University of Nottingham
UK | CHINA | MALAYSIA

MATH2019 Introduction to Scientific Computation

— Coursework 2 (10%) —

Submission deadline: ***3pm, Tuesday, 6 December 2022***

Note: This is currently **version 3-3** of the PDF document. (26th November, 2022)
Further questions will be added to this PDF in the upcoming weeks.

This coursework contributes **10%** towards the overall grade for the module.

Rules:

• Each student is to submit their own coursework.

• You are allowed to work together and discuss in small groups (2 to 3 people), but *you must write your own coursework and program all code by yourself.*

• Please be informed of the UoN Academic Misconduct Policy (incl. plagiarism, false authorship, and collusion).

Coursework Aim and Coding Environment:

• In this coursework you will develop Python code related to algorithms that solve linear systems, and you will study some of the algorithms' behaviour.

• As discussed in the lecture, you should write (and submit) plain Python code (.py), and you are strongly encouraged to use the **Spyder IDE** (integrated development environment). Hence you should *not* write IPython Notebooks (.ipynb), and you should *not* use Jupyter.

How and Where to run Spyder:

• Spyder comes as part of the Anaconda package (recall that Jupyter is also part of Anaconda). Here are three options on how to run Spyder:

(1) You can choose to install Anaconda on you personal device (if not done already).

(2) You can open Anaconda on any University of Nottingham computer.

(3) You can open a UoN Virtual Desktop on your own personal device, which is virtually the same as logging onto a University of Nottingham computer, but through the virtual desktop. The simply open Anaconda. Here is further info on the UoN Virtual Desktop.

• The A18 Computer Room in the Mathematical Sciences building has a number of computers available, as well as desks with dual monitors that you can plug into your own laptop.

Time-tabled Support Sessions (Mondays 12-1pm, Fridays 11-12noon, ESLC C13 Computer Room):

• You can work on the coursework whenever you prefer.

• We especially encourage you to work on it during the (optional) time-tabled drop-in sessions.

Piazza (https://piazza.com/class/l7z5sbys7m74fq):

• You are allowed and certainly encouraged(!) to also ask questions using Piazza to obtain clarification of the coursework questions, as well as general Python queries.

• However, when using Piazza, please ensure that you are not revealing any answers to others. Also, please ensure that you are not directly revealing any code that you wrote. Doing so is considered Academic Misconduct.

• When in doubt, please simply attend a drop-in session to meet with a PGR Teaching Assistant or the Lecturer.

Helpful resources:

• Python 3 Online Documentation – Spyder IDE (integrated development editor) – NumPy User Guide – Matplotlib Quick Start Guide – Moodle: Core Programming 2021-2022 – Moodle: Core Programming 2020-2021

• You are expected to have basic familiarity with Python, in particular: logic and loops, functions, NumPy and matplotlib.pyplot. Note that it will always be assumed that the package numpy is imported as np, and matplotlib.pyplot as plt.

Some Further Advice:

• Write your code as neatly and read-able as possible so that it is easy to follow. Add some comments to your code that indicate what a piece of code does. Frequently run your code to check for (and immediately resolve) mistakes and bugs.

• Coursework Questions with a "$\star$" are more tedious, but can be safely skipped, as they don't affect follow-up questions.

Submission Procedure:

• Submission will open after 28 November 2022.

• To submit, simply upload the requested .py-files on Moodle. (Your submission will be checked for plagiarism using *turnitin*.)

• Your work will be marked (mostly) automatically: This is done by running your functions and comparing their output against the true results.

Getting Started:

• Download the contents of the "Coursework 2 Pack" folder from Moodle into a single folder.
(More files may be added in later weeks.)

▶ Gaussian elimination without pivoting

(**This is a warm up exercise. No marks will be awarded for it. Its solution is available on Moodle as file `warmup_solution.py`.** Further clarification of what you need to do to obtain marks is given in Question 2 below.)

Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{b} \in \mathbb{R}^{n}$. Suppose that $\boldsymbol{A}$ is such that there exists a unique solution $\boldsymbol{x} \in \mathbb{R}^{n}$ to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ and that forward elimination can be performed without any row interchanges to arrive at an augmented matrix to which backward substitution can be applied to find $\boldsymbol{x}$.

