

Improved Fitting of Constrained Multivariate Regression Models using Automatic Differentiation

Trevor Ringrose and Shaun Forth

Applied Mathematics and Operational Research Group, Cranfield
University, RMCS Shrivenham, Swindon SN6 8LA, Great Britain.

Abstract. Regression models for multivariate response surfaces are proposed, and an illustration is given of numerical maximisation of likelihoods using automatic differentiation being superior to that using finite differences.

Keywords. automatic differentiation, multivariate regression, response surface methods

1 Introduction

Response Surface Methodology (RSM) is a collection of techniques for designing and analysing experiments, with the response variable viewed as defining a surface over the explanatory variables. The fitted surface is often a second order polynomial regression model, and it is common to use ‘Canonical Analysis’ to express the surface in its canonical form, usually centred at the stationary point and with orthogonal axes defining the main directions of curvature. These orthogonal axes and their associated curvatures are the eigenvectors and eigenvalues of a symmetric matrix formed from the coefficients of the second-order terms in the model. This canonical form can aid interpretation of the fitted model, and can discover ‘ridges’, defined by zero or near-zero eigenvalues, where explanatory (input) variables can be changed in certain ratios without changing the response (output) variable.

In many cases there are several response variables, but approaches to the multivariate problem are less well developed, with response surfaces and canonical forms still being fitted separately, leading to difficulties in interpretation. However, the response surfaces may share similar features, e.g. the axes or the stationary points may be similar to each other, and we might then ask whether the observed differences could be assigned to random variation.

Hence we propose fitting constrained multivariate regression models directly in their canonical form, with constraints applied in various combinations on the eigenvectors, eigenvalues and stationary points. A particularly promising model is that where all sets of eigenvectors (orthogonal axes in the canonical form) are constrained to be equal, since this aids interpretability and seems quite likely in practice. This is strongly related to the Common Principal Component (CPC) method for grouped multivariate data (Flury, 1988), where each group is constrained to have the same principal component axes (eigenvectors of the covariance matrix) but with no constraints on the eigenvalues. Similarly the stationary points can be constrained to equality or the surfaces can be constrained to be scalar multiples of each other.

Assuming normal errors, these models can be fitted numerically with maximum likelihood and the appropriateness of the constraints assessed with the

likelihood ratio test. The constraints are implemented either by re-expressing the problem or by constrained optimisation using Lagrange multipliers. All optimisation here was performed using the MATLAB command `fmincon`.

Numerical optimisation of any smooth objective function is more efficient and robust if the appropriate first (and possibly second) derivatives are known. If not then conventionally they must be approximated using finite-differences, with the ensuing problem of choice of step-size (a balance between truncation error accuracy and floating point round-off errors). Fitting the models proposed here using finite-difference gradients proved to be unreliable with problems of both accuracy and robust convergence. We therefore used the alternative of automatic differentiation to supply derivatives, accurate to machine round-off error, directly from the objective function.

2 Constrained multivariate regression models

Suppose we have n observations on p response and q explanatory variables. A second order polynomial model can conveniently be rewritten as

$$E(Y) = X\beta = X_0\beta_0 + X_1\beta_1 + X_2\beta_2 \quad (1)$$

where Y is the matrix of n observations by p responses, X_0, X_1, X_2 contain the columns of the design matrix X corresponding to block/constant, linear and quadratic terms respectively, and β_0, β_1 and β_2 similarly contain columns of the parameter matrix β . In addition, the superscript (j) will be used to denote the j -th response, so for example $\beta_1^{(j)}$ is the (column) vector of linear-term parameters for response j , i.e. the j -th column of β_1 . We assume that the p responses are normally distributed with covariance matrix Σ .

The canonical form of the j -th response surface can be derived from a symmetric $q \times q$ matrix $B^{(j)}$, whose diagonal elements are the coefficients of the square terms in $\beta_2^{(j)}$ and whose off-diagonal elements are half the coefficients of the cross-product terms. Its eigendecomposition is $B^{(j)} = M^{(j)}\Lambda^{(j)}M^{(j)T}$ where the eigenvalues $\lambda_h^{(j)}$ are the diagonal elements of $\Lambda^{(j)}$ and the orthogonal eigenvectors $\mathbf{m}_h^{(j)}$ are the columns of $M^{(j)}$. This matrix is defined so that, if \mathbf{x}_i^T is the i -th row of X_1 , the i, j -th element of $X_2\beta_2$ in (1) can be rewritten as

$$\{X_2\beta_2\}_{ij} = \mathbf{x}_i^T B^{(j)} \mathbf{x}_i = \mathbf{x}_i^T M^{(j)} \Lambda^{(j)} M^{(j)T} \mathbf{x}_i = \sum_{h=1}^q \lambda_h^{(j)} \left(\mathbf{m}_h^{(j)T} \mathbf{x}_i \right)^2 \quad (2)$$

