OPEN SOURCE ENGINEERING

openeering

A SCILAB PROFESSIONAL PARTNER

powered by ENGINSOFT

DATA FITTING IN SCILAB

In this tutorial the reader can learn about data fitting, interpolation and approximation in Scilab. Interpolation is very important in industrial applications for data visualization and metamodeling.

Level 

www.openeering.com

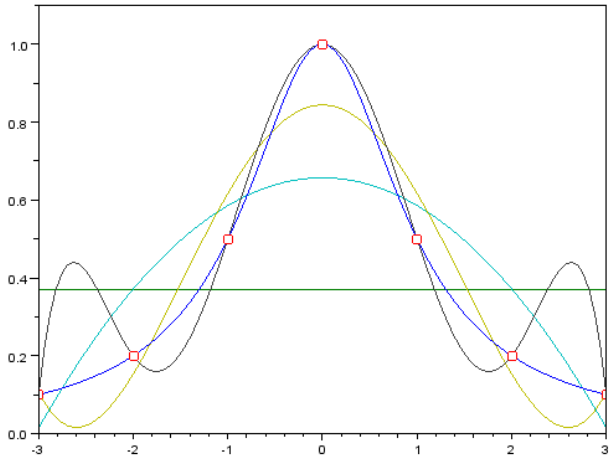
Step 1: Purpose of this tutorial

Many industrial applications require the computation of a fitting function in order to construct a model of the data.

Two main data fitting categories are available:

- Interpolation which is devoted to the development of numerical methods with the constraint that the fitting function fits exactly all the interpolation points (measured data);
- Approximation which is devoted to the development of numerical methods where the type of function is selected and then all parameters are obtained minimizing a certain error indicator to obtain the best possible approximation.

The first category is useful when data does not present noise, while the second one is used when data are affected by error and we want to remove error and smooth our model.



Step 2: Roadmap

In the first part we present some examples of polynomial interpolation and approximation. After we propose exercises and remarks.

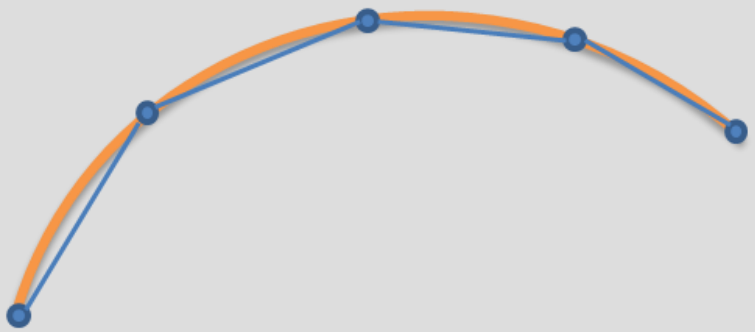
Descriptions	Steps
Interpolation	3-16
Approximation or curve fitting	17-20
Notes	21-22
Exercise	23
Conclusion and remarks	24-25

Step 3: Interpolation

The idea of approximation is to replace a function $f(x)$ with a function $g(x)$ selected from a given class of approximation function Φ.

Two main cases exist:

- ***Continuous function***: In this case the function $f(x)$ is known analytically and we want to replace it with an easier function, for example we may want to replace a complex function with a polynomial for which integration or differentiation are easy;
- ***Discrete function***: In this case only some values of the function $f(x)$ are known, i.e. $f(x_i)$ and we want to make a mathematical model $g(x)$ which is close to the unknown function $f(x)$ such that it is possible to establish the value of $f(x)$ outside the known points.



Example of approximation of a continuous function using a piecewise approximation

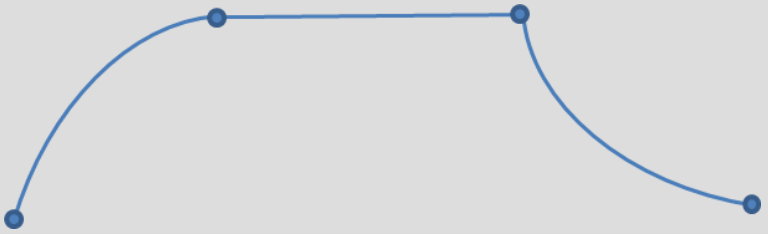
Step 4: Main class of interpolation function

Several families Φ of interpolation functions exist. The most common are:

- ***Polynomial interpolation of degree n***: In this case, we approximate data with a polynomial $P_n(x)$ of degree n of the form

$$P_n(x) = a_0 + a_1 x + \cdots + a_n x^n$$

- ***Piecewise polynomial***: In this case the interval is subdivided into subintervals in which we define a polynomial approximation of low degree with or without continuity on the derivatives between each subinterval.



