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

March 19, 2018
Marco Chiarandini

DM559 – Linear and Integer Programming

Obligatory Assignment 0.1, Spring 2018

Solution:

Contains Solutions!

Deadline: Saturday, March 3 at 12:00.

In red the modifications after publication.

This is the first obligatory assignment in DM559. The number 0.1 indicates that this is Assignment 1 of the part 0 of the course (the part on Linear Algebra). The part 1 will start in week 12.

The Assignment has to be carried out individually. You can consult with peers, if you are unable to proceed, but without exchanging full solutions.

The submission is electronic via:

`http://valkyrien.imada.sdu.dk/milpApp/`.

The deliverable is a PDF document, similarly as it will be at the written exam. Hence, experiment and get acquainted with the tools and forms you want to use for producing this document in a similar setting as the exam. In any case, you have to put your answers in this template You can handwrite your answers and add them as picture in the template. Be aware that at the exam you can use digital pen or hand scanner (that is, a silent scanner) but you cannot bring handycameras. You can of course also typset your answers in LaTeX. You can use either Danish or English. Keep the newpage separation after each exercise present in the template. Write your name and CPR number where indicated in the template.

In some parts, you are asked to write Python code. You have to include the code in the PDF document *together with the output of the execution of this code*. Use the LaTeX environment `lstlisting` as shown in the template for doing this. Your code must work correctly if fully copied and pasted from your report.

In your answers, you have to justify the steps that you are doing. If theorems and definitions from the slides of the course or from the Leon's text book (specify which edition) are used, give reference.

The Tasks are all about practical applications of linear algebra. Your goal is to answer correctly to as many subtasks as you can. Tasks and subtasks are presented in increasing order of difficulty. You are wlecome to ask in class for explanations that could put you on the right path for solving the tasks.

Exercises 1-5 must be completed to pass the assignment and I expect that they can be carried out in less than one working day if you are up-to-date with the subject. Exercises 6-8 are real-life applications of linear algebra. Exercise 8 might be challenging and it is however optional, that is, it does not count in the assessment (but give it a try if you have time).

Exercise 1*

For each of the following problems, answer whether the given matrix-matrix product is valid or not. If it is valid, give the number of rows and the number of columns of the resulting matrix (you need not provide the matrix itself).

1. $\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 3 & 1 & 2 \end{bmatrix}$
2. $\begin{bmatrix} 3 & 3 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 & 1 \\ 1 & 7 & 2 \end{bmatrix}$
3. $\begin{bmatrix} 3 & 3 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 & 1 \\ 1 & 7 & 2 \end{bmatrix}^T$
4. $\begin{bmatrix} 1 & 4 & 1 \\ 1 & 7 & 2 \end{bmatrix} \begin{bmatrix} 3 & 3 & 0 \end{bmatrix}^T$
5. $\begin{bmatrix} 1 & 4 & 1 \\ 1 & 7 & 2 \end{bmatrix} \begin{bmatrix} 3 & 3 & 0 \end{bmatrix}$
6. $\begin{bmatrix} 2 & 1 & 5 \end{bmatrix} \begin{bmatrix} 1 & 6 & 2 \end{bmatrix}^T$
7. $\begin{bmatrix} 2 & 1 & 5 \end{bmatrix}^T \begin{bmatrix} 1 & 6 & 2 \end{bmatrix}$

Solution:

- not valid
- not valid
- valid, 1×2
- valid, 2×1
- not valid
- valid, 1×1
- valid, 3×3

Exercise 2*

Let a, b be numbers and let $A = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix}$ $B = \begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix}$. What is AB? Write it in terms of a and b.

Solution:

$$\begin{bmatrix} 1 & b+a \\ 0 & 1 \end{bmatrix}$$

Exercise 3*

For each of the following matrix-vector equations, find the solution:

(a) $\begin{bmatrix} 2 & 0 & 1 & 3 \\ 0 & 0 & 5 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 3 \end{bmatrix}$

Solution:

I write only the solutions for the remaining ones, however, in the exam one has to show how the solution is reached.

