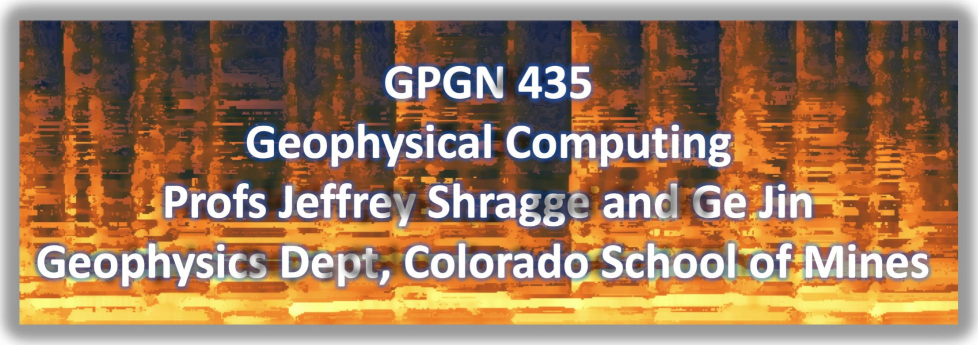
05_Regression

September 30, 2019

[1]:
```
from IPython.display import HTML

HTML('''<script>
code_show=true;
function code_toggle() {
 if (code_show){
 $('div.input').hide();
 } else {
 $('div.input').show();
 }
 code_show = !code_show
}
$( document ).ready(code_toggle);
</script>
The raw code for this Jupyter notebook is by default hidden for easier reading.
To toggle on/off the raw code, click <a href="javascript:code_toggle()">here</
↪a>.''')
```

[1]:
```
<IPython.core.display.HTML object>
```


GPGN 435
Geophysical Computing
Profs Jeffrey Shragge and Ge Jin
Geophysics Dept, Colorado School of Mines

title

1 Regression

The linear problems we have learned so far are all balanced or square system, in which the number of equations equals the number of unknowns. However, in the real cases, overdetermined linear systems are much more common: we almost always take more measurements than the unknowns in order to reduce the effect of noises. How can we solve a overdetermined system?

1.1 Linear Least Square Regression

Consider an overdetermined linear system $Ax = b$, where A is a $n \times m$ rectangular matrix, where $n > m$. We also require A to be full rank. Due to the error in the measurement b, in most cases there are no x existing that fully satisfies all the equations in the system.

To solve the problem, we need to find a x that minimize the error between the model prediction (Ax) and measurement (b), which can be presented by a column vector as:

$$e = Ax - b$$

The most common way to solve x is to minimize L-2 norm of the error vector e, which is defined as:

$$E = \sum_{i=1}^{m} e_i^2 = e^T e$$

where e^T is the transpose of e.

By substituting $e = Ax - b$, we have

$$E = e^T e = (Ax - b)^T (Ax - b) = x^T A^T Ax - b^T Ax - x^T A^T b + b^T b = x^T A^T Ax - 2x^T A^T b + b^T b$$

To find x that minimizes E, we set the derivatives of E with respect to x to zero:

$$\frac{\partial E}{\partial x} = -2A^T b + 2A^T Ax = 0$$

which gives us one of the **normal equations**:

$$A^T Ax = A^T b A^\dagger x = b^\dagger$$

This brings us back the problem we have learned in the last section, where $A^\dagger = A^T A$ is a full-rank square matrix, $b^\dagger = A^T b$ is a column vector.

1.1.1 Exercise

By knowing the following matrix differentiation rules, please prove the normal equations.

$$\alpha = Ax \implies \frac{\partial \alpha}{\partial x} = A$$

$$\alpha = x^T A \implies \frac{\partial \alpha}{\partial x} = A^T$$

$$\alpha = x^T Ax \implies \frac{\partial \alpha}{\partial x} = x^T \left(A + A^T\right)$$

The proof of the matrix differentiation rules can be found here.

1.2 1-D Linear Regression: Fitting a Straight Line

Two variables α and β that we know are linearly related, which can be presented as

$$\beta = c_1\alpha + c_0$$

where a and b are the unknown constant to be determined. By taking a series of measurements, we have n data points $(\alpha_1, \beta_1), (\alpha_2, \beta_2), ..., (\alpha_n, \beta_n)$. The goal is to find the optimized a and b such that the error

$$\varepsilon^2 = \sum_{i=1}^{n} (c_1\alpha_i + c_0 - \beta_i)^2$$

is minimized.

