# <u>CE/CZ 1104 Linear Algebra for Computing</u>

<u>Lab 1</u>

**<u>Instructions:</u>** There are 3 exercises in this lab with questions for each exercise. All questions require answers to be in the form of outputs to your Python script running in Jupyter notebook. Convert the notebook to pdf format (File -> Print Preview) and upload at Content -> Part 1 -> Lab -> Lab 1 -> your lab group folder.

## <u>Exercise 1: Computer Security - Simple authentication scheme [1]</u>

An authentication scheme allows a human to log onto a computer over an insecure network. The most familiar scheme is based on *passwords* where Harry, the human, sends his password to Carole, the computer, and the computer verifies that it is the correct password. However, if Eve, the eavesdropper, can read the bits going over the network, she can learn the password by observing just one log-in.

A more secure scheme is a *challenge-response* scheme. In a series of trials, Carole repeatedly asks Harry questions that someone not possessing the password would be unlikely to answer correctly. If Harry answers the questions correctly, Carole concludes that Harry knows the password. Let us look at an example.

Suppose the password is an $n$-bit string, i.e., a vector $\mathbf{x}$ with $n$ entries, where the entries are binary (zeros and ones), and chosen uniformly at random. More formally, the vector is said to be defined over *GF(2)*, which is short for *Galois Field 2*. The field *GF(2)* has only two elements, 0 and 1. Addition in *GF(2)* is modulo 2, i.e., equivalent to exclusive-OR. Multiplication in *GF(2)* is just like ordinary multiplication. In this exercise, all data is assumed to be in *GF(2)*.

In the $i$th trial, Carole selects a nonzero vector $\mathbf{c}_i$, a *challenge vector*, and sends it to Harry. Harry is required to send back a single bit $\beta_i$, which is supposed to be the dot product of $\mathbf{c}_i$ and the password $\mathbf{x}$. Carole then checks whether $\beta_i = \mathbf{c}_i \cdot \mathbf{x}$. If Harry passes enough trials, Carole concludes that Harry knows the password, and allows him to log in.

<u>Question 1:</u>

Suppose the password $\mathbf{x}$ is 10111. Harry initiates log-in. What is Harry's response to Carole's challenge vectors $\mathbf{c}_1$ = 01011 and $\mathbf{c}_2$ = 11110.

[NOTE: In Python, the dot product of two vectors $\mathbf{v}_1$ = 1101 and $\mathbf{v}_2$ = 1111 will return 3. However *GF(2)* has only the elements 0 and 1. To ensure that the answer is in *GF(2)*, you should compute 3 mod 2.]

<u>Question 2:</u>

Enter Eve! Suppose Eve had observed Harry's response ( $\beta_1$ *and* $\beta_2$) and the first two challenge vectors ($\mathbf{c}_1$and $\mathbf{c}_2$). Subsequently, she tries to login as Harry and Carole happens to send her as a challenge vector the sum of $\mathbf{c}_1$ = 01011 and $\mathbf{c}_2$ = 11110. Even though Eve does not know the password, she can use the distributive property to compute the dot product of this sum with the password $\mathbf{x}$:

$$(01011 + 11110) \cdot \mathbf{x} = 01011 \cdot \mathbf{x} + 11110 \ \cdot \mathbf{x}$$

Find the response to this challenge vector without using **x**. Next, since you know the password, verify that this is indeed the correct response to the challenge vector by adding the terms in the bracket and taking the dot product with **x**.

Question 3:

Extending the above idea, Eve can compute the right response to the sum of any number of previous challenges for which she has the right response. Mathematically,

$$\text{if } \mathbf{c}_1 \cdot \mathbf{x} = \beta_1, \mathbf{c}_2 \cdot \mathbf{x} = \beta_2, \cdots, \mathbf{c}_k \cdot \mathbf{x} = \beta_1, \quad then \quad (\mathbf{c}_1 + \mathbf{c}_2 + \cdots + \mathbf{c}_k) \cdot \mathbf{x} = (\beta_1 + \beta_2 + \cdots + \beta_k)$$

Assume Eve knows the following challenges and responses:

| Challenge | Response |
|---|---|
| 110011 | 0 |
| 101010 | 0 |
| 111011 | 1 |
| 001100 | 1 |

Show how she can derive the right response to two new challenges $c_a$ = 011001 and $c_b$ = 110111. You can consider your Python script to be a function whose inputs are Ch and c, where Ch is the matrix whose rows are the challenges that she already knows (from the table) and c is either $c_a$ or $c_b$. The output of the function will be the response (to $c_a$ or $c_b$).