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -3 \\ 0 \\ -2 \\ 3 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} t, \qquad t \in \mathbb{R}$$

(b) $\begin{bmatrix} 1 & 3 & -2 & 1 & 0 \\ 0 & 0 & 2 & -3 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ 2 \end{bmatrix}$

Solution:

The last row states that $0 = 2$ which is clearly not correct, hence the system is inconsistent.

(c) $\begin{bmatrix} 10^{-20} & 0 & 1 \\ 1 & 10^{20} & 1 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

Solution:

$$\begin{bmatrix} 10^{-20} & 0 & 1 & 1 \\ 1 & 10^{20} & 1 & 2 \\ 0 & 1 & -1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 10^{2} & 10^{2} \\ 1 & 10^{20} & 1 & 2 \\ 0 & 1 & -1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 10^{20} & 10^{20} \\ 0 & 10^{20} & 1-10^{20} & 2-10^{20} \\ 0 & 1 & -1 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 10^{20} & 10^{20} \\ 0 & 1 & 10^{-20}-1 & 2\cdot 10^{-20}-1 \\ 0 & 1 & -1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 10^{20} & 10^{20} \\ 0 & 1 & 10^{-20}-1 & 2\cdot 10^{-20}-1 \\ 0 & 0 & 10^{-20} & 4-10-20 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 10^{20} & 10^{20} \\ 0 & 1 & 10^{-20}-1 & 2\cdot 10^{-20}-1 \\ 0 & 0 & 1 & 4\cdot 10^{20}-1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 10^{20} & 10^{20} \\ 0 & 1 & 0 & (2\cdot 10^{-20}-1)-(10^{-20}-1)(4\cdot 10^{20}-1) \\ 0 & 0 & 1 & 4\cdot 10^{20}-1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 10^{20}-(4\cdot 10^{20}-1)10^{20} \\ 0 & 1 & 0 & (2\cdot 10^{-20}-1)-(10^{-20}-1)(4\cdot 10^{20}-1) \\ 0 & 0 & 1 & 4\cdot 10^{20}-1 \end{bmatrix}$$

And the last column gives the solution values of the variables.

Let's try in Python:

```
import numpy as np
AA = np.array([[1e-20,0,1,1],[1,1e20,1,2],[0,1,-1,3]])
AA[0,] = AA[0,]*1e20
AA[1,]=AA[1,]-AA[0,]
print(bmatrix((AA))
```

$$\begin{bmatrix} 1.e+00 & 0.e+00 & 1.e+20 & 1.e+20 \\ 0.e+00 & 1.e+20 & -1.e+20 & -1.e+20 \\ 0.e+00 & 1.e+00 & -1.e+00 & 3.e+00 \end{bmatrix}$$

Hence we observe that after the subtracting the first row to the second we obtain a quite different row than in our third matrix above:

$$\begin{bmatrix} 0.e+00 & 1.e+20 & -1.e+20 & -1.e+20 \end{bmatrix}$$

instead of

$$\begin{bmatrix} 0 & 10^{20} & 1-10^{20} & 2-10^{20} \end{bmatrix}$$

In particular the value ! in position A[2,3] is simply lost because it is a too small amount with respect to 10^{20}.

If we continue trying to put the last row in reduced form:

```
AA[2,]=AA[2,]-AA[1,]*1e-20
```

the matri that we obtain is:

$$\begin{bmatrix} 1.e+00 & 0.e+00 & 1.e+20 & 1.e+20 \\ 0.e+00 & 1.e+20 & -1.e+20 & -1.e+20 \\ 0.e+00 & 0.e+00 & 0.e+00 & 4.e+00 \end{bmatrix}$$

which tells us, differntly from what we found manually, that the system is inconsistent. So we come to a fasle conclusion.

Even using the direct methods in numpy

```
A = np.array([[1e-20,0,1],[1,1e20,1],[0,1,-1]])
b = np.array([1,2,3])
np.linalg.solve(A,b)
```

we are told that the matrix A is singular and the system inconsistent.

