An Algorithm for Generating the Design Matrix for Multivariate Polynomial Least Squares Fitting

Sami F. Al-Hamdan
Computer Engineering Department,
Hijjawi Faculty for Engineering Technology
Yarmouk University, Irbid, Jordan
shamdan@yu.edu.jo

Journal of Digital Information Management

ABSTRACT: *This paper presents an algorithm for generating the design matrix that is usually used for multivariate polynomial least square fitting. The design matrix is used in least squares fitting algorithm to construct a set of linear equations whose solution is the required polynomial coefficients. The developed algorithm was coded in MATLAB. The coded function named mv_polyfit(X,Y,ord) accepts as inputs two matrices: the first argument is a 2D matrix of the independent variable X, the second argument is the dependent variable vector Y, and the last argument is the required degree of the fitting polynomial. The function returns the coefficient vector C and the design matrix A. Number of data points (k) needed for the function should be large enough for the solution of the set of linear equations to exist.*

Categories and Subject Descriptors
G. 3 [Probability and statistics]; Multivariate statistics **I.1.1** [**Expressions and Their Representation**]; Representations (general and polynomial): **F.2.1** [**Numerical Algorithms and Problems]** Computations on polynomials

General Terms
Multivariate Polynomial Least Squares, 2 D Matrix

Keywords: Multivariate Polynomial Fitting, Curve Fitting, Numerical Algorithms, MATLAB

Received 17 December 2006; Revised and Accepted 12 February 2007

1. Introduction

Modeling is perhaps the first level of computer use in science and engineering. Starting from physical principles and design ideas, the computer is used to mimic nature. After this, the results are examined and the modeling program is modified then the program is tested again until a satisfactory model is reached. The next deeper level of computer use is that the computer itself examines the results of modeling and reruns the modeling job. This deeper level is variously called “fitting” or “estimation” or “inversion” [1]

When Scientists try to find a model to describe a certain set of data which might have been collected from experimental work or from simulation programs such as SPICE; they usually try to find a function that would describe the relation of the dependant variable to the independent variable(s) as close as possible. The curve fitting problem is how to choose from an infinite number of curves the one which fits best the given data points, normally by finding a mathematical expression to create the curve.(see Figure. 1).

Commonly used procedures are least squares fitting [2][3][4][5], linear regression, and nonlinear regression [6][7][8][9][10]. One of the difficulties in curve fitting is to choose the functional form of the data for parameter optimization. Computers are normally used to perform curve fitting procedures and they do this by solving a system of equations to find the parameters of the function that minimize the squared error. Frequently used methods are the gradient descent algorithm [11][12], the Gauss-Newton algorithm [13], and the Levenberg-Marquardt algorithm [14] .

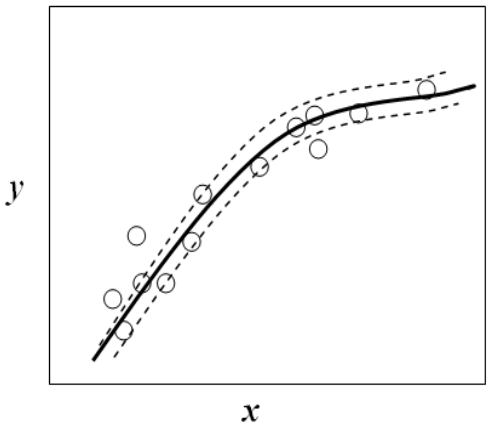

Figure 1. Best fitting polynomial

In mathematics, polynomial functions, or polynomials, are an important class of simple and smooth functions. Polynomials are usually constructed using only multiplication and addition. Polynomials are also infinitely differentiable which gives them their smoothness nature. Because of their simple structure, polynomials are easy to evaluate, and are used extensively in numerical analysis for polynomial interpolation or to numerically integrate more complex functions.

The Least squares fitting algorithm is a very popular technique used for curve fitting [15][16]. This algorithm is also used in various modeling purposes [17].

Least squares fitting algorithm is used in recent work on developing graphical techniques for judging metamodels’ quality-of-fit [18][19]. In this work, MATLAB is our primary tool in investigating and calculating different polynomial models coefficients. A general MATLAB function for multi variable polynomial fitting was needed to fit the simulated data of some electronic circuits. This was the actual motive behind the development of this algorithm.

