Repeated Evaluation

Exercises: MAT2319 – Introduction to Programming with Julia
(Hanoi University of Science)

Otober 20, 2021

The following exercises help students develop basic programming skills, focusing on using control structures (*if*, *while*, *for*) and functions. Each question should be solved using a separate function.

1. Approximate the exponential function using the Maclaurin series. We know that[1]

$$\begin{aligned} e^x &= 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \\ &= \sum_{n=0}^{\infty} \frac{x^n}{n!}. \end{aligned}$$

   (a) Write a function which approximates this infinite series up to a predefined natural numbers N, as follows:

```
function myExp(x, N)
  ...
end
```

   (b) Suppose that we use the built-in `exp(x)` function of Julia to compute the "exact" value of the exponential function, and we want that our own `myExp` can approximate `exp(1.0)` accurately up to 10^{-8}. What is the smallest N?

2. Consider the following series

$$\begin{aligned} S &= 1 + \frac{1+2}{2!} + \frac{1+2+3}{3!} + \frac{1+2+3+4}{4!} + \cdots \\ &= \sum_{n=1}^{\infty} \frac{n(n+1)}{2(n!)} \end{aligned}$$

   (a) What is the exact value of this series? You don't need a computer to answer this question, use your mathematical skill, with paper and pencil.

[1]Note that by convention $0! = 1$.

(b) Write a function which approximates this infinite series up to a predefined natural number N, as follows:

```
function mySum(N)
  ...
end
```

3. Find the greatest common divisor (gcd) of two natural numbers.

```
function gcd(a, b)
  ...
end
```

Generate randomly a vector of 100 natural numbers which are less than 1000 (use the built-in `rand` function of Julia) and finds the gcds of all different pairs of these numbers.

4. A natural number is called perfect if it is the sum of their divisors except itself. For example, 6 is a perfect number because $6 = 1+2+3$. Similarly, 28 is a perfect number since $28 = 1+2+4+7+14$. Perfect numbers are quite rare. Write a function which tests a given number is perfect or not:

```
function isPerfect(n)
  ...
end
```

Then use the built-in function `filter` of Julia to list all perfect numbers less than one billion. Report the computation time on your computer.

5. A pair of natural numbers (a, b) where $a < b$ is called friendly if one number is the sum of proper divisors of the other number. For example, $(220, 284)$ is a friendly pair. Write a function which tests a given pair is friendly or not.

```
function isFriendly(a, b)
  ...
end
```

List all friendly pairs less than one billion. Report the computation time on your computer.

6. Write a function which tests the primality of a natural number:

```
function isPrime(n)
  ...
end
```

List all prime numbers less than one billion. Report the computation time on your computer. *Hint: for a faster computation, you should think of what a non-prime (compound) number is rather than what a prime number is.*

7. Given a vector v of numbers, write a function which finds the maximal number without using the `maximum` function of Julia.

```
function findMax(v)
  ...
end
```

8. Find the smallest number of natural numbers whose sum of squares is 2021.

9. Find the number of digits of a given natural number n. For example, $n = 123321$ has 6 digits.

```
function numberOfDigits(n)
  ...
end
```

10. A natural number is divisible by 3 if the sum of their digits is divisible by 3. Use this divisibility rule to check whether a number is divisible by 3 or not.

```
function isDivisibleBy3(n)
  ...
end
```

11. What is the divisibility rule by 11? Similar to the previous question, write a function to implement the check:

```
function isDivisibleBy11(n)
  ...
end
```

12. Write a function which factors a given natural number n into prime numbers. For example, the factorization of 20 is $[2, 2, 5]$; that of 5100 is $[2, 2, 3, 5, 5, 17]$.

```
function factorize(n)
  ...
end
```

13. To compute the square root of a positive number, we can apply the Babylonian method.[2]. The method is also known as Heron's method, after the first-century Greek mathematician Hero of Alexandria who gave the first explicit description of the method in his AD 60 work Metrica.

    The basic idea is that if x is an overestimate to the square root of a non-negative real number S then $\frac{S}{x}$ will be an underestimate, or vice versa, and so the average of these two numbers may reasonably be expected to provide a better approximation. This is equivalent to using Newton's method to solve $x^2 - S = 0$. If x is our initial guess of $\sqrt{S}$ and ϵ is the error such that $S = (x+\epsilon)^2$, then we have the approximation

$$\epsilon = \frac{S - x^2}{2x + \epsilon} \approx \frac{S - x^2}{2x}, \text{ since } \epsilon \text{ is small.}$$

    From this we have

$$x + \epsilon \approx x + \frac{S - x^2}{2x} = \frac{S + x^2}{2x} = \frac{\frac{S}{x} + x}{2}.$$

    The Babylonian method is as follows:

    (a) $x_0 = a$ (pick an initial guess, a can be arbitrary)

    (b) $x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right), \forall n = 0, 1, \ldots$

    (c) $\sqrt{S} = \lim_{n\to\infty} x_n$

    Write a function to compute the square root of a given number S using the method above with N iterative steps.

```
function Babylonian(S, N)
  ...
end
```

    Compare the approximation error of this method with the result of the built-in function $\sqrt{S}$ of Julia with different N. Plot a figure showing approximation errors with respect to N.