This problem can be mitigated by choosing the pivot element carefully:

Partial Pivoting at each iteration of the Gaussian elimination, among rows with nonzero entries in the first column choose row with entry having the largest absolute value.

In our case, in the first iteration of the algorithm, the row with largest value is the second row, hence let's swap it with the first and continue from there:

$$\begin{bmatrix} 10^{-20} & 0 & 1 & 1 \\ 1 & 10^{20} & 1 & 2 \\ 0 & 1 & -1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 10^{20} & 1 & 2 \\ 10^{-20} & 0 & 1 & 1 \\ 0 & 1 & -1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 10^{20} & 1 & 2 \\ 0 & -1 & 1-10^{-20} & 1-2\cdot 10^{-20} \\ 0 & 1 & -1 & 3 \end{bmatrix}$$

In the next selection of the pivot the absolute values are the same we can keep the matrix as it is:

$$\begin{bmatrix} 1 & 10^{20} & 1 & 2 \\ 0 & 1 & 10^{-20}-1 & 2\cdot 10^{-20}-1 \\ 0 & 1 & -1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 10^{20} & 1 & 2 \\ 0 & 1 & 10^{-20}-1 & 2\cdot 10^{-20}-1 \\ 0 & 0 & -10^{-20} & 4-2\cdot 10^{-20} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 10^{20} & 1 & 2 \\ 0 & 1 & 10^{-20}-1 & 2\cdot 10^{-20}-1 \\ 0 & 0 & 1 & 2-4\cdot 10^{20} \end{bmatrix}$$

Let's see in Python:

```
AA = np.array([[1,1e20,1,2],[1e-20,0,1,1],[0,1,-1,3]])
AA[1,]=AA[1,]-AA[0,]*10e-20
```

$$\begin{bmatrix} 1.e+00 & 1.e+20 & 1.e+00 & 2.e+00 \\ -9.e-20 & -1.e+01 & 1.e+00 & 1.e+00 \\ 0.e+00 & 1.e+00 & -1.e+00 & 3.e+00 \end{bmatrix}$$

We observe that Python would still have troubles dealing with the attempt to put to zero the entry A[2,1].

Partial pivoting is the technique most used in practice.

A more advance technique is:

Complete Pivoting Instead of selecting the order of the columns beforehand, in each iteration choose the column to maximize absolute value of pivot element.

Exercise 4*

Given

$$A = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 1 & 1 \\ -2 & 2 & -1 \end{bmatrix}$$

compute by hand $\det(A)$, $\text{adj}(A)$, and A^{-1}. Verify this last one in Python. Report your results.

Solution:

```
import numpy as np
A = np.array([[1,3,1],[2,1,1],[-2,2,-1]])
np.linalg.det(A)
# 2.9999999999999996
```

Python calculates the determinant by LU decomposition: Recall from slide 30 of Lec 5:

$$|AB| = |A||B|$$

Hence, for $A = PLU$ it is $|A| = |P||L||U|$. Since L and U are a lower and upper triangular matrix, respectively, their determinant is easily calculated by the product of the elements in the diagonal. P is a permutation matrix whose determinat is either $+1$ or -1 depending how many times rows are permuted:

```
import scipy.linalg as sl
P,L,U = sl.lu(A)
sl.det(P)*sl.det(L)*sl.det(U)
# 3.0
```

Hence, the matrix is invertible. The cofactor of a matrix is the matrix of minors of the matrix. The adjont of a matrix is the transpose of the cofactor.

$$\text{adj}(A) = \begin{bmatrix} -3 & 5 & 2 \\ 0 & 1 & 1 \\ 6 & -8 & -5 \end{bmatrix}$$

Then the inverse of the matrix is:

$$A^{-1} = \frac{1}{\det(A)}\text{adj}(A) = \begin{bmatrix} -1 & 5/3 & 2/3 \\ 0 & 1/3 & 1/3 \\ 2 & -8/3 & -5/3 \end{bmatrix}$$

Exercise 5*

Write the Cartesian equation and vector form of the line through the points: $[-3/2, 2]$ and $[3, 0]$.

