SIAM J. SCI. COMPUT.
Vol. 31, No. 3, pp. 1840–1857

MULTIVARIATE REGRESSION AND MACHINE LEARNING WITH SUMS OF SEPARABLE FUNCTIONS*

GREGORY BEYLKIN†, JOCHEN GARCKE‡, AND MARTIN J. MOHLENKAMP§

Abstract. We present an algorithm for learning (or estimating) a function of many variables from scattered data. The function is approximated by a sum of separable functions, following the paradigm of separated representations. The central fitting algorithm is linear in both the number of data points and the number of variables and, thus, is suitable for large data sets in high dimensions. We present numerical evidence for the utility of these representations. In particular, we show that our method outperforms other methods on several benchmark data sets.

Key words. multivariate regression, machine learning, curse of dimensionality, separation of variables

AMS subject classifications. 62J02, 62H99, 65D15, 68T05

DOI. 10.1137/070710524

1. Introduction.

We consider the multivariate regression problem in high dimensions. Starting from a set of scattered data

$$D = \{(\mathbf{x}_j, y_j)\}_{j=1}^N = \left\{ \left(x_1^j, \ldots, x_d^j; y_j \right) \right\}_{j=1}^N , \tag{1.1}$$

the goal is to construct a function $g(\mathbf{x})$ such that $g(\mathbf{x}_j) \approx y_j$ in some (usually average) sense, and g provides a reasonable model when evaluated at other $\mathbf{x}$. Such problems arise frequently in statistics and machine learning.

We are interested in the case where the dimension d is large, but the underlying function that generated the data is fairly "simple." It is easy to construct a function whose discretization would require a grid with M points in each direction and, thus, $N = M^d$ samples. Since M^d is impossibly large for even moderate values of M and d, we must assume $N \ll M^d$ and, therefore, the data cannot describe such a function to begin with. Such consequences of the curse of dimensionality exclude the representation of arbitrary functions. Instead, our construction should accommodate a sufficiently rich class of functions so that we are able to form reasonable regression functions to fit the data. In this work we represent $g(\mathbf{x})$ as a sum of separable functions. This class of functions allows surprisingly accurate approximations for a wide variety of important examples, while also allowing algorithms that scale linearly in both N and d.

As a simple instructive example of the type of problems we are interested in, consider learning to touch your forehead with a finger while your eyes are closed.

*Received by the editors December 10, 2007; accepted for publication (in revised form) October 27, 2008; published electronically March 13, 2009.
http://www.siam.org/journals/sisc/31-3/71052.html
†Department of Applied Mathematics, University of Colorado at Boulder, 526 UCB, Boulder, CO 80309-0526 (beylkin@colorado.edu). The research of this author was supported by DARPA/ARO grant W911NF-06-1-0254.
‡Institut für Mathematik, Technische Universität Berlin, Sekretariat MA 3-3, Straße des 17. Juni 136, 10623 Berlin, Germany (garcke@math.tu-berlin.de).
§Department of Mathematics, Ohio University, 321 Morton Hall, Athens, OH 45701 (mjm@math.ohiou.edu). This research of this author was supported by NSF grant DMS-0545895 and DARPA/ARO grant W911NF-06-1-0254.

Before making any decisions on how to move your arm and hand, you need some estimate of the current position of your finger. Between your forehead and your finger are several joints, with (at least) $d = 10$ angles. One strategy is to use sensory input from your muscles and joints to determine these angles and then use geometry to calculate the distance (or vector) from your finger to your forehead. A second strategy, which we advocate for here, is to learn this distance function by gathering data and forming a regression function. Using your eyes, you can determine the distance that corresponds to a given set of angles (or the raw sensory input) and so acquire a data point. By moving your arm around, you can acquire a training set. After building a regression function, you can close your eyes and simply evaluate this function at the current angles to estimate the distance, instead of computing it using geometry. In this example the function has many variables but is not inherently complicated, and so our method would be appropriate.

A prerequisite for the construction of $g(\mathbf{x})$ is the ability to represent and manipulate functions of many variables. In high dimensions it appears that one is forced into either a radial (see, e.g., [16, 18]) or separable approach. The separable approach is based on the classical approximation of such a function as a separable function

$$g(\mathbf{x}) = \prod_{i=1}^{d} g_i(x_i)\,. \tag{1.2}$$

When this approximation is not good enough, it is natural to consider a sum of separable functions

$$g(\mathbf{x}) = \sum_{l=1}^{r} s_l \prod_{i=1}^{d} g_i^l(x_i)\,. \tag{1.3}$$

We call r the *separation rank*. The coefficients s_l are solely for convenience so that we can have $\|g_i^l\| = 1$. Many methods are based on this form but differ in how they use it. A tensor product basis chooses the functions g_i^l from a preselected master set of orthogonal functions, forms all combinations, and then determines the coefficients s_l. If the one-variable basis has M elements, then there are $r = M^d$ combinations, which is terribly large for even moderate parameter values. Thus, the *curse of dimensionality* manifests itself, and the full basis cannot be used. Sparse grid methods (see, e.g., [5]) use decay estimates justified by hierarchical properties to eliminate many of the combinations, resulting in a sparse tensor product decomposition that retains $\mathcal{O}(M(\log M)^{d-1})$ terms. They have been used in this context in [13, 12, 11]. Both the tensor product basis and sparse grid basis produce linear approximation methods and result in exponential growth of r with d.

In the statistics literature, representations of the form (1.3) appear under the names "parallel factorization" or "canonical decomposition"; see, e.g., [15, 20, 21, 4, 7, 25]. They are used primarily to analyze data on a grid, typically in $d = 3$. Since the goal is to interpret data, constraints on g_i^l, such as positivity when one is interested in probabilities, are often imposed. Similarly, since they describe only data on a grid, a general regression function is not built.

In this work we demonstrate a method that also uses functions of the form (1.3) but without constraints such as orthogonality or positivity on the g_i^l. By removing the constraints we switch from a linear to a nonlinear approximation method (see, e.g., [9]). In this context we call (1.3) a *separated representation*. The functions g_i^l may be constrained to a *subspace* but are not restricted to come from a particular *basis set*.

We found in [1, 2, 23] that this extra freedom allows one to find good approximations with surprisingly small r and reveals a much richer structure than one would believe beforehand. Although there are at present no useful theorems on the size r needed for a general class of functions, there are examples where removing constraints produces expansions that are *exponentially* more efficient than one would expect a priori, i.e., $r = d$ instead of 2^d or $r = \log d$ instead of d. These examples are discussed in detail in [2, 23], but we will sketch a few here as illustrations. Notice that these examples include the widely used additive and linear models as well as representations with Gaussians.

