Lecture 7: Models and data—Basics of least squares fitting

Recap

In Lecture 6, we took a close look at the autocovariance and at decorrelation scales. Today we'll move on to ask how we can fit a model to data, and of course we'll take advantage of what we've learned about the autocovariance and decorrelation as we go.

Introduction to data and models

Now that we've built a statistical framework, let's put it to work to interpret our data. When we looked at correlation, we considered data y and we wanted to represent with an approximation $\hat{y} = \alpha x$, and we sought an α that would minimize the difference betwen y and $\hat{y}$. Now we want to expand to a more complicated case.

To do this, let's lay out a little notation. First, we need data. Let's write the data as an N-vector of real numbers

$$\mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{bmatrix} \tag{1}$$

Here we use bold lowercase letters to indicate vectors, and in handwritten work, we'll underline vectors with a single line (e.g. $\underline{d}$. There are necessarily a finite number of data.

In addition we need a model. We will consider discrete inverse theory, in which case there are a finite number of model parameters, an M-vector

$$\mathbf{m} = \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_M \end{bmatrix} \tag{2}$$

The relationship between the data and model parameters may, in general, be stated as a series of L equations

$$\mathbf{f}(\mathbf{d}, \mathbf{m}) = \begin{bmatrix} f_1(\mathbf{d}, \mathbf{m}) \\ f_2(\mathbf{d}, \mathbf{m}) \\ \vdots \\ f_L(\mathbf{d}, \mathbf{m}) \end{bmatrix} = \mathbf{0}. \tag{3}$$

The equality in (3) implies that both the data and model are perfect. We will relax this unlikely assumption as we find solutions. It is nearly always possible to write the relationship as:

$$\mathbf{d} = \mathbf{g}(\mathbf{m}) \tag{4}$$

where g is a vector function. If g is linear then we can expres this as a matrix equation.

$$\mathbf{d} = \mathbf{G}\mathbf{m} \tag{5}$$

where $\mathbf{G}$ is an $N \times M$ matrix. In standard notation, bold capitalized letters indicate matrices, and in handwritten work, they're denoted with a double underline (e.g. $\underline{\underline{G}}$).

Models and data

Numerical weather prediction models and ocean state estimates are complex, multi-equation, non-linear systems that use data to constrain models. In class, we looked at examples from 4-dimensional variational assimilation systems in the California Current and in the Southern Ocean. In essence, these systems solve a minimization problem to reduce the misfit between the observations and the model.

Before we think about the complicated systems used to deliver weather forecasts, let's consider linear problems. In a linear problem, the model parameters $\mathbf{m}$ are linear multipliers of some set of model functions.

For example, we could imagine representing the measured temperature values T_i through a model estimate $\hat{T}_i$:

$$\hat{T}_i = \sum_{m=1}^{M} \alpha_m f_m(x_i), \tag{6}$$

where our data in this case would be

$$\mathbf{d} = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_N \end{bmatrix} \tag{7}$$

and our model parameters are:

$$\mathbf{m} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_M \end{bmatrix} \tag{8}$$

Classically we set this up as an overdetermined problem, with more equations than unknowns, so $N > M$, which means that we have more information than unknowns and we can hope to solve for our unknowns.

We can write this linear system of equations as a matrix equation of the form

$$\mathbf{d} = \mathbf{G}\mathbf{m} \tag{9}$$

where $\mathbf{G}$ is an $N \times M$ matrix.

For example, consider global mean sea level rise. Suppose that sea level is rising quadratically. We might hope to fit observed values of sea level ($\eta(t)$) to a constant, a linear trend, and a quadratic term:

$$\eta(t) = \eta_0 + \alpha t + \beta t^2 \tag{10}$$

We could write this as a matrix equation, with $d_i = \eta_i$, and

$$\mathbf{G} = \begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ \vdots & \vdots & \vdots \\ 1 & t_N & t_N^2 \end{bmatrix}. \tag{11}$$

where the column of ones allows us to identify a constant (η_0). The unknown model parameters are:

$$\mathbf{m} = \begin{bmatrix} \eta_0 \\ \alpha \\ \beta \end{bmatrix}. \tag{12}$$

How do we solve for our model parameters? First we have to decide on a goal. We'd like to minimize the misfit between $\mathbf{Gm}$ and $\mathbf{d}$, which we can define as $\mathbf{e} = \mathbf{Gm} - \mathbf{d}$. What do we actually minimize?

