CIVL2530: Questionnaire 1

Suggestions:

1. Each question should be saved in a **separate file** where the filename is made up of lowercase characters. Example: **a1_1.jl**
2. In **all** files, add a comment like the one below at the **header** of the file to help you with organising your work:

```
# CIVL2530 Assignment 1 Question 1
# Author: FirstName LastName
# Date: 16/03/2017.
```

Section 1. Fundamentals

QUESTION 1: Create 6 variables named as you wish. Then print the contents of each variable with a nice message. Each one variable must represent:

1. an integer
2. a float point number (FPN: approximation of a real number)
3. a string
4. an one-dimensional array with 5 components, each being an integer
5. a bi-dimensional array with 10 components, each being an FPN
6. an array with inner arrays, filled with components of any type (you choose), in a way that the total number of items is 6

QUESTION 2: Create an array **v** with the following values: 6.0, 5.1, 4.2, 3.3, 2.4, 1.5, 0.6. Print the array with a nice message. Then (the following sequence is mandatory):

1. initialise the (pseudo)random numbers generators with a seed equal to 2530
2. compute and print the number of components of the array **v**
3. print the sum of all components of **v**
4. create a copy (**w**) of this array
5. change the middle component of **w** to 33.33
6. print the original array (**v**)
7. print the copy array (**w**)
8. create a shuffled copy (**s**) of the copy array (**w**) and print the results
9. find and print the largest value of **s** and the location (index) where the largest value is located. See definition of `findmax` function.
10. shuffle the original array **v** (i.e. shuffle it *in place*) and print the results
11. sort the original array **v** (*in place*) and print the results
12. compute and print the mean value of **v**. See function `mean`.

QUESTION 3: For each one of the following "comprehension loops":

```
A = [Float64(i)     for i=1:10]
B = [2.0 * i        for i=1:10]
C = [i % 2          for i=1:10]
D = [(2.0*i) % 2    for i=1:10]
E = [Float64(i+j)   for i=1:3, j=1:4]
F = [(i+j) % 2      for i=1:3, j=1:4]
G = [1 - (i+j) % 2 for i=1:3, j=1:4]
```

1. Print the results with a nice message, e.g. `println("A = ", A)` and so on
2. Convert each "comprehension loop" to the general form (non-compact one) that prints the results one after another; i.e in a *step-by-step* fashion

QUESTION 4: Create and print two empty 1D arrays (**X** and **Y**) of FPNs; i.e. two vectors. Next, create and print two variables **x** and **y** with 0.0 each; i.e. two scalars. Also, create and print two scalars **cx=0.4** and **cy=0.5**. Then:

1. Run a loop that will stop when either **x^2+y^2** is greater than **4.0** or the number of steps is greater than **50** (be careful with infinite loops! if that happens, press Ctrl+C to stop it)
2. During each step, do:
   a. create a copy of **y** named **t** (aka temporary)
   b. compute **2*x*y + cy** and assign to **y**
   c. compute **x^2 - t^2 + cx** and assign to **x**
   d. append the new **x** to the **X** array
   e. append the new **y** to the **Y** array
3. After the loop ends, print the two arrays **X** and **Y**
4. Plot the dataset with red dots and continuous lines (aka "**r.-**") and save the results in a figure named **a1_4.png**. You may use the code below for this step:

```
using PyPlot          # let's use PyPlot here
plot(X, Y, "r.-")     # plot X vs Y with red dots and lines
axis("equal")         # use equal scales
grid()                # add a grid of dotted lines
xlabel("x")           # add the horizontal label
ylabel("y")           # add the vertical label
savefig("a1_4.png")   # save figure
```

Note: you should get something like Figure 1.

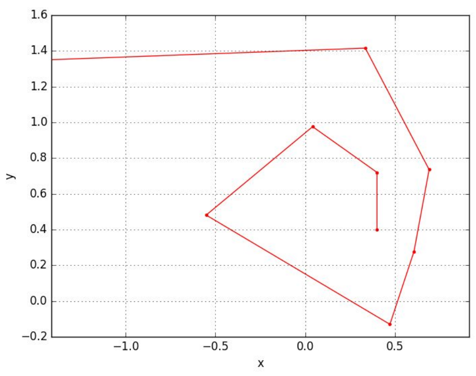


Figure 1. Graph generated by Question 3.