This problem is equivalent to the least-square problem we just discussed, if we rewrite the problem in the matrix form as:

$$e = \begin{bmatrix} \alpha_1 & 1 \\ \alpha_2 & 1 \\ \vdots & \vdots \\ \alpha_n & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_0 \end{bmatrix} - \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

and by minimizing $e^T e$ we have

$$A^T A x = A^T b$$

where

$$A = \begin{bmatrix} \alpha_1 & 1 \\ \alpha_2 & 1 \\ \vdots & \vdots \\ \alpha_n & 1 \end{bmatrix}, \quad x = \begin{bmatrix} c_1 \\ c_0 \end{bmatrix}, \quad b = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

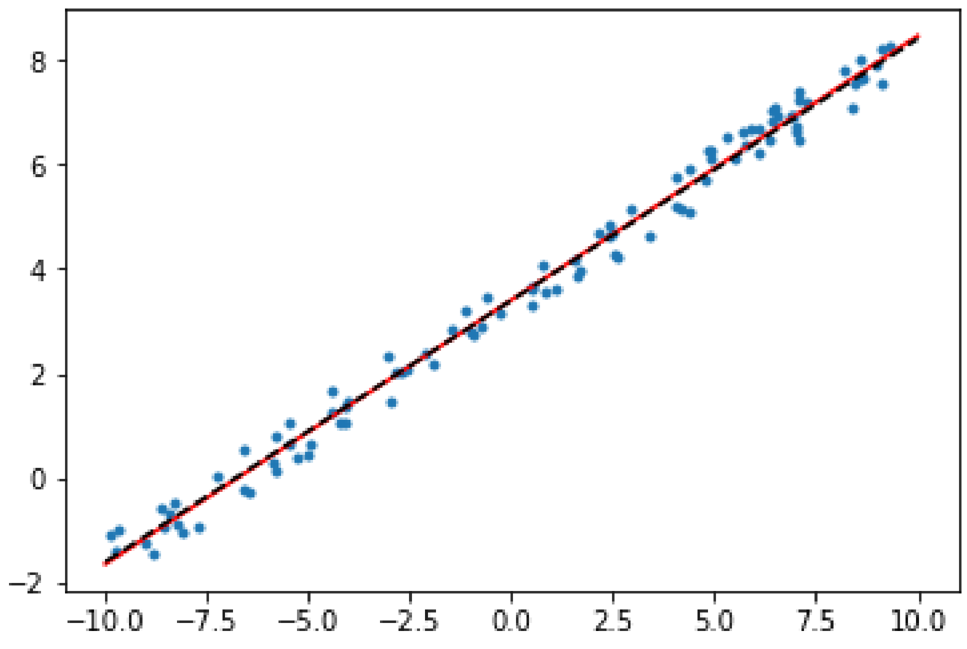
c_0 and c_1 can be solved by

$$\begin{bmatrix} c_1 \\ c_0 \end{bmatrix} = \left(A^T A\right)^{-1} A^T b$$

[49]:
```python
# python program to perform a 1-D linear inversion
import numpy as np
import matplotlib.pyplot as plt

# defining the true relation and noise level
c1 = 0.5; c0 = 3.4;
noise_level = 1
true_fun = lambda x: c1*x+c0

# generate the data
x = (np.random.rand(100)-0.5)*20
y = true_fun(x)
y += noise_level*(np.random.rand(100)-0.5) # add noise to data

# perform least-square fitting
A = np.hstack((x.reshape(-1,1),np.ones(len(x)).reshape(-1,1)))
A_dagger = A.T.dot(A)
para = np.linalg.inv(A_dagger).dot(A.T.dot(y))
```

```
# print out the result
print('True answer: c_1:{},c_0:{}'.format(c1,c0))
print('Inverted answer: c_1:{:.2f},c_0:{:.2f}'.format(para[0],para[1]))


predict_x = np.array([-10,10])
predict_y = para[0]*predict_x+para[1]

plt.figure()
plt.plot(x,y,'.')
plt.plot(predict_x,predict_y,'r')
plt.plot(predict_x,true_fun(predict_x),'k--')
plt.show()
```

```
True answer: c_1:0.5,c_0:3.4
Inverted answer: c_1:0.50,c_0:3.41
```