1. Minimize the absolute value of the misfit, which is called the L_1 norm:
$$||e||_1 = \sum_{i=1}^{N} |e_i| \tag{13}$$

2. Minimize the squared misfit, the L_2 norm:
$$||e||_2 = \left(\sum_{i=1}^{N} e_i^2 \right)^{1/2} \tag{14}$$

3. Minimize a higher-order norm, the L_n norm:
$$||e||_n = \left(\sum_{i=1}^{N} |e_i|^n \right)^{1/n} \tag{15}$$

4. Minimize the highest-order norm possible, the L_∞ norm:
$$||e||_\infty = \max_i |e_i| \tag{16}$$

Which is the best norm for you depends on the problem. The different norms imply different weighting. The L_1 norm, for example, gives less weight to large outliers than the L_2 norm. While the L_1 norm weights all misfits equally, the L_∞ norm minimizes the size of the largest misfit only.

The traditional least squares problem employs the L_2 norm as the "best" measure. The L_2 norm has several useful traits. First, it's more mathematically tractable than the L_1 norm, which has an inconvenient absolute value. Second, if the statistics of the misfit have a normal distribution, then there is an argument that the L_2 norm is appropriate. We'll see the practical advantage in using the L_2 norm in solving this problem. Our goal is to find $\mathbf{m}$ such that $||e||_2$ is minimized. Specifically, the quantity
$$\epsilon = \mathbf{e}^T\mathbf{e} = (\mathbf{Gm} - \mathbf{d})^{\mathbf{T}}(\mathbf{Gm} - \mathbf{d}) \tag{17}$$
is to be minimized with respect to $\mathbf{m}$. To accomplish this, we differentiate by each of the components m_i, set the result equal to zero, and solve for m_i. To do this, we'll need two useful identities.

Identity 1

Consider the scalar α formed by an inner product of two vectors $\mathbf{a}$ and $\mathbf{b}$:
$$\alpha = \mathbf{a}^T\mathbf{b}. \tag{18}$$

This can be written as a sum over the vectors' components a_i, b_i:
$$\alpha = \sum_{i=1}^{N} a_i\, b_i. \tag{19}$$

Now differentiate with respect to one of the components b_k:

$$\frac{\partial \alpha}{\partial b_k} = a_k. \tag{20}$$

The differentiation in (20) can be done for each component, and then the result arranged as a vector

$$\frac{\partial \alpha}{\partial \mathbf{b}} = \mathbf{a}. \tag{21}$$

Identity 2

Consider the scalar β formed by a quadratic product involving vector $\mathbf{b}$ and square matrix $\mathbf{A}$:

$$\beta = \mathbf{b}^T\mathbf{A}\mathbf{b}. \tag{22}$$

Write this as a sum

$$\beta = \sum_{i=1}^{N}\sum_{j=1}^{N} b_i A_{ij} b_j. \tag{23}$$

Differentiate with respect to b_k

$$\frac{\partial \beta}{\partial b_k} = \sum_{j=1}^{N} A_{kj} b_j + \sum_{i=1}^{N} b_i A_{ik}. \tag{24}$$

Now arranging as a vector

$$\frac{\partial \beta}{\partial \mathbf{b}} = \mathbf{A}\mathbf{b} + \mathbf{A}^T\mathbf{b} \tag{25}$$

If $\mathbf{A}$ is symmetric, then $\mathbf{A} = \mathbf{A}^T$, and:

$$\frac{\partial \beta}{\partial \mathbf{b}} = 2\mathbf{A}\mathbf{b}. \tag{26}$$

Solution of the classic overdetermined least squares problem

Use the identities to differentiate the misfit (17) with respect to the model parameters $\mathbf{m}$, and set the result equal to zero.