Hence all observations can be expressed with respect to these axes, which define the principal axes of curvature on the response surface, with the eigenvalues and eigenvectors giving the curvatures and directions respectively. The canonical form then has its stationary point at $\mathbf{x}_0^{(j)} = -0.5(B^{(j)})^{-1}\beta_1^{(j)}$.

The usual unconstrained multivariate regression model can therefore be fit directly in its canonical form by rewriting the model as in (2) and maximising the log-likelihood numerically, using Lagrange multipliers to constrain the $M^{(j)}$ to be orthogonal. Constrained regression models in fact require fewer constraints when the likelihood is maximised in its canonical form, since usually (2) can be simplified. For example, to fit the CPC-type model we put $M^{(j)} = M \forall j$ and so have only one matrix to constrain to orthogonality.

3 Automatic Differentiation

Automatic Differentiation (AD) concerns the mathematical and software process of taking a (possibly) vector-valued function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ defined by computer code and creating new code that calculates the Jacobian $\mathbf{Jf}(\mathbf{x}) = [\partial \mathbf{f}_i / \partial \mathbf{x}_j]$. The underlying premise of AD is that computer codes define functions in terms of the few well defined mathematical constructs given by the programming language's arithmetic operations and intrinsic functions. Hence each statement in the code may be differentiated and, in *forward (or tangent) mode* AD, directional derivatives are propagated through the entire code using the chain rule of differentiation in time $O((1+n)C(\mathbf{f}))$ where n is the number of directional derivatives and $C(\mathbf{f})$ is the time required to calculate the function. Adjoint, or reverse mode, AD enables calculation of the gradient of a scalar valued function in time $O(C(\mathbf{f}))$ i.e. independent of the number of inputs (the so-called *cheap gradient result*). Adjoint mode AD requires propagating adjoint values backwards through the code and requires complicated compiler-like tools or storing and manipulating a record (or *tape*) of all the calculations performed. Well established AD tools are available for the differentiation of codes written in Fortran (ADIFOR, TAMC, Odyssee, ADO1) and C (ADIC, ADOL-C). See references in Griewank (2000) and Corliss et al (2002) for more details.

Since release 5.0 MATLAB has supported object-oriented (OO) programming. The ADMAT package of Verma (1998) clearly demonstrated the feasibility of using AD in MATLAB via these OO techniques. We found ADMAT to run slower than theory predicts, leading to development of the MAD package.

3.1 MAD - Matlab Automatic Differentiation

MATLAB utilises several intrinsic classes of variables and associated intrinsic functions. For example the `double` class consists of arrays of floating point numbers and functions such as `plus` to add two variables of class `double` together. The function `plus` is called by the MATLAB interpreter each time it encounters the plus sign `+` between two such variables and returns another appropriate value of type `double`. We say that the operator `+` is *overloaded* by the function `plus`. To facilitate forward mode AD in MATLAB we have designed and implemented two new classes of variable `fmad` and `derivvec`. The `fmad` class is used to store an array variable's value and a number of directional derivatives for that variable. Now if the MATLAB interpreter encounters such `fmad` variables it looks to the `fmad` class for an associated `plus` function.

3.1.1 The `fmad` Class

The `fmad` class is made up of functions written so as to propagate values and derivatives of variables. There is a class constructor necessarily called `fmad`. For two arrays `valx`, `derivx` of class `double` the statement `x=fmad(valx,derivx)` returns a variable `x` of `fmad` class. Within this variable `x` are stored its value `x.value = valx` and derivative `x.derivs = derivx`¹. Assuming `valx` is an array of size (or dimensions) `[i1,i2,...,in]` then the supplied `derivx` must consist of an integer multiple `nd` of `nel=i1×i2×...×in` values. Each successive set of these `nel` values, taken in array element order, is interpreted as a directional derivative of `x`.