1 Open the file `warmup.py`. Note that this file already contains an unfinished function with the following first line:

```
def no_pivoting(A,b,n,c)
```

This function returns the output `M` which is of type `numpy.ndarray` and has shape $(\mathtt{n}, \mathtt{n}+1)$. The input `A` is of type `numpy.ndarray` and has shape $(\mathtt{n}, \mathtt{n})$ and represents the square matrix $\boldsymbol{A}$. The input `b` is of type `numpy.ndarray` and has shape $(\mathtt{n}, 1)$ and represents the column vector $\boldsymbol{b}$. The input `n` is an integer such that $\mathtt{n} \geq 2$. The input `c` is an integer such that $1 \leq \mathtt{c} \leq \mathtt{n}-1$, and it is used to (prematurely) stop the forward elimination algorithm; see details below.

- A pseudocode algorithm for performing forward elimination without row interchanges on the matrix $\boldsymbol{M} \in \mathbb{R}^{n \times n+1}$ is the following:

For $i = 1$ to $n-1$ do

For $j = i+1$ to n do

Set $g = m_{j,i}/m_{i,i}$

Set $m_{j,i} = 0$

For $k = i+1$ to $n+1$ do

Set $m_{j,k} = m_{j,k} - g m_{i,k}$

End do

End do

End do

In the above pseudocode algorithm, the entries of the matrix $\boldsymbol{M}$ are $m_{i,j}$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n+1$. However, the elements in M (a numpy.ndarray with shape $(\mathtt{n}, \mathtt{n}+1)$) are M[i,j] for $\mathtt{i} = 0, 1, \ldots, \mathtt{n}-1$ and $\mathtt{j} = 0, 1, \ldots, \mathtt{n}$.

- Complete the no_pivoting function so that the output M is an array representing the augmented matrix arrived at by starting from the augmented matrix $[\ \boldsymbol{A} \quad \boldsymbol{b}\ ]$ and performing forward elimination without row interchanges until all of the entries below the main diagonal in the first c columns are 0.
- The warmup.py file contains an unfinished function with the following first line:

```
def backward_substitution(M,n)
```

The input M is of type numpy.ndarray and has shape $(\mathtt{n}, \mathtt{n}+1)$ and represents an augmented matrix of the form $[\ \boldsymbol{U} \quad \boldsymbol{v}\ ]$ where $\boldsymbol{U} \in \mathbb{R}^{n \times n}$ is an upper triangular matrix with all entries on its main diagonal being nonzero and $\boldsymbol{v} \in \mathbb{R}^n$. The input n is an integer such that $\mathtt{n} \geq 2$.

- Complete the backward_substitution function so that the output x is of type numpy.ndarray and has shape $(\mathtt{n}, 1)$, and represents the solution $\boldsymbol{x}$ to $\boldsymbol{U}\boldsymbol{x} = \boldsymbol{v}$. **Hint:** See Lecture 4 Slides, Algorithm 6.1, Steps 8–9 for a pseudocode of backward substitution.
- The warmup.py file contains the following finished function:

```
def no_pivoting_solve(A,b,n)
```

The input A is of type numpy.ndarray and has shape $(\mathtt{n}, \mathtt{n})$ and represents the matrix $\boldsymbol{A}$. The input b is of type numpy.ndarray and has shape $(\mathtt{n}, 1)$ and represents the vector $\boldsymbol{b}$. The input n is an integer such that $\mathtt{n} \geq 2$. The output x is of type numpy.ndarray and has shape $(\mathtt{n}, 1)$, and represents the solution $\boldsymbol{x}$ to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ computed using Gaussian elimination without pivoting. One may note that the no_pivoting_solve function calls the no_pivoting and `backward_substitution` functions in order to do this.

- Test your `no_pivoting` and `backward_substitution`, and the `no_pivoting_solve` functions by running the `main_warmup.py` file. For the `A`, `b` and `n` in the `main_warmup.py` file, with `c`=1, the output from the `no_pivoting` function should be:

```
[[  1.   -5.    1.    7.]
 [  0.   50.   10.  -64.]
 [  0.   35.   -6.  -31.]]
```

For the `A`, `b` and `n` in the `main_warmup.py` file, with `c`=2, the output from the `no_pivoting` function should be:

```
[[  1.    -5.     1.     7. ]
 [  0.    50.    10.   -64. ]
 [  0.     0.   -13.    13.8]]
```

The output from the `backward_substitution` function and the `no_pivoting_solve` function obtained by running the `main_warmup.py` file should be:

```
[[ 2.72307692]
 [-1.06769231]
 [-1.06153846]]
```

(turn page)

▶ **Gaussian elimination with scaled partial pivoting**

Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$. Suppose that $\boldsymbol{A}$ is such that there exists a unique solution $\boldsymbol{x} \in \mathbb{R}^n$ to $\boldsymbol{Ax} = \boldsymbol{b}$ and that forward elimination with scaled partial pivoting can be performed to arrive at an augmented matrix to which backward substitution can be applied to find $\boldsymbol{x}$.