$$\begin{aligned} \epsilon &= \mathbf{m}^T\mathbf{G}^T\mathbf{G}\mathbf{m} - \mathbf{2d^TGm} + \mathbf{d^Td} & (27)\\ \frac{\partial \epsilon}{\partial \mathbf{m}} &= 2\mathbf{G}^T\mathbf{G}\mathbf{m} - \mathbf{2d^TG} = \mathbf{0} & (28) \end{aligned}$$

The equation to be solved is then

$$2\mathbf{G}^T\mathbf{G}\mathbf{m} = \mathbf{2d^TG}, \tag{29}$$

which is a set of M equations in M unknowns. Since (29) is linear in $\mathbf{m}$, it is easy to solve, which is a practical advantage of using the L_2 norm, or any other quadratic measure of misfit. The solution is

$$\mathbf{m} = (\mathbf{G}^T\mathbf{G})^{-1}\mathbf{G}^T\mathbf{d}. \tag{30}$$

as long as the $M \times M$ square matrix $\mathbf{G}^T\mathbf{G}$ is invertible. The notion of invertibility can be discussed with great mathematical precision, and we will do that later. For now, it is worth making a few comments. In the least squares problem, there are N linear combinations of the model parameters that we want to be close to the data. A linear combination is just a weighted sum of the components of a vector, in this case defined by the N rows of $\mathbf{G}$. As long as M of the rows are independent, then the M model parameters can be unambiguously determined. So invertibility is a property of the matrix $\mathbf{G}$, which is itself an expression of our model. Thus invertibility is a statement about the quality of the model that we have set up. If we set up a problem that is not invertible, that's our mistake.

Least-squares fit example

In class we looked at recent estimates of sea level rise, which has been modeled (or fitted) with a quadratic. Let's generate some fake data to test this out. Here we'll generate data with known parameters and test how well we can recover these parameters:

```
% first define the parameters for fake data.
x0=1;  % constant
x1=2;  % linear trend
x2=0.5;  %quadratic growth
sigma=500;  % standard deviation of noise

t=(0:1:150)';  % time variable
data_true = x0 + x1*t + x2*t.^2;  % create noise-free ``true'' data
data = x0 + x1*t + x2*t.^2 + sigma*randn(size(t));  % create noisy data

% plot data
plot(t,data_true,'LineWidth',2); hold on
plot(t,data,'.','LineWidth',3)
h=gca; set(h,'FontSize',14)
xlabel('"time"','FontSize',14)
ylabel('"data values"','FontSize',14)
```

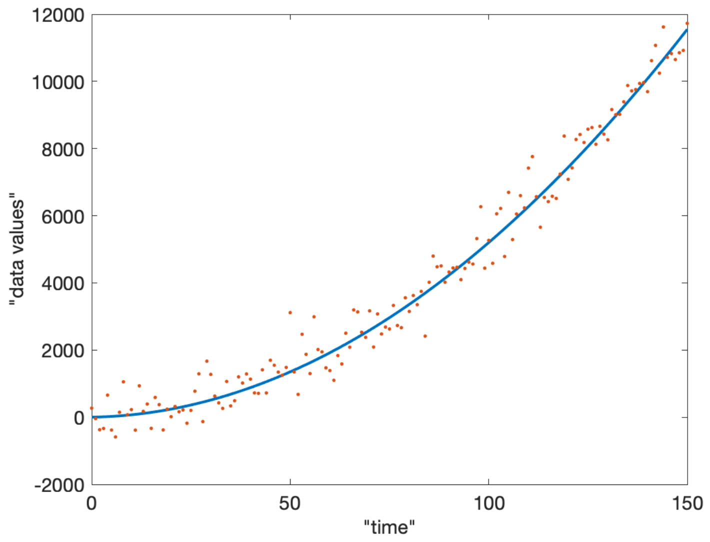
or

```
import numpy as np
import scipy
import xarray as xr
import cmocean as cmo
import matplotlib.pyplot as plt
from numpy import linalg as LA

# first define the parameters for fake data.
x0=1;  # constant
x1=2;  # linear trend
x2=0.5;  #quadratic growth
sigma=500;  # standard deviation of noise

t=np.arange(0,150,1)  # time variable
data_true = x0 + x1*t + x2*t**2 # create noise-free ``true'' data
data = data_true + sigma*np.random.normal(size=[len(t)]) # create noisy data
```

```
#  plot data
plt.plot(t,data_true,linewidth=2)
plt.plot(t,data,'.',linewidth=3)
plt.xlabel('"time"',fontsize=14)
plt.ylabel('"data values"',fontsize=14)
```



