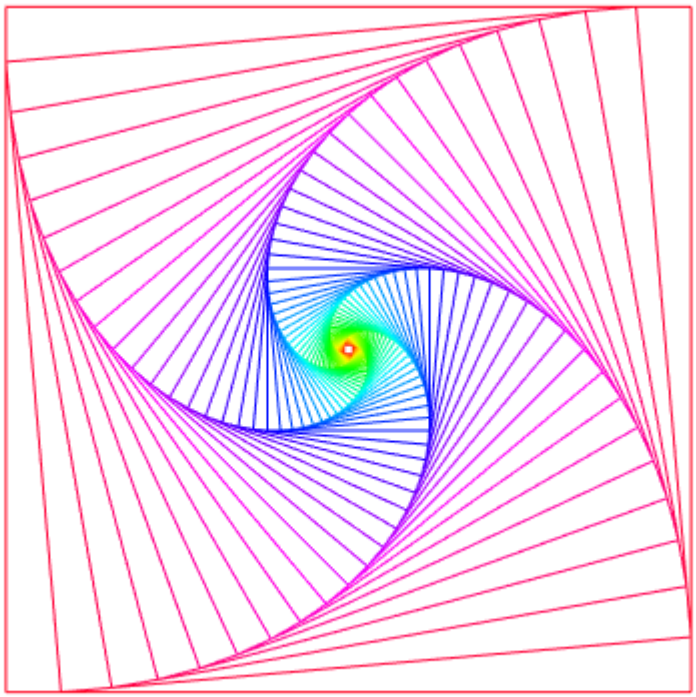
is a multiple of 90, the same kind of visual effect will be created. For instance,

```
clf; spiralgon(m, 55, 5, rotate, shift);
```



Let us also confirm here that the code works with negative `d_angle` value. (We expect to see spirals drawn in the opposite orientation.)

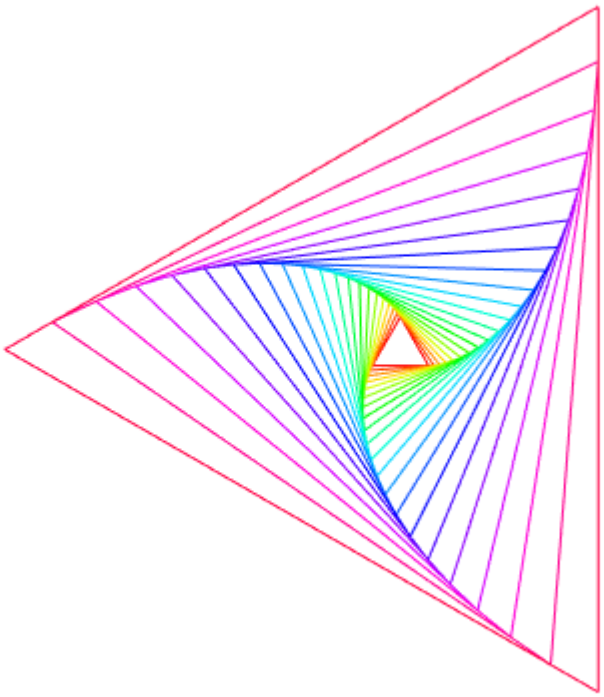
```
clf; spiralgon(m, 55, -5, rotate, shift);
```



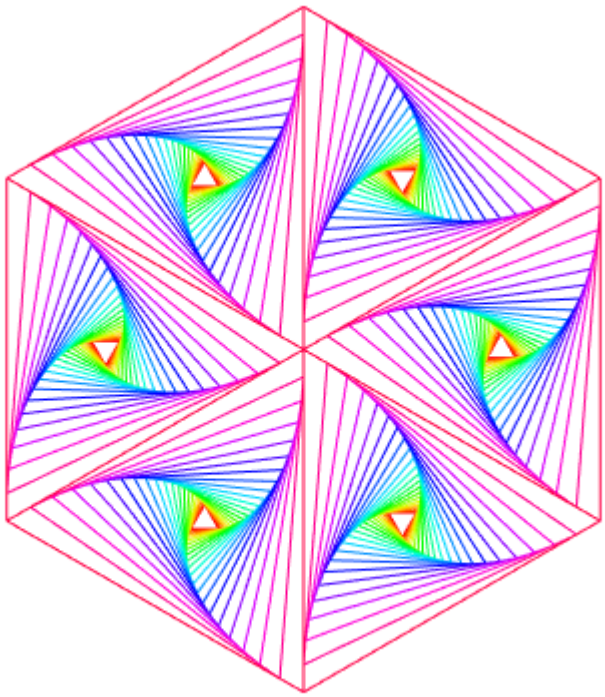
This will become useful in a bit.

(a) Tiling with spiral triangles

```
clf
n = 3; m = 21; d_angle = 4.5; d_rot = 90; shift = [0 0]';
V = spiralgon(n, m, d_angle, d_rot, shift);
hold off

shift = -V(:,1);
spiralgon(n, m, d_angle, d_rot, shift);
```

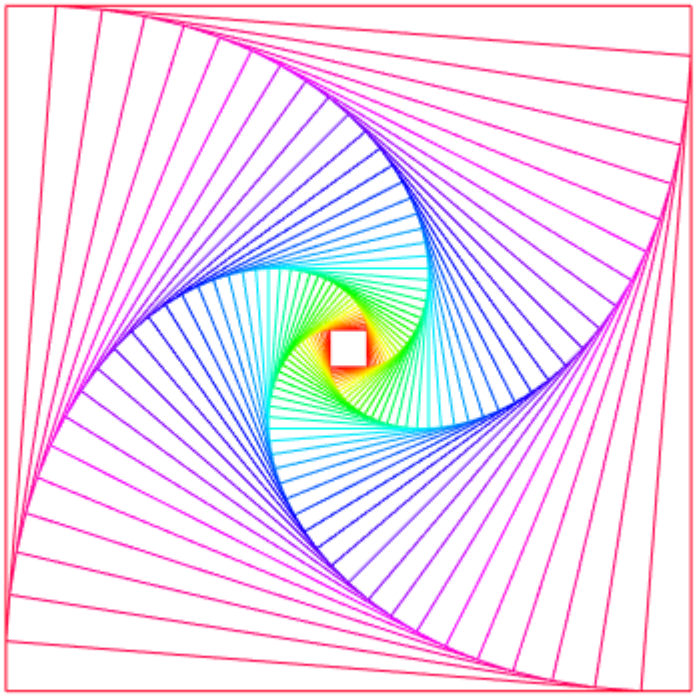


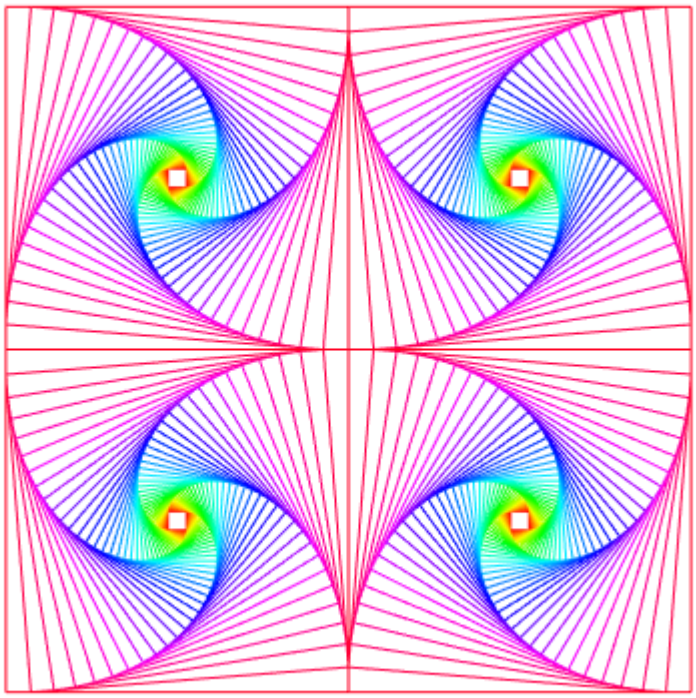
```
R = [cosd(60) -sind(60);
    sind(60)  cosd(60)];
for i = 2:6
    d_rot = d_rot + 60;
    shift = R*shift;
    spiralgon(n, m, d_angle, d_rot, shift);
end
```



(b) Tiling with spiral squares

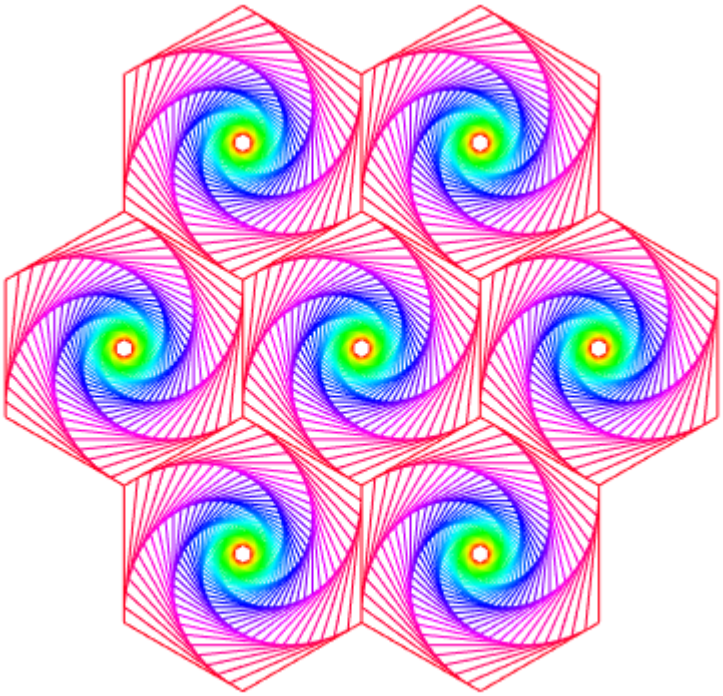
```
clf
n = 4; m = 41; d_angle = 4.5; d_rot = 45; shift = [0 0]';
V = spiralgon(n, m, d_angle, d_rot, shift);
hold off

shift = -V(:,1);
R = [0 -1;      % 90-degree rotation
     1  0];
for i = 1:4
    if i > 1
        shift = R*shift;
    end
    spiralgon(n, m, (-1)^i*d_angle, d_rot, shift);
end
```





(c) Tiling with spiral hexagons

```
clf
V = spiralgon(6, 51, 6, 30, [0 0]);
for i = 1:6
    rotate = sum(V(:,i:i+1), 2);
    spiralgon(6, 51, 6, 30, rotate);
end
```



Problem 3 (LM 9.3--3a)

```
format long g
```

Successor of 8:

```
(8 + 4*eps) - 8
(8 + 4.01*eps) - 8
```

Observe that the gap between `8 + 4.01*eps` and `8` is not `4.01*eps`, but rather `8*eps`.

```
8*eps
```

Predecessor of 16:

```
16 - (16 - 4.01*eps)
16 - (16 - 4*eps)
```

Note that `16 - 4*eps` is registered to be the same as `16` in MATLAB while `16 - 4.01*eps` is rounded down to `16 - 8*eps`. This is how we know that `16 - 8*eps` comes immediately before `16` on the floating-point number system.

Neighbors of 2^{10}:

The gap between 2^{10} and the next floating-point number is $2^{10} \cdot \texttt{eps} = 2^{-42}$.

```
(2^10 + 2^9*eps) - 2^10
(2^10 + (2^9+1)*eps) - 2^10
2^(-42)
```

The gap between 2^{10} and the one before is $2^{9} \cdot \texttt{eps} = 2^{-43}$.

```
2^10 - (2^10 - 2^8*eps)
2^10 - (2^10 - (2^8+1)*eps)
2^(-43)
```

Problem 4 (9.3--10)

```
format long e
```

(a) Since $e^x = 1 + x + \frac{1}{2}x^2 + \cdots = \sum_{j=0}^{\infty} \frac{x^j}{j!}$, the limit is clearly 1.

$$\lim_{x \to 0} f(x) = \lim_{x \to 0} \frac{\sum_{j=1}^{\infty} x^j / j!}{x} = \lim_{x \to 0} \sum_{j=1}^{\infty} \frac{x^{j-1}}{j!} = 1 .$$

(b) The following are used commonly in all subparts.

```
k = [1:20]';
x = 10.^(-k);
```

(i) Note the use of elementwise division below.

```
fx = (exp(x) - 1)./x;
```

(ii) In our textbook, $\log$ denotes the natural log $\ln$, and so the denominator $\log e^x = \ln e^x = x$.

```
f1x = (exp(x) - 1)./log(exp(x));
```

(iii) The function `expm1` computes "exponental function e^x minus 1".

```
f2x = expm1(x)./x;
```

Let's package the results together simply using `disp`.

```
disp([x fx f1x f2x])
```

The first one $f(x)$ suffers horribly from the catastrophic cancellation since $e^x \approx 1$ for small x. To understand why $f_1(x)$ is doing a better job, note that the denominator $\log e^x$, for small x, behaves as follows:

$$\log e^x = \log(1 + x + x^2/2 + \cdots) = x + (\text{higher-order terms}).$$

Another round of power-series-fu reveals that

$$f_2(x) = (e^x - 1)/\log(e^x) = (x + x^2/2 + x^3/6 + \cdots)/(x + \cdots) = 1 + (\text{higher-order terms}),$$

and this is why MATLAB is much more tamed with this encoding. However, when x gets sufficiently small, e^x gets very close to 1 to a point that is not distinguishable on the floating-point number system. In our experiment, that happened when $k \geq 16$:

```
x_small = 1e-16;
exp(x_small)
```

When this is fed into $\log$, it outputs zero resulting in NaN.

```
log(exp(x_small))
```

This explain why we had NaN's for the last 5 results ($16 \leq k \leq 20$).

The function `expm1` was designed to avoid the catastrophic cancellation occurring in calculating $e^x - 1$ for small x. See

```
help expm1
```

Problem 5 (Inverting hyperbolic cosine)

```
t = -4:-4:-16;
x = cosh(t);
```

(a) Let $f(x) = \log(x - \sqrt{x^2 - 1}) = \operatorname{acosh}(x)$. Calculation shows that

$$\kappa_f(x) = \left|\frac{x f'(x)}{f(x)}\right| = \left|\frac{x}{\sqrt{x^2-1}} \cdot \frac{1}{\log(x - \sqrt{x^2-1})}\right|.$$

We evaluate the condition number at the entries of `x` by

```
f  = log(x - sqrt(x.^2-1));
fp = -1./sqrt(x.^2-1);
kappa = abs( x.*fp./f )
```

```
kappa = 1×4
        0.250167787600418         0.125000028131124         0.083333332387735 ···
```

(b) We have already evaluted $t = f(x)$ in part (a), saved as `f`. We compare against the original values stored in `t`;

```
absErr = abs(f - t)';
relErr = absErr./abs(t);
for j = 1:length(x)
    if j == 1
        fprintf(' %10s %16s %16s\n', 'x', 'abs error', 'rel error')
        fprintf(' %45s\n', repmat('-', 1, 45))
    end
    fprintf(' %10.4e %16.8e %16.8e\n', x(j), absErr(j), relErr(j))
end
```

```
         x        abs error        rel error
 ---------------------------------------------
 2.7308e+01   4.61852778e-14   1.15463195e-14
 1.4905e+03   1.71089809e-10   4.27724522e-11
 8.1377e+04   1.37072186e-07   3.42680466e-08
 4.4431e+06   1.37512880e-03   3.43782200e-04
```

Note from the expression for $\kappa_f(x)$ that the condition number becomes large as x increases because the denominator $(x - \sqrt{x^2-1})\log(x - \sqrt{x^2-1}) \to 0$ as $x \to 0$. This explains why the numerical computation incur larger errors, both absolute and relative, when x is large.

(c,d) Let $g(x) = -2\log\left(\sqrt{\frac{x+1}{2}} + \sqrt{\frac{x-1}{2}}\right)$. Analytically, $g(x) = f(x)$. Unlike $f(x)$, however, numerical evaluation of $g(x)$ is done much more stably:

```
g = -2*log(sqrt((x+1)/2) + sqrt((x-1)/2));
absErr = abs(g - t)';
relErr = absErr./abs(t);
for j = 1:length(x)
    if j == 1
        fprintf(' %10s %16s %16s\n', 'x', 'abs error', 'rel error')
        fprintf(' %45s\n', repmat('-', 1, 45))
    end
    fprintf(' %10.4e %16.8e %16.8e\n', x(j), absErr(j), relErr(j))
end
```

```
         x        abs error        rel error
 ---------------------------------------------
 2.7308e+01 0.0000000000e+00 0.0000000000e+00
 1.4905e+03 0.0000000000e+00 0.0000000000e+00
 8.1377e+04 0.0000000000e+00 0.0000000000e+00
 4.4431e+06 0.0000000000e+00 0.0000000000e+00
```

The key difference is that the expression for $g(x)$ does not involve any ill-conditioned steps whereas $f(x)$ requires a subtraction which is prone to catastrophic cancellation for large x as seen in part (b).

Functions Used

Log Product

```
function [f, m] = logprod(a)
% LOGPROD computes the product of elements of  a  using
% sum of logs of its elements
% input: a
% output: f, m
if any(a==0)                  % if any of the elements is zero
    f = 0; m = 0;
    return
end
nrNeg = length(find(a<0));    % number of negative elements
sumLog = sum(log10(abs(a)));
f = (-1)^nrNeg * 10^(mod(sumLog, 1));
m = floor(sumLog);
end
```

Spiral Polygon

```
function V =  spiralgon(m, n, d_angle, rotate, shift)
% FSPRIALGON plots spiraling regular m-gons
% input:   m = the number of vertices
%          n = the number of regular m-gon
%          d_angle = the degree angle between successive m-gons
%          (can be positive or negative)
%          rotate = the degree angle of rotation of the initial m-gon
%          shift = (2-vector) location of the center of all m-gons
% output:  V = the vertices of the outermost m-gon

th = linspace(0, 360, m+1) + rotate;
V = [cosd(th);
    sind(th)];
C = colormap(hsv(n));
scale = sind(90 + 180/m - abs(d_angle))/...
    sind(90 - 180/m);
R = [cosd(d_angle) -sind(d_angle);
    sind(d_angle) cosd(d_angle)];
% hold off
for i = 1:n
    if i > 1
        V = scale*R*V;
    end
    plot(V(1,:)+shift(1), V(2,:)+shift(2), 'Color', C(i,:))
    hold on
end
set(gcf, 'Color', 'w')
axis equal, axis off
```

```
end
```

Updated on February 16, 2021

Math 3607: Homework 5

Due: 11:59PM, Monday, February 22, 2021

TOTAL: 20 points

1. (Improved triangular substitutions; adapted from **FNC** 2.3.5) If $B \in \mathbb{R}^{n\times p}$ has columns $\mathbf{b}_1, \ldots, \mathbf{b}_p$, then we can pose p linear systems at once by writing $AX = B$, where $X \in \mathbb{R}^{n\times p}$ whose jth column $\mathbf{x}_j$ solves $A\mathbf{x}_j = b_j$ for $j = 1, \ldots, p$.

   (a) Modify `backsub.m` and `forelim.m` from Lecture 13 so that they solve the case where the second input is an $n \times p$ matrix, for $p \geq 1$.

   (b) If $AX = I$, then $X = A^{-1}$. Use this fact to write a MATLAB function `ltinverse` that uses your modified `forelim` to compute the inverse of a lower triangular matrix. Test your function on at least two nontrivial matrices, that is, compare the numerical solutions against the exact solutions.

2. (PLU factorization) Complete the program `myplu.m` on p. 20 of Lecture 14 slides. Then test your code by running it on a 500×500 matrix with random entries, *e.g.*, generated by `rand`, `randi`, or `randn`.

   Hint. Read Section 2.1–2.7 of **NCM**, which is freely available at `https://www.mathworks.com/moler/chapters.html`. The code `lutx` found in Section 2.7 may be particularly helpful.

3. (**FNC** 2.4.6) When computing the determinant of a matrix by hand, it is common to use cofactor expansion and apply the definition recursively. But this is terribly inefficient as a function of the matrix size.

   (a) Explain why, if $A = LU$ is an LU factorization,

   $$\det(A) = u_{11}u_{22}\cdots u_{nn} = \prod_{i=1}^{n} u_{ii}.$$

   (This part is an analytical question and you may need to review related linear algebra.)

   (b) Using the result of part (a), write a MATLAB function `determinant` that computes the determinant of a given matrix `A` using `mylu` from Lecture 14. Use your function and the built-in `det` on the matrices `magic(n)` for $n = 3, 4, \ldots, 7$, and make a table (using `disp` or `fprintf`) showing n, the value from your function, and the relative error when compared to `det`.

4. (Row and column operations) Read and study the appendix to Lecture 14. Then do **LM** 10.1–8.

5. Do **LM** 10.1–12(a,b,d).

6. Do **LM** 10.2–1.

7. **Graphics exercise of the week: Koch snowflake**

This is an *optional* problem for those interested in further developing programming skills and creating cool graphics. Read **LM** 7.2 on recursion and do **LM** 7.2–12 and 7.2–13 to generate Koch snowflakes.

The figures below are all generated with `level=6`.

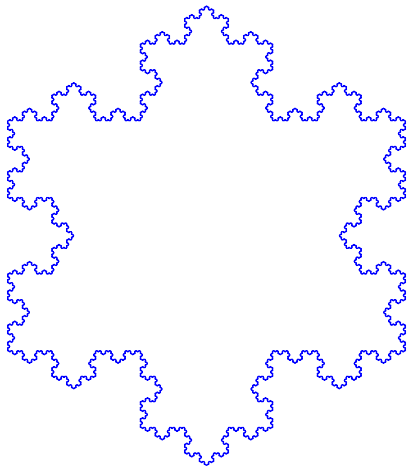


Figure 1: Koch curves around a *negatively* oriented triangle. Generated by letting the angles be $0, -120$, and -240.

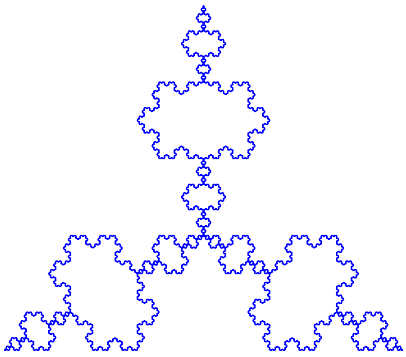


Figure 2: Koch curves around a *positively* oriented trianlge. Generated by letting the angles be $0, 120$, and 240.

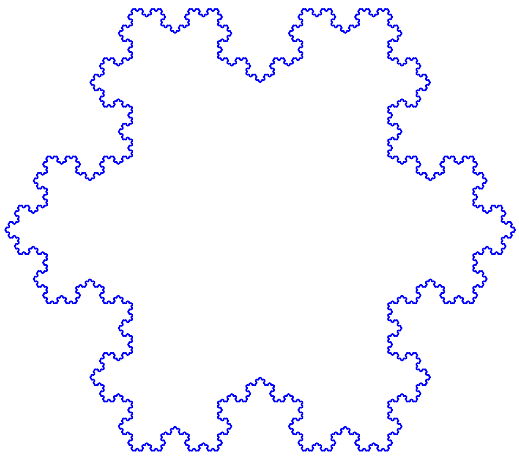


Figure 3: Koch curves around a hexagon. Generated by letting the angles be $0, 60, \ldots, 300$.

Homework 5 (Solution)

Math 3607

Tae Eun Kim

Table of Contents


```
clear
```

Problem 1 (Triangular Substitutions)

(a) The codes provides in lecture can be easily improved to handle multiple triangular systems at once.

Backward Substitution (multiple systems)

The following version can handle $UX = B$ where $B \in \mathbb{R}^{n \times p}$, which demands $X \in \mathbb{R}^{n \times p}$ as well. Pay attention to how the added dimensions in X and B are handled in the code below.

```
function X = backsub(U,B)
% BACKSUB X = backsub(U,B)
% Solve multiple upper triangular linear systems.
% Input:
%   U    upper triangular square matrix (n by n)
%   B    right-hand side vectors concatenated into an (n by p) matrix
% Output:
%   X    solution of UX=B (n by p)
    [n,p] = size(B);
    X = zeros(n,p); % preallocate
    for j = 1:p
        for i = n:-1:1
            X(i,j) = ( B(i,j) - U(i,i+1:n)*X(i+1:n,j) ) / U(i,i);
        end
    end
end
```

Note that the function can handle a single system $U\mathbf{x} = \mathbf{b}$ without any issue using the same syntax as the original version. So we can use it to fully replace the original version presented in class.

