Numerical Solution and Simulation in Engineering with Python

By:

Ayoob Salari

Chapter 6

Table of Contents

SECTION 1

Probability, Statistics, and Data Analysis

This chapter presents some of the Python functions used in probability calculations, basic statistics, and data analysis. The discrete *Markov chain* model is introduced and its stationary points identified using eigenanalysis, linking linear algebra and this class of stochastic model. Based on this, the *PageRank algorithm* is discussed, which underpins Google's websearch engine. Importantly, reading data from and writing data to a file is demonstrated. Curve fitting is introduced, as is the correlation coefficient.
For a more detailed treatment of statistics and and machine learning topics, consult *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, by Hastie, Tibshirani and Friedman (2009, 2nd Ed, Springer Series in Statistics)

1.1 Combinatorics and Counting

Probability calculations often involve counting combinations of objects, and the study of combinations of objects is the realm of *combinatorics*. Many formulas in combinatorics are derived simply by counting the number of objects in two different ways and setting the results equal to each other. Often the resulting proof is obtained by a "proof by words." Formulas are not necessarily derived from manipulations of factorial functions as some students might think. Some of Python's combinatorial functions are illustrated in this section, which are likely familiar to you.

1.1.1 Permutations and combinations

A *permutation* is an ordered arrangement of the objects in set S. If $|S| = n$, then there are:

$$n(n-1)(n-2)\dots(2)(1) = n!$$

different permutations of these n objects. An example is the size of the set of different orders that a group n of people could appear in a queue, or that labeled balls are drawn from a bucket.
For these types of counting problems, Python provides a `math.factorial` function. However, it is generally preferable (because it is quicker and can be used for much larger numbers), to use the `gamma` function to calculate factorials:

$$\Gamma(n) = \int_0^\infty x^{n-1} e^{-x} dx$$

and use the fact that $\Gamma(n+1) = n!$ for positive integer values of n. For example,

```
import math
print ("The␣factorial␣of␣5␣is␣:␣", math.factorial(5))
print ("The␣gamma(6)␣is␣[factorial(5)]␣:␣", math.gamma(6))
```

```
The factorial of 5 is :  120
The gamma(6) is [factorial(5)] :  120.0
```

Exercise 1.1

1. Write a `for` loop to calculate the factorial of n.
2. define a `function` that can calculate factorial using recursion.

Solution:

```
# Part 1
n = 5
fact = 1
for i in range(1,n+1):
    fact = fact * i
print ("The factorial of 5 is : ", fact)
# Part 2
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
print("The factorial of 5 is : ", factorial(5))
```

More generally, a k-permutation arises when we choose k objects in order from a set of n distinct objects. The total number of different permutations of size k of these n objects is:

$$n(n-1)(n-2)\dots(n-k+1) = \frac{n!}{(n-k)!}$$

where dividing by the number of remaining items $(n-k)!$ truncates the factorial computation at the appropriate point in the series.

Alternatively, we may not care about the order in which the objects are selected. There are two cases.

First, if we sample *with replacement*, the number of combinations is simply n^k.

The second is referred to as sampling without replacement (i.e. as if we are picking balls from a bucket and not returning them). In this case, we consider a *combination* of k objects dawn from n, which is written: $\begin{pmatrix} n \\ k \end{pmatrix}$. We can figure out the formula for $\begin{pmatrix} n \\ k \end{pmatrix}$ just by counting.

First, we know there are $n!$ permutations of all of the objects; set this to the LHS of our expression. Next, let us construct the RHS by first, counting the number of k-sized permutations of n objects by dividing the n objects into a group of k selected objects and the remaining $(n-k)$ objects. We know that there is a total of $k!$ permutations of the k selected objects, and likewise, a total of $(n-k)!$ permutations of the $(n-k)$ remaining objects. Therefore, the total number of permutations of the n objects is:

$$n! = \begin{pmatrix} n \\ k \end{pmatrix} k!(n-k)!$$

Rearranging this, we have:

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!}$$

different permutations of size k of these n objects. Note that we have derived this formula simply by counting, not by expanding factorial functions. However, now we know the answer, we can also see that the same expression can be found from the size of the set of k-permutations of n objects, divided by the number of permutations of the drawn objects themselves, because we don't care about their order:

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\dots(n-k+1)}{k(k-1)(k-2)\dots(2)(1)}$$

Python provides the `math.comb(n, k)` function for calculating combinations, for example, for $n = 5, k = 3$:

```
import math
print ("C(7,5)=", math.comb(7, 5))
```

```
C(7,5) = 21
```

Example 1.1 Combination

a) Verify that by using Python's `factorial` and `math.comb` functions for a few example numbers, e.g. (n=12,k=7) and (n=24,k=6).

b) Test the equation below using Python's `math.comb` function and a few example numbers: (n=10, k=4), (n=20, k=8).

$$\binom{n}{k-1} + \binom{n}{k} = \binom{n+1}{k}$$

Solution:

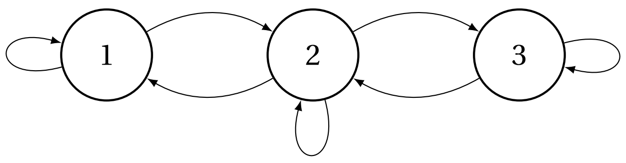
If you want to know how to prove the relationship above, read the following, otherwise just skip it.

Proof: By definition $\binom{n+1}{k}$, is the number of ways to choose k items out of $n+1$ items. We can break the items into one group, Group A, with n items an another group, Group B, with only 1 item, and consider the number of ways to choose items from these two groups. Assume we pick the 1 item from Group B, we then have to choose $k-1$ items from Group A. The number of ways we can do this is clearly $\binom{n}{k-1}$. We could also choose not to pick the 1 item in Group B, so that we have to select all items from Group A. The number of ways we can do this is $\binom{n}{k}$. We must conclude then that the total number of ways to choose k items out of $n+1$ items is the sum of these two cases, given by: $\binom{n}{k-1} + \binom{n}{k} = \binom{n+1}{k}$.

1.2 Stochastic matrices and Markov chains

A Markov chain is stochastic discrete-time, discrete-state transition system. Markov chains are used extensive to model systems with predictable, stochastic, state transistions, including activity models, target tracking, regression models with mode- or regime-switching, for simulating wind and weather patterns, and in finance. They are also used extensively as a computational routine for implementing Baysian models, which require efficient resampling methods, an approach called the *Monte Carlo Makov chain* method.

A three-state Markov chain system is illustrated below:



Here, the three states are illustrated as circles, with chance transitions between them, as indicated by edges. Each edge has associated with it a probability, such that at each time step, the probability of moving from state s to state s' is given by $P(s'|s)$, and edges indicate transitions that have strictly positive probabilities.
Note that, in this example:

- not all states are directly linked with all others, but
- each state can be reached from all others by some path, and
- no two states have a periodic cycle between them.

When the second and third condition above hold, we say that the Markov chain is *ergodic*. We are going to mix an application of probability and eigendecomposition to analyse the steady state behaviour of an ergodic Markov chain.
Let the state-transition probabilities for the diagram above be given by a matrix:

$$P = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0.2 & 0.5 & 0.3 \\ 0 & 0.5 & 0.5 \end{bmatrix}$$

The entries in P are read as the probability that, starting in an initial state corresponding to a given row, the system moves to the state indicated by the column. Importantly, note that all rows sum to 1; this is the definition of a *stochastic matrix*. Indeed, you can think of each row of a stochastic matrix as a valid probability distribution, specifically a *categorical* distribution. (In the chapter, categorical distributions over k possible outcomes were mentioned as the basic probability model underpinning *multinomial distributions*, in the same way that a *Bernoulli distribution* over binary outcomes underpin the *binomial distribution*. What we see here is a much more general model that includes a notion of state.)