In this paper, we present an algorithm that can be used to generate the design matrix **A** discussed in the next section. The proposed algorithm is general for any number of independent variables and for polynomials of any degree as will be demonstrated.

To demonstrate the use of the algorithm in reality a simple practical test example for surface curve fitting is used. A complete MATLAB implementation of the algorithm is also listed in the appendix.

2. The Least Squares Technique: Univariate Case

For a one independent variable with a set of data points (x_i, y_i), $i = 1,\ldots,k.$; least squares fitting consists of constructing a polynomial $\boldsymbol{q_n(x)}$ as in Equation. 1 below

$$q_n(x) = \sum_{i=0} a_i x^i \qquad (1)$$

Where $\boldsymbol{q_n(x)}$ **is** an n-degree polynomial that provides the minimum sum of squared errors (SSE); i.e. using $\boldsymbol{q_n(x)}$ then

$$\sum_{i=0}^{k} (y_i - q_n(x_i))^2 = min \sum_{i=0}^{k} Y_i - p(x_i))^2 \qquad (2)$$

Where $\boldsymbol{p(x_i)}$ belongs to the set of all polynomials with degree n, typically n < k. see [3] for more details. To find the polynomial that would give the minimum SSE the term to the right in Equation 2 is differentiated with respect to each coefficient of the polynomial and equated to zero. This process leads to the solution of a system of linear equations shown in Equation 3 below

$$\mathbf{(A^TA)C = A^TY} \qquad (3)$$

Where **A** is a k × (n+1) matrix defined as shown below for the case of one independent variable x with k points and a polynomial of n degree (Equation 4).

$$\mathbf{A} = \begin{bmatrix} x_1^n & x_1^{n-1} & \ldots & 1 \\ x_2^n & x_2^{n-1} & \ldots & 1 \\ \vdots & \vdots & \ldots & 1 \\ x_k^n & x_k^{n-1} & \ldots & 1 \end{bmatrix} \qquad (4)$$

The **C** matrix is the coefficient matrix, while **Y** is the dependent variable vector (see Equations 5&6).

$$\mathbf{C} = [c_0 \quad c_1 \quad \ldots \quad c_n]^T \qquad (5)$$

$$\mathbf{Y} = [y_0 \quad y_1 \quad \ldots \quad y_k]^T \qquad (6)$$

The product $\mathbf{A^TA}$ is a symmetric and non-singular matrix if $\boldsymbol{x_i} \neq \boldsymbol{x_j}$ for i ≠ j. **A** is usually called the design matrix [20]. For 1D problem; generating **A** for least squares fitting is straightforward. However, for a more general multivariate case it becomes non trivial.

2. The Multivariate Case

When the number of independent variables in the problem is two or more the generation of the design matrix **A** becomes more elaborate. For example, a second-order polynomial of three independent variables may contain some or all of the following terms: $\boldsymbol{x_1^2}$, $\boldsymbol{x_2^2}$, $\boldsymbol{x_3^2}$, $\boldsymbol{x_1x_2}$, $\boldsymbol{x_1x_3}$, $\boldsymbol{x_2x_3}$, $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $\boldsymbol{x_3}$, and a constant term. The maximum number of terms in a polynomial of m variables and of order n is given by Eq. 7 below.

$$\text{Max\_Num\_terms} = (n + m)! / n!m! \qquad (7)$$

For example, if the polynomial is of order two (i.e. n = 2) and has three independent variables (i.e. m =3), then the maximum possible number of terms in this polynomial will be (2+3)!/2!3! = 10.

In general, a polynomial of degree n contains terms like $\boldsymbol{x_1^i x_2^j \ldots x_m^l}$, such that $i+j\ldots+l \leq n$, where m is number of independent variables. The proposed algorithm is used to generate the design matrix **A**. For example, for a second order polynomial of three independent variables **A** is given by Eq. 8 below.

$$\mathbf{A = [X_1^2 \quad X_1X_2 \quad X_1X_3 \quad X_2^2 \quad X_2X_3 \quad X_3^2 \quad X_1 \quad X_2 \quad X_3 \quad 1]} \qquad (8)$$

Where $\mathbf{X_1}$ is the column vector of all k values (k is the number of measurement points for the first independent variable $\boldsymbol{x_1}$). $\mathbf{X_1X_2}$ is the dot product of column vector $\mathbf{X_1}$ with $\mathbf{X_2}$, and so on. The last column has the value of one in all of its elements.

