Matlab Basics
Lecture 4

Juha Kuortti

October 30, 2018

Creating and accessing files `save,load`

Variables are erased from memory after quitting Matlab (`>>quit` or `>> exit`).

- The command `>>save` saves all workspace variables into the file `matlab.mat` in the current directory.
- `>>save A B` saves just `A` and `B`.
- `>>save myfile A B C*` saves `A,B` and all variables starting with `C` into `myfile.mat`. **Note:** If you happen to have a variable called `myfile` in your workspace, then this variable together with the above variables will be stored in `matlab.mat (:-))`.

Files: Loading from a .mat-file

- `>>load` reads the file `matlab.mat` (in current directory or on matlab path) and loads all variables in the workspace, i.e. restores the state of the workspace after the corresponding `save`-command.
- `>>load myfile` does the same with `myfile.mat`.
- `>>load myfile A B` loads just variables `A,B`.

Note: These `.mat`-files are in Matlab's internal format. The next slide treats `ASCII`-file handling.

Important: `save` and `load` can't be used to save your session. Usually, much more important than saving variables, is saving the commands that created those variables, i.e. saving your session. For that you need scripts and `.m` - files.

ASCII Files, `load textmat.dat`

- To create user-readable files, append the flag `-ascii` to the end of a save command.
- **Note:** In this case, MATLAB does not append any extension to the file name, so you may want to add an extension such as `.txt` or `.dat` .

Example: Create a text-file `textmat.dat` outside Matlab.

textmat.dat:

```
1 2 3 4
5 6 7 8
```

```
>> load -ascii textmat.dat
>> % You can omit -ascii here
>> textmat
textmat =
     1     2     3     4
     5     6     7     8
```

ASCII Files, `save -ascii Afile.dat`

Save a Matlab-variable into an ascii-file:

```
>> A=magic(3);
>> save -ascii Afile.dat A
>> % or:  >> save -ascii -double Afile.dat A
>> clear
>> load Afile.dat
>> who
Your variables are: Afile
>> Afile
Afile =
     8     1     6
     3     5     7
     4     9     2
```

Importing data

MATLAB has several specialty functions to import different types of data.

Examples:

- `xlsread` — for reading Excel files
- `csvread` — for reading CSV delimeted data
- `imread` — for reading image data
- `audioread` — for reading audio data

There are countless others. When trying to import data, a good first step is to drag and drop the data file on the desktop, which will lead MATLAB trying to identify the file type.

Exporting data

Almost every "read" function in MATLAB has a corresponding "write" function

Examples:

- `xlswrite` — for writing Excel files
- `csvwrite` — for writing CSV delimeted data
- `imwrite` — for writing image data
- `audiowrite` — for writing audio data

Excercise

- Read the file `gasprices.csv` in the MATLAB (available on the course web page)
- Find all entries in the file that have NaN entries, meaning they have no data, and replace the NaN value with value 1.37 (**Hint:** use `isnan` and logical indexing).
- Plot the data, using column `year` as the x-axis.

Very short introduction to Singular Value Decomposition

Let's go back to matrices for a minute

All data can be thought of as matrices (vectors are matrices too). However if your data has high dimensionality, gleaning patterns can be hard.

Similarily, many interesting problems will lead to linear systems that are not solvable in exact sense.

The fantastic SVD

Both of these can be solved (amongst other things) via using Singular Value Decomposition. The decomposition writest any matrix $\mathbf{A}$ in form

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

where $\mathbf{S}$ is a diagonal matrix, and $\mathbf{U}, \mathbf{V}$ are orthogonal matrices.

Mathematics behind this is a little bit complicated, but one can think of the values in $\mathbf{S}$ (the eponymous "Singular Values") tell how much the corresponding columns of $\mathbf{U}$ and $\mathbf{V}$ contribute to the whole matrix.

Solving equations with SVD

From linear algebra we know that matrix inverse is

$$\mathbf{A}^{-1} = \mathbf{V}\mathbf{S}^{-1}\mathbf{U}^{T}$$

However, if matrix is not invertible there are zeroes on the diagonal of $\mathbf{S}$, making it singular. Likewise, if the matrix is *badly conditioned*, there are very small singular values.