Forward Elimination (multiple systems)

Similar modification is made below for forward elimination.

```
function X = forelim(L,B)
% FORELIM X = forelim(L,B)
% Solve multiple lower triangular linear systems.
% Input:
%   L    lower triangular square matrix (n by n)
%   B    right-hand side vectors concatenated into an (n by p) matrix
% Output:
%   X    solution of LX=B (n by p)
    [n,p] = size(B);
    X = zeros(n,p); % preallocate
    for j = 1:p
        for i = 1:n
            X(i,j) = ( B(i,j) - L(i,1:i-1)*X(1:i-1,j) ) / L(i,i);
        end
    end
end
```

(b) The full program `ltinverse` is listed at the end of the file. The key segment of the program is just one line

```
X = forelim(L, eye(size(L)));
```

and note how directly the code translates the mathematical description of the problem of solving $LX = I$ to fine the inverse $X = L^{-1}$ using forward elimination. One more to note is that `eye(size(L))` generates the identity matrix that is of the same size as L, which reads very naturally.

I will compare the code with MATLAB's built-in solver. (Note that this is different from how the question asked to confirm the code.)

```
n = 10;
L = tril(magic(n))
```

```
L = 10×10
    92     0     0     0     0     0     0     0     0     0
    98    80     0     0     0     0     0     0     0     0
     4    81    88     0     0     0     0     0     0     0
    85    87    19    21     0     0     0     0     0     0
    86    93    25     2     9     0     0     0     0     0
    17    24    76    83    90    42     0     0     0     0
    23     5    82    89    91    48    30     0     0     0
    79     6    13    95    97    29    31    38     0     0
    10    12    94    96    78    35    37    44    46     0
    11    18   100    77    84    36    43    50    27    59
```

We can find its inverse by asking MATLAB to solve $LX = I$ using the backslash:

```
Linv = L\eye(n);
```

The residual $LX - I$ has a small norm

```
norm(L*Linv-eye(n))
```

```
ans =
      1.99225401607197e-15
```

at the level of machine epsilon.

Let's calculate the inverse using our code and compute the norm of the residual:

```
myLinv = ltinverse(L);
norm(L*myLinv - eye(n))
```

```
ans =
      3.00841133167772e-15
```

This looks good as well.

Lastly, let's see how the two results compare:

```
norm(Linv - myLinv)
```

```
ans =
      7.90892925731122e-17
```

As expected, the two results show a very good agreement.

Problem 2 (PLU Factorization)

G.E. with partial pivoting.

This was one of the exercises in the lecture. This provides a good starting point for PLU factorization code.

```
function x = GEpp(A, b)
% GEPP pivoted Gaussian Elimination (instructional version)
% Input:
%   A = square matrix
%   b = right-hand side vector
% Output:
%   x = vector solving A*x = b
    % row reduction to upper triangular system
    S = [A, b];                         % augmented matrix
    n = size(A, 1);
    for j = 1:n-1
        [~, iM] = max(abs(A(j,j:end))); % index of pivot element
        iM = iM + j - 1;                % adjusted index
        if j ~= iM                      % pivot if necessary
```

```
            piv = [j iM];
            S(piv, :) = S(flip(piv), :).
        end
        for i = j+1:n
            mult = -S(i,j)/S(j,j);      % R_i --> R_i + (-S(i,j)/S(j,j))*R_j
            S(i,:) = S(i,:) + mult*S(j,:);
        end
    end
    % back subs (pretending that there is no 'backsub' routine)
    U = S(:,1:end-1);
    beta = S(:,end);
    x(n) = beta(n)/U(n,n);
    for i = n-1:-1:1
        x(i) = (beta(i) - U(i,i+1:end)*x(i+1:end))/U(i,i);
    end
end
```

Note that it is written as a self-contained program which contains a hard-coded backward substitution routine. One can confirm with the knowledge of Gaussian transformation matrices that the multiplier $a_{i,j}/a_{j,j}$ is directly related to the construction of L matrix. See how it is coded below.

Solution.

```
function [L,U,P] = myplu(A)
% MYPLU   PLU factorization code (instructional version)
% Input:
%   A = square matrix
% Output:
%   P,L,U = permutation, unit lower triangular, and upper triangular
%           matrices such that P*A = L*U.
    n = length(A);
    P = eye(n);   % preallocate P
    L = eye(n);   % preallocate L
    % A will be overwritten below to be U
    for j = 1:n-1
        [~, iM] = max(abs(A(j:n, j)));
        iM = iM + j - 1;                % adjustment
        if j ~= idx_adj
            piv = [j, iM];              % pivot
            P(piv, :) = P(flip(piv), :); % update permutation matrix
            A(piv, :) = A(flip(piv), :); % update A
            L(:, piv) = L(:, flip(piv)); % update L by emulating P*G*P
            L(piv, :) = L(flip(piv), :); % where P is an elem. perm. mat.
        end
        for i = j+1:n
            L(i,j) = A(i,j) / A(j,j);    % row multiplier
            A(i,j:n) = A(i,j:n) - L(i,j)*A(j,j:n);
        end
    end
    U = triu(A);  % to ensure that U come out as a clean upper-trian. mat.
end
```

Problem 3 (Determinant)

(a) Recall from linear algebra that $\det(LU) = \det(L)\det(U)$. In addition, recall that the determinant of a triangular matrix is the product of its diagonal entries. Since L in the LU factorization is a unit lower triangular matrix, it follows that

$$\det(A) = \det(LU) = \det(L)\det(U) = (1 \cdot 1 \cdots 1)(u_{11}u_{22}\cdots u_{nn}) = u_{11}u_{22}\cdots u_{nn}.$$

(b) The previous part suggests that we can write the following program computing the determinant of a given matrix:

```
function D = determinant(A)
    [~,U] = mylu(A);
    D = prod(diag(U));
end
```

Since L is not needed in the computation, the LU factorization routine is called with `[~,U] = mylu(A)`. Let's test it.

```
for n = 3:7
    A = magic(n);
    myDet = determinant(A);
    Det = det(A);
    if n == 3
        fprintf(' %2s %20s %20s %12s\n', 'n', 'myDet', 'Det', 'rel. error')
        fprintf(' %57s\n', repmat('-', 1, 57))
    end
    fprintf(' %2d %20.8g %20.8g %12.4g\n', n, myDet, Det, abs((myDet-Det)/Det))
end
```

```
  n                myDet                  Det    rel. error
 ---------------------------------------------------------
  3                 -360                 -360             0
  4         3.623768e-13       -1.4495072e-12          1.25
  5              5070000              5070000     7.348e-16
  6                    0       -8.0494971e-09             1
  7        -3.480528e+11        -3.480528e+11     3.507e-16
```

Problem 4 (LM 10.1--8)

Let $P_{1,4}$ be the 5-by-5 elementary permutation matrix obtained by interchanging its 1st and 4th rows:

$$P_{1,4} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

(a) $P_{1,4}A$

(b) $AP_{1,4}$

(c) $P_{1,4}AP_{1,4}$

(d) By carrying out the described column operations onto I, we obtain the following permutation matrix, call it Q:

$$Q = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

To reiterate the same column operations onto A, multiply Q to its right: AQ. Note that each time Q is multiplied to a matrix (from the right), it shifts first four columns to the left in a cyclic fashion while leaving the fifth one intact. Hence, 4 repeated multiplication by Q will leave A unchanged, in other words, $Q^4 = I$. Note that $Q^k = I$ for any k that is a multiple of 4.

The action of Q can be decomposed into:

1. Swap 1st and 2nd columns. → column order: (2,1,3,4,5)
2. Swap 2nd and 3rd columns. → column order: (2,3,1,4,5)
3. Swap 3rd and 4th columns. → column order: (2,3,4,1,5)

It follows that $Q = P_{1,2}P_{2,3}P_{3,4}$. The order is important here.

```
I = eye(5);
P12 = I(:,[2 1 3 4 5]);
P23 = I(:,[1 3 2 4 5]);
P34 = I(:,[1 2 4 3 5]);
Q = P12*P23*P34
```

```
Q = 5×5
     0     0     0     1     0
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     0     1
```

```
R = P34*P23*P12
```

```
R = 5×5
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     1     0     0     0     0
     0     0     0     0     1
```

Note that AQ cycles first four columns of A

```
A = reshape(1:25,5,5);
A*Q
```

```
ans = 5×5
     6    11    16     1    21
     7    12    17     2    22
     8    13    18     3    23
     9    14    19     4    24
    10    15    20     5    25
```

while RA cycles first four rows of A

```
R*A
```

```
ans = 5×5
     2     7    12    17    22
     3     8    13    18    23
     4     9    14    19    24
     1     6    11    16    21
     5    10    15    20    25
```

Problem 5 (LM 10.1--12a,b,d)

All vectors appearing are in $\mathbb{R}^n$; all matrices are in $\mathbb{R}^{n\times n}$.

(a) We calculate $\mathbf{x} = ABCD\mathbf{b}$ as below

```
x = A * (B * (C * (D * b)))
```

What MATLAB does is:

1. form a vector `D*b`: $\sim 2n^2$ flops
2. left-multiply the previous result by `C`: $\sim 2n^2$ flops
3. left-multiply the previous result by `B`: $\sim 2n^2$ flops
4. left-mulitply the previous result by `A`: $\sim 2n^2$ flops

In total, it takes $\sim 8n^2$ flops to calculate `x`.

If it were to be computed by, say,

```
x = (A * (B * (C * D))) * b
```

the computation of `C*D` itself already take $\sim 2n^3$ flops already.

Lesson. Try to avoid (matrix) × (matrix) multiplication if possible!

(b) To compute $\mathbf{x} = BA^{-1}\mathbf{b}$ efficiently, do

```
x = B * (A \ b);
```

This way:

1. calculate $A^{-1}\mathbf{b}$ using backslash operator `A\b`: $\sim \frac{2}{3}n^3$ flops (because `\` does a pivoted GE in general)

2. left-multiply the previous result by `B`: $\sim 2n^2$ flops (since it is a (matrix) $\times$ (vector) multiplication.)

So, all in all, it take $\sim \frac{2}{3}n^3$ flops. (Recall that we only retain the dominant term in the asymptotic notation.)

(d) To efficiently compute $\mathbf{x} = B^{-1}(C + A)\mathbf{b}$, do

```
x = B \ ((C + A) * b)
```

By doing this:

1. form `C + A`: $\sim n^2$ flops
2. multiply it by `b`: $\sim 2n^2$ flops
3. left-"divide" by `B` using `\` (pivoted GE): $\sim \frac{2}{3}n^3$ flops

Altogether, it takes $\sim \frac{2}{3}n^3$ flops.

Problem 6 (LM 10.2--1)

(a) **No**. The norm of a matrix A is 0 if and only if A is the zero matrix by definition.

(The following is an extra explanation. The previous sentence is good enough justification for your submission.) Any nontrivial singular matrix has a positive norm, *e.g.*,

```
A = [1 0;
     0 0]
```

```
A = 2×2
     1     0
     0     0
```

is evidently singular ($\det A = 0$), but its norm is 1.

```
norm(A, 1)
```

```
ans =
     1
```

```
norm(A, 2)
```

```
ans =
     1
```

```
norm(A, Inf)
```

```
ans =
```

```
     1
```

```
norm(A, 'fro')
```

```
ans =
     1
```

(b) **Yes**. The alternate definition given in (10.20) on p.1481 is useful when A^{-1} does not exists, *i.e.,* when A is singular. Suppose A is singular. Then the equation $A\mathbf{x} = \mathbf{0}$ has a nontrivial solution $\mathbf{x}$ and so $\min_{\|\mathbf{x}\|_p=1} \|A\mathbf{x}\|_p = 0$.

Therefore, in light of the alternate definition, the condition number of a singular matrix is infinite. MATLAB adheres to this convention:

```
cond(A)
```

```
ans =
   Inf
```

(c) **Yes**, because

$$\kappa_p(A) = \|A\|_p \|A^{-1}\|_p = \|(A^{-1})^{-1}\|_p \|A^{-1}\|_p = \kappa_p(A^{-1}).$$

(d) **No.** This was explained in lecture. The essense of the argument was that

$$1 = \|I\|_p = \|AA^{-1}\|_p \leq \|A\|_p \|A^{-1}\|_p = \kappa_p(A).$$

(e) The last bullet point on p. 1481 is helpful here which states that

$$\kappa_2(A) = \sqrt{\frac{\lambda_{\max}(A^{\mathrm{T}}A)}{\lambda_{\min}(A^{\mathrm{T}}A)}}.$$

In case A is symmetric, the above reduces to

$$\kappa_2(A) = \frac{\max_i |\lambda_i|}{\min_i |\lambda_i|}.$$

Exercise. Confirm it!

Let's use the second formulation to construct a matrix $A \in \mathbb{R}^{10\times 10}$ with the desired properties. To keep things simple, take A not just symmetric, but diagonal as in

$$A = \begin{bmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_{10} \end{bmatrix}, \text{ with } d_j\text{'s all positive.}$$

We know from linear algebra that $\det A = d_1 d_2 \cdots d_{10}$ and the diagonal entries $d_1, d_2, \ldots, d_{10}$ are its eigenvalues. Since we want $\kappa_2(A) = 1$, we need all d_j's to have the same (absolute) value, say d. Then $\det A = d^{10} = 10^{-20}$, which implies that $d = 10^{-2}$. Therefore,

$$A = \begin{bmatrix} 10^{-2} & & & \\ & 10^{-2} & & \\ & & \ddots & \\ & & & 10^{-2} \end{bmatrix} = 10^{-2} I,$$

will do. Let's confirm this on MATLAB:

```
A = 1e-2*eye(10);
det(A)
```

```
ans =
                    1e-20
```

```
cond(A)
```

```
ans =
     1
```

Functions Used

Backward Substitutions

```
function X = backsub(U,B)
% BACKSUB X = backsub(U,B)
% Solve multiple upper triangular linear systems.
% Input:
%   U    upper triangular square matrix (n by n)
%   B    right-hand side vectors concatenated into an (n by p) matrix
% Output:
%   X    solution of UX=B (n by p)
    [n,p] = size(B);
    X = zeros(n,p); % preallocate
    for j = 1:p
        for i = n:-1:1
            X(i,j) = ( B(i,j) - U(i,i+1:n)*X(i+1:n,j) ) / U(i,i);
        end
    end
end
```

Forward Elimination

```
function X = forelim(L,B)
% FORELIM X = forelim(L,B)
% Solve multiple lower triangular linear systems.
% Input:
%   L    lower triangular square matrix (n by n)
```

```
%   B    right-hand side vectors concatenated into an (n by p) matrix
% Output:
%   X    solution of LX=B (n by p)
    [n,p] = size(B);
    X = zeros(n,p); % preallocate
    for j = 1:p
        for i = 1:n
            X(i,j) = ( B(i,j) - L(i,1:i-1)*X(1:i-1,j) ) / L(i,i);
        end
    end
end
```

Inverse of a lower triangular matrix

```
function X = ltinverse(L)
% LTINVERSE X = ltinverse(L)
% Find the inverse of a lower triangular matrix.
% Input:
%   L    lower triangular square matrix (n by n)
% Output:
%   X    inverse of L, i.e., LX = I. (n by n)
    if ~istril(L) || diff(size(L))~=0 % if input is not lower triangular nor square
        error('The input must be a lower triangular square matrix');
    end
    X = forelim(L, eye(size(L)));
end
```

Determinant based on LU factorization

```
function D = determinant(A)
% DETERMINANT D = determinant(A)
% Calculate the determinant of A using LU factorization.
% Input:
%   A   square matrix
% Output:
%   D   determinant of A
% Dependency: mylu (see below)
    [~,U] = mylu(A);
    D = prod(diag(U));
end
```

LU factorization routine

```
function [L,U] = mylu(A)
    n = length(A);
    L = eye(n);
    for j = 1:n-1
        for i = j+1:n
            L(i,j) = A(i,j) / A(j,j);
            A(i,j:n) = A(i,j:n) - L(i,j)*A(j,j:n);
        end
    end
    U = triu(A);
end
```

Updated on February 23, 2021

Math 3607: Homework 6

Due: 11:59PM, Monday, March 1, 2021

TOTAL: 20 points

1. Do **LM** 12.5–3. (Understanding matrix multiplication)

2. Do **LM** 12.6–2. (Gram-Schmidt)

3. (Adapted from **FNC** 3.3.3.) Let $x_1, x_2, \ldots, x_m$ be m equally spaced points in $[-1, 1]$. Let V be the Vandermonde-type matrix appearing on p.10 of Lecture 17 slides for $m = 400$ and $n = 5$. Find the thin QR factorization of $V = \widehat{Q}\widehat{R}$, and, on a single graph, plot every column of $\widehat{Q}$ as a function of the vector $\mathbf{x} = (x_1, x_2, \ldots, x_m)^{\mathrm{T}}$. (Use MATLAB to solve this problem.)

4. (Visualizing matrix norms; adapted from **LM** 9.4–26.) For $p \in [1, \infty]$, recall the definition of the matrix p-norm,
$$\|A\|_p = \max_{\|\mathbf{x}\|_p = 1} \|A\mathbf{x}\|_p \,.$$
To understand this definition, we will work in two-dimensional space so that we can easily plot the results. For this problem, use
$$A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}. \tag{1}$$
As an illustration, we study the case $p = 2$ following the steps below.

   - Create unit vectors $\mathbf{x}_j$ in 2-norm,
   $$\mathbf{x}_j = \begin{bmatrix} \cos\theta_j \\ \sin\theta_j \end{bmatrix}, \quad 1 \leqslant j \leqslant 361 \tag{2}$$
   using 361 evenly distributed θ_j in $[0, 2\pi]$. Make sure $\mathbf{x}_1 = \mathbf{x}_{361} = (1, 0)^{\mathrm{T}}$, just as in the spiral polygon problem. Plot these points, which lie on the unit circle. Make sure the plot looks like a circle.
   - For each j, let $\mathbf{y}_j = A\mathbf{x}_j$. Plot all points $\mathbf{y}_j$. In addition, store $\|\mathbf{y}_j\|_2$ for all j in a vector.
   - Plot $\|\mathbf{y}_j\|_2$ as a function of θ_j.
   - Find the maximum value of $\|\mathbf{y}_j\|_2$ over all j. This estimates $\|A\|_2$. Compare this against the actual value computed by `norm(A,2)`.

   These steps are carried out by the following script.

```
A = [2 1; 1 3];
theta = linspace(0, 2*pi, 361);
X = [cos(theta); sin(theta)];
Y = A*X;
norm_Y = sqrt(sum(Y.^2, 1));
```

```
clf
subplot(2,2,1)
plot(X(1,:), X(2,:)), axis equal
title('x: unit vectors in 2-norm')
subplot(2,2,2)
plot(Y(1,:), Y(2,:)), axis equal
title('Ax: image of unit vectors under A')
subplot(2,1,2)
plot(theta, norm_Y), axis tight
xlabel('\theta')
ylabel('||y||')
fprintf(' p = 2\n')
fprintf(' approx. norm: %18.16f\n', max(norm_Y))
fprintf(' actual  norm: %18.16f\n', norm(A, 2))
```

which generates Figure 1 and the following outputs in the Command Window:

```
p = 2
approx. norm: 3.6179964204609893
actual  norm: 3.6180339887498953
```

Modify the script to carry out the same tasks for $p = 1, \infty$. Pay particular attention to the lines where `X` is defined and `norm_Y` is calculated.

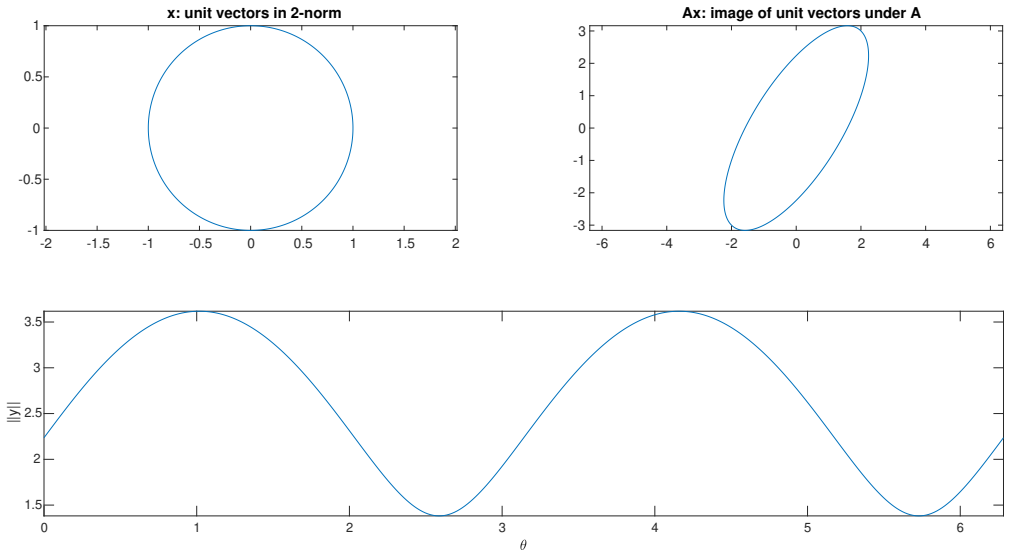


Figure 1: Plots illustrating the definition of matrix norm.

5. **Graphics exercise of the week: Julia and Mandelbrot Sets**

   This is an *optional* problem for those interested in further developing programming skills and creating cool graphics. Read **LM** 6.8.3 on Julia and Mandelbrot sets. Then generate fractals by working out **LM** 6.8–69.

Homework 6 (Solution)

Math 3607

Tae Eun Kim

Table of Contents


```
clear, close all, format short g
```

Problem 1 (LM 12.5--3)

In what follows, $\mathbf{a}_j \in \mathbb{R}^m$ denotes the jth column of A.

(a) $\mathbf{r}\mathbf{w}^{\mathrm{T}}$ is an $n \times n$ matrix whose ith row is

$$r_i \mathbf{w}^{\mathrm{T}} = \left[r_i w_1, \, r_i w_2, \, \ldots, \, r_i w_n \right] \in \mathbb{R}^{1 \times n}.$$

(b) $\mathbf{b}^{\mathrm{T}} A$ is a row vector with n element whose jth element is

$$\mathbf{b}^{\mathrm{T}} \mathbf{a}_j = \sum_{i=1}^{m} b_i a_{ij}.$$

(c) C is an $m \times p$ matrix whose (i, j)-entry is given by

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}.$$

(d) D is a $p \times m$ matrix whose (i, j)-entry is given by

$$d_{i,j} = \sum_{k=1}^{n} b_{k,i} a_{j,k}.$$

(e) Let $E = C^{\mathrm{T}}$. Then E is a $p \times m$ matrix whose (i, j)-entry equals the (j, i)-entry of C, that is,

$$e_{i,j} = c_{j,i} = \sum_{k=1}^{n} a_{j,k} b_{k,i}.$$

Note that it equals $d_{i,j}$ for all i and j. Since both D and E are of the same size and all their entries match, $D = E$.

(f) $A^{\mathrm{T}}A$ is an $n \times n$ matrix whose (i, j)-entry is given by

$$\sum_{k=1}^{m} a_{k,i} a_{k,j}.$$

(g) AA^{T} is an $m \times m$ matrix whose (i, j)-entry is given by

$$\sum_{k=1}^{n} a_{i,k} a_{j,k}.$$

(h) $AA^{\mathrm{T}}\mathbf{b}$ is a column vector with m elements whose ith element is given by

$$\sum_{j=1}^{m} \sum_{k=1}^{n} a_{i,k} a_{j,k} b_j.$$

Suggestion for Further Study. Having done this exercise, please read the derivation of the normal equation from LLS formulation using calculus (Appendix to Lecture 17). The very last step of the proof requires your familiarity with this exercise.

