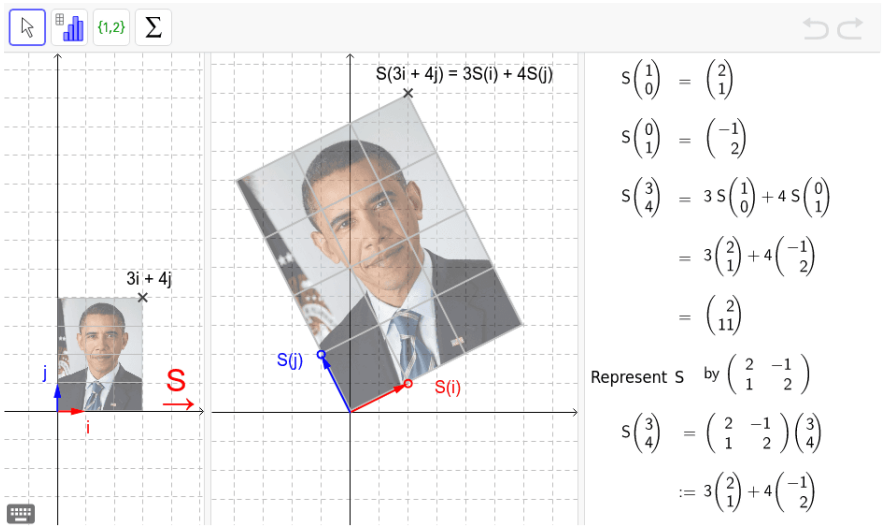
Linear Algebra
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/mpp_2023/linear_algebra/linear_algebra.pdf

---



Linear transformations are a basic model of change.

Linear Algebra

- *Mathematical models often are formed using linear algebraic equations;*
- *The* `numpy` *function* `np.linalg.solve()` *solves a linear system.*
- *The* `numpy` *function* `np.linalg.lstsq()` *solves least squares problems.*
- *The* `numpy` *function* `np.linalg.eig()` *solves eigenvalue problems.*
- *The* `numpy` *function* `np.linalg.svd()` *computes the singular value decomposition.*

1 A boundary value problem

To appreciate linear algebra, let's start with a problem involving a differential equation, reformulate it as a discrete numerical problem, and discover that we have created a system of coupled linear equations, whose solution will give us our desired answer. Along the way, we learn, or be reminded of, how Python creates and manipulates the vectors and matrices that are the subject of linear algebra.

Consider the following boundary value problem (often abbreviated to "BVP), in which a scalar function $u(x)$ is defined over the interval $[a, b]$, and satisfies the following second order differential equation:

$$-\frac{d^2u}{dx^2} = f(x) \quad \text{for } a < x < b$$

along with the boundary conditions:

$$u(a) = g_{left}$$
$$u(b) = g_{right}$$

Our task is to discover a formula for $u(x)$ from this information. For certain textbook problems, it is possible to determine the solution, but in general, an analytic formula for the solution $u(x)$ cannot be determined.

However, for every problem of this sort, we can apply a numerical method that may be able to estimate the shape of the solution. Here, we will do so uing a simple version of the *finite difference method*, which replaces analytic derivative expressions by differences of the function value at discrete points.

Given an interval $[a, b]$, we specify an equally spaced grid of n points, or *nodes*, with spacing $h = \frac{b-a}{n-1}$, so that a typical discrete point is:

$$x_i = \frac{(n-i) \cdot a + i \cdot b}{n-1}$$

Instead of seeking a formula for a function $u(x)$, we now are only interested in the value u_i at each node x_i. Since our original equation involves the second derivative of $u(x)$, we need to translated this into an expression in terms of the discrete values. To do so, we rely on the following centered difference approximation to the second derivative at the point x_i, which is valid for data at equal spacings h:

$$\frac{d^2u(x)}{dx^2}(x_i) = \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + O(h^2) \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}$$

Now it turns out that, at every node x_i, we can write a boundary condition or a discrete version of the original differential equation, involving only the discrete values u_i:

$$\begin{aligned}
\text{Node 0: } & u_0 = g_{left} \\
\text{Node 1: } & \frac{-u_0 + 2u_1 - u_2}{h^2} = f(x_1) \\
\text{Node 2: } & \frac{-u_1 + 2u_2 - u_3}{h^2} = f(x_2) \\
& \ldots \\
\text{Node n-2: } & \frac{-u_{n-3} + 2u_{n-2} - u_{n-1}}{h^2} = f(x_{n-2}) \\
\text{Node n-1: } & u_{n-1} = g_{right}
\end{aligned}$$

It is easy to see that we have created a system of n linear equations in the variables u_i, which we can rephrase in matrix-vector notation as:

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & \ldots & 0 \\
\frac{-u_0}{h^2} & \frac{2u_1}{h^2} & \frac{-u_2}{h^2} & 0 & 0 & \ldots & 0 \\
0 & \frac{-u_1}{h^2} & \frac{2u_2}{h^2} & \frac{-u_3}{h^2} & 0 & \ldots & 0 \\
0 & 0 & \frac{-u_2}{h^2} & \frac{2u_3}{h^2} & \frac{-u_4}{h^2} & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & 0 & 0 & \ldots & \frac{-u_{n-1}}{h^2} \\
0 & 0 & 0 & 0 & 0 & \ldots & 1
\end{pmatrix} \cdot \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ \ldots \\ u_{n-2} \\ u_{n-1} \end{pmatrix} = \begin{pmatrix} g_{left} \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ \ldots \\ f(x_{n-2}) \\ g_{right} \end{pmatrix}$$

Now, we think of this linear system in the form $A \cdot x = b$, and we need to find some linear algebra software to get our answer. Since we are using Python, we use the function `np.linalg.solve()`:

```
x = np.linalg.solve ( A, b )
```

Listing 1: Solving a linear system with `np.linalg.solve()`.