Question 4:

Suppose Eve eavesdrops on communication, and learns $m$ pairs $(\mathbf{c}_1, \beta_1), \cdots, (\mathbf{c}_m, \beta_m)$ such that $\beta_i$ is the correct response to challenge $\mathbf{c}_i$. Then the password **x** is a solution to

$$\begin{bmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_m \end{bmatrix} [\mathbf{x}] = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} => C\mathbf{x} = \mathbf{b} \qquad (1)$$

What is the condition on the vectors $\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_m$ in eq. (1) to have a solution? In addition to the data in the table in Question 3, she observes the following two responses also:

| Challenge | Response |
|---|---|
| 011011 | 0 |
| 110100 | 1 |

Solve eq. (1) to find the password **x**. Use scipy.linalg.solve (see here).

Optional Question 4a: You do not need to submit the answer to this question. However, this is a good exercise for practice. Write a python program to solve eq. (1) by Gaussian elimination.

## Exercise 2: Machine learning – Linear regression

Regression searches for relationships among variables [2].

For example, you can observe several employees of some company and try to understand how their salaries depend on the **features**, such as experience, level of education, role, city they work in, and so on.

This is a regression problem where data related to each employee represent one **observation**. The presumption is that the experience, education, role, and city are the independent features, while the salary depends on them.

Similarly, you can try to establish a mathematical dependence of the prices of houses on their areas, numbers of bedrooms, distances to the city center, and so on.

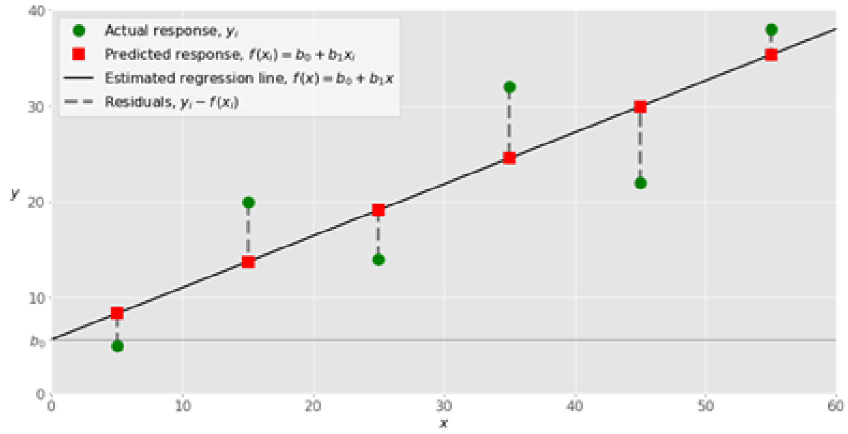
Let $x_i$ represent an independent feature, e.g., area of house and $y_i$ represent the dependent feature, e.g., price. Given a set of $n$ data points $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$, we determine a function $\hat{y} = f(x)$ that fits the data. The error in fitting data point $i$ at $(x_i, y_i)$ is the difference between $y_i$ and $f(x_i)$, i.e.

$$e_i = y_i - \hat{y}_i \,.$$

The error is also referred to as *residual*.

We determine the function $f(x)$ such that the sum of the squares of the errors $\varepsilon$ is minimized, where $\varepsilon = \sum_{i=1}^{n} e_i^2$ .

First, we consider a simple linear regression where a given set of $n$ data points (green dots) are fitted on a straight line, i.e. $\hat{y} = mx + c$ (see figure below). Here we determine the values of $m$ and $c$ such that $\varepsilon$ is minimized.



Substituting $e_i = y_i - \hat{y}_i$ and $\hat{y}_i = m x_i + c$ into $\varepsilon$, we get:

$$\begin{aligned}\varepsilon &= \sum_{i=1}^{n}(y_i - mx_i - c)^2 \\ &= \left(\sum_{i=1}^{n} y_i^2\right) + \left(m^2\sum_{i=1}^{n} x_i^2\right) - \left(2m\sum_{i=1}^{n} x_i y_i\right) - \left(2c\sum_{i=1}^{n} y_i\right) + \left(2mc\sum_{i=1}^{n} x_i\right) + \left(nc^2\right)\end{aligned}$$

To determine the values of $m$ and $c$ that minimize $\varepsilon$, we take the derivatives of $\varepsilon$ with respect to $m$ and $c$, and equate to zero to obtain a system of linear equations, i.e.