First, as a simple example, note that in our approach we can have a two-term representation

$$\prod_{i=1}^{d} \phi_i(x_i) + \prod_{i=1}^{d} (\phi_i(x_i) + \phi_{i+d}(x_i)) \tag{1.4}$$

where $\{\phi_j\}_{j=1}^{2d}$ form an orthonormal set. To represent the same function as (1.4) while requiring all factors to come from a master orthogonal set would force one to multiply out the second term and, thus, obtain a representation with 2^d terms. Thus, a function that would have $r = 2^d$ in an orthogonal basis may be reduced to $r = 2$. Second, consider the additive model $\sum_{i=1}^{d} \phi_i(x_i)$, and note that

$$\sum_{i=1}^{d} \phi_i(x_i) = \lim_{h \to 0} \frac{1}{2h} \left(\prod_{i=1}^{d} (1 + h\phi_i(x_i)) - \prod_{i=1}^{d} (1 - h\phi_i(x_i)) \right) . \tag{1.5}$$

Thus, we can approximate a function that naively would have $r = d$ using only $r = 2$. This formula provides an example of converting addition to multiplication; it is connected to exponentiation, since one could use $\exp(\pm h\phi_i(x_i))$ instead of $1 \pm h\phi_i(x_i)$. Third, notice that using the usual trigonometric identity $\sin(A + B) = \sin(A)\cos(B) + \cos(A)\sin(B)$ recursively it appears that $\sin(\sum_{j=1}^{d} x_j)$ requires $r = 2^{d-1}$. We discovered that

$$\sin\left(\sum_{j=1}^{d} x_j \right) = \sum_{j=1}^{d} \sin(x_j) \prod_{k=1, k \neq j}^{d} \frac{\sin(x_k + \alpha_k - \alpha_j)}{\sin(\alpha_k - \alpha_j)} \tag{1.6}$$

for all choices of $\{\alpha_j\}$ such that $\sin(\alpha_k - \alpha_j) \neq 0$ for all $j \neq k$ (see [23]), and, thus, only $r = d$ is needed. Fourth, note that using complex exponentials we can write

$$\sin\left(\sum_{j=1}^{d} x_j \right) = \frac{1}{2i} \exp\left(i \sum_{j=1}^{d} x_j \right) - \frac{1}{2i} \exp\left(-i \sum_{j=1}^{d} x_j \right) \tag{1.7}$$

$$= \frac{1}{2i} \prod_{j=1}^{d} \exp(ix_j) - \frac{1}{2i} \prod_{j=1}^{d} \exp(-ix_j) , \tag{1.8}$$

and, thus, we can reduce to $r = 2$. This observation generalizes to any function with a short Fourier series (the terms of which do not have to be consecutive). Note also that in this case a rotation of the coordinate axes would introduce constants in front of the x_j but would not effect r. Thus, any linear model ($g(\mathbf{x}) = \phi(\sum a_i x_i + b)$) can be accommodated, with r depending only on the complexity of the outer function ϕ,

as measured by the decay of its Fourier transform. Fifth, note that Gaussians are separable, since

$$\exp\left(-c\|\mathbf{x}-\mathbf{z}\|^2\right) = \prod_{i=1}^{d} \exp\left(-c(x_i - z_i)^2\right) . \tag{1.9}$$

By expanding a radial function in Gaussians, one can obtain a separated representation for it. Several important operators, such as the inverse Laplacian, have radial kernels and, thus, can be represented using this technique (see, e.g., [2]).

The goals of this paper are to present algorithms to construct regression functions of the form (1.3) and to give preliminary numerical evidence that such representations are worth using.

First, in section 2, we construct and present the basic algorithm for constructing a regression function of the form (1.3). We also present several variants; in particular, we show how to handle vector-valued data and how to incorporate regularization to encourage smoothness and avoid overfitting. The algorithms are linear in both N and d and so are suitable for large data sets in high dimensions. The basic algorithm depends (quadratically) on r, and, thus, the central remaining issue is how large r must be in practice.

Second, in section 3, we demonstate by numerical experiments that interesting functions can indeed be well approximated in the form (1.3) with small r. One can of course construct functions where these methods would fail. However, our experiments confirm that the class of functions that we can approximate well is wide enough to include "naturally occurring" functions of many variables, and so these methods are useful in practice. In particular, we show that our method outperforms other methods on several benchmark data sets.

Remark 1.1. It may be appropriate to first transform the data locations using a dimensionality-reduction technique such as principle component analysis (see, e.g., [16]), the Johnson–Lindenstrauss lemma [19], or manifold learning (e.g., [6]) and then apply our method in the reduced coordinates. Thus, while our method may sometimes be used instead of these techniques, it can also be used in conjunction with them.

2. Description of the algorithm.

In this section we describe the basic algorithm and several variants. First, we give the core principles, which allow us to reduce to one-dimensional subproblems. Second, we describe how to solve this one-dimensional subproblem when using a linear function space and how to set up the problem in the nonlinear case. Third, we consider how to incorporate procedures to avoid overfitting. Finally, we extend the method to vector-valued data and functions.

2.1. Core principles.

2.1.1. Data-driven inner product.

We define a pseudo inner product on functions by

$$\langle f, g\rangle_D = \sum_{j=1}^{N} f(\mathbf{x}_j) g(\mathbf{x}_j) . \tag{2.1}$$

We note that (2.1) is not a true inner product since we could have $\langle g, g\rangle_D = 0$ if the support of g is disjoint from the data. Since this pseudo inner product involves only

evaluations at the data points, we may take inner products with our data, i.e.,

$$\langle D, g\rangle_D = \left\langle \{(\mathbf{x}_j, y_j)\}_{j=1}^N, g \right\rangle_D = \sum_j^N y_j g(\mathbf{x}_j)\,. \tag{2.2}$$

Thus, we can treat the data as if it were some unknown function. In the associated pseudo norm the usual least-squares error is then simply given by

$$\left\| \{(\mathbf{x}_j, y_j)\}_{j=1}^N - g \right\|^2 = \sum_{j=1}^N (y_j - g(\mathbf{x}_j))^2\,. \tag{2.3}$$

Remark 2.1. Symmetries in the data can sometimes be built into the inner product (2.1). See [2] for an example where the antisymmetry constraint in quantum mechanics was built into a pseudo inner product.

2.1.2. Collapse to one-dimensional subproblems. We now assume that an initial g of the form (1.3) is given, with some choice of representation for g_i^l (to be discussed later). We fix the components in all directions but one and so collapse to a one-dimensional problem. For ease of exposition we describe the case for direction $m = 1$ and so fix g_m^l for $m > 1$. We define the (fixed) partial products from the remaining directions by

$$p_j^l = s_l \prod_{i=2}^d g_i^l\left(x_i^j\right)\,, \qquad l = 1, \ldots, r, \quad j = 1, \ldots, N\,. \tag{2.4}$$

The error (2.3) then reduces to

$$\sum_{j=1}^N \left(y_j - \sum_{l=1}^r p_j^l g_1^l\left(x_1^j\right) \right)^2\,. \tag{2.5}$$

To minimize (2.5) we must solve a one-dimensional least-squares problem involving r one-dimensional functions g_1^l. The method to do so will depend on the choice of representation of g_1^l, and, in particular, whether g_1^l is linear in the free parameters (see section 2.2).

2.1.3. Iterative improvement. If we can solve the one-dimensional subproblems, then we can iteratively solve such problems to reduce the error (2.3). One strategy for ordering the iteration is the well-known alternating least-squares (ALS) approach (see, e.g., [15, 20, 21, 4, 7, 25]), which was extended in [1, 2]. In this approach one minimizes using direction $m = 1$ to obtain improved g_1^l, then minimizes using direction $m = 2$ to obtain improved g_2^l, etc. and, thus, loops through ("alternates" in) the directions $m = 1, \ldots, d$. This approach is robust in that the error can never increase, but its theoretical properties are not well-understood. A second strategy is to update all directions simultaneously (in which case a parallel computer could be used). This second approach does not have the ordering bias present in ALS but has the disadvantage that it may actually increase the error.

In both cases, one should repeat this process and monitor the change in error to detect convergence. It is certainly possible to hit local minima, in which case one would need to restart with a different guess or increase r. Even when we approach the true minima, we have no reason to expect any better than linear convergence. The minimization problem can be ill-posed [8]; in section 2.3, we discuss a method to avoid overfitting that also ensures the problem is well-posed.