Solution:

The vector equation is given by calculating the vector parallel to the line (hence $\mathbf{p}_2 - \mathbf{p}_1$) and a translation to one of the two points. Hence, a point $\mathbf{x}$ is on the line if:

$$\mathbf{x} = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1) = \begin{bmatrix} -3/2 \\ 2 \end{bmatrix} + t \left(\begin{bmatrix} 3 \\ 0 \end{bmatrix} - \begin{bmatrix} -3/2 \\ 2 \end{bmatrix} \right) = \begin{bmatrix} -3/2 \\ 2 \end{bmatrix} + t \begin{bmatrix} 9/2 \\ -2 \end{bmatrix}, \quad t \in \mathbb{R}$$

In class we commented that a linear space plus a translation is an affine space, hence its elements can be obtained by an affine combination of its points. We can then write the vector form of the line also as:

$$\mathbf{x} = \alpha_1 \begin{bmatrix} 3 \\ 0 \end{bmatrix} + \alpha_2 \begin{bmatrix} -3/2 \\ 2 \end{bmatrix}, \quad \alpha_1 + \alpha_2 = 1, \alpha_1, \alpha_2 \in \mathbb{R}$$

from where substituting $\alpha_2 = 1 - \alpha_1$ we reobtain the form above:

$$\mathbf{x} = \alpha_1 \begin{bmatrix} 3 \\ 0 \end{bmatrix} + (1 - \alpha_1) \begin{bmatrix} -3/2 \\ 2 \end{bmatrix} = \begin{bmatrix} -3/2 \\ 2 \end{bmatrix} + \alpha_1 \left(\begin{bmatrix} 3 \\ 0 \end{bmatrix} - \begin{bmatrix} -3/2 \\ 2 \end{bmatrix} \right), \quad \alpha_1 \in \mathbb{R}$$

To find the Cartesian equation we need to eliminate the parameter t in the following expression:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -3/2 \\ 2 \end{bmatrix} + t \begin{bmatrix} 9/2 \\ -2 \end{bmatrix}, \quad t \in \mathbb{R}$$

From the second component of the vector we obtain: $t = 1 - 1/2y$ and substituting in the first component:

$$4x + 9y - 8 = 0$$

Exercise 6

Consider a market with 3 industries producing 3 different commodities. The market is interdependent, meaning that each industry requires input from the other industries and possibly even its own commodity. In addition, there is an outside demand for each commodity that has to be satisfied. We wish to determine the amount of output of each industry which will satisfy all demands exactly; that is, both the demands of the other industries and the outside demand.

Let a_{ij} indicate the amount of commodity i, $i = 1, 2, 3$, necessary to produce one unit of commodity of commodity j, $j = 1, 2, 3$, in a market without external demand. Actually, it is more convenient to work in scaled monetary terms; thus we will assume that a_{ij} tells us the cost of the commodity i necessary to produce one unit profit of commodity j.

For example, to produce an amount of commodity j worth 100 dkk, one needs an amount of commodity i worth 30 dkk (and some other input from other commodities). Or equivalently, after scaling, to produce an amount of commodity j worth 1 dkk, one needs an amount of commodity i worth 0.3 dkk (and some other input from other commodities). In other terms, to produce one unit worth of commodity j one needs $a_{1j} + a_{2j} + a_{3j}$.

It follows that, in a market without external demand, the production of each commodity j is not profitable unless

$$\sum_{i=1}^{3} a_{ij} < 1 \qquad \forall j = 1, 2, 3. \tag{1}$$

Hence, we will assume that $a_{ij} \geq 0$ and (1) holds.

In our case, however, we do have also an external demand d_i for each commodity i, $i = 1, 2, 3$, which we consider also expressed in units of currency.

