# Tutorial 6: Least-Squares

In this tutorial you will learn how to solve least squares problems with Matlab and how to fit polynomials to data.

## The least-squares problem

Recall from class that the least squares problem is as follows. Given an mxn matrix **A** and a vector $\mathbf{b} \in R^m$, find a $\hat{\mathbf{x}} \in R^n$ such that

$$||\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}|| < ||\mathbf{b} - \mathbf{A}\mathbf{x}||,$$

for all $\mathbf{x} \in R^n$.

The solution of the least squares problem is given by the normal equations

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}.$$

We already know how to solve linear systems. For example, you can use the backslash command (\) to solve for $\hat{\mathbf{x}}$, since

$$\hat{\mathbf{x}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}.$$

In Matlab you can do this as follows.

```
A = [4 0
     0 2
     1 1];
b = [2
     0
     11];
 xhat = (A'*A)\(A'*b)
```

```
xhat = 2×1
     1
     2
```

You can try that this $\hat{\mathbf{x}}$ indeed solves the least-squares problem by comparing $||\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}||$ to a few $||\mathbf{b} - \mathbf{A}\mathbf{x}||$ for some randomly selected vectors $\mathbf{x}$. Here, you compute the norm of a vector using the command norm. Use the help command to learn how to use it.

```
help norm
```

```
  NORM   Matrix or vector norm.
       NORM(X,2) returns the 2-norm of X.
```

```
    NORM(X) is the same as NORM(X,2).

    NORM(X,1) returns the 1-norm of X.

    NORM(X,Inf) returns the infinity norm of X.

    NORM(X,'fro') returns the Frobenius norm of X.

  In addition, for vectors...

    NORM(V,P) returns the p-norm of V defined as SUM(ABS(V).^P)^(1/P).

    NORM(V,Inf) returns the largest element of ABS(V).

    NORM(V,-Inf) returns the smallest element of ABS(V).

  By convention, NaN is returned if X or V contains NaNs.

  See also COND, RCOND, CONDEST, NORMEST, HYPOT.

  Reference page in Doc Center
     doc norm
```

Now you are ready to try it.

```
x = randn(2,1); %generate a random 2x1 vector
norm(A*x-b)
```

```
ans = 16.4839
```

```
norm(A*xhat-b)
```

```
ans = 9.1652
```

If you execute this code several times, you wil see that norm(A*xhat-b) is always smaller than norm(A*x-b) for any randomly selcted x.

## Least-squares and QR

In class you also learned about the QR factorization. In Matlab, you can compute the QR-factorization of a matrix using the command qr. Use help to find out how to use it.

```
help qr
```

```
QR     Orthogonal-triangular decomposition.
   [Q,R] = QR(A), where A is m-by-n, produces an m-by-n upper triangular
   matrix R and an m-by-m unitary matrix Q so that A = Q*R.

   [Q,R] = QR(A,0) produces the "economy size" decomposition.
   If m>n, only the first n columns of Q and the first n rows of R are
   computed. If m<=n, this is the same as [Q,R] = QR(A).

   If A is full:

   [Q,R,E] = QR(A) produces unitary Q, upper triangular R and a
   permutation matrix E so that A*E = Q*R. The column permutation E is
   chosen so that ABS(DIAG(R)) is decreasing.

   [Q,R,e] = QR(A,'vector') returns the permutation information as a
```

Now you are ready to try it.

```
[Q,R]=qr(A);
```

Check that that **A** = **QR**:

```
A-Q*R
```

```
ans = 3×2
10^-15 ×
   -0.8882   -0.0555
         0         0
         0         0
```

Note the now familiar numerical error.

To use the QR-factorization, recall that

$$\hat{\mathbf{x}} = \mathbf{R}^{-1}\mathbf{Q}^T\mathbf{b}.$$

In Matlab, you can now compute $\hat{\mathbf{x}}$ via the QR-factorization:

```
xhat_qr = R\(Q'*b)
```

```
xhat_qr = 2×1
```

```
    1.0000
    2.0000
```

Not surprisingly, you get the same answer as before.