Now we can use our least-squares solution to fit the data:

```
% define the matrix G
G=[ones(size(t)) t t.^2];

% fit data: here with a full inversion:
model_fit=inv(G'*G)*G'*data
```

or

```
# define the matrix G
G=np.ones([len(t),3])
G[:,1]=t
G[:,2]=t**2

# fit data: here with a full inversion:
# find paremeters
GTG = np.matmul(G.T,G)
D = np.matmul(LA.inv(GTG), G.T)
model_fit = np.matmul(D,data)

# or if we're brave enough to dump everything in one line

model_fit2 = np.matmul(np.matmul(LA.inv(np.matmul(G.T,G)),G.T),data)
```

If we use “data_true”, then the results of this show:

$$\mathbf{m}_{true} = \begin{bmatrix} 1 \\ 2 \\ 0.5 \end{bmatrix} \tag{31}$$

while for “data”

$$\mathbf{m}_{fitted} = \begin{bmatrix} 42.8 \\ 3.4 \\ 0.49 \end{bmatrix}. \tag{32}$$

Of course the exact values will be different each time I generate new random numbers. You’ll notice that in this case, the quadratic fit is fairly accurate, but the the other two terms don’t appear too close to the solution.

Matlab provides a quick solution for a matrix inversion problem of this type. We can get the same results using a backslash:

```
model_fit = G\data
```

Python does not provide such compact syntax, but you can find the leastsquares solution in one step using:

```
model_fit=np.linalg.lstsq(G,data)
model_fit[0]
```

Regardless of the solution strategy that we use, we can overlay our fit over the original data:

```
plot(t,G*model_fit,'LineWidth',2)
legend('"true" model','"noisy" observations','model fit','FontSize',14)
```

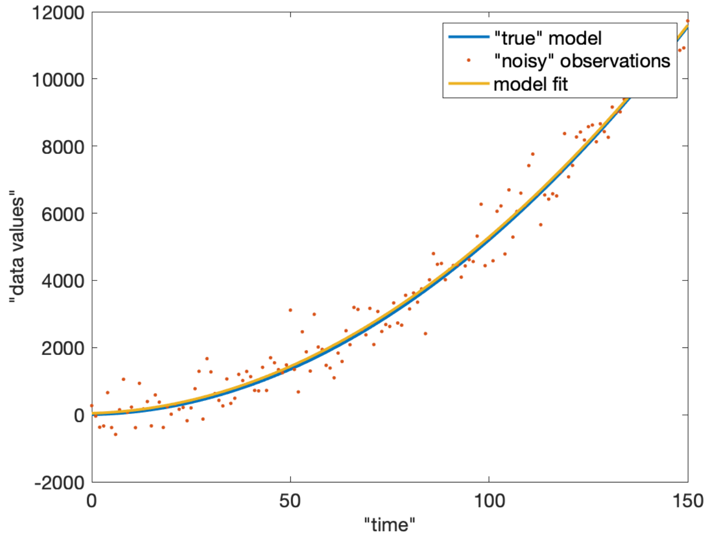
or

```
plt.plot(t,data_true,linewidth=2,label='"true" model')
plt.plot(t,data,'.',linewidth=3,label='"noisy" observations')
plt.plot(t,np.matmul(G,model_fit2),linewidth=2,label='model fit')
plt.xlabel('"time"',fontsize=14)
plt.ylabel('"data values"',fontsize=14)
plt.legend(fontsize=14)
```

Variations on the basic theme Now what happens if some of our data points are wild outliers with crazy values? These could really obliterate our results, all the more so since the L_2 norm squares the misfit so is designed to minimize the misfit relative to extreme outliers.