1.3 1-D Polynomial Curve Fitting

It makes sense that we can use linear least square method to solve a linear function problem, but what if the variables α and β are not related linearly, but through a m^{th}-order polynomial function?

$$\beta = c_m\alpha^m + c_{m-1}\alpha^{m-1} + ... + c_1\alpha + c_0$$

It turns out that we can use the exact same setting as the linear equation, by forming matrix and vectors:

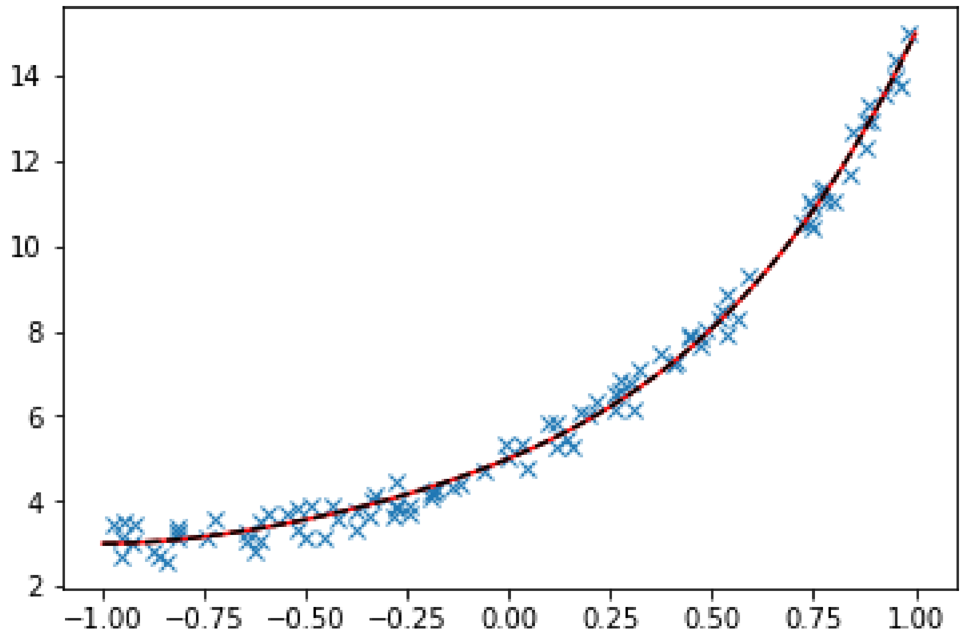
$$A = \begin{bmatrix} \alpha_1^m & \alpha_1^{m-1} & ... & \alpha_1 & 1 \\ \alpha_2^m & \alpha_2^{m-1} & ... & \alpha_2 & 1 \\ \vdots & \vdots & ... & \vdots & \vdots \\ \alpha_n^m & \alpha_n^{m-1} & ... & \alpha_n & 1 \end{bmatrix}, \quad x = \begin{bmatrix} c_m \\ c_{m-1} \\ \vdots \\ c_0 \end{bmatrix}, \quad b = \begin{bmatrix} \beta_m \\ \beta_{m-1} \\ \vdots \\ \beta_0 \end{bmatrix},$$

same as above, the coefficients $c_0, c_1, ..., c_m$ can be solved by:

$$\begin{bmatrix} c_m \\ c_{m-1} \\ ... \\ c_0 \end{bmatrix} = \left(A^T A\right)^{-1} A^T b$$