The idea using SVD is that we only invert the non-zero elements of $\mathbf{S}$.

Example

```
M = magic(16); % degree is even so M is singular
b = M*ones(16,1); % let's create a right side for ...
    the equation
x1 = M\b %doesn't work
[u,s,v] = svd(M); % make the svd
semilogy(diag(s),'*') % plot the singular values ...
    against their indices. It would appear that only ...
    first three are meaningful, rest are almost zero
sDiagonal = diag(s); % extract the singular values ...
    into vector
sInverseDiagonal = 1./sDiagonal % invert only the ...
    first three
sInverseDiagonal(4:end) = 0;% set the rest to 0
% now solve the system: remember x =A^-1*b = v*s^-1*u^t
x2 = v*diag(sInverseDiagonal)*u'*b
% If you're in a hurry
x = pinv(M)*b
```

Interpolation

Parameter fitting

Noisy Data

Interpolation

Problem: suppose we have some discrete measurement data, and we wish to construct new evaluation points within the range of those measurements.

This problem is called *interpolation*. Interpolation is a complicated problem that generally has no unique solution — in order for intepolation to provide a meaningful results, usually some *a priori* knowledge is needed.

Interpolation: the tools of trade

Your primary interpolation tools within the MATLAB are the `interp`-functions. They are numbered 1–3, with the number telling the dimensionality of the data.

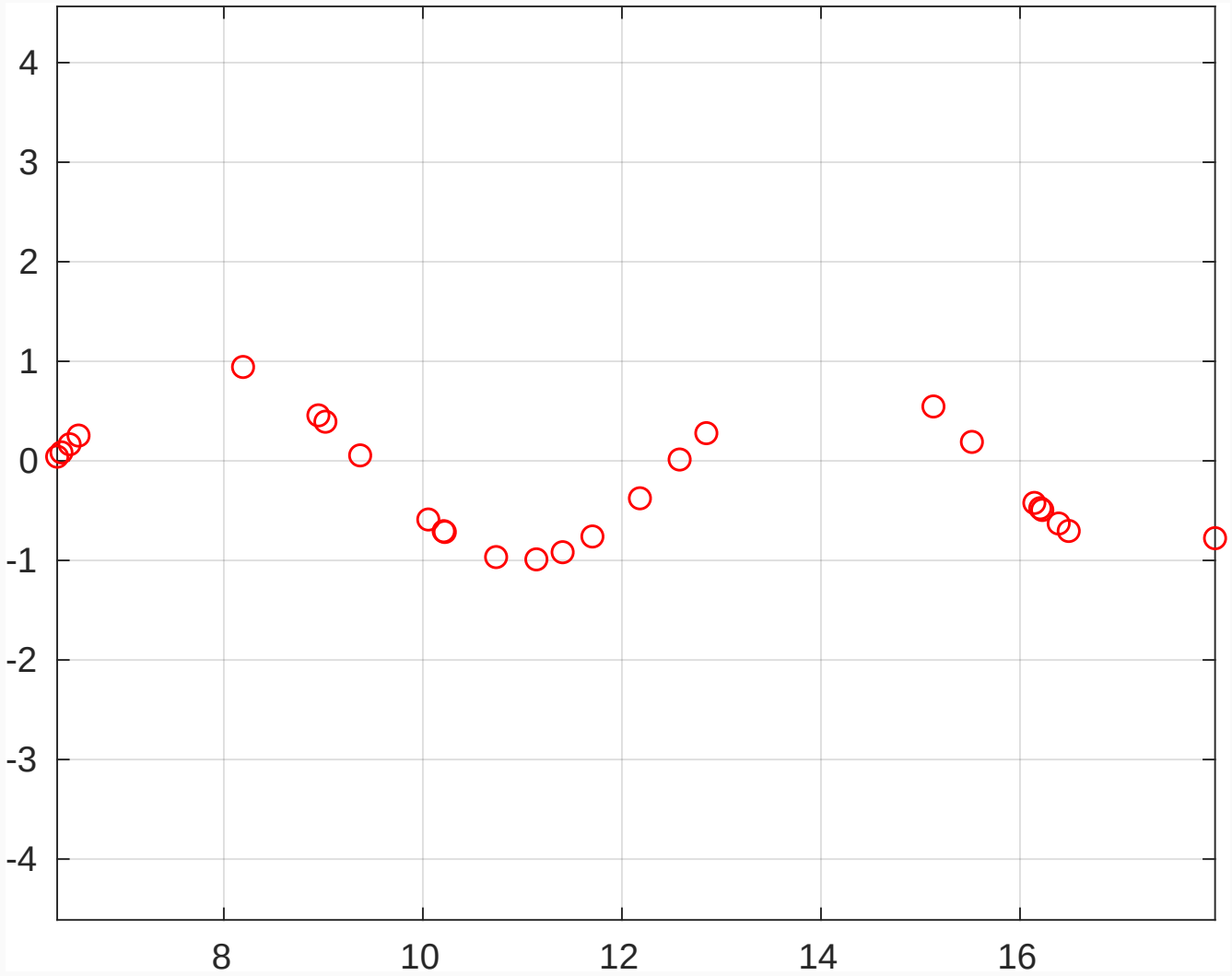
For our examples we'll use 1D data, so `interp1`.