Section 2. Julia Sets (Fractals)

The term *fractal* comes from the Latin verb *frangere* which means "to fragment". It was coined by the mathematician Benoit Mandelbrot who is known for having initiated the fascinating field of fractals. Mandelbrot began by studying the earlier work by Gaston Julia who created the *Julia sets* in the early twentieth century. An interesting fact about fractals is that they share similarities with other nature structures such as bumps of broccoli, branches of lightening, and coastlines.

By learning Julia sets (and fractals) you will certainly improve your skills in developing recursive computations and iterations. These skills are certainly important when solving many other Engineering problems.

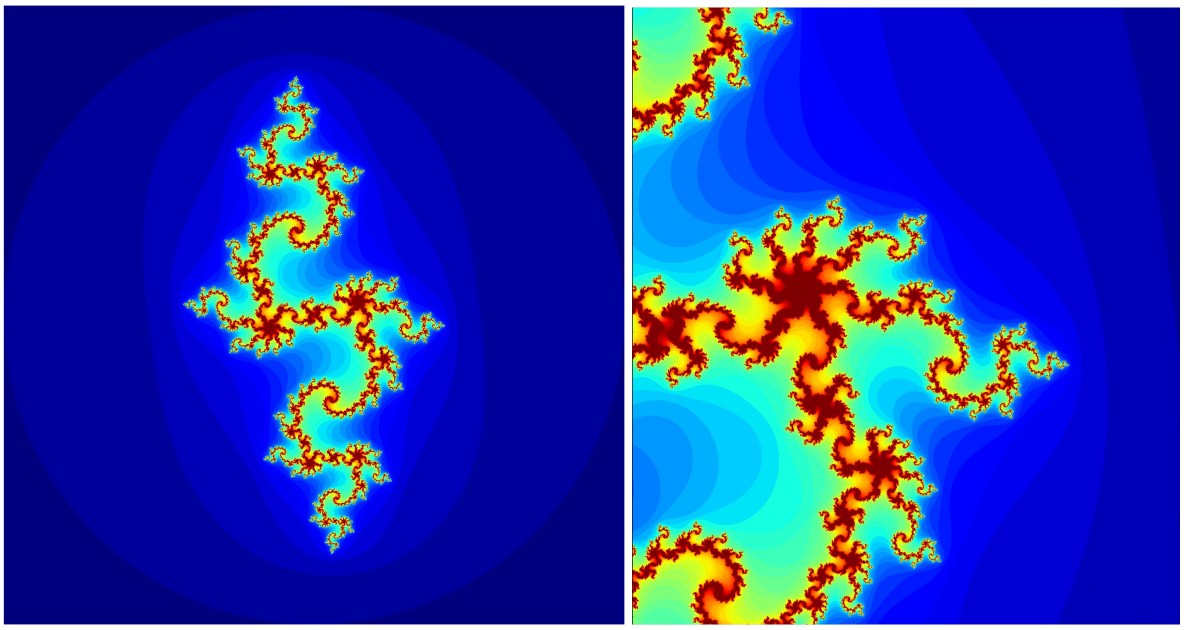
See also: https://mitpress.mit.edu/books/computational-beauty-nature

QUESTION 5: Generate and draw a Julia fractal. Steps:

1. Create a function, named **`juliaset`**, that performs the same loop that was asked by Question 4 but that now returns the number of steps taken up to convergence. Notes:
    a. In this function, the loop does not need to append x or y values to any array; i.e. the arrays are not necessary
    b. There are 4 input arguments to this function: **`x`**, **`y`**, **`cx`** and **`cy`**; all are Float64 and this must be enforced
    c. There is 1 output argument: the number of steps which must be defined as Int64
2. Just after the function is defined (after the **end** command corresponding to the **function** command plus a newline, or more), create the following auxiliary constants:
    a. **`cx = -0.835`** and
    b. **`cy = -0.2321`**
3. Define a 2D grid by creating the following auxiliary variables: **`xmin`**, **`xmax`**, **`ymin`**, **`ymax`**, **`dx`**, **`dy`** and **`N`** as follows:
    a. set `N = 100`
    b. set `xmin` and `ymin` to -2.0
    c. set `xmax` and `ymax` to 2.0
    d. compute `dx = (xmax - xmin) / N`
    e. compute `dy = (ymax - ymin) / N`
4. Create a 2D array (matrix) named **`T`** which is initially filled with zeros and that has a size equal to `N` by `N`
5. Create a nested loop with **`i`** going from **`1`** to **`N`** and **`j`** going from **`1`** to **`N`**
6. Within this nested loop, do:
    a. compute `x = xmin + i * dx`
    b. compute `y = ymin + j * dy`
    c. call the `juliaset` function with the just computed x and y values in addition to the pre-set constants cx and cy
    d. save the results of the `juliaset` function call to the 2D array **`T`**
7. Draw a figure where the pixels correspond to the values of T. This can be accomplished as follows:

```
using PyPlot          # let's use PyPlot here
imshow(T)             # creates an image with pixels defined by T
savefig("a1_5.png")   # saves the figure
```

Note: with more points in your grid (e.g. with higher N), you may get something like Figure 2.