Example piecewise interpolation with first derivative continuous and discontinuous connections

Step 5: Test case: Runge function

The Runge function is defined as

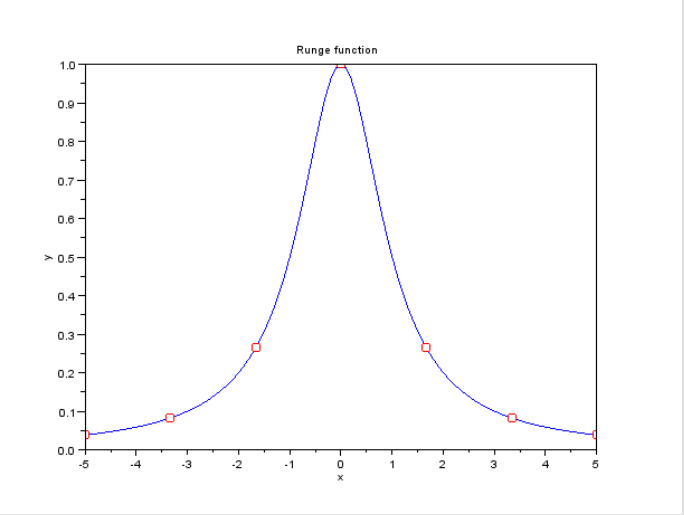
$$y = f(x) = \frac{1}{1+x^2}$$

Typically is considered in the interval [-5,5].

In our examples we consider 7 interpolation points, denoted with (x_i, y_i), equally distributed in the interval [-5,5].

The following code implements the Runge function and perform visualization.

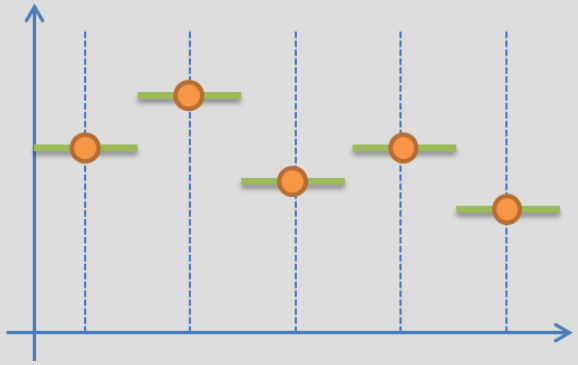
```
// Define Runge function
deff('[y]=f(x)','y = 1 ./(1+x.^2)');

// Interpolation points
xi = linspace(-5,5,7)'; yi = f(xi);

// Data
xrunge = linspace(-5,5,101)'; yrunge = f(xrunge);

// Plot Runge function
scf(1); clf(1);
plot(xrunge,yrunge,'b-');
plot(xi,yi,'or');
xlabel("x");
ylabel("y");
title("Runge function");
```



Interpolation function

Step 6: Piecewise constant interpolation

Piecewise constant interpolation is the simplest way to interpolate data. It consists on locating the nearest data value and assigning the same value to the unknown point.

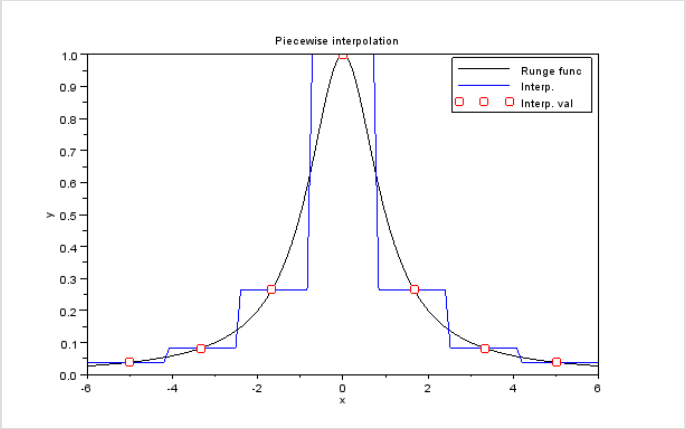


Piecewise constant interpolation

Step 7: Piecewise constant interpolation in Scilab

The Scilab command used to perform piecewise interpolation is "`interp1`" where the third argument is "`nearest`". The fourth argument specifies if an extrapolation method should be used when the evaluation points are outside the interval of the interpolation points.

```
// Interpolation
// Evaluation points
xval = linspace(-6,6,101)';

xx_c = xval;
yy_c = interp1(xi,yi,xx_c,'nearest','extrap');

// Plot
scf(3);
clf(3);
plot(xrunge,yrunge,'k-');
plot(xx_c,yy_c,'b-');
plot(xi,yi,'or');
xlabel("x");
ylabel("y");
title("Piecewise interpolation");
legend(["Runge func";"Interp.";"Interp. val"]);
```