Problem 2 (LM 12.6--2)

Below is a version which is more or less a direct translation of the Gram-Schmidt procedure:

```
function [Q,R] = gs(A)
    [m,n] = size(A);
    Q = zeros(m,n);
    R = zeros(n,n);
    for j = 1:n
        if j > 1
            R(1:j-1,j) = Q(:,1:j-1).' * A(:,j);
        end
        v = A(:,j) - Q(:,1:j-1)*R(1:j-1,j);
        R(j,j) = norm(v);
        Q(:,j) = v/R(j,j);
    end
end
```

We can have `A` overwirtten by `Q` to save memory storages. In addition, we can also get by without the auxilliary vector `v` introduced right after the if statement. The code included at the end of this document incorporates all these modifications.

Let's check if the code works correctly.

```
m = 1000; n = 30;
A = rand(m,n);
```

Using `gs.m`:

```
[Q,R] = gs(A);
istriu(R)                    % Is R upper triangular?
```

```
ans = logical

   1
```

```
norm(Q'*Q - eye(n), 'inf')    % Is Q orthogonal?
```

```
ans =
    2.639e-14
```

```
norm(Q*R - A, 'inf')          % Is Q*R = A?
```

```
ans =
    4.996e-16
```

Using `qr`:

```
[Q0,R0] = qr(A,0);
istriu(R0)                   % Is R upper triangular?
```

```
ans = logical

   1
```

```
norm(Q0'*Q0 - eye(n), 'inf') % Is Q orthogonal?
```

```
ans =
   2.9764e-15
```

```
norm(Q0*R0 - A, 'inf')       % Is Q*R = A?
```

```
ans =
   3.7814e-13
```

Note 1.

Several runs with random matrices shows that both code works as expected. However, Q and R produced the codes are not necessarily the same.

```
norm(Q-Q0, 'inf')
```

```
ans =
```

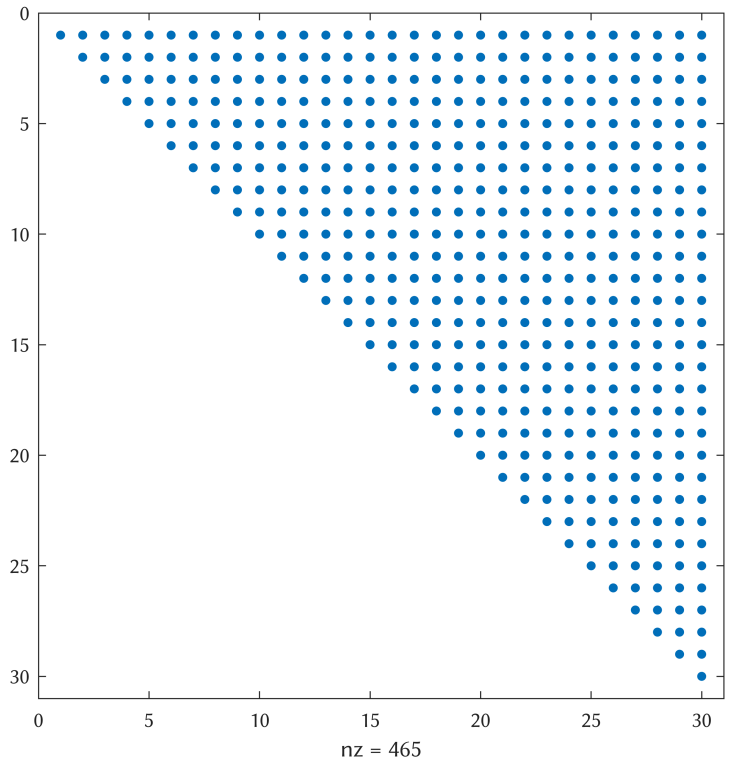
```
1.247
```

as more is going on for MATLAB's `qr` to ensure numerical stability.

Note 2.

There is a visually convenient way to confirm the triangularity of R:

```
spy(R)
```



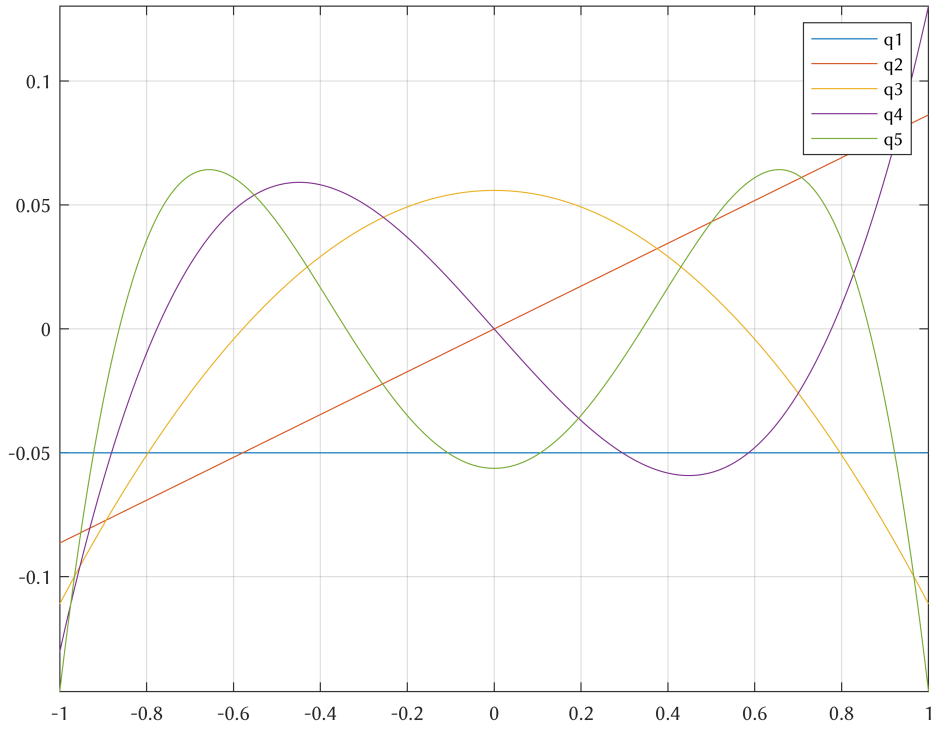
To learn about this function, see

```
help spy
```

```
  spy Visualize sparsity pattern.
     spy(S) plots the sparsity pattern of the matrix S.

     spy(S,'LineSpec') uses the color and marker from the line
     specification string 'LineSpec' (See PLOT for possibilities).

     spy(S,markersize) uses the specified marker size instead of
     a size which depends upon the figure size and the matrix order.

     spy(S,'LineSpec',markersize) sets both.

     spy(S,markersize,'LineSpec') also works.

     Documentation for spy
```

Problem 3 (Orthogonal Polynomials)

```
m = 400; n = 5;
x = linspace(-1, 1, m)';
V = x.^(0:n-1);
[Q,R] = qr(V,0);    % thin QR factorization

clf
plot(x, Q)
axis tight, grid on
legend('q1', 'q2', 'q3', 'q4', 'q5')
```



Note 1. The `plot` function smartly plots all columns of `Q` as functions of the common `x` when called with `plot(x, Q)`. This is the same as

```
for j = 1:size(Q, 2)
    plot(x, Q(:,j)), hold on
end
```

Where does one learn about such tricks? Please read the very first paragraph of the help document for `plot`:

```
help plot
```

```
  plot   Linear plot.
     plot(X,Y) plots vector Y versus vector X. If X or Y is a matrix,
     then the vector is plotted versus the rows or columns of the matrix,
     whichever line up.  If X is a scalar and Y is a vector, disconnected
     line objects are created and plotted as discrete points vertically at
     X.

     plot(Y) plots the columns of Y versus their index.
```

If Y is complex, **plot**(Y) is equivalent to **plot**(real(Y),imag(Y)).
In all other uses of **plot**, the imaginary part is ignored.

Various line types, plot symbols and colors may be obtained with
plot(X,Y,S) where S is a character string made from one element
from any or all the following 3 columns:

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

For example, **plot**(X,Y,'c+:') plots a cyan dotted line with a plus
at each data point; **plot**(X,Y,'bd') plots blue diamond at each data
point but does not draw any line.

plot(X1,Y1,S1,X2,Y2,S2,X3,Y3,S3,...) combines the plots defined by
the (X,Y,S) triples, where the X's and Y's are vectors or matrices
and the S's are strings.

For example, **plot**(X,Y,'y-',X,Y,'go') plots the data twice, with a
solid yellow line interpolating green circles at the data points.

The **plot** command, if no color is specified, makes automatic use of
the colors specified by the axes ColorOrder property. By default,
plot cycles through the colors in the ColorOrder property. For
monochrome systems, **plot** cycles over the axes LineStyleOrder property.

Note that RGB colors in the ColorOrder property may differ from
similarly-named colors in the (X,Y,S) triples. For example, the
second axes ColorOrder property is medium green with RGB [0 .5 0],
while **plot**(X,Y,'g') plots a green line with RGB [0 1 0].

If you do not specify a marker type, **plot** uses no marker.
If you do not specify a line style, **plot** uses a solid line.

plot(AX,...) plots into the axes with handle AX.

plot returns a column vector of handles to lineseries objects, one
handle per plotted line.

The X,Y pairs, or X,Y,S triples, can be followed by
parameter/value pairs to specify additional properties
of the lines. For example, **plot**(X,Y,'LineWidth',2,'Color',[.6 0 0])
will create a plot with a dark red line width of 2 points.

Example

```
   x = -pi:pi/10:pi;
   y = tan(sin(x)) - sin(tan(x));
   plot(x,y,'--rs','LineWidth',2,...
                   'MarkerEdgeColor','k',...
                   'MarkerFaceColor','g',...
                   'MarkerSize',10)
```

See also plottools, semilogx, semilogy, loglog, plotyy, plot3, grid,

```
title, xlabel, ylabel, axis, axes, hold, legend, subplot, scatter.

Documentation for plot
Other functions named plot
```

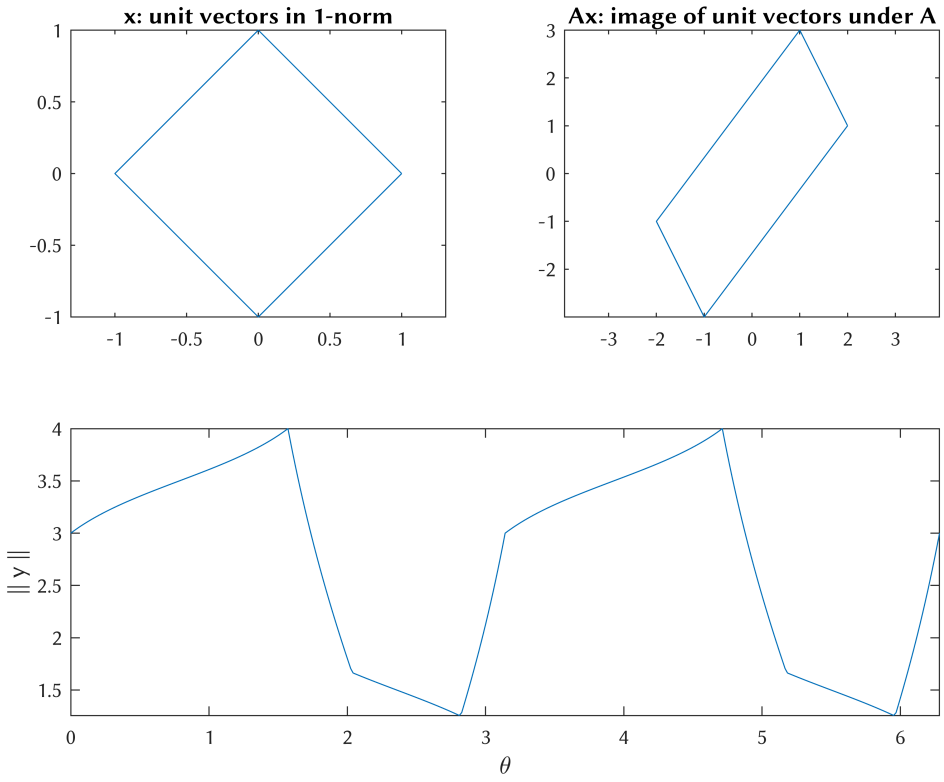
Problem 4 (Visualizing Matrix Norms)

See the function `visMatrixNorms` at the end of this document.

```
A = [2 1;
     1 3];
```

$p = 1$:

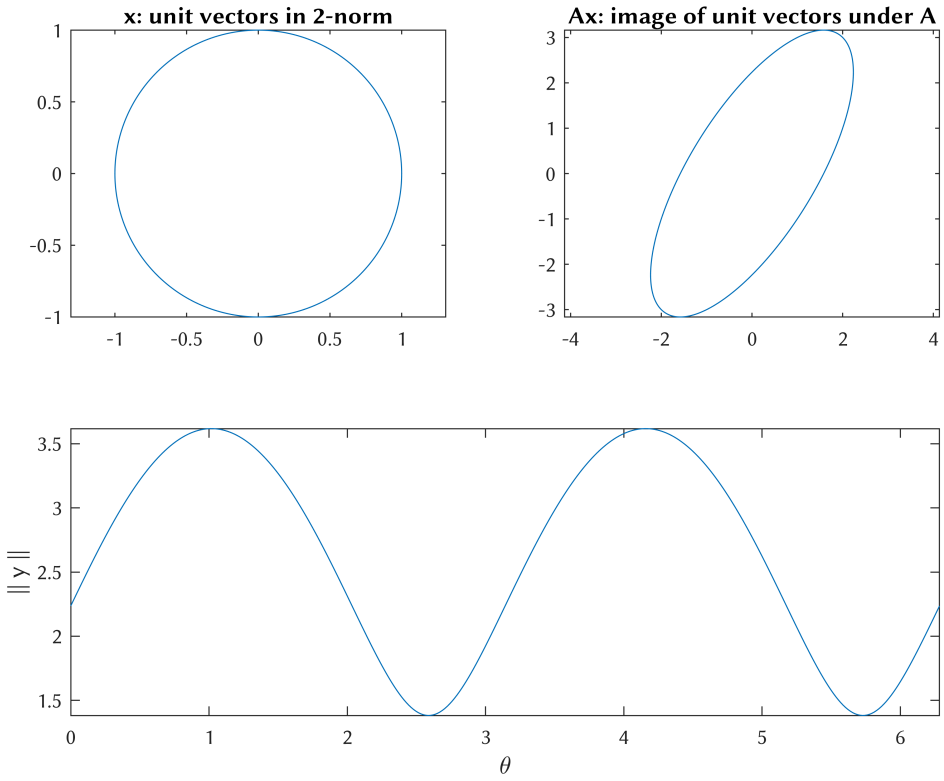
```
visMatrixNorms(A, 1);
```




```
p = 1
approx. norm: 4.0000000000000000
actual  norm: 4.0000000000000000
```

$p = 2$:

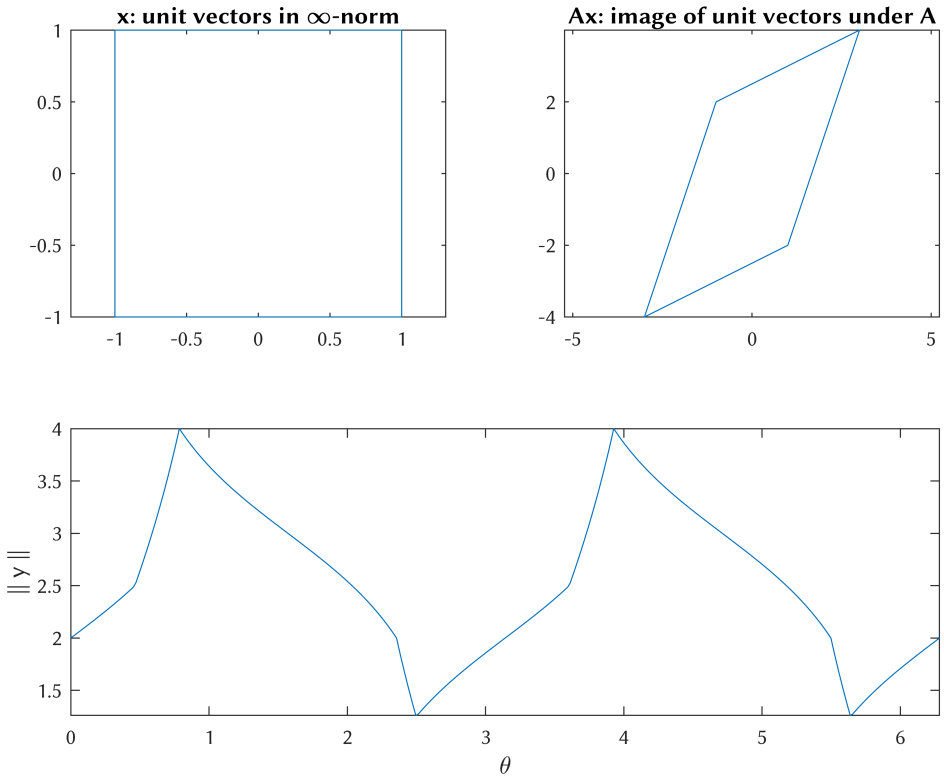
```
visMatrixNorms(A, 2);
```




```
p = 2
approx. norm: 3.6179964204609893
actual  norm: 3.6180339887498953
```

$p = \infty$:

```
visMatrixNorms(A, Inf);
```




```
p = Inf
approx. norm: 3.9999999999999996
actual  norm: 4.0000000000000000
```

Explore.

Modify the script/function even further so that it can handle any $p \in [1, \infty]$. MATLAB's `norm` function can calculate vector p-norm for any value p, but it can only calculate $\|A\|_p$ for $p = 1, 2, \infty$ and $\|A\|_F$ (Frobenius norm). So for p other than $1, 2, \infty$, your program must only output the approximate $\|A\|_p$.

Functions Used

Gram-Schmidt

```
function [Q,R] = gs(A)
    n = size(A,2);
    Q = A;
    R = zeros(n);
    for j = 1:n
        if j > 1
            R(1:j-1,j) = Q(:,1:j-1)' * Q(:,j);
            Q(:,j) = Q(:,j) - Q(:,1:j-1)*R(1:j-1,j);
        end
        R(j,j) = norm(Q(:,j));
        Q(:,j) = Q(:,j)/R(j,j);
    end
end
```

Visualization of Matrix Norms

```
function normA = visMatrixNorms(A, p)
% VISMATRIXNORM visMatrixNorms(A, p)
% Generate plots of unit vectors (in p-norm) and their images under A.
% Also approximates the matrix p-norm of A according to the definition of
% induced norms.
%
% Input:
%   A       2-by-2 matrix
%   p       1, 2, or Inf
% Output:
%  normA    matrix p-norm of A (approximation)
    if size(A,1)~=2 || size(A,2)~=2
        error('A must be a 2-by-2 matrix.')
    elseif p~=1 && p~=2 && p~=Inf
        error('p must be either 1, 2, or Inf.')
    end


    theta = linspace(0, 2*pi, 361);
    U = [cos(theta); sin(theta)];   % unit circle

    % Calculate p-norm of vectors on the unit circle
    if p == 1
        norm_U = sum( abs(U), 1 );
    elseif p == 2
        norm_U = sqrt( sum(abs(U).^2, 1) );
    else
        norm_U = max( abs(U), [], 1 );
    end

    % Normalize to unit vectors in respective norm
    X = U ./ norm_U;

    % Calculate y's
    Y = A*X;

    % Calculate p-norm of y's
    if p == 1
        norm_Y = sum( abs(Y), 1 );
    elseif p == 2
        norm_Y = sqrt( sum(abs(Y).^2, 1) );
    else
        norm_Y = max( abs(Y), [], 1 );
    end
    normA = max(norm_Y);

    % Plotting routine
    clf
    subplot(2,2,1)
```

```
    plot(X(1,:), X(2,:)), axis equal

    if p ~= Inf
        str = sprintf('x: unit vectors in %g-norm', p);
    else
        str = sprintf('x: unit vectors in \\infty-norm', 'Interpreter', 'latex');
    end
    title(str)

    subplot(2,2,2)
    plot(Y(1,:), Y(2,:)), axis equal
    title('Ax: image of unit vectors under A')

    subplot(2,1,2)
    plot(theta, norm_Y), axis tight
    xlabel('\theta')
    ylabel('|| y ||')

    fprintf(' p = %g\n', p)
    fprintf(' approx. norm: %18.16f\n', normA)
    fprintf(' actual  norm: %18.16f\n', norm(A, p))
end
```

Updated on March 8, 2021

Math 3607: Homework 7

Due: 11:59PM, Monday, March 15, 2021

TOTAL: 20 points

1. (**FNC** 4.1.4) A basic type of investment is an annuity: One makes monthly deposits of size P for n months at a fixed annual interest rate r, and at maturity collects the amount

$$\frac{12P}{r}\left(\left(1+\frac{r}{12}\right)^n - 1\right).$$

   Say you want to create an annuity for a term of 300 months and final value of $1,000,000. Using `fzero`, make a table of the interst rate you will need to get for each of the different contribution values $P = 500, 550, \ldots, 1000$.

2. (**FNC** 4.1.6) Lambert's W function is defined as the inverse of xe^x. That is, $y = W(x)$ if and only if $x = ye^y$. Write a function `y = lambertW(x)` that computes W using `fzero`. Make a plot of $W(x)$ for $0 \leqslant x \leqslant 4$.

3. (Adapted from **FNC** 4.2.1 and 4.2.2.) In each case below,

   - $g(x) = \frac{1}{2}\left(x + \frac{9}{x}\right)$, $r = 3$.
   - $g(x) = \pi + \frac{1}{4}\sin(x)$, $r = \pi$.
   - $g(x) = x + 1 - \tan(x/4)$, $r = \pi$.

   (a) (*by hand*) Show that the given $g(x)$ has a fixed point at the given r and that fixed point iteration can converge to it.

   (b) (*computer*) Apply fixed point iteration in MATLAB and use a log-linear graph (using `semilogy`) of the error to verify linear convergence. Then use numerical values of the error to determine an approximate value for the rate σ (see Lecture 22).

4. Answer the following questions *by hand*, without using MATLAB.

   (a) Discuss what happens when Newton's method is applied to find a root of

$$f(x) = \operatorname{sign}(x)\sqrt{|x|},$$

   starting at $x_0 \neq 0$. [1]

   (b) In the case of a multiple root, where $f(r) = f'(r) = 0$, the derivation of the quadratic error convergence is invalid. Redo the derivation to show that in this circumstance and with $f''(r) \neq 0$ the error converges only linearly.

[1] $\operatorname{sign}(x)$ is 1 if $x > 0$, -1 if $x < 0$, and 0 if $x = 0$.

5. (**FNC** 4.5.5) Suppose one wants to find the points on the ellipsoid $x^2/25 + y^2/16 + z^2/9 = 1$ that are closest to and farthest from the point $(5, 4, 3)$. The method of Lagrange multipliers implies that any such point satisfies

$$\begin{aligned} x - 5 &= \frac{\lambda x}{25}, \\ y - 4 &= \frac{\lambda y}{16}, \\ z - 3 &= \frac{\lambda z}{9}, \\ 1 &= \frac{1}{25}x^2 + \frac{1}{16}y^2 + \frac{1}{9}z^2 \end{aligned}$$

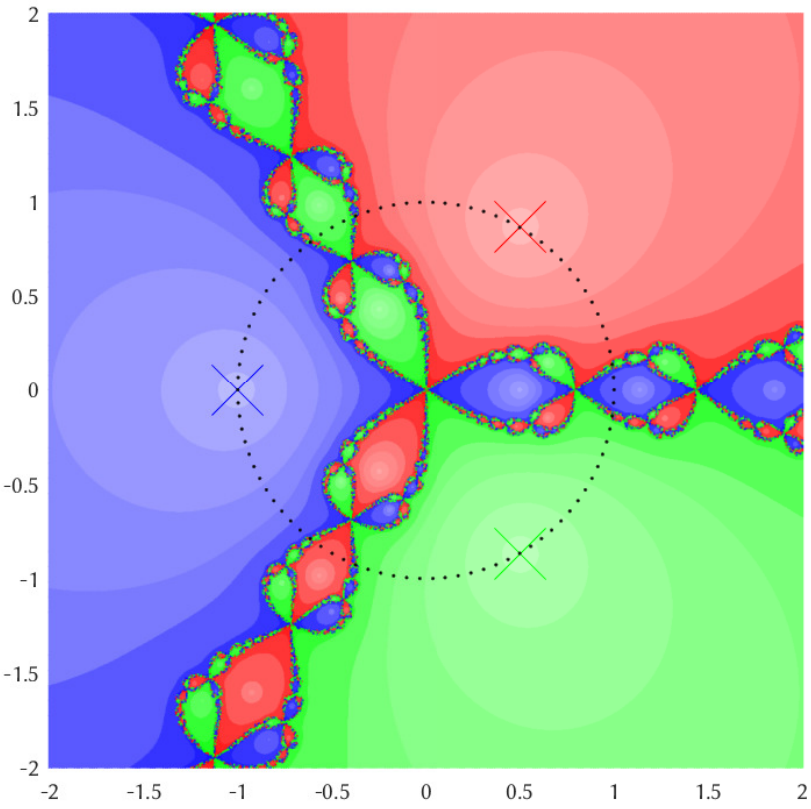
for an unknown value of λ.

