Lectures on Curve Fitting with MATLAB

5.2.2019 *Heikki Apiola*

Mattie/lectures/CurveFitting.pdf

...opetus/materiaali03/nummenet/CurveFitting.tex

1 Curve fitting

General question: *How to approximate a given function with a class of simpler functions.*

1. Interpolation: 1) Polynomial 2) Splines (later)
2. Linear Least Squares approximation: 1) Polynomial, 2) General basis functions
3. Non-linear LSQ
4. Advanced ...

Assume we are given the following table of values:

x_0	x_1	x_2	$\ldots$	x_n
y_0	y_1	y_2	$\cdots$	y_n

Table 1:

The table may consist of some measured data or values y_k of a given function at the given x_k-points

If we know or if there's reason to believe, that the y-values represent values of a smooth [1] function, it may be reasonable to look for a function in a given class of functions that passes through all the data points. This is called *interpolation.*

[1] By *smooth* we mean a function that is at least continuous and has sufficiently many derivatives for the application

In case the measurements are inaccurate or there are other reasons, like lots of data, it is often more reasonable to look for the trend of the data instead, and thus give up the requirement of the model function to exactly pass through the data points. Instead of interpolation we then usually look for *a least squares approximation*.

Typically the class of simpler functions is taken to be polynomials but other "basis functions" are also possible. For instance periodic data is better approximated by trigonometric polynomials.

In addition to linear approximations, where problem parameters appear linearly, there are natural non-linear models, whose solution requires non-linear optimization techniques.

1.1 Polynomial interpolation

Interpolation task:

Given $n+1$ x- and y-values (Table 1), find a polynomial p of degree at most n, that satisfies

$$p(x_k) = y_k, k = 0, \ldots, n.$$

1.1.1 MATLAB-solution using `polyfit`

Let's start by using MATLAB-functions `polyfit` and `polyval` as "black-box" routines. Let's first recall the way MATLAB represents polynomials: A polynomial is the vector of coefficients starting from the most significant (highest power) entry. For instance `c=[1 2 0 -1]` would represent the polynomal $x^3 + 2x^2 - 1$. MATLAB function `polyval` can be used to evaluate the polynomial at given points.

The function `polyfit` takes the `xdata`- and `ydata`- vectors as arguments and gives the coefficient-vector as the result. Here's an example

```
xdata=[-2 -1 3];
ydata=[1 -2 5];
n=length(xdata)-1 % Deg. of polynomial: one less than nr. of points.
coeff=polyfit(xdata,ydata,n)
coeff =
  0.9500   -0.1500   -3.1000
xev=linspace(-2.5,3.5);     % Points, where polynomial is evaluated.
```

```
p=polyval(coeff,xev);       % Values of polynomial at the points
plot(xdata,ydata,'o',xev,p) % Plot data and interpolation polynomial
```

The reader is asked to run the commands in MATLAB or Octave. Check that the polynomial function passes through the datapoints.

Giving the parameter `n` of `polyfit` smaller values, one would get polynomials which would give lower degree LSQ-approximations to the data. We will return to these in a moment.

1.1.2 Solution by linear system of equations

Now we will open the "black box" showing how to return the interpolation problem to that of solving a linear system of equations. This will also give us basic tools and ideas of how to handle more general LSQ-problems besides polynomials.

Let $p(x) = a_0 + a_1 x + \ldots + a_n x^n$.

Note: Now we use the more commonly adopted order of polynomial representation. We will try to make it clear, which order is used, and why we need to use the commands `fliplr, flipud` every once in a while.

There are $n+1$ unknown coefficients $a_0, a_1, \ldots a_n$, and the known data points give us $n+1$ equations $p(x_k) = y_k, k = 0, \ldots n$. So it's reasonable to hope for a unique solution.

Let's start with an example:

Example 1 *Let* $x_0 = -2, x_1 = -1, x_2 = 3$ *and* $y_0 = 1, y_1 = -2, y_2 = 5$. *We are looking for a* 2^{nd} *degree polynomial*

$p(x) = a_0 + a_1 x + a_2 x^2$, *satisfying* $p(x_k) = y_k, k = 0, 1, 2$.

