Novel algorithm for multi dimensional regression of polynomes with arbitrary coefficients by means of the method of least squares

Serge Zihlmann

ThirdWay

October 7, 2010

1 Abstract

The adjustment of polynomes to a given set of data is one of the most common methodes of regression. Simple problems can easly be solved by use of standard programs. Though, if userdefined polynomes are required, the algorithms have to be definitely programmed since the system of equations become complex quite fast. The case is even more demanding if the polynome depends of multiple variables.

In this framework exponent matrices are applied which are built up systematically and represent the structure of the system of equations. Once these matrices are built as many datapoints as demanded can be included.

The novel algorithm solves two problems at the same time. First the desired polynome can easily be prompted in the shape of a potence matrix. Thus it can be adjusted within seconds. Furthermore the dependence of multiple variables is handled with ease and it's fast[1].

[1] 1.45 seconds for 55'696 datapoints in 2 dimensions with 15 coefficients in Scilab

Contents

2 Introduction

There are plenty of algorithms which calculate polynome coefficients to a given set of data with the method of least squares using systems of linear equations. Many of them are faster and also able to have a domain of many dimensions. This work doesn't claim to enhance this situation. However, the weak point of many of these algorithms is that the desired polynomes with their coefficients have to be constantly included in the code or that the coefficients have to follow a given pattern in order the algorithm is able to set up the system of equations. Conventional algorithms are for example intended to adjust polynomes like

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_nx^n \tag{2.1}$$

to the data. A polynome with a defined domain in two dimensions coud have the following shape:

$$f(x;y) = a_0 + a_1x + a_2y + a_3xy + a_4x^2y + a_5xy^2 + a_6x^2y^2... \tag{2.2}$$

In the second example often the highest potence can be entered. The coefficients will then be built up according the preliminary pattern. None of the coefficients can then be taken out what is often desired. Furthermore the algorithms build up their systems of equations quite unclear.

The following algorithm eplained in this work is an easy solution to only calculate *particular* coefficients. The construction of the system of equations would not anymore be possible applying present algorithms. An example delivers the following polynome:

$$f(x;y) = a_0 + a_1x + a_2y^{27} \tag{2.3}$$

The coefficients don't follow a pattern. Equations are complicated to be built up.

3 Description of the issue and solution

3.1 Method of least squares

A very demonstrative example delivers the following polynome defined in two dimensions x & y with 4 coefficients. A dataset consisting of x, y, z values must be given.

$$f(x;y) = a_0 + a_1 x + a_2 x y^2 + a_3 y^{27} \tag{3.1}$$

The error between a measured value z_i and the regressed polynome is given as (i: ith value):

$$r_i = f(x_i;y_i) - z_i \tag{3.2}$$

On summation negative and positive regression errors are compensating. Thus the signum of the error has to be eliminated. Therefore the error r is squared. Disadvantegeous is that strong runaway values of a measurement count more.

$$e_i = r_i^2 = (f(x_i;y_i) - z_i)^2 \tag{3.3}$$

The total error is yielded by summation of all partial errors over all values i.

$$E = \sum_{i=1}^{n} e_i = \sum_{i=1}^{n} (f(x_i;y_i) - z_i)^2 \tag{3.4}$$

By insertion the original polynome:

$$E = \sum_{i=1}^{n} (a_0 + a_1 x_i + a_2 x_i y_i^2 + a_3 y_i^{27} - z_i)^2 \tag{3.5}$$

3.1.1 Minimisation of the total error

The error E has to become the lowest possible value in dependance of the coefficients $a_{0..3}$. Thus the partial derivations of E with respect to the coefficients has to be zero.

From this the equations below follow:

$$\frac{\partial E}{\partial a_0} = \sum_{i=1}^{n} 2 \cdot (a_0 + a_1 x_i + a_2 x_i y_i^2 + a_3 y_i^{27} - z_i) \cdot (1) = 0 \tag{3.6}$$

$$\frac{\partial E}{\partial a_1} = \sum_{i=1}^{n} 2 \cdot (a_0 + a_1 x_i + a_2 x_i y_i^2 + a_3 y_i^{27} - z_i) \cdot (x_i) = 0 \tag{3.7}$$

$$\frac{\partial E}{\partial a_2} = \sum_{i=1}^{n} 2 \cdot (a_0 + a_1 x_i + a_2 x_i y_i^2 + a_3 y_i^{27} - z_i) \cdot (x_i y_i^2) = 0 \tag{3.8}$$

$$\frac{\partial E}{\partial a_3} = \sum_{i=1}^{n} 2 \cdot (a_0 + a_1 x_i + a_2 x_i y_i^2 + a_3 y_i^{27} - z_i) \cdot (y_i^{27}) = 0 \tag{3.9}$$