(a) (b)

Figure 2. Julia fractal (a) Unzoomed (b) Zoomed.

Section 3. Numerical Integration (aka Quadrature)

Subsection 3.1 Trapezoidal rule

The trapezoidal rule is a numerical method to approximate definite integrals such as

$$A = \int_a^b f(x)\ dx$$

The method works by adding trapeziums ("skewed strips") that approximate the region under the $f(x)$ function as illustrated in Figure 3.

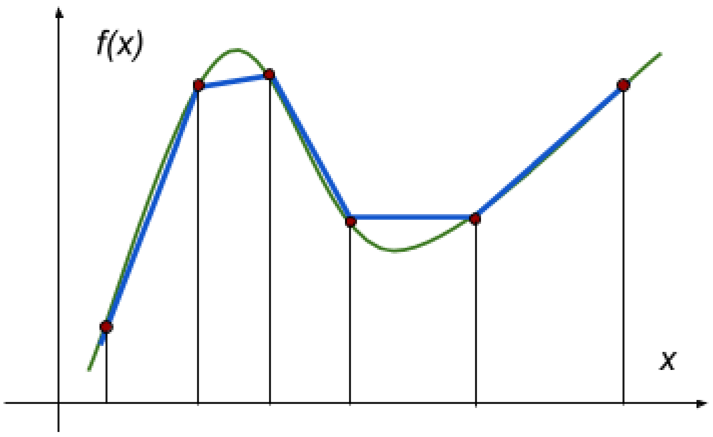


Figure 3. Approximation of f(x) with trapeziums.

By using *N* equally spaced *x* values (aka an "uniform grid") such that

$$h = \frac{b-a}{N}$$

the approximation of the above integral becomes

$$A \approx \frac{h}{2} \sum_{k=1}^{N} \left[f(x_{k+1}) + f(x_k) \right]$$

QUESTION 6: Implement a function called **`trapezoid`** that takes another function called **`f`** (that takes a Float64 and returns another Float64), two scalars **`a`** and **`b`** (both Float64), and the number of spaces **`N`** (an Int64) as input arguments and returns the area under `f(x)` from `a` to `b` using the trapezoidal method. Then:

1. define a function $f(x) = x$
2. define a function $g(x) = \sqrt{1 + sin^3(x)}$
3. set `a` and `b` equal to 0.0 and 1.0, respectively
4. create a variable holding the correct area under f(x) from a to b; i.e. `Abest = 0.5`
5. call the function trapezoid for f(x) with `N=1` and save the result in `Atrap`
6. compute the error between `Atrap` and `Abest`; i.e. `error = abs(Abest - Atrap)`
7. print Abest, Atrap and the error with a nice message
8. call again the function trapezoid but now with g(x) and with `N=20` and save the result in `Atrap` as before
9. compute the error and print Abest, Atrap and the error again

Subsection 3.2 Monte Carlo Integration

Quadrature (numerical integration) may be also performed by using **random numbers**. Nonetheless, in this way, the results are said to be stochastic and, to obtain reasonable accuracy, a large number of "iterations" (steps) must be taken. This approach is known as **Monte Carlo** integration and may certainly help with estimating complex integrals. Therefore, it is also very useful in engineering.

The (naive version of the) method is actually very simple and may be "physically" interpreted as:

In a box with a bucket inside, throw many balls at random and count the number of balls that fall inside the bucket.

Then, we can estimate the volume of the bucket V_{bucket} as follows:

$$V_{bucket} \approx V_{box} \frac{N_{balls\ Inside}}{N_{balls}}$$

Note that the above expression "makes sense", when looked in the following way:

$$\frac{V_{bucket}}{V_{box}} \approx \frac{N_{balls\ Inside}}{N_{balls}}$$

i.e. the ratio of volumes is proportional to the rate of balls.