4. The Algorithm

To generate all the terms that are needed to fill the design matrix **A**, we start with four matrices: **X**, **D**, **E**, and **A** which are initialized as shown below.

$\mathbf{X = [X_1 \quad X_2 \quad X_3]}$ **,** 2D matrix of independent variables set of points

$\mathbf{D = [X_1 \quad X_2 \quad X_3]}$, 2D matrix of i'th order terms, initially filled with first order terms of the independent variables.

E = [—-empty**——],** temporary 2D matrix that is filled in each round with the results of dot product of **X• D**

A = [1 X1 X2 X3] , Design matrix initially filled for first order polynomial solution.

Note that the initial values of **A** represent the design matrix for a first degree polynomial. The algorithm proceeds on rounds such that in each round all terms of the same degree are generated by multiplying elements of the **D** matrix with those of the **X** matrix as explained below. The resultant terms are filled in the **E** matrix. **E** is empty at the beginning of each round. When a round is finished, the **E** matrix is copied into the **D** matrix which is also concatenated with the **A** matrix as shown in equation 9 below. **E** is then emptied to prepare it for a new round of calculations where terms of the next higher order terms are calculated. **X** stays the same for all the time.

$$\mathbf{A = [A \quad D]} \qquad (9)$$

A is initially filled with columns adequate to solve for the coefficients of a first-order polynomial. The algorithm starts first by filling up the empty matrix **E** by the dot product of columns of **D** with columns of **X** without repeating similar terms. In the above example we do first $\mathbf{X_1 \cdot [X_1 \; X_2 \; X_3]}$ as shown in Figure 2.

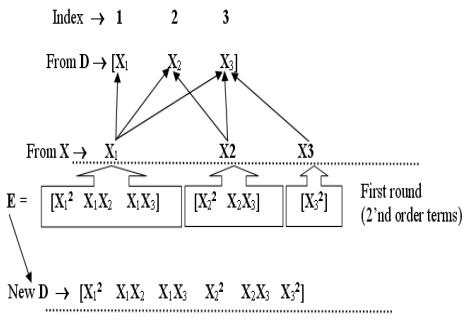

Figure 2. First round of dot products

In the next step we multiply $\mathbf{X_2}\cdot[\mathbf{X_1}\ \mathbf{X_2}\ \mathbf{X_3}]$ but starting at column 2 as shown in Figure 3 below. The result of the new multiplications are concatenated with the **E** matrix to produce the updated **E** matrix: $[\mathbf{X_1^2}\ \ \mathbf{X_1X_2}\ \ \mathbf{X_1X_3}\ \ \mathbf{X_2^2}\ \ \mathbf{X_2X_3}]$.

For $\mathbf{X_3}$ we start from the third column and this produces $\mathbf{X_3^2}$ term which is again concatenated with **E**. Now **E** contains terms that are all of second-order, i.e. the sum of the powers of all independent variables in each term found in the columns of **E** is two.

A complete round has now finished. If it is desired to find terms for a third order polynomial, the algorithm continues by replacing **D** matrix with **E** matrix, updating the design matrix as in Equation 9 above, and emptying the **E** matrix for a new round.

By looking at the pattern of each variable (in **X**) starting point of multiplication in **D,** we notice that $\mathbf{X_1}$ always starts at column 1; however the remaining variables starting points change from round to round in a pattern that can be predicted. To illustrate, for the three variables example mentioned earlier, if we put the number of columns(terms) generated by each variable in each round in a matrix **P**, such that each row of **P** represents the number of terms generated by each variable in the different rounds, then for a second order polynomial, **P** = [3 2 1] where 3 is number of columns generated by $\mathbf{X_1}$, 2 number of columns generated by $\mathbf{X_2}$, and 1 is the number of columns generated by $\mathbf{X_3}$ (see Figure 2 above). Note that after the first round is finished the **D** matrix contains 6 columns. If a second round is performed, (i.e. if we wanted a polynomial of third order), then the second row of **P** should contain [6 3 1]. For a fourth order, a new row with values [10 4 1] will be generated, and so on.