Now that we have an idea of how a BVP is turned into a linear system and solved, we will look at a specific example. Our interval will be $[0, 1]$, and the differential equation is:

$$-\frac{d^2u}{dx^2} = x \cdot (x+3) \cdot e^x$$

along with the boundary conditions:

$$u(0) = 0$$
$$u(1) = 0$$

Here is a code to discretize the geometry, set up the linear system, and solve it:

```
def bvp ( n ):
  import numpy as np
  x = np.linspace ( 0.0, 1.0, n )
  h = 1.0 / ( n - 1 )

  A = np.zeros ( [ n, n ] )
  b = np.zeros ( n )

  for i in range ( 0, n ):
    if ( i == 0 )
      A[i,i] = 1.0
      b[i] = 0.0
    elif ( i < n - 1 ):
      A[i,i-1] = - 1.0 / h**2
      A[i,i]   =   2.0 / h**2
      A[i,i+1] = - 1.0 / h**2
      b[i] = x[i] * ( x[i] + 3.0 ) * np.exp ( x[i] )
    else
      A[i,i] = 1.0
      b[i] = 0.0

  u = np.linalg.solve ( A, b )

  return u
```

Listing 2: Exact solution for bvp.

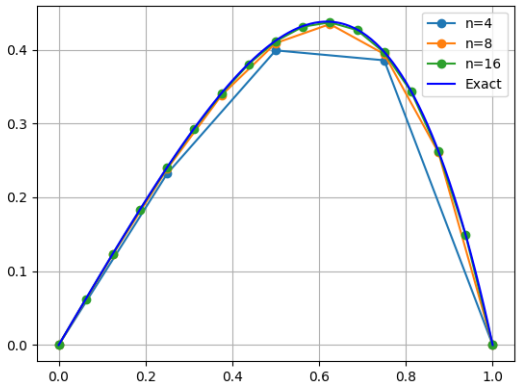
There is an exact solution for this problem:

$$u(x) = -x \cdot (x-1) \cdot e^x$$

which we will program as the function `bvp_exact()`, and now we can solve our problem for a given value of n and compare our discrete and exact solutions:

```
for n in [ 5, 9, 17 ]:
  x1 = np.linspace ( 0.0, 1.0, n )
  u1 = bvp ( n )
  plt.plot ( x1, u1, 'o-' )

x2 = np.linspace ( 0.0, 1.0, 101 )
u2 = bvp_exact ( x2 )
plt.plot ( x2, u2, 'b-' )
plt.show ( )
```

Listing 3: Exact solution for bvp.



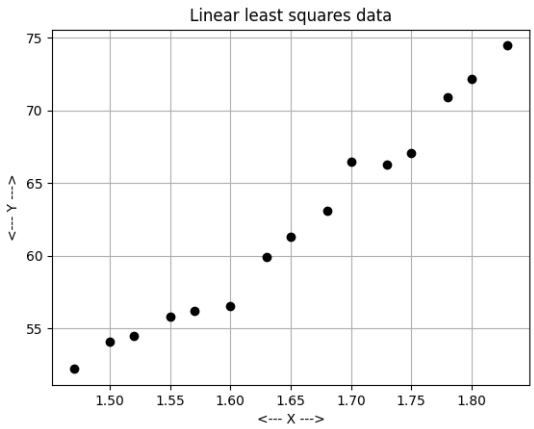
Three approximations and the exact solution to the BVP.

At this point, it is worth looking back at our work and seeing the ways in which Python enabled us to carry out our linear algebra computations.

First, note that the `numpy` library allows us to create vectors and matrices, and to operate on them efficiently. We used `np.linspace()` to create the BVP grid, and to help us in setting up the plot. Because `x` is a `numpy` array, the `bvp_exact()` function was able to evaluate the exact solution at all the points using a single statement, rather than a loop. The `np.zeros()` function allowed us to set space for the matrix `A` and vector `b` that we needed in order to define our linear system. And the function `np.linalg.solve()`, of course, returned our solution `u` as another `numpy` array.

2 Approximating data

Suppose that we have collected n samples of data (x_i, y_i). After examining a plot of this data, we suspect that it can be approximated by some linear function $y = m \cdot x + b$:



A set of 15 data pairs that suggest a linear relationship.

Any two pairs of data could be used to solve for a value for m and b. The resulting formula will go exactly through those two pairs. But we'd much rather use all the data to estimate our line, and we'd like the approximation error to be more evenly spread out.

To do so, we can suppose that if we pick any values for m and b, we can write the corresponding errors as a vector whose components are $e_i = m \cdot x_i + b - y_i$. Then we can try to minimize the function $E(b, m)$, the sum of the squares of these errors:

$$E(b,m) = \frac{1|}{2} \sum_{i=0}^{i<n} (m \cdot x_i + b - y_i)^2$$

The minimum can be found by solving $\frac{\partial E}{\partial b} = 0$ and $\frac{\partial E}{\partial m} = 0$.

$$\frac{\partial E}{\partial b} = \sum_{i=0}^{i<n} (m \cdot x_i + b - y_i) = 0$$
$$\frac{\partial E}{\partial m} = \sum_{i=0}^{i<n} (m \cdot x_i + b - y_i) \cdot x_i = 0$$

This results in the following linear system:

$$\begin{pmatrix} n & \sum_{i=0}^{i<n} x_i \\ \sum_{i=0}^{i<n} x_i & \sum_{i=0}^{i<n} x_i^2 \end{pmatrix} \cdot \begin{pmatrix} b \\ m \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{i<n} y_i \\ \sum_{i=0}^{i<n} y_i \cdot x_i \end{pmatrix}$$

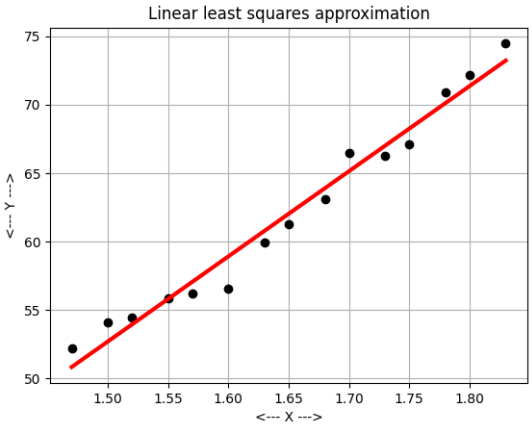
And once again, we have a square (2×2) linear system of the form $A \cdot x = rhs$ which we can solve using `np.linalg.solve()`.