Our goal then becomes to determine the amount expressed in units of currency that we need toi invest in each commodity in order to satisfy the flow of money in the market.

(a) Using the mathematical symbols a_{ij} and d_i write a model of the problem.

Solution:

For each commodity, the outside demand is covered by the production of the commodity after the subtraction of the amount of commodity that has to go in the other industries and the amount that has to go in the same industry.

Hence:

$$x_i - \sum_{j=1}^{n} a_{ij} x_j = d_i$$

For $n = 3$ we have:

$$\begin{aligned} x_1 - a_{11}x_1 - a_{12}x_2 - a_{13}x_3 &= d_1 \\ x_2 - a_{21}x_1 - a_{22}x_2 - a_{23}x_3 &= d_2 \\ x_3 - a_{31}x_1 - a_{32}x_2 - a_{33}x_3 &= d_3 \end{aligned}$$

In matrix terms:

$$I\mathbf{x} - A\mathbf{x} = \mathbf{d}$$

or

$$(I - A)\mathbf{x} = \mathbf{d}$$

which is a system of linear equations. To make sense the solution $\mathbf{x}$ must be non-negative. It can be shown (see eg, Leon p. 373) that under the conditions that all terms are positive and (1) expressed above the solution to the system is unique and non-negative.

(b) Write a very short Python procedure using the module numpy to generate the coefficients a_{ij} for the case $n = 3$, ensuring that the conditions $a_{ij} \geq 0$ and (1) hold. Use the random number generator of numpy with the random seed set to the first 6 digits of your CPR number and set print precision to 3 digits. Here is some starting code:

```
#!/usr/bin/python
import numpy as np

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

def generate_data():
   '''
   return the matrix A
   Example:
   >>> A = generate_data()
   '''
   pass;


d = np.random.randint(10000,100000,3)
A = generate_data()

print A
print d
```

Solution:

```
#!/usr/bin/python
import numpy as np

np.random.seed(0000000001)
np.set_printoptions(precision=3)

def generate_data():
   '''
   return the matrix A
   Example:
   >>> A = generate_data()
   '''
   np.set_printoptions(precision=3)
   for i in range(1000):
       A=np.random.rand(3,3)
       s=np.sum(A,axis=1)
       if (s<1).all(): break
   return A

d = np.random.randint(10000,100000,3)
A = generate_data()

print A
print d
```

```
[[ 0.017  0.399  0.381]
 [ 0.659  0.071  0.153]
 [ 0.017  0.114  0.652]]
[87708 15192 60057]
```

(c) Using the numerical values for a_{ij} and d_i determined at the previous point (those printed) find the amounts in monetary terms to be invested in each of the three commodities in order to satisfy all demands exactly. Report the code you use and the numerical values you find.

Solution:

```
I=np.identity(3)
print np.linalg.solve(I-A,d)
```

```
[ 313368.111  284808.138  280466.611]
```

Exercise 7

A *field* is an algebraic structure with notions of addition, subtraction, multiplication and division and satisfying a number of properties. Any field may be used to give the scalars for a vector space. In class, we have worked with the field of Real numbers. There are other fields of interest, such as the field of complex numbers. In this task we work with the *Galois Field 2*, $GF(2)$: it has just two elements: 0 and 1. The operation of addition is like exclusive-or and the multiplication as an ordinary multiplication:

$+$	0	1
0	0	1
1	1	0

$\times$	0	1
0	0	0
1	0	1

Usual algebraic laws, like the distributive property, still hold.