In the second round (i.e. when a third-order polynomial is required), the starting point of multiplication in **D** for $\mathbf{X_2}$ is 4, while the starting point for $\mathbf{X_3}$ is 6, …etc.

So for our example with three variables, we need to generate a matrix (**S)** of starting points of.

$$S=\begin{bmatrix}1 & 1 & 1\\ 2 & 4 & 7\\ 3 & 6 & 10\end{bmatrix}\begin{matrix}x_1\ \textit{starting positions}\\ x_2\ \textit{starting positions}\\ x_3\ \textit{starting positions}\end{matrix} \quad (10)$$

Equation 11 below shows an instance of the values of each row of **P** for our example. Note that we always have row one of **P** filled with ones for a reason that will be apparent soon.

$$\mathbf{P}=\begin{bmatrix}1 & 1 & 1\\ 3 & 2 & 1\\ 6 & 3 & 1\end{bmatrix} \quad (11)$$

By observing the values of each row in **P** in Equation 11 above we notice that they are the cumulative sums of the values of the preceding row but in a reversed order (i.e. from tail to head). For example, the first element in row 3 in matrix **P** which has a value of 6 is obtained by summing all the elements of row 2 (i.e. 1 + 2 + 3), while the second element of row 3 which has a value of 3 is the sum of (1 + 2) (second and third elements of row 2) , and so on.

In general, the cumulative sum of a row vector $\mathbf{R} = [\,r_1\ \ r_2\ \ r_3\ \ \ldots.\ \ r_n\,]$ for example is computed as shown in Equation 12.

$$\mathbf{CSR} = [r_1 \quad (r_1+r_2) \quad (r_1+r_2+r_3)\ldots] \quad (12)$$

Our goal is to generate an array which contains the starting points (in the **D** matrix) of multiplication for each variable (in the **X** matrix) for each round of the calculations. These starting positions are directly related to the number of terms generated by each variable in each round. To illustrate, $\mathbf{X_1}$ starting positions are always 1's in all rounds. However, $\mathbf{X_2}$ starts with position 2 (see Fig. 3) in the first round, while the starting point for $\mathbf{X_3}$ is 3, and so on. Note that the first round generates the second-order terms, multiplications as in Equation 12 below where the first row in **S** represents the starting points of multiplications for $\mathbf{X_1}$ for three rounds (i.e. up to a fourth degree polynomial). The second row is for $\mathbf{X_2}$, and the third row for $\mathbf{X_3}$ …etc

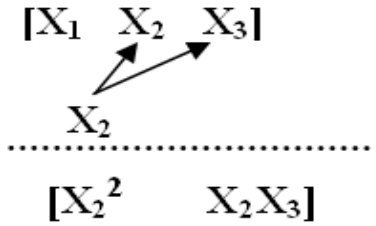

Figure 3. Second step of dot products

In order to compute **S** we start **P** with one row filled with ones, then we proceed by filling each next row by the cumulative sum of the row immediately above, producing for example the matrix in Equation 13 below.

$$\mathbf{P}=\begin{bmatrix}1 & 1 & 1\\ 1 & 2 & 3\\ 1 & 3 & 6\end{bmatrix} \quad (13)$$

Note that row 2 is the cumulative sum of row 1, and row 3 is the cumulative sum of row 2.

Then we flip **P** horizontally giving Equation 14 below

$$\mathbf{P}=\begin{bmatrix}1 & 1 & 1\\ 3 & 2 & 1\\ 6 & 3 & 1\end{bmatrix} \quad (14)$$

P is then replaced by its transpose (Eq. 15)

$$\mathbf{P}=\mathbf{P}^T\begin{bmatrix}1 & 3 & 6\\ 1 & 2 & 3\\ 1 & 1 & 1\end{bmatrix} \quad (15)$$

If we compute the cumulative sums of all columns in the new **P** and then add 1 to each element we get for the case presented above Eq. 16

$$\mathbf{P}=\begin{bmatrix}1 & 3 & 6\\ 2 & 5 & 9\\ 3 & 6 & 10\end{bmatrix}+1=\begin{bmatrix}2 & 4 & 7\\ 3 & 6 & 10\\ 4 & 7 & 11\end{bmatrix} \quad (16)$$

By deleting the last row in the resultant matrix above, and adding a row of ones to its top we get the required matrix **S** for the starting points of multiplications as shown in Eq. 10 above.