Which is the “best”?

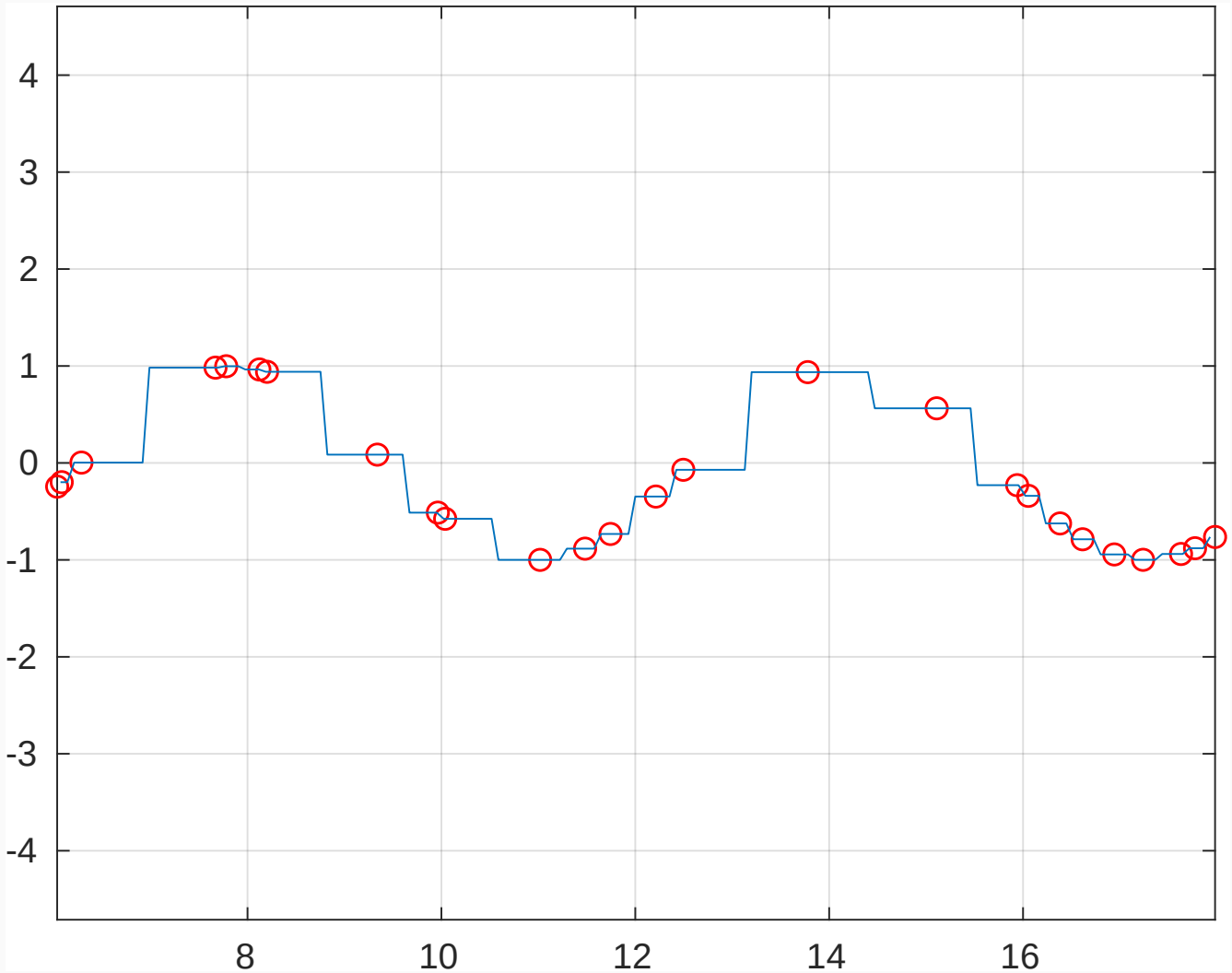
Let us create a sample data by sampling values of a sine function randomly:

```
xpts = 12*rand(26,1)+6;
xpts = sort(xpts);
ypts = sin(xpts);
plot(xpts,ypts,'ro')
axis equal
```

