ELSEVIER

Chemometrics and Intelligent Laboratory Systems 55 (2001) 13–22

Chemometrics and intelligent laboratory systems

www.elsevier.com/locate/chemometrics

Equation-oriented system: an efficient programming approach to solve multilinear and polynomial equations by the conjugate gradient algorithm

Ji-Hong Wang, Philip K. Hopke *

Departments of Chemical Engineering and Chemistry, Clarkson University, Box 5705, Potsdam, NY 13699, USA

Received 21 June 2000; accepted 15 October 2000

Abstract

The factor analysis problem can be conceptualized as an expansion of polynomial equations that are solvable using least-squares methods. The equation-oriented system (EOS) is introduced as a method for solving polynomial equations using a preconditioned conjugate gradient (CG) algorithm for the normal equations. EOS is a fast, easy to program, low computer memory requirement method for accomplishing this task. EOS can be used to solve multilinear and PARAFAC problems. The practical aspects of implementing EOS in MATLAB are discussed.

Keywords: PARAFAC; Bilinear model; Trilinear model; Polynomial equation; Conjugate gradient (CG) algorithm; Equation-oriented system (EOS)

1. Introduction

Algebraic expressions such as 3, $5f_1$ or $6f_1f_2$ are called monomials, and the sums and/or differences of monomials are called polynomials such as $f_1 + 2f_2 - 3f_1^2f_2$. In this work, all of the variables in a polynomial carry only positive integral exponents. Further more, the polynomial equation is one in which the right hand side of the equation is a polynomial and the left hand side is a constant value [1]. Some example polynomial equations are

$$1.02 = 12f_1f_2 - f_2^2 + f_3 + 1$$

$$0 = f_1f_2f_3$$

and

$$-0.12 = f_2f_3 - 10f_2 + f_1f_2.$$

PARAFAC [2] and other multilinear models are special cases of polynomial equations that are well known in chemometrics. Paatero [3] has previously

* Corresponding author. Tel.: +1-315-268-3861; fax: +1-315-268-6654.

E-mail address: hopkepk@clarkson.edu (P.K. Hopke).

0169-7439/01/$ - see front matter
PII: S0169-7439(00)00110-6

shown that the conjugate gradient (CG) [4,5] method is a powerful tool for solving polynomial equations in his program, the multilinear engine (ME). ME is a very flexible way to solve a wide variety of multilinear models that can fit multilinear and quasi-multilinear models to two-, three- and many-dimensional data arrays. In ME, a large table of integer code values is used to specify the equations. The CG used in ME must form a matrix of derivatives called the Jacobian matrix, or simply the Jacobian. Whenever the Jacobian is needed, the program has to retrieve information from the table. Although ME is a relatively efficient algorithm particularly when compared to alternating least squares methods, additional improvements will be useful in a number of applications where the multilinear model is useful.

In this work, the equation-oriented system (EOS) is introduced to solve the problem. EOS places these equations in a central role in the organization of the program and develops a new approach to treat the Jacobian. Computer time and the memory requirements have been reduced significantly. The programming of EOS is sufficiently easy to setup their own code for input equations.

2. Preconditioned conjugate gradient algorithm

Consider the problem of finding the unknown parameters $\mathbf{f}$ (of size N by 1) from the known data $\mathbf{x}$ and polynomial equations having the following array relationship

$$\mathbf{x} = \mathbf{P}(\mathbf{f}). \tag{1}$$

EOS uses several different kinds of input equations that the user may add. The equations that specify Eq. (1) are called model equations. Eq. (1) is a collection of all the equations that connect the unknown parameters and known data. Typically, the user will use two additional types of equations. For example, for a bilinear data set, EOS always needs one equation to specify the bilinear model, and another one to normalize one factor model in the solution. The normalization equation is called as auxiliary equation in the naming convention used by Paatero [3]. The m-th component of $\mathbf{P}$ is denoted by $\mathbf{P}_m$ and Eq. (1) can be expressed as

$$x_m = \mathbf{P}_m(\mathbf{f}), \qquad m = 1, \ldots, M \tag{2}$$

where x is the least-squares solution to the Eq. (1) if x minimizes the following equation called the object function

$$\sum_{m=1}^{M} \left(x_m - \mathbf{P}_m(\mathbf{f})\right)^2. \tag{3}$$

In most practical situations, one needs to adjust the importance of each equation in Eq. (2). This task can be accomplished by weighted least-squares (WLS), where there is a weight w_m associated with the m-th equation, and the solution to the all equations is obtained by solving the alternative equations

$$\sqrt{w_m}\, x_m = \sqrt{w_m}\, \mathbf{P}_m(\mathbf{f}) \tag{4}$$

or by minimizing of the following equation

$$\sum_{m=1}^{M} w_m \left(x_m - \mathbf{P}_m(\mathbf{f})\right)^2. \tag{5}$$

In this paper, equations described by Eq. (4) will be considered.