[50]:
```
# python program to perform a 1-D linear inversion
import numpy as np
import matplotlib.pyplot as plt

# defining the true relation and noise level
c = [1,2,3,4,5]
noise_level = 1
true_fun = lambda x: np.polyval(c,x)

# generate the data
x = (np.random.rand(100)-0.5)*2
y = true_fun(x)
y += noise_level*(np.random.rand(100)-0.5) # add noise to data

# perform least-square fitting
A = np.ones((len(x))).reshape(-1,1)
for i in range(1,len(c)):
    A = np.hstack((x.reshape(-1,1)**i,A))
A_dagger = A.T.dot(A)
para = np.linalg.inv(A_dagger).dot(A.T.dot(y))

# print out the result
print('True answer: ',c)
print('Inverted answer: ',para)

predict_x = np.linspace(-1,1,100)
predict_y = np.polyval(c,predict_x)

plt.figure()
plt.plot(x,y,'x')
plt.plot(predict_x,predict_y,'r')
plt.plot(predict_x,true_fun(predict_x),'k--')
plt.show()
```

```
True answer:  [1, 2, 3, 4, 5]
Inverted answer:  [1.21493346 1.54076732 2.86591468 4.31234803 5.00514973]
```



1.4 2-D Polynomial Surface Fitting

The same idea applies to higher dimension problems. Say variable γ is depended on variables α and β through a 2-D 2nd-order polynomial function

$$\gamma = c_0 + c_1\alpha + c_2\alpha^2 + c_3\alpha\beta + c_4\beta + c_5\beta^2$$

we can again form the linear equation set Ax with

$$A = \begin{bmatrix} \beta_1^2 & \beta_1 & \alpha_1\beta_1 & \alpha_1^2 & \alpha_1 & 1 \\ \beta_2^2 & \beta_2 & \alpha_2\beta_2 & \alpha_2^2 & \alpha_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \beta_n^2 & \beta_n & \alpha_n\beta_n & \alpha_n^2 & \alpha_n & 1 \end{bmatrix}, \quad x = \begin{bmatrix} c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}$$

and solve it using least square method.

1.5 Weighted Least Square Method

One of the problem for least square inversion is that it is very sensitive to outliers or measurements with large error, due to the nature of L_2 norm.

One way to reduce the impact of bad data is to give them less weight in the error function. Because the actual problem we solve for least square problems is to minimize $||Ax - b||_2$, the elementary operations of linear equation set do not apply any more (except for the interchange rule). By multiplying one row of A and b at the same time, we change the ratio of the prediction error of the measurement in the total error. This gives us the opportunity to apply different weights to the measurements with different accuracy.

Multiple each row with a different weight w_i, the error e can be written as

$$e = WAx - Wb = A'x - b'$$

where $A' = WA$, $b' = Wb$, and W is a diagonal matrix with $W_{ii} = w_i$.

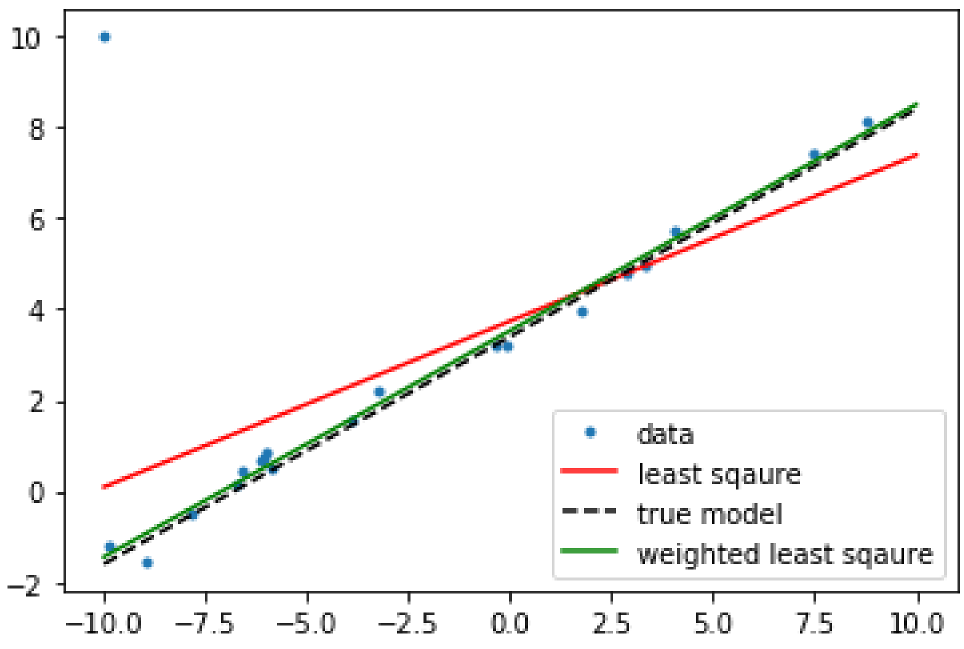
Replacing A' and b' in the normal equations we have

$$(WA)^T(WA)x = (WA)^T WbA^T W^T WAx = A^T W^T WbA^T W' Ax = A^T W' b$$

where $W' = W^T W$, which is also a diagonal matrix with $W'_{ii} = w_i^2$.

