Department of Mathematics and Computer Science
University of Southern Denmark, Odense

November 27, 2018
Marco Chiarandini

DM561/DM562 – Linear Algebra with Applications

In *curve fitting* in $\mathbb{R}^2$ we are given m points (pairs of numbers) $(x_1, y_1), \ldots, (x_m, y_m)$ and we want to determine a function $f(x)$ such that

$$f(x_1) \approx y_1, \cdots, f(x_m) \approx y_m.$$

The type of function (for example, polynomials, exponential functions, sine and cosine functions) may be suggested by the nature of the problem (the underlying physical law, for instance), and in many cases a polynomial of a certain degree will be appropriate.
Let's assume we have a set of data collected by some measurements. For example, they can be temperature in the atmosphere taken at different days of the year or the cost of houses given their square meters.
For carrying out the computation we will simulate the set of points using the following Python script.

```
#!/usr/bin/python
import numpy as np
import matplotlib.pyplot as plt



np.random.seed(FIRST_6_DIGITS_OF_YOUR_CPRN))
np.set_printoptions(precision=3)



m=101
x = np.linspace(0, 1, m)
y = x**3-7*x+np.random.exponential(1,m)

plt.plot(x, y, '.')
#plt.axis.xlabel=('x')
#plt.axis.Axis.ylabel=('x')
plt.show()

print x
print y
```

In the script you have to use the first 6 digits of CPR number.
The script produces two arrays x and y. The jth elements of these arrays give us the jth point, (x_j, y_j). An example of collected points is shown in Figure 1.

(a) The first case we consider is fitting a straight line $y = a + bx$. Clearly, there is no line that passes through all the points at the same time. However, we can search for the line that minimizes the distance from all the points. More precisely, given the points $(x_1, y_1), \ldots, (x_m, y_m)$ we search for the line such that the sum of the squares of the distances of those points from the straight line is minimum, where the distance from (x_j, y_j) is measured in the vertical direction (the y-direction).

Formally, each point j with abscissa x_j has the ordinate $a + bx_j$ in the fitted line. The distance from the actual data (x_j, y_j) is thus $|y_j - a - bx_j|$ and the sum of squares is

$$q = \sum_{j=1}^{m} (y_j - a - bx_j)^2$$

Hence, q depends on a and b whose values we are trying to determine, while the values x_j and y_j are given being the coordinates of the points available. From calculus we know that the minimum of a function occurs where the partial derivatives are zero.

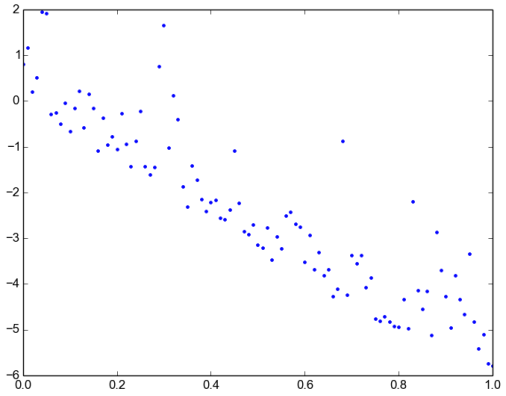


Figure 1: An example of points generated by the Python script plotted on the plane xy.

Do this, calculate the partial derivatives and find a and b with the help of Python. Report the symbolic derivations, the code you wrote and a plot that shows the points you generated and the line you fitted. The code must be short and use as much as possible matrix calculations by means of numpy or scipy arrays.

[Hint: you can look up in Wikipedia for an explanation of partial derivatives. In short, the partial derivatives of q with respect to the variables a and b, denoted $\frac{\partial q}{\partial a}$ and $\frac{\partial q}{\partial b}$, are, respectively, the derivatives of q with respect to a and b while the other variable is considered like a constant.]