Here is how we would program this computation for our set of data.

```
import numpy as np

ndata = 15

xdata = np.array ( [ \
  1.47, 1.50, 1.52, 1.55, 1.57, \
  1.60, 1.63, 1.65, 1.68, 1.70, \
  1.73, 1.75, 1.78, 1.80, 1.83 ] )

ydata = np.array ( [ \
  52.21, 54.12, 54.48, 55.84, 56.20, \
  56.57, 59.93, 61.29, 63.11, 66.47, \
  66.28, 67.10, 70.92, 72.19, 74.46 ] )

n = 2
A = np.zeros ( [ n, n ] )
rhs = np.zeros ( n )

A[0,0] = ndata
A[0,1] = np.sum ( xdata )
rhs[0] = np.sum ( ydata )

A[1,0] = np.sum ( xdata )
A[1,1] = np.sum ( xdata**2 )
rhs[1] = np.sum ( ydata * xdata )

x = np.linalg.solve ( A, rhs )
```

Listing 4: Solving a linear least squares problem.

which returns the results `b = x[0] = -40.50758429`, `m = x[1] = 62.1479711`. Now, we can replot our original data, and the line $y = m \cdot x + b$, to verify our results:

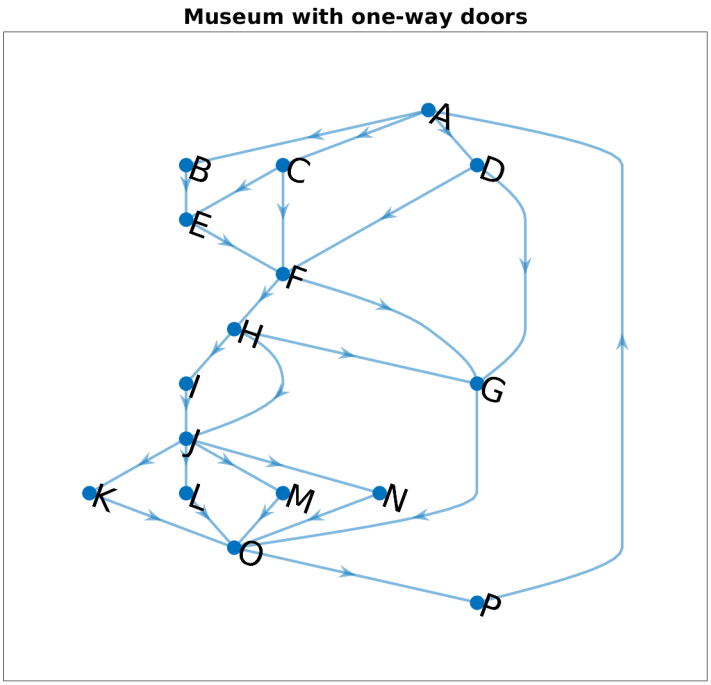


The least squares linear approximation to 15 data pairs.

Again, we have used some features of Python and `numpy` to carry out our linear algebra tasks. We created and valued `xdata` and `ydata` using the `np.array()` function. Because these were row vectors, we simply created a list of values, enclosed in square brackets and separated by commas. Because we had to compute the entries of matrix A and vector rhs, we first used the `np.zeros()` function to allocate space. We then used square brackets to designate the indices of the array or vector into which we wanted to store each computed value. Finally, the `np.sum()` function enabled us to compute the sum of various vectors easily, without having to create a loop that would sum the entries one by one.

3 The Midnight Museum

You stayed too late at the museum and were accidentally locked in. The museum comprises 16 rooms, labeled **A** through **P**. Normally, visitors enter room **A** and wander around, eventually reaching room **P** after which they can only proceed to room **A** where they can exit. But tonight you have no exit, and are doomed to wander through the museum. Fortunately, you've had a lot of coffee, and just can't stop walking from room to room. If you have several choices of the next room to visit, you pick one at random. By the morning, you have entered 1600 rooms.



A museum with 16 rooms, and one-way doors. Your tour starts in room A.

Here are some peculiar questions:

- Can we create a model of the layout of the museum?
- Can we model the process by which you are wandering through the museum?
- Was every room visited about the same number of times?
- Given that you started in room `A`, where are you now, most likely?

To turn our museum map into something that the computer can deal with, we start by creating the *incidence matrix* for the map. Every room of the museum appears as a row and as a column of the matrix A. If there is a door that leads from room i to room j, then $A_{i,j} = 1$. Thus, from room H (#8) there are passages to

rooms G (#7), I (#9), and J (#10) and so in row #8, we see 1's in just those columns.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
A	.	1	1	1	.	.	.	.	.	.	.	.	.	.	.	.
B	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.	.
C	.	.	.	.	1	1	.	.	.	.	.	.	.	.	.	.
D	.	.	.	.	.	1	1	.	.	.	.	.	.	.	.	.
E	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.
F	.	.	.	.	.	.	1	1	.	.	.	.	.	.	.	.
G	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.
H	.	.	.	.	.	.	1	.	1	1	.	.	.	.	.	.
I	.	.	.	.	.	.	.	.	.	1	.	.	.	.	.	.
J	.	.	.	.	.	.	1	.	.	.	1	1	1	1	.	.
K	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.
L	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.
M	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.
N	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1	.
O	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
P	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Another way to describe the incidence matrix is that it lists the number of ways that you can get from room i to room j in a single step. (This is either 0 or 1). The reason for rephrasing the description in this way is the surprising fact that if we compute the matrix $A^2 = A \cdot A$, then entries report the number of ways to get from i to j using exactly 2 steps; looking at higher powers of A gives us reports for journeys of 3, 4, 5 steps and so on.