14. The numbers $n!$ increase extremely rapidly with n. For example,

$$10! = 3,628,800$$
$$15! = 1,307,674,368,000$$

    and 100! has 158 digits. Hence from either the theoretical or the computational point of view, it is important to know Stirling's formula:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \exp\left(\frac{\theta_n}{12n}\right), \quad 0 < \theta_n < 1.$$

[2] https://en.wikipedia.org/wiki/Methods_of_computing_square_roots

Write a function to approximate factorials of n for all $n = 10, 11, \dots, 100$ and plot the relative errors with respect to different chosen θ_n. Use the built-in function `factorial(n)` for the exact factorials.

15. To find the binary representation (base 2) of a natural number n in base 10, we repeatedly divide it by 2 and record the remainders in the reverse order. Write a function which binarizes a natural number.

```
function binarize(n)
  ...
end
```

16. A bank offers an interest rate of $r\%$ per term (aka period) for its saving accounts. Interest are compounded; that is the obtained interest of period t is added to the balance to compute the interest of period $t + 1$, for $t = 1, 2, \dots$. Suppose that your initial balance is 1, the rate is $r = 0.08$/year.

    (a) You lend your money to the bank for n years. What is your final balance that the bank need to pay you at due date? Draw the plot of amounts with respect to n, for n ranges from 1 to 100.

    (b) Suppose that you plan to have at least s as your balance at the due date, for example $s = 3$. How long do you have to wait for?

    (c) Suppose that the bank offers different packages with different terms and rates, of 4% per six months, of 2% per quarter, and of 0.67% per month. You plan to deposit for 5 years. What package should you choose?

17. Write a function which returns unique numbers of a given vector of numbers. For example `uniq([1, 3, 2, 2, 3, 1])` is `[1,3,2]`. This is similar to the built-in `unique()` function of Julia.

```
function uniq(v)
  ...
end
```

18. Given a matrix of shape $m \times n$ (m rows, n columns). Write some basic functions that do the following:

    (a) Suppose that the matrix is square, $m = n$. Write a function to compute the sum of elements on its main diagonal. Write another function to compute the sum of elements on its back-diagonal.

    (b) Again, suppose that $m = n$. Write a function to compute the sum of elements on the upper-right of the matrix, that is all elements above the main diagonal.

    (c) Write a function to compute the sum of elements on the same columns of the matrix. The result should be a vector of n elements.

(d) Same as the question right above but change columns to rows. Now the result should be a vector of m elements.

19. Write a function to fill in a matrix representing a Pascal triangle up to a predefined n. For example, with $n = 4$, we have

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1

And for $n = 5$, we should have the following matrix.

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

Empty cells contains zeros.

20. Write a function to fill in a matrix of shape $m \times n$ with natural numbers from 1 to $m * n$ in a clockwise cyclic way. For example, with $m = n = 3$, we have the matrix

1	2	3
8	9	4
7	6	5

With $m = 4$ and $n = 5$, we have the following matrix:

1	2	3	4	5
14	15	16	17	6
13	20	19	18	7
12	11	10	9	8

Hint: Is your programme easy for modification? If we want to fill the numbers in the anti-clockwise direction, do we need to rewrite most of the lines of code?

21. A (gray-scale) image can be seen as a matrix. After setting appropriate values for all elements of a matrix A, we can display it as image using the function `heatmap(A)` of the Julia `Plots` library. In this problem, we will generate Julia sets. The term Julia here has different meaning, not the Julia programming language. You should read the article "Julia set"