Solution:

A necessary condition for q to be minimum is

$$\frac{\partial q}{\partial a} = -2\sum_{j=1}^{m}(y_j - a - bx_j) = 0$$

$$\frac{\partial q}{\partial b} = -2\sum_{j=1}^{m} x_j(y_j - a - bx_j) = 0$$

We can rewrite as:

$$\begin{cases} am & + & b\sum x_j & = & \sum y_j \\ a\sum x_j & + & b\sum x_j^2 & = & \sum x_j y_j \end{cases}$$

which is a system of linear equations in the variables $[a, b]$. Hence, the solution to this systems gives the values of a and b that minimize the square distance.

```
# Task 1
a_11 = len(x)
a_12 = sum(x)
a_22 = sum(x**2)
b_1 = sum(y)
b_2 = sum(x*y)


A = np.array([[a_11, a_12],[a_12,a_22]])
b = np.array([b_1, b_2])



a_11 = len(x)
a_12 = sum(x)
a_22 = sum(x**2)
b_1 = sum(y)
```

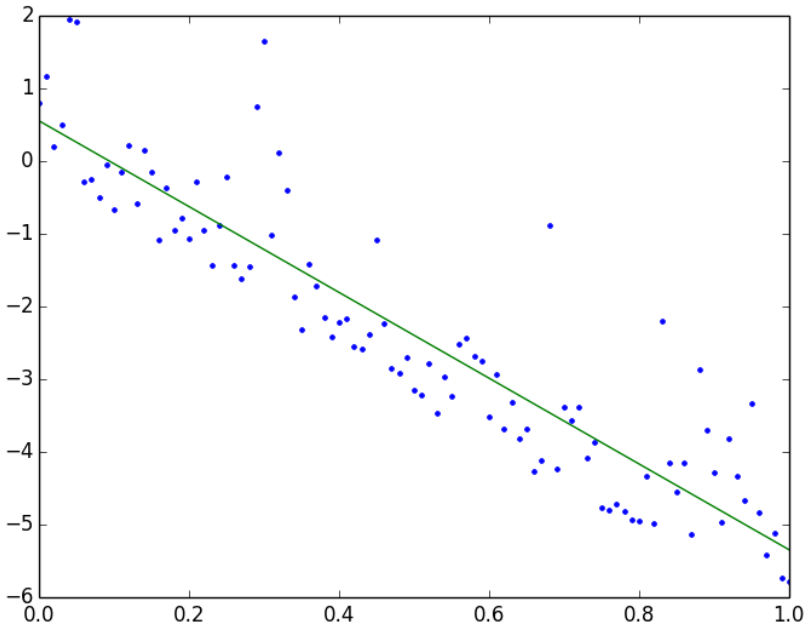


Figure 2:

```
b_2 = sum(x*y)


A = np.array([[a_11, a_12],[a_12,a_22]])

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

# y = coeff[0] + coeff[1]*x


plt.plot(x, y, '.')
plt.plot([0,1],[coeff[0],sum(coeff)])
plt.show()
```

The resulting plot is reported in Figure 2.

(b) Let us now try to obtain a better fit by using a second order polynomial approximation. We can generalize the polynomial $y = ax + b$ to a polynomial of degree k

$$p(x) = b_0 + b_1 x + \cdots + b_k x^k$$

where $k \leq m - 1$. Then q takes the form

$$q = \sum_{j=1}^{m} (y_j - p(x_j))^2$$

and depends on $k + 1$ parameters $b_0, \cdots, b_k$. As before the setting of the parameters that yields the minimum q can be found via partial derivatives.

For the case of a second order polynomial, ie, for $k = 2$, calculate the partial derivatives and find the parameters $b_0, \cdots, b_2$ with the help of Python. Report the symbolic derivations, the code you wrote and a plot that shows the points you generated and the cubic curve you fitted. The code must be short and use as much as possible matrix calculations by means of numpy or scipy arrays.

Discuss which of the two curves fitted, the straight line and the second order polynomial, yields the best model for the points.

Solution:

The necessary condition for q to be minimum gives a $k+1$ system of linear equations:

$$\begin{aligned}\frac{\partial q}{\partial b_0} &= 0\\ \frac{\partial q}{\partial b_1} &= 0\\ &\vdots\\ \frac{\partial q}{\partial b_k} &= 0\end{aligned}$$