## Least-squares solutions with \

Matlab's backslash (\) is "overloaded". This means that the operation can do multiple things and will figure out what you want to do based on the matrices/vectors you use the backslash on. In particular, the backslash can also be used to compute least-squares solutions. If you use the backslash with a square matrix, it will solve the linear system. If you use the backslash with a mxn matrix with $m \neq n$, it will solve the least squares problem.

```
xhat_backslash = A\b
```

```
xhat_backslash = 2×1
    1.0000
    2.0000
```

Again, you get the same result as before. When you solve more difficult problems, using the backslash is computationally the best choice you have (because many Matlab-engineers have worked hard to make solving least-squares problems fast.

## Fitting data with linear models

You learned in class that you can use least-squares to fit linear models to data. In this case, you solve the least-squares problem given by

$$\mathbf{X}\boldsymbol{\beta} = \mathbf{y},$$

where **y** are the *m* data points you have and where **X** is a m x 2 matrix whose first column is a vector of ones and the second columns cotains the *m* values of x you are given. The vector $\boldsymbol{\beta}$ contains the coefficients of your linear model:

$$y = \beta_0 + \beta_1 x$$

Load a data set to get started with fitting linear models. In matlab, you load files using the load command. Use help to find out how.

```
clear % delete old variables
help load
```

```
 LOAD Load data from MAT-file into workspace.
    S = LOAD(FILENAME) loads the variables from a MAT-file into a structure
    array, or data from an ASCII file into a double-precision array.
    Specify FILENAME as a character vector or a string scalar. For example,
    specify FILENAME as 'myFile.mat' or "myFile.mat".

    S = LOAD(FILENAME, VARIABLES) loads only the specified variables from a
```

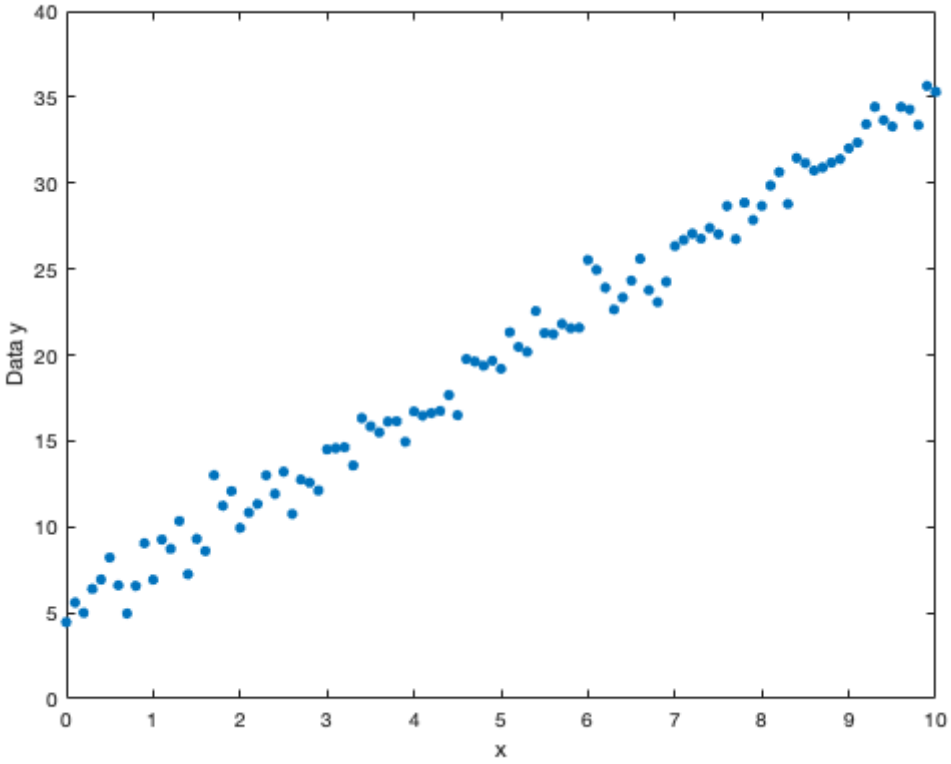
```
MAT-file. Specify FILENAME and VARIABLES as character vectors or string
scalars. When specifying VARIABLES, use one of the following forms:

    VAR1, VAR2, ...          Load the listed variables.  Use the '*'
                             wildcard to match patterns.  For
                             example, load('A*') loads all variables
                             that start with A.
    '-regexp', EXPRESSIONS   Load only the variables that match the
```