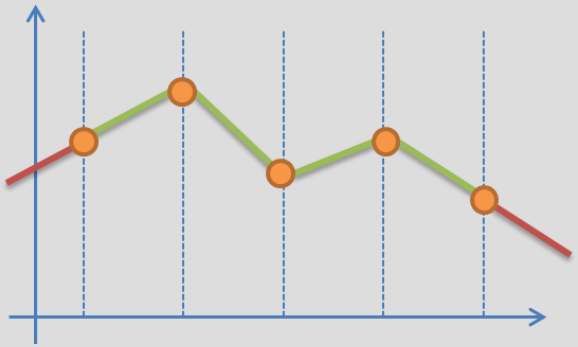


Piecewise constant interpolation

Step 8: Piecewise linear interpolation

Linear interpolation is a polynomial of degree 1 that connects two points, (x_1, y_1), and (x_2, y_2) the interpolant is given by

$$y = y_1 + (y_2 - y_1) \cdot \frac{(x - x_1)}{(x_2 - x_1)}$$

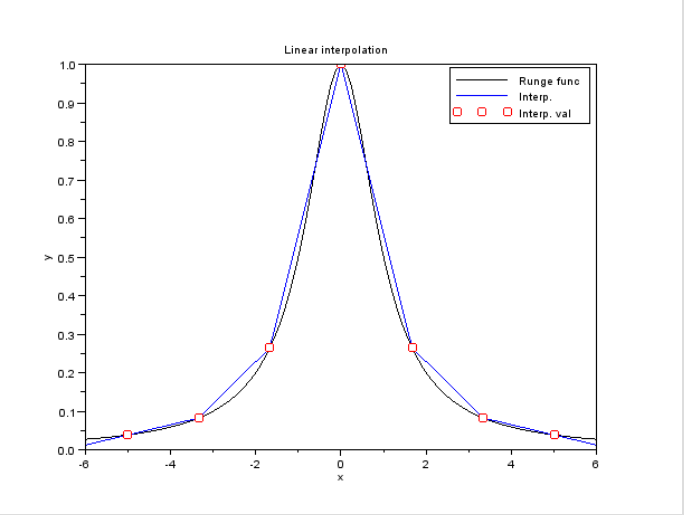


Piecewise linear interpolation (green) and extrapolation (red)

Step 9: Linear interpolation in Scilab

The Scilab command used to perform linear interpolation is again "`interp1`" but now the third argument is "`linear`". We can note that the code is similar to the previous one for piecewise interpolation.

```
// Interpolation
xx_l = xval;
yy_l = interp1(xi,yi,xx_c,'linear','extrap');

// Plot
scf(4);
clf(4);
plot(xrunge,yrunge,'k-');
plot(xx_l,yy_l,'b-');
plot(xi,yi,'or');
xlabel("x");
ylabel("y");
title("Linear interpolation");
legend(["Runge func";"Interp.";"Interp. val"]);
```

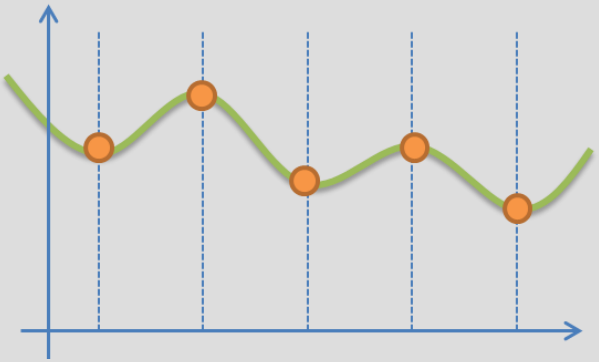


Piecewise linear interpolation

Step 10: Polynomial interpolation

Given a set of $n+1$ data points (x_i, y_i), where all x_i are different, we find the polynomial of degree n which exactly passes through these points.

Polynomial interpolations may exhibits oscillatory effects at the end of the points. This is known as ***Runge phenomenon***. For example, the Runge function has this phenomenon in the interval [-5,+5] while in the interval [-1,1] this effect is not present.