For $k = 3$ the system we obtain is (summations are all from 1 to m):

$$\begin{cases} b_0 m + b_1 \sum x_j + b_2 \sum x_j^2 + b_3 \sum x_j^3 = \sum y_j \\ b_0 \sum x_j + b_1 \sum x_j^2 + b_2 \sum x_j^3 + b_3 \sum x_j^4 = \sum x_j y_j \\ b_0 \sum x_j^2 + b_1 \sum x_j^3 + b_2 \sum x_j^4 + b_3 \sum x_j^5 = \sum x_j^2 y_j \\ b_0 \sum x_j^3 + b_1 \sum x_j^4 + b_2 \sum x_j^5 + b_3 \sum x_j^6 = \sum x_j^3 y_j \end{cases}$$

which is a system of linear equations in the variables $[b_0, \ldots, b_k]$. Hence, the solution to this systems gives the values of $[b_0, \ldots, b_k]$ that minimize the square distance.

```
# Task 2
a_0 = len(x)

a_1 = sum(x)
a_2 = sum(x**2)
a_3 = sum(x**3)

b_1 = sum(y)

a_4 = sum(x**4)
b_2 = sum(x*y)

a_5 = sum(x**5)
b_3 = sum((x**2)*y)

a_6 = sum(x**6)
b_4 = sum((x**3)*y)


A = np.array([[a_0, a_1, a_2,a_3],
              [a_1,a_2,a_3,a_4],
              [a_2,a_3,a_4,a_5],
              [a_3,a_4,a_5,a_6]
])

coeff = np.linalg.solve(A,[b_1,b_2,b_3,b_4])

P=np.poly1d(coeff[::-1])
print P

plt.plot(x, y, '.')
xx = np.linspace(0.0, 1.0, 50)
plt.plot(xx, P(xx), '-')
#plt.axhline(y=0)
plt.title('Polynomial of order 3');
plt.show()
```

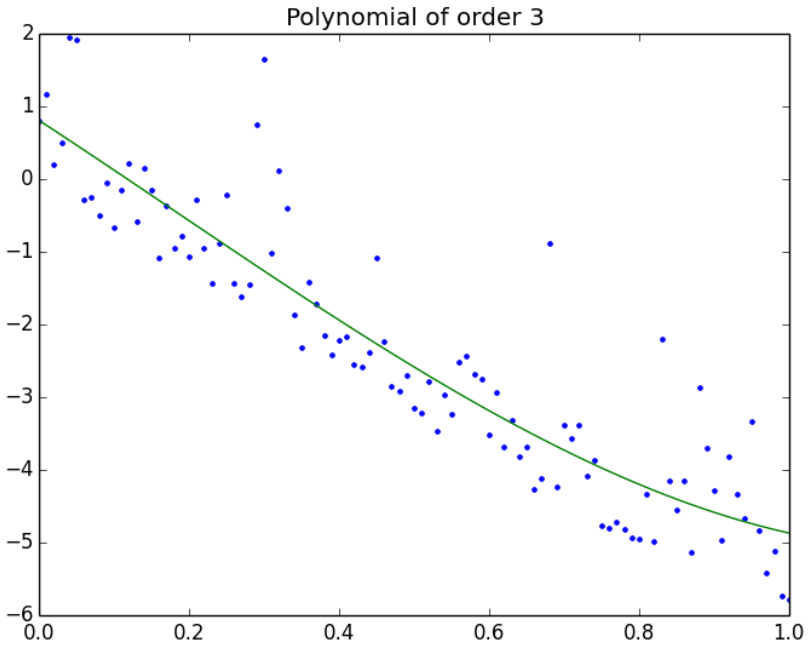


Figure 3:

The resulting plot is reported in Figure 3.

We see that the third degree polynomial fits better the points since the line has more degree of freedom. The decision about which fitting is the best may depend on the application and on the knowledge about the underlying model. If it is known that there is a linear correspondence between the variables then the model to use is the linear. Otherwise, a third degree might be better provided that it does not overfit the data. Cross validation is a technique used in statistical learning to avoid overfitting.

(c) In the previous points we moved forward to a solution using calculus. Now we take a different approach using linear algebra and we will show that we reach the same result. In doing this we generalize the method to the space $\mathbb{R}^n$. This is useful, for example, in tasks of machine learning where one is interested in learning from data the type of dependency of a variable y on some predictor variables $x_1, \ldots, x_n$. For example, the temperature on the ground may depend, beside on the day of the year, also on the latitude and maybe, in the case of global warming, on the development through years. The price of a house may depend on the square meters but also on the distance from city center and the year of construction.