2.1.4. Computational cost, so far. Although we have deferred the discussion on solving the one-dimensional subproblems, it is worthwhile at this point to account for the computational cost to set up these problems. We assume that the number of ALS iterations or number of simultaneous parallel updates is K and that the cost to evaluate a single g_i^l at a single point is $\mathcal{O}(M)$. The cost to compute all p_j^l for a single m is then $\mathcal{O}(rdMN)$. If we use the parallel update rule with d processors, then the cost (per processor) is, thus, $\mathcal{O}(rdMNK)$. It would appear that the ALS method would have cost $\mathcal{O}(rd^2MNK)$ since it has a loop through the d directions. However, when we switch from, say, $m = 1$ to $m = 2$, we can simply update p_j^l by multiplying it by $g_1^l(x_1^j)/g_2^l(x_2^j)$ at cost $\mathcal{O}(rMN)$. The total cost for the ALS formulation is, thus, also

$$\mathcal{O}(rdMNK)\,. \tag{2.6}$$

Note that this cost is linear in all of the parameters.

2.2. The one-dimensional subproblem.

2.2.1. With a linear function space. We now consider how to solve the one-dimensional subproblem in section 2.1.2 in the case when g_1^l depends linearly on some set of coefficients. We assume that we are given a function space of (finite) dimension M_l in which to search for g_1^l. For example, we could choose polynomials of some degree. This space may be different for each term l in the sum, each attribute m, and in general also for each (l, m) pair. We next choose some basis $\{\phi_k^l\}_{k=1}^{M_l}$ for this function space, but we emphasize that the results are independent of the particular choice. The function g_1^l will be represented by M_l coefficients $c_l(k)$, organized in the vector $\mathbf{c}_l$. We organize our basis functions ϕ_k^l into a vector-valued function $\mathbf{\Phi}_l(x)$, defined by

$$\mathbf{\Phi}_l = \begin{bmatrix} \phi_1^l \\ \phi_2^l \\ \vdots \\ \phi_{M_l}^l \end{bmatrix}. \tag{2.7}$$

Using $(\cdot)^*$ to denote transpose, we then have $g_1^l = \mathbf{\Phi}_l^* \mathbf{c}_l$.

We will solve for the values of $\mathbf{c}_l$ for all l, so those are the free parameters with respect to which we minimize the error (2.5). Taking the gradient with respect to these parameters and setting it equal to zero, we obtain the usual linear normal equations

$$\mathbb{A}\mathbf{z} = \mathbf{b}\,. \tag{2.8}$$

Since our free parameters have two indices (l, k), the system has a natural block structure. We can hide the index k of the basis functions by using the vector notation $\mathbf{\Phi}_l$. The blocks of $\mathbb{A}$ are then defined by

$$A(l,l') = \left\langle s_l \mathbf{\Phi}_l \prod_{i=2}^{d} g_i^l, s_{l'} \mathbf{\Phi}_{l'} \prod_{i=2}^{d} g_i^{l'} \right\rangle_D = \sum_{j=1}^{N} p_j^l \mathbf{\Phi}_l \left(x_1^j\right) p_j^{l'} \mathbf{\Phi}_{l'}^* \left(x_1^j\right) \tag{2.9}$$

for $1 \le l, l' \le r$, each of which is a $M_l \times M_{l'}$-matrix. The (block) entries in $\mathbf{b}$ are defined by

$$b(l) = \left\langle s_l \mathbf{\Phi}_l \prod_{i=2}^{d} g_i^l, \{(\mathbf{x}_j, y_j)\}_{j=1}^{N} \right\rangle_D = \sum_{j=1}^{N} y_j p_j^l \mathbf{\Phi}_l \left(x_1^j\right), \tag{2.10}$$

each of which is a vector of length M_l. The matrix $\mathbb{A}$, thus, depends only on the data locations $\{\mathbf{x}_j\}$, whereas $\mathbf{b}$ also depends on the values $\{y_j\}$. Once $\mathbb{A}$ and $\mathbf{b}$ have been constructed, we will solve the system. Since we expect that after a few steps of the iteration in section 2.1.3 we will have good starting values, we use the conjugate gradient iterative method (see, e.g., [14]) to solve (2.8). Once $\mathbf{z}$ is determined, we set $\mathbf{c}_l = z(l, \cdot)$, then renormalize g_1^l, and incorporate the norm into s_l.

For the computational cost bounds we now assume $M_l = M$ for all l and that the cost to evaluate ϕ_k^l is $\mathcal{O}(1)$, which would be the case, e.g., for monomials. Given the p_j^l, it costs $\mathcal{O}(r^2M^2N)$ to compute $\mathbb{A}$ and $\mathcal{O}(rMN)$ to compute $\mathbf{b}$. We denote by S the number of conjugate gradient iterations needed, so the cost to solve the system is $\mathcal{O}(r^2M^2S)$. Although S in theory could be as many as rM, we usually have a very good starting point and so expect only a very small number to be needed. The computation cost for this method to solve the one-dimensional subproblem is, thus,

$$\mathcal{O}\left(r^2M^2(N+S)\right). \tag{2.11}$$

If we incorporate this algorithm into the overall method and account for the costs and considerations in section 2.1.4, our total cost is

$$\mathcal{O}\left(Kdr^2M^2(N+S)\right). \tag{2.12}$$

The cost is linear in both d and N, and so the method is feasible for large data sets in high dimensions.

Remark 2.2. The normalization for g_1^l is not determined by the inner product (2.1). Since s_l is not strictly necessary, we need not normalize at all; we do so only to prevent over/underflows.

Remark 2.3. It is common in machine learning that the coordinates in certain directions x_j are categories rather than numbers. In this case our function space is a vector space, so we use a basis of vectors rather than functions and index their coordinates by the categories.

2.2.2. With nonlinear dependence. If g_1^l depends nonlinearly on the parameters $\mathbf{c}_l$, then the one-dimensional subproblem in section 2.1.2 requires a nonlinear optimization (see, e.g., references in [16]). Typically, the input for a nonlinear optimization routine is the vector of errors, which in our case is

$$\{e_j\}_{j=1}^N = \left\{ y_j - \sum_{l=1}^{r} p_j^l g_1^l(x_1^j) \right\}_{j=1}^N. \tag{2.13}$$

For methods that require a derivative, we can compute

$$\frac{\partial}{\partial c_l(k)} e_j = -p_j^l \frac{\partial}{\partial c_l(k)} g_1^l\left(x_1^j\right). \tag{2.14}$$

With this information one can use one's preferred nonlinear optimization method.

Remark 2.4. If one formulates the error (2.5) using something besides least-squares, e.g., Huber-loss [16, 17] for regression or loss functions for classification like maximum likelihood or the hinge loss used for support vector machines [16], then one obtains a nonlinear optimization problem as in this section. Some loss functions, such as the hinge-loss, are not Frechet-differentiable and so require a general descent method such as subgradient-based approaches to minimize.

2.3. Avoiding overfitting. One issue to address for regression/learning algorithms is the avoidance of overfitting. As an extreme example of overfitting, one could use the function

$$g(\mathbf{x}) = \sum_{j=1}^{N} y_j \exp(-c\|\mathbf{x} - \mathbf{x_j}\|^2) \tag{2.15}$$

to represent the data. For large enough c, this function would match the given data nearly exactly but be completely unreasonable for other locations. There are two standard approaches to avoid overfitting. In parametric methods, g is constrained to be of a certain form, with only a few free parameters to determine. Assuming that the model for g is correct, there is no room to overfit. In nonparametric methods, g is chosen from a much wider class of functions, with some mechanism encouraging the choice of a nice (smooth) function. We will demonstrate how both of these approaches apply within our method.