Vectors over $GF(2)$ can be defined in a similar way as we have seen for vectors over the real numbers. The inner product between two vectors in $GF(2)$ is defined in the usual way. For example, for two vectors $\mathbf{x} = [1,1,0,1,0,1]$ and $\mathbf{y} = [0,1,1,0,0,1]$:

$$\begin{aligned}\langle \mathbf{x}, \mathbf{y} \rangle &= x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \\ &= 1 \times 0 + 1 \times 1 + 0 \times 1 + 1 \times 0 + 0 \times 0 + 1 \times 1 \\ &= 0 + 1 + 0 + 0 + 0 + 1 = 0\end{aligned}$$

where we have used the operations over $GF(2)$ as defined above. The inner product over $GF(2)$ satisfies a few properties such as

- $\mathbf{v} \cdot \mathbf{u} = \mathbf{u}\mathbf{v}$
- $(\alpha \mathbf{v}) \cdot \mathbf{u} = \alpha(\mathbf{u} \cdot \mathbf{v})$
- $(\mathbf{v}_1 + \mathbf{v}_2)\mathbf{u} = \mathbf{v}_1\mathbf{u} + \mathbf{v}_2\mathbf{u}$

Let's consider an application of $GF(2)$ on a simple authentication scheme. A common way to log on remotely to a computer is by means of a password. An user types in the password via a connection and the machine checks whether the password we are using when logging onto the system matches with one of those stored. This operation is a risky operation because if an eavesdropper, say Eve, intercepts the password then she can have easy access to the system. An alternative method is not to type in the password to log on but to answer a number of questions by the system that can be answered correctly only if the user knows the password.

This method is the *challenge-response scheme* and is based on the scalar product of vectors in $GF(2)$. Let the password be an n vector $\hat{\mathbf{x}}$. The machine sends the ith challenge, that is, a random n vector $\mathbf{a}_i$. The user sends back the answer, a scalar β_i calculated from $\langle \mathbf{a}_i, \hat{\mathbf{x}} \rangle$. This is repeated several times until the machine asserts that the user knows the password. This method is also not very secure, however, as there are a number of ways Eve could break it if she knows a bit of linear algebra.

(a) One way Eve can cheat is by discovering the password. Indeed, if she can observe the challenges $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ and the responses: $\beta_1, \beta_2 \ldots, \beta_m$ she can figure out the password. How can she do it? How many possible passwords can there be? What are the conditions to have only a unique possibility such that in this way the Eavesdropper can be sure to have found the password?

Solution:

If Eve knows the answer to the queries $\mathbf{a}_1, ..., \mathbf{a}_m$ then she can calculate the password by solving the system of linear equations:

$$\begin{aligned}\mathbf{a}_1\hat{\mathbf{x}} &= \beta_1 \\ &\vdots \\ \mathbf{a}_m\hat{\mathbf{x}} &= \beta_m\end{aligned}$$

or

$$A\mathbf{x} = \mathbf{b}$$

If $rank(A) = rank(A|b) = r$ then there is one unique solution if A is full rank, ie, if $r = m$. If $r < m$ then there are infinite solutions.

Exercise 8

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.

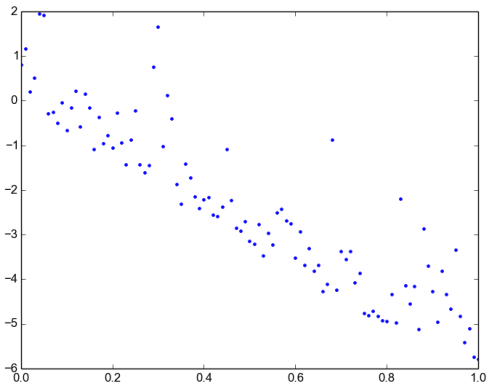

Figure 1: An example of points generated by the Python script plotted on the plane xy.

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.

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)
b_2 = sum(x*y)