$$\begin{aligned}\frac{\partial\varepsilon}{\partial m} &= 2m\sum_{i=1}^{n} x_i^2 - 2\sum_{i=1}^{n} x_i y_i + 2c\sum_{i=1}^{n} x_i &= 0 \\ \frac{\partial\varepsilon}{\partial c} &= -2\sum_{i=1}^{n} y_i + 2m\sum_{i=1}^{n} x_i + 2nc &= 0\end{aligned}$$

The above set of linear equations can be written in matrix form as follows:

$$\begin{bmatrix}\sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i & n\end{bmatrix}\begin{bmatrix} m \\ c\end{bmatrix} = \begin{bmatrix}\sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} y_i\end{bmatrix} \qquad (2)$$

The above equation in the form of $A\mathbf{x} = \mathbf{b}$ can be solved for $m$ and $c$. Hence we obtain the function $\hat{y} = m\,x + c$.

Question 5:

Consider the following data regarding house prices:

| House | $x_1$ (area in 1000 sq ft) | $y$ (price in 1000 dollars) |
|---|---|---|
| 1 | 0.846 | 115.00 |
| 2 | 1.324 | 234.50 |
| 3 | 1.150 | 198.00 |
| 4 | 3.037 | 528.00 |
| 5 | 3.984 | 572.50 |

- What is the matrix $A$ and the vector **b** for the above data?
- **WITHOUT** using scipy.linalg.solve, solve eq. (1) for $m$ and $c$.
- Plot the data $(x_1, y)$ and the fitted line.

```
import matplotlib.pyplot as plt
xs = np.linspace(0,1,5)
ys = c + m*xs
plt.plot(xs,ys,'r',linewidth=4)  <- plots fitted line
plt.scatter(x,y); <- plots data(x1,y)
plt.show()
```

Question 6:

Suppose we have additional information on the houses in the form of number of bedrooms. The above table including this information is shown below:

| House | $x_1$ (area in 1000 sq ft) | $x_2$ (number of bedrooms) | $y$ (price in 1000 dollars) |
|---|---|---|---|
| 1 | 0.846 | 1 | 115.00 |
| 2 | 1.324 | 2 | 234.50 |
| 3 | 1.150 | 3 | 198.00 |
| 4 | 3.037 | 4 | 528.00 |
| 5 | 3.984 | 5 | 572.50 |

Now, it is no longer a line but a plane that should be fit in 3D space. The plane is of the form

$$y = a_0 + a_1 x_1 + a_2 x_2.$$

[Note: Since $x_2$ represents categorical (discrete) data, fitting a plane is not totally correct because no one would be interested in fractional number of bedrooms! However, here, we ignore this discrepancy.]

If we have *n* observations, each observation *i* can be written as

$$y_i = a_0 + a_1 x_{i1} + a_2 x_{i2}.$$

In least squares linear regression, we want to minimize the sum of squared errors

$$SSE = \sum_{\mathrm{i}} \left(\mathrm{y_i} - (\mathrm{a_0} + \mathrm{a_1} x_{i1} + a_2 x_{i2})\right)^2$$

which can be expressed as $SSE = \|\mathbf{y} - X\mathbf{a}\|^2$, where $\| \quad \|$ is the norm of a vector,

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}, X = \begin{bmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ 1 & x_{31} & x_{32} \\ 1 & x_{41} & x_{42} \\ 1 & x_{51} & x_{52} \end{bmatrix} \text{ and } \mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}.$$

It can be shown that the coefficients $\mathbf{a} = [a_0 \quad a_1 \quad a_2]^{\mathrm{T}}$ that minimize the sum of squared errors is the unique solution of the system

$$(X^T X)\mathbf{a} = X^T \mathbf{y}. \tag{3}$$

Derivation of eq. (3) will be discussed in Part 2 of the course.

- Use scipy.linalg.solve to determine the vector **a**.
- Plot the 3D data $(x_1, x_2, y)$ and the fitted plane. [See here for help on 3D plotting]
- Using the obtained vector **a**, predict the prices of each of the 5 houses.

## Exercise 3: Cryptography – Threshold secret sharing [1]

“Secret sharing refers to methods for distributing a secret amongst a group of participants, each of whom is allocated a share of the secret. The secret can be reconstructed only when a sufficient number, of possibly different types, of shares are combined together; individual shares are of no use on their own.”[3]. Secret sharing is used in applications such as secure multiparty computation, Bitcoin signatures, access control etc.