There are two ways in which overfitting can occur in our method. The first is when r is too large. The algorithm can then attempt to fit the noise and/or match the data points individually. The parametric approach to avoid this effect is to choose r very low. This is the natural approach to take, since r is the main complexity parameter. As with all parametric methods, various more-or-less justified tests, or simple cross validation, can be used to choose the appropriate r.

The second way overfitting can occur is when there is overfitting in the one-dimensional functions g_i^l. In some sense this issue is off the topic of this paper, since the collapse to one-dimensional subproblems in section 2.1.2 allows users to choose their favorite method (see, e.g., [16]). There are, however, two natural ways to avoid overfitting within our framework. The first is to use a parametric approach and choose M_l small in section 2.2. The second is to use a nonparametric approach and incorporate regularization to encourage smoothness, as we describe next. Suppose that in section 2.2.1 we choose the basis $\{\phi_k^l\}_{k=1}^{M_l}$ ordered with smoother functions at the beginning. For example, if we are using polynomials, we could choose the Legendre polynomials and order them by degree; if we are using wavelets, we can order them with coarse-scale functions listed first. We now choose a list of penalty weights $0 \le \lambda_1^l \le \cdots \le \lambda_{M_l}^l$. Instead of minimizing (2.5) we minimize

$$\sum_{j=1}^{N} \left(y_j - \sum_l p_j^l g_1^l \left(x_1^j \right) \right)^2 + \sum_l s_l^2 \sum_k \lambda_k^l |c_l(k)|^2 \,, \tag{2.16}$$

where $c_l(k)$ are the expansion coefficients from section 2.2.1. Tracking this change into the normal equations, we see that the diagonal elements of $\mathbb{A}$ in (2.9) are modified by adding to the block $A(l,l)$ a matrix with $s_l^2 \lambda_k^l$ on the diagonal. The relative sizes of the λ_k^l can often be justified given the particular choice of a basis. The overall size, however, will likely need to be determined by cross validation. By choosing all $\lambda_k^l > 0$, this form of regularization also prevents $\mathbb{A}$ from becoming singular in the case where a basis function has support disjoint from the data and ensures the minimization problem is well-posed; see [2] for discussion on controlling condition number in this way.

2.4. Vector-valued functions. In some machine learning problems there are multiple response variables; i.e., g itself is vector-valued. One approach would be to approximate each response variable separately. In the scalar case described above,

our representation cost is rdM; so if we do v independent problems, then our cost is $vrdM$.

In this section we describe another approach, where we incorporate vector-valued functions by replacing the scalar s_l with a vector $\mathbf{s}_l$. We then have another "direction," which indexes the coordinate of the output and in which we also fit. An importance weighting for the coordinates can be included easily. Since this direction is discrete, it is natural to take the unit coordinate vectors as a basis. Our approach tries to use correlations between different response variables to get a more compact representation and results in a representation cost of $r(v + dM)$. We present a numerical example using this algorithm in section 3.3 but do not have enough experience with it to make specific claims about its effectiveness or the wisdom of looking for correlations in the response variables.

We replace (1.3) with

$$g(\mathbf{x}; n) = \sum_{l=1}^{r} s_l(n) \prod_{i=1}^{N} g_i^l(x_i), \tag{2.17}$$

define $\mathbf{g}(\mathbf{x}) = g(\mathbf{x}; \cdot)$, and replace the inner product (2.1) with $\langle \mathbf{f}, \mathbf{g} \rangle_D = \sum_{j=1}^{N} \langle \mathbf{f}(\mathbf{x}_j), \mathbf{g}(\mathbf{x}_j) \rangle$, where $\langle \cdot, \cdot \rangle$ is the vector inner product $\langle \mathbf{w}, \mathbf{z} \rangle = \mathbf{z}^* \mathbf{w}$. When fitting in direction $m = 1$, as described in section 2.1.2, the error (2.5) becomes

$$\sum_{j=1}^{N} \left\| \mathbf{y}_j - \sum_{l=1}^{r} \mathbf{p}_j^l g_1^l \left(x_1^j \right) \right\|^2, \tag{2.18}$$

since the partial product (2.4) is now vector-valued. When we fit in the new direction, which corresponds to the coordinate of the output, we define $q_j^l = \prod_{i=1}^{d} g_i^l(x_i^j)$ and the error is

$$\sum_{j=1}^{N} \left\| \mathbf{y}_j - \sum_{l=1}^{r} \mathbf{s}_l q_j^l \right\|^2. \tag{2.19}$$

If we use a linear representation, as described in section 2.2.1, the blocks of the matrix $\mathbb{A}$ in (2.9) become $A(l, l') = \sum_{j=1}^{N} \langle \mathbf{p}_j^l, \mathbf{p}_j^{l'} \rangle \mathbf{\Phi}_l(x_1^j) \mathbf{\Phi}_{l'}^*(x_1^j)$ and the vector $\mathbf{b}$ in (2.10) becomes $b(l) = \sum_{j=1}^{N} \langle \mathbf{y}_j, \mathbf{p}_j^l \rangle \mathbf{\Phi}_l(x_1^j)$. When we fit in the direction of the coordinate of the output, the matrix $\mathbb{A}$ in (2.9) becomes $A(l, l') = \sum_{j=1}^{N} q_j^l q_j^{l'}$ and the vector $\mathbf{b}$ in (2.10) becomes $b(l) = \sum_{j=1}^{N} q_j^l \mathbf{y}_j^*$, which makes the entries of $\mathbf{b}$ (row) vectors themselves.

3. Numerical results. In this section we give numerical results for several benchmark problems. We show the following:

1. Given enough noise-free data, several interesting functions can be well approximated in the form (1.3).
2. Given a smaller set of noisy data, by incorporating the method to avoid overfitting we can still fit well and in several cases outperform existing methods.

We first report the results of our exploratory testing on three commonly used synthetic data sets originating from [10]. To avoid confounding issues, we use plenty of data and do not include noise. Next, we compare the performance of our algorithm with the results in an extensive benchmark study from machine learning [22]. This study includes the synthetic data sets from [10] as well as several real world data sets. Finally, we consider an example with vector-valued data to validate the approach in section 2.4.

TABLE 3.1
Mean squared error (MSE) for fitting the Friedman1 data set with no noise, using polynomials in each direction. The degree 0 entry estimates the variance in our realization of the data.

	$r=1$	$r=2$	$r=3$	$r=4$
degree 0	23.8			
degree 1	6.3	5.9	5.9	5.9
degree 2	2.1	1.0	0.33	0.024
degree 3	2.1	1.0	0.32	0.011

