Workshop 12: Solutions of Algebraic Equations

Beth Wingate

The purpose of this workshop is to learn how to solve systems of algebraic equations using Python. This builds on the last workshop, where we studied how to use matrices and vectors. In this workshop we are concerned with the solution of equations of the form,

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{1}$$

where $\boldsymbol{A}$ is a matrix, $\boldsymbol{b}$ is a vector, and $\boldsymbol{x}$ is a vector of unknowns. The goal is to find the unknown values. This type of problem comes up repeatedly in scientific applications. In addition to solving linear algebraic equations, you will also have a brief introduction to linear algebra, such as using Python to compute the determinant of $\boldsymbol{A}$. As a part of this experience you will learn to use numpy linear algebra functions and compare them to your own hand calculations which will serve as unit tests.

Systems of linear equations

In this section we look at how systems of algebraic equations transform into the form Eq (1). To begin let us consider a simple example[1].

$$4y + 10z = 20, \tag{2}$$
$$2y + 4z = 4. \tag{3}$$

[1] Recall from mathematics how to solve this system by hand. You will find that there is only one possible solution which is $y = -10$ and $z = 6$

Think about how you would solve a problem by hand if there were 10 unknowns rather than y and z? What about 100 unknowns? Consider writing these equations in terms of the form in Eq (1),

$$\underbrace{\begin{bmatrix} 4 & 10 \\ 2 & 4 \end{bmatrix}}_{\boldsymbol{A}} \underbrace{\begin{bmatrix} y \\ z \end{bmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{bmatrix} 20 \\ 4 \end{bmatrix}}_{\boldsymbol{b}} \tag{4}$$

To solve this problem for general systems, you can use the inverse of $\boldsymbol{A}$, denoted by $\boldsymbol{A^{-1}}$. First multiply both sides of Eq (1) by $\boldsymbol{A^{-1}}$,

$$\boldsymbol{A^{-1}A}\boldsymbol{x} = \boldsymbol{A^{-1}b}, \tag{5}$$
$$\boldsymbol{I}\boldsymbol{x} = \boldsymbol{A^{-1}b}, \tag{6}$$
$$\boldsymbol{x} = \boldsymbol{A^{-1}b}, \tag{7}$$

where $\boldsymbol{I}$ is the identity matrix. Let's solve our example problem by hand. First compute the inverse of matrix $\boldsymbol{A}$ using techniques you learned in mathematics[2],

[2] Note: Use $\boldsymbol{A^{-1}A} = \boldsymbol{I}$

$$\boldsymbol{A^{-1}} = \begin{bmatrix} -1 & 5/2 \\ 1/2 & -1 \end{bmatrix}. \tag{8}$$

Next, carry out the step in Eq (7) to find $\boldsymbol{x}$.

$$\boldsymbol{A^{-1}b} = \begin{bmatrix} -1 & 5/2 \\ 1/2 & -1 \end{bmatrix} \begin{bmatrix} 20 \\ 4 \end{bmatrix} = \begin{bmatrix} -20 + (5 \times 4)/2 \\ (1 \times 20)/2 - 4 \end{bmatrix} = \begin{bmatrix} -10 \\ 6 \end{bmatrix}. \tag{9}$$

The vector, on the right of Eq (9) is the answer presented in margin note (1). While this is straight-forward to do by hand for a 2x2 system, sometimes you will have very large systems of equations where the matrices are generated using data, and therefore it is useful to know how to write a program to solve a large system of equations. To begin we will first solve this problem using the `linalg` package that comes as a part of `numpy`. To see a list of available functions for manipulating matrices, and vectors, do a web search on 'numpy linalg' – scroll down in your web browser to look at the types of functions that are available. Some of the functions that we used in Workshop 1 belong to this package, but now we'll be looking at the functions: `numpy.linalg.inv`, `numpy.linalg.solve`. Below you can see a program that carries out the operations that you did above by hand.

```
import numpy as np
A    = np.array([[4, 10], [2, 4]])
b    = np.array([20, 4])
Ainv = np.linalg.inv(A)    # Compute the inverse
x    = np.dot(Ainv, b)      # Compute the unknowns by using Eq (7).
print(x)
```

Look at the syntax we used to access the matrix inverse. The first reference, `np`, refers to the numpy package, the second word refers to the `linalg` part of numpy and the last word refers to the function itself `inv` (the inverse of a matrix). Notice that we've used the `numpy.dot` function that we used in Workshop 1. What would happen if you typed `x = Ainv*b` into your program rather than using `numpy.dot`? Why does this behave differently than using `numpy.dot`? You will be responsible for understanding this type of difference on your May exam.

Let's consider trying the function `numpy.linalg.solve`. A short description of `numpy.linalg.solve` is at the right. [3]

The program at right uses this function to find the vector x (values of y and z). Try this now yourself.

[3] numpy.linalg.solve(a, b) takes two arguments, a and b, where a is a matrix, b is a vector. It returns a vector, in this case our x.

```
import numpy as np
A = np.array([[4, 10], [2, 4]])
b = np.array([20, 4])
x = np.linalg.solve(A, b)
print(x)
```

Poorly conditioned matrices

As you learned in mathematics, not all matrices can be inverted. Some square matrices are singular, which means their determinant

is zero. When the determinant of a matrix is nearly zero, this means if you change the vector $\boldsymbol{b}$ just a little, you could get a very different solution for $\boldsymbol{x}$. In that case we say that the matrix $\boldsymbol{A}$ is poorly conditioned. One way to tell if this the case of our matrix $\boldsymbol{A}$, is to compute its **determinant**. For the case when $\boldsymbol{C}$ is a 2x2 matrix[4]

$$\det(C) = c_{22} \times c_{11} - c_{21} \times c_{12}, \tag{10}$$

Compute the determinant for our example matrix (Ans. is -4). To compute this using Python, consider the following example code:

```
import numpy as np
A = np.array([[4, 10], [2, 4]])
detA = np.linalg.det(A)
if np.abs(detA) < .01:
    print("detA is very small, solutions may be unreliable!", detA)
else:
    print("The determinant of matrix A is not close to zero", detA)
```

We shall experiment with detecting matrices that might give unreliable solutions in the exercises, below.

[4] Recall our notation from last week, where c_{ij} is the element of matrix $\boldsymbol{C}$ where i indicates the row, and j the column. So a 2x2 matrix $\boldsymbol{C} = \begin{bmatrix} c_{11}c_{12} \\ c_{21}c_{22} \end{bmatrix}$.

Remember to use unit tests

Reminder: always test your understanding of functions using Unit Tests. One of the advantages of participating in Python programming community is having access to the wealth of already-programmed and tested modules and functions, but care has to be taken that you understand the differences between Python's default processes and those of module functions, as well as how you might program something on your own. To make sure you are using module functions correctly use unit tests. You will explore this idea in the questions below.

Questions

1. As a scientist you are sometimes faced with choices as to which package you might use to do some problem. In this question you will write a Python code to compute the solutions of the following systems of equations. In parts 1c and 1d, there are three systems to solve (e.g. $\boldsymbol{A}_3\boldsymbol{x}_3 = \boldsymbol{b}_3$, $\boldsymbol{A}_3\boldsymbol{x}_4 = \boldsymbol{b}_4$, and $\boldsymbol{A}_3\boldsymbol{x}_5 = \boldsymbol{b}_5$)
   - Your program should use comments to explain what the code is doing.
   - Your program should first check the determinant of the operator matrix and inform the user if it is well-conditioned or ill-conditioned. [5] (i.e. $|det(A)| >> 0$ or $|det(A)| \approx 0$) *Note:* use

[5] **Determinant answers:**
$det(\boldsymbol{A}_1) = -0.6$
$det(\boldsymbol{A}_2) = 23992077593.05$
$det(\boldsymbol{A}_3) = 2.00e-05$
$det(\boldsymbol{A}_4) = 3.02e-49$

`numpy.linalg.det`

- Then your program should solve the linear system, and display the result [6] *Note:* Use numpy functions `numpy.linalg.solve` or `numpy.linalg.inv`.
- Finally, you should check to see if your answer is accurate by multiplying your answer, vector $\boldsymbol{x}$ by your matrix to see if you get exactly $\boldsymbol{b}$. Use comments within your code to indicate where you are testing the accuracy of the answer.