   (a) (*by hand*) Write out this system in the form $\mathbf{f}(\mathbf{u}) = \mathbf{0}$.

   (b) (*by hand*) Write out the Jacobian matrix of this system.

   (c) (*computer*) Use `newtonsys` from class with different initial guesses to find the two roots of this system. Which is the closest point to $(5, 4, 3)$ and which is the farthest?

6. (Optional) Do **LM** 13.1–33 and 34. This is a long problem. The final outcome of the lengthy process is the colorful representation of so-called the *basin of attraction* as shown below.



Homework 7 (Solution)

Math 3607

Tae Eun Kim

Table of Contents


```
clear, close all, format short
```

Problem 1 (FNC 4.1.4)

Let's write a script solving the problem for a single P value:

```
P = 500;
n = 300;
FV = 1e6;  % value at maturity
```

We need to find an interest rate r which satisfies

$$\frac{12P}{r}\left(\left(1+\frac{r}{12}\right)^n - 1\right) = 1,000,000.$$

In other words, r is a root of

$$f(r) = \frac{12P}{r}\left(\left(1+\frac{r}{12}\right)^n - 1\right) - 1,000,000.$$

```
f = @(r) 12*P/r*( (1+r/12)^n - 1 ) - FV; % objective function
r = fzero(f, 0.01)                       % use 0.01 as an initial guess
```

```
r = 0.1235
```

Now we carry out the same computation for $P = 500, 550, \ldots 1000$, while all other parameters are fixed.

```
n = 300;
```

```
FV = 1e6;
P = 500:50:1000;
for j = 1:length(P)
    f = @(r) 12*P(j)/r*( (1+r/12)^n - 1 ) - FV;
    r = fzero(f, 0.01);
    if j == 1
        fprintf(' %4s %8s\n', 'P', 'r')
        fprintf(' %13s\n', repmat('-', 1, 13))
    end
    fprintf(' %4d %8.4f\n', P(j), r)
end
```

```
    P        r
 -------------
  500   0.1235
  550   0.1181
  600   0.1132
  650   0.1086
  700   0.1043
  750   0.1003
  800   0.0965
  850   0.0929
  900   0.0895
  950   0.0862
 1000   0.0831
```

Problem 2 (FNC 4.1.6, Lambert's W function)

Since $y = W(x)$ iff $x = ye^y$, y is a root of $f(y) = x - ye^y$ for a given x, which can be found using `fzero` as follows:

```
% if x is stored
y = fzero(@(y) x - y*exp(y), 1); % use 1 as an initial guess
```

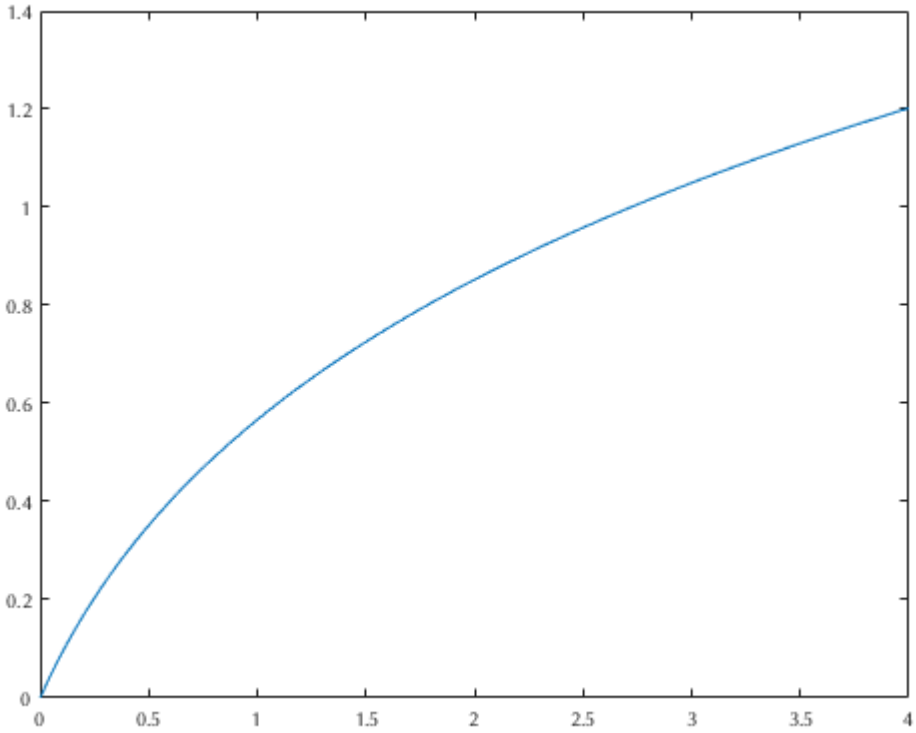
Thus we can write a MATLAB function which computes $W(x)$ by

```
function y = lambertW(x)
% LAMBERT y = lambertW(x)
% Evaluate Lambert's W function y = W(x)
% by solving x = y*exp(y).
%
% Input:
%   x    an array input
% Output:
%   y    W(x)
    y = zeros(size(x));
    for j = 1:numel(x)
        y(j) = fzero( @(y) x(j) - y*exp(y), 1 );
    end
end
```

Note that the function can take an array input `x` and produce corresponding values in an array `y` of the same dimensions as `x`. So we can use this function just as any other built-in mathematical functions, say to plot its graph:

```
x = linspace(0, 4, 100);
```

```
plot(x, lambertW(x))
```



Note. MATLAB actually comes with this function; it is named `lambertw`. Let's compare:

```
norm( lambertW(x) - lambertw(x) )
```

```
ans = 7.3983e-16
```

This confirms that our code works very nicely!

Under the hood, MATLAB's `lambertw` uses a very fast rootfinding algorithm called *Halley's method*, which exhibits the cubic convergence! Unlike Newton or secant method which uses a linear model, Halley uses a Padé approximation (linear-over-linear rational function) to generate iterates. Take a look at the source code by:

```
type lambertw.m
```

Problem 3 (FNC 4.2.1 and 2)

(a) For easy distinction, I will denote the three functions by g_1, g_2, and g_3. We confirm that the given r is a fixed point by showing $g(r) = r$.

- $g_1(3) = \frac{1}{2}\left(3 + \frac{9}{3}\right) = 3.$

- $g_2(\pi) = \pi + \frac{1}{4}\sin(\pi) = \pi$.
- $g_3(\pi) = \pi + 1 - \tan(\pi/4) = \pi + 1 - 1 = \pi$.

Fixed point iteration converges when $|g_j'(r)| < 1$.

- $g_1'(x) = \frac{1}{2}\left(1 - \frac{9}{x^2}\right) \implies g_1'(3) = 0$ (converge)
- $g_2'(x) = \frac{1}{4}\cos(x) \implies g_2'(\pi) = -\frac{1}{4}$ (converge)
- $g_3'(x) = 1 - \frac{1}{4}\sec^2(x/4) \implies g_3'(\pi) = 1 - \frac{1}{2} = \frac{1}{2}$ (converge)

(b) Begin by defining g_j as anonymous functions

```
g1 = @(x) (x + 9/x)/2;
g2 = @(x) pi + sin(x)/4;
g3 = @(x) x + 1 - tan(x/4);
```

and the noted fixed points:

```
r1 = 3;
r2 = pi;
r3 = pi;
```

Inspired by the function `fpi` from Lecture 22 and the examples from the accompanying live script, we write the following helper function:

```
function x = myfpi(g, x0, n)
% Generates fixed point iterates x_0, x_1, ..., x_{n-1}.
% All iterates are stored in a single (column) vector x.
    x = zeros(n, 1);
    x(1) = x0;
    for k = 1:n-1
        x(k+1) = g(x(k));
    end
end
```

(This function is also included at the end of this file.)

Let's study the sequence $x_0, x_1, \ldots, x_{14}$ generated by $x_{k+1} = g_1(x_k)$:

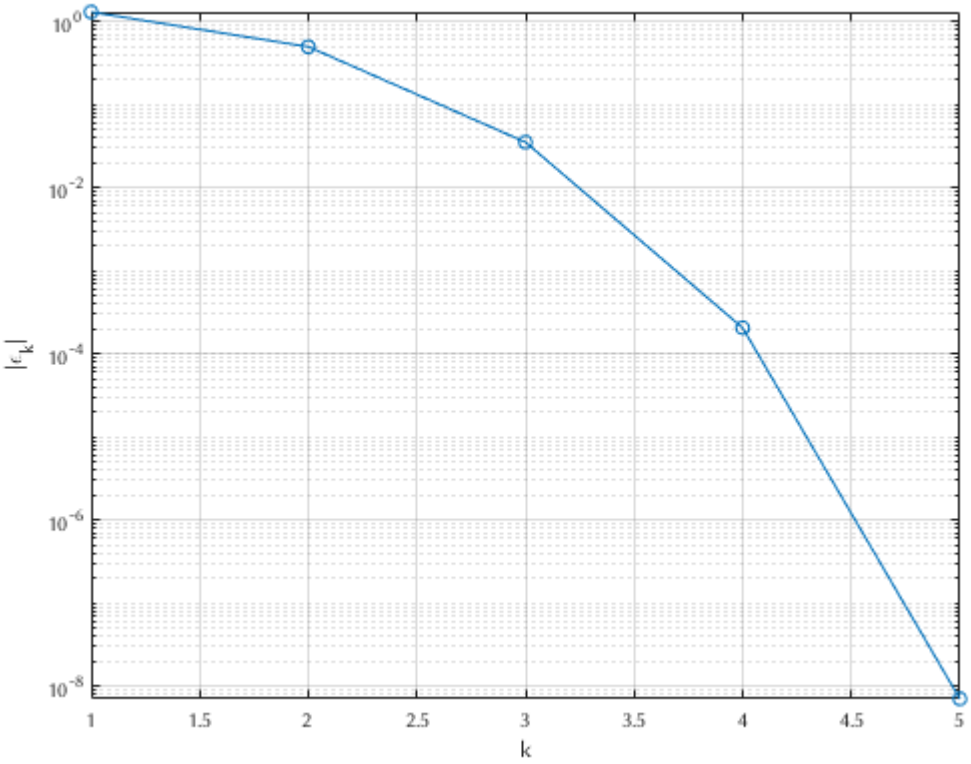
```
format long e
n = 10;
x = myfpi(g1, 1.7, n)
```

```
x = 10×1
     1.700000000000000e+00
     3.497058823529412e+00
```

```
3.035325038341661e+00
3.000205555964860e+00
3.000000007041726e+00
3.000000000000000e+00
3.000000000000000e+00
3.000000000000000e+00
3.000000000000000e+00
3.000000000000000e+00
```

This looks good. Let's analyze the errors.

```
err = abs(x - r1);
clf
semilogy(err, 'o-'), axis tight, grid on
xlabel('k'), ylabel('|\epsilon_k|')
```



Wait, this is faster than linear convergence? Yes, but for a good reason. We discovered in the previous part that $g_1'(3) = 0$ which, according to the FPI convergence theorem, implies that

$$\lim_{k\to\infty} \frac{|\epsilon_{k+1}|}{|\epsilon_k|} = |g'(3)| = 0, \text{ superlinear convergence!}$$

Unfortunately, this is difficult to confirm as the denominator quickly underflows (to zero):

```
err(2:end) ./ err(1:end-1)
```

```
ans = 9×1
```

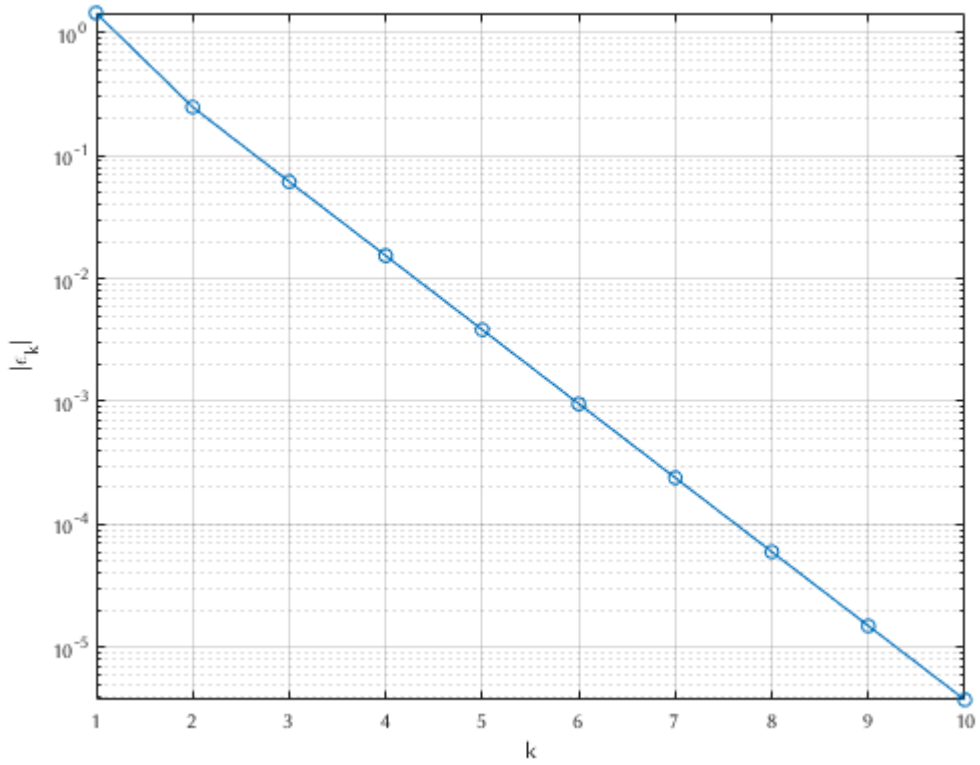
```
     3.823529411764706e-01
     7.106812447434803e-02
     5.818987735329192e-03
     3.425697871600165e-05
                         0
                       NaN
                       NaN
                       NaN
                       NaN
```

Moving onto g_2:

```
format long e
n = 10;
x = myfpi(g2, 1.7, n)
```

```
x = 10×1
     1.700000000000000e+00
     3.389508856202910e+00
     3.080246551988983e+00
     3.156919561362624e+00
     3.137761076665939e+00
     3.142550545476954e+00
     3.141353180654625e+00
     3.141652521823013e+00
     3.141577686531497e+00
     3.141596395354367e+00
```

```
err = abs(x - r2);
clf
semilogy(err, 'o-'), axis tight, grid on
xlabel('k'), ylabel('|\epsilon_k|')
```



In this case, we see that the errors draw a nice straight line on the log-linear graph. Furthermore, the ratios of errors converge beautifully to the expected $\sigma = |g_2'(\pi)| = 1/4$:

```
err(2:end) ./ err(1:end-1)
```

```
ans = 9×1
     1.719738249191035e-01
     2.474469234128392e-01
     2.498432234955372e-01
     2.499902120273960e-01
     2.499993882928684e-01
     2.499999617684353e-01
     2.499999976100850e-01
     2.499999998479356e-01
     2.499999999851645e-01
```

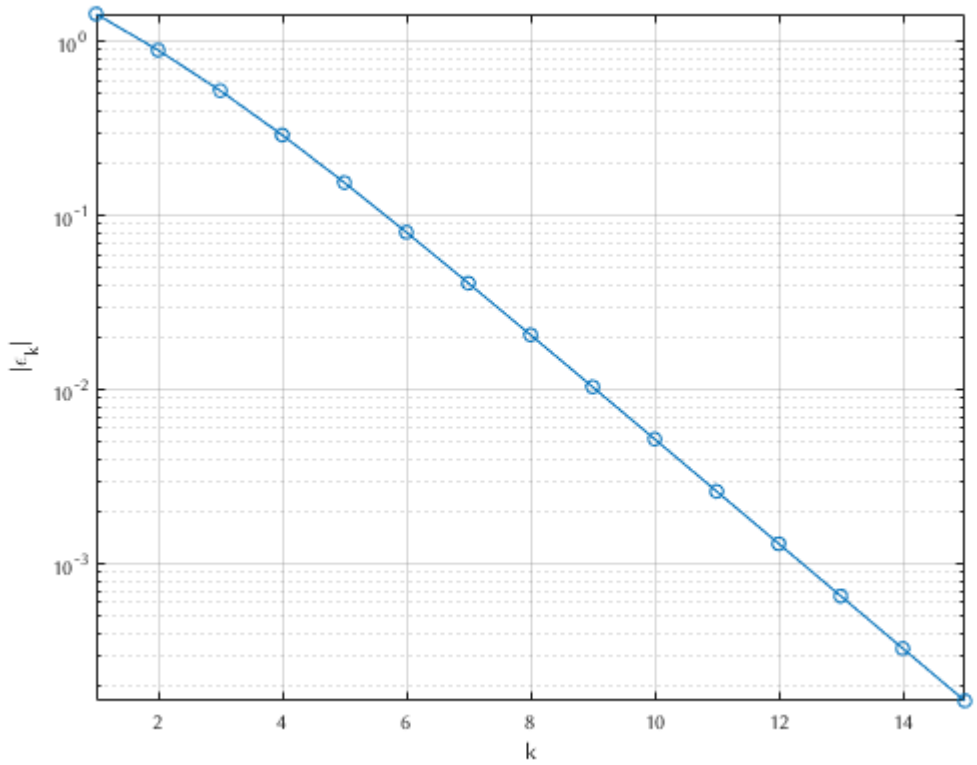
Lastly for g_3:

```
format long e
n = 15;
x = myfpi(g3, 1.7, n)
```

```
x = 15×1
     1.700000000000000e+00
     2.247416847573013e+00
     2.617881264257347e+00
     2.850599005225683e+00
     2.986452451970267e+00
     3.061162389406461e+00
```

```
    3.100590041247924e+00
    3.120884031572326e+00
    3.131185104358556e+00
    3.136375386158261e+00
    ⋮
```

```
err = abs(x - r3);
clf
semilogy(err, 'o-'), axis tight, grid on
xlabel('k'), ylabel('|\epsilon_k|')
```



We also see that the errors draw a nice straight line on the log-linear graph. Furthermore, the ratios of errors converge beautifully to the expected $\sigma = |g_3'(\pi)| = 1/2$:

```
err(7:end) ./ err(6:end-1)
```

```
ans = 9×1
    5.097908450034160e-01
    5.050561619051057e-01
    5.025708240006618e-01
    5.012964450721191e-01
    5.006510261133111e-01
    5.003262197391994e-01
    5.001632872715480e-01
    5.000816880784664e-01
    5.000408551610744e-01
```

Note that convergence is not as fast as in the previous case (even though both converge linearly) because the convergence rate $\sigma = 1/2$ for this case is larger than the previous one.

Problem 4

(a)

Simple Calculus 1 exercise shows that

$$f'(x) = \frac{1}{2\sqrt{|x|}}, \text{ for all } x \in \mathbb{R}.$$

One Newton iteration takes any nonzero initial iterate $x_0 \neq 0$ to

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - 2\text{sign}(x_0)|x_0| = -x_0, \text{ regardless of the sign of } x_0.$$

If you are unsure, examine two cases, $x_0 > 0$ and $x_0 < 0$, separately. Repeating the same computation with x_0 replaced by $-x_0$, we find that

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = -x_0 - 2\text{sign}(-x_0)|-x_0| = x_0.$$

Back to the starting point! This means that the iterates generated by the Newton's iteration formula will just rock back and forth between x_0 and $-x_0$:

$$x_0, -x_0, x_0, -x_0, \ldots$$

a hopeless divergence scenario.

The following (minimal) Newton iteration code confirms our prediction.

```
f = @(x) sign(x).*sqrt(abs(x));
fprime = @(x) 1./(2*sqrt(abs(x)));
x = 1;
for k = 1:10
    x = x - f(x)/fprime(x)
end
```

```
x =
    -1
x =
     1
x =
    -1
x =
     1
x =
    -1
x =
     1
x =
```

```
    -1
x =
     1
x =
    -1
x =
     1
```

Ponder. Why do you think it is happening? Is it violating the convergence theorem for Newton's method?

(b)

Set-up

Let $\{x_k\}$ be iterates generated by Newton's iteration formula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad (*)$$

Let r be a double root of f, that is,

$$f(r) = f'(r) = 0 \quad \text{but} \quad f''(r) \neq 0.$$

Let $\epsilon_k = x_k - r$ as in lecture.

Substitute $x_k = r + \epsilon_k$ into $(*)$

$$\epsilon_{k+1} = \epsilon_k - \frac{f(r+\epsilon_k)}{f'(r+\epsilon_k)}$$

Taylor-expand at r

$$\epsilon_{k+1} = \epsilon_k - \frac{f(r) + f'(r)\epsilon_k + \frac{f''(r)}{2}\epsilon_k^2 + \frac{f'''(r)}{6}\epsilon_k^3 + O(\epsilon_k^4)}{f'(r) + f''(r)\epsilon_k + \frac{f'''(r)}{2}\epsilon_k^2 + O(\epsilon_k^3)}$$

by assumption

$$= \epsilon_k - \frac{\frac{f''(r)}{2}\epsilon_k^2\left(1 + \frac{1}{3}\frac{f'''(r)}{f''(r)}\epsilon_k + O(\epsilon_k^2)\right)}{f''(r)\epsilon_k\left(1 + \frac{1}{2}\frac{f'''(r)}{f''(r)}\epsilon_k + O(\epsilon_k^2)\right)}$$

Geometric series

$$= \epsilon_k - \frac{1}{2}\epsilon_k\left(1 + \frac{1}{3}\frac{f'''(r)}{f''(r)}\epsilon_k + O(\epsilon_k^2)\right)\left(1 - \frac{1}{2}\frac{f'''(r)}{f''(r)}\epsilon_k + O(\epsilon_k^2)\right)$$

$$= \frac{1}{2}\epsilon_k + \frac{1}{12}\frac{f'''(r)}{f''(r)}\epsilon_k^2 + O(\epsilon_k^3) = \frac{1}{2}\epsilon_k + O(\epsilon_k^2)$$

Conclusion $\epsilon_{k+1} \approx \frac{1}{2}\epsilon_k$, linear convergence!

Problem 5 (FNC 4.5.5)

(a)

Moving all terms to one side and simplifying, we have

$$\begin{aligned}
\left(1 - \frac{\lambda}{25}\right)x - 5 &= 0 \\
\left(1 - \frac{\lambda}{16}\right)y - 4 &= 0 \\
\left(1 - \frac{\lambda}{9}\right)z - 3 &= 0 \\
\frac{1}{25}x^2 + \frac{1}{16}y^2 + \frac{1}{9}z^2 - 1 &= 0
\end{aligned}$$

This is good enough. But for the last part, it is advantageous to write in the form $\mathbf{f}(\mathbf{u}) = \mathbf{0}$ as suggested by the problem. Let $\mathbf{u} = (u_1, u_2, u_3, u_4)^{\mathrm{T}} = (x, y, z, \lambda)^{\mathrm{T}}$, the vector consisting of all four unknowns. Then, abstractly, we may view/write the system of these four equations as a single vector equation $\mathbf{f}(\mathbf{u}) = \mathbf{0}$,

$$\mathbf{f}(\mathbf{u}) = \begin{bmatrix} f_1(u_1, u_2, u_3, u_4) \\ f_2(u_1, u_2, u_3, u_4) \\ f_3(u_1, u_2, u_3, u_4) \\ f_4(u_1, u_2, u_3, u_4) \end{bmatrix} = \begin{bmatrix} (1 - u_4/25)u_1 - 5 \\ (1 - u_4/16)u_2 - 4 \\ (1 - u_4/9)u_3 - 3 \\ u_1^2/25 + u_2^2/16 + u_3^2/9 - 1 \end{bmatrix}$$

simply by replacing x, y, z, λ by u_1, u_2, u_3, u_4, respectively.