This is such an amazing "free gift" from linear algebra that it's worth glancing at A^2 and checking the museum map to see that the report is correct. Note in particular that there are now 5 ways to get from J to O in 2 steps.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
A	.	.	.	.	2	2	1	.	.	.	.	.	.	.	.	.
B	.	.	.	.	.	1	.	.	.	.	.	.	.	.	.	.
C	.	.	.	.	.	1	1	1	.	.	.	.	.	.	.	.
D	.	.	.	.	.	.	1	1	.	.	.	.	.	.	1	.
E	.	.	.	.	.	.	1	1	.	.	.	.	.	.	.	.
F	.	.	.	.	.	.	1	.	1	1	.	.	.	.	1	.
G	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
H	.	.	.	.	.	.	1	.	.	1	1	1	1	1	.	.
I	.	.	.	.	.	.	1	.	.	.	1	1	1	1	.	.
J	.	.	.	.	.	.	.	.	.	.	.	.	.	.	5	.
K	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
L	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
M	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
N	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
O	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	1
P	.	1	1	1	.	.	.	.	.	.	.	.	.	.	.	.

In order to answer the questions that we posed about the midnight museum wanderer, we have to modify our incidence matrix to take into account the probabilistic choice of the next room to visit. To do so, we will create the *transition matrix*, which we will represent by T, which will help us analyze the likely paths. We

create T by transposing the incidence matrix, and normalizing each row so that its entries sum to 1. Now ...

$$\begin{array}{c|cccccccccccccccc}
 & A & B & C & D & E & F & G & H & I & J & K & L & M & N & O & P \\
A & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 \\
B & \frac{1}{3} & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
C & \frac{1}{3} & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
D & \frac{1}{3} & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
E & . & 1 & \frac{1}{2} & . & . & . & . & . & . & . & . & . & . & . & . & . \\
F & . & . & \frac{1}{2} & \frac{1}{2} & 1 & . & . & . & . & . & . & . & . & . & . & . \\
G & . & . & . & \frac{1}{2} & . & \frac{1}{2} & . & \frac{1}{3} & . & \frac{1}{5} & . & . & . & . & . & . \\
H & . & . & . & . & . & \frac{1}{2} & . & . & . & . & . & . & . & . & . & . \\
I & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . \\
J & . & . & . & . & . & . & . & \frac{1}{3} & 1 & . & . & . & . & . & . & . \\
K & . & . & . & . & . & . & . & . & . & \frac{1}{5} & . & . & . & . & . & . \\
L & . & . & . & . & . & . & . & . & . & \frac{1}{5} & . & . & . & . & . & . \\
M & . & . & . & . & . & . & . & . & . & \frac{1}{5} & . & . & . & . & . & . \\
N & . & . & . & . & . & . & . & . & . & \frac{1}{5} & . & . & . & . & . & . \\
O & . & . & . & . & . & . & 1 & . & . & . & 1 & 1 & 1 & 1 & . & . \\
P & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & .
\end{array}$$

Now the amazing thing about this matrix is that if we start with a vector x which is zero except for a 1 in the first entry, then the i-th entry of vector $T \cdot x$ gives us the probability that, after 1 step, we are in room i. This vector is, of course $[0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$ Then $T^2 \cdot x$ gives us the probabilities after two random steps, and so on. Here is how these probabilities develop over the first ten random steps:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
0:	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1:	0.00	0.33	0.33	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2:	0.00	0.00	0.00	0.00	0.50	0.33	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3:	0.00	0.00	0.00	0.00	0.00	0.50	0.17	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.17	0.00
4:	0.00	0.00	0.00	0.00	0.00	0.00	0.31	0.25	0.06	0.06	0.00	0.00	0.00	0.00	0.17	0.17
5:	0.17	0.00	0.00	0.00	0.00	0.00	0.09	0.00	0.08	0.14	0.01	0.01	0.01	0.01	0.31	0.17
6:	0.17	0.06	0.06	0.06	0.00	0.00	0.03	0.00	0.00	0.08	0.03	0.03	0.03	0.03	0.14	0.31
7:	0.31	0.06	0.06	0.06	0.08	0.06	0.04	0.00	0.00	0.00	0.02	0.02	0.02	0.02	0.14	0.14
8:	0.14	0.10	0.10	0.10	0.08	0.14	0.06	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.11	0.14
9:	0.14	0.05	0.05	0.05	0.15	0.19	0.13	0.07	0.01	0.01	0.00	0.00	0.00	0.00	0.06	0.11
10:	0.11	0.05	0.05	0.05	0.07	0.20	0.14	0.09	0.02	0.03	0.00	0.00	0.00	0.00	0.13	0.06

This means that, if you told me you had just taken 10 random steps starting at room A, I have my best chance of finding you in room F, although the probability is only 20% of actually finding you precisely there.

It might seem like, as you continue to wander randomly, you will be equally likely to be in any room. But this is not the case. In fact, if we take 100 random steps, and then 101, and 102, here is what the probability table looks like now:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
100:	0.139	0.046	0.046	0.046	0.069	0.116	0.108	0.058	0.019	0.038	0.007	0.007	0.007	0.007	0.139	0.139
101:	0.139	0.046	0.046	0.046	0.069	0.116	0.108	0.058	0.019	0.038	0.007	0.007	0.007	0.007	0.139	0.139
102:	0.139	0.046	0.046	0.046	0.069	0.116	0.108	0.058	0.019	0.038	0.007	0.007	0.007	0.007	0.139	0.139

Mathematically, the system has reached a sort of steady state. Some rooms (A and P) are as much as 20 times more "popular' than others (K, L, M, and N).

It turns out that by the time we have taken 100 steps, the vector x has the interesting property that taking one nore step makes no difference, in other words, $T \cdot x = x$. This is just another way of saying that x is an eigenvector of matrix T with eigenvalue 1.

Was this just a trick, a peculiar feature of the museum map, or does a random walker always end up eventuially defining a fixed distribution of visiting times? There is actually more mathematics behind this story than we want to discuss right now. However, the exercises suggest a few experiments for further exploration. If you are intrigued, you might look up the Perron-Frobenius theorem, and then try to decide why the museum problem satisfies the necessary conditions.

4 Spinning

When an object undergoes spinning, its behavior depends strongly on its shape, and the direction of the axis about which it spins. The equations of motion involve what is known as the *moment of inertia* tensor, which has the form:

$$M = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}$$

where I_{ij} records the moment of inertia about the i axis when the object is rotated about the j axis. Although physicists call M a tensor, for good reasons, we can be excused for thinking of it as just another matrix.