“In one type of secret sharing scheme there is one dealer and *n* players. The dealer gives a share of the secret to the players, but only when specific conditions are fulfilled will the players be able to reconstruct the secret from their shares. The dealer accomplishes this by giving each player a share in such a way that any group of *t* (for threshold) or more players can together reconstruct the secret but no group of fewer than *t* players can. Such a system is called a (*t*, *n*)-threshold scheme (sometimes it is written as an (*n*, *t*)-threshold scheme).” [3]. This scheme was invented independently by Shamir [4] and Blakely [5].

The mid-term exam is approaching! But there is a slight problem – the professor has to be away at a conference on the day of the exam. She has four teaching assistants (TAs) to help conduct the exam, but she does not want to provide each TA with the question papers because she does not trust them. What if one TA leaks the question paper ahead of time?! The question paper is password protected and the professor wants to split the password (secret) among the four TAs. She employs threshold secret sharing by which any 3 TAs could jointly recover the secret (it is risky to rely on all 4 TAs showing up for the exam) but any 2 TAs could not.

The professor uses *GF(2)* to define ten 6D vectors $\mathbf{a}_0$, $\mathbf{b}_0$, $\mathbf{a}_1$, $\mathbf{b}_1$, $\mathbf{a}_2$, $\mathbf{b}_2$, $\mathbf{a}_3$, $\mathbf{b}_3$, $\mathbf{a}_4$, $\mathbf{b}_4$, i.e., there are 6 entries in each vector and the entries are either 0 or 1. They can be considered to be forming 5 pairs: Pair 0 consists of $\mathbf{a}_0$ and $\mathbf{b}_0$, Pair 1 consists of $\mathbf{a}_1$ and $\mathbf{b}_1$, and so on. The requirement is that for any 3 pairs, the corresponding six vectors are linearly independent. These vectors are known to everyone.

Now, suppose the professor wants to share two secret bits *s* and *t*. She chooses a secret 6D vector $\mathbf{u}$ that is randomly generated such that $\mathbf{a}_0 \cdot \mathbf{u} = s$ and $\mathbf{b}_0 \cdot \mathbf{u} = t$. She then gives TA1 the two bits $\beta_1 = \mathbf{a}_1 \cdot \mathbf{u}$ and $\gamma_1 = \mathbf{b}_1 \cdot \mathbf{u}$. TA2 gets two bits $\beta_2 = \mathbf{a}_2 \cdot \mathbf{u}$ and $\gamma_2 = \mathbf{b}_2 \cdot \mathbf{u}$ and similarly for TA3 and TA4. Each TA’s share thus consists of a pair of bits.

Recoverability: How can 3 TAs get together to recover the secret bits *s* and *t* ? Suppose TA1, TA2 and TA3 come together. They can use their bits and the 6D vectors to solve the following equation (here $\mathbf{a}_i$’s and $\mathbf{b}_i$’s represent row vectors) for $\mathbf{u}$:

$$\begin{bmatrix} \mathbf{a}_1 \\ \mathbf{b}_1 \\ \mathbf{a}_2 \\ \mathbf{b}_2 \\ \mathbf{a}_3 \\ \mathbf{b}_3 \end{bmatrix} [\mathbf{u}] = \begin{bmatrix} \beta_1 \\ \gamma_1 \\ \beta_2 \\ \gamma_2 \\ \beta_3 \\ \gamma_3 \end{bmatrix}.$$

With the knowledge of **u**, the TAs can recover the secret bits. Since the vectors **a**$_i$ and **b**$_i$ are linearly independent, the matrix is invertible and hence, there is a unique solution to the equation.

<u>Secrecy:</u> Can 2 TAs recover the secret bits? Suppose TA1 and TA2 go rogue and try to recover *s* and *t*. The system of equations becomes (note that **a**$_i$'s and **b**$_i$'s are row vectors)

$$\begin{bmatrix} \mathbf{a}_0 \\ \mathbf{b}_0 \\ \mathbf{a}_1 \\ \mathbf{b}_1 \\ \mathbf{a}_2 \\ \mathbf{b}_2 \end{bmatrix} [\mathbf{u}] = \begin{bmatrix} \text{guess} - \text{s} \\ \text{guess} - \text{t} \\ \beta_1 \\ \gamma_1 \\ \beta_2 \\ \gamma_2 \end{bmatrix},$$

where the first two entries on the right hand side are guessed values of *s* and *t*. Since the vectors **a**$_0$, **b**$_0$, **a**$_1$, **b**$_1$, **a**$_2$, **b**$_2$, are linearly independent, the matrix is invertible and there is a unique solution, no matter what bit is chosen as guess-s and as guess-t. This shows that the shares of TA1 and TA2 tell them nothing about the true values of *s* and *t*.