(b)

In terms of the original variables x, y, z, λ, we can express the Jacobian matrix as

$$\mathbf{J}(x, y, z, \lambda) = \begin{bmatrix} 1 - \lambda/25 & 0 & 0 & -x/25 \\ 0 & 1 - \lambda/16 & 0 & -y/16 \\ 0 & 0 & 1 - \lambda/9 & -z/9 \\ 2x/25 & 2y/16 & 2z/9 & 0 \end{bmatrix}$$

which can be rewritten in terms of $\mathbf{u} = (u_1, u_2, u_3, u_4)^{\mathrm{T}} = (x, y, z, \lambda)^{\mathrm{T}}$ as

$$\mathbf{J}(\mathbf{u}) = \begin{bmatrix} 1 - u_4/25 & 0 & 0 & -u_1/25 \\ 0 & 1 - u_4/16 & 0 & -u_2/16 \\ 0 & 0 & 1 - u_4/9 & -u_3/9 \\ 2u_1/25 & 2u_2/16 & 2u_3/9 & 0 \end{bmatrix}.$$

The latter will be useful in the next part.

(c)

In order to use `newtonsys` to find roots of $\mathbf{f}$, we first need to write a function m-file calculating both $\mathbf{f}$ and $\mathbf{J}$. Inspired by the example in the live script accompanying Lecture 23, we write

```
function [f,J] = nlsystem(x)
    f = [ (1-u(4)/25)*u(1) - 5;
          (1-u(4)/16)*u(2) - 4;
          (1-u(4)/9)*u(3) - 3;
          u(1)^2/25 + u(2)^2/16 + u(3)^2/9 - 1];
    J = [ 1-u(4)/25, 0, 0, -u(1)/25;
          0, 1-u(4)/16, 0, -u(2)/16;
          0, 0, 1-u(4)/9,  -u(3)/9;
          2*u(1)/25, 2*u(2)/16, 2*u(3)/9, 0];
end
```

(This function is included at the end of this live script.)

Then we are able to use `newtonsys` as follows.

```
format short
x1 = newtonsys(@nlsystem, [1 1 1 1])
```

```
x1 = 4×7
    1.0000    4.6945    3.5843    3.4311    3.4241    3.4241    3.4241
    1.0000    3.4446    2.4818    2.3311    2.3268    2.3268    2.3268
    1.0000    1.8336    1.4283    1.3186    1.3167    1.3167    1.3167
    1.0000  -11.3310  -10.2192  -11.3800  -11.5053  -11.5056  -11.5056
```

Note that the name of the function must be passed with at-sign in its front since it is defined as an m-file. (If you defined a function in-line as an anonymous function, the at-sign is unnecessary.)

To find another solution, use a different initial guess.

```
x2 = newtonsys(@nlsystem, [-1 -1 -1 1])
```

```
x2 = 4×10
   -1.0000    2.1306   -0.8437  -11.7432   -6.6115   -4.7672   -4.4169   -4.4038 ···
   -1.0000   -0.6577   -1.5221   -3.5260   -2.5298   -1.8132   -1.7221   -1.7119
   -1.0000   -5.8583   -3.8719   -1.5987   -0.7062   -0.6950   -0.6045   -0.6084
    1.0000   74.8662   35.9427   31.7917   38.6124   47.7038   52.8890   53.3832
```

Check. Are the solutions really on the ellipsoid?

```
eq_ellips = @(u) u(1)^2/25 + u(2)^2/16 + u(3)^2/9 - 1;
p1 = x1(1:3,end);
p2 = x2(1:3,end);
format short e
[eq_ellips(p1), eq_ellips(p2)]'
```

```
ans = 2×1
            0
   2.2204e-16
```

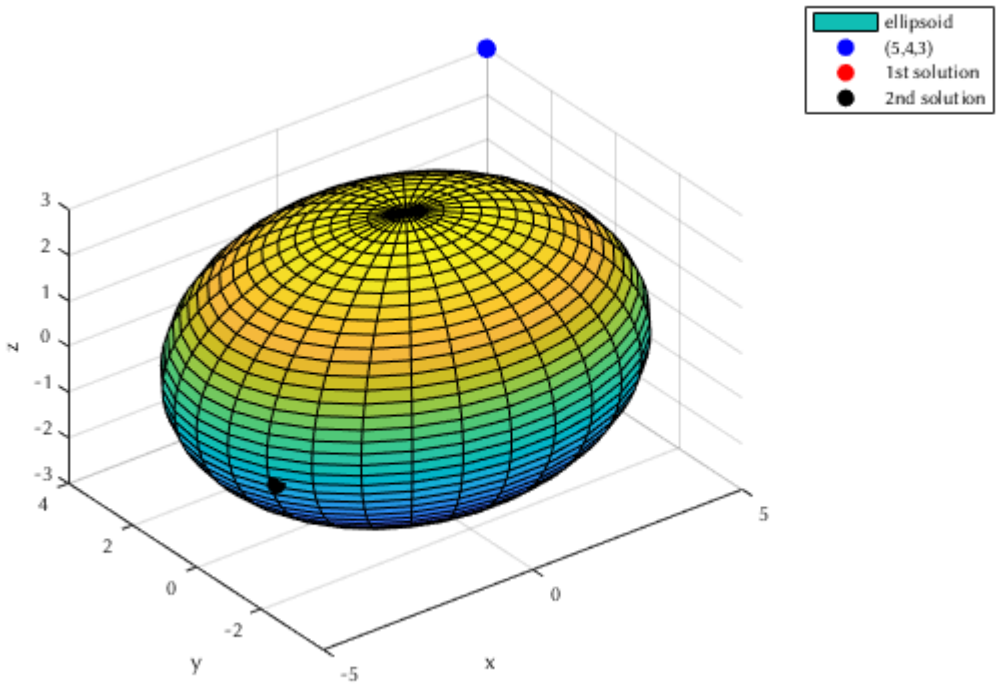
Yes, they are!

Closest or Farthest?

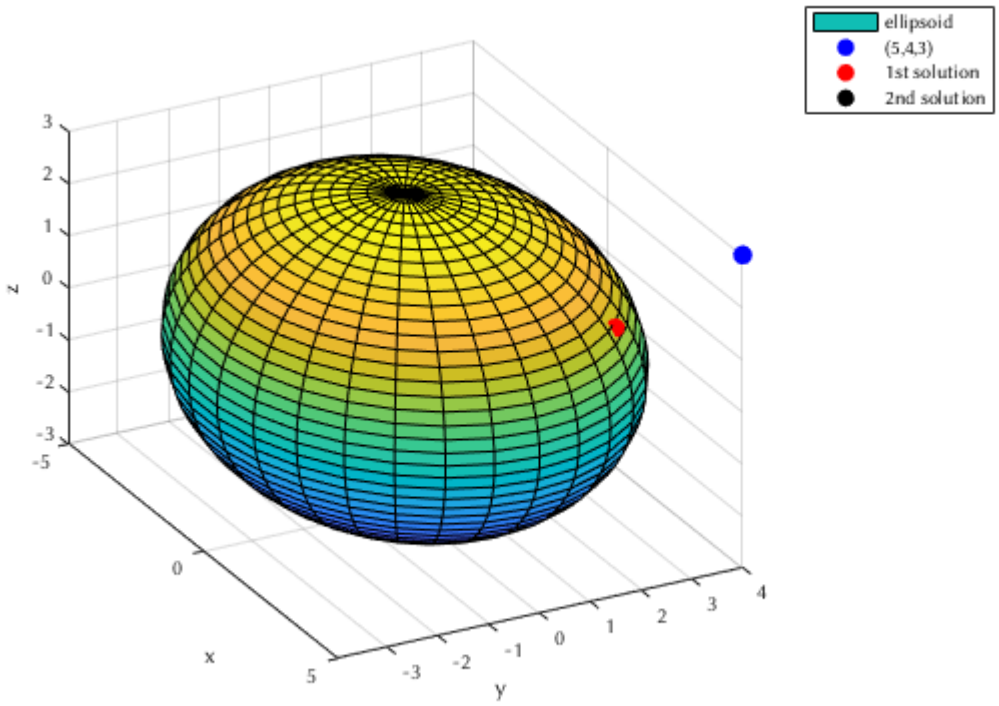
One of the two solutions have all positive components while the other has all negative components. The former should be closest while the latter farthest, because the given point $(5,4,3)$ lies in the first octant.

Run the following script and rotate the generated 3-D figure around to stop the closest and the farthest points.

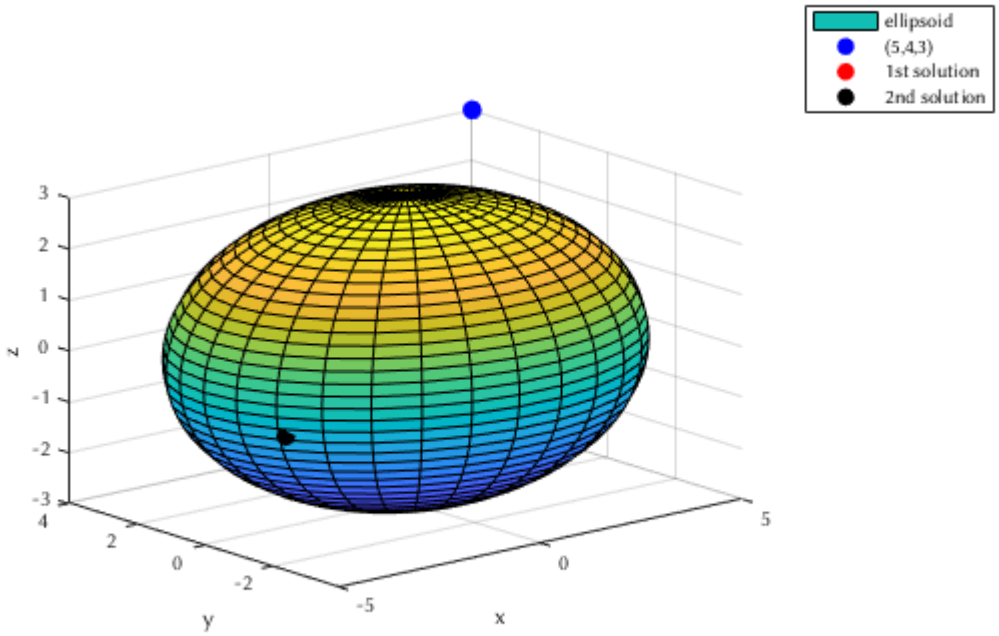
```
nr_th = 31;
nr_ph = 41;
th = linspace(0, 2*pi, nr_th);
ph = linspace(0, pi, nr_ph);
[TH,PH] = meshgrid(th, ph);
a = 5; b = 4; c = 3;
x = a*sin(PH).*cos(TH);
y = b*sin(PH).*sin(TH);
z = c*cos(PH);
clf
surf(x,y,z), hold on, axis equal
xlabel('x'), ylabel('y'), zlabel('z')
plot3(5,4,3, 'b.', 'MarkerSize', 30)
plot3(p1(1), p1(2), p1(3), 'r.', 'MarkerSize', 30)
plot3(p2(1), p2(2), p2(3), 'k.', 'MarkerSize', 30)
legend('ellipsoid', '(5,4,3)', '1st solution', '2nd solution')
```




```
view([62 25])    % better viewing angle for the first solution
```



```
view([-40 15])  % better viewing angle for the second solution
```



Functions Used

Lambert's W function

```
function y = lambertW(x)
% LAMBERT y = lambertW(x)
% Evaluate Lambert's W function y = W(x)
% by solving x = y*exp(y).
%
% Input:
%   x    an array input
% Output:
%   y    W(x)
    y = zeros(size(x));
    for j = 1:numel(x)
        y(j) = fzero( @(y) x(j) - y*exp(y), 1 );
    end
end
```

Fixed Point Iteration

```
function x = myfpi(g, x0, n)
```

```
% Generates fixed point iterates x_0, x_1, ..., x_{n-1}.
% All iterates are stored in a single (column) vector x.
    x = zeros(n, 1);
    x(1) = x0;
    for k = 1:n-1
        x(k+1) = g(x(k));
    end
end
```

Newton Iteration for Systems

```
function x = newtonsys(f,x1)
% NEWTONSYS   Newton's method for a system of equations.
% Input:
%   f        function that computes residual and Jacobian matrix
%   x1       initial root approximation (n-vector)
% Output
%   x        array of approximations (one per column, last is best)

% Operating parameters.
    funtol = 1000*eps;  xtol = 1000*eps;  maxiter = 40;

    x = x1(:);
    [y,J] = f(x1);
    dx = Inf;
    k = 1;

    while (norm(dx) > xtol) && (norm(y) > funtol) && (k < maxiter)
        dx = -(J\y);    % Newton step
        x(:,k+1) = x(:,k) + dx;

        k = k+1;
        [y,J] = f(x(:,k));
    end

    if k==maxiter, warning('Maximum number of iterations reached.'), end
end
```

Residual and Jacobian function for #5

```
function [f,J] = nlsystem(u)
    f = [ (1-u(4)/25)*u(1) - 5;
          (1-u(4)/16)*u(2) - 4;
          (1-u(4)/9)*u(3) - 3;
          u(1)^2/25 + u(2)^2/16 + u(3)^2/9 - 1];
    J = [ 1-u(4)/25, 0, 0, -u(1)/25;
          0, 1-u(4)/16, 0, -u(2)/16;
          0, 0, 1-u(4)/9,  -u(3)/9;
          2*u(1)/25, 2*u(2)/16, 2*u(3)/9, 0];
end
```

Updated on March 15, 2021

Math 3607: Homework 8

Due: 11:59PM, Monday, March 22, 2021

TOTAL: 20 points

1. (**FNC** 5.1.3) The following two point sets define the top and bottom of a flying saucer shape:

   Top:

x	0	0.51	0.96	1.06	1.29	1.55	1.73	2.13	2.61
y	0	0.16	0.16	0.43	0.62	0.48	0.19	0.18	0

   Bottom:

x	0	0.58	1.04	1.25	1.56	1.76	2.19	2.61
y	0	-0.16	-0.15	-0.30	-0.29	-0.12	-0.12	0

   Use piecewise cubic interpolation to make a picture of the flying saucer.

2. (**FNC** 5.1.4) Define

   $$q(x) = \frac{a}{2}x(x-1) - b(x-1)(x+1) + \frac{c}{2}x(x+1).$$

   (a) Show that q is a polynomial interpolant of the points $(-1, a)$, $(0, b)$, $(1, c)$.

   (b) Use a change of variable to find a quadratic polynomial interpolant p for the points $(x_0 - h, a)$, (x_0, b), $(x_0 + h, c)$.

3. (**FNC** 5.3.5) Although the cardinal cubic splines are intractable in closed form, they can be found numerically. Each cardinal spline interpolates the data from one column of an identity matrix. Define the nodes $\mathbf{t} = [0, 0.075, 0.25, 0.55, 1]^{\mathrm{T}}$. Plot over $[0, 1]$ the five cardinal functions for this node set over the interval $[0, 1]$.

4. (Adapted from **FNC** 5.3.6.) Do this problem by hand. Suppose you were to define a piecewise quadratic spline that interpolates n given values and has a continuous first derivative. Follow the derivation presented in lecture to express all of the interpolation and continuity conditions. How many additional conditions are required to make a square system for the coefficients? Justify your answer.

5. (Cubic splines in 2-D) Do **LM** 12.2–15.

Homework 8 (Solution)

Math 3607

Tae Eun Kim

Table of Contents


```
clear, close all, format short
```

Problem 1 (FNC 5.1.3)

One may directly type in the given coordinates. In real applications, such information may be stored in data files, *e.g.,*

```
%% flying saucer: top shape
% filename: shape_top.dat
% x, y
0.00, 0.00
0.51, 0.16
0.96, 0.16
1.06, 0.43
1.29, 0.62
1.55, 0.48
1.73, 0.19
2.13, 0.18
2.61, 0.00
```

and

```
%% flying saucer: bottom shape
% filename: shape_bottom.dat
% x, y
0.00,  0.00
0.58, -0.16
1.04, -0.15
1.25, -0.30
1.56, -0.29
1.76, -0.12
2.19, -0.12
2.61,  0.00
```

If these files are saved in the same directory as your main live script, you can import the data using `load` function

```
load('shape_top.dat')
load('shape_bottom.dat')
```

The data files are now loaded and are stored as `shape_top` and `shape_bottom`:

```
shape_top
```

```
shape_top = 9×2
         0         0
    0.5100    0.1600
    0.9600    0.1600
    1.0600    0.4300
    1.2900    0.6200
    1.5500    0.4800
    1.7300    0.1900
    2.1300    0.1800
    2.6100         0
```

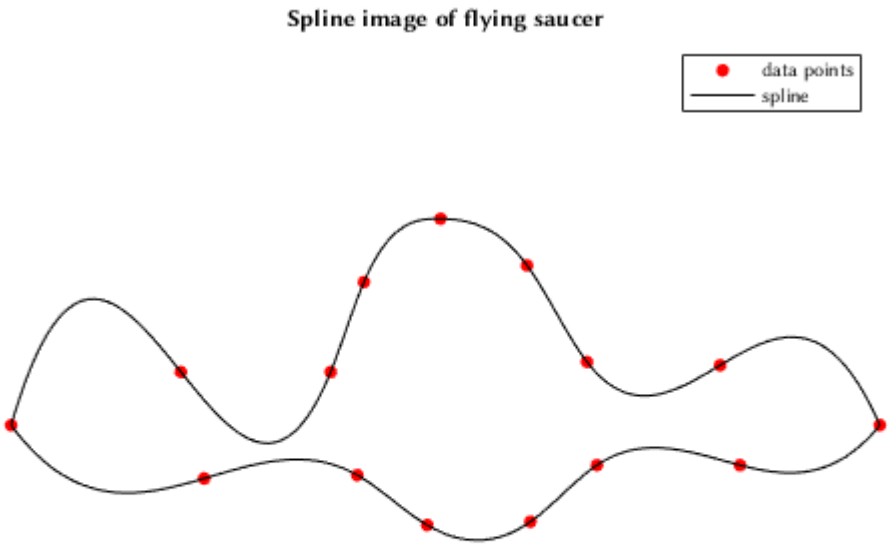
```
shape_bottom
```

```
shape_bottom = 8×2
         0         0
    0.5800   -0.1600
    1.0400   -0.1500
    1.2500   -0.3000
    1.5600   -0.2900
    1.7600   -0.1200
    2.1900   -0.1200
    2.6100         0
```

Version 1.

With the data-loading tip out of the way, let's go ahead and plot the saucer.

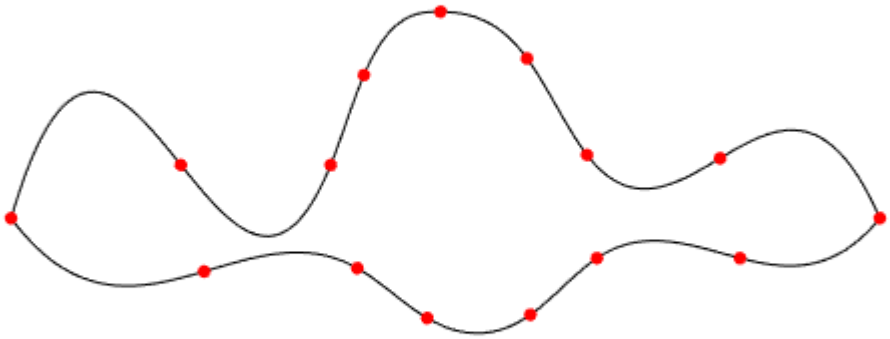
```
% top
xdp_t = shape_top(:,1);
ydp_t = shape_top(:,2);
x_t = linspace(xdp_t(1), xdp_t(end), 200)';
y_t = interp1(xdp_t, ydp_t, x_t, 'spline');

% bottom
xdp_b = shape_bottom(:,1);
ydp_b = shape_bottom(:,2);
x_b = linspace(xdp_b(1), xdp_b(end), 200)';
y_b = interp1(xdp_b, ydp_b, x_b, 'spline');

% plot
clf
plot([xdp_t; xdp_b], [ydp_t; ydp_b], 'r.', 'MarkerSize', 20), hold on
plot([x_t; flip(x_b)], [y_t; flip(y_b)], 'k')
axis equal, axis off
title('Spline image of flying saucer')
legend('data points', 'spline')
```



Version 2.

A cleaner version using a for-loop a new type of MATLAB array called *cells*:

```
DATA{1} = shape_top;          % DATA is a cell array.
DATA{2} = shape_bottom;
clf
for i = 1:2
    xdp = DATA{i}(:,1);
    ydp = DATA{i}(:,2);
    x = linspace(xdp(1), xdp(end), 200)';
    y = interp1(xdp, ydp, x, 'spline');
    plot(x, y, 'k'), hold on
    plot(xdp, ydp, 'r.', 'MarkerSize', 20)
    hold on
end
axis equal, axis off
```



If you learn more about cell arrays and structures, read Section 6.3.

Problem 2 (FNC 5.1.4)

Note that the function $q(x)$ is a quadratic polynomial:

$$q(x) = \frac{a}{2}x(x-1) - b(x-1)(x+1) + \frac{c}{2}x(x+1).$$

(a)

- $q(-1) = \frac{a}{2}(-1)(-2) - b(-2)(0) + \frac{c}{2}(-1)(0) = a.$
- $q(0) = \frac{a}{2}(0)(-1) - b(-1)(1) + \frac{c}{2}(0)(1) = b.$
- $q(1) = \frac{a}{2}(1)(0) - b(0)(2) + \frac{c}{2}(1)(2) = c.$

(b)

Let $X(z) = \frac{z - x_0}{h}$. Then

- $X(x_0) = 0$,
- $X(x_0 \pm h) = \pm 1$.

This provides a suitable change of variables: define p by

$$p(z) = q(X(z)) = \frac{a}{2}\frac{z - x_0}{h}\frac{z - x_0 - h}{h} - b\frac{z - x_0 - h}{h}\frac{z - x_0 + h}{h} + \frac{c}{2}\frac{z - x_0}{h}\frac{z - x_0 + h}{h}$$

which simplifies to

$$p(z) = \frac{1}{h^2}\left[\frac{a}{2}(z - x_0)(z - x_0 - h) - b(z - x_0 - h)(z - x_0 + h) + \frac{c}{2}(z - x_0)(z - x_0 + h)\right].$$

This is clearly a quadratic polynomial (as a composition of linear and quadratic polynomial). Furthermore, it interpolates all three points $(x_0 - h, a)$, (x_0, b), and $(x_0 + h, c)$ by construction. For instance,

$$p(x_0 - h) = q(X(x_0 - h)) = q(-1) = a,$$

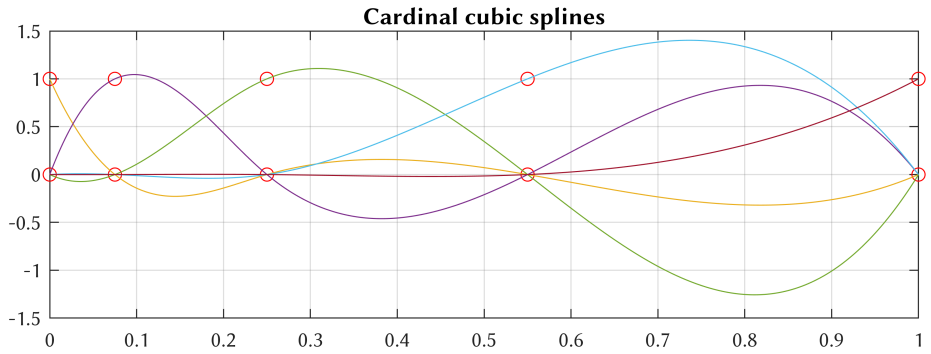
where the first equality follows from the definition $p(z) = q(X(z))$, the second due to the noted property of the map $X(z)$, and the last from the fact that q interpolates $(-1, a)$.