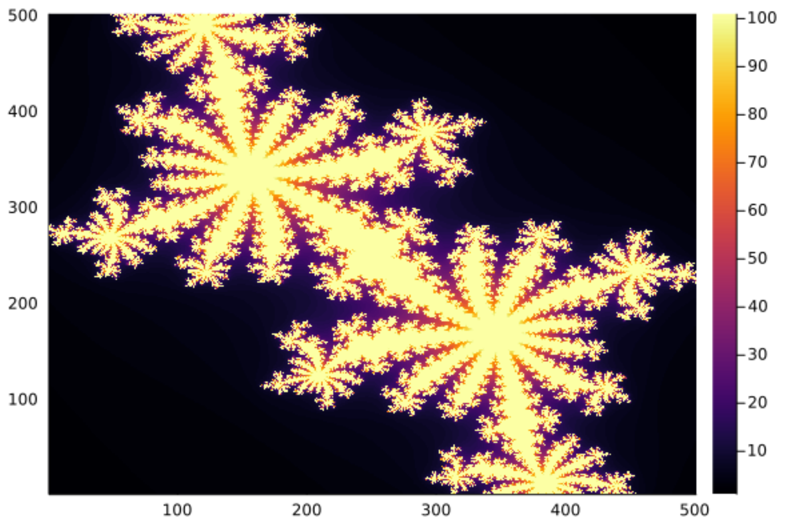
in the Wikipedia.[3] Given two complex numbers z_0 and c, we define the following iterative computation:

$$z_{n+1} = z_n^2 + c, \quad \forall n = 0, 1, 2, \ldots$$

This is a dynamical system known as *quadratic map*. Given a specific choice of z_0 and c, we have a sequence of complex numbers $z_1, z_2, z_3, \ldots$ which is called the *orbit* of z_0. A large range of orbit patterns is possible, depending on different choices of z_0 and c.

- For a given fixed c, most choices of z_0 yield orbits that tend towards infinity; that is, the modulus $|z_n|$ grows without limit as n increases.
- For some values of c, certain choices of z_0 yield orbits that eventually go into a periodic loop.
- Some starting values yield orbits that appear to dance around the complex plane, apparently at random (*a chaos*).

The starting values z_0 make up the *Julia set* of the map, denoted J_c. We will write a programme which visualizes the *filled-in Julia set*, denoted K_c, which is the set of all z_0 with orbits which do not tend towards infinity. The normal Julia set J_c is the edge of the filled-in Julia set K_c. The figure below shows a Julia set for one particular value of c.



The figure contains self-similar structures, which are called *fractals*. Follow the steps below to draw fractals.

[3]https://en.wikipedia.org/wiki/Julia_set

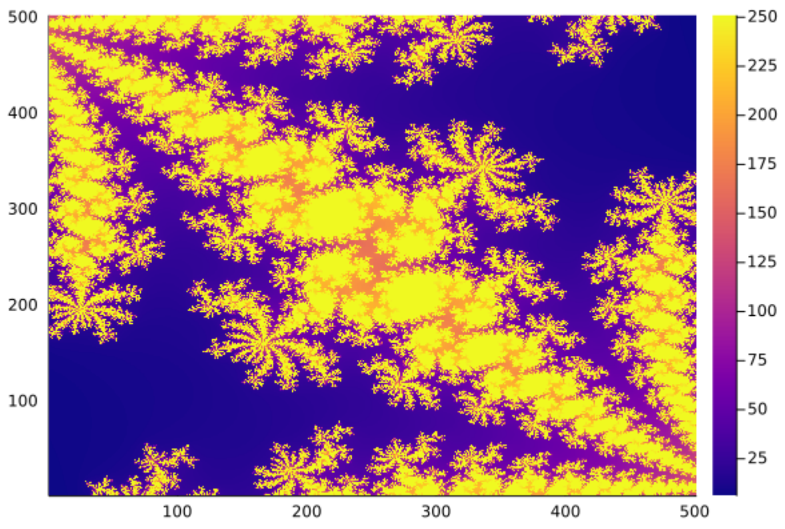
(a) It has been shown that if $|z_n| > 2$ for some n then it is guaranteed that the orbit will tend to infinity. The value of n for which this becomes true is called the *escape velocity* of a particular z_0. Write a function which returns the escape velocity of a given z_0 and c: `n = escapeVelocity(z0, c, N)`, where `N` is the maximum allowed escape velocity.[4]

(b) To generate the filled-in Julia set, write the following function `M = julia(zMax, c, N)`, where `zMax` is the maximum of the imaginary and complex parts of the various z_0's for which we will compute escape velocities. The `c` and `N` values are as described above; and `M` is the matrix that contains the escape velocity of various z_0's.

   i. First, make a matrix `Z` of shape 500×500 that contains complex numbers with real part between `-zMax` and `zMax`, and imaginary part between `-zMax` and `zMax`. Make the imaginary part vary along the column of this matrix.

   ii. For each element of `Z`, compute the escape velocity by calling the function `escapeVelocity` above and store it at the same location in the matrix `M`. When done, the matrix `M` should be the same size as `Z` and contains escape velocities with values between `1` and `N`.

   iii. Run your `julia` function with various `zMax`, `c`, and `N` values to generate various fractals. To display it, use the function `heatmap(M)`. The figure above was generated by setting `zMax=1`, `c=-0.297491+i*0.641051`, and `N=100`. Using the same `c` but setting `zMax=0.35` and `N=250`, we have the following fractal:



[4]If the modulus of z_n does not exceed 2 for $n < N$, return N as the escape velocity. This will prevent infinite loop.