You you can load a data set:

```
load('./Data/LS_Problem1.mat')
```

And plot it:

```
plot(x,y,'.','MarkerSize',15)
xlabel('x')
ylabel('Data y')
```




Now you have loaded two variables, x and y. You can see what you have loaded into memory by using the who command:

```
who
```

```
Your variables are:

x  y
```

You can see the size of the varibales using the size command:

```
size(x)
```

```
ans = 1×2
   101     1
```

```
size(y)
```

```
ans = 1×2
   101     1
```

You can see that you have 101 data points and 101 x values. This means that m = 101 (because you have 101 data points) and n = 2 (because you fit a linear model).

Now you can assemble the matrix **X** you need:

```
m = 101;
n = 2;
X  = [ones(m,1) x];
```

And now that you have **X** and **y**, you can solve the least-square problem. I use the backslash, because it is computationally the best choice, but you could also use QR-factorization.

```
beta = X\y;
beta
```

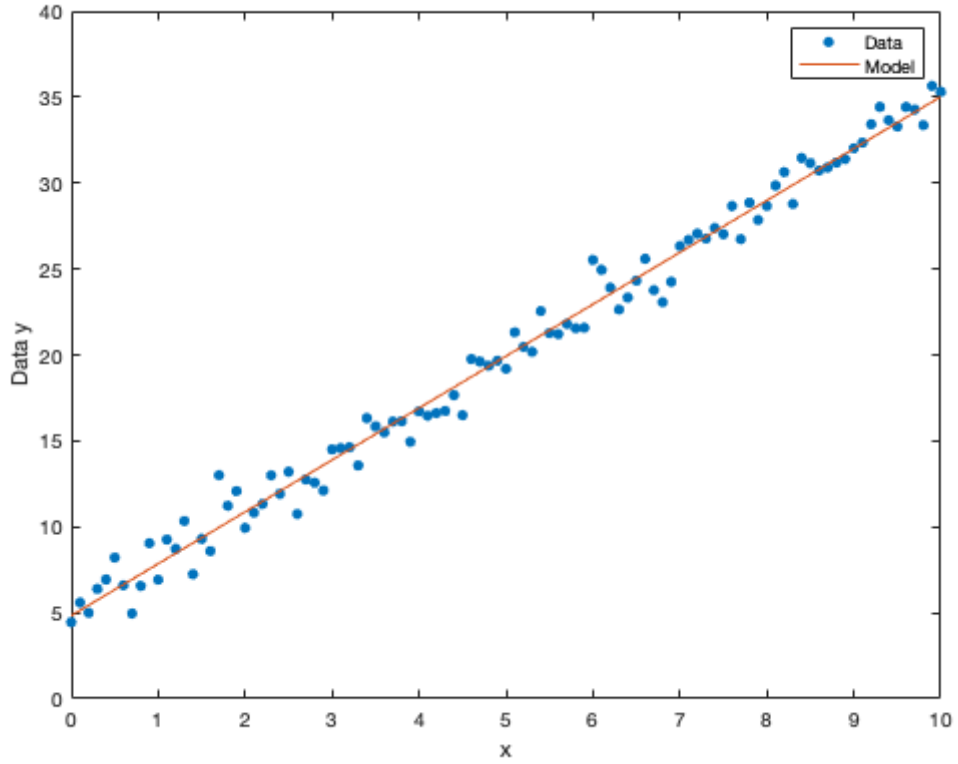
```
beta = 2×1
    4.8368
    3.0190
```