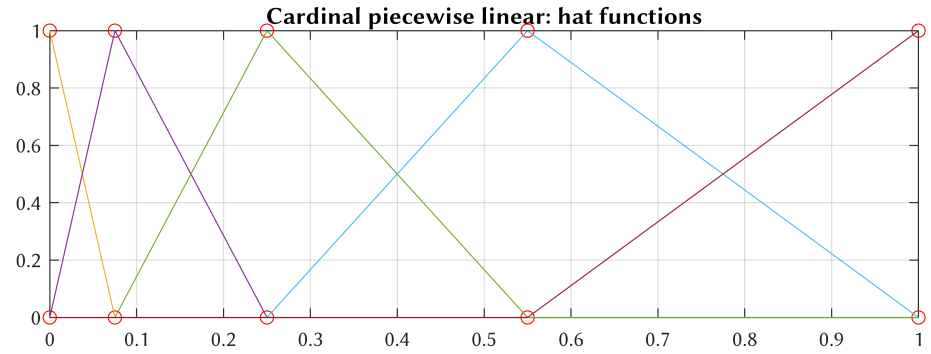
Problem 3 (FNC 5.3.5)

A slight modification of the example in the live script accompanying Lecture 25 would do. For the sake of comparison, cardinal piecewise linear interpolants, a.k.a. hat functions, are also plotted.

```
xdp = [0 0.075 0.25 0.55 1]';  % t is the nodes (x data points)
I = eye(length(xdp));          % identity matrix for easy access to cardinal data (y)
x = linspace(0, 1, 400)';
clf
subplot(2,1,1)
plot(xdp, zeros(size(xdp)), 'ro'), hold on
plot(xdp, ones(size(xdp)), 'ro')
for j = 1:length(xdp)
    plot(x, interp1(xdp, I(:,j), x, 'spline'))
end
grid on
title('Cardinal cubic splines')

subplot(2,1,2)
plot(xdp, zeros(size(xdp)), 'ro'), hold on
plot(xdp, ones(size(xdp)), 'ro')
```

```
for j = 1:length(xdp)
    plot(x, interp1(xdp, I(:,j), x))
end
grid on
title('Cardinal piecewise linear: hat functions')
```





Problem 4 (FNC 5.3.6)

Denote by $Q(x)$ the quadratic spline that interpolates (x_i, y_i), for $i = 1, \dots, n$, such that

$$Q(x) = q_i(x) = c_{i,1} + c_{i,2}(x - x_i) + c_{i,3}(x - x_i)^2 \text{ on } I_i \text{ for } i = 1, \dots, n-1$$

where $I_i = [x_i, x_{i+1})$ if $i = 1, \dots, n-2$ and $I_{n-1} = [x_{n-1}, x_n]$. So there are $3(n-1) = 3n-3$ coefficients to be determined.

The function Q must interpolate the given data:

$$Q(x_i) = y_i \text{ for } i = 1, \dots, n \qquad [\, n \text{ equations }]$$

Since Q must be $C^1[x_1, x_n]$, Q and Q' must be continuous at all interior breakpoints:

$$q_{i-1}(x_i) = q_i(x_i) \text{ for } i = 2, \dots, n-1 \qquad [\, n-1 \text{ equations }]$$

$$q'_{i-1}(x_i) = q'_i(x_i) \text{ for } i = 2, \dots, n-1 \qquad [\ n-1 \text{ equations }]$$

All together, there are $3n-2$ equations, that is, one condition short of the $3n-3$ conditions required. (Just in the case of cubic splines, there are various ways to impose this additional condition, *e.g.*, $Q'(x_1) = 0$ or $Q''(x_0) = 0$.

Problem 5 (LM 12.2--15; 2-D cubic splines)

Disclaimer: The solution below utiltizes parametrization by the arclength variable, not the pseudo-arclength variable. This is just to illustrate another way of solving the same problem. For a solution using the proper parametrization, please see p.1569 of LM or my tutorial video. (Any submission using arclength will not receive full credits since the problem specifically asked to use pseudo-arclength variable.)

(a)

Without loss of any generality, we will work with the unit circle $x^2 + y^2 = 1$, which can be parametrized by the arclength variable s as

$$\langle x(s), y(s) \rangle = \langle \cos(s), \sin(s) \rangle, \ s \in [0, 2\pi].$$

(Note that the parametrization by s is equivalent to the usual parametrization by θ because along the unit circle the length of the arc spanned by angle θ is equals θ.)

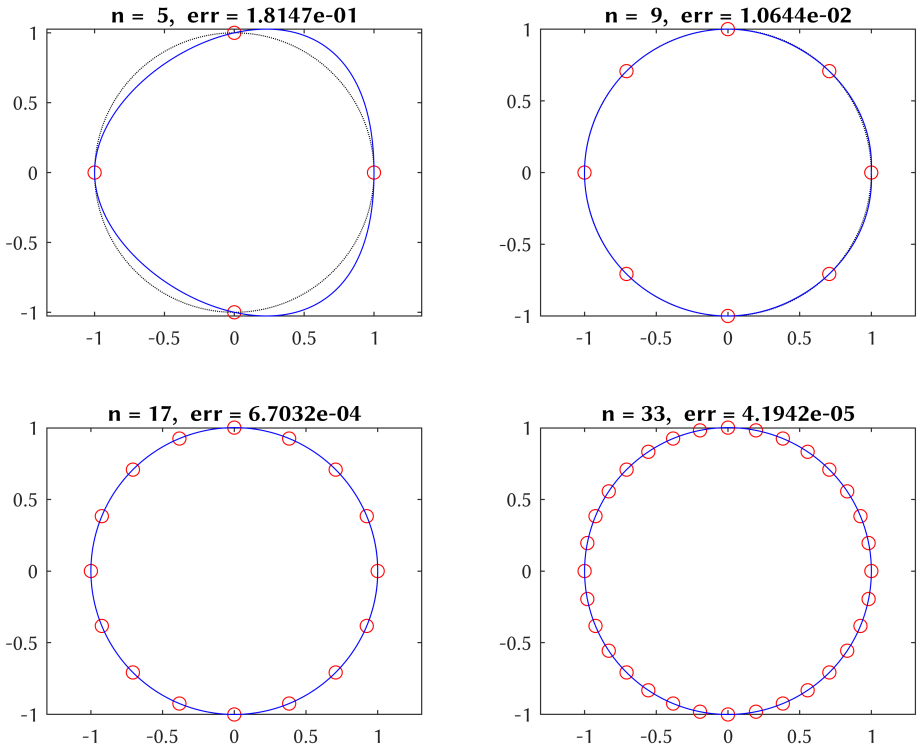
Thus we view x and y as separate variables dependent on the common variable s. Consequently, the data points on the unit circle must now be arranged into two data sets as

$$(s_j, x_j) = (s_j, \cos(s_j)) \text{ and } (s_j, y_j) = (s_j, \sin(s_j)) \text{ for } j = 1, \dots, n$$

where $0 = s_1 < s_2 < \cdots < s_n = 2\pi$.

```
nvals = [5 9 17 33];
clf
for j = 1:length(nvals)
    n = nvals(j);
    sdp = linspace(0, 2*pi, n)';    % nodes: arc length
    xdp = cos(sdp);                 % ordinates: x-coord.
    ydp = sin(sdp);                 % ordinates: y-coord.

    s = linspace(0, 2*pi, 1000)';   % query points
    xcirc = cos(s);                 % x-coord. on circle
    ycirc = sin(s);                 % y-coord. on circle
    x = interp1(sdp, xdp, s, 'spline');
    y = interp1(sdp, ydp, s, 'spline');

    err = max( sqrt((x-xcirc).^2 + (y-ycirc).^2) );
```

```
    subplot(2,2,j)
    plot(xdp, ydp, 'ro'), hold on    % plot data points
    plot(xcirc, ycirc, 'k:')         % plot circle
    plot(x, y, 'b')                  % plot spline
    axis equal
    title(sprintf('n = %2d,  err = %8.4e', n, err))
end
```



(a,i)

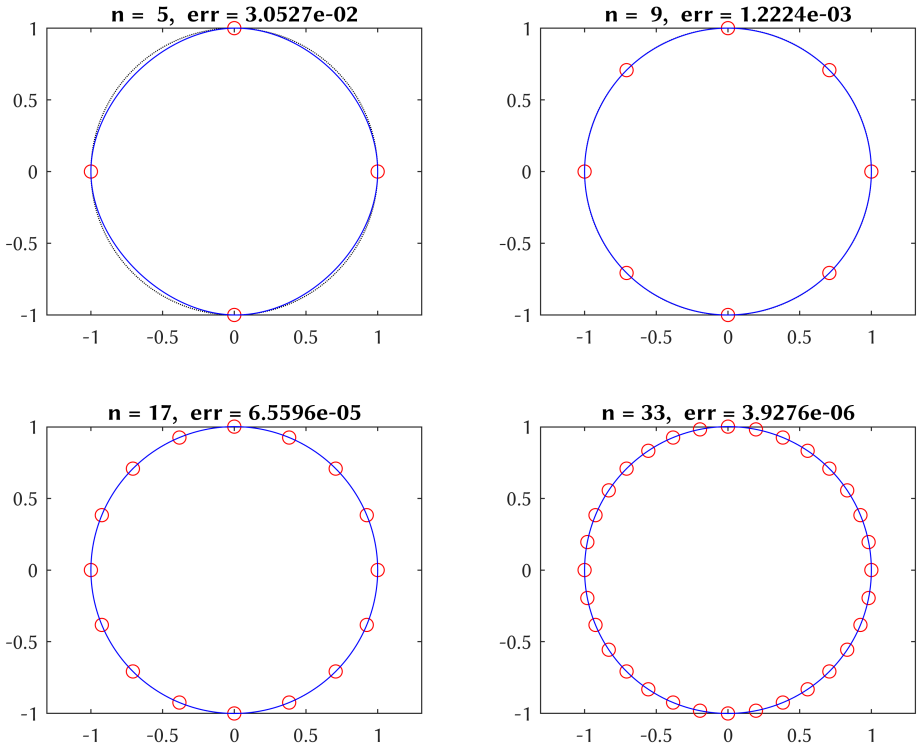
Though we were able to work out the previous part with `interp1`, we do need to use `spline` for this part to incorporate the first-derivative boundary conditions. The basic syntax of `spline` is very similar to that of `interp1`:

```
x = spline(sdp, xdp, s);
y = spline(sdp, ydp, s);
```

If the first derivatve values at the end points shall be specified (this is called the *clamped cubic spline*), prepend and append the values to the vector of ordinates, in our case, `xdp` and `ydp`. So when MATLAB recognizes that the second input to `spline` has exactly two more elements than the first input, then it uses the interior elements as data points while the first and the last elements as derivative values at the end points. As we will see below, this results in an improved interpolation as can be confirmed both numerically and graphically.

```
clf
```

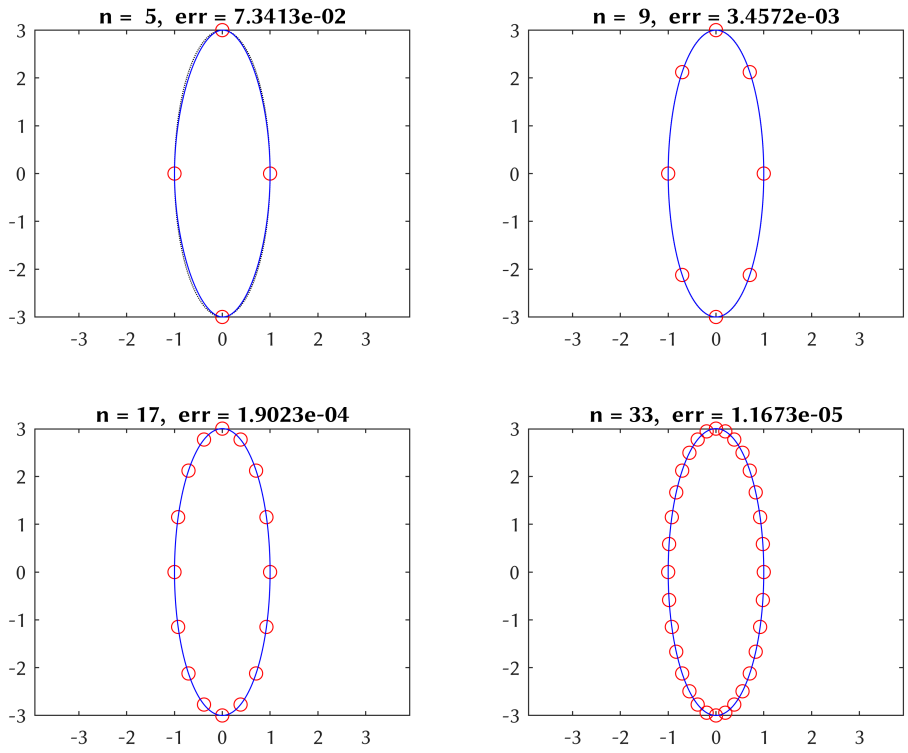
```
for j = 1:length(nvals)
    n = nvals(j);
    sdp = linspace(0, 2*pi, n)';     % nodes: arc length
    xdp = [0; cos(sdp); 0];          % ordinates: x-coord.
    ydp = [1; sin(sdp); 1];          % ordinates: y-coord.

    s = linspace(0, 2*pi, 1000)';    % query points
    xcirc = cos(s);                  % x-coord. on circle
    ycirc = sin(s);                  % y-coord. on circle
    x = spline(sdp, xdp, s);
    y = spline(sdp, ydp, s);

    err = max( sqrt((x-xcirc).^2 + (y-ycirc).^2) );

    subplot(2,2,j)
    plot(xdp(2:end-1), ydp(2:end-1), 'ro'), hold on   % plot data points
    plot(xcirc, ycirc, 'k:')         % plot circle
    plot(x, y, 'b')                  % plot spline
    axis equal
    title(sprintf('n = %2d,  err = %8.4e', n, err))
end
```



(b)

In this part, I will only show the clamped spline as in (a,i).

```
clf
for j = 1:length(nvals)
    n = nvals(j);
    tdp = linspace(0, 2*pi, n)';     % nodes: arc length
    xdp = [0; cos(tdp); 0];          % ordinates: x-coord.
    ydp = [3; 3*sin(tdp); 3];        % ordinates: y-coord.

    t = linspace(0, 2*pi, 1000)';    % query points
    xell = cos(t);                   % x-coord. on ellipse
    yell = 3*sin(t);                 % y-coord. on ellipse
    x = spline(tdp, xdp, t);
    y = spline(tdp, ydp, t);

    err = max( sqrt((x-xell).^2 + (y-yell).^2) );

    subplot(2,2,j)
    plot(xdp(2:end-1), ydp(2:end-1), 'ro'), hold on   % plot data points
    plot(xell, yell, 'k:')          % plot ellipse
    plot(x, y, 'b')                  % plot spline
    axis equal
    title(sprintf('n = %2d,  err = %8.4e', n, err))
end
```



Updated on April 5, 2021

Math 3607: Homework 10

Due: 11:59PM, Monday, April 12, 2021

TOTAL: 20 points

1. Calculate the third-order forward difference approximation to $f'(x)$, which can be written as

$$D_h^{[3f]}\{f\}(x) \approx c_1 f(x) + c_2 f(x+h) + c_3 f(x+2h) + c_4 f(x+3h).$$

   You may use the approach shown in lecture or follow the directions found in Problem 14.1–5.

2. Do **LM** 14.1–12.

3. Do **LM** 14.1–17.

4. Do **LM** 14.2–3(b).

5. Do **LM** 14.2–6.

6. Do **LM** 14.2–11(a).

Homework 10 (Solution)

Math 3607

Tae Eun Kim

Table of Contents

Several problems in this homework set involves lengthy by-hand calculations. These solutions are written in a separate document. This mlx file contains solutions for computer exercises.

Problem 1 (Higher-Order Forward Difference)

See the attached document.

Problem 2 (LM 14.1--12)

(a) Extrapolation for 4th-order centerend difference

The fourth-order centered difference formula is given by

$$D_h^{[4c]}\{f\}(x) = \frac{f(x-2h) - 8f(x-h) + 8(x+h) - f(x+2h)}{12h}.$$

See the attached document for derivation. We modify the textbook script `diff1` by implementing this formula as below:

```
%% script m-file: diff1 (modified)
f = @(x) sin(x.^2);
fdrv = @(x) 2*x.*cos(x.^2);
Df = @(x,h) (f(x+h) - f(x))./h;           % 1st-order FD
Dfc = @(x,h) (f(x+h) - f(x-h))./(2*h);    % 2nd-order CD
Df4c = @(x,h) ...                         % 4th-order CD
    (f(x-2*h) - 8*f(x-h) + 8*f(x+h) - f(x+2*h))./(12*h);
x = 1/3;
h0 = 0.1;
nr_h = 35;
h = h0*2.^(-[0:nr_h]');
A(:,1) = h;
A(:,2) = Df(x,h) - fdrv(x);
A(:,3) = Dfc(x,h) - fdrv(x);
```

```
A(:,4) = Df4c(x,h) - fdrv(x);
disp('      h          errors')
```

```
      h          errors
```

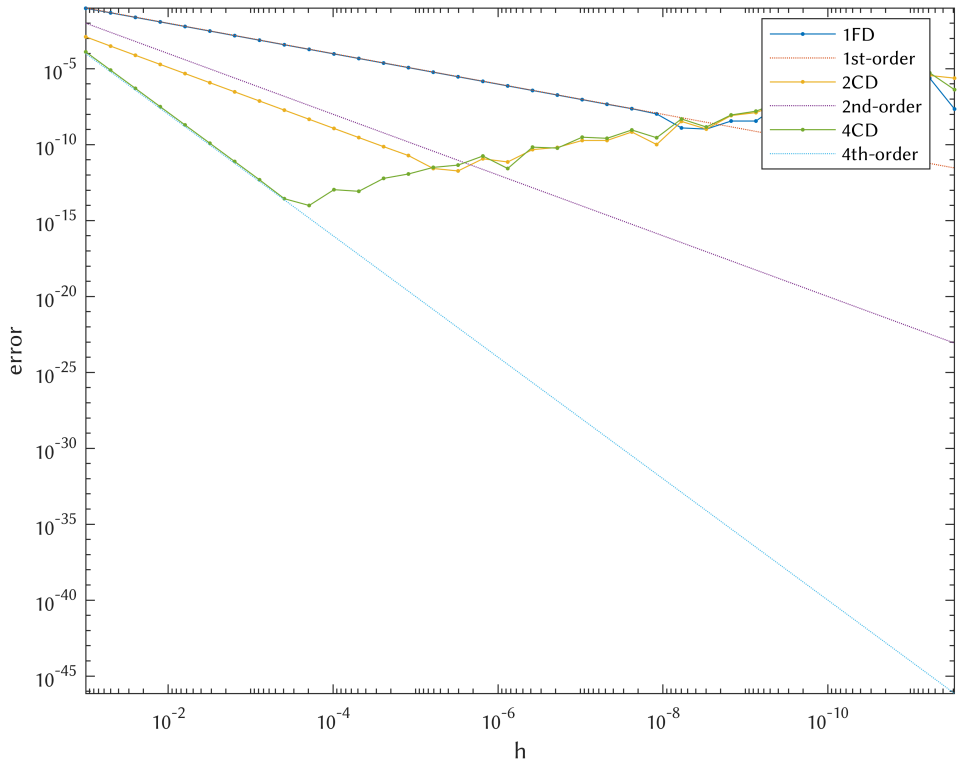
```
disp(A)
```

```
                     0.1        0.0953800307503522       -0.0012624336679421       0.000129128211100116
                    0.05        0.0481156495713236       -0.00030952874788992      8.10622546054685e-06
                   0.025        0.0241485170712513      -7.70018004009332e-05      5.07182095432768e-07
                  0.0125        0.0120951469886871       -1.9226669620287e-05      3.17073064470819e-08
                 0.00625       0.00605258336365655      -4.80518102807803e-06      1.98183547350794e-09
                0.003125       0.00302751828247738       -1.2012023592467e-06      1.23864696277565e-10
               0.0015625          0.00151406259914      -3.00294775157361e-07      7.75568498312396e-12
              0.00078125       0.00075710676755103      -7.50733292198547e-08      4.85944617878431e-13
            0.000390625       0.000378572201298888      -1.87683507624214e-08     -2.76445533131664e-14
           0.0001953125       0.000189290798871533       -4.6920906049408e-09     -9.88098491916389e-15
           9.765625e-05       9.46465732593049e-05      -1.17295007040497e-09      1.08468789505878e-13
          4.8828125e-05       4.73235799486327e-05       -2.9315605498681e-10       8.4821039081362e-14
         2.44140625e-05       2.36618628118856e-05      -7.37404581840906e-11     -6.02073946254222e-13
        1.220703125e-05       1.18309488211787e-05      -1.91707760777149e-11      -1.1705081348623e-12
        6.103515625e-06       5.91548045714152e-06      -2.68618460808057e-12      3.18756132600129e-12
       3.0517578125e-06       2.95774229608359e-06       1.86128890078407e-12       4.5138337512185e-12
      1.52587890625e-06       1.47885843626572e-06      -1.17811316258098e-11     -1.78445036524977e-11
      7.62939453125e-07       7.39430148777309e-07      -7.23365811694521e-12     -2.68629563038303e-12
      3.814697265625e-07       3.69756932294685e-07       4.7336023989430 5e-11      6.85574930159305e-11
      1.9073486328125e-07       1.84911229106355e-07       6.55259180248891e-11      5.94625459982012e-11
      9.5367431640625e-08        9.2215529101658e-08      -1.89132598471531e-10     -3.10398706737658e-10
     4.76837158203125e-08       4.57949195231677e-08      -1.89132598471531e-10     -2.61892285635668e-10
     2.38418579101563e-08       2.33849700714828e-08        6.8398231523048e-10      9.26514087673525e-10
     1.19209289550781e-08       1.05792846705199e-08       1.01905706095806e-10     -2.86145440675512e-10
     5.96046447753906e-09       1.26605892436515e-09      -3.39055394871224e-09      -4.9427583137529e-09
     2.98023223876953e-09      -1.06224751217354e-09      -1.06224751217354e-09     -1.45029865894486e-09
     1.49011611938477e-09       3.59436536090385e-09       8.25097823398124e-09      9.02708030547927e-09
     7.45058059692383e-10       3.59436536090385e-09       1.29075911070586e-08      1.60119996150954e-08
     3.72529029846191e-10      -5.22849891160249e-08      -5.22849891160249e-08     -6.78070321002977e-08
     1.86264514923096e-10      -5.22849891160249e-08      -5.22849891160249e-08     -5.22849892270472e-08
     9.31322574615479e-11        1.7123242879169e-07       2.45738234760928e-07      3.57496943603763e-07
     4.65661287307739e-11       2.22208168532134e-08        1.7123242879169e-07      2.45738234649906e-07
      2.3283064365387e-11      -5.73825630900693e-07      -5.73825630900693e-07     -6.73166705600359e-07
     1.16415321826935e-11       -1.1698720786546e-06       -1.1698720786546e-06     -1.26921315335426e-06
     5.82076609134674e-12       2.40640660786884e-06       3.59849950337665e-06      5.38663884652735e-06
     2.91038304567337e-12       2.22208168532134e-08       2.40640660786884e-06      4.19585115207788e-07
```

Confirm the accuracy on the log-log graph below.

```
clf
% 1FD
loglog(A(:,1), abs(A(:,2)), '.-'), hold on
loglog(A(:,1), A(:,1), ':')
% 2CD
loglog(A(:,1), abs(A(:,3)), '.-')
loglog(A(:,1), A(:,1).^2, ':')
% 4CD
loglog(A(:,1), abs(A(:,4)), '.-')
loglog(A(:,1), A(:,1).^4, ':')
% Prettifying
xlabel('h'),ylabel('error')
axis tight
legend('1FD', '1st-order', '2CD', '2nd-order', '4CD', '4th-order', 'Location', 'best')
```

```
set(gca, 'xdir', 'Reverse')
```



Question. Can you confirm from the previous graph the optimal *h* for each method?