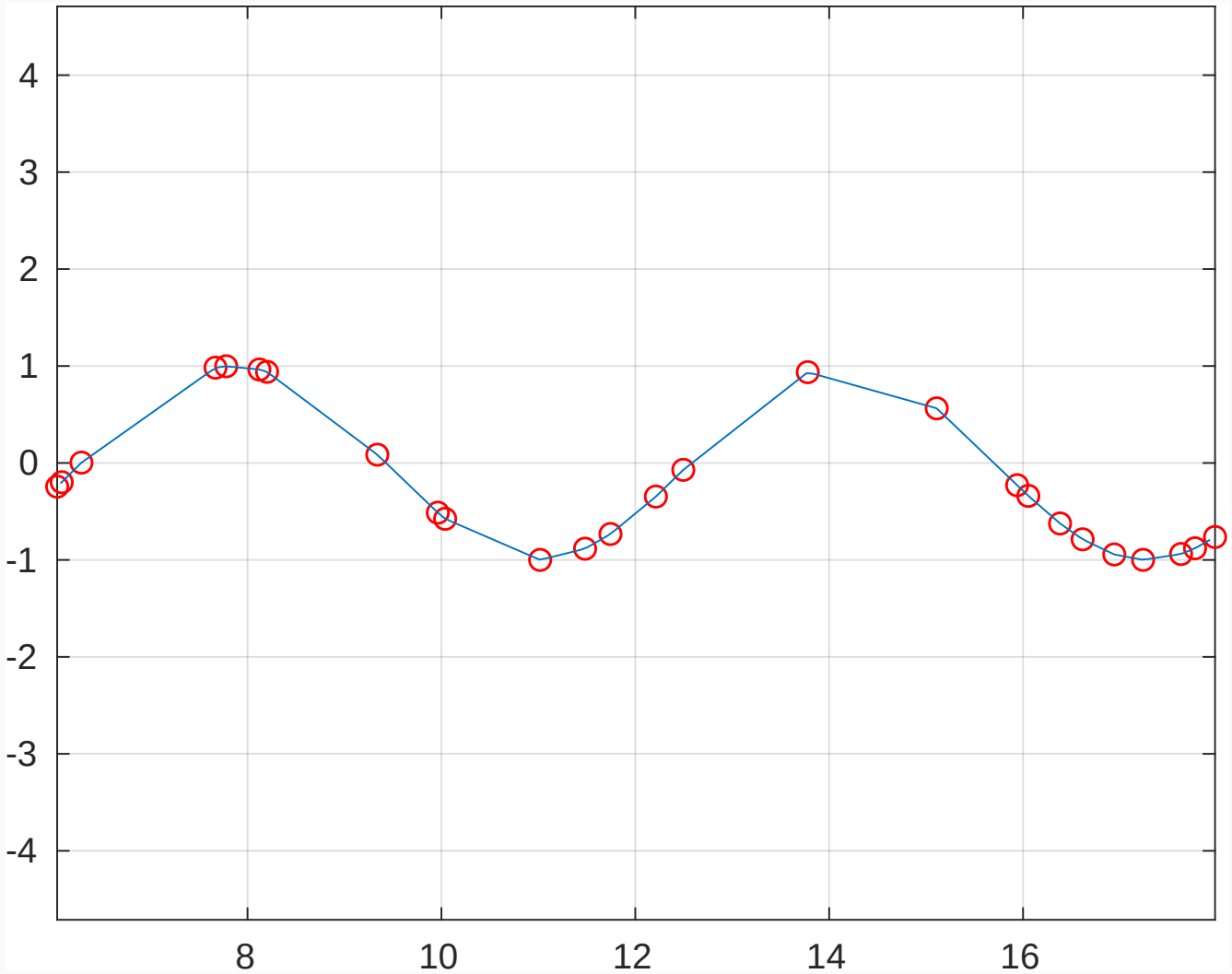
Which is the “best”?