As an illustration, what if we reset a handful of our data values to have really enormous departures from tthe true model. In this example I’ve reset the data values and given them a large mean, but I could have made things just as problematic merely by amplifying the error

```
% reset values 100 to 110 to be big, with big variance.
data(100:110)=14000+5*sigma*randn(11,1);
plot(t,data,'.','LineWidth',3)
legend('"true" model','"noisy" observations','model fit',...
  'observations with large outliers', 'FontSize',14)
```



or

```
# reset values 100 to 110 to be big, with big variance.
data_new=data
data_new[100:111]=14000+5*sigma*np.random.normal(size=[11])
plt.plot(t,data_true,linewidth=2,label='"true" model')
plt.plot(t,data,'.',linewidth=3,label='"noisy" observations')
plt.plot(t,np.matmul(G,model_fit2),linewidth=2,label='model fit')
plt.plot(t,data_new,'.',linewidth=3,label='observations with large outliers')
plt.xlabel('"time"',fontsize=14)
plt.ylabel('"data values"',fontsize=14)
plt.legend(fontsize=10)
```

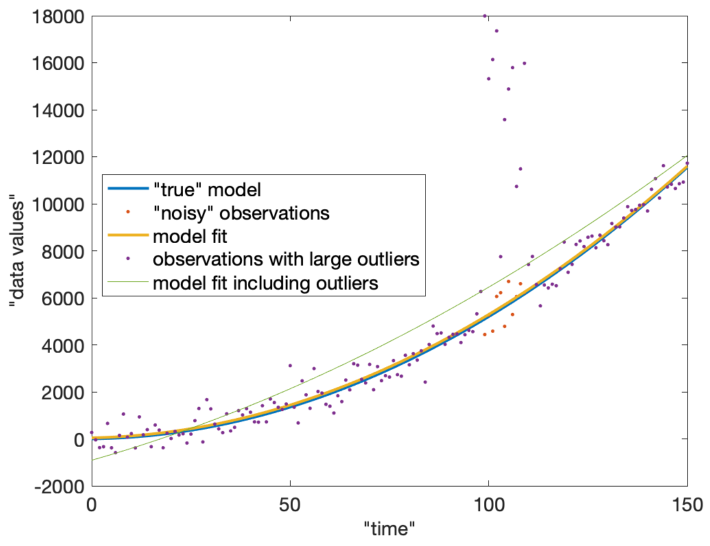
If we blindly fit, including these outliers, then we our fit will show large departures from the true parameters.

```
model_fit_new = G\data
plot(t,G*model_fit_new);
legend('"true" model','"noisy" observations','model fit',...
  'observations with large outliers',...
  'model fit including outliers','FontSize',14)
```

or

```
# blindly fitting with outliers
model_fit_new = np.linalg.lstsq(G,data_new)

plt.plot(t,data_true,linewidth=2,label='"true" model')
plt.plot(t,data,'.',linewidth=3,label='"noisy" observations')
plt.plot(t,np.matmul(G,model_fit2),linewidth=2,label='model fit')
plt.plot(t,data_new,'.',linewidth=3,label='observations with large outliers')
plt.plot(t,np.matmul(G,model_fit_new[0]),
         linewidth=3,label='model fit including outliers')
plt.xlabel('"time"',fontsize=14)
plt.ylabel('"data values"',fontsize=14)
plt.legend(fontsize=10)
```



What can we do about the outliers? To get us started, here are two possible strategies:

1. *Remove the bad data from the fitting process.* To do this, in Matlab I find the indices of the "good" data, and use only the good data to carry out my fit:

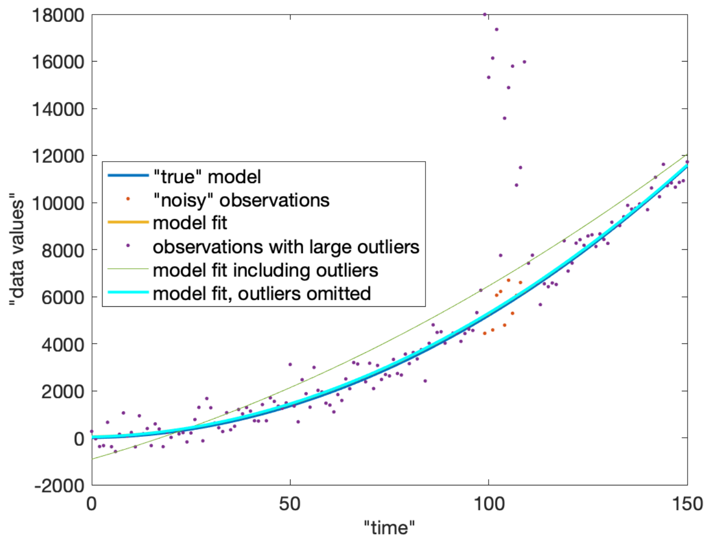
```
% find "good" data
xx=1:length(t); xx(100:110)=[];
% carry out fit using only "good" data and
% only the corresponding rows of G
model_fit_new2=G(xx,:)\data(xx)

plot(t,G*model_fit_new2,'c','LineWidth',2);
legend('"true" model','"noisy" observations','model fit',...
  'observations with large outliers',...
  'model fit including outliers',...
  'model fit, outliers omitted','FontSize',14)
```

   or

```
# find "good" data
indices = np.where(np.any([t<100,t>110],axis=0))
data_cleaned = data_new[indices]
G_cleaned = np.squeeze(G[indices,0:3])
# xx=1:length(t); xx(100:110)=[];
# carry out fit using only "good" data and
# only the corresponding rows of G
model_fit_new2=np.linalg.lstsq(G_cleaned,data_cleaned)

plt.plot(t,data_true,linewidth=2,label='"true" model')
plt.plot(t,data,'.',linewidth=3,label='"noisy" observations')
plt.plot(t,np.matmul(G,model_fit2),linewidth=2,label='model fit')
plt.plot(t,data_new,'.',linewidth=3,
         label='observations with large outliers')
plt.plot(t,np.matmul(G,model_fit_new[0]),linewidth=3,
         label='model fit including outliers')
plt.plot(t,np.matmul(G,model_fit_new2[0]),'c',linewidth=2,
         label='model fit, outliers omitted')
plt.xlabel('"time"',fontsize=14)
plt.ylabel('"data values"',fontsize=14)
plt.legend(fontsize=10)
```



Omitting the questionable data nicely restores our fit to the shape that we expect. You'll notice that although to solve for $\mathbf{m}$, I removed the rows of $\mathbf{G}$ corresponding to bad data, I didn't need to remove them when I computed $\mathbf{Gm}$ to reconstruct my fitted curve.

2. *Weight the least-squares fit so that bad data have less impact on the solution.* A second option is to think about the fact that the least-squares fit minimizes the raw misfit between data and model. In principle, you might suppose that the misfit should be comparable in size to the uncertainty in the data, so we might want to tell the least-squares fit to allow for more misfit for less certain points. In other words, we'd like the misfit for each datum to match the uncertainty: $e_i \approx \sigma_i$. We can do this by weighting each line of our matrix equation by the uncertainty. In the case that we've been examining, that would become:
$$\frac{d_i}{\sigma_i} = \frac{m_0 + m_1 t_i + m_2 t_i^2}{\sigma_i} \tag{33}$$
With the weighting, the misfit should be ~ 1, for each row of our matrix equation, so we can each row equivalently. Formally we implement this by including a weighting in the function that we minimize:
$$\epsilon = (\mathbf{Gm} - \mathbf{d})^T \mathbf{W_e} (\mathbf{Gm} - \mathbf{d}), \tag{34}$$
where
$$\mathbf{W}_e = \begin{bmatrix} \sigma_1^{-2} \\ \sigma_2^{-2} \\ \vdots \\ \sigma_N^{-2} \end{bmatrix} \mathbf{I} \tag{35}$$