Eq. (4) can be solved iteratively. From the estimated value $\mathbf{f}$, a new value, $\mathbf{f} + \mathbf{t}$, is calculated to yield a better fit to Eq. (4), where $\mathbf{t}$ is the increment to the $\mathbf{f}$. The task of each step is to find the $\mathbf{t}$.

Since the polynomial is always nonlinear, it will be helpful to approximate the polynomial by linear expressions. One way to accomplish this is to replace it by a two term Taylor expansion of $\mathbf{P}(\mathbf{f} + \mathbf{t})$ around the point $\mathbf{f}$. Then Eq. (4) becomes

$$\sqrt{w_m}\, x_m = \sqrt{w_m}\, \mathbf{P}_m(\mathbf{f}) + \sqrt{w_m}\, \mathbf{P}_{\mathbf{f},m}(\mathbf{f})\mathbf{t} \tag{6}$$

or

$$\sqrt{w_m}\, \mathbf{P}_{\mathbf{f},m}(\mathbf{f})\mathbf{t} = \sqrt{w_m}\left(x_m - \mathbf{P}_m(\mathbf{f})\right) \tag{7}$$

where $\mathbf{P}_{\mathbf{f}}(\mathbf{f})$ is the Jacobian of derivatives of $\mathbf{P}(\mathbf{f})$ with respect to $\mathbf{f}$.

The normal equations of Eq. (7) can be represented as

$$\mathbf{J}^T\mathbf{W}\mathbf{J}\mathbf{t} = \mathbf{J}^T\mathbf{r} \tag{8}$$

by setting the Jacobian matrix **J** as

$$\mathbf{J}(\mathbf{f}) = \mathbf{P}_{\mathbf{f}}(\mathbf{f}) \tag{9}$$

and

$$\mathbf{r}(\mathbf{f}) = \mathbf{W}(\mathbf{x} - \mathbf{P}(\mathbf{f})). \tag{10}$$

where **W** is the diagonal of matrix with values w_m.

J(**f**) and **r**(**f**) will be written as **J** and **r**, respectively, when there is no confusion as to meaning. A complete listing of symbols and notation are given in Appendix A.

The original polynomial equations are then transformed into the form of Eq. (8). There are a number of methods to use for obtaining **t**. The conjugate gradient (CG) method is an efficient technique to iteratively solve a system of equations. A preconditioning step is necessary, however, to speed-up the CG computation. In this work, the preconditioner is chosen as the diagonal of matrix $\mathbf{J}^T\mathbf{W}\mathbf{J}$.

In some practical situations, the unknown **f** needs to be limited with respect to its expected bounds such as restricting values to be non-negative. Paatero [3] suggested an inverse preconditioning method to achieve non-negativity, which is also used in EOS.

The CG is an iterative algorithm that solves Eq. (8) first, and then forms a new equation by setting **f** equal to $\mathbf{f} + \mathbf{t}$ in Eq. (8). The procedure is then repeated until convergence is achieved. This approach can be quite slow, and the **f** given in Eq. (8) is not necessarily an optimal one with respect to Eq. (1), so one does not need to get the best solution to Eq. (8). An alternative approach is to update the **J** and **r** in each step of CG.

The inverse preconditioned CG algorithm for solving Eq. (1) is given as the following.

