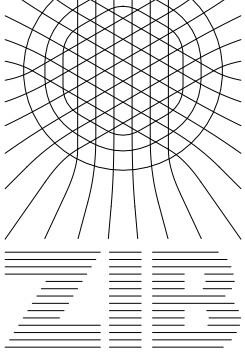
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7, D-14195 Berlin-Dahlem



Jörg Heroth

Are Sparse Grids Suitable for the Tabulation of Reduced Chemical Systems ?

Technical Report TR 97-2 (March 1997)

Are Sparse Grids Suitable for the Tabulation of Reduced Chemical Systems ?

Jörg Heroth

Abstract

Reduced chemical systems are used in the numerical simulation of combustion processes. An automatic approach to generate a reduced model is the so-called intrinsic-low-dimensional manifold (ILDM) by Maas and Pope. Thereby, the system state is tabulated as a function of some parameters. The paper analyses the storage requirements for usual interpolation schemes and for sparse grids. It also estimates the response time and gives a short description of an implementation.

Contents

Introduction 2

1. Storage Requirement 3

2. Response Time 7

3. Implementation 9

4. Interface 14

References 16

Introduction

Databases are often used in the simulation of complex processes to store some specific system data that can be computed in advance and reused for a lot of subsequent computations. It is much more efficient to compute those data once and for all then to generate them anew in each calculation they are needed for.

We focus here on a problem arising in the simulation of combustion processes. In technically relevant processes, such as the combustion in a diesel engine, complex fuels have to be considered leading to models with often serveral hundreds or even more than thousand chemical species. These models can be reduced by intrinsic low dimensional manifolds (ILDM), which were suggested by Maas and Pope [8]. Thereby, only a small number of parameters is needed to describe the chemical system to a certain accuracy if the system state is tabulated as a function over those parameters [7]. In the simulation of a diesel engine, for example, three reaction progress variables could be chosen as parameters together with enthalpy, pressure and the mixture fraction. This requires to tabulate a function with values in $\mathbf{R}^n$, $n \geq 1000$, over a domain $U \subset \mathbf{R}^6$.

The usage of sparse grids for the tabulation of these functions was suggested in [6] because they asymptically need far less knots than usual grids in higher dimensions. In Sec. 1, the question is discussed to what extent sparse grids can reduce the enormous amount of storage space. The time to access the data is estimated in Sec. 2 where it turns out that the complexity of the problem is not only determined by the amount of storage but by the speed of vector interpolation as well. Sec. 3 and 4 describe the implementation and the FORTRAN interface.

1 Storage Requirement

Mathematically speaking, the problem is to implement a function

$$f : \quad U \to \mathbf{R}^n$$

wherein U is a subset of $\mathbf{R}^d$ and f is sufficiently smooth. If the set U is very regular such as an interval, or a rectangle, the problem is well known and has already been treated by Gauß or Lagrange. As we are mainly interested in technically relevant accuracies like 1% or 5%, it does not seem to be very hard. But it gets more difficult if d and n are large or if the geometry of U is complicated.

The domain of an intrinsic low dimensional manifold can be quite complicated due to the highly nonlinear structure and singularities at the boundary. Because we want to step directly into applications, we do not try to resolve the whole domain, but suppose that a subdomain can be figured out which is sufficiently large for applications and can be transformed on a cube easily. We therefore assume U to be the most simple domain at all, the unit cube.

Nevertheless, the problem remains difficult due to the enormous amount of storage space needed. Coming back to the diesel engine example, we assume $d = 6$ and $n = 1000$. If we knew that f is an exponential or a trigonometric function, we could take special functions as an interpolation basis, but we do not know anything of this kind in our case. Thus the easiest approach to our problem is the application of a tensor product interpolation scheme with N points in each direction, for instance with splines [3] or other polynomials. We thus have nN^d numbers to store. Because high accuracy is not required, we confine ourselves to single precision representation (4 bytes per number) and end up with $4nN^d$ bytes to store. Fig. 1 shows the amount of storage space in megabytes for different N.

N	3	4	5	6	7	8	9	10
MB	3	16	63	187	470	1050	2125	4000

Figure 1: Megabytes for N knots per direction in a tensor product grid.

A really good workstation will allow a table with up to 5 or perhaps even 6

knots per direction to be stored in the fast random access memory (RAM). If hard disk storage is taken into account, 9 or maybe 10 knots are possible.