<u>Question 7:</u>

Define $\mathbf{a}_0 = [1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1]$ and $\mathbf{b}_0 = [1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1]$.

- Write a function `random_vector(s, t)` which takes as inputs bits *s* and *t* and outputs a 6D random binary vector **u** (vector with 6 elements) such that $\mathbf{a}_0 \cdot \mathbf{u} = s$ and $\mathbf{b}_0 \cdot \mathbf{u} = t$. [code for random binary number generation available in NTULearn].
- The next goal is to select vectors **a**$_1$, **b**$_1$, **a**$_2$, **b**$_2$, **a**$_3$, **b**$_3$, **a**$_4$, **b**$_4$ so that the requirement – for any three pairs, the corresponding six vectors are linearly independent – is satisfied. [example code to determine if a set of vectors is linearly independent is available in NTULearn].
- Let us say the password is "Potter". Convert this password string to bits using code available in NTULearn. Transform the generated list of bits to a $2 \times n$ matrix. Each column of this matrix represents the *s* and *t* bits. For each column of this matrix, use the previous function `random_vector(s,t)` to obtain a corresponding secret vector **u**. Thus, there will be *n* secret vectors $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_n$ such that $\mathbf{a}_0 \cdot \mathbf{u}_i = s_i$ and $\mathbf{b}_0 \cdot \mathbf{u}_i = t_i$.
- Next, we have to generate the secret bits $\beta_i, \gamma_i$ for each of the TAs. This can be done by taking the dot product of **a**$_i$, **b**$_i$ with **u**$_i$.
- <u>Recovery:</u> Choose any 3 TAs and their corresponding secret bits $\beta_i, \gamma_i$ and recover the first secret vector **u**$_1$ by solving (here we have chosen TAs 1, 2 and 3)

$$\begin{bmatrix} \mathbf{a}_1 \\ \mathbf{b}_1 \\ \mathbf{a}_2 \\ \mathbf{b}_2 \\ \mathbf{a}_3 \\ \mathbf{b}_3 \end{bmatrix} [\mathbf{u}_1] = \begin{bmatrix} \beta_1 \\ \gamma_1 \\ \beta_2 \\ \gamma_2 \\ \beta_3 \\ \gamma_3 \end{bmatrix}.$$

With the obtained $\mathbf{u}_1$, find the secret bits $s_1$ and $t_1$. Continue this process until all the $\mathbf{u}_i$'s are obtained and from which the secret bits $s_i$ and $t_i$ are recovered. Convert the bits to a string (using code available in NTULearn) to check that the password "Potter" is recovered.

## References

[1] Philip N. Klein, "Coding the matrix: Linear algebra through computer science applications", Newtonian Press, 2013.

[2] https://realpython.com/linear-regression-in-python/].

[3] Wikipedia contributors, "Secret sharing", Wikipedia, The Free Encyclopedia. May 20, 2020, at 16:27 UTC. Available at: https://en.wikipedia.org/w/index.php?title=Secret_sharing&action=history . Accessed June 26, 2020.

[4] A. Shamir, "How to share a secret" *Communications of the ACM*, vol. 22, no. 11, pp. 612-613, Nov. 1979.

[5] G. R. Blakley, "Safeguarding Cryptographic Keys", *Managing Requirements Knowledge, International Workshop on AFIPS*. vol. **48**, pp. 313–317, 1979.

# CE/CZ 1104 Linear Algebra for Computing

## Lab 1

In [37]:
```
# import all libraries needed
import numpy as np
from scipy import linalg
import math
import random
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
```

## Exercise 1

### Question 1

In [38]:
```
## declaration of variables

# password x
x = np.array([1,0,1,1,1])

#challenge vectors, c1 & c2
c1 = np.array([0, 1, 0, 1, 1])
c2 = np.array([1, 1, 1, 1, 0])
```

In [39]:
```
## Function that uses dot product and modulo

def respondToCV(x, cv):
    return np.dot(x, cv) % 2
```

In [40]:
```
## Harry's response to Carole's challenge vectors

# first CV
first = respondToCV(x, c1)
# second CV
second = respondToCV(x, c2)

#print results
print("beta1 : {}".format(first))
print("beta2 : {}".format(second))
```

```
beta1 : 0
beta2 : 1
```