All equations equal to zero. Hence the factor 2 can be eliminated. By expanding and moving the z_i factor on the other side:

$$a_0 \sum_{i=1}^{n} 1 + a_1 \sum_{i=1}^{n} x_i + a_2 \sum_{i=1}^{n} x_i y_i^2 + a_3 \sum_{i=1}^{n} y_i^{27} = \sum_{i=1}^{n} z_i \tag{3.10}$$

$$a_0 \sum_{i=1}^{n} x_i + a_1 \sum_{i=1}^{n} x_i^2 + a_2 \sum_{i=1}^{n} x_i^2 y_i^2 + a_3 \sum_{i=1}^{n} x_i y_i^{27} = \sum_{i=1}^{n} z_i x_i \tag{3.11}$$

$$a_0 \sum_{i=1}^{n} x_i y_i^2 + a_1 \sum_{i=1}^{n} x_i^2 y_i^2 + a_2 \sum_{i=1}^{n} x_i^2 y_i^4 + a_3 \sum_{i=1}^{n} x_i y_i^{29} = \sum_{i=1}^{n} z_i x_i y_i^2 \tag{3.12}$$

$$a_0 \sum_{i=1}^{n} y_i^{27} + a_1 \sum_{i=1}^{n} x_i y_i^{27} + a_2 \sum_{i=1}^{n} x_i y_i^{29} + a_3 \sum_{i=1}^{n} y_i^{54} = \sum_{i=1}^{n} z_i y_i^{27} \tag{3.13}$$

In matrices notation:

$$\sum_{i=1}^{n} \begin{pmatrix} 1 & x_i & x_i y_i^2 & y_i^{27} \\ x_i & x_i^2 & x_i^2 y_i^2 & x_i y_i^{27} \\ x_i y_i^2 & x_i^2 y_i^2 & x_i^2 y_i^4 & x_i y_i^{29} \\ y_i^{27} & x_i y_i^{27} & x_i y_i^{29} & y_i^{54} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \sum_{i=1}^{n} z_i \begin{pmatrix} 1 \\ x_i \\ x_i y_i^2 \\ y_i^{27} \end{pmatrix} \tag{3.14}$$

A common solution is to include these equations into an algorithm what's complicating the change of the amount of coefficients during runtime.

3.2 New method for building up the system of equations

The algorithm intruduced in this work is still based on the system of equations in the last section. However, it proposes an easy method to build up these equations. The algorithm doesen't take care of solving the equations in the end by the elimination method.

The elements of the matrix and the vector have to be determined so the matrix equation 3.14 can be solved for the coefficients. Therefore the measured values in the different dimensions have to be exponentiated with the right power, afterwards they are multiplicated and after all a sumation over all values is needed (see eqn. 3.14). This whole procedure must be implemented into code as compact as possible.

3.2.1 Exponent-matrices

The method uses the fact that multiplicated factors sum their exponents. The algorithm builds up tables in advance with the exponents for the different variables for each matrix element. So they can easily be accessed during calculation of the particular elements of the matrix. If the equation 3.14 is considered and in new matrices respectively new vectors of the same size the *exponents* for x and y are assigned, the following is obtained:

$$x : \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{3.15}$$

$$y : \begin{pmatrix} 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \\ 2 & 2 & 4 & 29 \\ 27 & 27 & 29 & 54 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 2 \\ 27 \end{pmatrix} \tag{3.16}$$

If these tables were given at the beginning, it was easy to set up the equation 3.14. For each element and for all measured values the exponents in the tables would be used to exponentiate x_i und y_i. These factors would then be multiplicated and the sum for all values would be built.