Piecewise polynomial interpolation schemes on adaptively refined rectangular meshes as used in FEM have turned out to be much more powerful than the tensor product approach. Starting with a very coarse grid that consists of a few rectangles only (maybe 1), the value $f(p)$ for each vertex p is computed and stored. Then the errors on each rectangle are estimated and all rectangles that do not meet a prescribed accuracy are marked. In the next step each marked rectangle, called father, is subdivided into 2^d smaller rectangles called sons. Iterating this procedure a whole tree of subdivided rectangles is built up.

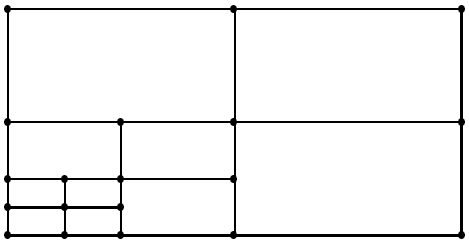


Figure 2: Refined rectangular mesh

The usual procedure to get a value at a point θ from the table is as follows: First, the tree is searched for the smallest rectangle that contains θ. Second, the data at the vertices is looked up. After that, a tensor product interpolation scheme on the rectangle yields the result. If only the smallest rectangle is taken, linear interpolation is used; if the father is considered also, a quadratic tensor product polynomial can be set up on the father rectangle. Considering neighbors or further fathers, cubic or quartic polynomials can be built up.

A table with a polynomial interpolation scheme as above realizes an approximation $\tilde{f}$ to f for which the following result is known to hold

$$\| f(\theta) - \tilde{f}(\theta) \| \ \leq \ C\, h^p. \tag{1}$$

Herein p is the order of our scheme, h is the meshsize of our grid and the constant C depends on the dimension d, the function f and p. Typically p has a close connection to the degree deg of the interpolating polynomials, we

usually have $p = deg + 1$ that is order $p = 2$ for linear polynomials, $p = 3$ for quadratic and so on.

Because too many interpolation knots cannot be allowed as seen in Fig. 1, $h = 0.1$ will be a typical meshsize in the grid for $d = 5$ or $d = 6$. Thus a crude approximation can be expected only for low orders unless the function f is extremely smooth. If f is smooth enough, technical tolerances such as 1% or 10% might be met.
An adaptive grid refinement is no real remedy against the huge number of knots because it is not possible to adapt the grid to all n components at the same time. It may help to distribute the local errors better and cut off some error peaks that are too large but it will not reduce the number of knots drastically.

Is it possible to get rid of the curse of dimensionality? A new method, the so-called *sparse grids*, was introduced by ZENGER in 1990 [12]. This method is based on a hierarchical basis decomposition of the approximation space used for interpolation and leaves out certain knots in the above scheme. BUNGARTZ proved in [1] that in sparse grids the total number of knots only grows with order $O(N(\log_2 N)^{d-1})$ instead of $O(N^d)$ for arbitrary dimension d. A meshsize $h = 2^{-l}$ and $N = h^{-1}$ is assumed here. The approximation error is only diminished by a logarithmic factor

$$\| f(\theta) - \bar{f}(\theta) \| \;\leq\; \bar{C}\, h^p\, (\log_2 h^{-1})^{d-1} \tag{2}$$

if an order p interpolation scheme is applied as shown in [2]. These order arguments seem very promising, but in our case only large meshsizes $h \approx 0.1$ are possible and the order arguments do not show anything. Let for instance be $d = 6$ as above and allow a meshsize $h = 0.125$, then the logarithmic term yields a factor 729 possibly dominating the error reduction by the h^p term. In order to decide which method should be taken, we rewrite equation (2) with a new constant $\hat{C}$ as $\|\Delta f(\theta)\| \leq \hat{C} h^p$ and compare the constants C and $\hat{C}$. Comparable values for these constants derived by the same mathematical techniques are given for $p = 2$ in [1] and for $p = 3$ in [2]. With the definitions

$$\begin{aligned} B_2(l,d) &= \left(1 + \sum_{i=1}^{d-1} \left(\frac{3}{4}\right)^i \begin{pmatrix} l+i-1 \\ i \end{pmatrix}\right) \\ B_3(l,d) &= \left(1 + \sum_{i=1}^{d-1} \left(\frac{7}{8}\right)^i \begin{pmatrix} l+i-1 \\ i \end{pmatrix}\right) \end{aligned}$$