If we suppose that the rigid object is actually composed of n discrete point masses, each with mass m_k and coordinates (x_k, y_k, z_k), then the formulas for the entries of the first row of the inertia matrix are:

$$I_{xx} = \sum_{k=0}^{k<n} m_k \, (y_k^2 + z_k^2) \quad \text{Diagonal entry}$$

$$I_{xy} = -\sum_{k=0}^{k<n} m_k \, x_k \, y_k \quad \text{Off-diagonal entry}$$

$$I_{xz} = -\sum_{k=0}^{k<n} m_k \, x_k \, z_k \quad \text{Off-diagonal entry}$$

with similar formulas for the y and z rows.

This representation of the inertia matrix assumes we are using the standard Cartestian coordinate system. It turns out that every rigid body has a preferred coordinate system in which the inertial matrix becomes diagonal. This coordinate system can be written as an orthogonal matrix Q, whose columns are the axes of the coordinate system. Because of the structure of Q, these axis vectors each have Euclidean length 1, and are pairwise perpendicular. Physically, the Q axis vectors align themselves to three natural rotation directions for the object. Mathematically, multiplying by Q can be regarded as simply twisting the original Cartesian axis system to a more favorable arrangement.

This process can be described as seeking Q such that $Q^T * M * Q$ becomes a diagonal matrix. As long as M is a symmetric matrix (and you can see that this is true!), we can solve the eigenvalue problem for M, with Q an orthogonal matrix of eigenvectors and Λ a diagonal matrix containing the real eigenvalues of M. That is:

$$M \cdot Q = Q \cdot \Lambda$$

which we can rewrite as

$$Q^T \cdot M \cdot Q = \Lambda$$

This tells us that in the Q coordinate system, the equations of motion are simplified, because rotations along each of the coordinate axes have become decoupled. The columns of Q are called the *principal axes of rotation*, and the diagonal entries of Λ the *principal moments of inertia*.

The behavior of a rotating body is strongly affected by the principal moments of inertia. If we sort these values from smallest to largest, and label them I_a, I_b, I_c, we have:

- $I_a = I_b = I_c$: the spherical top, spins like a ball;
- $I_a = I_b < I_c$: the oblate top, spins like a pancake;
- $I_a < I_b = I_c$: the prolate top, spins like a football;
- $I_a < I_b < I_c$: the asymmetric top, no nice spinning pattern;

An individual molecule can be regarded as a discrete construction of k point masses (atoms), held together rigidly by chemical bonds. As mentioned above, this means that the moment of inertia matrix M can be constructed using a finite sum over the k atoms. Once we have M, an eigenvalue analysis will give us the principal moments of intertia, I_a, I_b, I_c, which tells us how the molecule can rotate in response to an applied force.. Spectral analysis of a molecule bombards it with energy, to which it responds in part by rotation. The modes of rotation depend on the principal moments of inertia.

As an example, the following data defines the five atoms of CH_4, where the (x,y,z) coordinates are in atomic units:

```
m        x        y        z
12       0        0        0
1.0079   0.6276   0.6276   0.6276
1.0079   0.6276  -0.6276  -0.6276
1.0079  -0.6276   0.6276  -0.6276
1.0079  -0.6276  -0.6276   0.6276
```

From this data, we can construct M, and then the principal moments of inertia:

```
Ia=5.307906, Ib=5.307906, Ic=5.307906
```

and hence CH_4 rotates like a spherical top. This tells the spectral analyst what modes of energy absorption to study during the experiment.

The exercises ask you to try to carry out a similar analysis for a few other simple molecules.

This example is taken from Christian Hill, *Learning Scientific Programming with Python*.

5 Matrix compression

Consider the following 11×6 array of data A. It comes from a physical example, so the values are not "random". There is, presumably, some pattern in them.

```
A =
  10.0000    8.0400    9.1400    7.4600    8.0000    6.5800
   8.0000    6.9500    8.1400    6.7700    8.0000    5.7600
  13.0000    7.5800    8.7400   12.7400    8.0000    7.7100
   9.0000    8.8100    8.7700    7.1100    8.0000    8.8400
  11.0000    8.3300    9.2600    7.8100    8.0000    8.4700
  14.0000    9.9600    8.1000    8.8400    8.0000    7.0400
   6.0000    7.2400    6.1300    6.0800    8.0000    5.2500
   4.0000    4.2600    3.1000    5.3900   19.0000   12.5000
```

```
  12.0000   10.8400    9.1300    8.1500    8.0000    5.5600
   7.0000    4.8200    7.2600    6.4200    8.0000    7.9100
   5.0000    5.6800    4.7400    5.7300    8.0000    6.8900
```

Even for a rectangular matrix like this, we can compute a norm, namely, the Frobenius or "square root of the sum of the squares of all the entries" norm.

```
value = np.linalg.norm ( A, 'fro' )
```

so that we get a size estimate for A of 68.13.

Now let's compute the singular value decomposition of this matrix, which has the form

$$A = U \cdot S \cdot V$$

where U is an $m \times m$ orthogonal matrix, S is an $m \times n$ diagonal matrix, and V is an $n \times n$ orthogonal matrix.

```
U, S, V = np.linalg.svd ( A )
```

[number=none] U and V tell us something about the input and output coordinate systems preferred by the matrix A, but S tells us how important each component is. So for now, we will ignore U and V, and concentrate on the diagonal values in S, usually called the *singular values*.

$$S = \begin{pmatrix} 65.9197 & 0 & 0 & 0 & 0 & 0 \\ 0 & 15.6834 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4.8767 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3.9058 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2.8167 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2.0286 \end{pmatrix}$$