This means that the "best" (in the least squares sense) linear function that fits the data is given by

```
y_model = beta(1) + beta(2)*x;
```

You can look at this model by plotting the data and the output of your model in the same figure:

```
plot(x,y,'.','MarkerSize',15)
hold on, plot(x,y_model)
legend('Data','Model')
xlabel('x')
ylabel('Data y')
```

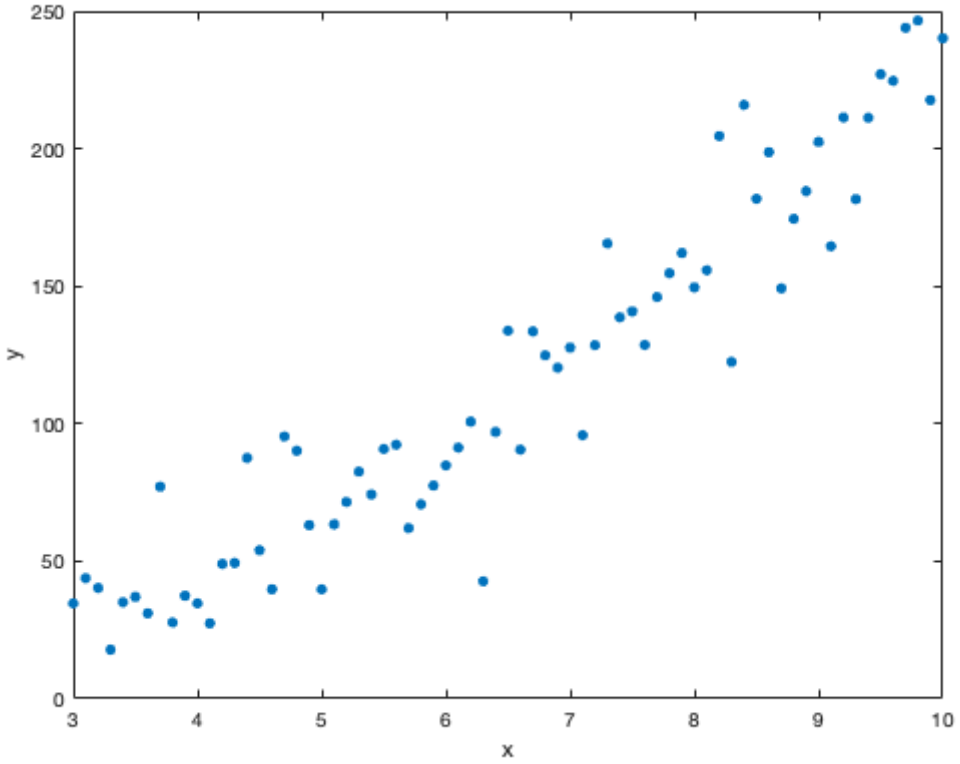



You can see that the model is very good: all data points are near the linear function your model predicts.