1. Input **W** and initial **f**. Set ϵ to be a small positive value such as 10^{-12}. Set $c_i = 1.0$. $\rho = 0$. $u_n = \Sigma_m w_m j_{mn}^2$. Use $\|\mathbf{r}\|$ to represent the Euclidean norm of **r**, and function $\psi(\mathbf{f})$ changes **f** to meet constraints on **f**.
2. While not reaching convergence and the preset limit on the number of iterations is not reached, do
   (a) $\mathbf{g} = \mathbf{J}^T\mathbf{r}$
   (b) $z_n = c_n g_n / u_n$
   (c) $\rho^{\text{old}} = \rho$, $\rho = \mathbf{g}^T\mathbf{z}$
   (d) If $\rho^{\text{old}} = 0$, then $\beta = 0$, else $\beta = \rho/\rho^{\text{old}}$, endif
   (e) $\mathbf{t} = \beta\mathbf{t} + \mathbf{z}$
   (f) $\mathbf{v} = \mathbf{J}\mathbf{t}$
   (g) $t = \mathbf{v}^T\mathbf{W}\mathbf{v}$
   (h) $a = \rho/t$
   (i) $\mathbf{f} = \mathbf{f} + a\mathbf{t}$
   (j) For $n = 1, \ldots, N$, do
      If f_n meets the constraints, then $c_n = \min(2 \times c_n, 1)$, else $c_n = \max(c_n/32, \epsilon)$, $t_n = \max(t_n/2, \epsilon)$. Endif
   (k) Enddo
   (l) While $\|\mathbf{r}(\psi(\mathbf{f} + a\mathbf{t}))\|$ is not less than $\|\mathbf{r}(\mathbf{f})\|$, do $a = a/2$, enddo
   (m) If it is impossible to find an a to satisfy $\|\mathbf{r}(\psi(\mathbf{f} + a\mathbf{t}))\| < \|\mathbf{r}(\mathbf{f})\|$, or convergence is slowing down and the user-specified restart limits allow it, then $r = 0$, $u_n = \Sigma_m w_m j_{mn}^2$, else $\mathbf{f} = \psi(\mathbf{f} + a\mathbf{t})$, endif
3. Enddo

Most portions of this algorithm are easy to implement with the exception of the following three expressions: $\mathbf{g} = \mathbf{J}^T\mathbf{r}$, $\mathbf{v} = \mathbf{J}\mathbf{t}$, $u_n = \Sigma_m w_m j_{mn}^2$. The computation of **r**(**f**) also requires special treatment. Only the codes for those three equations are related with the equations. In the Section 3, the approaches to calculate the three variables are sequentially discussed.

3. Equation-oriented system

It is well known that the Jacobian can be very large when attempting to solve very large problems. Thus, it is generally not practical to store it in memory for large data sets. In some cases, one can use sparse matrix techniques to store the Jacobian, but this approach is also not very efficient. It is computationally expensive to construct and retrieve information from the Jacobian matrix. The equation-oriented system (EOS) provides a way to eliminate the problems

associated with the Jacobian in the CG algorithm by eliminating the need for the Jacobian matrix altogether.

3.1. Presentation of polynomial equation

In the model Eq. (1), the unknown parameters and known data are assumed to be in vector form. Representation of the CG algorithm (Section 2) in vector form is very short. However, it is not always convenient to use vectors to organize the data. In most cases, a multidimensional array is a more natural way to represent the data. However, this approach creates several problems. One problem is that sparse techniques are necessary. Another problem is that the CG algorithm needs to frequently reshape or retrieve information from the vectors. These two problems make the implementation not very efficient. In some algorithms, this kind of overhead will not reduce the speed significantly. For EOS, however, each CG iteration step is quite short and the overhead of vector-based method can be high. In EOS, all data sets are stored in the original array form, so the sparse method is avoided and the data sets do not need to be reshaped.

In EOS, a special form of presentation is used to input and handle polynomial equations. The presentation is very close to the ordinary mathematical form in which the equations would be written. The following steps are used to transfer a mathematical equation to its presentation.

1. Expand the polynomial in a sum of monomials. The variables with an exponent n in the polynomials are replaced by multiplication of the variables n times.
2. The left-hand side of equation is the data values that need to be fitted. The right-hand side is the model used to fit the data.
3. Write equations using a summation, S. Then remove the summation sign.

For example, mathematical equation

$$x_{ijk} = \sum_{h=1}^{H} f1_{ih} f2_{jh} f3_{kh} + \sum_{h=1}^{H} f4_{ijh} f5_{kh} \qquad (11)$$

is represented as

$$x_{ijk} = f1_{ih} f2_{jh} f3_{kh} + f4_{ijh} f5_{kh}. \qquad (12)$$

In Eq. (11), the known data is $\underline{\mathbf{X}}$, and the unknown variable sets are **F1**, **F2**, **F3**, **F4** and **F5**.

