General hints for acing the following set of problems. The algorithms that you need to code for solving these exercises involve linear algebra operations such as the multiplication of a matrix and a vector, the dot product of two vectors, or computing the norm of a vector. It may be tempting to code them as element–by–element operations (e.g. a dot product can be coded as a loop over the elements of the two vectors). If you will follow this road, your code will be long, unreadable, and it will be very easy to make mistake and introduce bugs. In addition, because Python is not a low-level language such as Fortran77 or C, the code will also be *very* slow. In Python you should try as much as possible to use built-in library functions that allow you to express those operations as a single instruction. In order to do so, you must adhere to some basic coding standards:

- Don't use python lists to represent vectors and matrices. Use numerical python's arrays. Arrays are not available in vanilla python, you need to import numpy. Or, better, import pylab (an "umbrella" library which imports numpy, scipy, and matplotlib).
- Learn slice indexing: e.g. `https://www.tutorialspoint.com/numpy/numpy_indexing_and_slicing.htm`

Once you are in control of these basics, then you can do the following:

- Multiply the $n \times n$ matrix A by the n-elements vector $\boldsymbol{b}$: `dot(A,b)`
- Dot product between the two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$: `dot(u,v)`
- Extract the diagonal of the $n \times n$ matrix A: `diag(A)`; the result is an n-elements vector.
- Create a diagonal matrix from the n-elements vector $\boldsymbol{u}$: `diag(u)`; the result is an $n \times n$ matrix.
- Create a diagonal matrix that has the same diagonal as A: `diag(diag(A))`; (obviously!)
- Compute the dot product of the $i-$th line of A with the vector $\boldsymbol{x}$: `dot(A[i], x)`
- Compute the dot product of the first j elements of the $i-$th line of A with the first j elements of $\boldsymbol{x}$: `dot(A[i,:j], x[:j])`
- Substitute the $i-$th line of A with the vector $\boldsymbol{x}$: `A[i] = x`
- Compute the infinity norm of $\boldsymbol{x}$: `norm(x, ord=inf)`
- Find the number of row and columns of A or the number of elements of $\boldsymbol{x}$: `A.shape`; `x.shape`; these return a python tuple.
- You can invert a matrix with `inv(A)`, but for these exercises don't use it.... essentially you are learning how to code matrix inverters. But it might be handy for debugging purposes, so it's good to know that you have it.

- Remember that if A and B are $n \times n$ matrices, then `A*B` is the element–by–element multiplication. The standard row–by–column matrix multiplication is `dot(A,B)`. (Yes, in python "`dot`" is some sort of swiss army knife of linear algebra multiplication, look at its documentation.)

Happy coding!

Exercise 1.

Write a python function that solves for x the system of linear equations

$$PLU\boldsymbol{x} = \boldsymbol{b}$$

by backsubstitution. Here P, L, U are the LU decomposition with partial pivoting of an $n \times n$ matrix A, and $\boldsymbol{b}$ is an $n-$elements vector. To test your function, generate matrices of different size with the "`create_matrix`" and "`create_matrix_2`" python functions (see attached `homework5.py`) and use the "`lu`" python function from scipy.linalg to perform the PLU decomposition. Generate the vector b in a random way with the instruction

```
b = rand(n)
```

(you need "`from pylab import *`" at the beginning of your code in order to have "`rand`"). Check the correctness of your solution by multiplying A by x and subtract from the result b (that is, by computing the backward error).

Exercise 2.

Write a python function that solves the linear system of equations

$$A\boldsymbol{x} = \boldsymbol{b}$$

by means of Jacobi iterations. As the exit criterion, use the size of the relative backward error. To test your function, generate matrices of different size with the "`create_matrix`" python function (see attached `homework5.py`). Generate the vector b in a random way with the instruction

```
b = rand(n)
```

(you need "`from pylab import *`" at the beginning of your code in order to have "`rand`"). You must obtain a relative backward error of no more than 10^{-12} in the infinity norm.

Exercise 3.

Just as exercise 2, but using the Gauss-Seidel method. In view of exercise 4, it's best if you write a code that implements the s.o.r. method, and then use it with $\omega = 1$. Also in view of exercise 4, in addition to an exit criterion based on the relative backward error, you should also code an exit criterion based on the number of iterations.

Exercise 4.

Solve the same problem as exercise 2, but with the s.o.r. (successive over-relaxation) method. The matrix A must be 100×100. The initial guess $\boldsymbol{x}_0$ must be a vector whose components are all equal to 1. Use exactly 30 s.o.r. iterations. Plot the relative backward error as a function of the overrelaxation parameter ω, for $\omega \in [1, 1.2]$.