NUMERICAL SOLUTION AND SIMULATION IN ENGINEERING WITH PYTHON

By:

Ayoob Salari

Chapter 5

Table of Contents

SECTION 1

Advanced Numerical Methods for Systems of Linear Equations

This chapter presents more advanced topics on numerical methods. The eigen decomposition and SVD are reviewed. SVD is applied to a data compression problem. Finally, a linear programming problem is posed and solved, demonstrating how to take a graphical model description and formulate it algebraically and in matrix form, before solving it using Python's `linprog` routine.

Let us begin by recalling the eigenvalue and eigenvalue vector computations from chapter 2.

```
import numpy as np
A = np.array([[25,15,25],[17,11,17],[13,9,18]])
w, v = np.linalg.eig(A)
print('A=', A)
print('EigenValues=', w)
print('EigenVectors=', v)
 # Checking if matrix is positive semi-definite (
   eigenvalues are positive)
print( "PSD:", w > 0 )
```

```
A =  [[25 15 25]
      [17 11 17]
      [13  9 18]]
EigenValues =  [50.25918498  3.09871455  0.64210047]
EigenVectors =  [[-0.73873316 -0.50807554  0.33121536]
                 [-0.51046385 -0.46572948 -0.91588622]
```

```
                [-0.44011359  0.72453799  0.22682332]]
PSD: [ True  True  True]
```

1.1 SVD

The *singular value decomposition* (SVD) allows us to see so many properties of whatever matrix we are decomposing. What is even better is that it works on any matrix just like the polar decomposition on a complex number. As such, it is the appropriate tool for analyzing a mapping from one vector space into another vector space, possibly with a different dimension. Most systems of simultaneous linear equations fall into this second category.

The SVD is defined as follows: Any m-by-n matrix of complex numbers A can be factored into:

$$A = USV^T$$

where U is a unitary m-by-m matrix and the columns of the U are the eigenvectors of AA^T . Likewise, V is unitary n-by-n matrix and the columns of the V are the eigenvectors of A^TA. The matrix S is diagonal and it is the same size as A. Its diagonal entries, also called $\Sigma = [\sigma_1, \cdots, \sigma_r]$, are the square roots of the nonzero eigenvalues of both AA^T and A^TA. They are the singular values of matrix A and they fill the first r places on the main diagonal of S, where r is the rank of A.

Example 1.1 SVD

Let $A = \begin{bmatrix} 2 & 2 \\ -1 & 1 \end{bmatrix}$. You are going to compute its SVD via two eigendecompositions.

a) Compute AA^T and find its eigendecomposition.

b) Compute A^TA and find its eigendecomposition.

c) Compute S from the eigenvalue matrix of either of the above.

d) Given the values you have already computed, write down the SVD of A as USV^T.

e) Use the command `numpy.linalg.svd(A)` to check your answer to d.

Solution:

1.2 From SVD to the matrix polar decomposition

We know that the matrix V is unitary. This means that we can perform the following operations:

$$\begin{aligned} A &= USV^\top \\ &= UISV^\top \\ &= UV^\top VSV^\top \end{aligned}$$

Because both US and V have orthonormal columns, their product also has orthonormal columns. Also, because the matrix S is a matrix containing only real positive values along the diagonal, when we multiply it by V and V^{top}, the resulting matrix will be an $n \times n$ positive semi-definite matrix, which is unique, giving:

$$A = (UV^\top)(VSV^\top) = QP$$

By the reasoning above, the matrix $Q = UV^\top$ is unitary and while $P = VSV^\top$ is positive semidefinite, which fits the form of the polar decomposition of a matrix.

Example 1.2 SVD

Based on the previous example, answer the following:

a) Compute $Q = UV^\top$ and show that it is unitary.

b) Compute $P = VSV^\top$ and test that it is positive semidefinite.

c) Check your answers by comparing the product QP to the original A.

Solution:

1.3 SVD as a Compression Algorithm

Suppose A is the pixel intensity matrix of a large black-and-white image. The entry a_{ij} gives the intensity of the ij^{th} pixel. If A is $n \times n$, then transmitting A requires $O(n^2)$ numbers.

Instead, one could send A_k, that is, the top k singular values $\sigma_1, \sigma_2, \ldots, \sigma_k$, of A, along with the left and right singular vectors $u_1, u_2, \ldots, u_k$, and $v_1, v_2, \ldots, v_k$. This would require sending $O(kn)$ numbers instead of $O(n^2)$. If $k \ll n$, this results in communication savings. For many images, a k much smaller than n can be used to reconstruct the image provided that a very low resolution version of the image is sufficient. Thus, one could use SVD as a compression method.