Thus we get the system of equations

$$\begin{cases} a_0 + a_1 x_0 + a_2 x_0^2 = y_0 \\ a_0 + a_1 x_1 + a_2 x_1^2 = y1 \\ a_0 + a_1 x_2 + a_2 x_2^2 = y2 \end{cases}$$

for solving the coefficients a_0, a_1, a_2. In matrix form: $A\,c = y$, where

$$A = \begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix}, y = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}, c = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}.$$

With the above data we could compute in MATLAB:

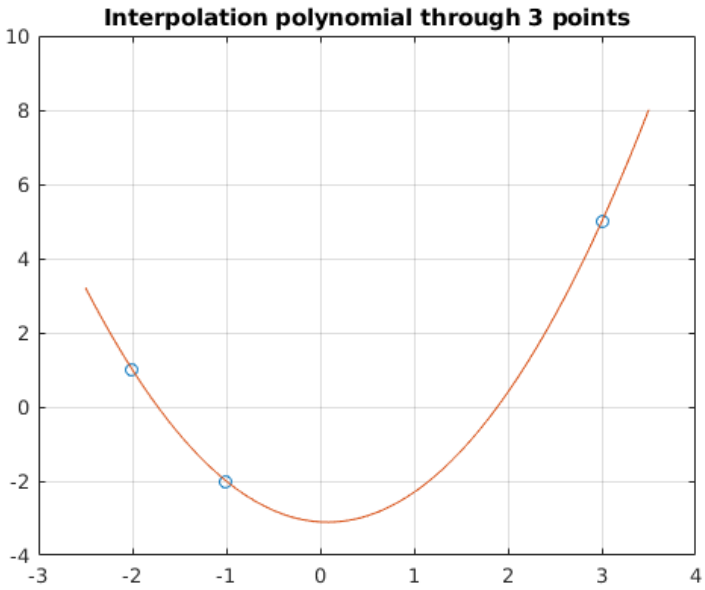
```
>> x=[-2;-1;3];y=[1;-2;5];
>> A=[ones(size(x)) x x.^2]
A =
     1    -2     4
     1    -1     1
     1     3     9
     >> c=A\y
c =
   -3.1000
   -0.1500
    0.9500

>> (A*c)'  % Should give y.
ans =
    1.0000   -2.0000    5.0000
% Yes, it does
```

How do we evaluate the polynomial at an arbitrary set of points? Well, we have just been taught the use of `polyval`, which is of course available here. So `polyval(flipud(c),xev)` would do it for the vector `xev` of points of evaluation. But as we are heading towards cases beyond polynomials, let's instead just use standard matrix algebra to turn a linear combination of vectors into matrix multiplication.

Let $t = [t_1, t_2, \ldots, t_n]'$. The values of the polynomial at the vector t are:

$$p(t) = a_0 \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} + a_1 \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix} + a_2 \begin{bmatrix} t_1^2 \\ t_2^2 \\ \vdots \\ t_n^2 \end{bmatrix} = \begin{bmatrix} 1 & t_1 & t_1^2 \\ \vdots & \vdots & \vdots \\ 1 & t_n & t_n^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$

Note: The coefficient vector is here in the "low-to-up" order.



Vandermonde matrix

The structure of the matrix on the right - call it V - is the same as that of our A in the above MATLAB-session, except the x-datavector is replaced by the (usually much longer) vector `t` of evaluation points:

. |column 1: `ones` | column 2: *tpoints* | column 3: $(tpoints)^2$|.

It is now obvious how to form the interpolation polynomial in the general case of degree n where we have $n+1$ datapoints $x_0, \ldots, x_n$

The matrix that is needed for both solving the polynomial coefficients and for computing the values at selected points has the name *Vandermonde* matrix, and its general form is:

$$V = \begin{bmatrix} 1 & x_0 & \ldots & x_0^n \\ 1 & x_1 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & \ldots & x_n^n \end{bmatrix}.$$

Note: The above matrix is square, it is non-singular as long as the x-datapoints are

distinct, as discussed below. This is the matrix used to solve the coefficients (model parameters) of the interpolation polynomial.
A similar matrix formed replacing the x-data column by the (long) vector of evaluation points (t-column in the example) is used to compute values of the (polynomial) model at selected points.

Here's a continuation of our example showing the evaluation:

```
>> t=(-2.5:.1:3.5)';  % Vector of points where polynomial will be ...
   evaluated.
>> V=[ones(size(t)) t t.^2]; Vandermonde matrix determined by vector t.
>> p=V*c;
>> plot(x,y,'o',t,p); grid on
>> title('Interpolation polynomial through 3 points')
```

MATLAB has the function `vander`, which forms the above matrix with its columns in the reversed order, as discussed. Try for instance `vander(1:5)` . (`vander` accepts its argument as row or column vector.)

