Python exercises for Chapter 5. Session 1

Instructions for uploading the exercises

1. File names:
   - **Names of python scripts** are given according to the numbering of the list of exercises. Like `exercise_1.py`, `exercise_2.py`, etc.
   - **Names of output files** where the outputs are written to follow a similar naming format:
     - `exercise_1.txt`, if using the functions `print`, and `open` and `close`,
     - `exercise_1.npz`, if using the function `numpy.savez`, etc.
   - The **name of the zip file** must be `Surname1Surname2Name`, without white spaces, and excluding non-ASCII characters, such as tildes and *ñ*. For instance,

     Lucía Martín Cañas must write `MartinCanasLucia.zip`

     Include only the `exercise_*.py` files in your zip.
2. Ckeck that:
   - **Each script runs without errors**. To do this, in Spyder, or in any other IDE, restart the kernel (to clean variables) and run the script in the command window.
   - **The solution, and only the solution, is printed to the required output file.** Do not print intermmediate results in the final version of the script.

Exercises

1. Write a script for computing the solution of a linear system of equations, $Ax = b$, by the iterative method of Jacobi in the form of formula (5.3) of the Handbook. To do this, define the function `jacobi` with
   - Input: The matrix, A, the independent vector, b, the initial guess, x_0 (optional), the tolerance, tol (optional), and the maximum number of iterations, $maxiter$ (optional).
   - Output: The solution, x, and the number of iterations performed until reaching the tolerance, $numiter$.

   For computing the difference between consecutive iterations, use the infinity norm (with function `norm` from `numpy.linalg`).

   Since there are some inputs which are optional, you must give some default values in the definition of the function. Set the following: `x0 = False`, `tol = 1.e-6`, and `maxiter = 1000`.

   In the case in which `x0` is not `False`, the function will use the `x0` provided as argument. Otherwise, it should use `x0 = np.zeros_like(b, dtype=np.float)` (you have to code this).

   Use it for
   $$A = \begin{pmatrix} 4 & 3 & -1 \\ 4 & 5 & -3 \\ -2 & 3 & -6 \end{pmatrix}, \quad b = \begin{pmatrix} 5 \\ 3 \\ 1 \end{pmatrix}.$$

   Compute the solution and the number of iterations for the following calls to `jacobi`:

(a) `x1, numiter1 = jacobi(A, b)`.

(b) `x2, numiter2 = jacobi(A, b, False, tol)`, with `tol = 1.e-4`.

(c) `x3, numiter3 = jacobi(A, b, x0, tol, maxiter)`, with `tol = 1.e-9`, `x0 = 100*np.ones_like(b)`, `maxiter = 1000`.

Finally, save them through `numpy.savez('exercise_1', x1=x1, x2=x2, x3=x3, numiter1=numiter1, numiter2=numiter2, numiter3=numiter3)`.

Hint: To check if `x0` is passed or not, you can use the python function `type` in an `if-else` statement. Also, recall that the assignment `x_old = x` gives a reference to `x`, not a copy of `x`. To make a copy, use `x_old = x.copy()`.

To check your results, these are the number of iterations:

(a) 186 iterations.

(b) 120 iterations.

(c) 308 iterations.

2. Write a script for computing the solution of a linear system of equations, $Ax = b$, by the iterative method of Gauss-Seidel in the form

$$x_i^{(k)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)}).$$

To do this, define the function `gauss_seidel` with

- Input: The matrix, A, the independent vector, b, the initial guess, x_0 (optional), the tolerance, tol (optional), and the maximum number of iterations, $maxiter$ (optional).
- Output: The solution, x, and the number of iterations performed until reaching the tolerance, $numiter$.

For computing the difference between consequtive iterations, use the infinity norm (with function `norm` from `numpy.linalg`).

Since there are some inputs which are optional, you must pass some default values. Set the following: `x0 = False`, `tol = 1.e-6`, and `maxiter = 1000`.

In the case in which `x0` is not `False`, the function will use the `x0` provided as argument. Otherwise, it should use `x0 = np.zeros_like(b, dtype=np.float)` (you have to code this).

Use it for

$$A = \begin{pmatrix} 4 & 3 & -1 \\ 4 & 5 & -3 \\ -2 & 3 & -6 \end{pmatrix}, \quad b = \begin{pmatrix} 5 \\ 3 \\ 1 \end{pmatrix}.$$

Compute the solution and the number of iterations for:

(a) `x1, numiter1 = gauss_seidel(A, b)`.

(b) `x2, numiter2 = jacobi(A, b)` (of Exercise 1).

Finally, save them through

`numpy.savez('exercise_2', x1=x1, x2=x2, numiter1=numiter1, numiter2=numiter2)`.

To check your results, these are the number of iterations:

(a) 111 iterations.

(b) 186 iterations.

3. Write a script for computing the solution of the partial differential equation

$$\frac{\partial u(t,x)}{\partial t} - \frac{\partial^2 u(t,x)}{\partial x^2} = 0,$$

with the initial data $u_0(x) = x(1-x)(1+0.1\sin(16\pi x))$, and the boundary conditions $u(t,0) = u(t,L) = 0$. That is, for each time step τ, solve the system

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ -r & 1+2r & -r & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -r & 1+2r & -r & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -r & 1+2r & -r & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -r & 1+2r & -r \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_2 \\ y_3 \\ \vdots \\ y_{m-2} \\ y_{m-1} \\ y_m \end{pmatrix} = \begin{pmatrix} b_0 \\ b_2 \\ b_3 \\ \vdots \\ b_{m-2} \\ b_{m-1} \\ b_m \end{pmatrix}$$

where $r = \tau/h^2$, and

- The mesh is $\{x_0, x_1, \ldots, x_m\}$.
- The values of u in consecutive time steps are $b_j = u(t_i, x_j)$ and $y_j = u(t_{i+1}, x_j)$.

Follow these steps:

(a) Introduce the data: $L = 1$, $T = 0.01$, $\tau = 0.001$, 101 nodes por the mesh of $(0, L)$ (and, thus, $h = 0.01$), and u_0 as specified above.

(b) Construct the matrix of the system, A, using the numpy function `numpy.diag(v,k)`, where `v` is a vector (a diagonal), and `k` is the index of the diagonal: `k=0` is the main diagonal, `k=-1` is the diagonal below, and `k=1` is the diagonal above.

(c) Modify the non-zero elements of the first and last rows of A according to (5.13).

(d) Make a loop in time, advancing τ in each step, until the final time T is reached. In the loop, solve the linear system of equations given above with the numpy function `numpy.linalg.solve`. Observe that b is the solution of the previous time step, and y the solution of the new time step.

Finally, if `u` is the approximated solution at time T, save it using

```
numpy.savez('exercise_3', u).
```

To check your results, plot the approximated solution and compare to Figure 3.

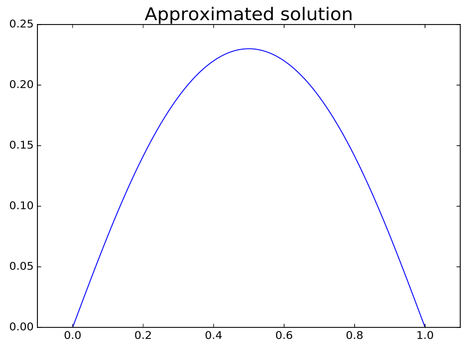

Figure : Exercise 3