2 A pseudocode algorithm for finding the smallest positive integer p such that

$$|v_p| = \max_{j\in\{1,2,\ldots,n\}} |v_j|$$

is the following:

Set $m = 0$

Set $p = 0$

For $j = 1$ to n do

  If $|v_j| > m$

    Set $m = |v_j|$

    Set $p = j$

  End If

End do

- Open the file `systemsolvers.py`. The `systemsolvers.py` file contains an unfinished function with the following first line:

```
def find_max(M,s,n,i)
```

The input `M` is of type `numpy.ndarray` and has shape $(\mathtt{n}, \mathtt{n}+1)$. The input `s` is of type `numpy.ndarray` and has shape $(\mathtt{n},)$ and is such that all of its elements are positive. The input `n` is an integer such that $\mathtt{n} \geq 2$. The input `i` is a nonnegative integer such that $\mathtt{i} \leq \mathtt{n}-2$.

- Complete the `find_max` function so that the output `p` is of type `int` and is the smallest integer such that $\mathtt{p} \geq \mathtt{i}$ and

$$\frac{|\mathtt{M[p,i]}|}{\mathtt{s[p]}} = \max_{\mathtt{j}\in\{\mathtt{i},\mathtt{i}+1,\ldots,\mathtt{n}-1\}} \frac{|\mathtt{M[j,i]}|}{\mathtt{s[j]}}.$$

Hint: To calculate the absolute value, one can use the command `abs`, `np.abs` or `np.absolute`; see numpy.org/doc.

- Test your `find_max` ~~scaled_partial_pivoting and spp_solve functions~~ by running the `main.py` file. The output from the `find_max` function obtained by running the `main.py` file should be $\mathtt{p} = 1$ (because $\frac{|\mathtt{M[j,0]}|}{\mathtt{s[j]}}$ is 0.2, 0.5 and 0.5, for $j = 0, 1, 2$, respectively).

Sentence corrected on **18 Oct, 11am** ←

► **Assessment**

When submitting your coursework, you will only be asked to upload your `systemsolvers.py` file.

Marks can be obtained for your `find_max` function definition for generating the required output, for certain set(s) of inputs for $\{\mathtt{M}, \mathtt{s}, \mathtt{n}, \mathtt{i}\}$. The correctness of the following will be checked: **[8 / 40]**

- The `type` of output `p`
- The value of output `p`.

---

3 • The `systemsolvers.py` file contains an unfinished function with the following first line:

```
def scaled_partial_pivoting(A,b,n,c)
```

The input `A` is of type `numpy.ndarray` and has shape $(\mathtt{n}, \mathtt{n})$ and represents the square matrix $\boldsymbol{A}$. The input `b` is of type `numpy.ndarray` and has shape $(\mathtt{n}, 1)$ and represents the column vector $\boldsymbol{b}$. The input `n` is an integer such that $\mathtt{n} \geq 2$. The input `c` is an integer such that $1 \leq \mathtt{c} \leq \mathtt{n} - 1$, and it is used to (prematurely) stop the forward elimination with scaled partial pivoting algorithm; see details below.

• Complete the `scaled_partial_pivoting` function so that the output `M` is of type `numpy.ndarray` and has shape $(\mathtt{n}, \mathtt{n}+1)$, and represents the augmented matrix arrived at by starting from the augmented matrix $[\ \boldsymbol{A} \quad \boldsymbol{b}\ ]$ and performing forward elimination with scaled partial pivoting (as described in the Linear Systems: Direct Methods II slides) until all of the entries below the main diagonal in the first `c` columns are 0. The `scaled_partial_pivoting` function should call the `find_max` function in order to do this.

Hint: Note that `M[[i,p],:]=M[[p,i],:]` can be used to interchange the row `M[i,:]` and the row `M[p,:]`. Also note that `s[[i,p]]=s[[p,i]]` can be used to interchange `s[i]` and `s[p]`.