The example of matrix **S** above provides us with the starting points for three variables up to a fourth order polynomial fitting. However, higher-orders polynomials make **S** increase in column size, while increasing the number of variables makes **S** increase in row size.

Note that at the end of the calculations the resultant matrix **A** is arranged in reversed order (i.e. flipped horizontally) compared to the one shown in Eq. 4. This is not a problem since the solution for the polynomial coefficients filled in **A** will just be in reversed order to that shown in equation 5.

4. Test Example

The example that will be presented here demonstrates the ability of using the multivariate polynomial fitting algorithm developed here for surface fitting. The sample data was generated using the **peaks** function from MATLAB. 256 data points are stored in each of X1,X2, and Y. These data points generated from the peaks MATLAB function are used to find an interpolating function for this two variable problem. The original data are plotted in figure 4 below.

The following MATLAB script is used to generate a 3D shape of size 16x16 points as shown in figure 4

```
[X1,X2,Y]= peaks(32);
X1=X1(8:23, 8:23);
X2=X2(8:23, 8:23);
Y=Y(8:23, 8:23);
colormap('gray');
surfl(X1,X2,Y)
```

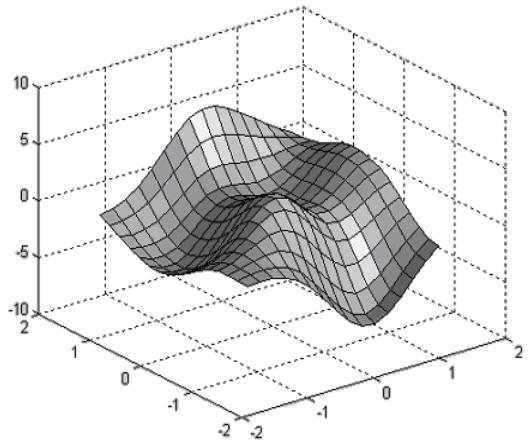

Figure 4. 3D surface test example

X1 and X2 are the independent variables where Y is the dependent variable.

Two polynomials were tried: A third degree polynomial and a fifth degree. The data in X1, X2, and Y are fed to our multivariate polynomial fitting program **mv_polyfit** to calculate all the coefficients for a3'rd degree polynomial. The design matrix **A** is used with **C** to evaluate this 3'rd degree polynomial over the space of the problem. The resulting data (Yhat) is used to plot the curve shown in figure 5 below. It is obvious that a 3'rd degree polynomial is not good enough to fit the original data.

The following script should follow the previous one used for figure 4 above.

```
X=[X1(:)X2(:)];
Y=Y(:);
[C,A]=mv_polyfit(X,Y,3);

Yhat = (A*C)';
Y2=[];
for i=1:16:256
  Y2 = [Y2; Yhat(i:i+15)];
end

figure(2);

colormap('gray');
surfl(X1,X2,Y2);
```

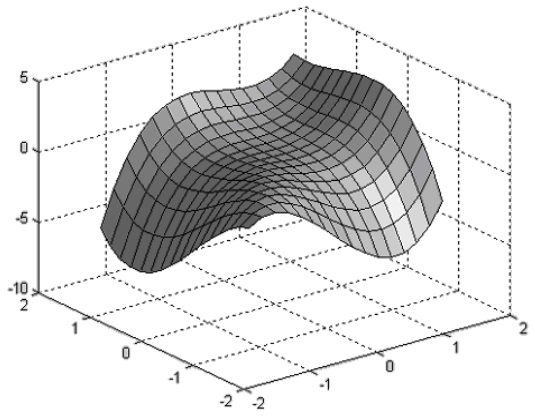

Figure 5. Surface fitting using a 3'rd degree polynomial

The above process was repeated to compute the coefficients for a 5'Th degree polynomial. Again, these new coefficients are used to evaluate the polynomial producing Yhat that is then plotted giving the new surface shown in figure 6 below. Now it seems that a much better fit is produced.

The MATLAB script for using a 5'th degree polynomial to do the surface fitting of the above example is shown below

```
[C,A]= mv_polyfit_3(X,Y,5);
Yhat = (A*C)';
Y3=[];
for i=1:16:256
   Y3 = [Y3; Yhat(i:i+15)]
end

figure(2);
colormap('gray');
surfl(X1,X2,Y3);
```

The 21 coefficients produced by **mv_polyfit** function for a fifth degree polynomial are shown in table 1 below. It is clear that the new polynomial fits the data much better than the previous 3'rd degree one.