for the meshsizes $h = 2^{-l}$, we have in the case $p = 2$

$$\begin{aligned} C_2 &= \|\frac{\partial^{2d} f}{\partial^2 x_1 \dots \partial^2 x_d}\|_\infty \frac{d}{6^d} \\ \hat{C}_2 &= \|\frac{\partial^{2d} f}{\partial^2 x_1 \dots \partial^2 x_d}\|_\infty \frac{B_2(l,d)}{6^d} \end{aligned}$$

and in the case $p = 3$

$$\begin{aligned} C_3 &= \|\frac{\partial^{3d} f}{\partial^3 x_1 \dots \partial^3 x_d}\|_\infty \frac{d}{14^d} \\ \hat{C}_3 &= \|\frac{\partial^{3d} f}{\partial^3 x_1 \dots \partial^3 x_d}\|_\infty \frac{B_3(l,d)}{14^d}. \end{aligned}$$

Fig. 3 shows the the quotient $q_3 = \hat{C}_3/C_3$ for $d = 6$ and $d = 4$ and different meshsizes $h = 2^{-l}$. It is immediately clear that in the interesting domain

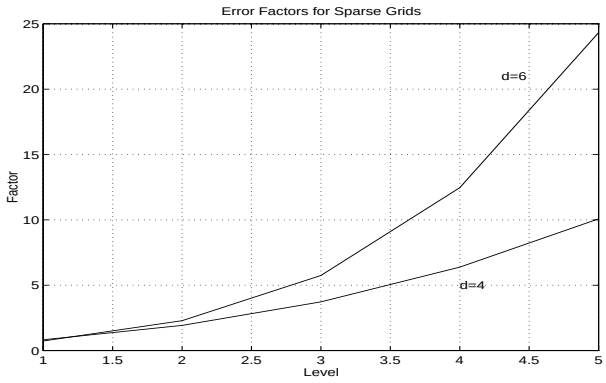


Figure 3: Error factor q_3 for $d = 6$ and $d = 4$

$2 \le l \le 5$ sparse grids need one, two or even three levels more to meet the same accuracy requirement as the usual full grid. As sparse grids do need much less knots per level than the associated full grids, they meet a prescribed tolerance of 1% or 5% with as many or even fewer knots than the full grids but the difference in the total number of knots does not exceed $\pm 10\%$. This has also been confirmed by numerical experiments with different functions. Sparse grids therefore provide no real improvement for technical tolerances between 0.1% and 5% although the asymptotic order estimates suggest that they should be much better. Additionly, the implementation is much more difficult than for the simple full grids.

2 Response Time

The storage requirements of a table in dimension $d = 5$ or $d = 6$ can easily exceed any resource if the function f is complicated. But we also have to consider the amount of time that is needed to compute a value $\tilde{f}(\theta)$ from our table. In our case it turns out that the response time is very critical.

In the numerical simulation of a combustion process, partial differential equations such as the Navier-Stokes equations with nonlinear reaction terms have to be solved in a two or three dimensional domain. In order to do this with a finite difference method, a mesh of discrete points is introduced and the partial derivatives are replaced by differences between the values at the chosen meshpoints. After this, a very large nonlinear system of ordinary differential equations remains to be solved. Roughly speaking, this is done by evaluating the rates of change at each point in our mesh and adding those to the current state.

Typical FDM or FEM meshes consist of a few thousands up to over a million knots. Let us assume that we deal with a mesh of $K = 10^4$ knots in two space dimensions. At each knot (x_1, x_2) we have a certain system state given by the parameters $(\theta_1, \ldots, \theta_d)$. In order to compute the rate of change of our state, we have to look up the tabulated values $f(\theta)$ for all points (x_1, x_2) in the discretization mesh. Therefore, our table has to answer K requests in each timestep of our integrator. As our integrator has to do a lot of timesteps, we want our K requests to be answered very fast, in one or two seconds, say. As a consequence we require that the whole table should be stored in the RAM which is 10^5 times faster than the storage on harddisk. But even if we set ahead a fast storage, a lot of time is spent in the numerical calculations.