Example 1.2 Markov Chain Stochastic Matrix

a) Write a script encoding P as a Python matrix, and use a logical test to check that it indeed is a stochastic matrix.

b) An initial state x_0 can be encoded as vector, with a value of 1 indicating the initial state and zeros everywhere else. Write a vector x encoding an initial state of 1.

c) Compute the distribution of states of the Markov chain 2 time steps after starting in state 2 by multiplying again by P. Repeat the multiplication for 4, 10, 20 and 100 time steps. What is happening to the resulting distribution of states?

Solution:

1.2.1 Stationary distribution of a Markov chain

A stationary distribution of a Markov chain is a probability distribution that remains unchanged in the Markov chain as time progresses. Typically, it is represented as a row vector whose entries are probabilities summing to 1, and given transition matrix P, it satisfies:

$$xP = x$$

In other words, x is *invariant* by the matrix P.
An important result in matrix theory known as the *Perron–Frobenius theorem* applies to stochastic matrices. It concludes that a nonzero solution of the equation above exists and is unique to within a scaling factor. If this scaling factor is chosen so that:

$$\sum_i x_i = 1,$$

then x is a state vector of the Markov chain, and is itself a stationary distribution over states.
In the exercise above, we saw an iterative approach to computing the stationary distribution of a Markov chain. This process is called the *power method*, for a reason that should be obvious.
However, note that the equation for the stationary distribution looks very similar to the column vector equation $Pq = \lambda q$ for eigenvalues and eigenvectors, with $\lambda = 1$. Also you can test that P is positive-definite, so P has an eigendecomposition — this is a general characteristic of stochastic matrices. In fact, for a technical reason, we transpose the matrices to get them into an appropriate form for eigenanalysis. This allows us to find the stationary distribution as a left eigenvector (as opposed to the usual right eigenvectors) of the transition matrix. The operations are as follows:

$$(xP)^T = x^T = P^T x^T$$

In other words, the transposed transition matrix P^T has eigenvectors with eigenvalue 1 that, when normalized to sum to 1, are stationary distributions expressed as column vectors. Therefore, if the eigenvectors of P^T are known, then so are the stationary distributions of the Markov chain with transition matrix P. In an ergodic Markov chain, this stationary distribution is unique.

Example 1.3 Stationary distribution of Markov Chain

a) Write a script to: (i) compute the left eigenvectors of P, (ii) check that there is at least one eigenvalue of 1, and (iii) display the associated stationary distribution of the Markov chain P.

b) Compute the distribution of states of the Markov chain 2 time steps after starting in state 2 by multiplying again by P. Repeat the multiplication for 4, 10, 20 and 100 time steps.

Solution:

When there are multiple eigenvectors associated to an eigenvalue of 1, each such eigenvector gives rise to an associated stationary distribution. However, this can only occur when the Markov chain is reducible (i.e. can be broken into smaller independent chains).

1.2.2 PageRank algorithm (Google)

The *PageRank algorithm* was developed by Google's founders, Larry Page and Sergey Brin. PageRank is determined entirely by the link structure of the World Wide Web. In the past, it was recomputed about once a mont and did not involve the actual content of any Web pages or individual queries. Google has continued to improve on PageRank since, but their underlying methods are very much based on the algorithm that is about to be described. For any particular query, Google finds the pages on the Web that match that query, and lists those pages in the order of their PageRank, and it works as follows.
Imagine going from page to page by randomly choosing an outgoing link from one page to get to the next. In this way, webpages can be thought of as states on a Markov chain, and web links out of a webpage each have equal positive probability. However, the random walk can lead to dead ends at pages with no outgoing links, or cycles around cliques of interconnected pages. If this can happen, the Markov chain is not *ergodic*. So, a certain fraction of the time, the algorithm simply chooses a random page from the web.
Formally, let G be an n-by-n connectivity matrix, with g_{ij} indicating a web link from page i to page j by a one, and zeros everywhere else. We are going to fill in the values of P, the n-by-n ergodic Markov chain transition matrix associated with G for the PageRank algorithm.
Define:

$$r_i = \sum_j g_{ij} \qquad i = 1, \dots, n$$

to be the row sum of G, or page i's *outdegree*. If a state's *outdegree* is 0, then it is a dead-end.
Next, for those states that are not dead-ends, let p be the probability that a particular link is followed. This means that with probability $1-p$ some arbitrary edge is chosen, and denote:

$$\delta = (1-p)/n$$

as the probability that a particular edge is followed. Assuming a uniform distribution over links followed, the probability of traversing a particular edge is given by:

$$p_{\text{ij}} = \frac{p g_{ij}}{r_i} + \delta \quad \text{if} \quad r_i > 0.$$

For a dead end, the algorithm picks a new state at random with uniform probability, so that:

$$p_{\text{ij}} = \frac{1}{n} \quad \text{if} \quad r_i \leq 0.$$

These probabilities together make up all the elements of P, the transition matrix of PageRank.
Given P, the limiting probability x^*, satisfying the stationarity condition $x^*P = x^*$, that an infinitely procrastinating visitor lands on any particular page, is its PageRank. Thus, a page has higher PageRank if other pages with high rank link to it.

Example 1.4 Google Page Rank

Set $p = 0.85$, and let the start and end points of edges be given in the following two vectors:

$$s = \begin{pmatrix} 2 & 6 & 3 & 4 & 4 & 5 & 6 & 1 & 1 \end{pmatrix}$$

and

$$t = \begin{pmatrix} 1 & 1 & 2 & 2 & 3 & 3 & 3 & 4 & 6 \end{pmatrix}$$

a) Use the `scipy.sparse.csr_matrix()` function to build a sparse matrix G with this data. Check the Python documentation on how to use `scipy.sparse.csr_matrix` with edge index vectors. Then convert G to a full matrix using `G.todense()` and inspect it.

b) Calculate P using a `for` loop over the rows of G. Explain why a `for` loop is needed here.

c) Compute the distribution of states of the Markov chain 2 time steps after starting in state 2 by multiplying again by P. Repeat the multiplication for 4, 10, 20 and 100 time steps.

   [Hint, the PageRanks for this small section of the WWW are: $(0.2832, 0.2311, 0.1698, 0.1453, 0.025, 0.1453)$]

Solution:

1.3 Reading and Writing Data

Data analysis relies on being able to read an write data to a file. Three formats for reading and writing data to a file are described in this chapter: (i) ASCII (.CSV), (ii) binary (.NPY), and (iii) compressed (.NPZ file).
First we will define our data as a 10x10 matrix containing the numbers up to 100:

```
import numpy as np
X = np.arange(1,11)
mydata = np.arange(1,11)
for i in range(2,11):
    mydata = np.append(mydata,i*X)

mydata = mydata.reshape(10,10)
```

```
print(mydata)
```

In the following we will show how to save and load this data in different formats.

1.3.1 ASCII data files

CSV, or the comma-separated variable format, is the most used format for storing numerical data in files. NumPy arrays can be saved as CSV files using the `savetxt()` method. This method accepts as parameters a filename and an array, and saves the array in CSV format. You must also indicate the delimiter; this is the character used to separate each variable in the file, most often a comma. This is controlled by the "delimiter" parameter.
The following Python code shows how to save data in ASCII format and load it:

```
from numpy import savetxt
from numpy import loadtxt
 # save to csv file
savetxt('mydata.csv', mydata, delimiter=',')
 # load array
data = loadtxt('mydata.csv', delimiter=',')
print(data)
```

1.3.2 Python binary data files

Occasionally, we have a large amount of data in `NumPy` arrays that we desire to preserve efficiently, but which we will only utilise in another Python application. Therefore, `NumPy` arrays can be saved in a native binary format that is efficient for both saving and loading. This is typical for input data that has been prepared, such as converted data, and will be used to test a variety of machine learning models or conduct several experiments in the future. This use case is acceptable for the `.npy` file format, often known as the "NumPy format." This can be accomplished by using the `save()` NumPy function and passing it the filename and array to save.

```
from numpy import save
from numpy import load
 # save to npy file
save('data.npy', mydata)
 # load array
data = load('data.npy')
print(data)
```