A number of functions associated with the `fmad` class are also supplied. For example, when MATLAB attempts to execute the statement `z = x + y`, for either or both `x` and `y` of `fmad` class, then MATLAB's interpreter detects that the

¹ the dot denotes components of a structure in MATLAB

intrinsic MATLAB plus function is inappropriate and invokes the overloaded plus function of figure 1. The MATLAB intrinsic function `isa(x, 'fmad')` re-

```
function z=plus(x,y)
if isa(x,'fmad')&isa(y,'fmad')
    value=x.value+y.value;
    deriv=x.deriv+y.deriv;
elseif isa(x,'fmad')
    value=x.value+y;
    deriv=x.deriv;
else
    value=x+y.value;
    deriv=y.deriv;
end
z.value=value;
z.deriv=deriv;
z=class(z,'fmad');
```

Fig. 1. plus function for fmad class

turns the value true if `x` is of class `fmad` and false otherwise. The MATLAB `class` function takes a MATLAB structure and returns a variable of the `class` given by the second argument. With this knowledge we see that such code is simple and intuitive, requiring little MATLAB expertise to understand.

MATLAB is an interpreted language and variables are not declared. A variable's class is determined by the function (and its inputs) which create it. Since `fmad` functions return `fmad` variables, then once a first variable is defined to be of `fmad` class via the class constructor `fmad`, all other variables depending on the first will be of `fmad` class. In turn any other variable, depending via intermediates on the first will also be of `fmad` class. In this manner derivatives are propagated through a user's code and any modifications of MATLAB code to enable AD are minimal. MAD provides accessor² functions `getvalue` and `getderivs` to obtain the values and directional derivatives of `fmad` variables. So we may initialise one `fmad` array variable `x`, perform some numerical operations and then extract derivatives. For example, given the function $y = \sum_{i=1}^5 x_i^2$ with $\mathbf{x} = [1 \ 2 \ 3 \ 4 \ 5]$, we may calculate the directional derivative in the x_1 direction $[1 \ 0 \ 0 \ 0 \ 0]$, i.e. $\partial y / \partial x_1$ by,

```
x=fmad([1 2 3 4 5],[1 0 0 0 0]);
y=sum(x.^2);
dydx1=getderivs(y)
```

which correctly calculates the value 2. In the above we have utilised overloading of the `sum` and `power` intrinsics. With MAD all directional derivatives are calculated to machine precision so there are no approximation errors as is the case with finite-differencing. Unlike computer algebra packages, MAD readily handles arbitrarily complicated MATLAB code (loops, branches user-defined functions, etc.) and does not experience the exponential explosion in the complexity of terms common to such packages.

To deal with arbitrary numbers of directional derivatives we have designed and implemented the `derivvec` class.

² In object oriented programming an accessor function allows an application programmer to get a value from an object of a class without any knowledge of the internal structure of the object.

3.1.2 The derivvec Class

In the `fmad` constructor function described above, and for the case when the number of directional derivatives `nd` is greater than one, then the vector of the size `sx` of `valx` (`sx = size(valx) = [i1 i2 ... in]`) and the supplied derivatives `derivx` are passed to the `derivvec` class constructor function. This function stores the derivatives in the returned `derivvec` class variable as a (two-dimensional) matrix of size `[nel nd]` so that the j^{th} column of the matrix stores the j^{th} directional derivative, "unrolled" in array element order. Functions are provided to add and subtract variables of `derivvec` class, multiply them by matrices etc., all via operator overloading and in such a manner that when writing the `fmad` functions we may write MATLAB expressions correct for a single directional derivative, assured that the coding of the `derivvec` class will handle arbitrary numbers of directional derivatives.

We claim that our forward mode AD package of MAD is an improvement over that of ADMAT (Verma 1998) due to this separation of derivative manipulation from function value calculation because it: (a) improves clarity of the `fmad` class functions as seen in figure 1, allowing users to add their own functionality; (b) makes performance optimisation possible since, for a large number of directional derivatives, most of the floating point operations are performed in a few `derivvec` functions; and (c) for functions with sparse Jacobians and which use arrays of arbitrary rank (ADMAT restricted to vectors), allows use of variables of MATLAB's sparse matrix class within the `derivvec` class.