## Fitting data with nonlinear models

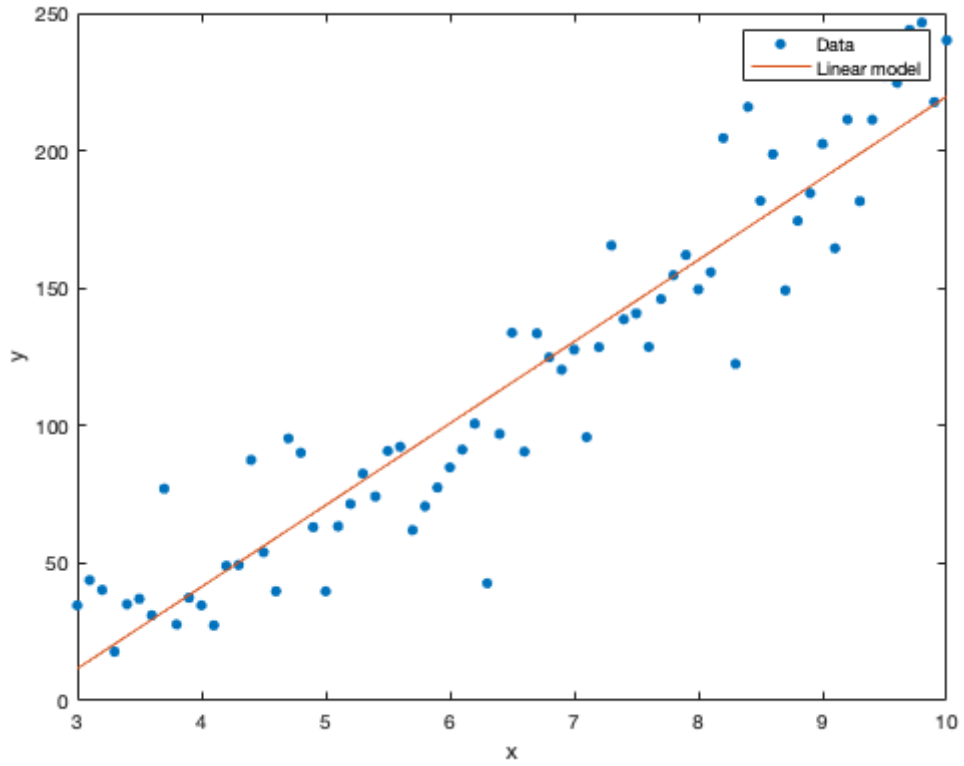
Let's take a look at another data set.

```
clear % delete old variables
load('./Data/LS_Problem2.mat')
m = length(y);
figure
plot(x,y,'.','MarkerSize',15)
xlabel('x')
ylabel('y')
```




We fit a linear model:

```
X_lin = [ones(m,1) x];
beta_lin = X_lin\y;
y_lin_model = beta_lin(1)+beta_lin(2)*x;
hold on, plot(x,y_lin_model);
legend('Data','Linear model')
```




It looks "ok", but perhaps we can do better by fitting a quadradtic function. To do that, we solve the least-squares probelm

$$\mathbf{X}\boldsymbol{\beta} = \mathbf{y},$$

where **y** are the *m* data points you have and where **X** is a m x 3 matrix and where the 3x1 vector $\boldsymbol{\beta}$ contains the coeffients of your quadratic model:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2.$$

In this case, the m x 3 matrix **X** has as its first column a vector that contains only ones, the second column is a vector that contains x, the third column contains $x^2$. You create this matrix as follows:

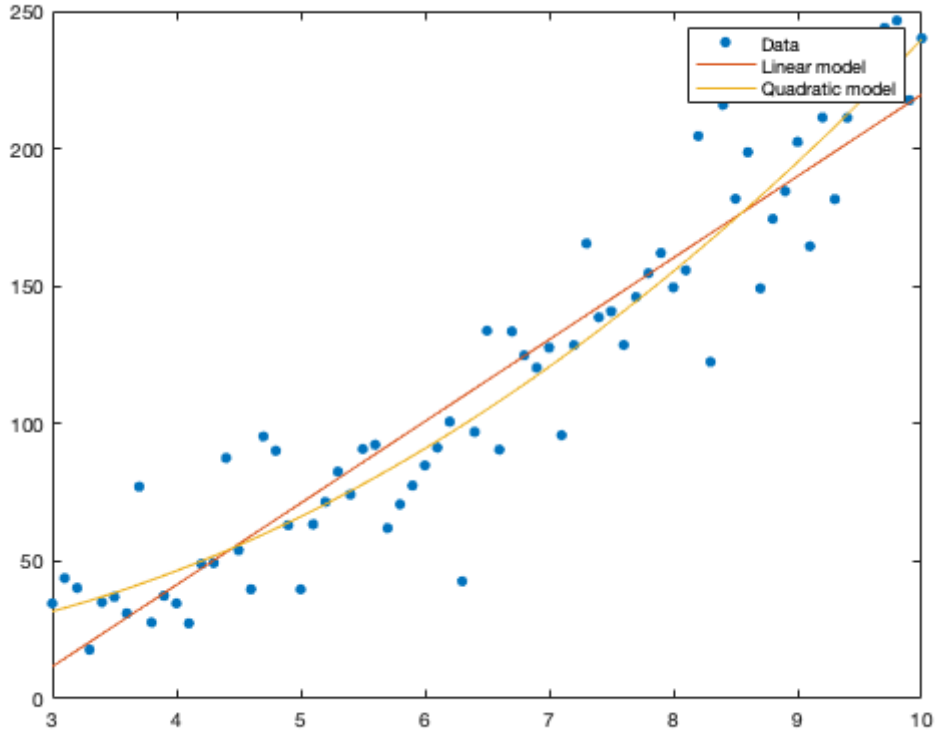
```
X_quad = [ones(m,1) x x.^2];
```

Now you are ready to solve the least-squares problem:

```
beta_quad = X_quad\y;
```

You can now compare your linear and quadratic model to the data.

```
y_quad_model = beta_quad(1)+beta_quad(2)*x+beta_quad(3)*x.^2;
figure
plot(x,y,'.','MarkerSize',15)
hold on, plot(x,y_lin_model);
hold on, plot(x,y_quad_model);
legend('Data','Linear model','Quadratic model')
```




It is difficult to say which model is better. To find out more, we can look at the "error", defined by:

$$e = ||\mathbf{y} - \mathbf{X}\boldsymbol{\beta}||.$$

First compute the error for the linear model:

```
e_lin = norm(y-X_lin*beta_lin)
```

```
e_lin = 179.9877
```

Now compute the error of the quadratic model:

```
e_quad = norm(y-X_quad*beta_quad)
```

```
e_quad = 161.7072
```

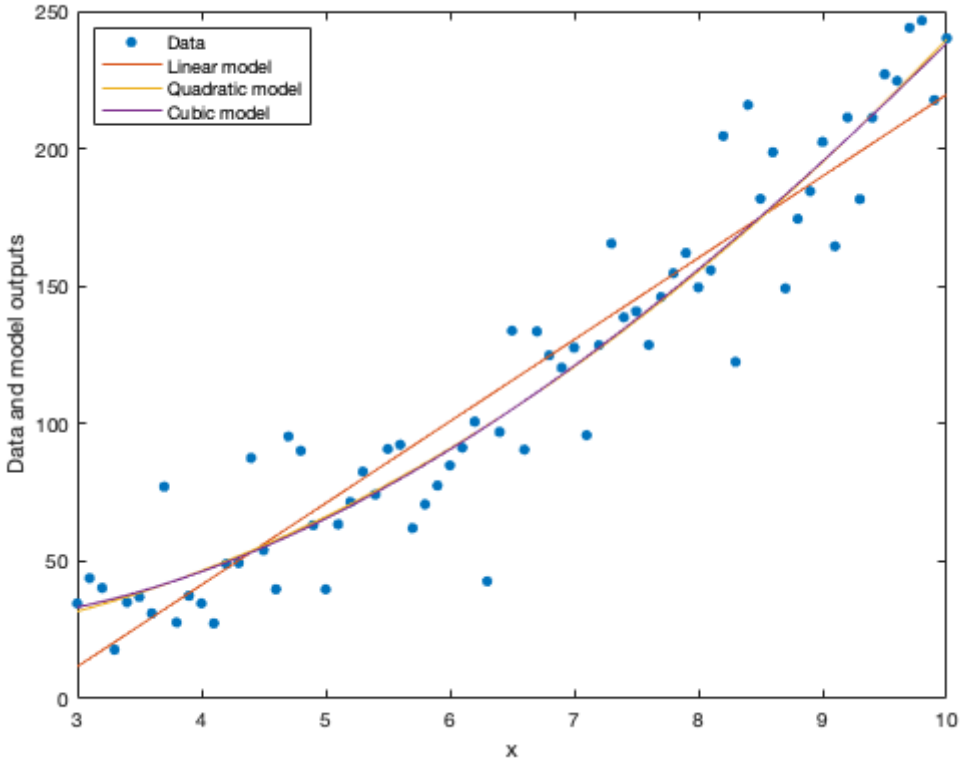
The error of the quadratic model is smaller than the error of the linear model, which suggests that the quadratic model is better (in the sense that it leads to a smaller error).