QUESTION 7: Approximate the volume of a sphere of unit radius centered at [0,0,0] using the Monte Carlo method. Note that the volume of the box is 2*2*2. Note also that you can compute a value x along -2.0 to 2.0 with:

- `x = -1.0 + 2.0 * rand()`

Then:

1. initialise the (pseudo)random numbers generator with 2530
2. compute the numerical volume (Monte Carlo estimate) with N=1000 and save into `Vmc`
3. compute the analytical solution $V ana = 4\,\pi^2 / 3$
4. compute the absolute error between the Monte Carlo estimate and the analytical solution
5. print Amc, Aana and the error

More generally, Monte Carlo simulation can be used to approximate a multidimensional definite integral such as

$$\int_{\Omega} f(\bar{x})\, d\bar{x} \approx \frac{V}{N} \sum_{i=1}^{N} f(\bar{x}_i)$$

where $\bar{x}$ is a vector of any dimension and $\bar{x}_i$ are uniformly distributed sample points over the region Ω with (hyper)volume V.

We give now a simple derivation of the Monte Carlo Integration method in 1D. Suppose we wish to evaluate the 1D integral

$$I = \int_a^b f(x)\, dx$$

The average value f_{ave} of a function can be calculated by means of (check out "the properties of Integrals")

$$f_{ave} = \frac{1}{b-a} \int_a^b f(x)\, dx$$

We also argue that the average value of a function can be estimated by computing $f(x_i)$ and averaging the results for many (N) randomly generated (and uniform) x values in the interval $[a, b]$. The more values, the better. In this second approach:

$$f_{ave} \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i)$$

Then, by comparing these last two expressions, we can find that

$$I \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i)$$

Note that (b-a) is equivalent to the (hyper)volume V.

QUESTION 8: Implement a Monte Carlo function called **`montecarlo1d`** to approximate 1D integrals. Note that this function has the same input and output as the `trapezoid` function of Question 6. Note also that a random uniform sample **x** between **a** and **b** can be generated by means of:

- `L = b - a`
- `x = a + L * rand()`

Then:

1. initialise the (pseudo)random numbers generator with 2530
2. define the function $f(x) = e^{-x} sin(x)$
3. define the coefficients $a = 0$ and $b = 2\pi$
4. integrate $f(x)$ by using the `montecarlo1d` function from a to b and save the results in `Amc`
5. compute the analytical solution $A = \int_a^b f(x)\, dx = \frac{1}{2}(1 - e^{-2\pi})$ with N=1000 and save the results in `Aana`
6. compute the absolute error between the Monte Carlo estimate and the analytical solution
7. print Amc, Aana and the error

Section 5. Linear Interpolation

Data from experiments or from observing a physical phenomenon can at times be used to perform predictions. For example, knowing the temperature at two nearby locations may assist in finding the temperature at a third location either in between the two initially observed locations or even a little further away. This process is a simple application of geometry also known as linear interpolation/extrapolation (see Figure 4).

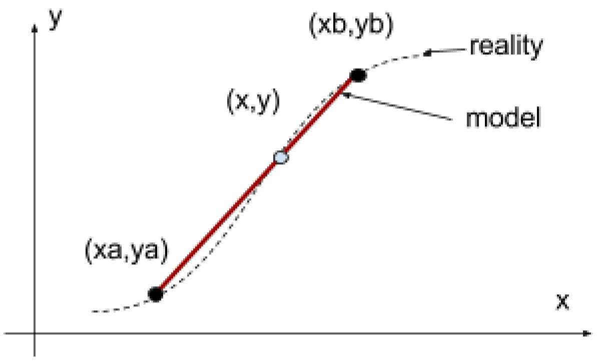


Figure 4. Linear interpolation: find *y* for given *xa*, *ya*, *xb*, *yb* and *x*.

QUESTION 9: Develop a Julia function (named **`lininterp`**) to compute *y=f(x)* for a given **set of** *x* and *y* points. For instance, X is an array storing all x values and Y is another array storing all y values. In this question, *x* values outside the *[xa, xb]* range are not allowed. Thus, make sure to return an error message if the "user" inputs something wrong. Steps:

1. derive the equation y=Interp(x) for given x, xa, ya, xb and yb (see Figure 4)
2. in the `lininterp` function, "throw" an error like this (with n being the number of points):
```
if x < X[1] || x > X[n]
    throw("x is out of range")
end
```
3. then loop over all points starting from the second one and check if $x \leq X[i]$. If so, the branch illustrated in Figure 4 is found and you can use y=interp(x)
4. for the given observations listed in Table 1, use your function to compute the velocities at times 2.6 seconds and 3.5 seconds. Print the results as follow:
```
println("y(2.6) = ", lininterp(2.6, X, Y))
println("y(3.5) = ", lininterp(3.5, X, Y))
```
5. create a "test" array with N=20 x values going from 0.0 to 5.0. Name this array `Xt`
6. create an empty "test" array named `Yt`
7. loop over the N values and compute: `Yt[i] = lininterp(Xt[i], X, Y)`
8. plot the original data with green circles ("go")
9. plot the interpolated points (Xt, Yt) with blue lines ("b-")
10. save a figure named "a1_9.png"
11. the last steps can be accomplished with
```
using PyPlot
plot(X, Y, "go")
plot(Xt, Yt, "b-")
grid()
xlabel("x")
ylabel("y")
savefig("a1_9.png")
```

Table 1. Observations

Time [h]	Velocity [km/h]
0.0	0.0
1.0	20.0
2.0	60.0
3.0	68.0
4.0	77.0
5.0	110.0

Note: you should get a graph like Figure 5.

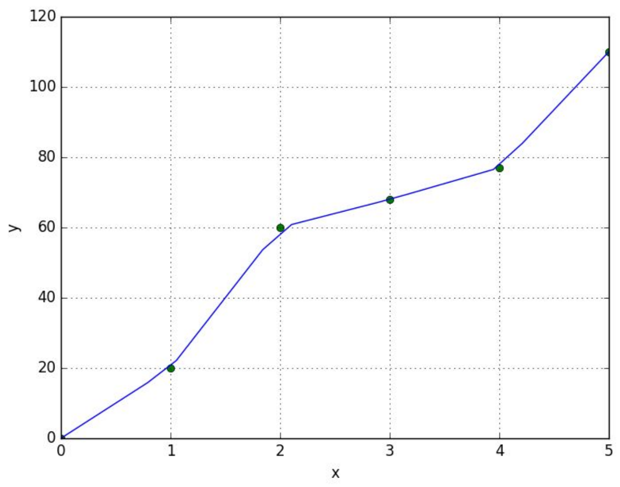


Figure 5. Output of Question 9

Section 6. Terrain Navigation

In some areas, algorithms for assisting in terrain navigation are critical; for instance, in the design of robotic spacecraft or self-driving vehicles. This section deals with a very simple terrain navigation problem where the peaks of a landscape have to be identified. Having this information, the best path could be planned (this latter goal is not part of this section). Therefore, the topic relates also with transport phenomenon in Civil Engineering. Furthermore, locating peaks (spots) or other features in a "grid" may be useful in other areas such as finding the location and movement of contaminants—an Environmental Engineering topic.

QUESTION 10: Given the following elevation data

```
5039 5127 5238 5259 5248 5310
5150 5392 5410 5401 5320 5820
5290 5560 5490 5421 5530 5831
5110 5429 5430 5411 5459 5630
4920 5129 4921 5821 4722 4921
5023 5129 4822 4872 4794 4862
```

implement a Julia code to find the peaks. Note: you may ignore the edges and corners. Steps:

1. create a matrix `E` with the elevation data
2. create a matrix of zeros called `F` with the flags (1 or 0) indicating whether there is a peak or not

3. loop over rows and columns (ignoring edges) and set the flag F[i,j] = 1 when (i,j) is a peak
4. print the resulting flags matrix `F`
5. plot the peaks using

```
using PyPlot
imshow(F)
savefig("a1_10.png")
```

Note: you should get something like Figure 6.

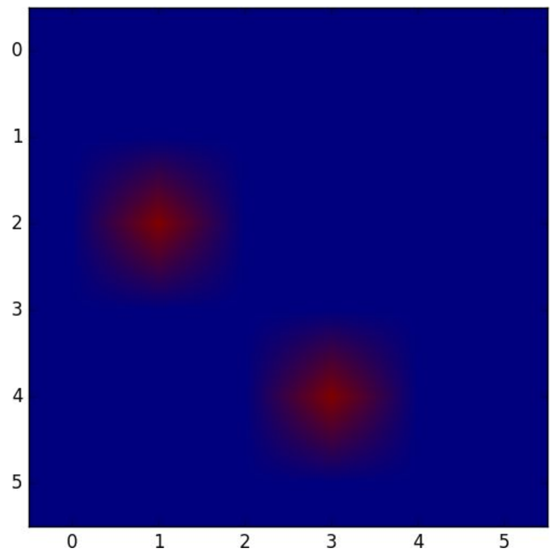


Figure 6. Output of question 10