Returning to the data of the previous MATLAB session:

```
>> xdata=[-2 -1 3]';
>> vander(xdata)  % vander accepts row or column as argument.
ans =
     4    -2     1
     1    -1     1
     9     3     1
>> fliplr(ans)    % lr means ``Left Right''
ans =
     1    -2     4
     1    -1     1
     1     3     9
```

The reason for MATLAB's "flipping" the columns is the way MATLAB presents polynomials as the vector of coeffcients, **starting** with the **most significant** one, as discussed above. MATLAB has the function **polyval** for evaluating a polynomial with the coefficient vector `c`. **BUT**: For the above reason, the coefficient vector has to be "flipped" to start with the highest degree.

The session continues:

```
>> t=(-2.5:.1:3.5)';  % Vector of points where polynomial will be ...
   evaluated.
>> V=[ones(size(t)) t t.^2];
>> p=V*c;
>> q=polyval(flipud(c),t);  % c is a column vector, hence ``ud''
>> max(abs(p-q))            % Difference of the two computations
ans =
   8.8818e-16               % ... is of order round off unit
>> eps
ans =
   2.2204e-16              % ``machine epsilon''
```

1.1.3 Existence and uniqueness of solution

In mathematics, these are the basic questions. Here everything boils down to the question of non-singularity of the Vandermonde matrix.

As everyone knows, there is a unique solution provided $\det(V) \neq 0$ for the Vandermonde matrix V. It is possible to derive a nice formula for the determinant as a product of differences of the x_k-points, thus showing that it's different from zero as long as the x_k-points are distinct.

However, we don't need to do this exercise, as the existence can be proved by a clever direct construction due to *Lagrange* (or an alternative one by *Newton*), and uniqueness follows from basic properties of polynomials.

All details are shown here: interpolation16.pdf

Note 1 *A Vandermonde matrix (with distinct $x_0, \ldots, x_n$) is non-singular, but it gets more and more* ***ill-conditioned*** [2] *as n increases.*

Thus the method of *Lagrange* mentioned above, or its suitable variations are better ways to construct the interpolation polynomial.

[2] Briefly: small errors in data cause large errors in results

Exercises

Problem 1 *Write a function* `vandinterp`*:*

```
function coeff = vandinterp(xdata,ydata)
% Call: coeff = vandinterp(xdata,ydata)
% Return coefficient vector of interpolation polynomial to xdata,ydata
% Example:
% xdata=0:5; ydata=xdata.*sin(xdata);c=vandinterp(xdata,ydata);
% ...
```

It is instructive to write the code for Vandermonde matrix, alternatively you can use the function `vander`*.*

Your first test can be the help example above.

Continue writing a script where you evaluate the polynomial at a reasonably dense set of points on the interval `[min(xdata-.25`*, max(xdata)+.25]. Plot the data and the interpolation polynomial. Pay attention to the polynomial passing through the datapoints.*

Also, pay attention to the order of the coefficient vector/columns of the Vandermonce matrix.

1.2 Least squares solution

The term *Least squares* (LSQ) refers to solving an overdetermined system of equations or inexactly specified datafitting task so that the residual erros will be minimized in the euclidean norm (i.e. the sum of squares of the residuals will be minimized). In the case of an overdeterminded linear system it means to find a solution vector c that minimizes the residual $||y - Ac||$, where A is an $m \times n$ matrix with $m > n$ and y is the datavector of y values. Think of A being a matrix which depends on the `xdata` (like the Vandermonde-matrix).

1.2.1 Least squares polynomial using polyfit

Let's start again by "black box" computing techniques using our old friends `polyfit` and `polyval`.

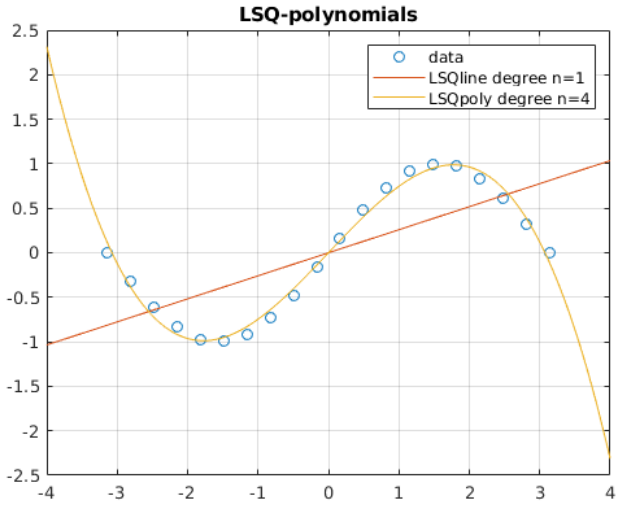
Recall the calling sequence: `c=ployfit(xdata,ydata,n)`. In case of interpolation, the

degree $n = length(xdata) - 1$, which gives an exact solution, i.e. makes the residual error 0. Choosing n smaller we will get the best approximation in the LSQ sense among polynomials of degree $\leq n$.