For a given presentation of an equation, an equivalent mathematical expression can be obtained using a simple rule. The rule is to perform the summation over all of the indices that only appear in the right-hand side for each monomial. For Eq. (12), the first monomial $f1_{ih} f2_{jh} f3_{kh}$ has four indices: i, j, k and h. The first three indices also appear in the left-hand side of Eq. (12) in x_{ijk}. Only h is unique to the first monomial, so the summation is over h. It is similar for the second monomial.

3.2. Associated variables

Although multidimensional arrays are preferred in the implementation of the CG in EOS, the vector representation is better for understanding the computations involved in the CG algorithm. A new term, associated variable, is introduced to connect the multidimensional array with its corresponding vector form. The concept of the associated variable bridges the understanding of the vector representation and the implementation of the array format in EOS. In the CG algorithm provided in Section 2, all of the data are presented as vectors. However, in EOS, every vector exists as a set of arrays. Conceptually, the arrays can be stacked together to form a vector as is done in other programs such as ME [3]. The arrays that can form a vector in Section 2's algorithm are called as the associated arrays of that vector. The associated variable is represented by connecting the vector and the array by underline. For example, if the model equation is Eq. (11), and all of the factors are unknown parameters, arrays c_f1_{ih}, c_f2_{jh}, c_f3_{kh}, c_f4_{ijh}, and c_f5_{kh} are arrays, and $f1_{ih}$, $f2_{jh}$, $f3_{kh}$, $f4_{ijh}$, and $f5_{kh}$'s **c** associated variables, respectively. The associated array has the same size of its original array, e.g. c_f1_{ih} has the same size of $f1_{ih}$.

3.3. The Jacobian in EOS

In the CG algorithm, the equations involving **J** have to deal with the Jacobian information. For ex-

ample, in the equation $\mathbf{v} = \mathbf{Jt}$, the exact equation should be $v_m = j_{mn}t_n$ in the term of the polynomial representation. In EOS, all of the terms in the equation are converted into corresponding associated variables, and then the equation is rearranged into polynomial form. This general rule permits handling of the Jacobian information in EOS. This approach avoids the problems associated with the large size of **J**.

The above discussion may be somewhat hard to understand. The following is a simple example to illustrate the idea. Consider a model

$$x = ab + 6c - 0.5 \tag{13}$$

where y is known data, and a, b and c are unknown parameters. The derivatives of y with respect to the unknowns are

$$x_a = b \tag{14}$$

$$x_b = a \tag{15}$$

$$x_c = 6. \tag{16}$$

The Jacobian array is $[b\ a\ 6]$ (denoted as **J**). To obtain the results of the product of the Jacobian array $[b\ a\ 6]$ with a vector $\mathbf{t}\ (=[t_1\ t_2\ t_3])$, EOS does not construct **J** and then calculate $\Sigma_{h=1,\ldots,3} j_h t_h$ as might be expected because it would require too much memory to store **J**. In EOS, the product is obtained directly from the expression $bt_1 + at_2 + 6t_3$, then **J** is not needed. Of course, the expression $bt_1 + at_2 + 6t_3$ needs to be induced in EOS.

In the CG algorithm, only the calculations of **v**, **g** and **u** involve **J**. With the special representation of polynomial equations and the concept of associated variables, the calculation of **v**, **g** and **u** will be given sequentially. The calculation **r** is also provided.

3.4. Calculating *r*

The representation of equations in EOS must be provided by the user. The calculation of the residuals of the fit is quite direct. **r** is obtained by subtracting the right-hand side variables from the left-hand side variable in Eq. (12). From Eq. (12), one obtains

$$r\_x_{ijk} = x\_x_{ijk} \times w\_x_{ijk} - f\_f1_{ih} \times f\_f2_{jh} \times f\_f3_{kh} \times w\_x_{ijk} - f\_f4_{ijh} \times f\_f5_{kh} \times w\_x_{ijk}. \tag{17}$$

3.5. Calculating *v*

The algorithm for calculating **v** is performed by changing the equations in the following steps.For each model equation,

1. replace the left-hand side variable with its **v** associated array;
2. for each monomial, assume there are h unknown variables in this monomial, yield h monomials by replacing one unknown variable with the product of its **t** associated array at one time. Add all of the h new monomials together and replace the original monomial.

From Eq. (11), the new equation is

$$v\_x_{ijk} = t\_f1_{ih} \times f\_f2_{jh} \times f\_f3_{kh} + f\_f1_{ih} \times t\_f2_{jh} \times f\_f3_{kh} + f\_f1_{ih} \times f\_f2_{jh} \times t\_f3_{kh} + t\_f4_{ijh} \times f\_f5_{kh} + f\_f4_{ijh} \times t\_f5_{kh}. \tag{18}$$

3.6. Calculating *g*

The algorithm for calculating **g** is to change the model equations using the following steps.