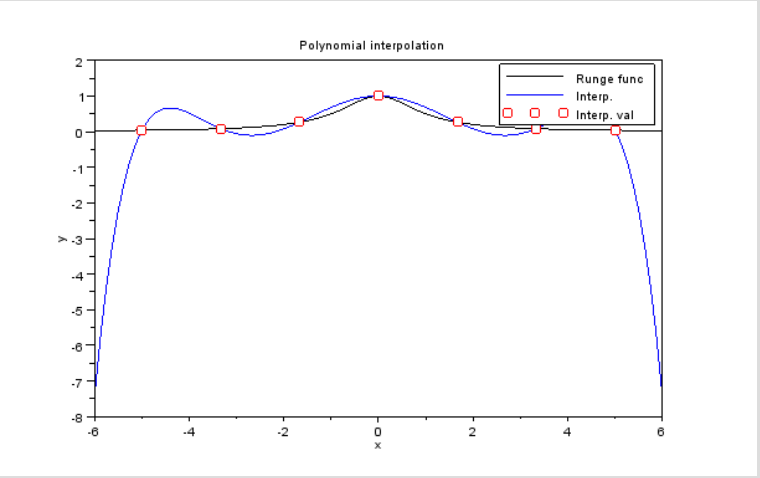
Polynomial interpolation

Step 11: Polynomial interpolation in Scilab

The Scilab command used to perform polynomial interpolation is "`polyfit`" which is included in the zip file (see references). The syntax requires the interpolation points (x_i, y_i) and the degree of the polynomial interpolation which is equal to the number of point minus one.

Note that the command "`horner`" evaluates a polynomial in a given set of data.

```
// Import function
exec("polyfit.sci",-1);
// Interpolation
xx_p = xval;
[Pn] = polyfit(xi, yi, length(xi)-1);
yy_p = horner(Pn,xx_p);

// Plot
scf(5);
clf(5);
plot(xrunge,yrunge,'k-');
plot(xx_p,yy_p,'b-');
plot(xi,yi,'or');
xlabel("x");
ylabel("y");
title("Polynomial interpolation");
legend(["Runge func";"Interp.";"Interp. val"]);
```

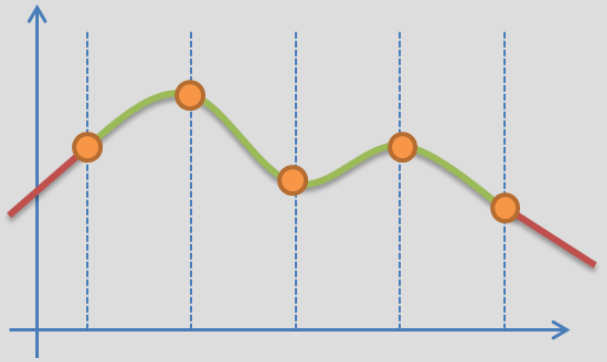


Polynomial interpolation

Step 12: Cubic spline interpolation

Cubic spline interpolation uses cubic polynomials on each interval. Then each polynomial is connected to the next imposing further continuity equations for the first and second derivatives.

Several kinds of splines are available and depend on how degrees of freedom are treated. The most well-known splines are: natural, periodic, not-a-knot and clamped.

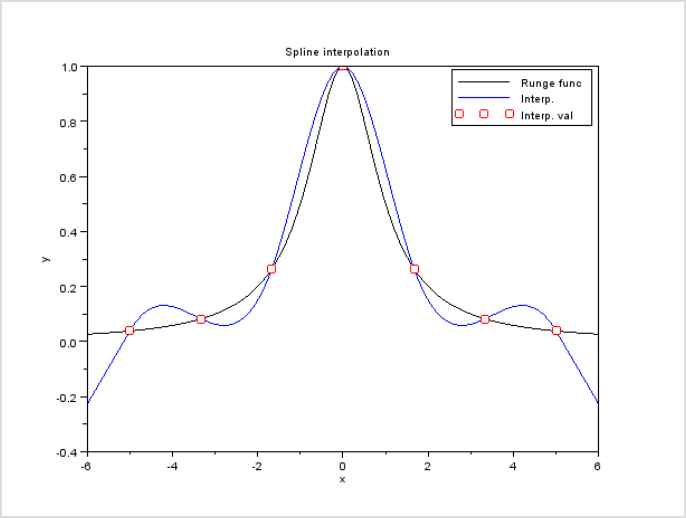


Spline interpolation

Step 13: Cubic spline in Scilab

The Scilab command used to perform cubic spline interpolation is "`splin`". Several types of splines exist and can be specified by setting the third argument of the function. The evaluation of the spline is done using the command "`interp`" where it is possible to specify the extrapolation strategy.

```
// Splines examples
d = splin(xi, yi,"not_a_knot");
// d = splin(xi, yi,"natural");
// d = splin(xi, yi,"periodic");

xx_s = xval;
yy_s = interp(xx_s, xi, yi, d, "linear");

// Plot
scf(6);
clf(6);
plot(xrunge,yrunge,'k-');
plot(xx_s,yy_s,'b-');
plot(xi,yi,'or');
xlabel("x");
ylabel("y");
title("Spline interpolation");
legend(["Runge func";"Interp.";"Interp. val"]);
```



Cubic spline interpolation with linear extrapolation

Step 14: Radial Basis Interpolation (RBF)

Radial basis function modeling consists of writing the interpolation function as a linear combination of n basis functions $\varphi(x)$ that depends only on the distance $\|x - x_i\|$ of the interpolation points x_i. This is equal to:

$$y(x) = \sum_{i=1}^{n} a\varphi(\|x - x_i\|)$$

Using the interpolation conditions $y(x_i) = y_i$ we have to solve the following linear system

$$[\Phi][a] = [y_i]$$

where the element $\Phi_{i,j} = \varphi(\|x_j - x_i\|)$.