(In a more complicated system, with correlated error, $\mathbf{W}_e$ could represent the full covariance matrix for the a priori uncertainty in the data: $\mathbf{W}_e = \langle \mathbf{d}'\mathbf{d}'^T \rangle^{-1}$.) If we implement this we can write:

```
sigma_vector=sigma*ones(size(t));
sigma_vector(100:110)=5*sigma_vector(100:110);
Gweighted = G./sigma_vector;
data_weighted = data./sigma_vector;
model_fit_weighted = Gweighted\data_weighted;
plot(t,G*model_fit_weighted,'r','LineWidth',2);
legend('"true" model','"noisy" observations','model fit',...
  'observations with large outliers','model fit including outliers',...
  'model fit, outliers omitted','weighted model fit','FontSize',14)
```

or

```
import numpy.matlib
sigma_vector=sigma*np.ones([len(t),1])
sigma_vector[100:111]=5*sigma_vector[100:111,:]
Gweighted = G/np.matlib.repmat(sigma_vector,1,3)
data_weighted = data_new/np.squeeze(sigma_vector)
model_fit_weighted = np.linalg.lstsq(Gweighted,data_weighted,rcond=-1)

plt.plot(t,data_true,linewidth=2,label='"true" model')
plt.plot(t,data,'.',linewidth=3,label='"noisy" observations')
plt.plot(t,np.matmul(G,model_fit2),linewidth=2,label='model fit')
plt.plot(t,data_new,'.',linewidth=3,label='observations with large outliers')
plt.plot(t,np.matmul(G,model_fit_new[0]),linewidth=3,
         label='model fit including outliers')
plt.plot(t,np.matmul(G,model_fit_new2[0]),'c',linewidth=2,
         label='model fit, outliers omitted')
plt.plot(t,np.matmul(G,model_fit_weighted[0]),'pink',linewidth=2,
         label='weighted model fit')
plt.xlabel('"time"',fontsize=14)
plt.ylabel('"data values"',fontsize=14)
plt.legend(fontsize=10)
```

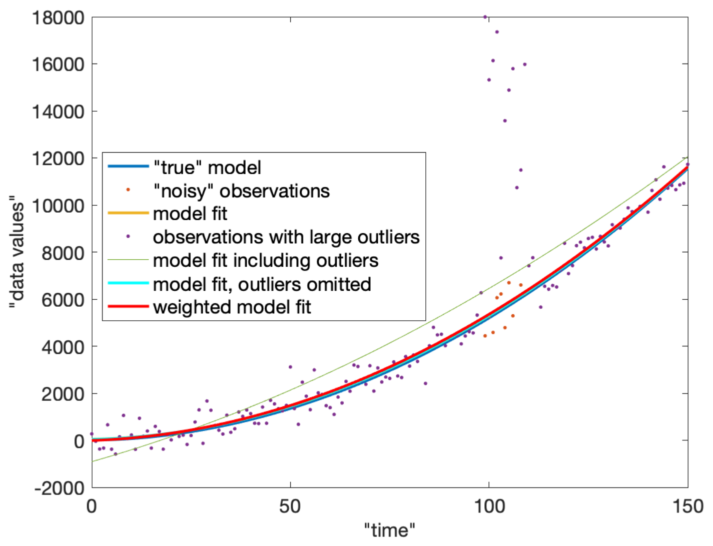
We finished up by commenting briefly on uncertainties. Formally, if you weight by σ, then the covariance matrix for the model coefficients is:

$$\mathbf{C} = (\mathbf{G}^T\mathbf{G})^{-1} \tag{36}$$

and the square root of the diagonal of $\mathbf{C}$ will tell you the uncertainties of your estimates:

```
sqrt(diag(inv(Gweighted'*Gweighted)))
```

More details still to come on this.


"true" model
"noisy" observations
model fit
observations with large outliers
model fit including outliers
model fit, outliers omitted
weighted model fit
"data values"
"time"
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
-2000
0
50
100
150