(a)

$$\boldsymbol{A}_1 = \begin{bmatrix} 1. & 2. & 5. \\ 2. & 4. & 8. \\ .5 & .7 & .6 \end{bmatrix} \boldsymbol{b}_1 = \begin{bmatrix} 3. \\ 3. \\ 3. \end{bmatrix}$$

(b)

$$\boldsymbol{A}_2 = \begin{bmatrix} 100. & 2. & 5. & 10. \\ 2. & 400. & 8. & 1. \\ .5 & .7 & 600. & 2. \\ .1 & 10. & 42. & 1000. \end{bmatrix} \boldsymbol{b}_2 = \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix}$$

(c)

$$\boldsymbol{A}_3 = \begin{bmatrix} 1 & 1 \\ 1 - .00001 & 1 + .00001 \end{bmatrix} \boldsymbol{b}_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \boldsymbol{b}_4 = \begin{bmatrix} 1.001 \\ 1 \end{bmatrix} \boldsymbol{b}_5 = \begin{bmatrix} 1.01 \\ 1 \end{bmatrix}$$

(d)

$$\boldsymbol{A}_4 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \\ 6 & 7 & 8 & 9 & 10 \\ 7 & 8 & 9 & 10 & 11 \end{bmatrix} \boldsymbol{b}_6 = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \boldsymbol{b}_7 = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \boldsymbol{b}_8 = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 6 \end{bmatrix}$$

Note: $\boldsymbol{A}_3$ and $\boldsymbol{A}_4$ are both ill-conditioned, compute the solutions for each vector $\boldsymbol{b}_i$ to compare how different the solutions are if the $\boldsymbol{b}$ vector is changed a small amount.

[6] **Linear system solutions:**

$\boldsymbol{A}_1\boldsymbol{x}_1 = \boldsymbol{b}_1$

$$\boldsymbol{x}_1 = \begin{bmatrix} 24.5 \\ -14.5 \\ 1.5 \end{bmatrix}$$

$\boldsymbol{A}_2\boldsymbol{x}_2 = \boldsymbol{b}_2$

$$\boldsymbol{x}_2 = \begin{bmatrix} 0.978 \\ 0.242 \\ 0.165 \\ 0.091 \end{bmatrix}$$

$\boldsymbol{A}_3\boldsymbol{x}_3 = \boldsymbol{b}_3$

$$\boldsymbol{x}_3 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

$\boldsymbol{A}_3\boldsymbol{x}_4 = \boldsymbol{b}_4$

$$\boldsymbol{x}_4 = \begin{bmatrix} 50.5005 \\ -49.4995 \end{bmatrix}$$

$\boldsymbol{A}_3\boldsymbol{x}_5 = \boldsymbol{b}_5$

$$\boldsymbol{x}_5 = \begin{bmatrix} 500.505 \\ -499.495 \end{bmatrix}$$

$\boldsymbol{A}_4\boldsymbol{x}_6 = \boldsymbol{b}_6$

$$\boldsymbol{x}_6 = \begin{bmatrix} 8.571e-01 \\ 4.448e+14 \\ -1.779e+15 \\ 2.224e+15 \\ -8.896e+14 \end{bmatrix}$$

$\boldsymbol{A}_4\boldsymbol{x}_7 = \boldsymbol{b}_7$

$$\boldsymbol{x}_7 = \begin{bmatrix} 7.143e-01 \\ 6.672e+14 \\ -2.669e+15 \\ 3.336e+15 \\ -1.334e+15 \end{bmatrix}$$

$\boldsymbol{A}_4\boldsymbol{x}_8 = \boldsymbol{b}_8$

$$\boldsymbol{x}_8 = \begin{bmatrix} 2.857e-01 \\ 1.473e+15 \\ -4.768e+15 \\ 5.115e+15 \\ -1.821e+15 \end{bmatrix}$$

You get these solutions with Numpy version 1.9.2. If you haver another version of Numpy, you may get different solutions for part 1d.