For our matrix A, the first two diagonal entries are much larger than the later ones. This leads us to suspect that a very large proportion of the information in A can be found in the first two components of the singular value decomposition. That is, it might be the case that we can approximate A as follows, (using 1-based indexing for the moment!):

$$A \approx A_2 = U(1:m, 1:2) \cdot S(1:2, 1:2) \cdot V(1:2, 1:n)$$

Let's actually start with a Python calculation that just uses the first, largest component:

```
A1 = U[0:m,0:1] * S[0:1,0:1] * V[0:1,0:n]
```

which produces the following approximation to A:

```
A1 =
  10.2141    8.1324    8.1899    8.0390    7.7625    6.8756
   8.6104    6.9678    6.9782    6.9562    7.5758    6.4772
  12.5272    9.8381    9.9548    9.6424    8.2723    7.6088
   9.8335    7.9801    7.9842    7.9802    8.8580    7.5333
  10.8390    8.6607    8.7113    8.5800    8.5208    7.4833
  12.4329    9.7068    9.8420    9.4783    7.6832    7.2039
   7.0947    5.8846    5.8444    5.9605    7.5581    6.2056
   3.3005    4.3875    3.8085    5.4085   18.6737   12.8913
  11.9884    9.3333    9.4727    9.0971    7.1656    6.7859
   7.2596    6.1177    6.0439    6.2530    8.6192    6.9342
   5.7494    5.0153    4.8990    5.2242    8.3904    6.5241
```

We can measure the Frobenius norm of the difference:

```
np.linalg.norm ( A–A1,' fro' ) = 17.23
```

Now let's go for the next possible approximation, using the first two components:

```
  A2 = U[0:m,0:2] * S[0:2,0:2] * V[0:2,0:n]
```

```
A2 =
 10.2141    8.1324    8.1899    8.0390    7.7625    6.8756
  8.6104    6.9678    6.9782    6.9562    7.5758    6.4772
 12.5272    9.8381    9.9548    9.6424    8.2723    7.6088
  9.8335    7.9801    7.9842    7.9802    8.8580    7.5333
 10.8390    8.6607    8.7113    8.5800    8.5208    7.4833
 12.4329    9.7068    9.8420    9.4783    7.6832    7.2039
  7.0947    5.8846    5.8444    5.9605    7.5581    6.2056
  3.3005    4.3875    3.8085    5.4085   18.6737   12.8913
 11.9884    9.3333    9.4727    9.0971    7.1656    6.7859
  7.2596    6.1177    6.0439    6.2530    8.6192    6.9342
  5.7494    5.0153    4.8990    5.2242    8.3904    6.5241
```

We can measure the Frobenius norm of the difference:

```
np.linalg.norm ( A–A2,' fro' ) = 7.14
```

A third attempt at approximation is:

```
  A3 = U[0:m,0:3] * S[0:3,0:3] * V[0:3,0:n]
```

which produces

```
A3 =
 10.0525    8.5868    8.3993    7.5761    7.8396    6.7830
  8.4444    7.4347    7.1933    6.4806    7.6550    6.3822
 13.4295    7.3008    8.7857   12.2273    7.8420    8.1255
  9.5482    8.7824    8.3539    7.1628    8.9940    7.3699
 10.7867    8.8079    8.7791    8.4301    8.5458    7.4533
 12.4866    9.5558    9.7725    9.6321    7.6576    7.2346
  6.8006    6.7118    6.2256    5.1178    7.6984    6.0371
  3.3802    4.1632    3.7051    5.6370   18.6357   12.9369
 11.5959   10.4370    9.9812    7.9727    7.3527    6.5611
  7.4222    5.6604    5.8332    6.7189    8.5417    7.0273
  5.7201    5.0978    4.9371    5.1401    8.4044    6.5073
```

```
np.linalg.norm ( A–A3,' fro' ) = 5.22
```

Let's try to be cheap, and imagine that we want to settle for $A2$ as our approximation to A. We can compare A and A_2 in Column 0 and in Column 3:

```
    Column 0              Column 3
    A      A2             A      A2

 10.00  10.21           7.46   8.03
  8.00   8.61           6.77   6.95
 13.00  12.52          12.74   9.64
  9.00   9.83           7.11   7.98
```

```
11.00  10.83          7.81   8.58
14.00  12.43          8.84   9.47
 6.00   7.09          6.08   5.96
 4.00   3.30          5.39   5.40
12.00  11.98          8.15   9.09
 7.00   7.25          6.42   6.25
 5.00   5.74          5.73   5.22
```

Agreement isn't perfect, but we can certainly recognize that $A2$ is trying hard to approximate, and not doing a bad job.

The singular value decomposition (SVD) is a powerful linear algebra tool. This example has demonstrated how the SVD can detect that a given set of $m \times n$ data (a matrix) has a pattern that can be approximated in a way that uses a significantly smaller amount of information.

In this small example, we approximated the 11×6 entries of A by a rank $k = 2$ approximation using 2*11 entries in U, 2 entries in S, and 2*6 entries of V, dropping from an exact representation using 66 numbers to an approximate representation that uses 36 numbers. In real applications, the data sizes are in the thousands and millions. Using a typical approximation of rank $k = 10$ or $k = 20$ can result in enormous reductions in storage and efficiency.

The behavor of the set of singular values produced by the SVD indicates whether a matrix is strongly patterned. If the first few singular values are much larger than later ones, then the information in the array can be greatly compressed. On the other hand, if the data is unpatterned or random, then usually most of the singular values will have roughly the same value, meaning that it is not possible to drop some of the later components and still have a good approximation.

6 Image compression

In a black and white image, (actually a "gray scale" image) a single number records the brightness of each pixel. Such an image is an $m \times n$ numeric array A. m and n are typically of the order of 1,000

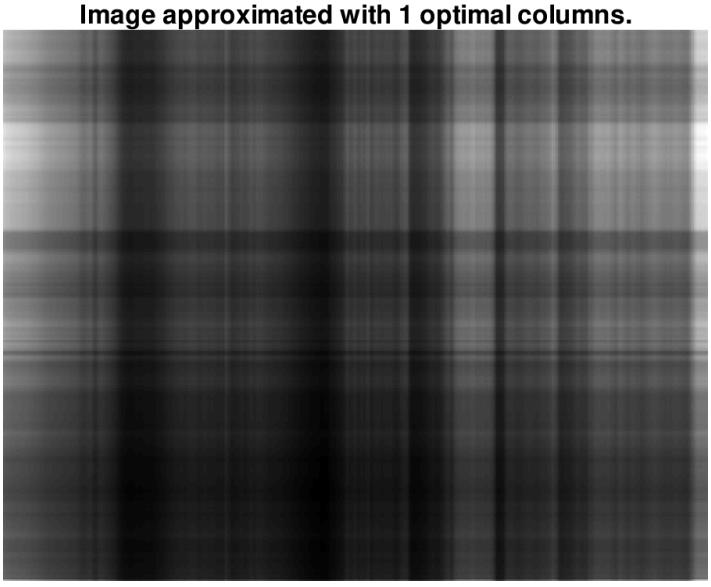
We will think of the image as a collection of n columns, each containing m values. Since this is a gray scale image, the values will be numbers between 0 and 1. Now imagine two consecutive columns of the image. It is very likely that the pair of values in the same row will be close, since they represent neighboring parts of the picture. So it's also likely that the values in column j and $j + 1$ are very similar. We are not dealing with random data, and so we may have a good chance of compressing our image, by approximating several similar columns by scaled copies of a single column.