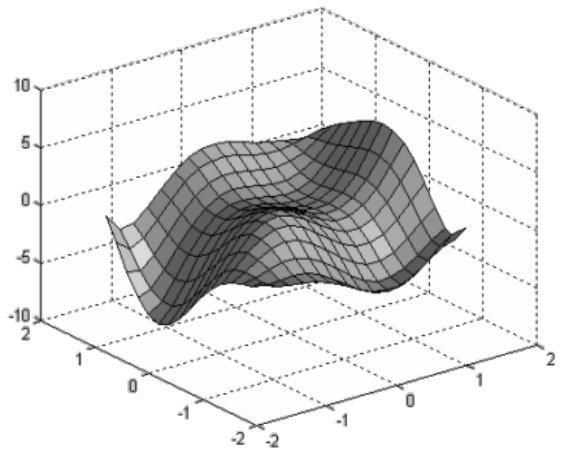

Figure 6. Surface fitting using a 5'th degree polynomial

5. Discussion and Conclusion

The algorithm presented in this paper is implemented in MATLAB and tested for several cases of model fitting. The

C0 =	0.8227	**C11** =	-0.6921
C1 =	-1.3563	**C12** =	-0.4916
C2 =	-.2315	**C13** =	-0.3549
C3 =	0.4540	**C14** =	-0.2558
C4 =	2.2547	**C15** =	-1.2512
C5 =	0.9288	**C16**	0.5724
C6 =	4.3537	**C17**	-0.1558
C7 =	-.0489	**C18**	-1.1612
C8 =	-2.2834	**C19**	0.6620
C9 =	5.2130	**C20**	-.9854
C10 =	0.3556		

Table 1. Coefficients for a 5'Th degree polynomial

algorithm proved successful in computing the design matrix **A** for all the tested cases. The algorithm generates a special indexing matrix **S** that is used in the calculations of the dot product of the variables which are filled in **A** later. The developed algorithm is capable of producing the design matrix needed for multivariate polynomial least squares fitting for any order and any number of variables, provided that the computers' memory is adequate to hold the required matrices needed for the algorithm, and the set of data is sufficient for the solution to exist.

Statistical measures such as the sum of the squared error between the original data and the polynomial approximations for the model can be used to judge the quality of fit; however, recent work on developing visual means for this purpose has led to new techniques such as the circle plot, the ordinal plot, and the marksman plot [18] [19]. By combining these new visual techniques and the algorithm developed here it will be possible to speedup the search process for better fitting polynomials for modeling purposes.

References

[1] http://sepwww.stanford.edu/sep/prof/pvi/ls/paper_html accessed 21 May 2007.

[2] Skeel, R., Keiper J (1993). Elementary Numerical Computing with Mathematica, McGraw Hill International Editions,

[3] Recktenwald G. (2001). Numerical Methods with MATLAB: Implementations and applications,. Prentice –Hall.

[4] Gill, P. E., Murray (1978). W. Algorithms for the solution of the nonlinear least-squares problem. *SIAM Journal on Numerical Analysis*, 15 (5) 977-992.

[5]http://www.efunda.com/math/leastsquares/leastsquares.cfm accessed May 2007

[6] http://mathworld.wolfram.com/topics/Regression.html cited May 2007

[7] http://en.wikipedia.org/wiki/Non-linear_regression cited May 2007

[8] Cohen, J., Cohen P., West, S.G., Aiken, L.S (2003). Applied multiple regression/correlation analysis for the behavioral sciences . (2nd ed.) Hillsdale, NJ: Lawrence Erlbaum Associates.

[9] Snyman, Jan A (2005). Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms. Springer Publishing.

[10] Draper, N.R., Smith H (1998). Applied Regression Analysis. Wiley Series in Probability and Statistics

[11] Mahony, R., Williamson R (2001). Prior Knowledge and Preferential Structures in Gradient Descent Learning Algorithms, *Journal of Machine Learning Research*, 1, 311-355.