Many radial basis functions exist, the most famous are:

- Gaussian:

$$\varphi(r) = e^{-\frac{r^2}{2\sigma}}$$

- Multiquadratic:

$$\varphi(r) = (\epsilon^2 + \mathrm{r}^2)^{\frac{1}{2}}$$

- Inverse multiquadratic:

$$\varphi(r) = \left(\frac{1}{\epsilon^2 + r^2}\right)^{\frac{1}{2}}$$

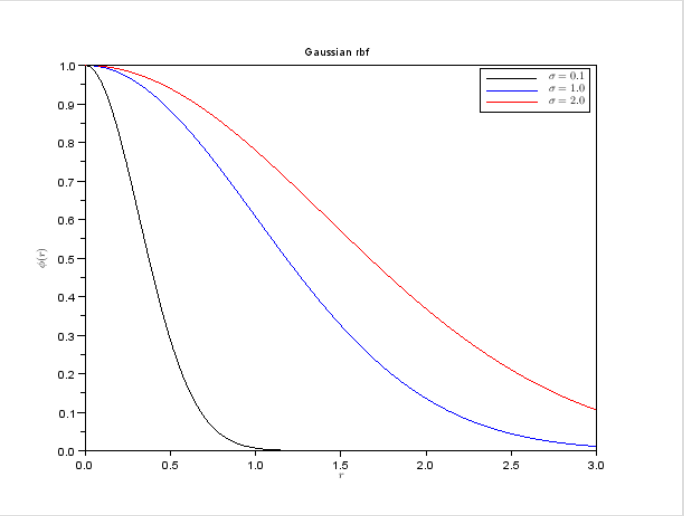
- Thin plate spline:

$$\varphi(r) = r^2 \ln(r)$$

Step 15: Gaussian RBF in Scilab

In our example we use the Gaussian RBF.

```
// Gaussian RBF
deff('[y]=rbf_gauss(r,sigma)','y = exp(-r.^2 ./(2*sigma))');

// Plot
scf(7);
clf(7);
r = linspace(0,3);
y1 = rbf_gauss(r,0.1);
y2 = rbf_gauss(r,1.0);
y3 = rbf_gauss(r,2.0);
plot(r,y1,'k-');
plot(r,y2,'b-');
plot(r,y3,'r-');
xlabel("$r$");
ylabel("$\phi(r)$");
title("Gaussian rbf");
legend(["$\sigma = 0.1$";"$\sigma = 1.0$";"$\sigma = 2.0$"]);
```



Gaussian RBF for different value of sigma

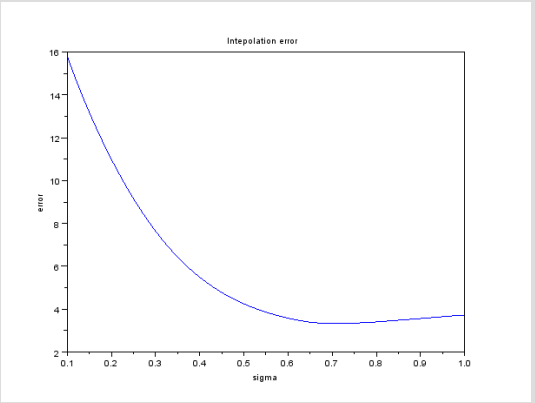
Step 16: RBF in Scilab

On the right we report the optimal Gaussian RBF obtained for the interpolation of the Runge function where we have optimized the σ parameters. The full code is reported in the Openeering web site.