We assume that the variable y is related to $\mathbf{x} \in \mathbb{R}^n$ linearly, so for some constants b_0 and $\mathbf{b}$, $y = b_0 + \mathbf{b}^T\mathbf{x}$. Given the set of m points in the ideal case we have that $y_j = b_0 + \mathbf{b}\mathbf{x}_j$, for all $j = 1, \ldots, m$. In matrix form:

$$\begin{bmatrix} 1 & x_{1,1} & \cdots & x_{1,k} \\ 1 & x_{2,1} & \cdots & x_{2,k} \\ \vdots & \vdots & & \\ 1 & x_{m,1} & \cdots & x_{m,k} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_k \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_m \end{bmatrix}$$

This can be written as $A\mathbf{z} = \mathbf{y}$.

Considering the same set of data in $\mathbb{R}^2$ from point 1 of this Task try to find $\mathbf{z}$ in Python. Report what happens and explain why it is so.

Solution:

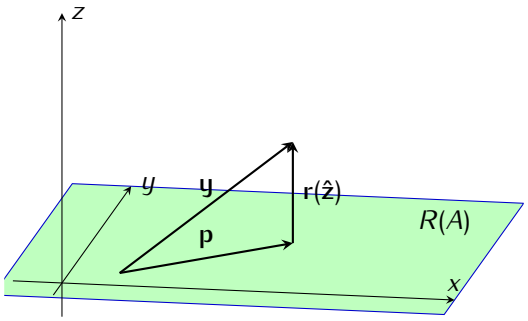


Figure 4: $\mathbf{y} \in \mathbb{R}^2$ and A is a 3×2 matrix of rank 2.

```
# Task 4

A = np.column_stack([np.ones(101),x])

np.linalg.solve(A,y)
```

Python will complain that the matrix is not square. The system is overdetermined.

(d) In general we cannot expect to find a vector $\mathbf{z} \in \mathbb{R}^{n+1}$ for which $A\mathbf{z}$ equals $\mathbf{y}$. Instead we can look for a vector $\mathbf{z}$ for which $A\mathbf{z}$ is closest to $\mathbf{y}$.

For each $\mathbf{z} \in \mathbb{R}^{n+1}$ we can form the *residual*

$$r(\mathbf{z}) = \mathbf{y} - A\mathbf{z}$$

Recalling the definition of the norm or length of a vector, the distance between $\mathbf{y}$ and $A\mathbf{z}$ is given by

$$\|r(\mathbf{z})\| = \|\mathbf{y} - A\mathbf{z}\|$$

Minimizing $\|r(\mathbf{z})\|$ is equivalent to minimize $\|r(\mathbf{z})\|^2$. So what we need is for $A\mathbf{z}$ to be the closest point of the form $A\mathbf{u}$ (for a generic vector $\mathbf{u} \in \mathbb{R}^{n+1}$) to $\mathbf{y}$. That is, $A\mathbf{z}$ has to be the closest point in $R(A)$ to $\mathbf{y}$.

Thus a vector $\hat{\mathbf{z}}$ will be the one that minimizes $\|\mathbf{y} - A\mathbf{z}\|^2$ if and only if $\mathbf{p} = A\hat{\mathbf{z}}$ is the vector in $R(A)$ that is closest to $\mathbf{y}$. The vector $\mathbf{p}$ is said to be the *projection* of $\mathbf{y}$ onto $R(A)$. It follows that

$$r(\hat{\mathbf{z}}) \in R(A)^{\perp}$$

that is, $r(\hat{\mathbf{z}})$ lays on the subspace of $\mathbb{R}^m$ that is the *orthogonal complement* of $R(A)$. These facts are explained visually in Figure 4 for the case of $\mathbb{R}^2$ but they can be proved to hold in more general terms for $\mathbb{R}^n$.

To proceed we need another fact. For an $m \times (n+1)$ matrix A,

$$R(A)^{\perp} = N(A^T),$$

that is, the orthogonal complement of the range of a matrix A is the null space of the transpose of the matrix A.