For any value k, we can ask the SVD to find A_k, the best rank k approximation of A. The storage of A_k can be much less than A requires, while A_k preserves most of the information in A.

Let's take a sample image, think of it as a matrix A, compute its SVD, and then look at some typical approximating images, just like we did for the little matrix example in the previous discussion.

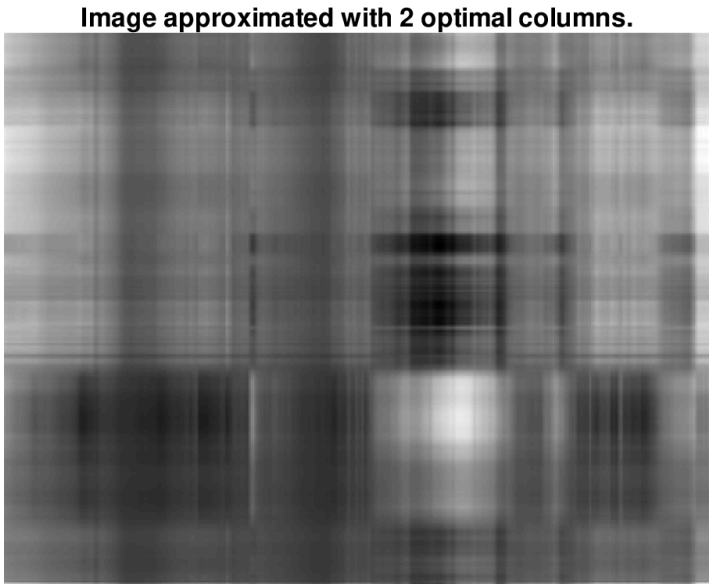
The original image, which uses $n = 460$ columns of data.

The original image is a still photo from the movie *Casablanca*. Now we compute the singular value decomposition, and create an approximate image $A1$ based on a single component of the SVD factorization. We can't expect much, but there is a tiny bit of light/dark information that is preserved:



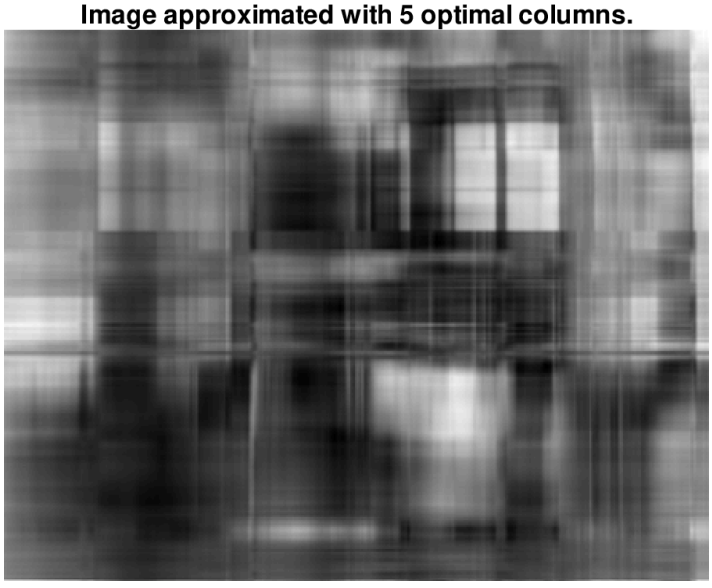
Approximation A1. The top of the image is bright, the bottom dark. There are a few columns of darkness.

When we move to using two components of the SVD, we still can't expect to get much. Nonetheless, the approximation can start to focus on some areas of the picture which are heavily bright or dark.



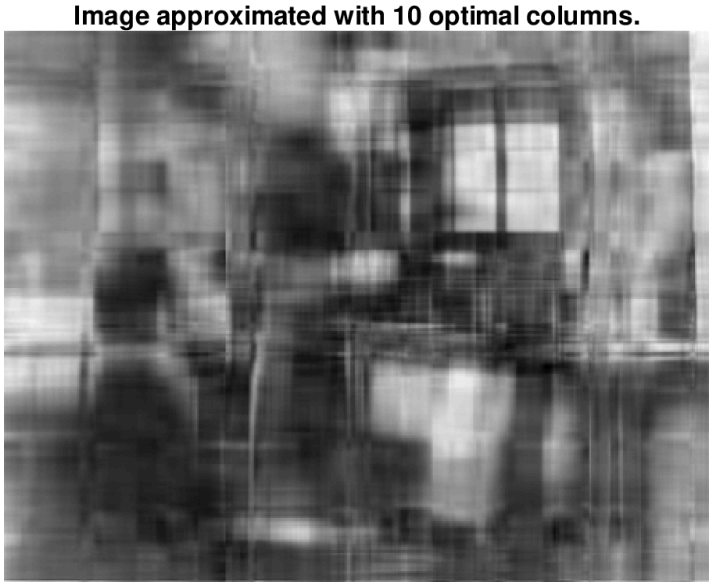
Approximation A2. The men's suits are showing up dark; The piano music is light.

Surprisingly, using just five components, the window, a feature from the original photo, seems to pop out. Since it represents a rectangle of brightness, it's not hard to imagine that the SVD noticed this regular structure.



Approximation A5. We can see the window! We can almost make out three people.