1.3.3 Python Compressed data files

Every once in a while, we prepare enormous data sets for modelling that must be utilised across numerous trials, yet the data sets are massive. This could be a collection of preprocessed NumPy arrays, such as a corpus of text (integers), or rescaled image data (pixels). In these situations, it is useful to both save the data to a file and compress them. This reduces gigabytes of data to hundreds of megabytes and facilitates transfer to other cloud computing servers for long algorithm runs. This situation calls for the `.npz` file format, which provides a compressed version of the native `NumPy` file format. The `savez_compressed()` NumPy function permits the storage of several NumPy arrays in a single compressed `.npz` file.

```
from numpy import savez_compressed
from numpy import load
 # save to npy file
savez_compressed('data.npz', mydata)
 # load dict of arrays
dict_data = load('data.npz')
 # extract the first array
data = dict_data['arr_0']
print(data)
```

1.3.4 Reading data from URL

In this section we will read Longley's Economic Regression dataset from a URL [**?**]. There are seven columns and sixteen rows of data in total, with each row defining a summary of economic statistics for one year (1947–1962). Number of people, year, and number of employed persons are listed in columns 5–7, respectively.
The code below loads the dataset from the URL, chooses "population" as the input variable and "employed" as the output variable, and generates a scatter plot.

```
from pandas import read_csv
from matplotlib import pyplot
 # load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/
   Datasets/master/longley.csv'
dataframe = read_csv(url, header=None)
Longly = dataframe.values
 # choose the input and output variables
x, y = Longly[:, 4], Longly[:, -1]
 # plot "Population" vs "Employed"
pyplot.scatter(x, y)
pyplot.show()
```

1.4 Fitting a Curve to Data

Frequently it is important to estimate the functional form of data by fitting a curve through the data points. This is known as *regression analysis.*

The curve fitted to the data is derived using a minimum least-squares error approximation. For example, suppose there are only three data points: $(x_1, y_1), (x_2, y_2), (x_3, y_3)$. Let:

$$e_i = y_i - mx_i - b$$

be called the regressions's *residuals*, that is, the parts of the data's variation that is not explained by the model. The least-squares sum of squared errors (SSE) in the linear fit, $y = mx + b$, is given by:

$$SSE = (y_1 - mx_1 - b)^2 + (y_2 - mx_2 - b)^2 + (y_3 - mx_3 - b)^2.$$

The squared terms are the square residuals, e_i^2, and the fitting problem is to minimise the sum of these values (this is what *least squares* refers to). That is, the fitted curve minimizes the least-square error.

Example 1.5 Linear Fitting

For a linear fit ($y = mx + b$), after the mathematics of minimizing error, we have

$$m = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$b = \bar{y} - m\bar{x}$$

where $\bar{x}$ and $\bar{y}$ are the mean of input variable x and output variable y, respectively. Moreover, n is the number of points.
Write a function that can find the best linear fit to population vs employment of the Longly dataset.

Solution:

```python
import numpy as np
def ols(x,y):
    y_ = y.mean()
    x_ = x.mean()
    m = np.sum((y-y_)*(x-x_))/np.sum((x-x_)**2)
    b = y_- m*x_
    return(m,b)
m, b = ols(x,y)
```

Suppose we would like to know whether the employment rate is following a linear, quadratic (power of 2), or exponential growth law. We can apply the following information:

1. Linear functional dependence ($y = mx + b$) is shown by obtaining a straight line when plotting the linear data, y versus x.

2. Power law functional dependence $y = bx^m$ is shown by obtaining a straight line when plotting $log(y)$ versus $\log(x)$.

3. Exponential functional dependence ($y = be^{mx}$) is shown by obtaining a straight line when plotting $\log(y)$ versus x.