The proof of this fact is as follows:

A vector $\mathbf{w}$ that belongs to $R(A)$ belongs to the linear span of A, ie, it is a linear combination of the columns of the matrix A. A vector $\mathbf{v}$ that belongs to $R(A)$, must therefore be orthogonal to each column of the matrix A. Consequently $A^T\mathbf{v} = \mathbf{0}$. Thus, $\mathbf{v}$ must be an element of $N(A^T)$ and hence $N(A^T) = R(A)^{\perp}$.

(e) Thus we have

$$r(\hat{\mathbf{z}}) \in N(A^T)$$

Show that this leads to a $(n+1) \times (n+1)$ system of linear equations in the $\mathbf{z}$ variables. The system is actually the same as the one derived earlier via calculus.

Solution:

Since $r(\hat{\mathbf{z}}) = \mathbf{y} - A\mathbf{z}$ we have

$$A^T(\mathbf{y} - A\mathbf{z}) = \mathbf{0}$$

Thus to solve the least square system of $A\mathbf{z} = \mathbf{y}$ we have to solve

$$A^T\mathbf{y} = A^TA\mathbf{z}$$

Remembering that we are solving in the $\mathbf{z}$ variables we have a system of size $(n+1)\times(n+1)$ because A is of size $m \times (n+1)$ and therefore A^TA is of size $(n+1) \times (n+1)$.

(f) Show that if A is an $m \times (n+1)$ matrix of rank $(n+1)$, then the system at the previous point has a unique solution:

$$\hat{\mathbf{z}} = (A^TA)^{-1}A^T\mathbf{y}$$

(Hint: assume that $A^TA\mathbf{u} = \mathbf{0}$ has only the trivial solution $\mathbf{u} = \mathbf{0}$.)

Solution:

Under the assumption that $A^TA\mathbf{u} = \mathbf{0}$ has only the trivial solution $\mathbf{u} = \mathbf{0}$ then the matrix A^TA is non-singular and hence invertible and the solution unique.

(g) Use the formula derived above to find the least square linear regression of point (a). Compare your result with the result you would obtain by using the function from numpy: `numpy.linalg.lstsq`.

Solution:

In Python, the least square solution as derived here to our original data can be computed as:

```
### http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html

A = np.vstack([np.ones(len(x)), x]).T

b_0, b_1 = np.linalg.lstsq(A, y)[0]
print b_0, b_1

# Plot the data along with the fitted line:

plt.plot(x, y, 'o', label='Original data', markersize=10)
plt.plot(x, b_1*x + b_0, 'r', label='Fitted line')
plt.legend()
plt.show()
```

(h) We now turn to another application in which we know that the data sampled come from a circle. For example, in image processing from a medical application we may collect the data regarding points of a bone and we want to determine the circumference of the bone.

The parametric equations for a circle with center $(3, 1)$ and radius 2 are

$$\begin{aligned} x &= 3 + 2\cos t \\ y &= 1 + 2\sin t \end{aligned}$$

We simulate a few points in Python by adding some noise to the points from a circle in correspondence of $t \in \{0.0, 0.5, 1.0, \ldots, 5.0, 5.5\}$:

```
#!/usr/bin/python
import numpy as np
import matplotlib.pyplot as plt

CPRN = 3 # your CPR number
np.random.seed(CPRN)
np.set_printoptions(precision=3)

t=np.arange(0,6,0.5)
x=3+2*np.cos(t)+0.3*np.random.rand(len(t))
y=1+2*np.sin(t)+0.3*np.random.rand(len(t))


plt.plot(x, y, '.')
plt.show()
```

Write in your report the symbolic derivations that lead to finding the center $\mathbf{c}$ and radius r of the circle that gives the best q that fits to the points, ie,:

$$x = c_1 + r\cos(t)$$
$$y = c_2 + r\sin(t)$$

Solve the model you found in Python. Report the code and the plot of the points and the fitted circle.

Solution:

The circle can be written as

$$(x - c_1)^2 + (y - c_2)^2 = r^2$$

Solving in y we obtain

$$y = \pm\sqrt{r^2 - (x - c_1)^2} + c_2$$

Then the least squares gives:

$$q = \sum (y_j \pm \sqrt{r^2 - (x_j - c_1)^2} + c_2)$$

which we need to optimize in r^2, c_1, c_2. This leads to a non linear system of equations that is not convenient for us.