Here's a script you can write, run and modify on MATLAB or OCTAVE.

Example 1 .

```
%% Example 1 on LSQ
% example1LSQ.m
clear;close all;format compact
%%
x=linspace(-pi,pi,20); % x-data
y=sin(x);              % y-data
c1=polyfit(x,y,1);     % coeffs of LSQ-line (name it p1)
c4=polyfit(x,y,4);     % coeffs of LSQ-polynomial of deg 4 (p4)
t=linspace(-4,4);      % Evaluation points (100)
y1=polyval(c1,t);      % Values of polynomial p1 at t-points
y4=polyval(c4,t);      % Values of polynomial p4 at t-points
plot(x,y,'o',t,y1,t,y4);grid on;shg
title('LSQ-polynomials')
legend('data','LSQline degree n=1','LSQpoly degree n=4')
```



1.2.2 Linear LSQ-fit, general basis functions

The general linear curve fitting problem:
Let $t \to y(t)$ be a function dependent on some parameters, that we want to determine so that the function approximates the given data $(x_i, y_i), i = 0, \ldots, m$.

In the linear case we assume the form:

$$y(t) = c_0\phi_0(t) + \ldots + c_n\phi_n(t),$$

where the $\phi_k(t)$-functions are assumed to be linearly independent.

The *design matrix* X is the $m \times n$ matrix:

$$x(i, j) = \phi_j(x_i), i = 0, \ldots, m, j = 0, \ldots n.$$

In case of polynomial approximation the basis functions are the monomials: $\phi_j(t) = t^j$ and the design matrix X consists of the first n colums of the Vandermonde matrix (last n of MATLAB's `vander`). We know already that the colums of a Vandermonde matrix are linearly independent on the interval under consideration.

Our assumption $m > n$ means that the design matrix X has more rows than colums, thus the system of (approximate) equations: $Xc \approx y$ is overdetermined.

If we multiply the equation from the left by X^T, we get the system of equations called the **normal equations**:

$$(X^T X)c = X^T y,$$

whose solution c is the LSQ-solution This can be proved by an orthogonality argumet or using partial derivatives wrt. the parameters. We will skip it here, references ... ***ref***Fo-Ma-Mo p. 201-202, neat, short.

The normal equations are not a reliable way with large data, as the condition number $cond(X^T X)$ is of the order $cond(X)^2$. In the case of a low degree polynomial the matrix $X^T X$ is small, though, for instance in case of an LSQ-line it is only 2×2, no matter how much data we have. But certaily even a small ill-conditioned matrix can produce poor results.

The good news is that MATLAB's *backslash* (\) does the job for us. So the syntax of solving the approximate equation is the same as solving a square system of linear equations: `c=X\y`

In fact, this algorithm does not just solve the "normal equations":$(X^TX)c = X^Ty$. Instead it works the numerically more reliable way of using the s.k. *QR-factorization*, which makes a difference especially with large problems. See **ref**Moler LSQ 5.5. The QR-Factorization.

Example 2 *Find a function model of the form* $F(x) = a\cos\pi x + b\sin\pi x$ *to approximate the data*

```
x = -1 -.5  0  .5  1
y = -1  0   1  2   1
```

in the LSQ-sense.

Plot the data and the LSQ-approximation.

Perhaps the model needs some adjustment. Try to include a constant term: $G(x) = a + b\cos\pi x + c\sin\pi x$.

Include the plot of G in your figure.

Solution:

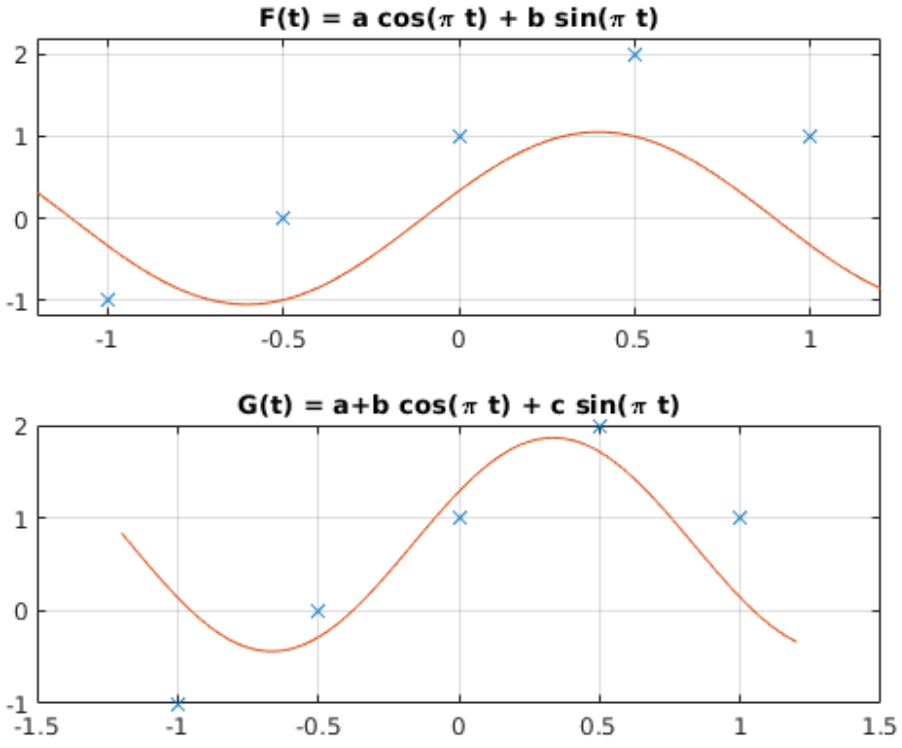
```
%% Example on LSQ with cos and sin - basisfunctions
% example1LSQsincos.m
clear;close all;format compact
%% Data:
x=[-1 -.5 0 .5 1]';
y= [-1 0 1 2 1]'
subplot(1,2,1)
plot(x,y,'x');axis([-1.2 1.2 -1.2 2.2]);shg
%% (a)  Only cos- and sin-terms
V=[ cos(pi*x) sin(pi*x)]   % Design matrix (``generalized Vandermonde'')
c=V\y                      % LSQ-solution gives parameters a,b.
t=linspace(min(x)-.2, max(x)+.2)'; % Points of evaluation
V=[cos(pi*t) sin(pi*t)];        % Design matrix at evaluation points
F=V*c;                          % Values of model F at t-points
hold on
plot(t,F);grid on
title(['F(t) = a cos(\pi t) + b sin(\pi t)'])
shg
```

It looks like a constant term is needed, let's do it.

```
%% (b)  $G(t)=a + b\cos \pi t + c\sin \pi t
V=[ones(size(x)) cos(pi*x) sin(pi*x)]  % Design matrix
c=V\y                                  % LSQ-parameters
V=[ones(size(t)) cos(pi*t) sin(pi*t)]  % Design matrix at t-points
G= V*c;                                % Values of model G at t-points
subplot(1,2,2)
plot(x,y,'x',t,G);grid on
title(['G(t) = a+b cos(\pi t) + c sin(\pi t)'])
```

As the following figures show, the extended model looks better.

Note: It would be a bit more elegant to define the design matrix V as a function: `V=@(u)[ones(size(u)), cos(pi*u), sin(pi*u)]`. Then you would just write `c=V(x)\y; G=V(t)*c;`



Here's a link to the above MATLAB-session run in the "Live editor" mode and exported into pdf: example2LSQsincosLIVE.pdf

1.3 Summary on linear curve fitting

Exercises

1.4 Nonlinear Least Squares

1.4.1 More examples of typical models, linear and non-linear

1.4.2 LSQ with SVD, singular case, non-uniqueness

***ref** Fo-Ma-Mo p. 220 -

Exercises

Example 3 *Fröberg p. 338 Routine LSQ-fit example*

Example 4 *Cheney-Kincaid p. 530 probl 19*

Find LSQ-appr: $f(x) = a\,\cos \pi x + b \sin \pi x$
for the data

```
x = -1 -.5  0  .5  1
y = -1  0   1   2   1
```

Example 5 *Moler NCM 5.7.*

Example 6

1.5 More advanced topics

doc vander Run MATLAB Functions with Distributed Arrays R2018b Hundreds of functions in MATLAB® and other toolboxes are enhanced so that they operate on distributed arrays. Distributed arrays are well suited for large mathematical computations, such as large problems

of linear algebra. You can also use distributed arrays for big data processing. For more information on distributing arrays, see Distributing Arrays to Parallel Workers.

Check Distributed Array Support in Functions If a MATLAB function has distributed array support, you can consult additional distributed array usage information on its function page. See Distributed Arrays in the Extended Capabilities section at the end of the function page.

You can also browse distributed support for functions, and filter by product. On the Help bar, click Functions and select a product. In the function list, at the bottom of the left pane, select Distributed Arrays.

For information about updates to individual distributed-enabled functions, see the release notes.

To check support for special distributed data types, consult the following sections.

Support for Sparse Distributed Arrays