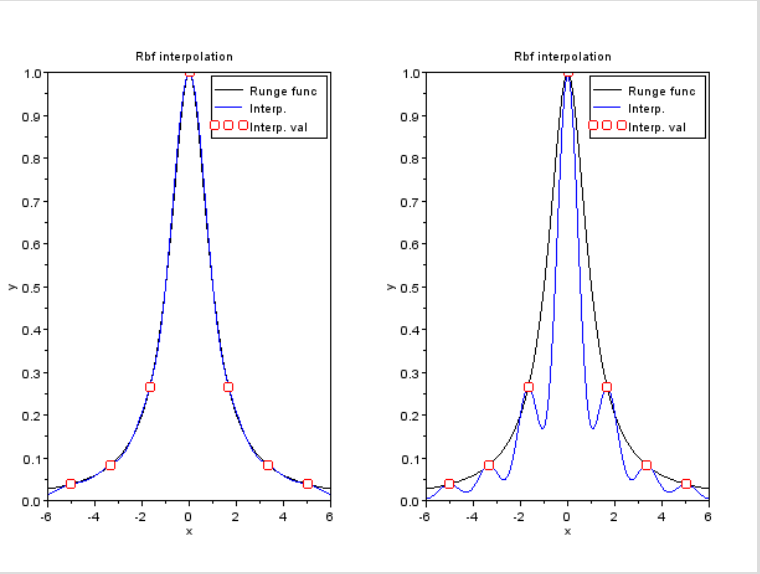
Generally, each radial basis function depends on a parameter and this parameter is known as modeling parameter. This parameter has effect on the oscillation behavior of the function and the optimal choice is not an easy task. Many techniques are available for finding the best modeling parameter, the most famous is probably the “leave one out”.

The leave-one-out cross-validation (**LOOCV**) consists in using a single point from the original set of data as a validation data. The validation of the model is given by that point. This process can be repeated for all the points in the data set such that, in the end, all the points are used once as validation point. This method can produce a mean value of all these leave-one-out errors and gives a global estimate of the model.

Since a value that estimates the model is available, we can use an optimization solver for finding the best parameter in order to minimize the error of the model.



Behaviour of the error versus sigma



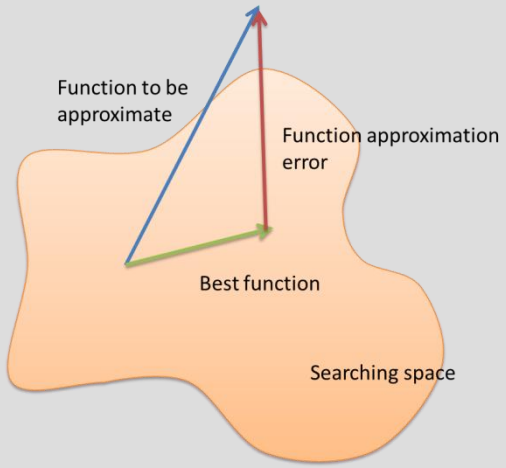
Optimal RBF with $\sigma \approx 0.523$ *(left), non optimal solution* $\sigma \approx 0.2$ *(right)*

Step 17: Approximation or curve fitting

When data is affected by errors, polynomial interpolation cannot be appropriate since the approximation function is constrained to be through the interpolation points. So it makes sense to fit the data starting from a given class of functions and minimizing the difference between the data and the class of functions, i.e.

$$\min_{\phi \in \Phi} \|f - \phi\|$$

The "`polyfit`" function computes the best least square polynomial approximation of data. If we choose $n = 1$ in the "`polyfit`" function, we approximate data with linear function of the form $P(x) = a_0 + a_1 x$, i.e. we compute the linear least squares fitting.



Schematic example of min interpretation

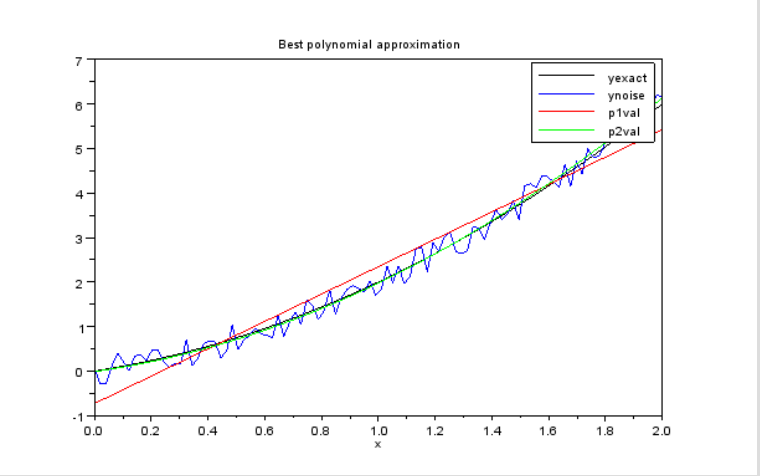
Step 18: 1D approximation

In this example we add noise to the function $f(x) = x^2 + x$ and then we make polynomial approximation of order 1 and 2.