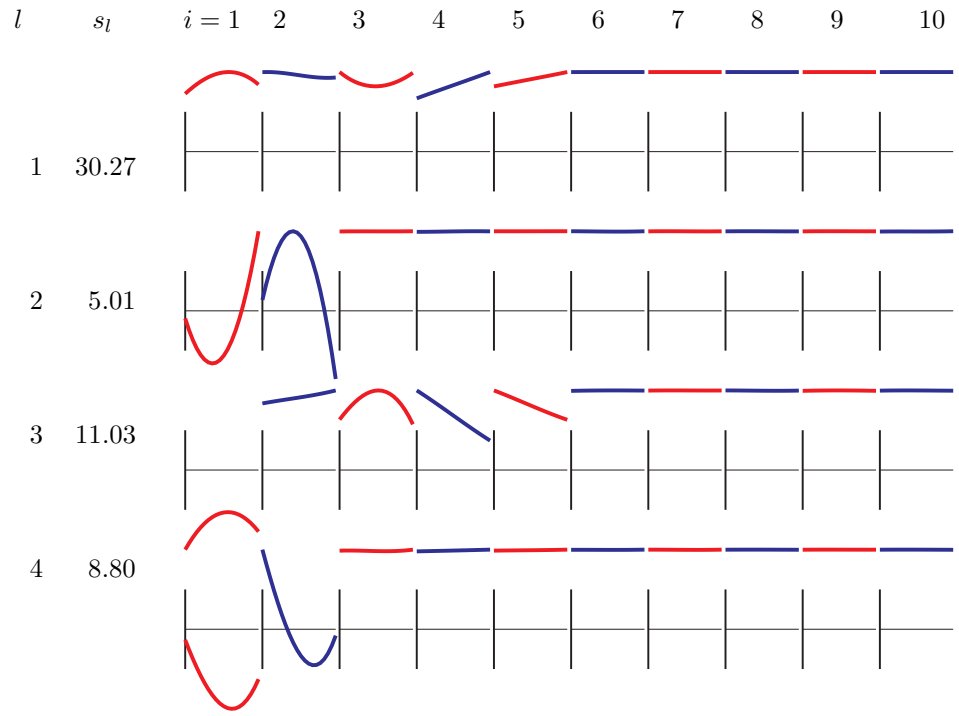

FIG. 3.1. *The function of the form* (1.3) *using polynomials of degree three that captures* 99.95% *of the variance of the Friedman1 function* (3.1). *Each subplot shows a* $g_i^l(x_i)$.

3.1. Exploratory testing on synthetic data sets. We first consider synthetic data to allow some comparison of the function of the form (1.3) produced by our approach with the real function. To avoid unwittingly constructing data customized for our approach we consider three standard test functions for regression from the literature [10]. In this section we are studying only how well we can approximate these functions, so we use $N = 20000$ data points, without any noise.

3.1.1. Friedman1 data set. This standard test is the function

$$y = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 \tag{3.1}$$

on ten variables $x_1, \ldots, x_{10}$ uniformly distributed over the ranges $0 \le x_i \le 1$, with the last five variables unused.

Choosing to use polynomials for the functions g_i^l, we obtain the results in Table 3.1. Using polynomials of degree three and $r = 4$, the model of the form (1.3) has captured 99.95% of the variance. An illustration of the function constructed is given in Figure 3.1. In the unused variables $x_6, \ldots, x_{10}$ the dependence is correctly approxi-

TABLE 3.2

MSE for fitting the Friedman2 data set with no noise, using polynomials in each direction. The degree 0 entry estimates the variance in our realization of the data.

	$r=1$	$r=2$	$r=3$
degree 0	143303		
degree 1	177	51	51
degree 2	162	21	20
degree 3	155	11	9

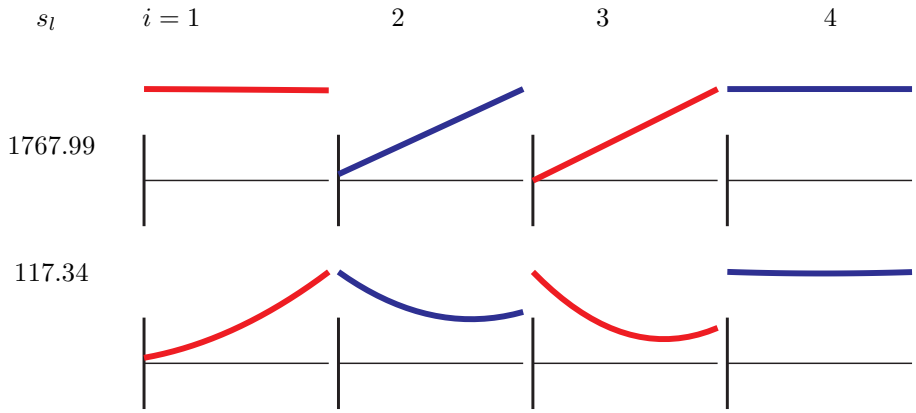

FIG. 3.2. *The function of the form* (1.3) *with* $r = 2$ *using polynomials of degree two that captures* 99.985% *of the variance of the Friedman2 function* (3.2). *Each subplot shows a* $g_i^l(x_i)$.

mated as constant, and in the variables x_3, x_4, x_5 with an additive contribution there seems to be a structure similar to (1.5).

3.1.2. Friedman2 data set. This standard test is the function

$$y = \sqrt{x_1^2 + \left(x_2 x_3 - \frac{1}{x_2 x_4}\right)^2}, \tag{3.2}$$

with the four variables uniformly distributed over the ranges $0 \le x_1 \le 100$, $40\pi \le x_2 \le 560\pi$, $0 \le x_3 \le 1$, and $1 \le x_4 \le 11$. This function has variance $375^2 = 140625$ and describes the physics of a simple alternating current series circuit [10], as does the following data set. For our test, we shifted and scaled the x_i variables to lie in $[0, 1]$.

Choosing to use polynomials for the functions g_i^l, we obtain the results in Table 3.2. Using polynomials of degree two and $r = 2$, the model of the form (1.3) has captured 99.985% of the variance. An illustration of the function constructed is given in Figure 3.2.

3.1.3. Friedman3 data set. This standard test is the function

$$y = \tan^{-1}\left(\frac{x_2 x_3 - (x_2 x_4)^{-1}}{x_1}\right), \tag{3.3}$$

with the four variables uniformly distributed over the ranges $0 \le x_1 \le 100$, $40\pi \le x_2 \le 560\pi$, $0 \le x_3 \le 1$, and $1 \le x_4 \le 11$. This function is reported to have variance $0.3^2 = 0.09$, but we measured the variance to be 0.100. Again, for our test, we shifted and scaled the x_i variables to lie in $[0, 1]$.

TABLE 3.3

MSE for fitting the Friedman3 data set with no noise, using polynomials in each direction. The degree 0 entry estimates the variance in our realization of the data.

	$r=1$	$r=2$	$r=3$	$r=4$
degree 0	0.100			
degree 1	0.045	0.035	0.032	
degree 2	0.029	0.016	0.013	
degree 3	0.023	0.0092	0.0068	0.0063
degree 4	0.021	0.0050	0.0043	0.0039
degree 5	0.021	0.0053	0.0034	0.0031
degree 6			0.0035	

TABLE 3.4

MSE for fitting the Friedman3 data set with no noise, using rational functions in each direction. Both the numerator and denominator are of the degree indicated.

	$r=1$	$r=2$	$r=3$	$r=4$
degree 1	0.018	0.0071	0.0012	0.00093
degree 2	0.018	0.0050	0.00081	0.00042
degree 3		0.0039	0.00081	0.00040

Using polynomials. Choosing to use polynomials for the functions g_i^l, we obtain the results in Table 3.3. The fitting is not as good as in the previous examples. Examining (3.3) we note that when $x_1 \approx 0$ the function is nearly discontinuous, with a jump from $-\pi/2$ to $\pi/2$ when the numerator changes sign. Thus, polynomials are not a good choice for a basis in x_1.

Using rational functions. We next chose to replace the polynomials with rational functions, which allows us to demonstrate the nonlinear one-dimensional fitting as described in section 2.2.2. The resulting errors are given in Table 3.4. Using rational functions of degree three in the numerator and denominator and $r = 4$, the model of the form (1.3) has captured 99.598% of the (measured) variance. An illustration of the function constructed is given in Figure 3.3.

Further testing, however, revealed that this method was unstable and subject to severe overfitting. The regularization mechanism we used to avoid overfitting was to penalize higher degree terms in both the numerator and denominator. With hindsight it is clear that in the denominator we should instead penalize zeros that are close to the real interval $[0, 1]$ in the complex plane so that we do not have poles or near poles. Although this seems possible, it is a bit off topic, so we stopped our testing of this approach at this point. We also note that the nonlinear fitting was much slower than the linear fitting methods.