(b) Second Derivatives

```
clear A
```

The second-order centered difference method for $f''(x)$ is given by

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}.$$

By Richardson extrapolation, we obtained the fourth-order centered difference formula for $f''(x)$

$$f''(x) \approx \frac{-f(x-2h) + 16f(x-h) - 6f(x) + 16f(x+h) - f(x+2h)}{12h^2}.$$

```
%% script m-file: diff1 (modified)
f = @(x) sin(x.^2);
f2drv = @(x) 2*cos(x.^2) - 4*x.^2.*sin(x.^2);
D2fc = @(x,h) (f(x+h) - 2*f(x) + f(x-h))./(h.^2);    % 2nd-order CD
D2f4c = @(x,h) ...                                    % 4th-order CD
    (-f(x-2*h) + 16*f(x-h) - 30*f(x) + 16*f(x+h) - f(x+2*h))./(12*h.^2);
x = 1/3;
h0 = 0.1;
```

```
nr_h = 20;
h = h0*2.^(-[0:nr_h]');
A(:,1) = h;
A(:,2) = D2fc(x,h) - f2drv(x);
A(:,3) = D2f4c(x,h) - f2drv(x);
disp('      h          errors')
```

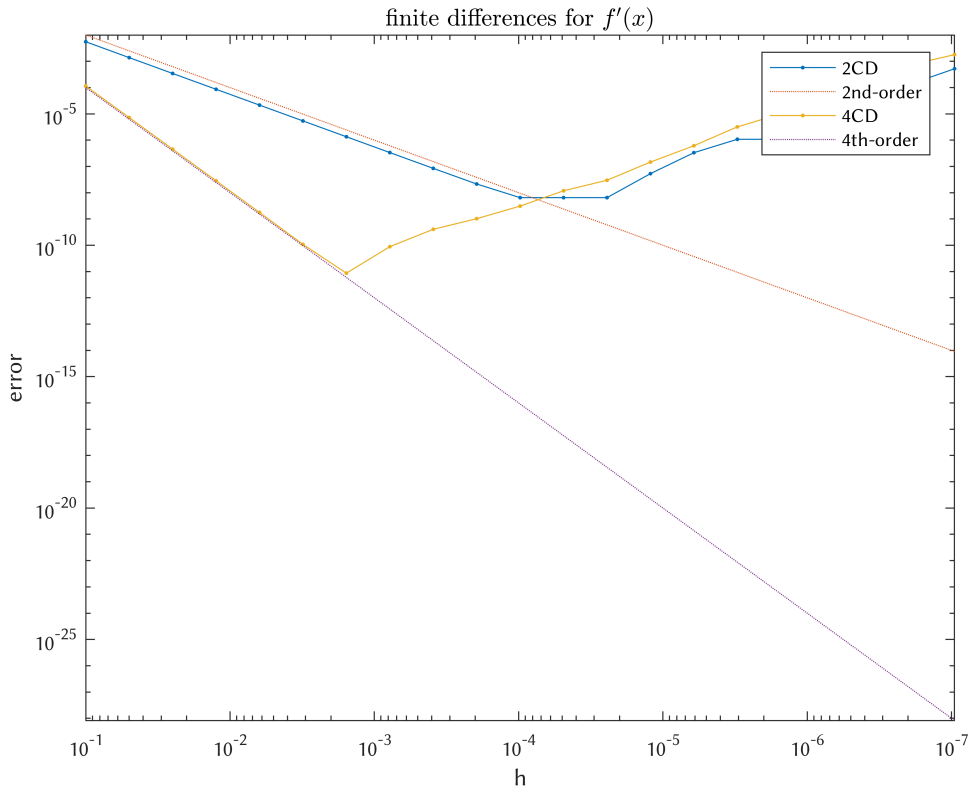
```
      h          errors
```

```
disp(A)
```

```
                    0.1      -0.00553656048525486       0.000114478122608119
                   0.05      -0.00137871608259621        7.2320516149027e-06
                  0.025      -0.00034433911895837       4.53202243289041e-07
                 0.0125     -8.60635219601669e-05       2.83435579451208e-08
                0.00625     -2.15145520576776e-05       1.77094716669046e-09
               0.003125     -5.37855569637813e-06       1.07981179553462e-10
              0.0015625     -1.34463973955334e-06      -8.66640093022397e-12
             0.00078125     -3.36197698569407e-07      -8.82471873353552e-11
            0.000390625     -8.42449288107616e-08       -4.0467540429745e-10
           0.0001953125     -2.13988449182523e-08      -1.02616359853869e-09
           9.765625e-05     -6.48313180917626e-09      -3.08768477452759e-09
          4.8828125e-05     -6.48313180917626e-09      -1.18188339115477e-08
         2.44140625e-05     -6.48313180917626e-09      -2.97661961745632e-08
        1.220703125e-05     -5.30492605399502e-08      -1.46181518001498e-07
        6.103515625e-06     -3.32446032924594e-07      -6.11842805309237e-07
       3.0517578125e-06     -1.07750409261698e-06      -3.18850192826403e-06
      1.52587890625e-06     -1.07750409261698e-06      -1.00182008089256e-05
      7.62939453125e-07      4.88296038492209e-06      -2.09457190176732e-05
     3.814697265625e-07     -4.28007554353904e-05       -9.8431757225681e-05
    1.9073486328125e-07     -0.000138168187076015        -0.0007421619207999
    9.5367431640625e-08     -0.000519637913638515       -0.00179120366884677
```

Plot the errors on the log-log graph to confirm the order of accuracy:

```
clf
loglog(A(:,1), abs(A(:,2)), '.-'),hold on
loglog(A(:,1), A(:,1).^2, ':')

loglog(A(:,1), abs(A(:,3)), '.-')
loglog(A(:,1), A(:,1).^4, ':')
xlabel('h'),ylabel('error')
title('finite differences for $f''(x)$', 'Interpreter', 'latex')
axis tight
legend('2CD', '2nd-order', '4CD', '4th-order', 'Location', 'best')
set(gca, 'xdir', 'Reverse')
```



Problem 3 (LM 14.1--17, Sequences Converging to π)

Begin by defining anonymous functions

```
p = @(n) n.*sin(pi./n);       % 2nd-order; underestimate
P = @(n) n.*tan(pi./n);       % 2nd-order; overestimate
B = @(n) (p(n) + P(n))/2;     % 2nd-order; better than p(n) and P(n)
R = @(n) (2*p(n) + P(n))/3;   % 4th-order; obtained from extrapolation
```

The question asks that we calculate the sequences for $n = 48$ and $n = 96$; I will do some more. Below is a quick and dirty way to calculate them.

```
n = 48*2.^(0:9)';
calc = [p(n), P(n), B(n), R(n)];
disp([n, calc])
```

```
   48   3.13935020304687   3.14608621513143   3.14271820908915
   96   3.14103195089051   3.14271459964537   3.14187327526794
  192   3.14145247228546   3.14187304997982   3.14166276113264
  384   3.14155760791186   3.14166274705685   3.14161017748435
  768   3.14158389214832   3.14161017660469    3.1415970343765
 1536   3.14159046322805   3.14159703432153   3.14159374877479
 3072   3.14159210599927   3.14159374877135   3.14159292738531
 6144   3.14159251669216    3.1415929273851   3.14159272203863
12288   3.14159261936538   3.14159272203861     3.141592670702
24576   3.14159264503369     3.141592670702   3.14159265786784
```

The errors:

```
disp([n, calc-pi])
```

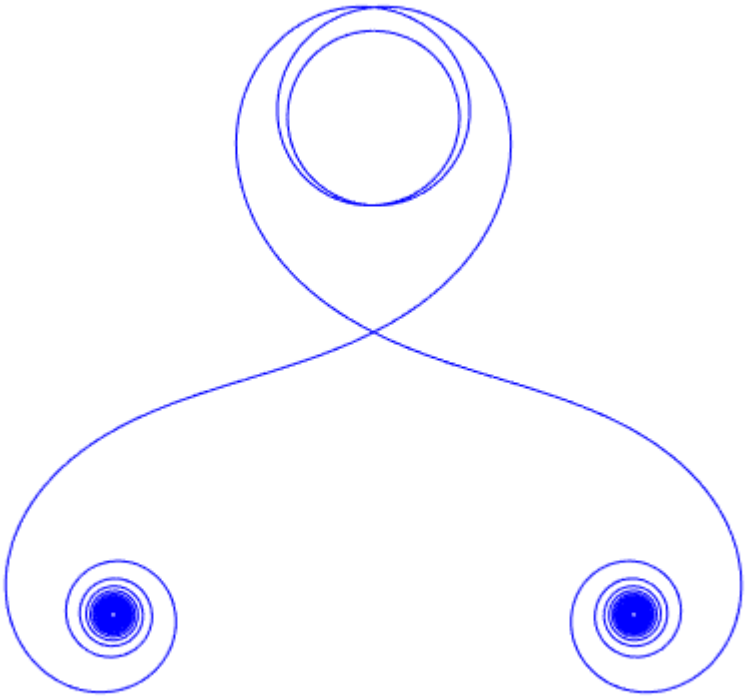
```
    48    -0.00224245054292638     0.00449356154164171     0.00112555549935767
    96    -0.00056070269928376     0.00112194605557514     0.00028062167814546 5
   192    -0.00014018130433157     0.00028039639003019     7.0107542849307e-05
   384    -3.5045677935663e-05     7.0093467054999e-05     1.7523894559458e-05
   768    -8.7614414754356e-06     1.7523014895993e-05     4.3807867102785e-06
  1536    -2.1903617435370e-06     4.3807317324784e-06     1.0951849942486e-06
  3072    -5.4759052181552e-07     1.0951815587745e-06     2.7379551870155e-07
  6144    -1.3689763633806e-07     2.7379530376237e-07     6.8448833712154e-08
 12288    -3.4224409528605e-08     6.8448820389477e-08     1.7112205430436e-08
 24576    -8.5561024931735e-09     1.7112204986347e-08     4.2780512465868e-09
```

If you want to have a finer control over how the numbers are formatted, use `fprintf` as demonstrated many times in other homework solutions or in class.

Problem 4 (LM 14.2--3(b), Spiral)

Simply modify the integrands in the Euler spiral code provided for Lecture 31 and 32.

```
T = 15;
h = 0.001;
t = (0:h:T);
tmid = ( t(1:end-1)+t(2:end) )/2;
fx = @(z) cos(0.25*z.^3-5.2*z);
fy = @(z) sin(0.25*z.^3-5.2*z);
Ix = h/6 * ( fx(t(1:end-1)) + 4*fx(tmid) + fx(t(2:end)) );
Iy = h/6 * ( fy(t(1:end-1)) + 4*fy(tmid) + fy(t(2:end)) );
x = cumsum(Ix);
y = cumsum(Iy);

clf
plot(x, y, 'b', -x, y, 'b')
axis equal, axis off
grid on
```



Problem 5 (LM 14.2--6, Smoothness and Accuracy)

To be updated.

Problem 6 (LM 14.2--11(a), Extrapolation for Composite Methods)

See the attached document.

Math 3607: Homework 10

Selected Solutions

1. (Derivation of 3rd-order forward difference formula; solution by extrapolation)

   Let $V = f'(x)$ and $V_h = D_h^{[1f]}\{f\}(x)$, the first-order forward difference formula, for simplicity. Recall that

   $$\begin{aligned} V_h &= V + \underbrace{\frac{f''(x)}{2}}_{b_1} h + \underbrace{\frac{f'''(x)}{6}}_{b_2} h^2 + \underbrace{\frac{f^{(4)}(x)}{24}}_{b_3} h^3 + O(h^4). \\ &= V + b_1 h + b_2 h^2 + b_3 h^3 + O(h^4). \end{aligned}$$

   To obtain a third-order method, the first two error terms $b_1 h$ and $b_2 h^2$ must be eliminated. To this end, we look for a linear combination of V_h, V_{2h}, and V_{3h} such that

   $$\alpha_1 V_h + \alpha_2 V_{2h} + \alpha_3 V_{3h} = V + O(h^3),$$

   where α_j are to be determined. Writing the left-hand side out and collecting like-terms, we have

   $$\begin{aligned} &\alpha_1 V_h + \alpha_2 V_{2h} + \alpha_3 V_{3h} \\ &\qquad = (\alpha_1 + \alpha_2 + \alpha_3)V + (\alpha_1 + 2\alpha_2 + 3\alpha_3)b_1 h + (\alpha_1 + 4\alpha_2 + 9\alpha_3)b_2 h^2 + O(h^3). \end{aligned}$$

   Matching coefficients, we obtain a linear system of three equations for the unknown weights α_j, for $j = 1, 2, 3$:

   $$\begin{aligned} \alpha_1 \;+\; \alpha_2 \;+\; \alpha_3 &= 1 \\ \alpha_1 \;+\; 2\alpha_2 \;+\; 3\alpha_3 &= 0 \\ \alpha_1 \;+\; 4\alpha_2 \;+\; 9\alpha_3 &= 0 \end{aligned}$$

   Solving the system (say by Gaussian elimination), we obtain

   $$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \\ 1 \end{bmatrix}.$$

   Therefore, $3V_h - 3V_{2h} + V_{3h}$ is a new formula approximating V with third-order accuracy:

   $$\begin{aligned} D_h^{[3f]}\{f\}(x) &= 3V_h - 3V_{2h} + V_{3h} \\ &= 3\frac{f(x+h) - f(x)}{h} - 3\frac{f(x+2h) - f(x)}{2h} + \frac{f(x+3h) - f(x)}{3h} \\ &= \frac{-11f(x) + 18f(x+h) - 9f(x+2h) + 2f(x+3h)}{6h}. \end{aligned}$$

2. (**LM** 14.1–12: 4th-order centered difference formulas)

(a) Recall that

$$D_h^{[2c]}\{f\}(x) = f'(x) + c_2h^2 + c_4h^4 + O(h^6).$$

It follows that

$$D_{2h}^{[2c]}\{f\}(x) = f'(x) + 4c_2h^2 + 16c_4h^4 + O(h^6),$$

and so

$$\frac{4D_h^{[2c]}\{f\}(x) - D_{2h}^{[2c]}\{f\}(x)}{3} = f'(x) - 4c_4h^4 + O(h^6).$$

Therefore, the fourth-order centered difference formula is given by

$$\begin{aligned} D_h^{[4c]}\{f\}(x) &= \frac{4D_h^{[2c]}\{f\}(x) - D_{2h}^{[2c]}\{f\}(x)}{3} \\ &= \frac{4}{3}\frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3}\frac{f(x+2h) - f(x-2h)}{4h} \\ &= \frac{f(x-2h) - 8f(x-h) + 8(x+h) - f(x+2h)}{12h}. \end{aligned}$$

(b) The second-order centered difference formula for $f''(x)$ is given by

$$D_h^2\{f\}(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = f''(x) + \frac{1}{12}f''''(x)h^2 + O(h^4).$$

(See Lecture 33 or **LM** p.1766-7.) By a similar argument as above, we see that

$$\frac{4D_h^2\{f\}(x) - D_{2h}^2\{f\}(x)}{3} = f''(x) + O(h^4),$$

yields a fourth-order centered difference formula for $f''(x)$:

$$\begin{aligned} &\text{(4th-order CD for } f''(x)) \\ &\quad= \frac{4D_h^2\{f\}(x) - D_{2h}^2\{f\}(x)}{3} \\ &\quad= \frac{4}{3}\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{1}{3}\frac{f(x+2h) - 2f(x) + f(x-2h)}{4h^2} \\ &\quad= \frac{-f(x-2h) + 16f(x-h) - 30f(x) + 16f(x+h) - f(x+2h)}{12h^2}. \end{aligned}$$

3. (**LM** 14.1–17: Sequences converging to π) Applying the suggested change of variables $h = 1/n$ and Taylor-expanding about $h = 0$, we obtain

$$\begin{aligned} p_n &= \frac{\sin(\pi h)}{h} = \pi - \frac{\pi^3}{6}h^2 + \frac{\pi^5}{120}h^4 + \cdots = \pi + a_1h^2 + a_2h^4 + \cdots \\ P_n &= \frac{\tan(\pi h)}{h} = \pi + \frac{\pi^3}{3}h^2 + \frac{2\pi^5}{15}h^4 + \cdots = \pi + b_1h^2 + b_2h^4 + \cdots . \end{aligned}$$

Note that both are second-order accurate. The average of the two algorithms gives another second-order algorithm as h^2 term survives:

$$\mathfrak{B}_n = \pi + c_1h^2 + c_2h^4 + \cdots ,$$

where

$$c_1 = \frac{a_1 + b_1}{2} = \frac{\pi^3}{12}, \quad c_2 = \frac{a_2 + b_2}{2} = \frac{17\pi^5}{240}.$$

One way to obtain a fourth-order algorithm is to extrapolate p_n and P_n. Calculation shows that

$$\mathfrak{R}_n \equiv \frac{2}{3}p_n + \frac{1}{3}P_n = \pi + \frac{\pi^5}{20}h^4 + \cdots .$$

The above is not the only way. One may, for instance, construct another fourth-order algorithm by extrapolating $\mathfrak{B}_{\mathfrak{n}}$ and $\mathfrak{B}_{\mathfrak{n}/2}$:

$$\mathfrak{S}_n \equiv \frac{4}{3}\mathfrak{B}_n - \frac{1}{3}\mathfrak{B}_{n/2} = \pi - \frac{17\pi^5}{60}h^4 + \cdots .$$

6. (**LM** 14.2–11(a): Derivation of the composite Simpson's method via extrapolation)

Begin by writing down the generic composite trapezoidal method with n evenly spaced out nodes $a = x_1 < x_2 < \cdots < x_n = b$:

$$\begin{aligned} I_h^{[\mathrm{t}]} &= h\left(\frac{1}{2}f(x_1) + f(x_2) + f(x_3) + \cdots + f(x_{n-1}) + \frac{1}{2}f(x_n)\right) \\ &= h\left(\frac{1}{2}\big(f(x_1) + f(x_n)\big) + \sum_{j=2}^{n-1} f(x_j)\right), \end{aligned} \tag{1}$$

where $h = (b-a)/(n-1)$ and $x_j = a + (j-1)h$. Now write down the composite trapezoidal method $I_{h/2}^{[\mathrm{t}]}$, the one with $2n-1$ evenly spaced out nodes on the same interval $[a, b]$:

$$\begin{aligned} I_{h/2}^{[\mathrm{t}]} &= \frac{h}{2}\left(\frac{1}{2}f(x_1) + f(x_{1+1/2}) + f(x_2) + \cdots + f(x_{n-1}) + f(x_{n-1+1/2}) + \frac{1}{2}f(x_n)\right) \\ &= \frac{h}{2}\left(\frac{1}{2}\big(f(x_1) + f(x_n)\big) + \sum_{j=2}^{n-1} f(x_j) + \sum_{j=1}^{n-1} f(x_{j+1/2})\right), \end{aligned} \tag{2}$$

where h and x_j are as above and $x_{j+1/2} = (x_j + x_{j+1})/2$. Since the composite trapezoidal method is second-order accurate,

$$I_h^{[\mathrm{t}]} = I + c_1 h^2 + O(h^4) \quad \text{and} \quad I_{h/2}^{[\mathrm{t}]} = I + \frac{1}{4}c_1 h^2 + O(h^4),$$

and so

$$\frac{4I_{h/2}^{[\mathrm{t}]} - I_h^{[\mathrm{t}]}}{3} = I + O(h^4).$$

The left-hand side is a fourth-order accurate algorithm for the integral I. By (1) and (2)

$$\begin{aligned}\frac{4I_{h/2}^{[\mathrm{t}]} - I_h^{[\mathrm{t}]}}{3} &= \frac{2}{3}h\left(\frac{1}{2}\left(f(x_1)+f(x_n)\right) + \sum_{j=2}^{n-1} f(x_j) + \sum_{j=1}^{n-1} f(x_{j+1/2})\right) \\ &\qquad - \frac{1}{3}h\left(\frac{1}{2}\left(f(x_1)+f(x_n)\right) + \sum_{j=2}^{n-1} f(x_j)\right) \\ &= \frac{1}{3}h\left(\frac{1}{2}\left(f(x_1)+f(x_n)\right) + \underbrace{\sum_{j=2}^{n-1} f(x_j)}_{\star} + 2\sum_{j=1}^{n-1} f(x_{j+1/2})\right);\end{aligned}$$

splitting the marked sum and re-indexing one of the two,

$$\begin{aligned}&= \frac{1}{3}h\left(\frac{1}{2}\left(f(x_1)+f(x_n)\right) + \frac{1}{2}\sum_{j=2}^{n-1} f(x_j) + \frac{1}{2}\sum_{j=1}^{n-2} f(x_{j+1}) + 2\sum_{j=1}^{n-1} f(x_{j+1/2})\right) \\ &= \frac{1}{3}h\sum_{j=1}^{n-1}\left(\frac{1}{2}f(x_j) + 2f(x_{j+1/2}) + \frac{1}{2}f(x_{j+1})\right).\end{aligned}$$

Note that this is exactly the composite Simpson's method!

Updated on April 14, 2021

Math 3607: Homework 11

(no due date)

You do not need to submit this assignment, yet you are highly encouraged to work out this problem set in preparation for Exam 4.

1 Optimal Step Size

In lecture, the optimal h for the second-order centered difference formula was shown to be about $\boxed{\text{eps}}^{1/3}$. At this optimal h, the leading error is $O(\boxed{\text{eps}}^{2/3})$. (Why?)

(a) (By hand) Determine the optimal h for the first-order forward difference formula by following a similar argument. Also determine the leading error at this optimal h.

(b) (By hand) Generalize the argument to determine the optimal h for an m-th order accurate method, where m is any positive integer. Also determine the leading error at this optimal h.

(c) (Computer) Complete the following program approximating the Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ using the first-order forward difference using the optimal step size determined in the previous parts.

```
function J = jacfd(f, x0)
% JACFD Approximation of a Jacobian by 1st-order forward difference
% Input:
%   f       function to be differentiated
%           which takes (n-by-1) column vector as an input
%           and produces (m-by-1) column vector as an output
%   x0      evaluation point
% Output:
%   J       approximate Jacobian (m-by-n)

    h = [.....];    % optimal step size




end
```

Hint. (**Tip for vectorization**) Recall that

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{1}$$

The jth column of $\mathbf{J}$ consists of all partial derivatives with respect to x_j:

$$\mathbf{J}(\mathbf{x})\mathbf{e}_j = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_j} \\ \dfrac{\partial f_2}{\partial x_j} \\ \vdots \\ \dfrac{\partial f_m}{\partial x_j} \end{bmatrix}$$

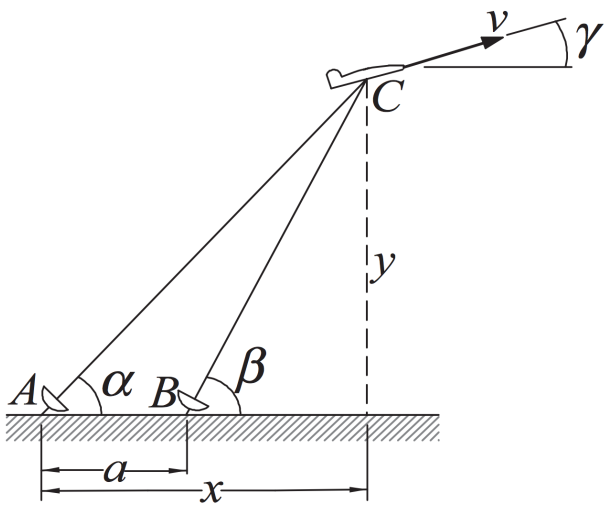
This column vector can be approximated by a finite difference formula involving a perturbation only in x_j:

$$\mathbf{J}(\mathbf{x})\mathbf{e}_j \approx \frac{\mathbf{f}(\mathbf{x} + h\mathbf{e}_j) - \mathbf{f}(\mathbf{x})}{h}, \quad j = 1, \ldots, n,$$

where h is optimally chosen according to the previous parts.