We can also fit a cubic model:

```
X_cub = [ones(m,1) x x.^2 x.^3];
beta_cub = X_cub\y;
y_cub_model = beta_cub(1)+beta_cub(2)*x + beta_cub(3)*x.^2 + beta_cub(4)*x.^3;
```

Let's plot the results of all models:

```
figure
plot(x,y,'.','MarkerSize',15)
hold on, plot(x,y_lin_model)
hold on, plot(x,y_quad_model)
hold on, plot(x,y_cub_model)
xlabel('x')
ylabel('Data and model outputs')
legend('Data','Linear model','Quadratic model','Cubic model','Location','NorthWest')
```




We can also compute the errors:

```
e_cub = norm(y-X_cub*beta_cub);
fprintf('Error of linear model: %g\n',e_lin)
```

```
Error of linear model: 179.988
```

```
fprintf('Error of quadratic model: %g\n',e_quad)
```

```
Error of quadratic model: 161.707
```

```
fprintf('Error of cubic model: %g\n',e_cub)
```

```
Error of cubic model: 161.623
```

Note that the error of the cubic model is not significantly smaller than the error of the quadratic model. In this scenario, you might be better of chosing the "simpler" model, i.e., the quadratic one, because there is little reason to use a more complicated model (the cubic one gives nearly the same error as the quadratic model).

For this exampe, I made up the "data" and it is generated by a quadratic function, corrupted by "noise" (some random numbers). Thus, for this example, our strategy of trying a few models and accepting the simplest one that leads to nearly as small an error as a more complicated one works just fine. In many case, however, the question of what model to chose is very difficult to answer because there is not much mathematical theory that helps you answering that question. Perhaps this motivates you to work on such a theory - if you are successful, you work will have a great impact on many problems in engineering and science.

## Exercise

Use least-squares to compute a model for the data in the file "Excercise.mat". You can load these data with this command:

```
clear % delete old variables
load('./Data/LS_Excercise.mat')
```

You can find out what the number of data points is using the "size" or "length" commands. Use help to find out about these two commands:

```
help size
```

```
  SIZE   Size of array.
     D = SIZE(X), for M-by-N matrix X, returns the two-element row vector
     D = [M,N] containing the number of rows and columns in the matrix.
     For N-D arrays, SIZE(X) returns a 1-by-N vector of dimension lengths.
     Trailing singleton dimensions are ignored.

     [M,N] = SIZE(X) for matrix X, returns the number of rows and columns in
     X as separate output variables.

     [M1,M2,M3,...,MN] = SIZE(X) for N>1 returns the sizes of the first N
     dimensions of the array X.  If the number of output arguments N does
     not equal NDIMS(X), then for:

     N > NDIMS(X), SIZE returns ones in the "extra" variables, i.e., outputs
                   NDIMS(X)+1 through N.
```

```
help length
```

```
  LENGTH   Length of vector.
     LENGTH(X) returns the length of vector X.  It is equivalent
     to MAX(SIZE(X)) for non-empty arrays and 0 for empty ones.
```

```
    See also NUMEL.

    Reference page in Doc Center
       doc length

    Other functions named length

       Bluetooth/length        gpib/length        serial/length
       categorical/length      gpuArray/length    tall/length
       codistributed/length    i2c/length         tcpip/length
       Composite/length        icdevice/length    tscollection/length
```

Now you are ready to use it:

```
length(x)
```

```
ans = 41
```

Thus, you have $m = 41$ data points.