For our example $y = \sum_{i=1}^6 x_i^2$ we may trivially calculate the gradient of \mathbf{y} ,

```
x=fmad([1 2 3 4 5 6],eye(6));
y=sum(x.^2);
dydx=squeeze(getderivs(y))'
```

which returns `dydx= [2 4 6 8 10 12]`, and where `eye(6) = I6`, the identity matrix and `squeeze` is used to eliminate singleton dimensions.

4 Results

To demonstrate the superiority of optimisation using automatic differentiation rather than finite differences to calculate the gradients for `fmincon` we performed a small experiment on the unconstrained model fitted directly in its canonical form, since in this case we know the correct answer and can directly compare the two numerical solutions. Data were randomly generated from underlying unconstrained or CPC-type models, with p and q each 2 or 6 and n chosen to give a central composite design. For each combination, 500 data sets were generated and the unconstrained model fitted using finite differences and MAD. For regression parameters, eigenvalues and eigenvectors the differences between the directly-fitted results and usual least-squares results were calculated and summarised. For all the above parameters, the mean absolute difference for finite differences was consistently of the order 10^{-3} whereas for automatic differentiation it was 10^{-6} . This suggests that in other cases optimisation using automatic differentiation is likely to be more accurate than that using finite differences. Finite differences also took 4–5 times longer.

We also fit the CPC-type model to the three responses measuring dough expansion in the dough mixing data from Gilmour & Ringrose (1999), where $n = 28, p = 3, q = 3$. Here finite differences took around 380 seconds, MAD

around 54 seconds (on a SUN Ultra 10 workstation). The results were equivalent to about 3 decimal places for all parameters, with the MAD likelihood larger in the 12th decimal place. The likelihood ratio test suggests that the model fits adequately.

Unconstrained									Constrained			
Y ₁			Y ₂			Y ₃			Y ₁	0.26	1.17	0.92
1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	Y ₂	0.07	0.40	0.27
1.25	0.81	0.28	0.53	0.17	0.05	0.85	0.51	0.29	Y ₃	0.24	0.87	0.55
-0.22	0.31	0.92	-0.16	-0.85	-0.51	-0.14	0.61	0.78		0.91	0.01	0.41
-0.92	0.26	-0.31	-0.97	0.04	0.23	-0.88	0.29	-0.38		-0.36	0.51	0.78
0.33	0.91	-0.23	0.17	-0.53	0.83	0.46	0.74	-0.49		-0.20	-0.86	0.47

Table 1 : Eigenvalues (above) and eigenvectors (below).

From table 1, e.g. the first unconstrained eigenvalue-eigenvector pair for Y₃ is 0.85 and (-0.14, -0.88, 0.46), while in the CPC-type model it is 0.87 and (0.01, 0.51, -0.86). In general the eigenvalues are all fairly close to the unconstrained ones, while some of the eigenvectors look quite different. However, given that one expects eigenvectors to have high variances this is not too surprising, and the constrained eigenvectors still show the same patterns as the unconstrained ones.

5 Conclusions

The proposed models offer the potential of more interpretable multivariate response surfaces, at the expense of more difficult model-fitting. However, automatic differentiation offers a straightforward way of calculating derivatives of complicated functions directly from the function-evaluation routine, hence increasing the speed and accuracy of numerical optimisation.

Automatic differentiation is therefore widely applicable, in particular for improved maximisation of complicated likelihood functions, and the package MAD implements this in the MATLAB environment.

References

- (ed.) Corliss, G., Faure, C., Griewank, A., Hascoet, L. and Naumann, U. (2002). *Automatic Differentiation: From Simulation to Optimization*. New York: Springer.
- Flury, B. (1988). *Common Principal Components and Related Multivariate Models*. New York: Wiley.
- Forth, S.A. (2001). *MAD - A Matlab Automatic Differentiation Toolbox, Version 1, beta Release, the Forward Mode*. AMOR Report 2001/5, Cranfield University (RMCS Shrivenham).
- Gilmour, S.G. & Ringrose, T.J. (1999). Controlling processes in food technology by simplifying the canonical form of fitted response surfaces. *Applied Statistics*, **48**, 91–101.
- Griewank, A (2000). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Philadelphia: SIAM.
- Verma, A. (1998). ADMAT: Automatic Differentiation in MATLAB Using Object Oriented Methods. In: *Proceedings of SIAM Interdisciplinary Workshop on Object Oriented Methods for Interoperability*, p174-183. Philadelphia: SIAM.