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

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

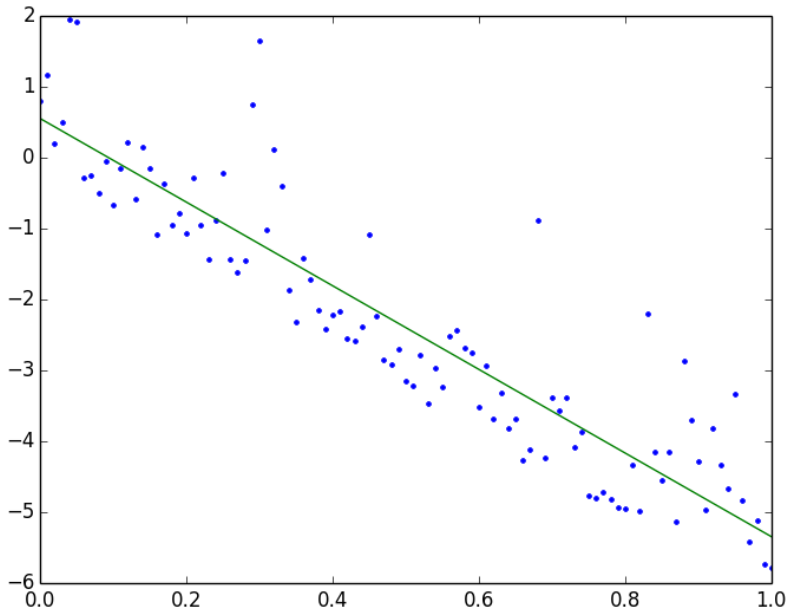

Figure 2:

```
# 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:

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

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 $[a, b]$. Hence, the solution to this systems gives the values of a and b 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()
```

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

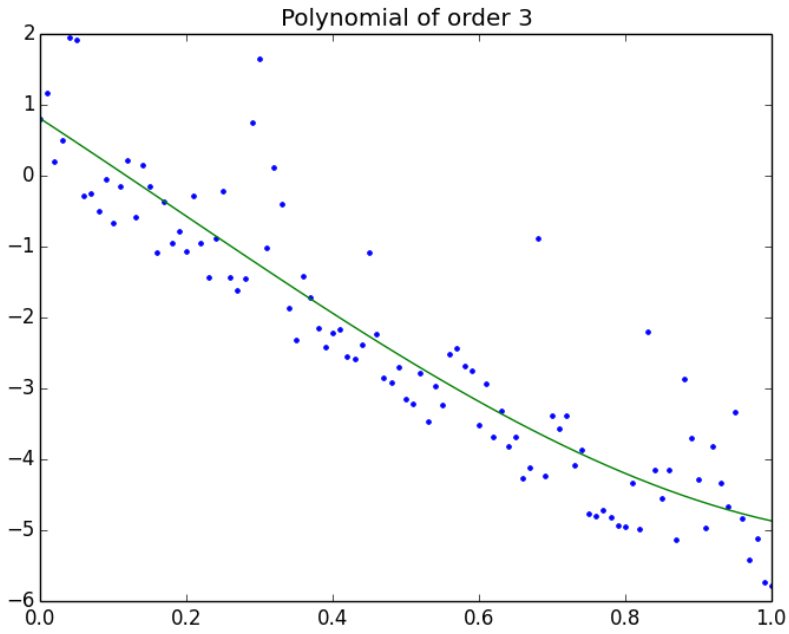

Figure 3:

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:

$$\begin{bmatrix} 1 & x_1^1 & \cdots & x_n^1 \\ 1 & x_1^2 & \cdots & x_n^2 \\ \vdots & \vdots & & \\ 1 & x_1^m & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} y^1 \\ y^2 \\ \cdots \\ y^m \end{bmatrix}$$

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:

```
# Task 4

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

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

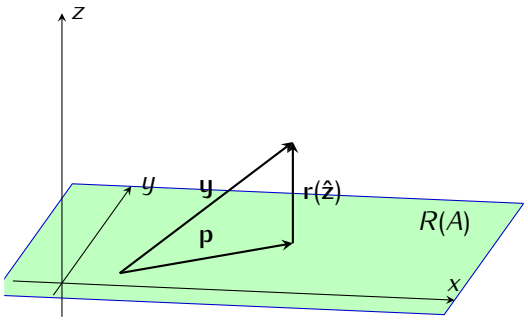

Figure 4: $\mathbf{y} \in \mathbb{R}^2$ and A is a 3×2 matrix of rank 2.

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 $\mathbf{b}$ 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 systems 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()
```