### Question 2

In [41]:
```
## (01011 + 11110) · x = 01011 · x + 11110 · x

def xorVectors(a, b):
    return np.bitwise_xor(a,b)

cE = xorVectors(c1,c2)
print("c1 + c2 = cE = {}".format(cE))

# check
print("Response: ", respondToCV(cE, x))
```

```
c1 + c2 = cE = [1 0 1 0 1]
Response:  1
```

## Question 3

In [42]:
```
# matrix whose rows are the challenges that Eve already knows
Ch = np.array([[1, 1, 0, 0, 1, 1],
              [1, 0, 1, 0, 1, 0],
              [1, 1, 1, 0, 1, 1],
              [0, 0, 1, 1, 0, 0]])

# matrix response of challenges
ChResponse = np.array([[0],
                      [0],
                      [1],
                      [1]])

ca = np.array([0, 1, 1, 0, 0, 1])
cb = np.array([1, 1, 0, 1, 1, 1])
```

In [43]:
```
def addWeights(a, w):
    # returns array of given shape and type with zeros
    res = np.zeros(a.shape[1], dtype=int)
    for i in range(len(w)):
        if(w[i] == 1):
            res = xorVectors(res,a[i])
    return res
```

In [44]:
```
def deriveResponse(cKnown, bKnown, cNew):
    w = [0 for i in range(cKnown.shape[0])]
    for i in range(2**cKnown.shape[0]):
    # Get all possible weights
        t1 = bin(i)[2:]
        # add 0 to t1
        while(len(t1) < cKnown.shape[0]):
            t1 = "0" + t1
        for j in range(cKnown.shape[0]):
            if(t1[j] == "1"):
                w[j] = 1
            else:
                w[j] = 0
        temp = addWeights(cKnown,w)
        if(np.all(np.equal(temp, cNew))):
            break
    else:
        print("Not found")
        return -1
    print("Linear combination weights:", w)
    return np.dot(w, bKnown)%2
```

In [45]:
```
# responses betaA & betaB
betaA = deriveResponse(Ch, ChResponse, ca)
print("Response to ca : ", betaA)

betaB = deriveResponse(Ch, ChResponse, cb)
print("Response to cb: ", betaB)
```

```
Linear combination weights: [1, 1, 0, 0]
Response to ca :  [0]
Linear combination weights: [0, 0, 1, 1]
Response to cb:  [0]
```

## Question 4

In [46]:
```
# Stack arrays in sequence vertically (row wise)
ChNew = np.vstack((Ch, [[0, 1, 1, 0, 1, 1], [1, 1, 0, 1, 0, 0]]))
```

```
ChRNew = np.vstack((ChResponse, [[0], [1]]))

# Solves the linear equation set a * x = b for the unknown x for square a matrix.
solveForX = linalg.solve(ChNew, ChRNew)
# solveForX = [ 1 0  1  0 -2  1] (under R^n)
# 1-D array (x under GF2); flatten() so that it's 1 line and astype(int) so that no
print("Password x: ", np.mod(solveForX,2).flatten().astype(int))
```

```
Password x:  [1 0 1 0 0 1]
```

## Exercise 2

### Question 5

In [47]:
```
area = np.array([0.846, 1.324, 1.150, 3.037, 3.984])
price = np.array([115.00, 234.50, 198.00, 528.00, 572.50])

#take sum of area squared
sumOfArea2 = np.sum(np.multiply(area, area))
sumOfArea = np.sum(area)
sumOfPrice = np.sum(price)
sumOfAP = np.sum(np.multiply(area, price))
n = len(area)
```

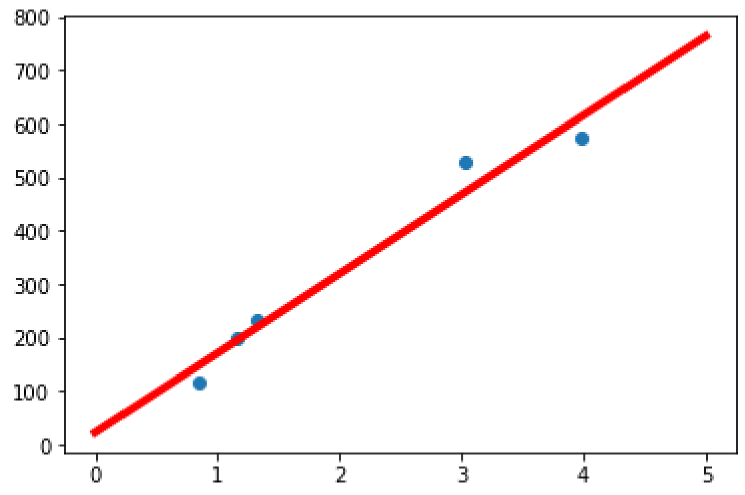
In [48]:
```
A = np.array([[sumOfArea2, sumOfArea], [sumOfArea, n]])
b = np.array([[sumOfAP],[sumOfPrice]])

print("Matrix A: ", A)
print("Vector b: ", b)
```

```
Matrix A:  [[28.886817 10.341   ]
 [10.341     5.      ]]
Vector b:  [[4519.844]
 [1648.   ]]
```