Luckily, we are given the points by means of their radial coordinate t rather than just with the (x, y) coordinates. (This is seldom the case since it implies knowing a priori the model that we are actually trying to find.)

Let's then try to minimize the squared distance of the points from the circle in the parameters r^2, c_1, c_2:

$$\begin{aligned} q &= \sum_{j=1}^{m} \left[(x_j - c_1 - r\cos(t_j))^2 + (y_j - c_2 - r\sin(t_j))^2\right] \\ &= \sum_{j=1}^{m} \left[(x_j - c_1)^2 + r^2\cos^2(t_j) - 2(x_j - c_1)(r\cos(t_j)) + (y_j - c_2)^2 + r^2\sin^2(t_j) - 2(y_j - c_2)(r\sin(t_j))\right] \\ &= \sum_{j=1}^{m} \left[(x_j - c_1)^2 - 2(x_j - c_1)(r\cos(t_j)) + (y_j - c_2)^2 - 2(y_j - c_2)(r\sin(t_j)) + r^2\right] \end{aligned}$$

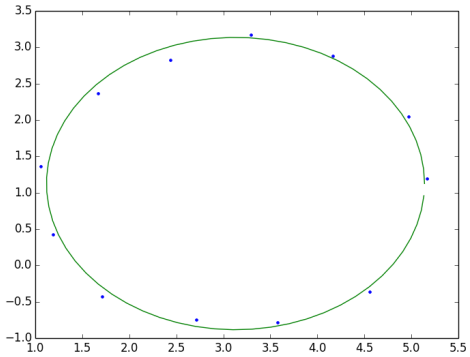


Figure 5:

$$\frac{\partial q}{\partial r} = \sum_{j=1}^{m} \left[-2(x_j - c_1)\cos\left(t_j\right) - 2(y_j - c_2)\sin\left(t_j\right) + 2r\right] = 0$$
$$\frac{\partial q}{\partial c_1} = \sum_{j=1}^{m} \left[-2(x_j - c_1) + 2(r\cos\left(t_j\right))\right] = 0$$
$$\frac{\partial q}{\partial c_2} = \sum_{j=1}^{m} \left[-2(y_j - c_2) + 2(r\sin\left(t_j\right))\right] = 0$$

Luckily, this gives rise to a linear system:

$$\begin{bmatrix} m & \sum\cos\left(t_j\right) & \sum\sin\left(t_j\right) \\ \sum\cos\left(t_j\right) & m & 0 \\ \sum\sin\left(t_j\right) & 0 & m \end{bmatrix} \begin{bmatrix} r \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \sum x_j\cos\left(t_j\right) + y_j\sin\left(t_j\right) \\ \sum x_j \\ \sum y_j \end{bmatrix}$$

Solving it we find r, c_1, c_2.

```
m=len(x)
A = np.array([[m,np.sum(np.cos(t)),np.sum(np.sin(t))],
              [np.sum(np.cos(t)),m,0],
              [np.sum(np.sin(t)),0,m]]
             )
b = np.array([np.sum(x*np.cos(t)+y*np.sin(t)),
              np.sum(x),
              np.sum(y)])

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

print rcc
t=np.arange(0,2*np.pi,0.1)
xx=rcc[1]+rcc[0]*np.cos(t)
yy=rcc[2]+rcc[0]*np.sin(t)

plt.plot(x, y, '.')
plt.plot(xx,yy,'-')
plt.show()
```

The result is shown in Figure 5.

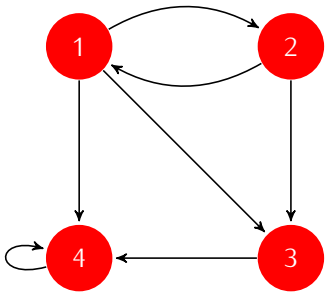
Exercise 1 Modeling a web surfer: Google's PageRank

[This exercise is an adaptation from Philip Klein's course: "Coding the Matrix"]
PageRank is the algorithm that Google originally used to determine the "importance" (or rank) of a web page.