22. Write a function `multiply(A,B)` which implements the multiplication of two matrices of compatible sizes.

23. A square matrix of shape $n \times n$ is called magic if it contains n^2 numbers from 1 to n^2, each number appears once, and the sum of its columns, the sums of its rows, the sum of its diagonal and the sum of its back-diagonal are all the same value ($\equiv n(n^2+1)/2$). For example, the following is a 3×3 magic matrix:

7	1	6
3	5	7
4	9	2

   (a) Given a square matrix of shape $n \times n$. Write a function `isMagic(A)` to check whether a matrix is magic or not.

   (b) An algorithm to fill in a magic matrix of odd size is as follows. Put number 1 at the center cell of the first row. Repeatedly put subsequent numbers $2, 3, \ldots, n^2$ at upper-right cells of the previous ones:

0	$k+1$
k	0

   Note three special cases:

   - If number $k+1$ is at row 0 then it is moved to row n.
   - If number $k+1$ is at column $n+1$ then it is moved to column 1.
   - If number $k+1$ is at the top-right cell (column n and row 1) or if its place has already occupied by a smaller number then it is placed to the cell right below k, as follows:

0	X
k	0
$k+1$	0

   You can observe this filling method in the 5×5 magic matrix as below:

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

   Note that the last number n^2 is always at the center cell of the last row. By symmetry, we can obtain more magic matrices from an original one by rotating it 90, 180 or 270 degrees. Fill yourself manually a magic matrix of shape 7×7 and verify its magic property. Write a function `magicMatrix(n)` to create a magic matrix of size n.

24. Fill two arrays u and v with 20 random integers in the range 1 to 10. Test whether or not all the numbers occurring in v also occur in u; in other words, v is a subset of u.

25. Generate 100,000 random integers in the range 1 to n. Use an array c to keep count of the frequency of occurrence of each integer (i.e. use $c[1]$ to keep track of the number of occurrences of the integer 1, $c[2]$ for the integer 2 and so on). Print out the number of times each integer from 1 to n was generated.

26. This exercise is about the sieve of Eratosthenes.[5] This is a nice exercise that involves writing a program to generate all the prime numbers less than a given number n. The Eratosthenes' sieve method is as follows:

    (a) Start with the first number not crossed off (i.e. 2). Call this k.

    (b) Cross off (or "sieve out") all the multiples of k that are less than or equal to n, since these are obviously not prime.

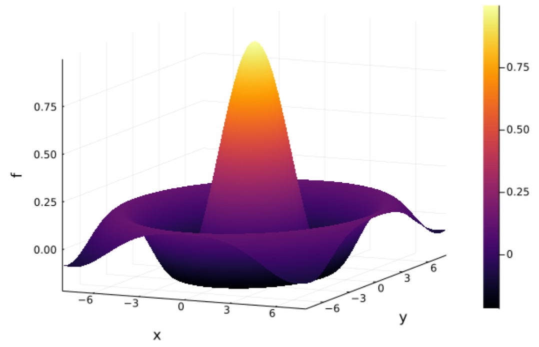
    (c) Repeat two steps above with the next number k that is not crossed off. Terminate when you have reached n.

    The prime numbers are all the numbers not sieved out. First, using paper and pencil to run Eratosthenes sieve by hand on the numbers $2 \dots 50$. Then write a program to carry out this method, use a boolean array with all elements initially set to false. Crossing off a number k involves setting the kth array element to true. How many prime numbers are there less than 1 million?

27. The `plot` function of the Julia Plots library can plot a 3d visualization of a two-variable function with an option `st=:surface`. For example, the Sombrero function is defined as

$$f(x,y) = \frac{\sin(\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}, \quad x, y \in \mathbb{R}.$$

    Write a program to visualize a two-variable function in predefined ranges of x and y.



[5] `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

28. A square matrix of size n is singular if its determinant is zero, or equivalently, its rank is less than n. Suppose that you are given a random matrix whose values are uniformly distributed in the range $[0, 1]$. Investigate the singularity behaviour of such matrices as n increases.