The critical code is reported here (only case 1):

```
np = 100; noise = 0.7*(rand(np,1)-0.5);
x = linspace(0,2,np)';
yexact = x.^2 + x;
ynoise = yexact + noise;

// degree 1 approximation
p1 = polyfit(x, ynoise, 1);
p1val = horner(p1,x);
scf(10); clf(10);
plot(x,yexact,'k-'); plot(x,ynoise,'b-'); plot(x,p1val,'r-');
```

For details download the zip file with the source codes.



Comparison of best fitting for degree 1 and 2

Step 19: 2D approximation

In this example we want to approximate scattered data with a linear least square fitting in two dimensions.

Given the polynomial approximation $P(x,y) = a + bx + cy$ and the n scattered interpolation points (x_i, y_i, z_i), the optimal computation of the unknown parameters (a, b, c) requires to solve the following overdetermined linear system

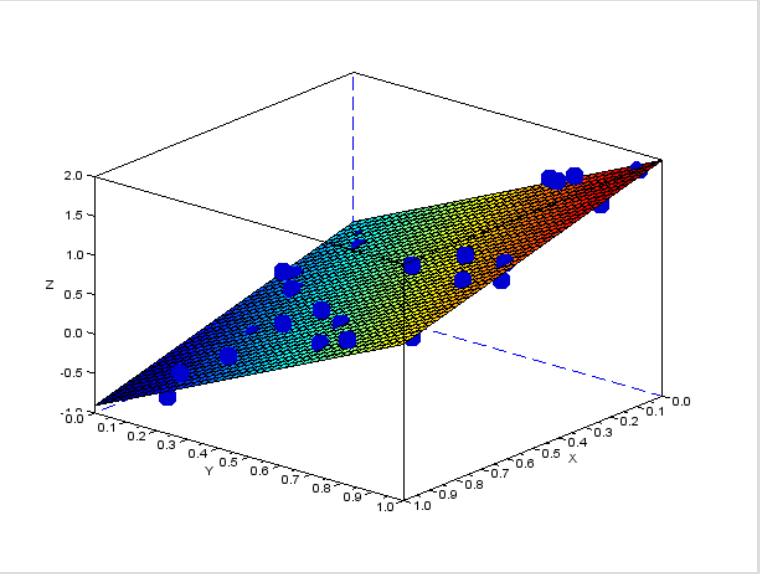
$$\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & y_n \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}$$

The matrix is known as the Vandermonde matrix and arises in polynomial interpolation. The i-th row of the matrix corresponds to the polynomial evaluated at point i-th. In our case, the i-th row corresponds to:

$$1 \cdot a + x_i \cdot b + y_i \cdot c = z_i$$

This is done using the Singular Value Decomposition (SVD) or equivalently using the backslash command. Here, we report only the solution stage. The full code can be downloaded from our web page.

```
// Generating random points along a plane
np = 30;
noise = 0.5*(rand(np,1)-0.5);

// Extract data
x = rand(np,1);
y = rand(np,1);
znoise = -x+2*y+noise;

// Vandermonde matrix for P(x,y) = a+b*x+c*y
V = [ones(np,1),x,y];
// Find coefficient i.e. minimize error norm
coeff = V\znoise;
```



Example of least square approximation in 2D

Step 20: nD linear approximation

The previous example can be easily extended to an n-dimensional problem, i.e. we search for an approximation of the form

$$P(x_1, x_2, \dots, x_n) = a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_n x_n.$$

Here, we report the code for the problem where some coefficients may be equal to zero and could be deleted from the estimation problem.

The idea here implemented consists of the following strategy:

1. Perform least square approximation of data;
2. Find the zero coefficients with some pre-fixed tolerance;
3. Re-perform least square approximation of the reduce problem where we have deleted the columns of the Vandermonde matrix corresponding to the zero coefficients.

The code reported on the right estimates these coefficients. The full code can be downloaded from our website.

```
n = 10;          // Problem dimensions
neval = 100;     // Number of evaluation points
pcoeff = 1:n+1; // Problem coefficient ([a0, a1, ..., an])
pcoeff([2,5,8]) = 0; // Some zero coefficients

// Evaluation points
[xdata,ydata] = generatedata(pcoeff, n, neval);

// Estimate coefficients
pstar = estimatecoeff(xdata, ydata);

// Find "zero" coefficient (define tolerance)
tol = 0.1;
zeroindex = find(abs(pstar)<=tol);

// re-estimate coefficients
pzero = estimatecoeffzero(xdata, ydata, zeroindex);
```

The code use the following functions:

- `y=evaldata(pcoeff,x)`
  that evaluates the polynomial $P(x_1, x_2, \dots, x_n)$ defined by coefficients $pcoeff = [a_0, a_1, \dots a_n]$ on point $x = [x_1, x_2, \dots, x_n]$ adding a uniform noise on the definition of the coefficients;
- `[xdata,ydata]=generatedata(pcoeff,n,neval)`
  that generates the evaluation points and their values;
- `pstar=estimatecoeff(xdata,ydata)`
  that estimates the polynomial coefficients using the least square method;
- `pzero=estimatecoeffzero(xdata,ydata,zeroindex)`
  that estimates the polynomial coefficients where some coefficients, specified by the vector zeroindex , are equal to zero.