[12] http://en.wikipedia.org/wiki/Gradient_descent , cited May 2007

[13] http://en.wikipedia.org/wiki/Gauss-Newton_algorithm , cited May 2007

[14]http://en.wikipedia.org/wiki/Levenberg-Marquardt_algorithm, cited May 2007

[15] Pratt, V (1987). Direct Least-Squares Fitting of Algebraic Surface, *Computer Graphics*, 21 (4) 145-152.

[16] Bajaj C., Ihm I., Warren, J (1993).Higher-Order Interpolation and Least-Squares Approximation Using Implicit Algebraic Surfaces. *ACM Transactions on Graphics*, 12 (4) 327-347.

[17] Chen, V., Tsui K., Barton R., Meckesheimer, M(2006). A review on design, modeling and applications of computer experiments. *IIE Transactions,* 38. 273 -291.

[18] Hamad, H., Hamdan, S (2005). Two new subjective validation methods using data displays, *In*: Proceedings of the 2005 Winter Simulation Conference, M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds. 2542-2545.

[19] Hamad H., Hamdan, S (2007). Discovering metamodels' quality-of-fit for simulation via graphical techniques, *European Journal of Operational Research*, 178. 543–559.

[20] Balcazar, J.,Lorite, F (1998). Teaching polynomial fitting to a set of data via orthogonal polynomials, I*nternational Journal of Mathematical Science and Technology* 29 (4) 481-487.

Appendix: MATLAB code for the multivariate polynomial fitting algorithm.

function [C,A]= mv_polyfit(X,Y,ord)

%The arguments for this function that %the user should pass are as follows:

%

% **X** = [X1 X1 ...Xm] matrix of

%independent variables data points

```
% where X1 is the column for the data
% of the first variable, X2 data of
% the second variable…etc. m is
% Number of independent variables.

% Y is the observation or simulation
%data of the dependent variable
%
% ord  is the required order(degree)
%for the fitting polynomial
%
% The function will build the design
% matrix A and use least square curve
% fitting to find the coefficients
% matrix C of
% the multi-variable function, C =[C0 % C1 ...CL].

[N,NV]=size(X); % find the size of X %to get NV (number of
variables), and
% number of data points Nc1=ones(N,1); % generate a
column of %ones to be added (conctenated) with %X
A =[c1 X]; % A now contains the data
%for polynomial fitting of order one
%
%
if(ord==1)
  if(NV+1>N)
    error('number of points is not sufficient');
   end;
  C=A\Y; % Use MATLAB backslash
%operator to solve for the
%Coefficients
  return;

elseif (ord==2)
  S=1:NV;
  S=S';
  if(sum(S)+1+NV>N)
    error('number of points is not sufficient');
   end;

else
  P=ones(1,NV);
  for j=2:ord
    B=cumsum(P(j-1,:));
    P=[P;B];
  end
  C=sum(sum(P))+1; %number of coef. in C

    if(C>N)
       error('number of points is not
sufficient');
     end;
     P(end,:)=[];
     P=fliplr(P);% flip the P matrix
                 %left-right
    for k=1:ord-1

P(k,:)=cumsum(P(k,:));
    end;
    S=P';
    S=S+1;
    P=ones(1,ord-1);
    S=[P;S];%add a row of ones to top S
    S(end,:)=[];% delete last row

 end

% Now start the rounds of column dot
% multiplications
D=X; % initialize D the data of the
%variables

for k=1:ord-1
    [R L]=size(D); % find number of
       %columns L of D at this stage
    E=[]; % start E with an empty matrix
    for i=1:NV % each variable in array
% D will have its turn to be multiplied
% by columns in X starting from a
% Column with the index extracted from
% rows belonging to this variable in S

       for j=S(i,k):L % j the number of
% the column in D to start
% multiplication with

            CI = X(:,i).*D(:,j);
% xi *. [D1 D2 ... Df]
             E = [E CI];
% each multiplication will generate a
%column CI that will be added to E
       end % end j loop
    end  % end i loop

    A = [A E]; % update A by adding E
    D=E; % Initialize D for a new round
 end % end for k loop

C=A\Y ; % compute the coefficients
% vector C
% END OF FUNCTION
```

Sami Al-Hamdan is an assistant professor in the computer engineering department at Hijjawi faculty for engineering technology, Yarmouk university, Jordan. His research interests include fringe analysis, 3D mapping, electronic circuits modeling, multiprocessors architectures, numerical algorithms, and DSP.