Extend the code from this Tutorial to build models which are polynomials of order up to eight. Which model do you recommend using to model these data?

**Solution**

First we plot the data:

```
figure
plot(x,y,'.','MarkerSize',15)
```

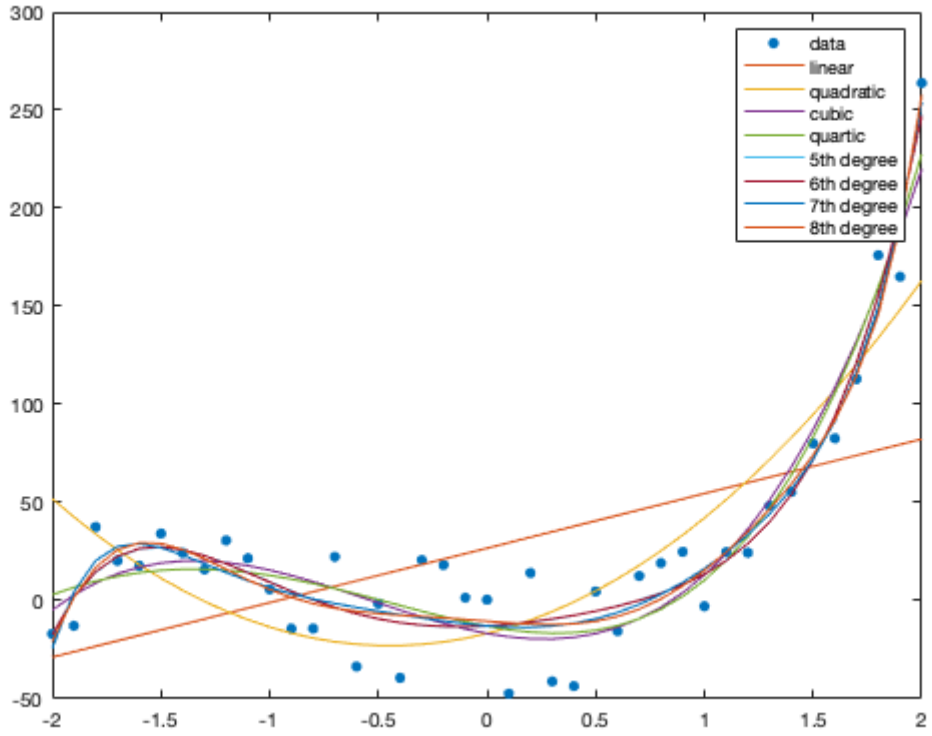
Next we can construct linear and nonlinear models. I use models that are polynomials of degree up to 8. I do that in a for loop, but you can also create these models individually.

```
m = length(x);
for kk=1:8
    X = [ones(m,1)];
    for ll=1:kk
        X = [X x.^ll];
    end
    bet = X\y;
    y_model = X*bet;
    e = norm(y-y_model);
```

```
    fprintf('Error of order %g model: %g\n',kk,e)

    hold on, plot(x,y_model)
end
```

```
Error of order 1 model: 323.502
Error of order 2 model: 206.648
Error of order 3 model: 132.067
Error of order 4 model: 130.369
Error of order 5 model: 117.891
Error of order 6 model: 117.886
Error of order 7 model: 115.666
Error of order 8 model: 114.896
```

```
legend('data','linear','quadratic','cubic','quartic', ...
     '5th degree', '6th degree', '7th degree', '8th degree')
```



Since the error does not decrease very much for models of degree 5 or higher, I would recommend a 5th order polynomial as a good model for these data.