Step 21: Another polyfit function

The numerical solution of the polynomial interpolation problem requires to solve the linear system with the Vandermonde matrix. Numerically, this problem is ill-conditioned and requires an efficient strategy for a "correct" solution. This is performed by using, for example, QR factorization.

The function "`polyfit_fulldemo.sce`", provided by ***Konrad Kmieciak***, and contained in the zip file, is an alternative to the `polyfit` function.

```
function [u]=polyfit(x, y, n)
      // Vandermonde
      for k=n:-1:0
          if k==n then w=0;
          end
          w=w+1;
          Xu(:,w)=[x.^k];
      end

      // QR
      [q r k]=qr(Xu,'0');
      s = inv(r) * (q' * y); // s = r \ (q' * y)
      for o=1:length(s)
          u(find(k(:,o)>0))=s(o);
      end
endfunction
```

Step 22: Industrial applications of data fitting

Interpolation and approximation are two major techniques for constructing mathematical models. Mathematical models can substitute complex model in real-case applications.

When the original model is complex, or when it requires long and costly evaluations (for example with finite element analysis – FEA), a simplified model of the original model is required.

The model of the model is often called ***metamodel*** (a.k.a. ***response surfaces***) and the metamodeling technique is widely used in industrial applications.

In real-case applications, when optimizing product or services, we need to evaluate several times the model. This means that having a metamodel that can be evaluated faster is, most of the time, the only way for finding optimal solutions.

Metamodeling
+
Response surface
+
Self-Organizing Maps
+
Neural Networks
=
Industrial applications

Step 23: Exercise on exponential decay

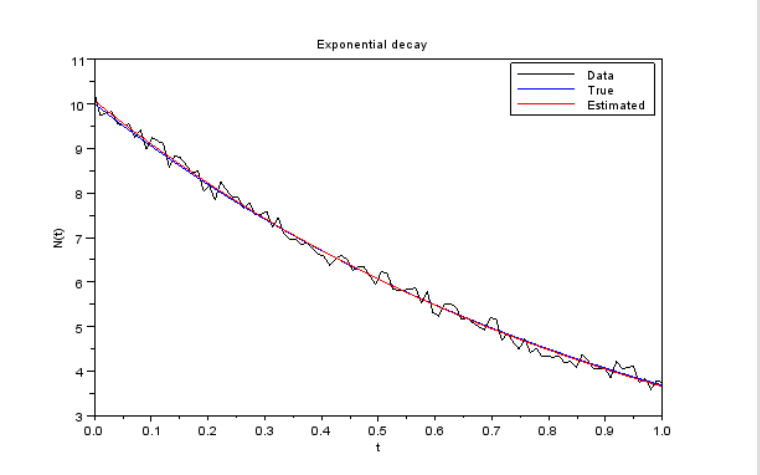
As an exercise, try to estimate the coefficient of a decay function of the form:

$$N(t) = N_0 \cdot e^{-\lambda t}$$

using a linear least square fitting of data.

Hits: Use a logarithmic change of variable to reduce the problem in the following form

$$\log N(t) = \log N_0 - \lambda t$$



Exponential decay fitting of data

Step 24: Concluding remarks and References

In this Scilab tutorial we have shown how to apply data fitting in Scilab starting from piece-wise interpolation, presenting polynomial fitting and cubic spline, and ending up with radial basis functions (RBF).

In the case of polynomial interpolation we used the function "`polyfit`" which can be downloaded from the Scilab webpage and is included in the provided source codes.

1. Scilab Web Page: Available: www.scilab.org.
2. Openeering: www.openeering.com.
3. Javier I. Carrero is the author of the polyfit function. The original code is available on the Scilab.org web pages and is included in the zip file.

Step 25: Software content

To report bugs or suggest improvements please contact the Openeering team. We thank Konrad Kmieciak for reporting us a new version of the polyfit function.

www.openeering.com

Thank you for your attention,

Manolo Venturin
Silvia Poles

```
--------------
Main directory
--------------
ex0.sce                      : Plotting of the first figure
ex1.sce                      : Solution of exercise 1
interpolation.sce            : Interpolation examples codes
polyfit.sci                  : Polyfit function
polyfit_manual.pdf           : Polyfit manual
polyfit fulldemo.sce         : Another version of polyfit
estimate lincoeff.sce        : Estimantion of nD linear model
license.txt                  : The license file
```