Using a multilevel basis. Next, we tried using a multilevel basis of tent functions on the interval $[0, 1]$ as was used, e.g., in [13]. On level 0 this consists of the functions 1 and x. On level 1 we additionally include the tent function of support 1 centered at $1/2$, i.e., the line segments from $(0, 0)$ to $(1/2, 1)$ and then to $(1, 0)$. Level 2 adds two tent functions of width $1/2$, centered at $1/4$ and $3/4$, etc. The resulting errors are given in Table 3.5. Using this multilevel basis up to level 6 and $r = 4$, the model of the form (1.3) has captured 99.627% of the (measured) variance. An illustration of the function constructed is given in Figure 3.4. At higher levels, a small amount of the regularization (2.16) was used to encourage a smoother graph and suppress any basis functions that might have support disjoint from the data.

3.2. Benchmark comparisons. We now compare the performance of our algorithm to the results of a benchmark study [22]. That study considered the generated

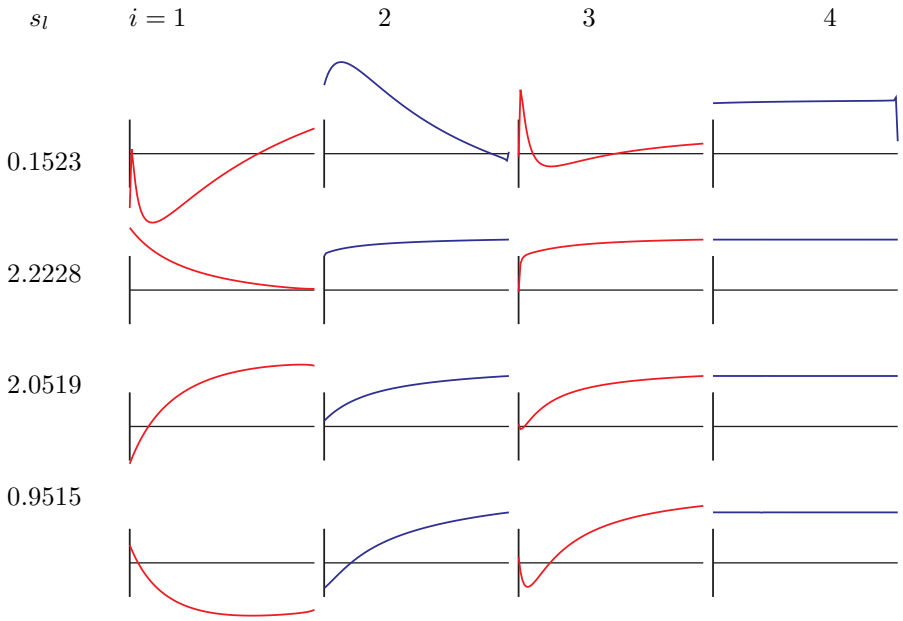

FIG. 3.3. *The function of the form* (1.3) *with* $r = 4$ *using rational function of degree two in the numerator and denominator that captures* 99.598% *of the variance of the Friedman*3 *function* (3.3). *Each subplot shows a* $g_i^l(x_i)$.

TABLE 3.5

*MSE for fitting the Friedman*3 *data set with no noise, using a multilevel basis of tent functions in each direction.*

	$r = 1$	$r = 2$	$r = 3$	$r = 4$
level 0	0.045	0.035	0.032	0.032
level 1	0.032	0.020	0.017	0.017
level 2	0.023	0.010	0.0066	0.0066
level 3	0.020	0.0063	0.0027	0.0024
level 4	0.019	0.0050	0.0016	0.00085
level 5	0.019	0.0047	0.0013	0.00046
level 6		0.0045	0.0012	0.00037

data sets from section 3.1, as well as several typical real data sets from different application domains; for the data, see http://www.ci.tuwien.ac.at/ meyer/benchdata/. Using the mean squared error (MSE) to measure the error, it empirically compared the following regression methods: linear regression, ϵ-support vector regression with a Gaussian radial basis function kernel (svm), neural networks (nnet), regression trees, projection pursuit regression (ppr), multivariate adaptive regression splines, additive spline models by adaptive backfitting (Bruto), bagging of trees, random forest (rForest), and multivariate adaptive regression trees (mart).

Tenfold cross validation was performed ten times; we report the means and medians of all 100 runs, whereas the standard deviation and interquartile range are computed with respect to the ten averages from the tenfold subsets. For comparison, we give the best result from the benchmark study and note the rank of our approach in comparison to the other methods used. Overall, we achieve the best performance of all the methods tested.

The representation (1.3) is quite flexible, but that also means there are many free parameters/choices. One needs to select the separation rank r, the one-dimensional

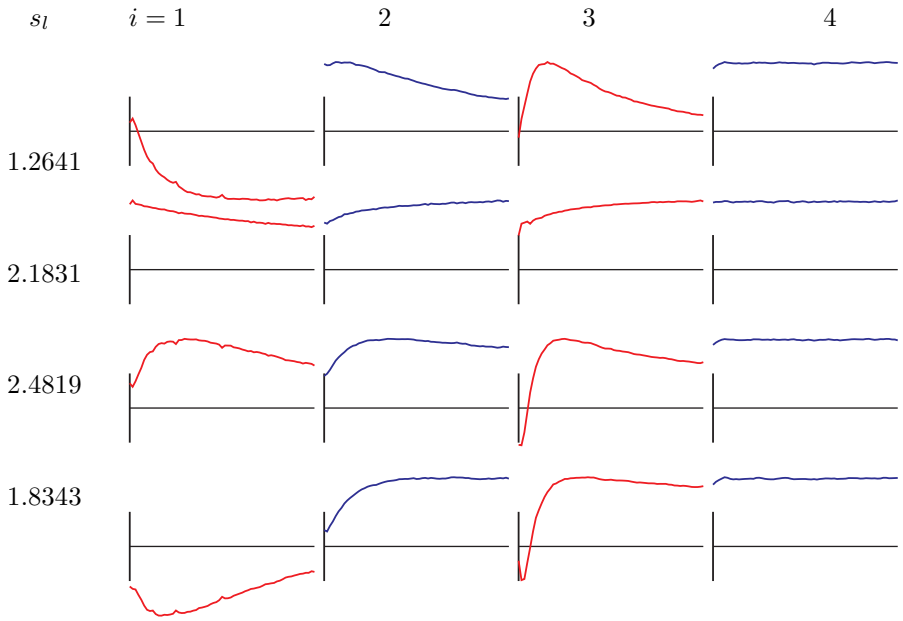

FIG. 3.4. *The function of the form* (1.3) *with* $r = 4$ *using a multilevel basis of tent functions that captures* 99.627% *of the variance of the Friedman3 function* (3.3). *Each subplot shows a* $g_i^l(x_i)$.

basis/representation (which in principle can change with l and i), and the parameters to avoid overfitting.

To choose these parameters we proceed here as in the benchmark study [22]. We split the training data and use two-thirds for learning and one-third for validation. Using this splitting we learn over a suitable range of the three involved parameters: r, the size of the one-dimensional basis, and the regularization factor. We pick the parameter set with the best results on the validation third and then learn on the whole training data and evaluate on the separate and as yet unseen test data. This result we use for comparison with the other approaches. More refined techniques for choosing good parameters might be possible, but the usual approaches of cross validation or splitting the training data into two data sets work sufficiently well for our method. We assume that this is in part due to the insensitivity of our approach to additional data positions since our representation is independent of the data locations.

Remark 3.1. Since our algorithm may encounter local minima, one should try multiple initial guesses and select the best results. We did not use or experiment with multiple guesses in our benchmark comparisons, although they might further improve the performance.