3.2.2 Sum of exponents

The question remains how to abtain these table with the exponents in advance. For each coefficient one equation can be obtained, thus the matrix is always square and possesses one solution. Because the equations are not differentiated for x or y the exponents won't change. Once again the original polynome respectively the second derivation of the total error:

$$f(x;y) = a_0 + a_1 x + a_2 x y^2 + a_3 y^{27} \tag{3.17}$$

$$\frac{\partial E}{\partial a_1} = \sum_{i=1}^{n} 2 \cdot (a_0 + a_1 x_i + a_2 x_i y_i^2 + a_3 y_i^{27} - z_i) \cdot (x_i) = 0 \tag{3.18}$$

It's easy realised that the derivations possess always the same pattern. Sigma sign, factor 2, the original polynome minus the z-value and a factor depending on the actual coefficient differentiated with respect to. Thus the exponents have always got the pattern of the polynome combined with the actual coefficient. The exponents for x and y in the polynome are assigned to a vector:

$$x : \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} ; x : \begin{pmatrix} 0 \\ 0 \\ 2 \\ 27 \end{pmatrix} \tag{3.19}$$

Due to the polynome always shaping the main structure, a square matrix is filled with the obtained x and y exponents.

$$x : \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} ; y : \begin{pmatrix} 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \end{pmatrix} \tag{3.20}$$

By the derivation the ith line is multiplied with the according x/y pair of the ith coefficient. Thus the figures (exponents) of the matrices and vectors can be added up entry-wise. From this follows that:

$$x : \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} +^{1} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \tag{3.21}$$

$$y : \begin{pmatrix} 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \end{pmatrix} +^{2} \begin{pmatrix} 0 \\ 0 \\ 2 \\ 27 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \\ 2 & 2 & 4 & 29 \\ 27 & 27 & 29 & 54 \end{pmatrix} \tag{3.22}$$

The right side of the system of equations 3.14 is originally only consisiting of the z-factor. Due to this fact the exponents for x and y are directly obtained from the exponents of the polynome. Means:

$$x : \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} y : \begin{pmatrix} 0 \\ 0 \\ 2 \\ 27 \end{pmatrix} \tag{3.23}$$

This way the exponent tables can be built up quite simple.

[1] Entry-wise

[2] Entry-wise

3.3 Application

In this section the step-by-step construction of the algorithm is explained.

The x-, y-, and z-values have to be fed into the program code. Furthermore the code has to be supplied with the polynome. The easiest way is to supply the x and y exponents of the polynome in a dual line array. For the given polynom

$$f(x;y) = a_0 + a_1 x + a_2 x y^2 + a_3 y^{27} \tag{3.24}$$

the following polynome representing array is obtained:

$$\begin{pmatrix} \text{x-exponents} \\ \text{y-exponents} \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 27 \end{pmatrix} \tag{3.25}$$

Now the exponent matrices must be built up for x and y by extracting the first and second line of the array. The lines are combined to an array with as many lines as coefficients avaiable (square matrix). Following the basic matrices:

$$x : \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} ; y : \begin{pmatrix} 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \\ 0 & 0 & 2 & 27 \end{pmatrix} \tag{3.26}$$

Now the first line of the obtained matrices is extracted and entry-wise added to the corresponding matrix according 3.22 and previous. The exponents for the solution vector of the equation 3.14 (right side) are the vectors just extracted (or the transponated polynome exponents).

$$x : \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} , y : \begin{pmatrix} 0 \\ 0 \\ 2 \\ 27 \end{pmatrix} \tag{3.27}$$

To finally set up the matrix according eqn. 3.14 for every single entry the sum with the measured values must be built. Therefore the exponent matrices are used as lookup table and the particular x_i and y_i value has to be exponentiated with the right power. This x_i^a is now multiplicated with y_i^b where a and b are the exponents in the tables and i ist the actual datapoint. This must be done with every single datapoint while the sum is built. For the vector (right side of equation), the z_i value must be multiplied as well. Once the matrix and the vector is set up, the matrix equation is solved by the Gaussian elimination method.

3.4 Generalisation

3.4.1 Expansion to several dimensions

A function is often depending on more than one variable. The function is then defined in n dimensions (domain).

$$f(D_1, D_2, ..., D_n) \tag{3.28}$$

The principle with the exponent matrices can easily be expanded to many dimensions. For each new dimension a new exponent matrix respectively a vector is needed. Furthermore the polynome representing matrix (see 3.25) has to be completed with a further line of exponents of the actual dimension.