• The `systemsolvers.py` file contains an unfinished function with the following first line:

```
def spp_solve(A,b,n)
```

The input `A` is of type `numpy.ndarray` and has shape $(\mathtt{n}, \mathtt{n})$ and represents the matrix $\boldsymbol{A}$. The input `b` is of type `numpy.ndarray` and has shape $(\mathtt{n}, 1)$ and represents the vector $\boldsymbol{b}$. The input `n` is an integer such that $\mathtt{n} \geq 2$.

• Complete the `spp_solve` function so that the output `x` is of type `numpy.ndarray` and has shape $(\mathtt{n}, 1)$, and represents the solution $\boldsymbol{x}$ to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ computed using Gaussian elimination with scaled partial pivoting. In order to do this, the `spp_solve` function should call the `scaled_partial_pivoting` function and the `backward_substitution` function. Note that a completed version of the `backward_substitution` function is in the `warmup_solution` module that is imported in `systemsolvers.py`.

• Test your `scaled_partial_pivoting` and `spp_solve` functions by running the `main.py` file. The output from the `scaled_partial_pivoting` function obtained by running the `main.py` file should be:

```
[[ 10.    0.   20.    6. ]
 [  0.   -5.   -1.    6.4]
 [  0.   10.  -11.    1. ]]
```

The output from the `spp_solve` function obtained by running the `main.py` file should be:

```
[[ 2.72307692]
 [-1.06769231]
 [-1.06153846]]
```

► Assessment

When submitting your coursework, you will only be asked to upload your `systemsolvers.py` file.

Marks can be obtained for your `scaled_partial_pivoting` function definition for generating the required output, for certain set(s) of inputs for $\{\mathtt{A}, \mathtt{b}, \mathtt{n}, \mathtt{c}\}$. The correctness of the following will be checked: **[10 / 40]**

- The `type` of output `M`
- The `np.shape` of output `M`
- The values of output `M`.

Marks can be obtained for your `spp_solve` function definition for generating the required output, for certain set(s) of inputs for $\{\mathtt{A}, \mathtt{b}, \mathtt{n}\}$. The correctness of the following will be checked:

- The `type` of output `x`
- The `np.shape` of output `x`
- The values of output `x`.

Note that in marking your work, different input(s) may be used.

(turn page)

► $\boldsymbol{PLU}$ **factorisation**

Given an appropriate matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, a pseudocode algorithm for obtaining a permutation matrix $\boldsymbol{P}$, a lower triangular matrix $\boldsymbol{L}$ and an upper triangular matrix $\boldsymbol{U}$ such that $\boldsymbol{A} = \boldsymbol{PLU}$ is the following:

Set $\boldsymbol{P} = \boldsymbol{I}$, the $n \times n$ identity matrix
Set $\boldsymbol{L}$ to be the $n \times n$ zero matrix
Set $\boldsymbol{U} = \boldsymbol{A}$
For $i = 1$ to $n - 1$ do
    Find the smallest integer s such that $i \leq s \leq n$ and $|u_{s,i}| > 10^{-15})$
    If $s \neq i$
        Interchange row i and row s of $\boldsymbol{P}$
        Interchange row i and row s of $\boldsymbol{L}$
        Interchange row i and row s of $\boldsymbol{U}$
    End if
    For $j = i + 1$ to n do
        Set $l_{j,i} = u_{j,i}/u_{i,i}$
        Set $u_{j,i} = 0$
        For $k = i + 1$ to n do
            Set $u_{j,k} = u_{j,k} - l_{j,i}u_{i,k}$
        End do
    End do
End do
Set $\boldsymbol{P} = \boldsymbol{P}^T$
Set $\boldsymbol{L} = \boldsymbol{L} + \boldsymbol{I}$

Note: As a first attempt to Question 4, you can ignore the blue parts related to the row interchanges (for $\boldsymbol{P}$, $\boldsymbol{L}$ and $\boldsymbol{U}$). Indeed, without the blue parts the algorithm then simply implements the $\boldsymbol{LU}$ factorisation. There is a dedicated test that should work without the blue parts (see below).