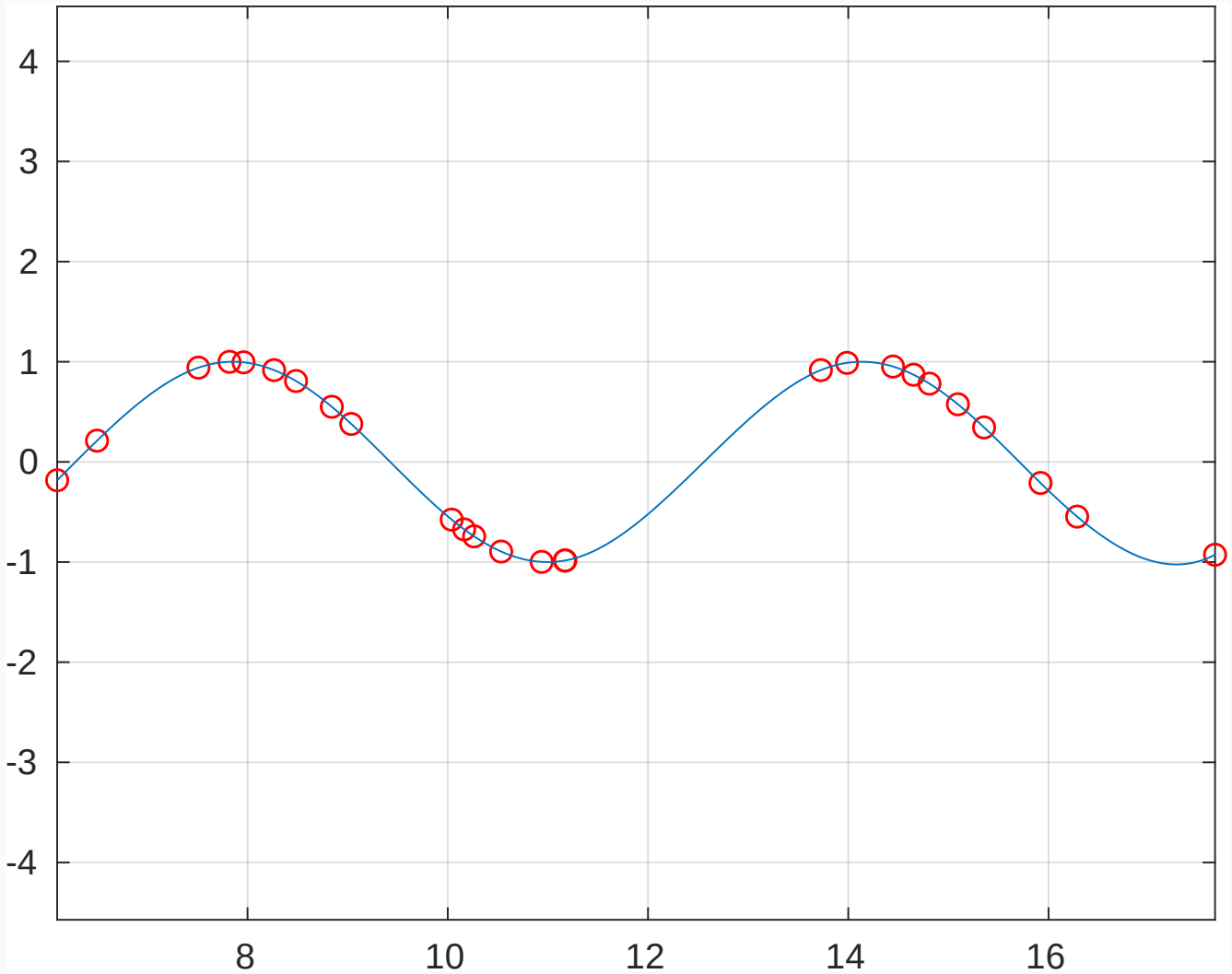
Which is the “best”?