How much numerical work in terms of floating point operations (flops) has to be done to answer a single request? An order p interpolation method requires p^d knots to set up the interpolation scheme. Numbers are given in Fig. 4. Taking for instance the Lagrangian basis defined by $L_i(z_j) = d_{ij}$, the contributions of each knot must be added up:

$$\tilde{f}(\theta) := \sum_{i=1}^{p^d} L_i(\theta) f(z_i)$$

p \ d	2	3	4	5	6
2	4	8	16	32	64
3	9	27	81	243	729
4	16	64	256	1024	4096
5	25	125	625	3125	15625

Figure 4: Interpolation knots of an order p method in dimension d

where $f(z_i)$ denotes the data stored at the interpolation knot z_i. Now each data has n components. Thus each evaluation of $\tilde{f}(\theta)$ requires p^d axpy operations[1] with vectors of length n. Totally

$$A = p^d n K$$

flops must be performed in each timestep.
Let us assume $d = 6$ in our diesel example. Using an order $p = 5$ method to save storage space, we have to do 15625 axpy operations with vectors of length $n = 10^3$ for one evaluation of $\tilde{f}(\theta)$. That is about 15 Mflops per evaluation. Thus 150 Gflops would be necessary for $K = 10^4$ requests during one timestep of integration. A real supercomputer[2] might do this in 5 seconds whereas a very good single processor workstation would need about 100 minutes to do one single timestep.
Thus high orders are clearly very expensive and the idea of choosing higher order methods to reduce the storage requirement as proposed in [10] is certainly misleading. Nothing would be different in our example if the table was stored as a single tensor product polynomial of degree $deg = 4$ without any grid or refinement. So we are confined to lower orders. Although, for $d = 6$, an interpolation scheme with order $p = 3$ is about 20 times faster than the order $p = 5$ method, it would still take 5 minutes on a really good workstation to answer all requests needed in a single timestep. So the computational limitations are not only determined by the storage requirement but by the speed of the vector interpolation as well.

In Sec. 1, it is shown that sparse grids have no real advantage over usual interpolation techniques as far as the storage is concerned. What about the acess time for sparse grids? Due to the hierarchical basis representation,

[1] An axpy operation is $y = y + a \cdot x$ and is counted with n flops.
[2] A CRAY T3D machine with 1000 processors in this case.

the value $u_n = f(\theta)$ on level n is evaluated as a sum of contributions from different spaces:

$$u_n \;=\; \sum_{i=1}^{r} u_i$$

We have $r \geq p^d$ for an order p method on the lowest level already. But in contrast to the nodal representation used above, new subspaces and new contributions u_i come in on each sparse grid level possibly increasing r far beyond the p^d mark. The increasing length of the sum to be evaluated in the interpolation may be neglectable in the scalar case but it slows down the interpolation time dramatically if n becomes considerably large. So sparse grids are not useful to us.

Remark. RHEINBOLDT has reported that a multidimensional pathfollwing method has been designed and implemented in the package MANPACK recently [11]. Pathfollowing methods are very elegant in computations on implicitly defined surfaces and may reduce the amount of storage space needed. But they will be restricted to lower orders as well and face the same interpolation problems. We therefore do not believe that there will be a real advantage over an adaptively refined grid as far as storage and interpolation is concerned. But pathfollowing methods may be a worthwhile tool in the computation of the domain U, for instance ALCON [4] has been used a lot for this task.

3 Implementation

An object orientated implementation in C++ is the easiest and best way to transfer the adaptive grid structure described in Sec. 1 directly into a module. Nevertheless, the FORTRAN subroutines in the BLAS and LAPACK packages are used to do the linear algebra work because they work fast and reliable and optimized versions are available on many machines. A very easy access to the table module from FORTRAN or C is provided by a documented interface, which is described in Sec. 4.