A demonstrative delivers the calculation of the specific resistance of a cylindrical conductor. The resistance is:

$$R(l,d) = \rho\frac{l}{A} = \rho\frac{4 \cdot l}{d^2\pi} = \underbrace{4\frac{\rho}{\pi}}_{C} l \cdot d^{-2} \tag{3.29}$$

The resistance is measured for different values of length and thickness. The factors 4, π and ρ are combined to one constanc C. This factor can now be determined by the regression. The polynome would be prompted to the algorithm in the following pattern:

$$\begin{pmatrix} 1 \\ -2 \end{pmatrix} \tag{3.30}$$

3.4.2 Negative exponents

Fractional polynomes can easily be implemented. Example:

$$f(x,y) = a_0 + a_1 x + a_2\frac{y}{x^2} + a_3 x^2 y + a_4\frac{1}{y^4} \tag{3.31}$$

The representing matrix for the polynome:

$$\begin{pmatrix} \text{x-exponents} \\ \text{y-exponents} \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & -2 & 2 & 0 \\ 0 & 0 & 1 & 1 & -4 \end{pmatrix} \tag{3.32}$$

Attention: If negative exponents are used, the measured values must not become zero in any dimension. Else divisions by zero would be generated.

3.4.3 Real exponents

As described in the last section, the algorithm can be run with any exponents. This is also valid for *non integer* values. Example:

$$f(x, y) = a_0 + a_1 x + a_2 \frac{y^{0.5}}{x^2} + a_3 x^{2.79} y^{-0.9} + a_4 \frac{1}{y^{4.1}} \tag{3.33}$$

The representing matrix for the polynome:

$$\begin{pmatrix} 0 & 1 & -2 & 2.79 & 0 \\ 0 & 0 & 0.5 & -0.9 & -4.1 \end{pmatrix} \tag{3.34}$$

3.5 Limitation to linear systems

The desired coefficients are only in linear dependance allowed. The following example is not possible:

$$f(x) = (a_0 x)^2 + sin(a_1 x) + a_2 x + a_2 x^2 \tag{3.35}$$

The last two addends use the same coefficient.

3.6 Computing time

The computation time is increasing with the square of the figure of desired coefficients, because the equations form square matrices. With increasing number of dimensions the work is only rising linear. The algorithm is fast anyway. In Scilab[3] it's using only 1.45 seconds for 55'696 datapoints in 2 dimensions with 15 coefficients.

[3]see proposed algorithm at the end

4 Algorithm suggestion

4.1 Basic proposed Scilab algorithm

The free program Scilab implies a large amount of functions. It's particular designed for matrix operations. Thus the algorithm can be implemented in very short code.

The algorithm must be supplied with three variables. The independent position of the measured value called 'data', the according measured value 'zval' and the polynome. The function must be called by 'PowerFit(data,zval,polynome)'. The data has to be the following shape:

$$data = \begin{array}{c} \text{dimensions} \rightarrow \\ \begin{pmatrix} D_{1,1} & D_{2,1} & \dots & D_{d,1} \\ D_{1,2} & D_{2,2} & \dots & D_{d,2} \\ \vdots & \ddots & & \\ D_{1,n} & D_{2,n} & \dots & D_{d,n} \end{pmatrix} \end{array}, \; zval = \begin{pmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_n \end{pmatrix}$$

n means the number of measured values. d is the number of dimensions where the polynome is defined. The polynome must be prompted in the following shape:

$$polynome = \begin{pmatrix} P(D_1,1) & P(D_1,2) & \cdots & P(D_1,k) \\ P(D_2,1) & P(D_2,2) & \cdots & P(D_2,k) \\ \vdots & \ddots & & \\ P(D_d,1) & P(D_d,2) & \cdots & P(D_d,k) \end{pmatrix}$$

Whereas the index D_i describes in which dimension the exponent is working. k describes the number of coefficients of the polynome, d describes the number of dimensions.

```
//------------------------------------------------------------------------
// This algorithm calculates regressions polynomes of various
// dimensions, coefficients and any combination of potences.
// Released 22. September 2010 by Serge Zihlmann
// THIS CODE IS FREE TO BE USED IN ANY KIND OF SOFTWARE
// For further questions contact: powerfit( at@at )thirdway.ch
//------------------------------------------------------------------------
//------------------------------------------------------------
function [y] = PowerFit(data, zval, polynome)
```

```
10
11    [ndata dim] = size(data);        // Get amount of datapoints
12
13    // Get number of free variables of polynome
14    [dim nCoeff] = size(polynome);  //Number of dimensions and coefficients
15
16    // Generate exponent tables
17    ExpMat = zeros(nCoeff,nCoeff,dim);
18    for i=1:nCoeff
19      for k=1:dim
20        ExpMat(i,:,k) = polynome(k,:)+polynome(k,i);
21      end
22    end
23    ExpVec = polynome';
24
25    // Generate matrices to fill
26    A = zeros(nCoeff,nCoeff); // Final matrix
27    b = zeros(nCoeff,1);      // Final solution vector
28
29    // Build up matrix A
30    for i=1:nCoeff
31      for k=1:nCoeff
32        t = ones(ndata,1);        // Temporary vector
33        for r=1:dim
34          t = t .* (data(:,r).^ExpMat(i,k,r));
35        end
36        A(i,k) = sum(t);
37      end
38    end
39
40    // Build up vector b
41    for i=1:nCoeff
42        t = zval;
43        for k=1:dim
44          t = t .* (data(:,k).^ExpVec(i,k));
45        end
46        b(i) = sum(t);
47    end
48
49    // Solve the system A*y=b for y
50    y = inv(A)*b; // Calculate solution, return parameters
51  endfunction
52  //------------------------------------------------------------
```