3.2.1. Synthetic data sets. For the synthetic data sets from section 3.1, the benchmark study [22] compared ten different regression methods by performing 100 repeats for training sets with 200 examples and test sets with 1000 examples, both including normally distributed noise with variance set as in [10]. Since noise is present and there are relatively few data points, we incorporated the regularization (2.16) to avoid overfitting. For each of the 100 repeats, we perform a separate parameter search as described above. In Table 3.6 we summarize the results for all three synthetic data sets.

TABLE 3.6

Summary of the results on synthetic data as described in sections 3.1.1, 3.1.2, *and* 3.1.3. *We give the mean (with standard deviation) and median (with interquartile range) of the MSE for our approach, the best results from* [22], *and our rank in comparison to all approaches used in* [22].

		Best result from [22]		Our approach	
Data set	Criteria	Method	Error	Error	Rank
Friedman1	mean	Bruto	3.22	2.44 (0.18)	1/11
	median	Bruto	3.20 (0.14)	2.34 (0.26)	1/11
Friedman2	mean	svm	18130	16897 (520)	1/11
	median	svm	17990 (760)	16658 (422)	1/11
Friedman3	mean	nnet	0.01812	0.02172 (0.00119)	2/11
	median	nnet	0.01625 (0.00129)	0.02048 (0.00174)	2/11

On the Friedman1 data set, the variance of the noise is set to 1. We chose a basis of monomials, penalized them by their degree, and scaled the penalties by the overall factor λ.

On the Friedman2 data set, the variance of the noise is set to $125^2 = 15625$, which gives a signal-to-noise ratio of 3:1. We noticed that the simple setting of polynomials of degree one and separation rank one was often chosen in the parameter search. Since the MSE is already close to the variance of the noise, using higher degree or separation rank more likely results in overfitting.

On the Friedman3 data set, the variance of the noise is $0.1^2 = 0.01$, again resulting in a signal-to-noise ratio of 3:1 (based on the reported signal variance). As discussed in section 3.1.3, polynomials are a poor choice, so we use the multilevel basis. We chose a penalty of 0 for the constant term, 1 for the x term, and then doubling at each level, and scaled the penalties by the overall factor λ.

For the first two Friedman data sets we achieve the best performance in comparison to the benchmark study [22], and on the third data set only a highly nonlinear approach achieves better results.

3.2.2. Real data sets. We now compare with some of the real data sets used in [22]. Preprocessing of the data consists of omitting missing values, like in [22], and scaling all data to $[0,1]^d$. We did not use two of the data sets since they were dominated by categorical attributes. In data sets with only two or three categorical values we use the usual binary coding scheme or choose a basis of vectors; see Remark 2.3. In our tests these two approaches gave similar numerical results.

In Table 3.7, we give the results of these tests, using polynomials and using the multilevel basis. For polynomials, when comparing the means we are among the top three methods for two of the data sets, whereas when comparing the medians we are among the top three methods for three data sets. These results indicate that our approach using polynomials is worthwhile trying on data sets, since it will at least give average results, but the promising results on the synthetic data did not transfer fully to real data. The situation changes when we use the multilevel basis. We achieve lower errors than before, in some cases quite significantly lower, on all data sets. Comparing the median, the preferred measurement in [22], we are now among the top two methods for five of the six data sets. This is the best overall performance among all approaches. Using the mean we also achieve the best performance among the approaches, achieving the lowest results for two of the data sets.

As shown in (2.12), the complexity of our approach is linear in the number of data and the number of attributes and quadratic in the rank and the size of the one-dimensional basis chosen. Both neural networks and support vector machines are

TABLE 3.7

Results on real data sets from the study [22]. *We give the mean (with standard deviation) and median (with interquartile range) for the best results from* [22] *and for our approach using polynomials and using the multilevel basis, and our rank in comparison to all approaches used in* [22].

		Best result from [22]		Us, polynomials		Us, multilevel	
Data set	Criteria	Method	Error	Error	Rank	Error	Rank
abalone	mean	nnet	4.31	4.52(0.04)	4/9	4.47(0.03)	3/9
	median	nnet	4.29(0.03)	4.46(0.06)	3/9	4.41(0.04)	2/9
autompg	mean	svm	7.11	7.46(0.22)	4/9	6.99(0.27)	1/9
	median	nnet	6.37(0.85)	7.09(0.30)	5/9	6.48(0.37)	2/9
boston-	mean	svm	9.60	10.19(0.79)	2/9	9.67(0.59)	2/9
housing	median	svm	8.01(0.72)	8.57(1.14)	2/9	8.12(0.40)	2/9
cpu	mean	ppr	3.16	3.24(0.94)	2/9	2.85(1.02)	1/9
($\times 10^3$)	median	rForest	1.27(0.65)	1.84(0.89)	2/9	1.72(1.05)	2/9
cpuSmall	mean	mart	7.55	9.50(0.58)	5/10	7.98(0.07)	2/10
	median	mart	7.53(0.12)	9.16(0.48)	5/10	7.98(0.09)	2/10
SLID	mean	rForest	34.1	39.3(0.20)	6/9	39.5(0.16)	6/9
	median	rForest	33.3(2.99)	39.6(0.26)	6/9	39.7(0.13)	6/9

nonlinear in the number of data but linear in the number of attributes. Mart is linear in both, random forest is $N \log N$ in the number data and scale with the square root of the number of attributes, and projection pursuit regression scales $N \log N$ and linear in the number of attributes. All methods involve other complexities in the solution process as well.

We observe run times for the solution of one problem for one parameter set between seconds and minutes, depending on the actual data set and the parameters of our apprach. These run times are quite preliminary since our current implementation is purely in the scripting language python. The run times of loops in python can often be improved by one or two orders of magnitude by using a compiled language for these computations.

3.3. Vector-valued data. We next consider a case with vector-valued data, with the simple goal of validating the method in section 2.4. We do not attempt to determine if this method performs better than fitting each vector entry separately.

The data is from a helicopter flight project [24], and the task is to use the current state to predict subdynamics of the helicopter for one time step later, in particular, its yaw rate, forward velocity, and lateral velocity. We found that simply using the values of these subdynamics in the current state as the predictor captures 99.969% of the variance, so we chose to subtract off the values in the current state and use the difference as the response variables. The noise level in the data is not known nor is the magnitude of the effect of influences not included in the state, such as wind. Again we linearly transform the domain of the predictor variable to $[0, 1]^d$.

For parameter fitting we again split the training data of 40000 into two-thirds for learning and one-third for validation. In Table 3.8 we give the results using the multilevel basis. At level five and separation rank six, we now learn on the 40000 training data and evaluate on 4000 as yet unseen testing data and achieve an MSE of 0.00099, which means the model of the form (1.3) has captured 81% of the variance. We also measured a mean absolute error of 0.0089 for the yaw rate, 0.0116 for the forward velocity, and 0.0172 for the lateral velocity; we compare with the values of 0.0083, 0.0147, and 0.0185, respectively, obtained in [3]. Note that if the variances for the response variables are different, which seems to be the case for this data set,

TABLE 3.8
MSE for fitting the helicopter data set, using a multilevel basis of tent functions in each direction.