1. Set all the unknown variables **g** associated variables to 0.
2. For each unknown variable in the monomial of each model equation
   (a) construct a new equation by keeping the left-hand side of the original equation, and keeping the current monomial in the right-hand side;
   (b) replace the left side variable with the product of its **r** associated array, replace the unknown variable with its **g** associated array;
   (c) exchange the left-hand side parts and the **g** associated array;
   (d) add the **g** associated array to the right-hand side.

From Eq. (11), we have

$$g\_f1_{ih} = g\_f1_{ih} + r\_x_{ijk} \times f\_f2_{jh} \times f\_f3_{kh} \quad (19)$$

$$g\_f2_{jh} = g\_f2_{jh} + f\_f1_{ih} \times r\_x_{ijk} \times f\_f3_{kh} \quad (20)$$

$$g\_f3_{kh} = g\_f3_{kh} + f\_f1_{ih} \times f\_f2_{jh} \times r\_x_{ijk} \quad (21)$$

$$g\_f4_{ijh} = g\_f4_{ijh} + r\_x_{ijk} \times f\_f5_{kh} \quad (22)$$

$$g\_f5_{kh} = g\_f5_{kh} + f\_f4_{ijh} \times r\_x_{ijk}. \quad (23)$$

3.7. Calculating **u**

The algorithm for calculating **u** is to change the model equations using the following steps.

1. Set all of the unknown variables **u** associated variables to 0.
2. For each unknown variable in the monomial of each model equation
   (a) construct a new equation by keeping the left-hand side of the original equation, and keeping the current monomial in the right-hand side;
   (b) Replace the unknown variable with its **u** associated array, replace the left-hand side variable with its **w** associated array;
   (c) exchange the **w** associated array with the **u** associated array;
   (d) square each variable in the right-hand side except the **w** associated arrays;
   (e) add the **u** associated array to the right-hand side.

The above algorithm does not produce **u** in an exact way when there are repetition factors in the right side of equations, it ignores the interaction terms. The algorithm can be modified to produce the exact values of **u**. Experience shows that the above simple version algorithm works quite well, so EOS employs the above algorithm.

For Eq. (11), we have

$$u\_f1_{ih} = u\_f1_{ih} + w\_x_{ijk} \times f\_f2^2_{jh} \times f\_f3^2_{kh} \quad (24)$$

$$u\_f2_{jh} = u\_f2_{jh} + f\_f1^2_{ih} \times w\_x_{ijk} \times f\_f3^2_{kh} \quad (25)$$

$$u\_f3_{kh} = u\_f3_{kh} + f\_f1^2_{ih} \times f\_f2^2_{jh} \times w\_x_{ijk} \quad (26)$$

$$u\_f4_{ijh} = u\_f4_{ijh} + w\_x_{ijk} \times f\_f5^2_{kh} \quad (27)$$

$$u\_f5_{kh} = u\_f5_{kh} + f\_f4^2_{ijh} \times w\_x_{ijk}. \quad (28)$$

4. Implementation of EOS in MATLAB

The implementation of EOS is divided into following steps.

(a) The user provides four variables to collect information for describing the problem.
(b) EOS performs initial steps to set default values for some variables.
(c) Following the algorithms in Section 3, produce the expression to calculate **g**, **v**, **u** and **r** in presentation forms where the expressions storied as strings in MATLAB.
(d) Developing the interpreter of the presentation of polynomial expression. This function will be called when the values of **g**, **v**, **u** and **r** are wanted in the main CG program.
(e) Programming the CG algorithm.

The results are stored as variables in the MATLAB workspace. EOS can be coded in any computer language. MATLAB [6] is a convenient language since it has been widely used by many chemometricians. In this section, the implementation of EOS in MATLAB is discussed. Being an interpreted language, MATLAB is slower than the compiled languages such as C, C + +, or FORTRAN. However, MATLAB is easy to implement EOS quickly, and it is easy to debug the program. In the five parts listed above, parts (a), (b) and (e) are quite easy to program. They do not need to be treated specially. The function for part (c) must be provided in EOS. This

function provides the expressions to calculate **g**, **v**, **u** and **r** in the presentation form. One can also code the expressions by hand for any special equations following the algorithms as outlined in Section 3, and does not need to be coded as a general function in EOS.