1.4.1 Goodness-of-fit

The quality of a curve fit is characterized by its coefficient of determination, aka r-squared or r^2 value. To understand this, it's useful to consider the following expression, which relates the total variation of y around its mean $\bar{y}$ to the variation explained by the model and the variation in the residuals:

$$\sum_{i=1}^{N}(\bar{y}-y_i)^2=\sum_{i=1}^{N}(f(x_i)-\bar{y})^2+\sum_{i=1}^{N}(y_i-f(x_i))^2$$

where $\bar{y}$ is the mean of the y-values and there are N data points. The term on the left is called the *total sum of squared* (SST), interpreted as the total squared variation. The middle term is the *sum of squares of the regression* (SSR), and the final term is the *sum of squares of the errors* (SSE), or sometimes the *residual sum of squares.* Given these terms, we can rewrite the expression above simply as:

$$SST = SSR + SSE$$

from which we see the total sum of squares (SST) is divided among the variation explained by the model (SSR) and that which is left to the residuals (SSE).
With this intuition in mind, the r-squared value is defined equivalently as follows:

$$\begin{aligned} r^2 &= 1-\frac{\sum_{i=1}^{N}(f(x_i)-y_i)^2}{\sum_{i=1}^{N}(y_i-\bar{y})^2} &&= 1-\frac{SSE}{SST} \\ &= \frac{\sum_{i=1}^{N}(f(x_i)-\bar{y})^2}{\sum_{i=1}^{N}(y_i-\bar{y})^2} &&= \frac{SSR}{SST}. \end{aligned}$$

Example 1.6 Goodness-of-fit

Write a function that calculates the r-squared value of the best linear fit to population vs employment of the Longly dataset.

Solution:

```
import numpy as np
def r2(f,y):
    sst = np.sum((y-y.mean())**2)
    sse = np.sum((f-y)**2)
    ssr = np.sum((f-y.mean())**2)
    r2 = 1-(sse/sst)
    return(r2)
f=m*x+b
r2(f,y)
```

1.5 Correlation Analysis

There might be complex, hidden connections between the dataset's variables. Identifying and quantifying the interdependencies between dataset variables is crucial. The existence of these dependencies will affect the performance of machine learning algorithms like linear regression, therefore understanding them will help us better prepare the data to fulfil the expectations.

A linear connection may exist between variables. This is a constantly additive connection between the two data samples. This link between two variables is referred to as the covariance. It is determined by taking the mean of the product of the centred values from each sample.

$$cov(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(X-\bar{X})(Y-\bar{Y})$$

It is easy to see that variance is a special case of covariance.

$$cov(X,X) = var(X) \equiv \sigma_X^2$$

The standard deviation of X is the square root of the variance of X.

Example 1.7 Checking Dataset

To test Longly dataset, check the mean, standard deviation, and covariance of population and employment.

Solution:

```
import numpy as np
print("x: mean=%.3f stdv=%.3f"%(np.mean(x), np.std(x)))
print("y: mean=%.3f stdv=%.3f"%(np.mean(y), np.std(y)))
print('Cov(x,y)= ', np.cov(x, y, bias=True))
```

In the above example, it should be noted that we have calculated the covariance matrix that is

$$\begin{bmatrix} cov(x,x) & cov(x,y) \\ cov(y,x) & cov(y,y) \end{bmatrix}$$

It should be noted that `np.cov(x,y)` by default calculates covariance by dividing the $\sum_{i=1}^{n}(X-\bar{X})(Y-\bar{Y})$ to $(n-1)$. If we want to divide it by n we either have to add `bias=True` or write the formula instead of using library.

```
# Writing formula for Covariance
print("Covariance=", np.sum( (x - np.mean(x)) * (y - np.
   mean(y)) )/(len(x)))
```

```
Covariance= 21.995
```

Pearson's Correlation

The strength of a linear relationship between two data samples may be summarised by using the Pearson correlation coefficient.

The Pearson's correlation coefficient is determined by dividing the covariance between the two variables by the product of their respective standard deviations. It is a score derived by normalising the covariance between the two variables.

$$\text{Pearson's Correlation Coefficient} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

```
 # Method 1
import numpy as np
print("Pearson's=", np.cov(x,y,bias=True)[0,1]/(np.std(x)
   * np.std(y)))
 # Method 2 - Using Library
from scipy.stats import pearsonr
corr,_ = pearsonr(x, y)
print("Pearson's_via_library=", corr)
```

```
Pearson's= 0.9603
Pearson's_via_library= 0.9603
```