Open a fairly well-known painting, in .pgm (greyscale) format, in a file called `mona_lisa.pgm` and have a look at the pgm format.

```
from PIL import Image
import numpy as np
 # select the path of image as your directory in Spyder
Imag_in = Image.open('mona_lisa.ascii.pgm')
print("Image_Format:",Imag_in.format)
print("Image_Size:",Imag_in.size)
print("Image_Mode:",Imag_in.mode)

Image_out = np.array(Imag_in)  # read image to array
Image_out_ver2 = np.asarray(Imag_in)  # read image to
   array - another method

Imag_resize = Imag_in.resize((250,360))  # resize
Imag_rotate = Imag_in.rotate(45)  # rotate
 # Cropping a region from an image:
Area_box = (50,150,100,200) # left, upper, right, lower
Cropped = Imag_in.crop(Area_box)

Image_reconstruction = Image.fromarray(Image_out)  #
   Getting back the image from converted Numpy Array
Image_reconstruction.save('my_pic.png')  # Saving
   reconstructed image
```

```
Image_Format: PPM
Image_Size: (250, 360)
Image_Mode: L
```

Example 1.3 SVD Application

Read the `mona_lisa.pgm` image and convert it to an array called A. Then:

a) Use `np.linalg.svd()` to determine the singular values of A, and check the dimensions of S, U and V.

b) Plot the singular values of A using bar plot. What does this tell you about the smallest 250 singular values?

c) Using the first $K = 5$ singular values $\sigma_1, \ldots \sigma_K$, compute the weighted sum of outer products given by

$$U_k \sigma_k V_k$$

for each k, and add them together. Check the dimensions of the output of these operations.
View the compressed image using `plt.imshow()`. That was a bit too much compression, and not enough information to recover the image.

Now try greater values of $K = 10, 20, 30$ and 40.

Solution:

1.4 Linear programming

Linear programming (LP) is a very useful tool for modeling a range of flow and scheduling optimisation problems that arise across all engineering disciplines. These optimsiation problems cannot be solved using calculus, so numerical methods are employed. The next two exercises go through how take a model description and formulate it algebraically, and then in matrix form, before solving it using Python. Before this, however, we formally define a general LP.

1.4.1 General LP formulation

The general form of an LP is written in *algebraic form* as:

$$
\begin{aligned}
\max_x: \quad & f_1 x_1 + f_2 x_2 + \ldots + f_n x_n \\
\text{subject to}: \quad & a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n \leq b_1 \\
& a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n \leq b_2 \\
& \quad \vdots \\
& a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n \leq b_m \\
& c_{11} x_1 + c_{12} x_2 + \ldots + c_{1n} x_n = d_1 \\
& c_{21} x_1 + c_{22} x_2 + \ldots + c_{2n} x_n = d_2 \\
& \quad \vdots \\
& c_{m_{eq}1} x_1 + c_{m_{eq}2} x_2 + \ldots + c_{m_{eq}n} x_n = d_{m_{eq}} \\
& x_1 \geq 0 \\
& x_2 \geq 0 \\
& \quad \vdots \\
& x_n \geq 0
\end{aligned}
$$

Here, first line is the *objective* function, in which we choose $(x_1, x_2, \ldots, x_n)$ to maximise the specified linear combination. The the remaining lines are the model *constraints*, including a set *inequality constraints*, then a set of *equality constraints*, and finally, the *nonnegative variable constraints*.
Note that all of these relationships are linear, hence this model is an LP. Also, because of

this, these expressions may be represented in *matrix form* as:

$$\begin{aligned} \max_{\boldsymbol{x}}: \quad & \boldsymbol{f}^\top \boldsymbol{x} \\ \text{subject to}: \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\ & C\boldsymbol{x} = \boldsymbol{d} \\ & \boldsymbol{x} \geq 0, \end{aligned}$$

or even more compactly, as:

$$\max \left\{ \boldsymbol{f}^\top \boldsymbol{x} \mid A\boldsymbol{x} \leq \boldsymbol{b} \wedge C\boldsymbol{x} = \boldsymbol{d} \wedge \boldsymbol{x} \geq 0 \right\},$$

where:

- $\boldsymbol{x}$ and $\boldsymbol{f}$ are (column) vectors with length n equal to the number of variables,
- $\boldsymbol{b}$ is a vector with length m equal to the number of inequality constraints, and A is a $m \times n$ matrix of inequality constraint coefficients, and
- $\boldsymbol{d}$ is a vector with length m_{eq} equal to the number of equality constraints, and C is a $m_{eq} \times n$ matrix of equality constraint coefficients.

The second, more compact form is written using set notation. In it, the symbol | means "such that", and $\wedge$ means "and", indicating both sets of constraints need to satisfied for the solution to the problem of maximising $\boldsymbol{c}^\top \boldsymbol{x}$ to be *feasible*.