Which is the “best”?



Which is the “best”?



Exercise

Load the mat file `interpData.mat` into MATLAB, and perform interpolation of your choosing on the data. Can you guess the underlying function?

Perils of polynomial interpolation

If you have n data points, you can (technically) fit a $n-1$ degree polynomial to the data.

If you try this on the previous data however

```
p = polyfit(xpts,ypts,45);
x = linspace(xpts(1),xpts(end),128);
plot(x,polyval(p,x));
```

you'll see why this is usually not a good idea.

Perils of polynomial interpolation

Polynomial fit will pass through all the data points, but the oscillations of the high-degree polynomial will make the result unusable.

However, if you can select the points on which you fit, the results can change dramatically:

```
x = linspace(-5,5,13); % datapoints
xx = linspace(-5,5,256); % interpolation points
f = @(x)1./(1+x.^2); % the function fitted for
pp = polyfit(x,f(x))
% Plot
plot(xx,f(xx),xx,polyval(pp,xx))
```

There are solutions, however

```
x = 5*cos(pi*(0:12)./12);
xx = linspace(-5,5,256); % query points
f = @(x)1./(1+x.^2); % the function fitted for
pp = polyfit(x,f(x))
% Plot
plot(xx,f(xx),xx,polyval(pp,xx))
```

Curve fitting

Unlike in interpolation, curve fitting can be thought of as a *minimization*-problem.

You'll want to find a curve that is as close to data as possible within the parameters, but does not necessarily need to pass through every datapoint.

Curve fitting - Example

The very basic example — fitting a trend line on a dataset.

```
    % create some data
x = 5:21;
y = 3.5*x+4 + 4*randn(size(x))+5;
plot(x,y,'ro')
%% the hardcore mathy way
V = [x',ones(size(x'))];
coeff = V\y';
xfit = linspace(x(1),x(end),256);
yfit = coeff(1)*xfit + coeff(2);
plot(x,y,'ro',xfit,yfit)
%% The more consistent way
p = polyfit(x,y,1);
plot(x,y,'ro',xfit,polyval(p,xfit))
```

Underfitting a polynomial

Let's try to fit a polynomial to US census data.

```
% create time vectors -- dense and sparse
t = 1900:10:1990;
tt = 1900:1:2010;
pop = [76 92 106 122 132 150 179 203 226 248];
% for show, try the maximum degree polynomial
P = polyfit(t,pop,length(pop)-1);
plot(tt,polyval(P,tt));
% It fails -- as is expected
P = polyfit(t,pop,3);
plot(tt,polyval(P,tt));
```

Nonlinear fitting

```
x = 20:65;
y = [0 0 0 1 2 5 15 65 71 79 55 48 46 26 25 25 16 9 ...
     18 8 8 6 4 6 5 5 2 6 4 2 0 0 1 1 1 0 1 1 0 0 0
     0 0 2 0 0];

f = @(beta,x)beta(1)*x.^9 .* exp(-beta(2)*x);
fobj = @(lam)norm(f(lam,x)-y);
beta0 = [1 -1];
betaOpt = fminsearch(fobj,beta0);

xfit = linspace(20,65,128);
plot(xfit,f(betaOpt,xfit),x,y)
```

Noise gating — example

```
% Create and visualize the data
t = linspace(0,0.5*pi,256);
signal = sin(2*pi*t) + sin(6*2*pi*t);
plot(t,signal)
%% add some noise
noisySignal = signal + 2*rand(size(signal))-1;
plot(t,signal,t,noisySignal)
%% Go to frequency spectrum
X = fft(signal);
plot(abs(X))
Xnoisy = fft(noisySignal);
plot(abs(Xnoisy))
%% Identify the noise component, and gate it out
Xnoisy(abs(Xnoisy)<30) = 0;
%% Go back to time domain
xnew = real(ifft(Xnoisy));
plot(t,signal,t,xnew)
```