4 • The `systemsolvers.py` file has been updated to include an unfinished function with the following first line:

```
def PLU(A,n)
```

The input `A` is of type `numpy.ndarray` and has shape $(\mathtt{n}, \mathtt{n})$ and represents an $\mathtt{n} \times \mathtt{n}$ matrix $\boldsymbol{A}$ which is assumed to be such that the above pseudocode algorithm can be used to obtain a $\boldsymbol{PLU}$ factorisation of $\boldsymbol{A}$. The input `n` is an integer such that $\mathtt{n} \geq 2$.

• Complete the `PLU` function so that it implements the above pseudocode algorithm for obtaining matrices $\boldsymbol{P}$, $\boldsymbol{L}$ and $\boldsymbol{U}$ such that $\boldsymbol{A} = \boldsymbol{PLU}$. The output `P` is of type `numpy.ndarray` and has shape $(\mathtt{n}, \mathtt{n})$ and is an array representing the permutation matrix $\boldsymbol{P}$, the output `L` is of type `numpy.ndarray` and has shape

$(\mathtt{n}, \mathtt{n})$ and is an array representing the lower triangular matrix $\boldsymbol{L}$, and the output U is of type `numpy.ndarray` and has shape $(\mathtt{n}, \mathtt{n})$ and is an array representing the upper triangular matrix $\boldsymbol{U}$.

• In the pseudocode algorithm, the entries of the matrices $\boldsymbol{P}$, $\boldsymbol{L}$ and $\boldsymbol{U}$ are, respectively, $p_{i,j}$, $l_{i,j}$ and $u_{i,j}$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$. However, the elements in P, L and U are, respectively, P[i,j], L[i,j] and U[i,j] for $\mathtt{i} = 0, 1, \ldots, \mathtt{n} - 1$ and $\mathtt{j} = 0, 1, \ldots, \mathtt{n} - 1$.

Hint: Note that `np.transpose(P)` returns the transpose of P.

Hint: Recall from Question 3, the hint on how to interchange rows in a matrix.

• Test your PLU function by running the updated `main.py` file. There are two test matrices A1 and A2: For A1, the PLU function should do a factorisation without doing row exchanges, hence the output should be (for P, L, U, respectively):

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 2. -2.  1.]]
[[ 1. -1.  2.]
 [ 0. -1. -3.]
 [ 0.  0. -8.]]
```

For A2, the PLU function should do a factorisation involving row exchanges, and the output should be:

```
[[1. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]]
[[1. 0. 0.]
 [2. 1. 0.]
 [1. 0. 1.]]
[[ 1. -1.  2.]
 [ 0.  2. -2.]
 [ 0.  0. -3.]]
```

▶ **Assessment**

When submitting your coursework, you will only be asked to upload your `systemsolvers.py` file.

Marks can be obtained for your PLU function definition for generating the required output, for certain set(s) of inputs for $\{\mathtt{A}, \mathtt{n}\}$. The correctness of the following will be checked: **[12 / 40]**

• The values of outputs P, L, and U.

Note that in marking your work, different input(s) may be used.

(turn page)

▶ Jacobi method

Suppose $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$, and $\boldsymbol{x} \in \mathbb{R}^n$ is the unique solution to $\boldsymbol{Ax} = \boldsymbol{b}$. Let $\boldsymbol{x}^{(k)}$ be the approximation to $\boldsymbol{x}$ obtained after performing k iterations of the Jacobi method, starting from the initial approximation $\boldsymbol{x}^{(0)}$.

5 The `systemsolvers.py` file has been updated to include an unfinished function with the following first line:

```
def Jacobi(A,b,n,x0,N):
```

- The input `A` is of type `numpy.ndarray` and has shape $(\mathtt{n},\mathtt{n})$ and represents the square matrix $\boldsymbol{A}$. The input `b` is of type `numpy.ndarray` and has shape $(\mathtt{n},1)$ and represents the column vector $\boldsymbol{b}$. The input `n` is an integer such that $\mathtt{n} \geq 2$. The input `x0` is of type `numpy.ndarray` and has shape $(\mathtt{n},1)$ and represents the initial approximation $\boldsymbol{x}^{(0)}$. The input `N` is a positive integer that is the number of iterations to be performed.
- Complete the `Jacobi` function so that the output `x_approx` is of type `numpy.ndarray` and has shape $(\mathtt{n},\mathtt{N}+1)$ and is such that, for $\mathtt{j} = 0, 1, \ldots, \mathtt{n}-1$, $\mathtt{x\_approx}[\mathtt{j},0] = x^{(0)}_{\mathtt{j}+1}$ and $\mathtt{x\_approx}[\mathtt{j},\mathtt{k}] = x^{(\mathtt{k})}_{\mathtt{j}+1}$ for $\mathtt{k} = 1, 2, \ldots, \mathtt{N}$.

Hint: You can choose to implement directly the element-based form of the Jacobi method ~~Gauss-Seidel method~~ (see Lecture 7 Slides and Notes), or you can choose to implement a vector-based form of the Jacobi method ~~Gauss-Seidel method~~ (see also Lecture 7). The following functions could be useful: `np.dot`, `np.tril`, `np.triu`, `np.diag`, `np.linalg.inv`, `np.matmul` or the '@' operator for matrix multiplication.

Sentence corrected on **25 Oct, 12noon** ← ←

- Test your `Jacobi` function by running the updated `main.py` file. The output from the `Jacobi` function obtained by running the `main.py` file should be:

```
[[ 0.     12.     12.375]
 [ 0.      1.5     3.75 ]
 [ 0.      6.      6.375]]
```

▶ Assessment

When submitting your coursework, you will only be asked to upload your `systemsolvers.py` file.

Marks can be obtained for your `Jacobi` function definition for generating the required output, for certain set(s) of inputs for $\{\mathtt{A}, \mathtt{b}, \mathtt{n}, \mathtt{x0}, \mathtt{N}\}$. The correctness of the following will be checked: **[6 / 40]**

- The `type` of output `x_approx`
- The `np.shape` of output `x_approx`
- The values of output `x_approx`.

(turn page)

6 The systemsolvers.py file contains an unfinished function with the following first line:

```python
def Jacobi_plot(A,b,n,x0,N):
```

- The input A is of type numpy.ndarray and has shape $(\mathtt{n},\mathtt{n})$ and represents the square matrix $\boldsymbol{A}$. The input b is of type numpy.ndarray and has shape $(\mathtt{n},1)$ and represents the column vector $\boldsymbol{b}$. The input n is an integer such that $\mathtt{n} \geq 2$. The input x0 is of type numpy.ndarray and has shape $(\mathtt{n},1)$ and represents the initial approximation $\boldsymbol{x}^{(0)}$. The input N is a positive integer that is the number of iterations to be performed.
- Complete the Jacobi_plot function so that it plots (in the same figure using the same axes) $\|\boldsymbol{x}-\boldsymbol{x}^{(k)}\|_\infty$ for $k = 0, 1, \ldots, \mathtt{N}$ and $\|\boldsymbol{x}-\boldsymbol{x}^{(k)}\|_2$ ~~and~~ $\cancel{\|\boldsymbol{x}-\boldsymbol{x}^{(k)}\|_\infty}$ for $k = 0, 1, \ldots, \mathtt{N}$. The approximations $\boldsymbol{x}^{(k)}$ to $\boldsymbol{x}$ should be computed by calling the Jacobi function. The solution $\boldsymbol{x}$ to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ should be computed by calling the no_pivoting_solve function. Note that a completed version of the no_pivoting_solve function is in the warmup_solution module that is imported in systemsolvers.py.

Sentence corrected on **26 Oct, 4:30pm** ←

Hint: To calculate the l_∞ norm of a one-dimensional array v, one can use the command np.linalg.norm(v,np.inf); see numpy.org/doc.
Hint: To calculate the l_2 norm of a one-dimensional array v, one can use the command np.linalg.norm(v,2); see numpy.org/doc.
Hint: If w is of type numpy.ndarray and has shape $(\mathtt{n},\mathtt{m})$ and p is a nonnegative integer less than m then w[0:n-1,p] is of type numpy.ndarray and has shape $(\mathtt{n},)$ not $(\mathtt{n},1)$.

- Test your Jacobi_plot function by running the updated main.py file.

▶ **Assessment**
When submitting your coursework, you will only be asked to upload your systemsolvers.py file.

Marks can be obtained for your Jacobi_plot function definition for generating the required output, for certain set(s) of inputs for $\{\mathtt{A},\mathtt{b},\mathtt{n},\mathtt{x0},\mathtt{N}\}$. The correctness of the following will be checked: **[4 / 40]**
- The points plotted.

Note that in marking your work, different input(s) may be used.

▶ Finished!