[27]:
```python
# demonstration of effect of outlier to least sqaure inversion
import numpy as np
import matplotlib.pyplot as plt

# defining the true relation and noise level
c1 = 0.5; c0 = 3.4;
noise_level = 1
true_fun = lambda x: c1*x+c0

# generate the data
data_number = 20
x = (np.random.rand(data_number)-0.5)*20
y = true_fun(x)
y += noise_level*(np.random.rand(data_number)-0.5) # add noise to data

# add an outlier
y[10] = 10
x[10] = -10

# perform normal least-square fitting
A = np.hstack((x.reshape(-1,1),np.ones(len(x)).reshape(-1,1)))
A_dagger = A.T.dot(A)
para = np.linalg.inv(A_dagger).dot(A.T.dot(y))

# perform weighted least-square fitting
w = np.ones(data_number)
w[10] = 0.1 # change the weight of the outlier
W = np.diag(w**2)
A_dagger = A.T.dot(W).dot(A)
para_w = np.linalg.inv(A_dagger).dot(A.T.dot(W).dot(y))

# print out the result
print('True answer: c_1:{},c_0:{}'.format(c1,c0))
print('Least Square answer: c_1:{:.2f},c_0:{:.2f}'.format(para[0],para[1]))
print('Weighted Least Square answer: c_1:{:.2f},c_0:{:.2f}'.
 ↪format(para_w[0],para_w[1]))

predict_x = np.array([-10,10])
predict_y = para[0]*predict_x+para[1]
predict_y_w = para_w[0]*predict_x+para_w[1]

plt.figure()
plt.plot(x,y,'.',label='data')
plt.plot(predict_x,predict_y,'r',label = 'least sqaure')
plt.plot(predict_x,true_fun(predict_x),'k--', label='true model')
plt.plot(predict_x,predict_y_w,'g',label = 'weighted least sqaure')
plt.legend()
plt.show()
```

```
True answer: c_1:0.5,c_0:3.4
Least Square answer: c_1:0.36,c_0:3.74
Weighted Least Square answer: c_1:0.50,c_0:3.53
```



1.5.1 Weight Selection

Ideally the weight w_i of each measurement should be its standard deviation. For example, if the measurements are carried out by two different instruments, with one being less than accurate the other, we should weight the measurements differently based on their instrument precision (standard deviation).

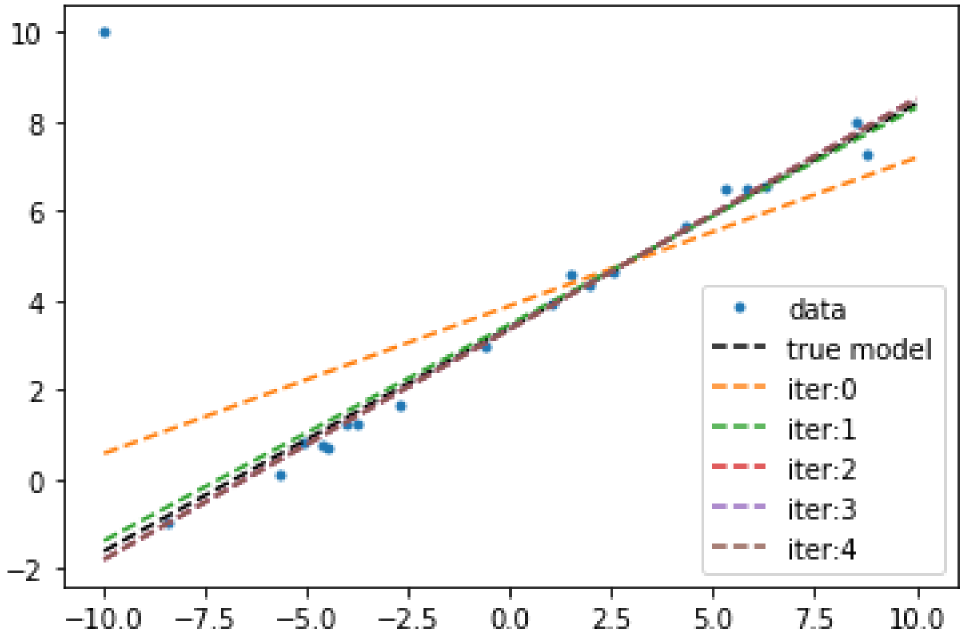
In the real situation many outliers are due to data error or instrument instability, of which the standard deviation is difficult to estimate. In this case we can solve the problem iteratively:

1. Perform a regular least-square inversion.
2. Calculate the error of each measurement.
3. Weight each measurement (row) based on the prediction error.
4. Perform weighted least-square inversion.

Step 2-4 should be repeated several times until the inversion result is stabilized.

[61]:
```python
# demonstration of effect of outlier to least sqaure inversion
import numpy as np
import matplotlib.pyplot as plt

# defining the true relation and noise level
c1 = 0.5; c0 = 3.4;
noise_level = 1
true_fun = lambda x: c1*x+c0

# generate the data
data_number = 20
x = (np.random.rand(data_number)-0.5)*20
y = true_fun(x)
y += noise_level*(np.random.rand(data_number)-0.5) # add noise to data

# add an outlier
y[10] = 10
x[10] = -10

plt.figure()
plt.plot(x,y,'.',label='data')
predict_x = np.array([-10,10])
plt.plot(predict_x,true_fun(predict_x),'k--', label='true model')
# perform iterative weighted least-square fitting
w = np.ones(data_number)
A = np.hstack((x.reshape(-1,1),np.ones(len(x)).reshape(-1,1)))
for iter_i in range(5):
    W = np.diag(w**2)
    A_dagger = A.T.dot(W).dot(A)
    para_w = np.linalg.inv(A_dagger).dot(A.T.dot(W).dot(y))
    y_pre = para_w[0]*x+para_w[1]
    err = np.abs(y_pre-y)
    w = 1/err**0.5
    predict_y_w = para_w[0]*predict_x+para_w[1]
    plt.plot(predict_x,predict_y_w,'--', label='iter:{}'.format(iter_i))

# print out the result
print('True answer: c_1:{},c_0:{}'.format(c1,c0))
print('Weighted Least Square answer: c_1:{:.2f},c_0:{:.2f}'.
 ↪format(para_w[0],para_w[1]))


plt.legend()
plt.show()
```

```
True answer: c_1:0.5,c_0:3.4
Weighted Least Square answer: c_1:0.51,c_0:3.36
```



1.6 Least Square Regularization

Although for least square problem $Ax = b$, the number of equations is usually much larger than the number of unknowns (the number of rows >> the number of columns for A), this does not guarantee the matrix $A^{\dagger} = A^T A$ is always full rank.

A quick demonstration shows as follow:

[24]:
```
A = np.
 ↪array([[1,2,3,1,1],[3,2,1,0,0],[1,4,6,5,5],[3,2,6,0,0],[2,65,2,2,2],[3,2,6,3,3],[6,3,6,2,2]
print('A=')
print(A)
# calculate A dagger
Ad = A.T.dot(A)
print('ATA=')
print(Ad)
# calculate A dagger inversion
```

```
print('ATA^-1=')
print(np.linalg.inv(Ad))

# true answer
x =np.array([1,2,3,4,5]).reshape(-1,1)
# calculate synthetic data
b = A.dot(x)
# perform inversion
x_inv = np.linalg.inv(Ad).dot(A.T).dot(b)
print('true x:',x.flatten())
print('inverted x:',x_inv.flatten())
```

```
A=
[[ 1  2  3  1  1]
 [ 3  2  1  0  0]
 [ 1  4  6  5  5]
 [ 3  2  6  0  0]
 [ 2 65  2  2  2]
 [ 3  2  6  3  3]
 [ 6  3  6  2  2]]
ATA=
[[  69  172   88   31   31]
 [ 172 4266  204  164  164]
 [  88  204  158   67   67]
 [  31  164   67   43   43]
 [  31  164   67   43   43]]
ATA^-1=
[[ 6.71188887e-02 -1.71831039e-03 -5.13577956e-02 -1.29655198e-01
   1.29655198e-01]
 [-1.71831039e-03  3.22467011e-04  1.58253481e-03  4.58013507e-04
  -4.58013507e-04]
 [-5.13577956e-02  1.58253481e-03  5.82102621e-02  2.86608255e-02
  -2.86608255e-02]
 [-2.53932853e-19  3.06662239e-03  9.81319164e-02  8.83893724e+14
  -8.83893724e+14]
 [ 3.81881316e-02 -5.52352471e-03 -1.57841954e-01 -8.83893724e+14
   8.83893724e+14]]
true x: [1 2 3 4 5]
inverted x: [ -35.16416063    4.3266865    59.54540541  183.16577019
-242.63160552]
```

The reason of large error in the inversion result is that the last two columns of A are identical, which makes $A^{\dagger}$ not full rank. The matrix inversion of $A^{\dagger}$ is performed using iteration method, which generates large numerical error in the inversion. In order to resolve this kind problem, we can regulate the model (unknowns) using our knowledge of the problem. This knowledge does not have to be accurate, but serve as important constrains to stabilize the inversion.

A commonly used regularization is to minimize the L_2 norm of all the unknowns. We can rewrite the normal equation as:

$$(A^T A + \alpha I)x = A^T b$$

where α is the weight of the regularization, which is a constant to be chosen. This type of regulation is called Tikhonov regularization.

[31]:
```
# continue from the last code cell
alpha = .01
x_regu = np.linalg.inv(A.T.dot(A)+alpha*np.identity(A.shape[1])).dot(A.T.dot(b))
print('True answer: ',x.flatten())
print('Inverted answer: ',x_regu.flatten())
```

```
True answer:  [1 2 3 4 5]
Inverted answer:  [0.99918729 2.00007375 3.00142044 4.49852262 4.49852262]
```

Now the first three unknowns can be correctly inverted. We still cannot invert the last two unknowns correctly, because there is no information provided from the data to distinguish the last two unknowns.

Besides Tikhonov regularization, we can also put other kinds constraint based on our knowledge on the model. In general, any constraint can be written in a form as:

$$\Gamma x = \gamma$$

can be put into the normal equation as:

$$(A^T A + \alpha \Gamma^T \Gamma)x = A^T b + \alpha \Gamma^T \gamma$$

This is equivalent to solving two least square problems $Ax = b$ and $\Gamma x = \gamma$ at the same time with different weight. Actually, we can concatenate Γ and γ to the end of matrix A and vector b to perform the regularization.

$$A^* = \begin{bmatrix} A \\ \alpha\Gamma \end{bmatrix}, \quad b^* = \begin{bmatrix} b \\ \alpha\gamma \end{bmatrix} \; A^{*T} A^* x = A^{*T} b^*$$

In the previous example, if we have the knowledge that the difference between the nearby unknown is 1, we can put this knowledge as regularization to the problem.

[43]:
```
# continue from the last code cell
alpha = 0.01

# form regularization matrix
N = A.shape[1]
Gamma = np.zeros((N-1,N))
for i in range(N-1):
    Gamma[i,i] = 1
    Gamma[i,i+1] = -1
g = -np.ones(N-1).reshape(-1,1)

# Concatenate regularization to A and b
As = np.vstack([A,alpha*Gamma])
bs = np.vstack([b,alpha*g])
```

```
x_regu = np.linalg.inv(As.T.dot(As)).dot(As.T.dot(bs))
print('True answer: ',x.flatten())
print('Inverted answer: ',x_regu.flatten())
```

```
True answer:  [1 2 3 4 5]
Inverted answer:  [1. 2. 3. 4. 5.]
```

2 Acknowledgement

Most of this teaching material is based on:
Kreyszig, E., 2018. Advanced Engineering Mathematics, 10-th edition.

[ ]:
```

```