	$r=1$	$r=2$	$r=3$	$r=4$	$r=5$	$r=6$
variance	0.0052					
level 0	0.0043	0.0037	0.0028	0.0018	0.0017	0.0016
level 1	0.0041	0.0037	0.0027	0.0018	0.0016	0.0014
level 2	0.0044	0.0034	0.0026	0.0016	0.0014	0.0013
level 3	0.0043	0.0030	0.0022	0.0015	0.0013	0.0012
level 4	0.0041	0.0030	0.0021	0.0013	0.0012	0.0011
level 5	0.0039	0.0032	0.0022	0.0013	0.0012	0.0010

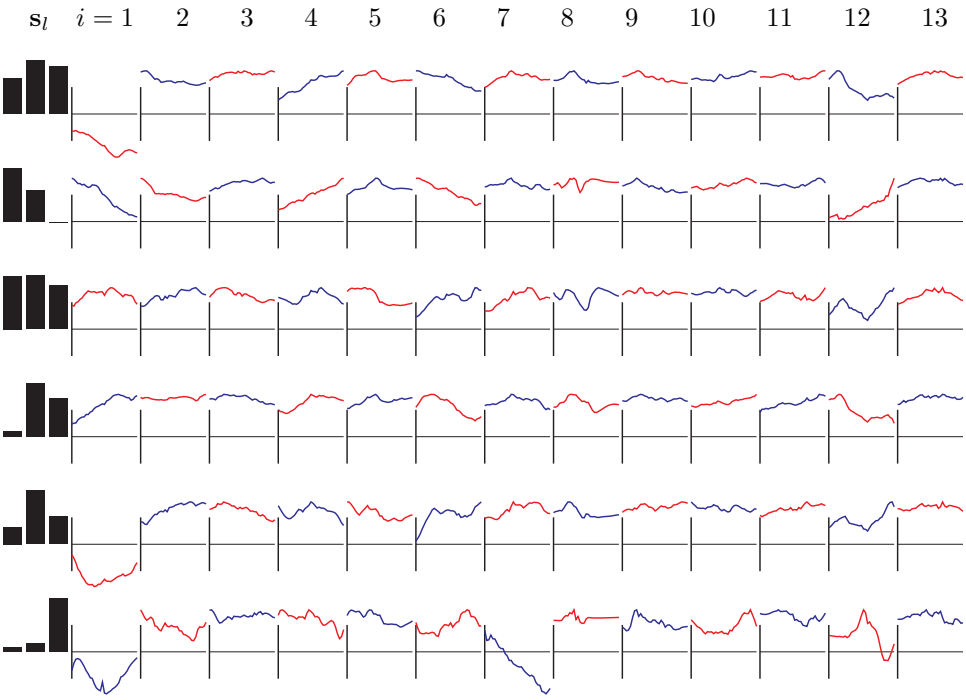

FIG. 3.5. *The vector-valued function of the form* (1.3) *with* $r=6$ *using the multilevel basis with level five that captures* 81% *of the variance of the helicopter data set. The first column shows* $\mathbf{s}_l$*, and the other subplots show* $g_i^l(x_i)$.

one can scale them to zero mean and variance one to allow a similar error reduction over all response variables. An illustration of the function constructed is given in Figure 3.5.

Acknowledgments. We would like to thank Aleksandra Orlova, who did background research on this project while a Master's student at Ohio University.

REFERENCES

[1] G. BEYLKIN AND M. J. MOHLENKAMP, *Numerical operator calculus in higher dimensions*, Proc. Natl. Acad. Sci. USA, 99 (2002), pp. 10246–10251.
[2] G. BEYLKIN AND M. J. MOHLENKAMP, *Algorithms for numerical analysis in high dimensions*, SIAM J. Sci. Comput., 26 (2005), pp. 2133–2159.
[3] S. BÖRM AND J. GARCKE, *Approximating Gaussian processes with* H^2*-matrices*, in Machine Learning: ECML Proceedings of the 18th European Conference on Machine Learning,

Warsaw, Poland, September 2007, J. N. Kok, J. Koronacki, R. L. de Mantaras, S. Matwin, D. Mladen, and A. Skowron, eds., Lecture Notes in Artificial Intelligence 4701, Springer, Berlin, 2007, pp. 42–53.

[4] R. Bro, *Parafac. tutorial & applications*, in Chemom. Intell. Lab. Syst., 38 (1997), pp. 149–171; also available online from http://www.models.kvl.dk/users/rasmus/presentations/parafac_tutorial/paraf.htm.

[5] H.-J. Bungartz and M. Griebel, *Sparse grids*, Acta Numer., 13 (2004), pp. 147–269.

[6] R. R. Coifman and S. Lafon, *Diffusion maps*, Appl. Comput. Harmon. Anal., 21 (2006), pp. 5–30.

[7] L. De Lathauwer, B. De Moor, and J. Vandewalle, *On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors*, SIAM J. Matrix Anal. Appl., 21 (2000), pp. 1324–1342.

[8] V. de Silva and L.-H. Lim, *Tensor rank and the ill-posedness of the best low-rank approximation problem*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 1084–1127.

[9] R. A. DeVore, *Nonlinear approximations*, Acta Numer., 7 (1998), pp. 51–150.

[10] J. H. Friedman, *Multivariate adaptive regression splines*, Ann. Statist., 19 (1991), pp. 1–141.

[11] J. Garcke, *Regression with the optimised combination technique*, in Proceedings of the 23rd International Conference on Machine Learning (ICML 2006), Pittsburgh, PA, 2006, ACM Press, pp. 321–328.

[12] J. Garcke and M. Griebel, *Classification with sparse grids using simplicial basis functions*, Intelligent Data Anal., 6 (2002), pp. 483–502.

[13] J. Garcke, M. Griebel, and M. Thess, *Data mining with sparse grids*, Computing, 67 (2001), pp. 225–253.

[14] G. Golub and C. V. Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, MD, 1996.

[15] R. A. Harshman, *Foundations of the Parafac Procedure: Model and Conditions for an "Explanatory" Multi-mode Factor Analysis*, Working Papers in Phonetics 16, UCLA, Los Angeles, 1970, http://publish.uwo.ca/∼harshman/wpppfac0.pdf.

[16] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, Springer Ser. Statist., Springer, New York, 2001.

[17] P. J. Huber, *The* 1972 *Wald lecture. Robust statistics: A review*, Ann. Math. Statist., 43 (1972), pp. 1041–1067.

[18] A. A. Jamshidi and M. J. Kirby, *Towards a black box algorithm for nonlinear function approximation over high-dimensional domains*, SIAM J. Sci. Comput., 29 (2007), pp. 941–963.

[19] W. B. Johnson and J. Lindenstrauss, *Extensions of Lipschitz mappings into a Hilbert space*, in Conference in Modern Analysis and Probability (New Haven, CT, 1982), Contemp. Math. 26, Amer. Math. Soc., Providence, RI, 1984, pp. 189–206.

[20] P. M. Kroonenberg and J. de Leeuw, *Principal component analysis of three-mode data by means of alternating least squares algorithms*, Psychometrika, 45 (1980), pp. 69–97.

[21] S. E. Leurgans, R. A. Moyeed, and B. W. Silverman, *Canonical correlation analysis when the data are curves*, J. Roy. Statist. Soc. Ser. B, 55 (1993), pp. 725–740.

[22] D. Meyer, F. Leisch, and K. Hornik, *The support vector machine under test*, Neurocomputing, 55 (2003), pp. 169–186.

[23] M. J. Mohlenkamp and L. Monzón, *Trigonometric identities and sums of separable functions*, Math. Intelligencer, 27 (2005), pp. 65–69; also available online from http://www.math.ohiou.edu/∼mjm/research/sine.pdf.

[24] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, *Autonomous inverted helicopter flight via reinforcement learning*, in Proceedings of the 11th International Symposium on Experimental Robotics, Singapore, Springer, 2004.

[25] A. Smilde, R. Bro, and P. Geladi, *Multi-way Analysis. Applications in the Chemical Sciences*, John Wiley & Sons, Chichester, 2004.