Part (d) is the only difficult one. MATLAB does not provide the function to perform multiplication on multidimensional arrays up to version 5.3. Function EVALND is coded to interpret the presentation of polynomial expression. EVALND can be used in a manner analogous to EVAL, except EVALND can execute the presentation of polynomial expressions. Fortunately, the implementation of EVALND can be completely separated with other parts of EOS.

More details of parts (c) and (d) are given below.

4.1. Interpreter of the presentation of polynomial expression

MATLAB does not provide a function to interpret expressions such as Eq. (11). Thus, a function, EVALND, has been developed. EVALND accepts expressions translated from the representation of the polynomial. For example, expression (12) is converted to

$$X(\_i,\_j,\_k) = f1(\_i,\_z)^{*} f2(\_j,\_z)^{*} f3(\_k,\_z) + f4(\_i,\_j,\_z)^{*} f5(\_k,\_z) \quad (29)$$

and then evaluated by EVALND

$$\text{evalnd}('X(\_i,\_j,\_k) = f1(\_i,\_z)^{*} f2(\_j,\_z)^{*} f3(\_k,\_z) + f4(\_i,\_j,\_z)^{*} f5(\_k,\_z)'). \quad (30)$$

Another important feature of EVALND is that it automatically groups the terms in an optimal way. For example, assume there are three matrices: **X1**, **X2** and **X3** of size 100 by 200, 200 by 500, and 500 by 2, respectively. It is much slower to calculate (**X1*** **X2**)* **X3** than **X1***(**X2*** **X3**). To EVALND, the following two expressions

$$\text{evalnd}('X(\_i,\_j) = X1(\_i,\_k)^{*} X2(\_k,\_t)^{*} X3(\_t,\_j)') \quad (31)$$

and

$$\text{evalnd}('X(\_i,\_j) = X1(\_j,\_k)^{*} X2(\_k,\_t)^{*} X3(\_k,\_j)') \quad (32)$$

will be evaluated as **X1***(**X2*** **X3**).

The implementation of EVALND avoids the use of FOR LOOPs to set or retrieve values from the array. For a large data set, the FOR LOOP will make the program extremely slow.

The implementation of EVALND is split into two functions. One is to arrange the sequence of execution of the string in EVALND. Another function, NDTIMES, performs the multiplication on only two arrays. NDTIMES can be coded in MATLAB or be replaced by using the MEX function. The easy way is to use MEX. Function NDTIMES can be obtained from authors.

To specify the equations, the user just writes the equations and then translate them into the proper representations that EOS can evaluate directly.

4.2. Function to give expressions for *v*, *g* and *u*

The algorithms used to implement this function are given in Section 3 of this paper. The user can also write expressions by hand to solve special problems following those algorithms.

In the algorithm provided in Section 2, the variables **t**, **z**, etc., are represented as vectors. In EOS, these variables exist as associated arrays. They need not be converted to vectors. For example, to get $\mathbf{t} = \beta \times \mathbf{t} + \mathbf{z}$ in step 2e in the algorithm, EOS implements Eq. (2) in the following way

$$t\_f1_{ih} = \beta \times t\_f1_{ih} + z\_f1_{ih} \quad (33)$$

$$t\_f2_{jh} = \beta \times t\_f2_{jh} + z\_f2_{jh} \quad (34)$$

$$t\_f3_{kh} = \beta \times t\_f3_{kh} + z\_f3_{kh} \quad (35)$$

$$t\_f4_{ijh} = \beta \times t\_f4_{ijh} + z\_f4_{ijh} \quad (36)$$

$$t\_f5_{kh} = \beta \times t\_f5_{kh} + z\_f5_{kh}. \quad (37)$$

5. Example

To illustrate what EOS is doing and give the reader a more complete image of EOS, an example is dis-

cussed in this section. The example is a trilinear model including the equations to normalize two factor modes

$$x_{ijk} = \sum_{h=1}^{H} f1_{ih} f2_{jh} f3_{kh} \quad (38)$$

$$normf2_h = \sum_{j=1}^{J} f2_{jh} \quad (39)$$

$$normf3_h = \sum_{k=1}^{K} f3_{kh} \quad (40)$$

where **X**, **F1**, **F2**, **F3**, **normf2**, and **normf3** are of size I by J by K, I by H, J by H, K by H, H by 1, and H by 1, respectively.