> The idea for PageRank is this: Define a Markov chain that describes the behavior of a random web-surfer. Consider the stationary distribution of this Markov chain. Define the weight of a page to be the probability of that page in the stationary distribution. Higher-probability pages are considered better. So when the user submits a query consisting of a set of words, present the web pages containing these words, in descending order of probability.
>
> Conveniently, the PageRank vector (the stationary distribution) does not depend on any particular query, so it can be computed once and then used for all subsequent queries. (Of course, Google periodically recomputes it to take into account changes in the web.)

We start with a rudimentary Markov chain and we discover why it needs to be improved.

a) In each iteration, the random surfer selects an outgoing link from his current web page, and follows that link. (If the current web page has no outgoing link, the surfer stays stand.) We consider the small Web network of the picture, consisting of only four sites linked together as shown.



Let $k(j)$ denote the number of links from page j to other pages in the network. If $k(j) \neq 0$ and no other knowledge about the relevance of the pages is given, then it is reasonable to assume that the surfer follows links from the current web page with probability:

$$a_{ij} = \begin{cases} 1/k(j) & \text{if there is a link from page } j \text{ to page } i \\ 0 & \text{otherwise} \end{cases}$$

Write the transition matrix A_1 of the Markov chain.

According to this Markov chain, how likely is that the random surfer is at each page after many iterations? What are the most likely pages?

Use the power method and carry out the operations in Python for the following cases:

- if it starts at page 2 and takes an even number of iterations
- if it starts at page 2 and takes an odd number of iterations
- if it starts at page 3 and takes an odd number of iterations

You should find that the answer depends on where the surfer starts and how many steps he takes.

Solution:

```
A=np.matrix([[f(0),f(1,2),f(0),f(0)],[f(1,3),f(0),f(0),f(0)],[f(1,3),f(1,2),f(0),f(0)
    ],[f(1,3),f(0),f(1,1),f(1)]])
np.linalg.det(A)
x_0=np.array([0,1,0,0])
print A
A_n=A
for i in range(21):
    A_n = np.dot(A_n,A)
x_n = np.dot(A_n,x_0)
x_n
```

The probabilities are almost the same except that the probabilities of 1 and 2 are swapped.

b) From the point of view of computing definitive pageranks using the power method, there are two things wrong with this Markov chain:

   i. There are multiple clusters in which the surfer gets stuck. One cluster is page 1 and page 2, and the other cluster is page 4.

   ii. There is a part of the Markov chain that induces periodic behavior: once the surfer enters the cluster page 1, page 2, the probability distribution changes in each iteration.

   The first property implies that there are multiple stationary distributions. The second property means that the power method might not converge. We want a Markov chain with a unique stationary distribution so we can use the stationary distribution as an assignment of importance weights to web pages. We also want to be able to compute it with the power method. We apparently cannot work with the Markov chain in which the surfer simply chooses a random outgoing link in each step.

   Consider then an alternative very simple Markov chain: the surfer jumps from whatever page he's on to a page chosen uniformly at random.

   Write the transition matrix A_2 for this Markov chain. Is there a unique stationary distribution? Which one?

c) As you may have discovered the latter Markov chain does not in any way reflect the structure of the Web network. Using the stationary distribution to assign weights would provide no information at all.

   Instead, we will use a mixture of these two Markov chains. That is, we will use the Markov chain whose transition matrix is

$$A = 0.85A_1 + 0.15A_2$$

   Show that the matrix A is a *stochastic matrix*, that is, the sum of the entries in all columns is 1.

Solution:

Since every column of A_1 sums to 1, every column of $0.85A_1$ sums to 0.85, and since every column of A_2 sums to 1, every column of $0.15A_2$ sums to 0.15, so (finally) every column of $0.85A_1 + 0.15A_2$ sums to 1.

The Markov chain corresponding to the matrix A describes a surfer obeying the following rule:

- With probability 0.85, select one of the links from the current web page, and follow it.
- With probability 0.15, jump to a web page chosen uniformly at random. (This is called teleporting in the context of PageRank.)

You can think of the second item as modeling the fact that sometimes the surfer gets bored with where he is. However, it plays a mathematically important role. The matrix A is a positive matrix (every entry is positive). A theorem ensures that there is a unique stationary distribution, and that the power method will converge to it.

Calculate how likely it is that the random surfer is at each page after many iterations and rank consequently the pages. You can use the theory of diagonalization or the power method and carry out the calculation with Python.