Line (11) determines the amount of data. (14) gets the number of coefficients and dimension of the prompted polynome whereas the number of dimension of the dataset has to be equal to the one of the polynome.

In the next step (17-23) the exponent matrices descriped in the last chapter are generated. The system of equations $A \cdot y = b$ is built up in the lines (30-47). Therefore the matrix A is filled entry-wise with the use of the exponent matrices. The solution vector is built after the same procedure but with the additional use of the z-values.

Line (50) solves the resulting system of equations and returns the polynome coefficients y.

4.2 Complete set of functions

In the last section only the basic algorithm was described. Two more functions are very comfortable. 'PrettyPrintF' returns the polynome in a nice shape, so the user can verify the input. The function can be called with further arguments so individual names for the dimensions and coefficients can be assigned. See therefore example 1 & 2. The second function 'EvalPol' evaluates the polynome using the obtained coefficients at one point.

```
//————————————————————————————————————————————————————————————
// This algorithm calculates regressions polynomes of various
// dimensions, coefficients and any combination of potences.
// Released 22. September 2010 by Serge Zihlmann
// THIS CODE IS FREE TO BE USED IN ANY KIND OF SOFTWARE
// For further questions contact: powerfit( at@at )thirdway.ch
//————————————————————————————————————————————————————————————
//————————————————————————————————————————————————
function [y] = PowerFit(data, zval, polynome)

  Error = 0;                //Start without errors
  [ndata dimD] = size(data); //Get amount of datapoints

  //get number of free variables of polynome
  [dim nCoeff] = size(polynome); //Number of dimensions and coefficients

  //Check for Errors
  if (dim <> dimD)
    printf("Different␣dimensions␣of␣polynome␣and␣data\n");
    Error = 1
  end
   if (ndata <> size(zval))
     printf("Different␣amount␣of␣data\n");
    Error = 1
  end


  if (Error <> 1) //only start if no errors occur
  //generate exponent tables
  ExpMat = zeros(nCoeff,nCoeff,dim);
  for i=1:nCoeff
    for k=1:dim
      ExpMat(i,:,k) = polynome(k,:)+polynome(k,i);
    end
  end
  ExpVec = polynome';

  // Generate matrices to fill
  A = zeros(nCoeff,nCoeff); // Final matrix
  b = zeros(nCoeff,1);        // Final solution vector

  // Build up matrix A
  for i=1:nCoeff
    for k=1:nCoeff
      t = ones(ndata,1);           // Temporary vector
      for r=1:dim
        t = t .* (data(:,r).^ExpMat(i,k,r));
      end
      A(i,k) = sum(t);
    end
  end

  // Build up vector b
  for i=1:nCoeff
      t = zval;
      for k=1:dim
        t = t .* (data(:,k).^ExpVec(i,k));
      end
      b(i) = sum(t);
  end

  // Solve the system A*y=b for y
  y = inv(A)*b; //calculate solution, return parameters
  end // end error
endfunction
//------------------------------------------------------------------------
//Evaluates polynome with coefficients, x=datapoint
function [y]=EvalPol(polynome, x, coefficients)
  [cx cy] = size(polynome);
  y = coefficients';
  for i=1:cx
    y = y.*(x(i).^polynome(i,:));
  end
  y = sum(y);
endfunction
//------------------------------------------------------------------------
//--Prints the input polynome in a pretty shape
function [y]=PrettyPrintF(polynom, coeffName, dimName)
  //prints out the symbolic function defined in polynom
  [cx cy] = size(polynom)
  printf("Regression function:\n");
  printf("-------------------------------\nf(");

  [ax ay] = size(dimName);
  [bx by] = size(coeffName);

  for i=1:cx
    if (i<=ay)
```

```
        printf("%s", dimName(i));
      else
        printf("D%i",i);
      end
      if(i<>cx) printf(","); end
    end
    printf(")␣=␣");
    U = [];
    for i=1:cy
      if (i<=by)
        U = U + coeffName(i);
      else
        U = U + "a"+string(i);
      end

      for k=1:cx
        if(polynom(k,i)<>0)//if power of x is greater than zero
          if (k<=ay)
            U = U + dimName(k);
          else
            U = U + "D" + string(k);
          end

          if((polynom(k,i)<>0) & (polynom(k,i)<>1))
            U = U+"^"+string(polynom(k,i));
          end
        end
      end

      if(i<cy)
        U = U + "␣+␣";
      end
    end
    printf("%s",U);
    printf("\n\n");
    y = []; //dont return anything
endfunction
```