EOS is a function which accepts information from the user and outputs the estimated unknown parameters. Eqs. (38), (39) and (40) are passed to EOS as

$$X(\_i,\_j,\_k) = F1(\_i,\_h)^{*} F2(\_j,\_h)^{*} F3(\_k,\_h) \quad (41)$$

$$\mathrm{NORMF2}(\_h) = F2(\_j,\_h) \quad (42)$$

$$\mathrm{NORMF3}(\_h) = F3(\_k,\_h). \quad (43)$$

The values **X**, **normf2** and **normf3** are provided by the user. The weights are stored in **w_X**, **w_normf2** and **w_normf3**, which are the same size as **X**, **normf2** and **normf3**, respectively. The MATLAB implementation of the algorithm was outlined in Section 2 above and is presented in Appendix B.

6. Conclusions

EOS provides a simple framework to solve very large, complex polynomial equations. The key to implementing EOS is in the user provided equations that describe both the data structure and the constraints on the solution. EOS can solve a variety of multilinear including the PARAFAC problem.

The memory requirements for EOS are quite small that makes it is a good choice for large data sets. For a standard multilinear model with M known parameters and N unknown parameters, the memory requirement is $3M + 6N$. The $3M$ space is to store the associated arrays **v**, **r** and **x**. The $6N$ space is to store the associated array **c**, **u**, **g**, **z**, **f** and **t**. When applied to the simultaneous analysis of several data sets, EOS has been found to be quite efficient. More quantitative evaluations of the efficiency of this algorithm are currently being conducted and will be reported in a future publication.

Acknowledgements

This work was supported by Unilever Research US. We thank Tom Hancewicz for his comments on drafts of this manuscript and for the useful discussions of the mixture resolution problem.

Appendix A. Notation and list of symbols

In general, the conventions used in this paper are as follows. Matrices are represented as bold uppercase letters and column vectors are represented as bold lowercase letters. Italic upper and lowercase letters are used for scalars.

The list of important symbols in the paper follows.

x	known data in a vector form
f	unknown parameter in a vector form
M	number of known data
N	number of unknown parameters
P(x)	vector function to specify the models
$\mathbf{P}_m(\mathbf{x})$	the m-th element of **P(x)**
w_m	weight of the m-th equation
t	the increase of the unknown parameters
$\mathbf{P}_f$	the derivative **P** of with respect to **f**
$\mathbf{P}_{f,m}$	the m-th element of $\mathbf{P}_f$
J	Jacobian matrix
$\mathbf{g}_i(\mathbf{f})$	the i-th element of **G**
r(f)	the weighted residual for the fit of **x**
$\mathbf{r}_i(\mathbf{f})$	the i-th element of **r(f)**
g	used in the CG algorithm, equals to $\mathbf{J}^T\mathbf{r}$
z_i	used in the CG algorithm, equals to $c_i\ g_i\ /u_I$
u_i	used in the CG algorithm, equals to $\Sigma_i\ g_{ik}^2$

c_i used in the CG algorithm, to implement the inverse preconditioning method

v used in the CG algorithm, equals to **Jt**

$\underline{\mathbf{f}}$ used in the CG algorithm, the new try of update **f** by basic CG step

$\psi(\mathbf{f})$ used in the CG algorithm, the function to change **f** to meet constraints

ϵ, α, β, ρ used in the CG algorithm, check the meaning from Refs. [4,5]

A, **B**, **C**, **S**, **T** unknown factors in the example mixture model

r_, **c_**, **g_**, **w_**, **t_** and **u_** prefixes of **r**, **c**, **g**, **w**, **t** and **u** associated variables, respectively

Appendix B. Implementation of the algorithm

The following table shows the MATLAB codes for the expressions in the algorithm given in Section 2.

Expression in algorithm	MATLAB code in EOS

$\mathbf{g} = \mathbf{J}^T\mathbf{r}$	g_F1 = zeros(I,H);
	g_F2 = zeros(J,H);
	g_F3 = zeros(K H);
	evalnd('g_F1(_i,_h) = g_F1(_i,_h) + r_X(_i,_j,_k) * f_F2(_j,_h) * f_F3(_k,_h)');
	evalnd('g_F2(_j,_h) = g_F2(_j,_h) + f_F1(_i,_h) * r_X(_i,_j,_k) * f_F3(_k,_h)');
	evalnd('g_F3(_k,_p) = g_F3(_k,_p) + f_F1(_i,_p) * f_F2(_j,_p) * r_X(_I,_j,_k)');
	evalnd('g_F2(_j,_h) = g_F2(_j,_h) + r_NORMF2(_h)');
	evalnd('g_F3(_k,_h) = g_F3(_k,_h) + r_NORMF3(_h)');