Most interesting to the user is the class *Table* that performs all necessary

tasks.

```
class Table {
  public:
    Table(const char* dir, int d, int n);
    virtual   ~ Table();
              ⋮
    Bool PrepareInterpolation(Vector<double> &p, Vector<double> &pp);
    void GetValue(Vector<double> & u);
    void GetJacobian(Matrix<double> & J);
    void GetHessian(Matrix<double> & H);
              ⋮
    Bool Write();
    Bool Read();
              ⋮
};
```

In the construction, a table must be given three parameters. Each table needs its own home directory *dir* where the data is stored, a dimension parameter d for the dimension of the domain and a dimension parameter n for the data length.
The interpolation is done in two steps. The function *PrepareInterpolation* is called first with the point p at which values are to be computed. The function checks if p is within the domain and transforms p onto the unit cube. If p is not within the domain, pp is the projection of p onto the boundary and all subsequent interpolations are done for pp instead of p. After that, the function sets up the interpolation cube and searches the database for the data.
Having prepared the interpolation tool, the user can obtain the interpolated value $u = \tilde{f}(p)$, the Jacobian matrix $J = D\tilde{f}(p)$ and the Hessian matrices $H = (D^2\tilde{f}_1, \ldots, D^2\tilde{f}_n)$. The Jacobian matrix J is stored columnwise as usual in FORTRAN. H is an (d^2, n) matrix and has in its k-th column the symmetric (d, d) Hessian matrix of the k-th component $\tilde{f}_k$. A tensor product interpolation scheme with piecewise quadratic polynomials (order 3) is applied by default. It has only order $p = 2$ for the Jacobian and $p = 1$ for the

Hessians, of course. Therefore the table must be rather accurate (0.1% or better) to produce a useful Jacobian. For useful Hessian matrices an even much more accurate table will be necessary in most cases. Compare example 2 to get an impression. To accelerate the interpolation, linear polynomials could be chosen, but then no Jacobians and Hessians are available.

The user does not need to know the ingredients of a table for an immediate application. But to ease the adaptation of the code to future needs, a short description of some details is given. A *Table* consists of a class *Domain* and a class *UnitTable*. The *Domain* performs the transformations between the domain and the d dimensional unit cube:

```
class Domain {
  public:
    Domain(int d);
    virtual   ~Domain();
              ⋮
    Bool TransformOnUnitCube(Vector<double> &p,Vector<double> &x);
    void TransformOnDomain(Vector<double> & x, Vector<double> & p);
    void TransformJacobian(Matrix<double> & M, Matrix<double> & J);
    void TransformHessian(Matrix<double> & M, Matrix<double> & H);
    void ReadDomain(const char *dir);                    ⋮
};
```

Only linear domains (parallelepipeds) are implemented up to now. The domain description is given by the vertex with the smallest coordinates and the d spanning vectors. The domain is read in from the file *domain.dat* in the table directory *dir* during the construction of the table. A more complicated domain can be treated by subdividing the domain in some linear ones or by changing the implementation in *domain.cc*. Other parts of the code only need the five element functions given above.

The *UnitTable* consists of the database *DataArray* and the refinement structure *RefTree*. It is derived from the base class *ITool* that does all interpolation tasks. The *RefTree* is a very simple tree structure. Each knot is a pointer to the 2^d sons if the corresponding cube is refined and 0 otherwise. A special

cube in the tree can be referenced by a sequence $(s_1, s_2, \ldots)$ which denotes the son s_2 of the son s_1 of the unit cube. The *ITool* sets up a reference element and computes the interpolated value according to the nodal basis of the element.

The *DataArray* is an associative array with a fast hierarchical searching algorithm that can be visualized in Fig. 5. In dimension $d = 1$, the database

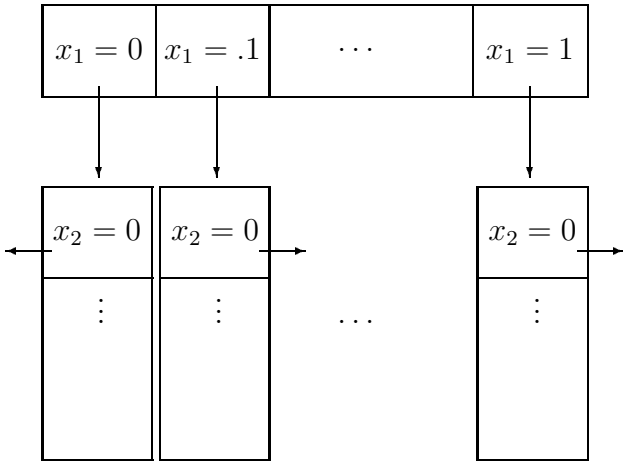


Figure 5: Data structure of the associative array

consists of a vector of knots that contain a coordinate value and a pointer to the data. In dimension $d = 2$, each pointer of a knot of the vector for $d = 1$ points to a vector of knots again. A recursive structure for higher dimensions is thus obtained. The searching for the data at $(x_1, \ldots, x_d)$ proceeds as follows: Take the vector for $d = 1$ and get the knot with the coordinate x_1. Then get the vector for $d = 2$ and get the knot with the coordinate x_2 and so on. In each dimension a fast bisection algorithm is applied to find the corresponding knot.