4.2.1 EvalPol()

The function must be called EvalPol(A,B,C). A is the polynome in exponent shape. B is the position where the polynome must be evaluated and C contains the calculated coefficients.

4.2.2 PrettyPrintF()

This function is called PrettyPrintF(A,B,C). Again A is the polynome. B & C are char arrays containing the names for the independent variables and for the coefficients. The function can also be called PrettyPrintF(A,[ ],[ ]). In this case the labeling is generated automatically.

4.3 Examples for calling the functions

4.3.1 Example 1

```
//———————————————————————————————————————————————
// This is a simple example with only basic use. Labeling
// for coefficients and dimensions is automatic. No values
// will be printed out
//———————————————————————————————————————————————

// Restart programme
clear;      // Clear memory
clc;        // Clear display
exec('Functions.sci'); // Get functions from extern file

// Enter data position
data =[
2 2
2 3
2 4
2 5
3 2
3 3
3 4
3 5
];

// Enter values
zval =[
45
63
83
103
93
138
183
228
];

// Input the polynome
polynome =[
0 1 2
0 1 1
];

PrettyPrintF(polynome, [], []);      // Print polynome in pretty shape
y = PowerFit(data,zval,polynome);   // Calc polynome (main function)

x=[2,2];

disp(EvalPol(polynome,x,y))

clf;

xx=0:0.1:6;;
yy=-6:0.1:7;

for i=1:length(xx)
  for k=1:length(yy)
    zz(i,k) = EvalPol(polynome,[xx(i),yy(k)],y)
  end
end


plot3d1 (data(:,1), data(:,2), (zval),flag=[-1, 2, 3]);
plot3d1 (xx, yy, (zz),flag=[-1, 2, 3]);
```

4.3.2 Example 2

```
//-----------------------------------------------------------------
// This is an asvanced example which also defines your own
// labeling for coefficients and dimensions. Furthermore
// plots are generated and values wiht the calculated poly-
// nome are ploted. But the data is only one dimensional
//-----------------------------------------------------------------

// Restart programme
clear;      // Clear memory
clc;        // Clear display
clf;        // Clear plot
exec('Functions.sci'); //Get functions from extern file

// Enter measured data: Main input section
data =[       // Define dataset
1;2;3;3.5;4;5;6];

zval =[       // Define values according to dataset
9;2;3;4.5;6;5;4];

// Input the polynome
polynome =[
0 1 -1 2
];

// Enter names for coefficients and dimensions
dimNames   = ["x"];   // Label of dimensions
coeffNames = ["a" "b" "c" "d" "e" "f" "g"];   // Names of dimensions

// Generate automatic names if missing
[cx cy] = size(coeffNames);
for i=1:length(polynome(1,:))
  if (i>cy)
    coeffNames(1,i) = "a" + string(i);
  end
end
clear cx, cy;

// Printout polynome formula in pretty shape
PrettyPrintF(polynome, [], []);                    // Print with auto names
PrettyPrintF(polynome, coeffNames, dimNames);   // Print with own names

// Printout results (main calculation)
printf("Result:\n---------\n");
y = PowerFit(data,zval,polynome);   // Calc polynome (main function)
[cx cy] = size(y);
disp([coeffNames(1:cx)' string(y)]);       // Printout coefficients
clear cx, cy;

// Generate nice plots
plot(data,zval,"or");                   // Plot entered data
D = (max(data)-min(data))*1.05/2;
C = (min(data)+max(data))/2;
xx = C-D:0.1:C+D;                      // Generate x area
for i=1:length(xx)                     // Calculate corresponding values
  yy(i) = sum((xx(i).^polynome).*(y'));
end
plot(xx,yy)                            // Plot regression polynome
```

All scilab files with additional examples can be downloaded from http://www.thirdway.ch.