$u_n = \Sigma_m w_m j_{mn}^2$	u_F1 = zeros(I,H);
	u_F2 = zeros(J,H);
	u_F3 = zeros(K,H);
	evalnd('u_F1(_i,_h) = u_F1(_i,_h) + w_X(_i,_j,_k)
	* f_F2. ^2(_j,_h) * f_F3. ^2(_k,_h)');
	evalnd('u_F2(_j,_h) = u_F2(_j,_h) + f_F1. ^2(_i,_h)
	* w_X(_i,_j,_k) * f_F3. ^2(_k,_h)');
	evalnd('u_F3(_k,_h) = u_F3(_k,_h) + f_F1. ^2(_i,_h)
	* f_F2. ^2(_j,_h) * w_X(_i,_j,_k)');
	evalnd('u_F2(_j,_h) = u_F2(_j,_h) + w_NORMF2(_h)');
	evalnd('u_F3(_k,_h) = u_F3(_k,_h) + w_NORMF3(_h)');

$\mathbf{v} = \mathbf{Jt}$	evalnd('v_X(_i,_j,_k) = t_F1(_i,_h) * f_F2(_j,_h) * f_F3(_k,_h) + f_F1(_i,_h)
	* t_F2(_j,_h) * f_F3(_k,_h) + f_F1(_i,_h) * f_F2(_j,_h) * t_F3(_k,_h)');
	evalnd('v_NORMF2(_h) = t_F2(_j,_h)');
	evalnd('v_NORMF3(_h) = t_F3(_k,_h)');

$t = \beta\mathbf{t} + \mathbf{z}$	t_F1 = β * t_F1 + z_F1; t_F2 = β * t_F2 + z_F2; t_F3 = β * t_F3 + z_F3;
$\rho = \mathbf{g}^T z$	ρ = 0; ρ = ρ + g_F1(:)' * z_F1(:); ρ = ρ + g_F2(:)' * z_F2(:); ρ = ρ + g_F3(:)' * z_F3(:);

Expression in algorithm	MATLAB code in EOS
$t = \mathbf{v}^T v$	t = 0;
	t = t + v_X(:)' * v_X(:);
	t = t + v_NORMF2(:)' * v_NORMF2(:);
	t = t + v_NORMF3(:)' * v_NORMF3(:);
$\underline{\mathbf{f}} = \mathbf{f} + a\mathbf{t}$	f_F1 = f_F1 + a * t_F1;
	f_F2 = f_F2 + a * t_F2;
	f_F3 = f_F3 + a * t_F3;
$z_i = c_i g_i / u_i$	z_F1 = c_F1. * g_F1./u_F1;
	z_F2 = c_F2. * g_F2./u_F2;
	z_F3 = c_F3. * g_F3./u_F3;
Compute $\mathbf{r}(\mathbf{f})$	evalnd('r_X(_i,_j,_k) = r_X(_i,_j,_k) − f_F1(_i,_h) * f_F2(_j,_h) * f_F3(_k,_h)');
	evalnd('r_NORMF2(_h) = r_NORMF2(_h) − f_F2(_j,_h)');
	evalnd('r_NORMF3(_h) = r_NORMF3(_h) − f_F3(_k,_h)');

Within the table, the variables such as α must be placed with the regular names for MATLAB. Again, the task shown in the table will be done by the program EOS, the user does not need to do any of what is presented.

References

[1] F.J. Mueller, Elements of Algebra, Prentice-Hall, Englewood Cliffs, 1969.
[2] R.A. Harshman, M.E. Lundy, PARAFAC: parallel factor analysis, Comput. Stat. Data Anal. 18 (1994) 39–72.
[3] P. Paatero, The multilinear engine—a table-driven least squares program for solving multilinear problems, including the n-way PARAFAC model, J. Comput. Graph. Stat. 8 (1999) 1–35.
[4] Y. Saad, Iterative Methods for Sparse Linear Systems, PWS Publishing, Boston, MA, 1996.
[5] C.T. Kelley, Iterative Methods for Linear and Nonlinear Equations, SIAM, Philadelphia, 1995.
[6] MATLAB, Mathworks, Natick, MA, 1999.