2 Air Plane Velocity from Radar Readings

(This exercise is adapted from an exercise in [1].) The radar stations A and B, separated by the distance $a = 500$ m, track a plane C by recording the angles α and β at one-second intervals. Your goal, back at air traffic control, is to determine the speed of the plane.



Let the position of the plane at time t be given by $(x(t), y(t))^{\mathrm{T}}$. The speed at time t is the magnitude of the velocity vector,

$$\left\| \frac{d}{dt} \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} \right\| = \sqrt{x'(t)^2 + y'(t)^2}. \tag{2}$$

The closed forms of the functions $x(t)$ and $y(t)$ are unknown (and may not exist at all), but we can still use numerical methods to estimate $x'(t)$ and $y'(t)$. For example, at $t = 3$, the second order centered difference quotient for $x'(t)$ is

$$x'(3) \approx \frac{x(3+h) - x(3-h)}{2h} = \frac{1}{2}(x(4) - x(2)).$$

In this case $h = 1$ since data comes in from the radar stations at 1 second intervals.

Successive readings for α and β at integer times $t = 7, 8, \ldots, 14$ are stored in the file `plane.dat`. Each row in the array represents a different reading; the columns are the observation time t, the angle α (in degrees), and the angle β (also in degrees), in that order. The Cartesian coordinates of the plane can be calculated from the angles α and β as follows:

$$x(\alpha, \beta) = a\frac{\tan(\beta)}{\tan(\beta) - \tan(\alpha)} \quad \text{and} \quad y(\alpha, \beta) = a\frac{\tan(\beta)\tan(\alpha)}{\tan(\beta) - \tan(\alpha)}. \tag{3}$$

(a) (By hand) Verify the equations in (3).

(b) (Computer) Load the data, convert α and β to radians[1], then compute the coordinates $x(t)$ and $y(t)$ at each given t using (3). Approximate $x'(t)$ and $y'(t)$ using the second-order forward difference for $t = 7$, the second-order backward difference for $t = 14$, and the second-order centered difference for $t = 8, 9, \ldots, 13$. Return the values of the speed at each t using (2).

3 Visualization of Spectra and Pseudospectra

(This exercise is adapted from Chapter 7 of [2].) The eigenvalues of *Toeplitz* matrices, which have a constant value on each diagonal, have beautiful connections to complex analysis. Define six 64×64 Toeplitz matrices using

```
z = zeros(1,60);
A{1} = toeplitz( [0,0,0,0,z], [0,1,1,0,z] );
A{2} = toeplitz( [0,1,0,0,z], [0,2i,0,0,z] );
A{3} = toeplitz( [0,2i,0,0,z], [0,0,1,0.7,z] );
A{4} = toeplitz( [0,0,1,0,z], [0,1,0,0,z] );
A{5} = toeplitz( [0,1,2,3,z], [0,-1,-2,0,z] );
A{6} = toeplitz( [0,0,-4,-2i,z], [0,2i,-1,2,z] );
```

(The variable `A` constructed hereinabove is a *cell array*. See my HW08 solutions for an example involving cell arrays.) For each of the six matrices, do the following. This is a computer exercise entirely.

(a) Plot the eigenvalues of `A{#}` as red dots in the complex plane. (Set `'MarkerSize'` to be 3.)

(b) Let E and F be 64×64 random matrices generated by `randn`. On top of the plot from part (a), plot the eigenvalues of the matrix $A + \varepsilon E + i\varepsilon F$ as blue dots, where $\varepsilon = 10^{-3}$. (Set `'MarkerSize'` to be 1.)

(c) Repeat part (b) 49 more times (generating a single plot).

Arrange all six plots in a 3×2 grid using `subplot`. Make sure all figures are drawn in 1:1 aspect ratio.

4 Vandermonde Matrix, SVD, and Rank

Let $\mathbf{x}$ be a vector of 1000 equally spaced points between 0 and 1, and let A_n be the $1000 \times n$ Vandermonde-type matrix whose (i, j) entry is x_i^{j-1} for $j = 1, \ldots, n$. This is a computer exercise.

[1] You may ignore this step and use `tand`.

(a) Print out the singular values of A_1, A_2, and A_3.

(b) Make a semi-log plot of the singular values of A_{25}.

(c) Use `rank` to find the rank of A_{25}. How does this relate to the graph from part (b)? You may want to use the help document for the `rank` command to understand what it does.

5 SVD and 2-Norm

Let $A \in \mathbb{C}^{m\times n}$ have an SVD $A = USV^*$. The following problem walks you through the proof of the fact that $\|A\|_2 = \sigma_1$. Do this by hand.

(a) Use the technique of Lagrange multipliers to show that among vectors that satisfy $\|\mathbf{x}\|_2^2 = 1$, any vector that maximizes $\|A\mathbf{x}\|_2^2$ must be an eigenvector of A^*A.

(*Hint.* If B is any hermitian matrix, *i.e.,* $B^* = B$, the gradient of the scalar function $\mathbf{x}^*B\mathbf{x}$ with respect to $\mathbf{x}$ is $2B\mathbf{x}$.)

(b) Use the result of part (a) to prove that $\|A\|_2 = \sigma_1$, the *principal singular value* of A.

References

[1] Jaan Kiusalaas. *Numerical methods in engineering with Python 3*. Cambridge university press, 2013.

[2] Lloyd N. Trefethen and Mark Embree. *Spectra and Pseudospectra: The Behavior of Nonnormal Matrices and Operators*. Princeton University Press, 2005.

Homework 11 (Solution)

Math 3607

Tae Eun Kim

Table of Contents

Problem 1 (Optimal Step Size)

To be updated.

Problem 2 (Airplane Velocity from Radar Readings)

To be updated.

Problem 3 (Plots of Spectra and Pseudospectra)

First, define matrices as instructed.

```
z = zeros(1,60);
AA{1} = toeplitz( [0,0,0,0,z], [0,1,1,0,z] );
AA{2} = toeplitz( [0,1,0,0,z], [0,2i,0,0,z] );
AA{3} = toeplitz( [0,2i,0,0,z], [0,0,1,0.7,z] );
AA{4} = toeplitz( [0,0,1,0,z], [0,1,0,0,z] );
AA{5} = toeplitz( [0,1,2,3,z], [0,-1,-2,0,z] );
AA{6} = toeplitz( [0,0,-4,-2i,z], [0,2i,-1,2,z] );
```

Digression.

Before we start solving the problem, let me briefly show you how `toeplitz` works. Take a look at the following simpler construction.

```
toeplitz([0,-1,-2,-3,-4],[0,1,2,3,4])
```

```
ans = 5×5
     0     1     2     3     4
    -1     0     1     2     3
    -2    -1     0     1     2
    -3    -2    -1     0     1
    -4    -3    -2    -1     0
```

This matrix has a constant value on each diagonal. Precisely, on the subdiagonals, starting from the main diagonal, we have

- main diagonal: 0
- 1st subdiagonal: -1
- 2nd subdiagonal: -2
- 3rd subdiagonal: -3
- 4th subdiagonal: -4

Likewise, on the superdiagonals, starting from the main diagonal, we have

- main diagonal: 0
- 1st superdiagonal: 1
- 2nd superdiagonal: 2
- 3rd superdiagonal: 3
- 4th superdiagonal: 4

These constant values on the diagonals are fed into `toeplitz` in the form of two vectors, one after another, and their first elements must agree. This seemingly simple structure enjoys many surprising analytical properties, and conversely, Toeplitz matrices appear here and there in mathematical modelling. Let's me just say this much and get back to the main problem.

Now, before we develop a script for the entire problem, let's play with one of the 6 matrices and build up from there.

```
A = AA{1};
[~, D] = eig(A);
```

Note that I used a placeholder `[~, D]` since we do not need eigenvectors. The sole output `D` must be a diagonal matrix and its diagonal entries are eigenvalues of `A`.

```
diag(D)
```

```
ans = 64×1
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     ⋮
```

But wait, all its eigenvalues are 0. In other words, there is only one eigenvalue $\lambda = 0$ whose algebraic multiplicity is 64! Does it make sense? Let's peak into A:

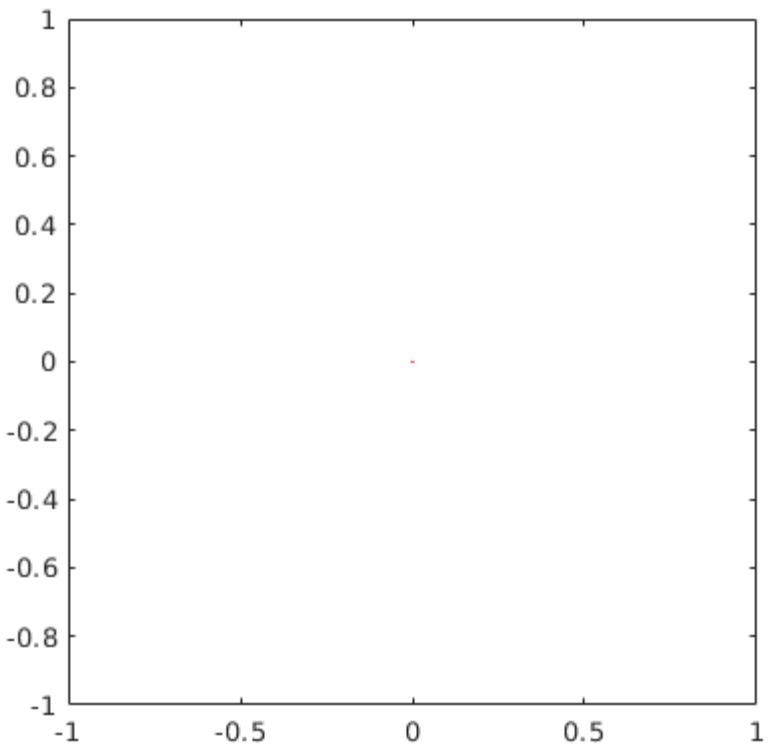
```
A(1:10, 1:10)
```

```
ans = 10×10
     0     1     1     0     0     0     0     0     0     0
     0     0     1     1     0     0     0     0     0     0
     0     0     0     1     1     0     0     0     0     0
     0     0     0     0     1     1     0     0     0     0
     0     0     0     0     0     1     1     0     0     0
     0     0     0     0     0     0     1     1     0     0
     0     0     0     0     0     0     0     1     1     0
     0     0     0     0     0     0     0     0     1     1
     0     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     0     0     0     0     0
```

As we see, all diagonals are zeros and it is not difficult to see that the characteristic polynomial of A is simply $\det(A - \lambda I) = \lambda^{64}$.

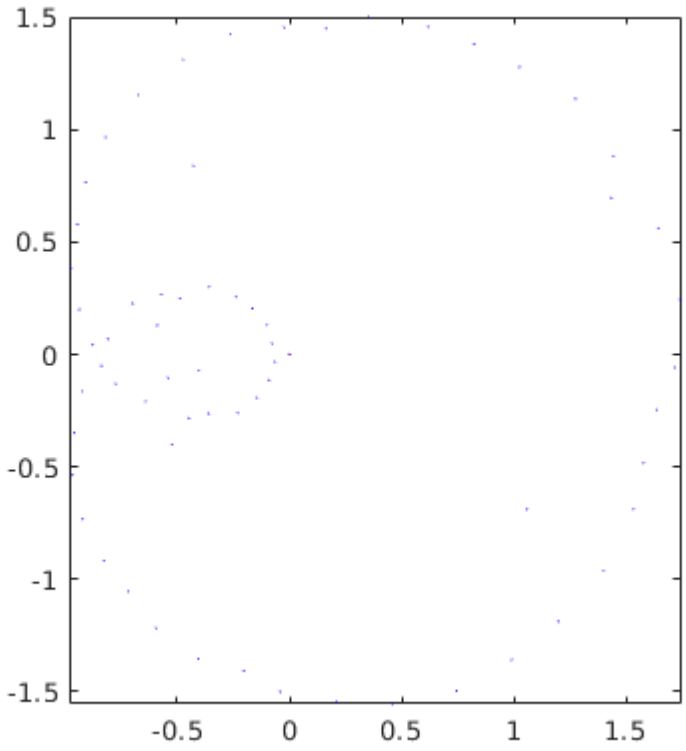
Alright, so if you plot all eigenvalues in the complex plane, you must see a single dot at the origin.

```
clf
plot(real(diag(D)), imag(diag(D)), 'r.', 'MarkerSize', 3)
axis image
```



Now, if we perturb this matrix A by a random complex matrix, all these 64 eigenvalues concentrated at the origin must be scattered. Let's see how these eigenvalues are perturbed in the complex plane.

```
E = randn(64);
F = randn(64);
[~, Dp] = eig(A+1e-3*(E+1i*F));
hold on
plot(real(Dp), imag(Dp), 'b.', 'MarkerSize', 1)
axis image
```

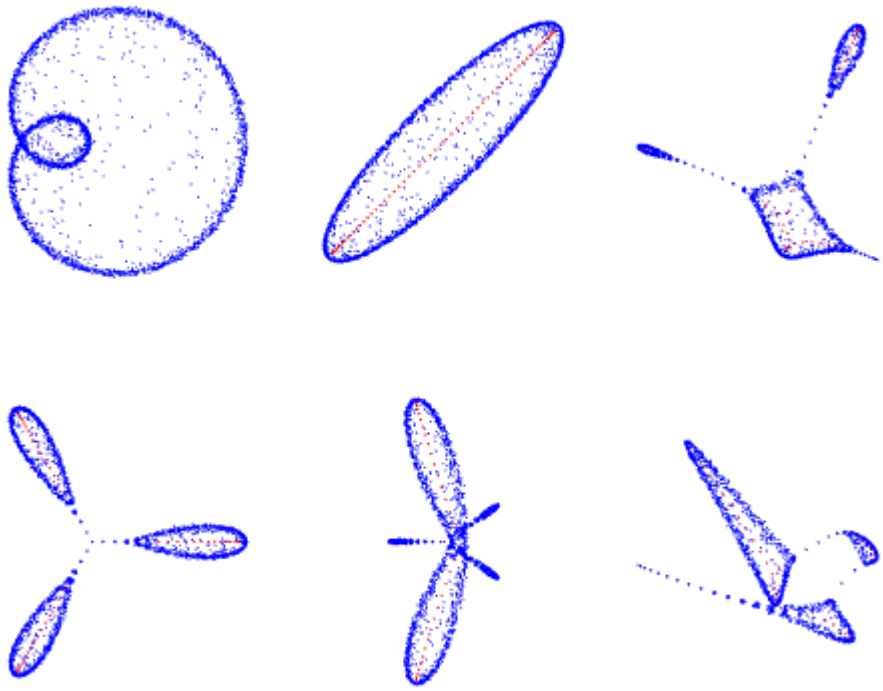


It appears that the eigenvalues are getting spread out in some pattern. Run this block multiple times and see that it traces out a Limacon with a loop.

Now that we have a clear idea of what the problem is asking us to do, let's write a neat script that goes through all parts with all six matrices and produces a nice plot.

```
clf
for j = 1:length(AA)
    A = AA{j};
    [~,D] = eig(A);
    subplot(2,3,j)
    plot(real(D), imag(D), 'r.', 'MarkerSize', 3)
    hold on
    for i = 1:50
        E = randn(64);
        F = randn(64);
```

```
        [Vp, Dp] = eig(A+1e-3*E+1i*1e-3*F);
        plot(real(Dp), imag(Dp), 'b.', 'MarkerSize', 1)
    end
    axis image, axis off
end
```



Problem 2 (SVD)

We have been playing with Vandermonde matrices several times in this semester.

```
x = linspace(0,1,1000)';
A = x.^(0:999);
```

This is certainly an overkill for this problem, but a 1000-by-1000 matrix is not big of a deal on modern machines.

(a) Let's print out singular values of A_1, A_2, and A_3. Just as in the previous problem, we only need singular values, which are the diagonal entries of in the SVD . So I will use placeholders ~ to prevent `svd` from outputing `U` and `V`. In addition, I will use the economy SVD to trim extra fat off from `S` matrix.

First, A_1 is a single column vector and it has only one singular value.

```
[~,S,~] = svd(A(:,1), 0);
diag(S)
```

```
ans = 31.6228
```

The matrix A_2 consists of the first two columns of A and so it has two singular values.

```
[~,S,~] = svd(A(:,1:2), 0);
diag(S)'
```

```
ans = 1×2
   35.6038    8.1161
```

The matrix A_3 has three singular values.

```
[~,S,~] = svd(A(:,1:3), 0);
diag(S)'
```

```
ans = 1×3
   37.5306   11.0704    1.6426
```

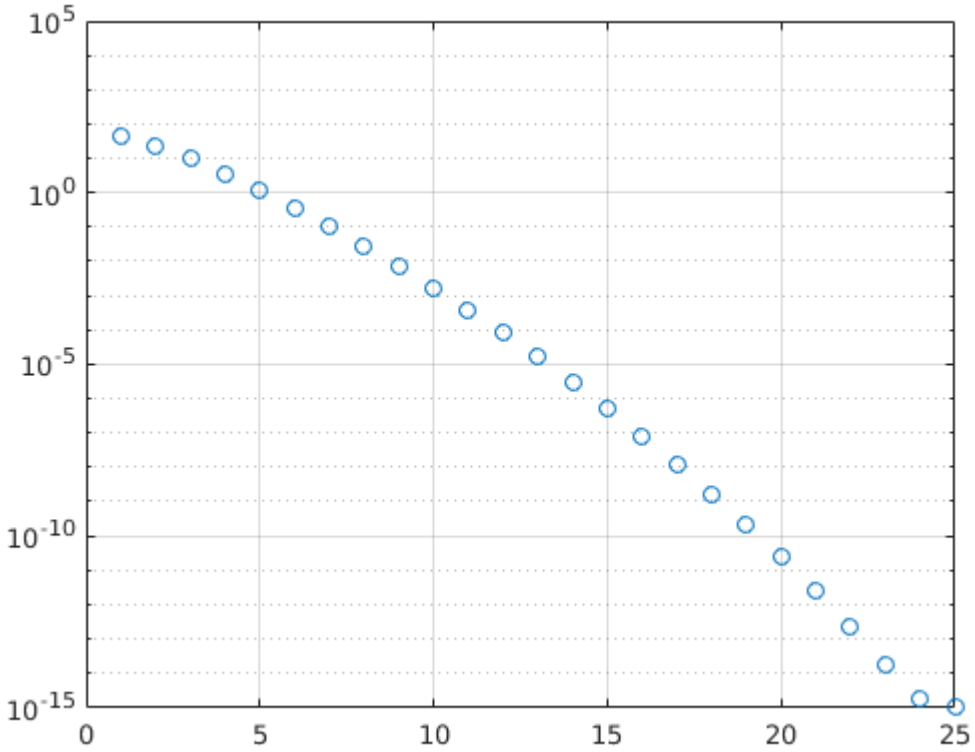
Let's use a loop to put the code into a nice script. I will use `fprintf` to format the outputs nicely for those who are familiar with it.

```
fmt = {'%5d', '%10.4f'};
for n = 1:3
    if n == 1
        fprintf('%5s %32s\n', 'n', 'singular values');
    end
    [~,S,~] = svd(A(:,1:n), 0);
    fprintf([fmt{[1 2*ones(1,n)]}, '\n'], n, diag(S)')
end
```

```
    n                 singular values
    1   31.6228
    2   35.6038    8.1161
    3   37.5306   11.0704    1.6426
```

(b) For A_{25}, we need to plot the singular values on a semi-log plot. Since we are plotting only a couple dozen discrete values, it makes sense to plot them in dots or circles, rather than connecting them using a line.

```
n = 25;
[~,S,~] = svd(A(:,1:n),0);
clf
semilogy(1:n, diag(S), 'o')
grid on
```



As we know from the theory of SVD, the singular values are arranged in descending order!

(c) In addition, we note from the previous plot that all singular values are non-negative. So, theoretically, the rank of A_{25} is 25, the number of nonzero singular values. However, MATLAB gives me a different answer:

```
A25 = A(:,1:25);
rank(A25)
```

```
ans = 20
```

What is going on? As the problem instructs, let's dig into the help document for `rank`:

```
help rank
```

```
  rank   Matrix rank.
     rank(A) provides an estimate of the number of linearly
     independent rows or columns of a matrix A.

     rank(A,TOL) is the number of singular values of A
     that are larger than TOL. By default, TOL = max(size(A)) * eps(norm(A)).

     Class support for input A:
        float: double, single

     Documentation for rank
     Other functions named rank
```

The second paragraph says that the tolerance (TOL) is set to be the larger of the dimensions of the given matrix (in our case, 25) multiplied by a variant of the machine epsilon. In our case, this default tolerance is about 7×10^{-12}.

```
tol = max(size(A25)) * eps(norm(A25))
```

```
tol = 7.1054e-12
```

This means that any singular values below this level is considered as zero taking into account the artifact of floating-point arithmetic. Let's confirm that this is indeed the case using logicals.

```
length( find(diag(S)>tol) )
```

```
ans = 20
```

Bingo!

Unless you absolutely know what you are doing, I would not recommend modifying the tolerance level. But if you wish, we can do something like:

```
rank(A25, 1e-16)
```

```
ans = 25
```

Problem 3 (SVD and 2-Norm)

(a) Since

$$\|A\|_2 = \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2,$$

the technique of Lagrange multiplier from Calc 3 is very relevant. Since a norm is maximized when its square is maximized, we need to solve the following *contrained optimization problem*:

- objective function (function to be maximized): $g(\mathbf{x}) := \|A\mathbf{x}\|_2^2 = \mathbf{x}^* A^* A \mathbf{x}$;
- constraint (restriction on independen variable): $h(\mathbf{x}) := \|\mathbf{x}\|_2^2 = \mathbf{x}^* \mathbf{x} = 1$.

By Lagrange, we set the gradient of one function to be a scalar multiple of the other,

$$\nabla g(\mathbf{x}) = \lambda \nabla h(\mathbf{x}),$$

and that already looks very promising due to the visual similarity to the eigenvalue problem. Furthermore, if you let $B = A^* A$, which appears in the definition of g, you realize that B is hermitian because $B^* = B$. Therefore, according to the hint (which can be confirmed by calculus with ease),

$$\nabla g(\mathbf{x}) = 2B\mathbf{x}.$$

On the other hand,

$$\nabla h(\mathbf{x}) = \nabla(x_1^2 + x_2^2 + \cdots + x_n^2) = \langle 2x_1, 2x_2, \cdots, 2x_n \rangle = 2\mathbf{x}.$$

So, upon cancellation, the Lagrange condition implies that

$$B\mathbf{x} = \lambda \mathbf{x},$$

that is, $\mathbf{x}$ must be an eigenvector of $B = A^*A$.

(b) Now, let $\lambda_1, \ldots, \lambda_n$ be eigenvalues of $B = A^*A \in \mathbb{C}^{n\times n}$, sorted in descending order, and let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be corresponding eigenvectors, all normalized to the unit length. Then,

$$\mathbf{x}_j^* B \mathbf{x}_j = \mathbf{x}_j^* \lambda_j \mathbf{x}_j = \lambda_j \mathbf{x}_j^* \mathbf{x}_j = \lambda_j$$

because $\mathbf{x}_j$ is a unit vector. It follows from the previous part that

$$\|A\|_2^2 = \max_{\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2^2 = \max_{1\le j\le n} \mathbf{x}_j^* B \mathbf{x}_j = \max_{1\le j\le n} \lambda_j = \lambda_1,$$

and so $\|A\|_2 = \sqrt{\lambda_1} =: \sigma_1$, the largest singular value of A, which completes the proof.

Note. The reaon why we were able to sort λ's in descending magnitude is due to the fact that the eigenvalues of any hermitian matrix are always real numbers. This is by the celebrated *spectral theorem* from linear algebra. Furthermore, since the hermitian matrix B is in a very special form $B = A^*A$, its eigenvalues are ensured to be not just real, but also nonnegative. This ensures that the square root of λ_j, namely, the singular value σ_j of A, is real as well; by choosing the positive square root, we are fully compliant to the conventions for SVD!