The 10-component approximation is still terrible, but we can actually start to see the shapes of the three characters, the piano, and the general setting.



Approximation A10. There are three people, although they look ghostly.

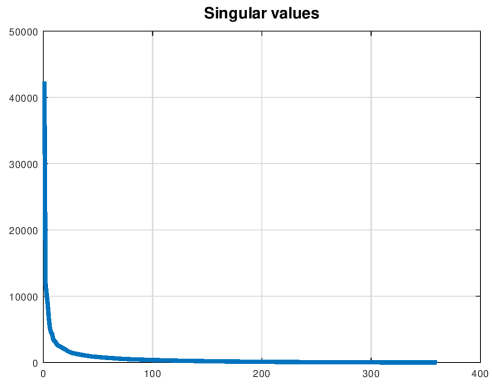
The 20, 40 and 80 component approximations all seem to present the information from the original photo, but with somewhat fuzzy outlines. If we were trying to be efficient, we might simply opt for the 40 component version.

Approximation A20. Looks like a badly printed photograph. But we can see all the important objects

Approximation A40. Now the image is grainy.

Approximation A80. It's hard to see anything wrong here. And yet we have used 80 columns of data and omitted 380.

Each singular value represents the size of the next addition to the approximation. For the photographic image, these values drop rapidly, suggesting that a low rank approximation gets most of the information.



The singular values of the image.

Assumptions:

- Choose the pixels of the image as our data items.
- Neighboring pixel values are usually related, not random.
- Groups of similar pixels have low information, and can be approximated.

Approximations:

- Rank 5 approximation shows light / dark patterns in the room;
- Rank 10 approxmation made $2\frac{1}{2}$ people visible;
- Rank 40 approximation shows all important features;

Conclusions:

- The photograph is a combination of information and minor details;
- SVD approximations expose important information first;
- Importance decreases with singular value size;
- A low rank SVD approximation may store all information we need;

7 Exercises

1. Consider a boundary value problem in the interval $[0.0, 5.0]$, of the form

   $$-u''(x) = -(2x + 5)\, e^x$$

   with boundary conditions

   $$u(0.0) = 1$$
   $$u(5.0) = 11\, e^5 \approx 1632.5$$

   for which the exact solution is

   $$u_{exact}(x) = (2x + 1)\, e^x$$

   Solve this problem using `n=21` nodes, and plot your estimated solution against the exact solution.
2. Our linear least squares problem can be solved directly, as a 15×2 overdetermined system of linear equations $A \cdot x = rhs$. Here, for example, the first entries of A and rhs will be $1, x_0$ and y_0. The `numpy` function `np.linalg.lstsq(A,b)` will set up the corresponding least squares system and return the solution. For "technical" reasons, you will want use a call statement like

```
x, _, _, _ = np.linalg.lstsq ( A, rhs, rcond = None )
```

where the three underscores allow you to ignore extra output. The array `x` will return the two values `b, m` that we are seeking. If you try this approach, you should get exactly the same values we got previously. This approach is much more convenient when the number of variables n increases.

3. Suppose we are trying to find an approximate formula for data, but plotting the data values produces a graph that is too curved to be approximated by a straight line. Then we could consider higher order models, starting with a quadratic of the form $y = c_0 + c_1 \cdot x + c_2 \cdot x^2$. What would be the form for the 3×3 matrix A and right hand side rhs then? Consider the following data:

```
x = [ 0,  2,  3,  5,  6,  8,  9,   11, 12, 14, 15 ]
y = [ 0, 10, 10, 10, 10, 10, 10.5, 15, 50, 60, 85 ]
```

Use the least squares approach to compute the linear and quadratic approximations to this data. Show the data and the two approximating functions on a single plot.

4. In the museum problem, the only passage leaving room K goes to room O. Investigate what happens if this passage is locked, so that room K becomes a dead end. Adjust the transition matrix T, and simulate 10, 20, and then 100 steps. What do you observe?
5. In the museum problem, from any starting room, you can reach any other room. Alter the museum map as follows: remove the path from A to B, from C to F, and from E to F. Add a path from C to B. Reverse the direction of the path from C to E, so that it becomes a path from E to C. Your map now has an isolated loop C -¿ B -¿ E -¿ C, which is inaccessible to room A. Update the transition matrix for this new situation, and call `w, v = np.linalg.eig ( T )` to compute the eigenvalues `w` and eigenvectors `v`. You should notice that you now have two eigenvalues equal to 1. Their corresponding eigenvectors describe wandering walks around the big loop (that includes room A) and the little loop, involving just C, B, and E. Because the map is now disconnected into two separate regions, it no longer satisfies the "irreducible" condition of the Perron-Frobenius theorem, and hence we no longer can guarantee a single dominant eigenvalue equal to 1.
6. Compute the principal moments of inertia for the molecule CH_3Cl, whose atomic information is

```
m        x        y        z
12       0        0        0
34.9689  0        0        1.7810
1.0079   1.0424   0       -0.3901
1.0079  -0.5212   0.9027  -0.3901
1.0079  -0.5212  -0.9027  -0.3901
```

7. Compute the principal moments of inertia for the molecule NH_3, whose atomic information is

```
m        x        y        z
14.0031  0        0        0
1.0079   0       -0.9377  -0.3816
1.0079   0.8121   0.4689  -0.3816
1.0079  -0.8121   0.4689  -0.3816
```

8. Compute the principal moments of inertia for the molecule O_3, whose atomic information is

```
m        x        y        z
15.9949  0        0        0
15.9949  0        1.0885   0.6697
15.9949  0       -1.0885   0.6697
```

9. We have suggested that matrices with random data are hard to compress. Carry out an example similar to the compression of the 11×6 matrix, but this time, initialize A to random values. What

are your singular values in the S matrix? Print the successive values of the Frobenius norm of the difference of A and A1, A2, ..., A6.

10. We have suggested that matrices with patterned data can sometimes be efficiently compressed. Let A be a 10×10 magic matrix, generated using the file *magic_matrix.py*. Compute the singular value decomposition of this matrix. What are your singular values in the S matrix? Print the successive values of the Frobenius norm of the difference of A and A1, A2, ..., A10. At what point is your relative approximation error less than 10%?