In [49]:
```
# Matrix product of two arrays after computing the (multiplicative) inverse of a mat
x = np.matmul(np.linalg.pinv(A), b)
#print("x : ", x.flatten())
m = x[0]
c = x[1]
print("m: ", m)
print("c: ", c)
```

```
m:  [148.20206538]
c:  [23.08848838]
```

In [50]:
```
xs = np.linspace(0,5,5)
ys = c + m*xs
plt.plot(xs,ys,'r',linewidth=4) #<- plots fitted line
plt.scatter(area,price); #<- plots data(x1, y)
plt.show()
```



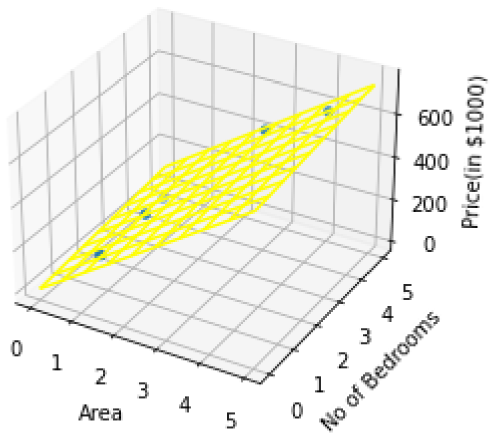
## Question 6

In [51]:
```
x1Area = np.array([0.846, 1.324, 1.150, 3.037, 3.984])
x2NumBR = np.array([1, 2, 3, 4, 5])
yPrice = np.array([115.00, 234.50, 198.00, 528.00, 572.50])

X = np.vstack((np.array([1,1,1,1,1]), x1Area, x2NumBR)).T
a = linalg.solve(np.matmul(X.T, X), np.matmul(X.T, yPrice))
print("vector a: ", a)
```

```
vector a:  [  9.97566234 130.67172705  16.45635726]
```

In [52]:
```
fig2 = plt.figure()
ax2 = plt.axes(projection='3d')

# labels
ax2.set_xlabel("Area")
ax2.set_ylabel("No of Bedrooms")
ax2.set_zlabel("Price(in $1000)")

x1L = np.linspace(0, 5, 10)
x2L = np.linspace(0, 5, 10)
XL, YL = np.meshgrid(x1L, x2L)
ZL = a[0] + a[1]*XL + a[2]*YL
ax2.plot_wireframe(XL, YL, ZL, color="yellow")
ax2.scatter3D(x1Area, x2NumBR, yPrice)
```

Out[52]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1b384183a60>
```



In [53]:
```
# Matrix product of X & a will result in the predicted prices
predictPrices = np.matmul(X, a)
for i in range(len(predictPrices)):
    # y price is in 1000 dollars, hence multiply 1000
    print("Predicted Price of House {0}: ${1}".format(i+1, predictPrices[i]*1000))
```

```
Predicted Price of House 1: $136980.3006797576
Predicted Price of House 2: $215897.7434696027
Predicted Price of House 3: $209617.22022323398
Predicted Price of House 4: $472651.1264256943
Predicted Price of House 5: $612853.6092017115
```

## Exercise 3

### Question 7

In [54]:

```
# Define a0 = [1 1 0 1 0 1] and b0 = [1 1 0 0 1 1]
a0 = np.array([1, 1, 0, 1, 0, 1])
b0 = np.array([1, 1, 0, 0, 1, 1])
```

In [55]:

```
## Python program for random binary string generation

# Function to create random binary string of length p
def rand_key(p):
    # Variable to store in an array
    key = []
    # Loop to find the string of desired length
    for i in range(p):
        # randint function to generate
        # 0, 1 randomly and add
        # the result into arry
        key.append(random.randint(0,1))
    return key
```

In [56]:

```
def random_vector(s, t):
    global a0, b0
    key = 0
    while(True):
        key = np.array(rand_key(6))
        if(respondToCV(a0, key) == s and respondToCV(b0, key) == t):
            break
    return key
```

In [57]:

```
'''Function to check if the generated vectors fulfil the independency requirements''
#input two lists a & b, each contains 4 vectors
#Return "True" if any 3 pairs of vectors from (a1,b1),(a2,b2),(a3,b3),(a4,b4) are li
def check_dependency(a,b):
    for v1 in range(1,3): #1st vector from 1 to 2
        for v2 in range(v1+1,4):
            for v3 in range(v2+1,5):
                squareMatrix = np.vstack((a[v1], b[v1], a[v2], b[v2],a[v3], b[v3]))
                determinant = np.linalg.det(squareMatrix)
                if determinant == 0: #if determinant is 0, the vectors are not linea
                    return False
    # check if a0,b0 and any two random selected pairs of vectors are linearly indep
    for v1 in range(1,4): #1st vector from 1 to 3
        for v2 in range(v1+1,5):
            squareMatrix = np.vstack((a[0], b[0], a[v1], b[v1], a[v2], b[v2]))
            determinant = np.linalg.det(squareMatrix)
            if determinant == 0: #if determinant is 0, the vectors are not linearly
                return False
    return True
```

In [58]:

```
def generateDep():
    global a0, b0
    aT = []
    bT = []
    while True:
        aT = [a0]
        bT = [b0]
        for i in range(4):
            aT.append(rand_key(6))
            bT.append(rand_key(6))
        # not linearly independent
        if(not check_dependency(aT, bT)):
            continue
        return aT, bT
```

In [59]:
```
at, bt = generateDep()
```

In [60]:
```
# assign
a1 = at[1]
b1 = bt[1]
a2 = at[2]
b2 = bt[2]
a3 = at[3]
b3 = bt[3]
a4 = at[4]
b4 = bt[4]
```

In [61]:
```
print(at)
```

```
[array([1, 1, 0, 1, 0, 1]), [0, 1, 0, 0, 1, 0], [0, 1, 1, 1, 1, 0], [1, 0, 1, 1, 0,
1], [1, 1, 1, 1, 0, 0]]
```

In [62]:
```
'''Function to converting String to binary array'''
def str2bits(s):
    res = ''.join(format(ord(i), 'b') for i in s)
    bitsArray = []
    for i in res:
        bitsArray.append(int(i))
    return bitsArray
```

In [63]:
```
pwdBit = np.array(str2bits("Potter"))
print("Password Bits:", pwdBit)
```

```
Password Bits: [1 0 1 0 0 0 0 1 1 0 1 1 1 1 1 1 1 0 1 0 0 1 1 1 0 1 0 0 1 1 0 0 1 0
1 1 1
 1 0 0 1 0]
```

In [64]:
```
n = pwdBit.shape[0]//2
pwdBT = pwdBit.reshape(2,n)

uV = []
for i in range(n):
    uV.append(random_vector(pwdBT[0][i], pwdBT[1][i]))
```

In [65]:
```
def FindBits(a,b):
    global uV
    bBits = []
    yBits = []
    for i in uV:
        bits = np.dot(a,i)
        bits2 = np.dot(b,i)

        bBits.append(bits)
        yBits.append(bits2)

    return np.vstack((bBits,yBits))
```

```
# each TA has a & b vector => dot product with u1 to u21
b1y1 = FindBits(a1,b1)
b2y2 = FindBits(a2,b2)
b3y3 = FindBits(a3,b3)
```

In [66]:
```
def bits2str(b):
    NumOfChar = len(b)//7
    string = ''
    for i in range(NumOfChar):
        bitsChar = ''.join(str(j) for j in b[7*i:7*i+7]) # 7 digits represents 1 cha
        decimalChar = int(bitsChar,2) #convert binary to decimal
        string = string + chr(decimalChar) #convert decimal to string
    return string
```

In [67]:
```
# stack together all 3
TABGBits= np.vstack((b1y1,b2y2,b3y3))
vectorBits = np.vstack((a1,b1,a2,b2,a3,b3))

uAns = (linalg.solve(vectorBits,TABGBits))
#print(uAns)
#print(np.allclose(np.dot(vectorBits, uAns), TABGBits)) <- check soln
a0Bits = np.array((respondToCV(a0,uAns)))
b0Bits = np.array((respondToCV(b0,uAns)))

combine = np.concatenate((a0Bits,b0Bits))
#print(combine)
#format to int64
orgPwd = bits2str(combine.astype('int64'))

print("Password: {}".format(orgPwd))
```

```
Password: Potter
```