Finally, we want to describe how the setup of a table can be done. A special class *Setup* is designed as a friend of *Table* and *UnitTable* and has access to the private data of those classes.

```
typedef void (*IFunctionPtr)(const int* d, const double* p, const int* n,
                             const double* y0, double *y, double* rpar, int* ipar, int* err);
```

```
class Setup {
   public:
      Setup(Table* table, IFunctionPtr ptr, double* rpar, int* ipar);
      ~Setup();

      Bool Initialize();
      Bool RefineLevel(int l);
      Bool SetTable();

      void SetTolerance(double tol);
      void SetScalingThreshold(double s);
      void SetMaxNumberOfKnots(int max);
            ⋮
};
```

The function $y = f(p)$ is given as a pointer to the *Setup* class. It has the usual FORTRAN or C calling convention. An initial vector y_0 is passed to the function that contains the interpolated value from the table set up so far. The parameters *rpar* and *ipar* are not used in the table setup, they can be used to transfer special parameters from outside to the function. See the examples and the remarks and hints in the file *Interface.cc* for a detailed description of all parameters.
The *Setup* class is associated with exactly one table. A pointer to that table is handed over in the construction. There are two functions to set the desired relative tolerance *tol* and a certain scaling parameter *scal*. All errors are computed with respect to a scaled sup-norm

$$\|u - \tilde{u}\| \;:=\; \sup_{i=1,\ldots,n} \frac{|u_i - \tilde{u_i}|}{\max(|u_i|, scal)} \quad \stackrel{!}{\leq} \quad tol$$

This is equivalent to choosing an absolute tolerance $atol = tol \cdot scal$. A third function sets the maximum number of knots allowed in the table. The default value is 15000 knots.
There are two possible ways to set up a table. The function *SetTable* initializes the table and refines it until the specified tolerance or the maximal number of knots is reached. A different approach is the refinement in single

steps using the *Initialize* and the *RefineLevel* functions. *Initialize* computes the values $f(p)$ at the vertices p of the domain and does one refinement step. Since there are no values in the table during the computation of the values at vertices, no initial values y_0 are given and $y_0 = 0$ is used instead.

In addition to the table modules, template classes for matrix and vectors structures are included in the package. They are implemented in a FORTRAN compatible format such that a large amount of linear algebra subroutines from LAPACK and BLAS can be accessed very easily via a special interface. See the code for further details.

4 Interface

The usage of the *Table* class from FORTRAN or C is very simple. A static *TableManager* holds a vector of pointers to the tables. Therefore each table can be accessed by an integer that is nothing else but the index of that table. Serveral functions can be called to perform actions on the desired table. The most important functions are:

void crtab_(const char dir, int* nofp, int* n, int* num, int* err);*
void rmtab_(const int num, int* err);*

void rdtab_(const int num, int* err);*
void wrtab_(const int num, int* err);*

void settab_(const int num, IFunctionPtr, double* rpar, int* ipar, const double* tol, const double* scal, const int* maxknots, int* err);*
void initab_(const int num, IFunctionPtr, double* rpar, int* ipar, int* err);*
void reftab_(const int num, IFunctionPtr, double* rpar, int* ipar, const double* tol, const double* scal, int* level, int* maxknots, int* err);*

The terrible naming of these functions is a result of the FORTRAN 77 standard that names must not be longer than six letters. All functions end with

tab to indicate that a table handling function is used. The first two or three letters indicate very cryptically which action is performed, so *crtab* means create table, *rmtab* means remove table, *rdtab* means read, *wrtab* means write and so on. The underscores are due to the usual FORTRAN linkage convention. Simply leave them away in the FORTRAN call and write *call crtab(...)* for instance. In C however, the underscores must be written, *crtab_(...)*. See the files example1.cc and example3.f for examples.

The function *crtab* returns the table number *num* that is used in all other functions to identify the table. It should not be necessary to use the remove function, except if you want to free memory space without aborting the program. The functions *settab*, *initab* and *reftab* are used for the setup, the initialization and the refinement as described in the preceding section. A detailed description of the parameters and the calling is given in the file *Interface.cc*. To obtain information from the table the following functions can be used:

```
void slntab_(const int* num, int* err);
void prptab_(const int* num, const int* d, double* p,
             int* isin, double* pp, int* err);
void valtab_(const int* num, const int* n, double* z, int* err);
void jactab_(const int* num, const int* d, const int* n, double* jac,
             int* err);
void hestab_(const int* num, const int* d, const int* n, double* hes,
             int* err);

void noktab_(const int* num, int* knots, int* err);
void noptab_(const int* num, int* nofp, int* err);
void novtab_(const int* num, int* nofval, int* err);
void noltab_(const int* num, int* maxlevel, int* err);
void errtab_(const int* num, double* errest, int* err);
```

The function *slntab* is used to fix d-linear interpolation instead of d-quadratic interpolation used by default. *prptab* is the prepare function as described in Sec. 4 and *valtab*, *jactab* and *hestab* are used to obtain the function values, the Jacobian and the Hessian matrices. The functions *no** return the number

of knots, parameters, data components or levels in the table and *errtab* gives an estimate of the relative error of the table with respect to the scaling chosen in the setup.

Acknowledgement

This work was supported by the DFG within the Schwerpunkt "Ergodentheorie, Analysis und effiziente Simulation dynamischer Systeme".

References

[1] H. Bungartz: *Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung.* PhD-thesis, Institut für Informatik, TU München (1992)

[2] H. Bungartz: *Higher Order Finite Elements on Sparse Grids.* in: A. V. llin and L. R. Scott (Eds.), Houston Journal of Mathematics: Proceedings of the 3rd Int. Conf. on Spectral and High Order Methods, 5.-9.6.1995, Houston pp. 159-170 (1995)

[3] C. de Boor: *A Practical Guide to Splines.* Springer-Verlag, Applied Mathematical Sciences 27, (1978)

[4] P.Deuflhard, B. Fiedler, P.Kunkel: *Efficient Numerical Pathfollowing Beyond Critical Points.* SIAM J. Numer. Anal. 24, pp. 912-927 (1987)

[5] P. Deuflhard, J. Heroth: *Dynamic Dimension Reduction in ODE Models.* in: F.Keil, W.Mackens, H.Voß, J.Werther (eds), Scientific Computing in Chemical Engeneering, Springer-Verlag, pp. 29-43 (1996)

[6] P. Deuflhard, J. Heroth, U. Maas: *Towards Dynamic Dimension Reduction in Reactive Flow Problems.* Konrad-Zuse-Zentrum Berlin, Preprint SC 96-27 (1996). In: Proc. 3rd Workshop on Modelling of Chemical Reaction Systems (CD-Version), Heidelberg (1996)

[7] U. Maas: *Automatische Reduktion von Reaktionsmechanismen zur Simulation reaktiver Strömungen.* Institut für Technische Verbrennung der Universität Stuttgart, Habilitation thesis, (1993)

[8] U. Maas, S. B. Pope: *Simplifying Chemical Kinetics: Intrinsic Low-Dimensional Manifolds in Composition Space.* Combustion and Flame, vol 88, pp. 239-264, (1992)

[9] U. Maas, S. B. Pope. 24th Symposium (International) on Combustion. The Combustion Institute, Pittsburgh (1992)

[10] H. Niemann, D. Schmidt, U. Maas: *An Efficient Storage Scheme for Reduced Chemical Kinetics Based on Orthogonal Polynomials.* Preprint SC 96-XX, Konrad-Zuse-Zentrum für Informationstechnik Berlin, (May 1996)

[11] W. C. Rheinboldt: *MANPACK: A Set of Algorithms for Computations on Implicititly Defined Manifolds.* Technical Report ICMA-96-198, Departement of Mathematics and Statistics University of Pittsburgh (1996)

[12] Ch. Zenger: *Sparse Grids.* in: Parallel Algorithms for Partial Differential Equations, Proceedings of the Sixth GAMM-Seminar, Kiel, January 1990. W.Hackbusch (ed.), Vieweg-Verlag, Braunschweig (1991)