1.4.2 Network flow problem formulation

A particularly interesting problem is the *maximum network flow problem*, defined as follows. Given:

- a weighted directed graph with two special *nodes*, a *source* s, and a *sink* (destination) t,
- edge weights (all positive) which can be interpreted as as capacities,

the goal is to find the *maximum flow* from s to t such that:

- the flow (non-negative) does not exceed capacity along any edge, and
- the flow at every vertex satisfies a conservation or equilibrium condition (flow in equals flow out).

For example, this might be oil flowing through pipes, the transportation of goods for a delivery company, internet routing or DC power flow. For this reason, the mode is often called a *commodity* network flow model, where the commodities can be anything from tomatoes to terawatts.
A network flow problem can be easily formulated as a linear program (LP). The LP that is *derived* from a maximum network flow problem has a large number of constraints, which are modeled as follows:

- Introduce *variables* to represent *flow* over each edge of the network.
- Formulate the edge *capacity* constraints and
- the node *conservation* constraints.
- Add a *feedback* link from the sink to the source, to represent the *total* flow.
- Last, the *objective function* of the *LP* is the *total* flow over the feedback link.

The key to convert a max-flow problem into a LP is the use of flow variables. The flow variable x_{ij} describes the amount of flow from node i to node j via link ij.

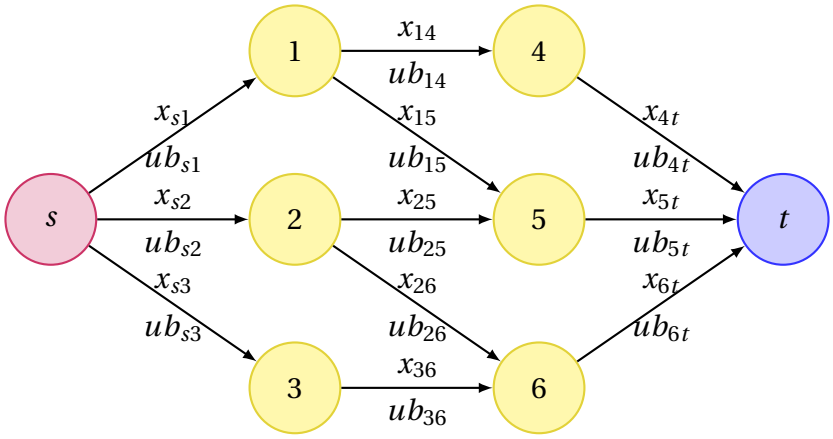

Figure 1: Network flow model

Consider the basic flow network in Figure 1. Each node represents a junction in the commodity flow graph, while the edges represent the possible links between them. The source node is labeled s and the sink is t, and the other nodes are labeled using letters. The *flow variables* on each link, x_{ij}, indicate how much of the commodity moves from one junction to another. A link's capacity is indicated by the problem *parameter* ub_{ij}, which defines and instantiation of this problem.

Example 1.4 Solving Linear Programming

In order for Python to correctly interpret your problem input, you have to choose the order in which the variables in $\boldsymbol{x}$ appear in the matrix C and vector d, as well as the variable lower and upper bounds. The variable vector $\boldsymbol{x}$ is never explicitly entered by you. Rather, it values are returned by the `scipy.optimize.linprog` function call.

Below is a list of the edge capacities for a problem instance:

$$\begin{array}{llll} ub_{s1}=3 & ub_{s2}=2 & ub_{s3}=2 & ub_{14}=5 \\ ub_{15}=1 & ub_{25}=1 & ub_{26}=3 & ub_{36}=1 \\ ub_{4t}=4 & ub_{5t}=2 & ub_{6t}=4 & \end{array}$$

The variables' upper bounds are given by these values (the lower bounds are all zeros). Write out the variable vector $\boldsymbol{x}$ and corresponding upper bound vector, called ub.

Fill out matrix C by writing the equations in matrix form for each node 1-6, s and t. The model is completed by adding an artificial flow from t back to s. For this, we need to introduce an artificial variable, x_{ts}, and add flow conservation constraints on the source and sink nodes. Write these constraints for s and t in algebraic form, using the artificial variable x_{ts}.

The function `scipy.optimize.linprog` has the following syntax:

```
scipy.optimize.linprog(f, A_ub=A, b_ub=b, A_eq=C, b_eq=d, bounds=
(lb,ub))
```

The last remaining element is f, the objective function coefficients. As noted above, the objective is to maximise the flow on the artificial edge between the sink and the source. As such f is zero for all variables except this flow.

However `linprog` is a *minimisation* routine but to convert it to a *maximisation* problem just reverse the sign of the coefficients. Beyond this, the choice of f_{ts} is arbitrary, but for simplicity, define a vector that encodes it as -1.

Solution: