Lehrstuhl für Informatik 10
Friedrich-Alexander-
Universität
Erlangen-Nürnberg

Master Thesis

Numerical Integration of Local Stiffness Matrices in 3D on Sparse Grids on Curvilinear Bounded Domains

Parikshit Upadhyaya

27th September, 2016

Examiner: Prof. Dr. Christoph Pflaum

Eidesstattliche Erklärung / Statutory Declaration

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbständig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

I hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Der Friedrich-Alexander-Universität, vertreten durch den Lehrstuhl für Informatik 10, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

Erlangen, 27th September, 2016

Parikshit Upadhyaya

Abstract

In this thesis, sparse grids and their application to numerical integration is studied in details. In the first chapter, we introduce the motivation behind the use of sparse grids. In the second chapter, both adaptive and non-adaptive sparse grids are considered, and the convergence of integrals of some functions on such grids is illustrated. In the third chapter, the concept of sparse grid integration is then applied towards constructing local stiffness matrices for solving PDEs on curvilinear bounded domains in 3D. Towards this end, some other concepts like transfinite mapping from rectangular to curvilinear domains is also studied.

CONTENTS

1

INTRODUCTION TO SPARSE GRIDS

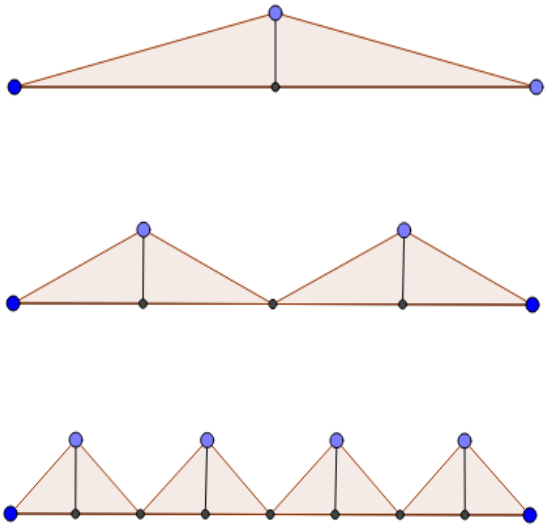
Numerical discretization and approximation of multivariate functions is a very important problem in the field of applied mathematics. It is the first step towards solving any problem that requires numerical quadrature or a numerical solution of PDEs. Hence, it lends itself to many applications in a variety of fields. Almost all widely used discretization teqchniques rely on representing the function as a linear combination of simple basis functions. What makes the techniques different from each other is the choice of basis functions and the distribution of points in the domain on which the discretization is done. Each point in this set of points has an associated basis function.

If N is the number of degrees of freedom along one dimension, and d is the number of dimensions, then a traditional full grid discretization requires $O(N^d)$ points and basis functions, to give an accuracy of $O(N^{-2})$. This is indeed incredibly expensive when the problem involves higher dimensions, and hence aptly called *"The curse of dimensionality"*. Hence, there is a need for a discretization technique that would possibly find a way around this curse without sacrificing a lot in accuracy. This is precisely the motivation behind the use of sparse grids.

In the sections that follow, we will see how the structure of hierarchy and careful omission of certain basis functions helps us to achieve this.

1.1 Hierarchichal basis representation

At the heart of sparse grids lies the concept of the standard hierarchical basis representation, for example the one used in [1]. A hierarchical basis respresentation, as the name suggests, can be divided into different subspaces of basis functions, each subspace corresponding to a different "level". All functions belonging to a specific level have the same support size. For an illustration of the nature of these subspaces built using hat functions on the domain $\Omega = [0, 1]$, let us have a look at the following figure:



As we can see, the n-th subspace T_n consists of 2^{n-1} basis functions, such that,

$$T_n = \{\phi_{n,i} | \phi_{n,i}(x) = W_n(x - x_{n,i}), x_{n,i} = 2^{-n}i, i \in I_n\} \tag{1.1}$$

where

$$I_n = \{i | i = 1, 3, 5, ..., 2^n - 1\} \tag{1.2}$$

and the hat function $W_n(x)$ is given by:

$$W_n(x) = \begin{cases} 1 + 2^n x & -2^{-n} \leq x < 0 \\ 1 - 2^n x & 0 \leq x \leq 2^{-n} \\ 0 & otherwise \end{cases} \tag{1.3}$$

Now, let us define

$$S_n = \bigcup_{i=1}^{n} T_i \tag{1.4}$$

then S_n is the $2^n - 1$ dimensional space of all piecewise linear functions f on $[0, 1]$ with meshwidth 2^{-n}, such that $f(0) = f(1) = 0$.

Hence,

$$f(x) = \sum_{i=1}^{n} f_i(x) \tag{1.5}$$

where each f_i is a function that represents the contribution of the subspace T_i to f, which can be further decomposed as a linear combination of the basis functions in T_i, that is,

$$f_i(x) = \sum\nolimits_{k \in I_i} c_k f_k(x) \tag{1.6}$$

Since our goal is to construct an interpolant for arbitrary functions, let us consider that $f(x)$ is not a piecewise linear function and an arbitrary function instead. Then 1.5 and 1.6 represent the approximation of f in S_n, which we call f_I. Thus, we can write:

$$f_I(x) = \sum_{l=1}^{n} \sum\nolimits_{i \in I_l} \alpha_{l,i} \phi_{l,i} \tag{1.7}$$

where $\alpha_{l,i}$ is the coefficient of $\phi_{l,i}$, where $\phi_{l,i} \in T_l$ and is centered around $x_{l,i}$ as defined in 1.1. As we can see, any basis function in S_n can be uniquely identified by l and i, which denote the level of refinement and the location respectively. Then, we have,

$$\alpha_{l,i} = f(x_{l,i}) - \frac{f(x_{l,i} - h_l) + f(x_{l,i} + h_l)}{2} = -h_l \int_{x_{l,i}-h_l}^{x_{l,i}-h_l} \phi_{l,i}(x) \frac{\partial^2 f(x)}{\partial x^2} dx \tag{1.8}$$

where $h_l = 2^{-l}$

$\alpha_{l,i}$ is often called the *hierarchical coefficient* or the *hierarchical surplus*. We can now generalize this interpolant for a multi-dimensional problem using a tensor product. For this, we must introduce the multi-index set notation . Let us consider we have a d dimensional problem.

$$\underline{l} = \{l_1, l_2, ..., l_d\} \tag{1.9}$$

Then,

$$\phi_{\underline{l},\underline{j}}(\overline{x}) = \prod_{i=1}^{d} \phi_{l_i,j_i}(\overline{x}) \tag{1.10}$$

is d-dimensional basis function where $\underline{l}$ is the multi-index that denotes the level of refinement along each dimension and $\underline{j}$ represents the location along each dimension.

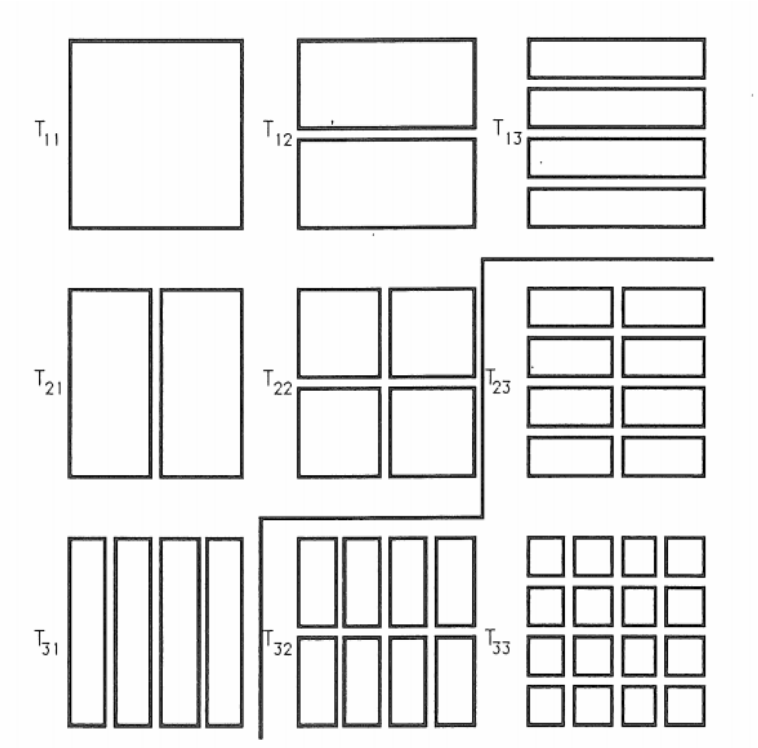


Figure 1.1: Subspaces of hierarchical basis functions in 2D, taken from [1]

We can now define d-dimensional hierarchical subspaces as follows:

$$T_{\underline{l}} = span\{\phi_{\underline{l},\underline{j}} | \underline{j} \in I_{\underline{l}}\} \tag{1.11}$$

The following figure illustrates how these subspaces look for a 2-dimensional problem:

We can now define the full grid $(2^n - 1)^d$ dimensional space $S_{\underline{n}}$ for a d-dimensional problem:

$$S_{\overline{n}} = \bigcup_{l_1=1}^{l_1=n} \bigcup_{l_2=1}^{l_2=n} ..... \bigcup_{l_d=1}^{l_d=n} T_{\underline{l}} \tag{1.12}$$

Now if f is an arbitrary function on Ω^d, then the interpolant f_I on $S_{\underline{n}}$ can be written as:

$$f_I(\overline{x}) = \sum_{\|\underline{l}\|_\infty \leq n} \sum_{i \in I_{\underline{l}}} \alpha_{\underline{l},\underline{i}} \phi_{\underline{l},\underline{i}}(\overline{x}) \tag{1.13}$$

Here, we are using $(2^n - 1)^d$ basis functions for constructing the interpolant. In the next section we will see how we drastically reduce the number of basis functions to get a sparse grid instead of a full grid.

1.2 Sparse Grids

As discussed before, we will get a sparse grid interpolant from the full grid interpolant of 1.13 by discarding some basis functions. It would definitely be a good idea to discard those subspaces which are high dimensional, but don't contribute much to the overall approximation. We can show this for a simple 2-dimensional case first. We have the following upper bound for the contribution of the subspace $T_{i,j}$ to the interpolant of f:

$$\|f_{i,j}\|_\infty \leq 4^{-i-j-1}|f| \tag{1.14}$$

where

$$|f| = \|\frac{\partial^4 f}{\partial x^2 \partial y^2}\|_\infty \tag{1.15}$$

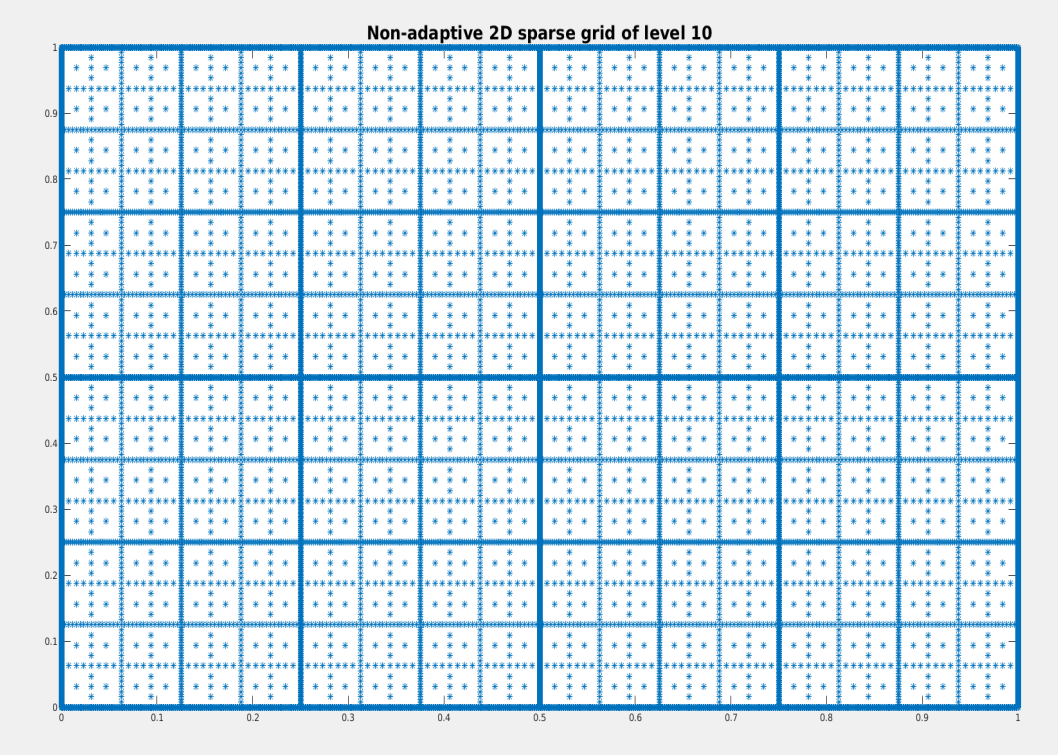
The ńumber of basis functions in $T_{i,j}$ is 2^{i+j-2} and as we can see from 1.14, the contribution to the error decays exponentially when $i+j$ becomes big. This means, for example, including the subspace $T_{1,3}$ is preferable to the subspace $T_{3,3}$. Keeping this in mind, we can have a triangular scheme of subspaces instead of a rectangular scheme and construct a different space of functions $\overline{S}_{n,n}$, which is our sparse grid space:

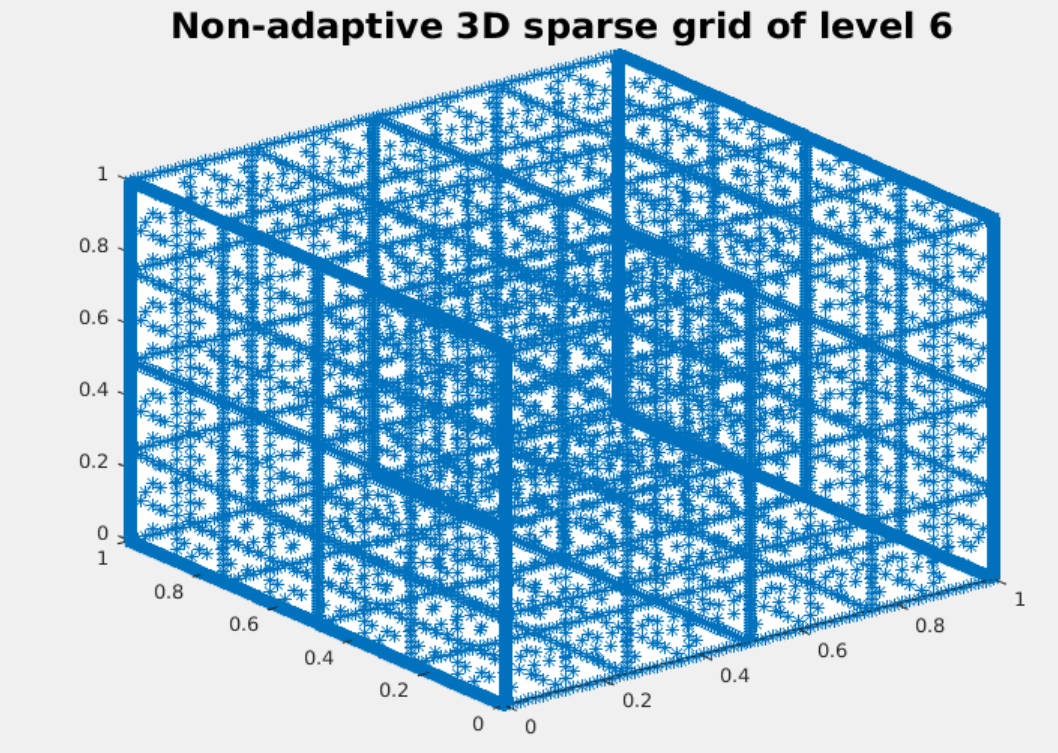
$$\overline{S}_{n,n} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{n-i+1} T_{i,j} \tag{1.16}$$

The number of basis functions involved in this union is:

$$\sum_{i=1}^{n} \sum_{j=1}^{n-i+1} 2^{i+j-2} = (n-1)2^n + 1 = O(h_n^{-1} \log h_n^{-1}) \tag{1.17}$$

instead of being $(2^n - 1)^2 = O(h_n^{-2})$, which is a drastic reduction.


Non-adaptive 2D sparse grid of level 10
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Non-adaptive 3D sparse grid of level 6
0
0.2
0.4
0.6
0.8
1

For the full grid interpolant, we have, as shown by Zenger in [1]

$$\begin{aligned}
\|f - f_I\|_\infty &\leq \|\sum_{i=1}^{\infty}\sum_{j=1}^{\infty} f_{i,j} - \sum_{i=1}^{n}\sum_{j=1}^{n} f_{i,j}\|_\infty \\
&\leq \sum_{i=1}^{n}\sum_{j=n+1}^{\infty} \|f_{i,j}\| + \sum_{i=n+1}^{\infty}\sum_{j=1}^{\infty}\|f_{i,j}\| \\
&\leq 2|f|\sum_{i=1}^{\infty}\sum_{j=n+1}^{\infty} 4^{-i-j-1} \\
&= 2|f|\sum_{i=1}^{\infty}4^{-i}\sum_{j=n+1}^{\infty} 4^{-j-1} \\
&\leq \frac{2}{9}4^{-n-1}|f| = \frac{1}{18}h_n^2|f|
\end{aligned}$$

Hence, we have second order convergence as is expected for a traditional full grid interpolation. For the sparse grid interpolant $\overline{f}_I$, we have,

$$\begin{aligned}
\|f - \overline{f}_I\|_\infty &\leq \|\sum_{i=1}^{\infty}\sum_{j=1}^{\infty} f_{i,j} - \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} f_{i,j}\|_\infty \\
&= \|\sum_{i=1}^{n}\sum_{j=n-i+2}^{\infty} f_{i,j} + \sum_{i=n+1}^{\infty}\sum_{j=1}^{\infty} f_{i,j}\|_\infty \\
&\leq (\sum_{i=1}^{n}\sum_{j=n-i+2}^{\infty} 4^{-i-j-1} + \sum_{i=n+1}^{\infty}\sum_{j=1}^{\infty}4^{-i-j-1})|f| \\
&= (\frac{n}{3}4^{-n-2} + \frac{4}{9}4^{-n-2})|f| \\
&= (\frac{1}{48}h_n^2(\log h_n^{-1} + \frac{4}{3}))|f|
\end{aligned}$$

Hence, the error is approximation using the sparse grid is $O(h_n^2 \log h_n^{-1})$ which isn't significantly worse than that for full grid interpolation. On the other hand, we have a drastic reduction in the dimension when compared to the full grid space.
For a d-dimensional problem, we can generalize the 2-dimensional construction of 1.16 as follows:

$$\overline{S_d(n)} = \bigcup_{|\underline{l}|_1 \leq n+d-1} T_{\underline{l}} \tag{1.18}$$

Then, we have, as in [8]

$$dim(\overline{S_d(n)}) = O(h_n^{-1}|\log_2 h_n|^{d-1}) = O(2^n n^{d-1}) \tag{1.19}$$

which is significantly better than $O(2^{nd})$ as in the case of the full grid space. For functions having bounded mixed second order derivatives i.e., for functions belonging to the Sobolev space $H_2^{mix}(\Omega^d)$ where,

$$H_2^{mix}(\Omega^d) = \{f : \Omega^d \to \mathbb{R} | D^\alpha f \in L_2(\Omega^d), |\alpha|_\infty \leq 2\} \tag{1.20}$$

we have the following result for the error in the sparse grid interpolant $\overline{f}_n$:

$$\|f - \overline{f}_n\| = O(h_n^2|\log h_n|^{d-1}) \tag{1.21}$$

The hierarchical coefficient $\alpha_{\underline{l},\underline{i}}$ of 1.13 can be obtained by applying a tensor product of the formula in 1.8 as given by the stencil $[-0.5, 1, -0.5]$:

$$\alpha_{\underline{l},\underline{i}} = \prod_{k=1}^{d} ([-0.5, 1, -0.5])_{l_k,i_k} f \tag{1.22}$$

2

NUMERICAL INTEGRATION USING SPARSE GRIDS

In the previous chapter, we looked at how to construct an interpolant for a multivariate function using a sparse grid as opposed to a traditional full grid. In this chapter, we will utilize the interpolant that we constructed in the previous chapter to perform numerical intergation on a non-curvilinear domain. We will first do this using a non-adaptive sparse grid, and later using an adaptive grid, using the hierarchical surplus as the criteria for tolerance. Finally, we will look into how the computed integral converges to the actual value for certain functions.

2.1 Problem formulation

As discussed in chapter 1, let us consider the function $f : \Omega^d \rightarrow \mathbb{R}$, where $\Omega = [0, 1]$. For the purpose of this thesis, we will consider problems only up to 3D. Our goal is to compute the integral of f, $I(f)$ over the domain Ω^d using the sparse grid interpolant $\overline{f}_n$.

Using the formula from 1.13 for the sparse grid interpolant, we have,

$$\overline{f}_n = \sum_{\|\underline{l}\|_1 \leq n+d-1} \sum_{i \in I_{\underline{l}}} \alpha_{\underline{l},\underline{i}} \phi_{\underline{l},\underline{i}}(x) \tag{2.1}$$

$$\implies \int_{\Omega^d} \overline{f}_n dx = \sum_{\|\underline{l}\|_1 \leq n+d-1} \sum_{i \in I_{\underline{l}}} \alpha_{\underline{l},\underline{i}} \int_{\Omega^d} \phi_{\underline{l},\underline{i}}(x) dx \tag{2.2}$$

We already have $\alpha_{\underline{l},\underline{i}}$ from 1.22. Hence, all we need to compute the integral is to compute $\int_{\Omega^d} \phi_{\underline{l},\underline{i}}(x)dx$, which is pretty easy since we are dealing with basis functions which are tensor products of hat functions. For example, in $1D$, it is the area under the triangle formed by the hat function. In $2D$, it will be the volume under the tetrahedron formed by the basis function. For a d-dimensional problem,

$$\int_{\Omega^d} \phi_{\underline{l},\underline{i}}(x)dx = \prod_{k=1}^{d} \frac{h_{l_k}}{2} \tag{2.3}$$

where

$$h_{l_k} = 2^{1-l_k} \tag{2.4}$$

The following code-snippet illustrates how we iterate through all the necessary subspaces and add their contribution to the final integration:

```
double integrate (double(* funcToIntegrate )(D3vector&), D3vector &limit1, D3vector
    &limit2, int max_level){
    double integral = 0.0;

    // Initialize limits of integration
    this->limit1 = limit1 ;
    this->limit2 = limit2 ;

    //check the dimension of the problem
    bool dim_2 = ( this->dim >= 2);
    bool dim_3 = ( this->dim >= 3);
    int max_y,max_z;

    int count = 0;
    for( int i=0;i<=max_level;i++){
        if (dim_2)
            max_y = max_level-i+this->dim-1;
        else
            max_y = 0;
        for( int j=0;j<=max_y;j++){
            if (dim_3)
                max_z = max_level-i-j+this->dim-1;
            else
```

```
                max_z = 0;
            for( int k=0;k<=max_z;k++){
                // Increment the integral using basis functions of the subspace
                    W(i,j,k)
                integral += subspaceIntegral( funcToIntegrate, i, j, k, count);
            }
        }
    }

    return count;
}
```

2.2 Dealing with non-zero boundaries

Till now, we have only dealt with functions that evaluate to 0 at the boundary. If we are to compute integrals for functions that don't behave so, we need to change our scheme of basis functions. As shown in [4], there are 2 approaches towards doing this. Firstly, we can just add extra basis functions at the boundary corresponding to a new 0-th level as shown below:

In the second approach, instead of adding extra basis functions, we can modify the existing basis functions at each level that are adjacent to the boundary, by folding them up towards the boundary. This results in the basis function of the 1-st level becoming a constant function. To better illustrate this, let us have a look at the following figure.

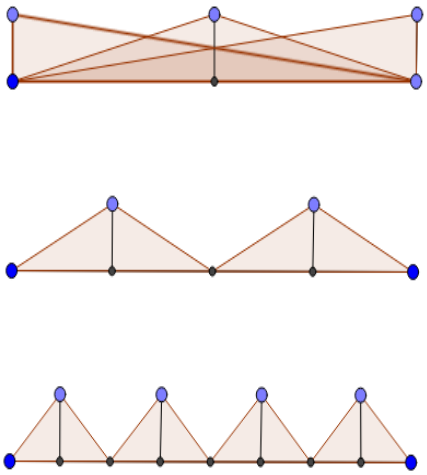
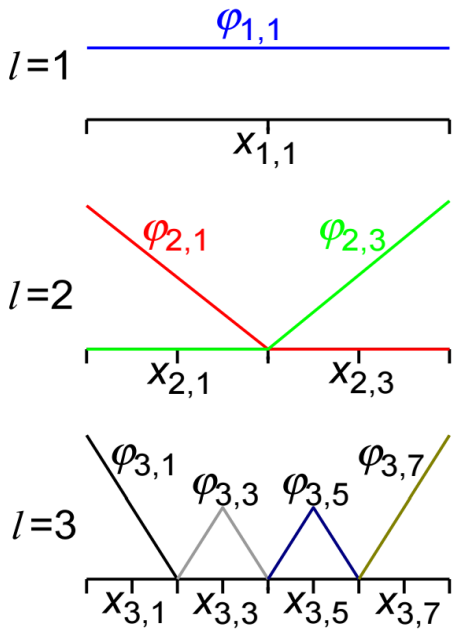


Figure 2.1: Scheme of basis functions for non-zero boundaries as in [4]



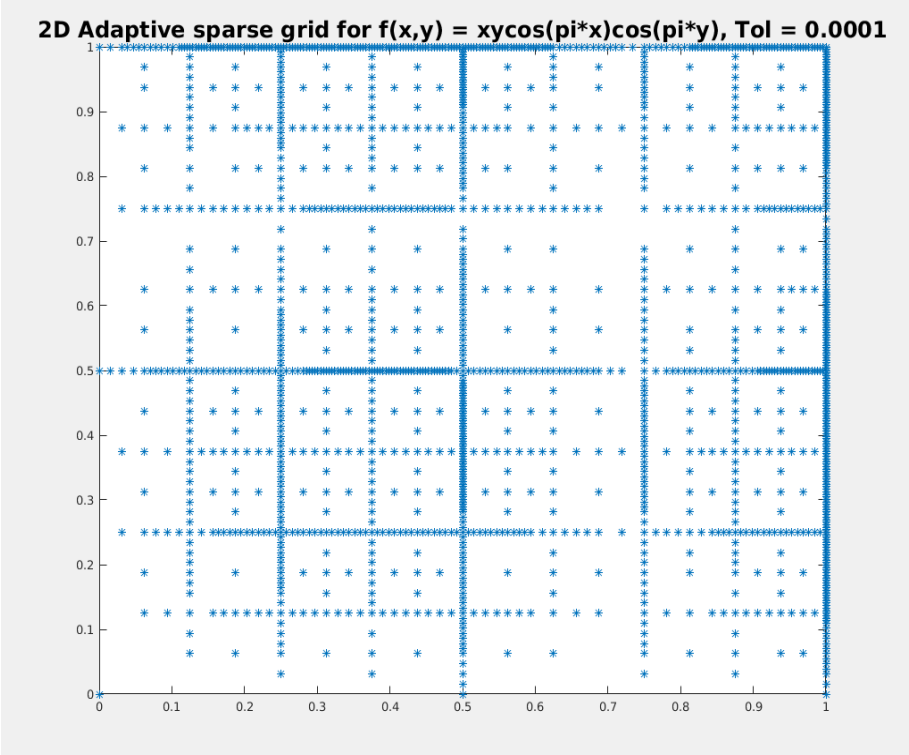
The second approach has an obvious advantage over the first because of the fact that we don't add any extra grid points at the boundary, and hence the number of basis functions used is significantly less.

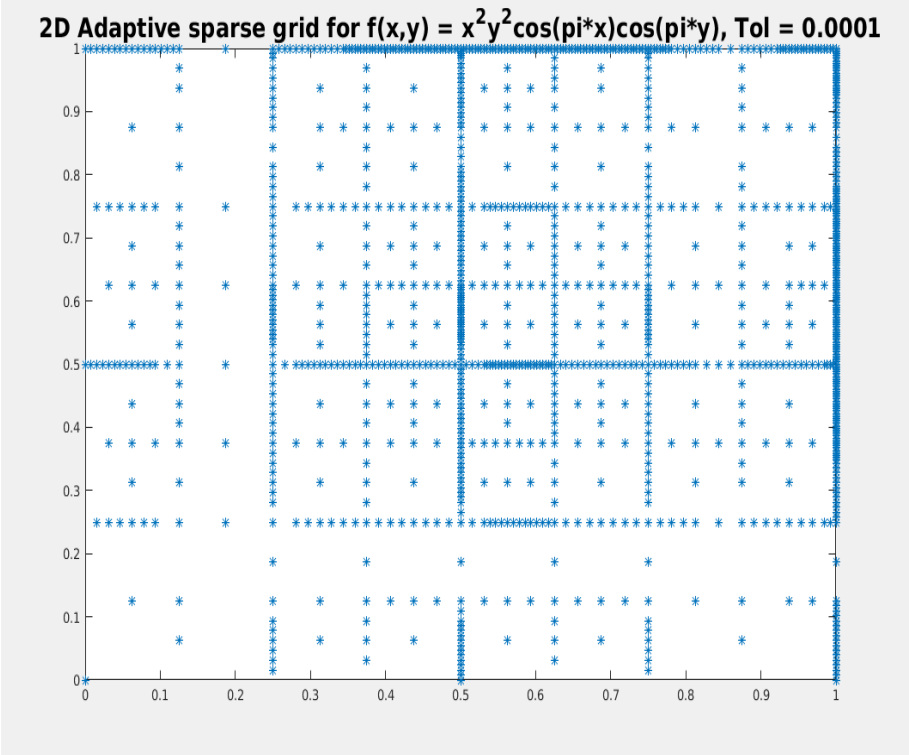
2.3 Adaptivity

Instead of using the triangular scheme of subspaces as discussed until now, we can choose to discard some basis functions based on the magnitude of the hierarchical surplus. We set a specific tolerance for the value of the hierarchical surplus and then discard all those basis functions for which the magnitude of the hierarchical surplus is below the tolerance. Depending on the behaviour of the function we are trying to integrate, some of the subspaces included using the triangular scheme may have very little contribution to the final integration, and hence need not be added. This will result in the formation of a sparse grid that looks different for different functions, and this adaptive interpolant can be formulated as shown below:

$$\overline{f}_n = \sum_{\underbrace{|\underline{l}|_\infty \leq n}} \sum_{\underbrace{i \in I_{\underline{l}}, |\alpha_{\underline{l},\underline{i}}| \geq \epsilon}} \alpha_{\underline{l},\underline{i}} \phi_{\underline{l},\underline{i}}(x) \tag{2.5}$$

The following figures show how an adaptive 2D sparse grid will look like for two different functions:





The following code snippet shoes our implemention of adaptive quadrature:

```
while(! refine_queue.empty()){
        current = refine_queue.front();
        // If the point hasn't been refined yet, find the local integral
        if(std::find(refined_set.begin(), refined_set.end(), current) ==
             refined_set.end()){
            surplus = findHierarchicalSurplus(current, funcToIntegrate);

            integral += surplus*current.volume();

            // Pop the current point & push it to the refined set
            refine_queue.pop(); refined_set.push_back(current);

            // Refine if hierarchical surplus is greater than tolerance
            if(fabs(surplus) >= this->tol || current.getLevels() == levels_0){
                refine(current);
            }
        }

        // if it has been refined already, just pop
        else{
            refine_queue.pop();
        }
    }
```

As we can see, we are maintaining a queue of grid points known as $refine\_queue$ which contains those grid points which are candidates for refinement. By refinement, we mean adding all the $2*dim$ neighbouring points which have one extra level of refinement along one of the dimensions. All grid points in the $refine\_queue$ are then checked to see if they are contained in $refined\_set$, which is the set of grid points that have been refined already. If not, then their contribution to the integration is added and they are refined further. It is important to maintain the $refined\_set$ to avoid the possibility of including grid points twice.

2.4 Results for convergence

Consider $I = \int_0^1 \int_0^1 sin(\pi x) sin(\pi y) = 0.40528473456$ and a non-adaptive sparse grid.

N(Level of refinement for sparse grid integration)	Error($\lvert I(f_N) - 0.40528473456 \rvert$)	Ratio of errors
1	0.15528473456000003	——
2	0.051731343966726295	0.3331
3	0.016143993340905882	0.3121
4	0.0048387681128084781	0.2997
5	0.00141044343263802	0.2915
6	0.00040280612375637181	0.2856
7	0.00011325103897630306	0.2812
8	3.145018962136259e-05	0.2777
9	8.6469041583048245e-06	0.2749
10	2.3578103086063784e-06	0.2727
11	6.3846840908654556e-07	0.2708
12	1.7186580186789868e-07	0.2692

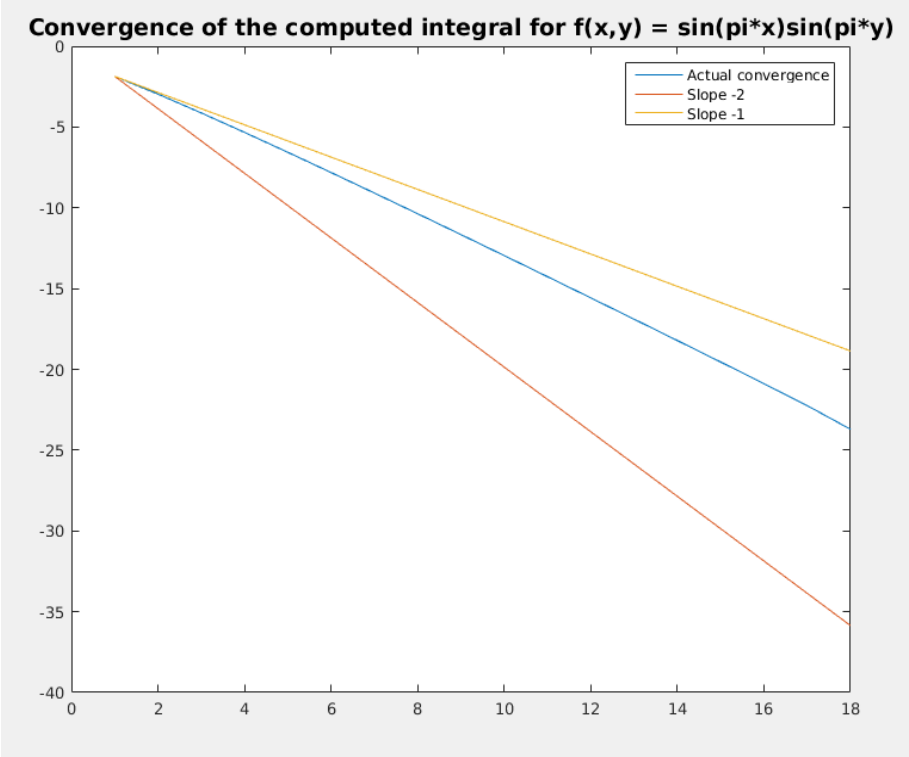


Figure 2.2: x-axis: Level of refinement, **y-axis**: Natural logarithm of the error

Consider $I = \int_0^1 \int_0^1 \int_0^1 sin(\pi x)sin(\pi y)sin(\pi z) = 0.25801227546$

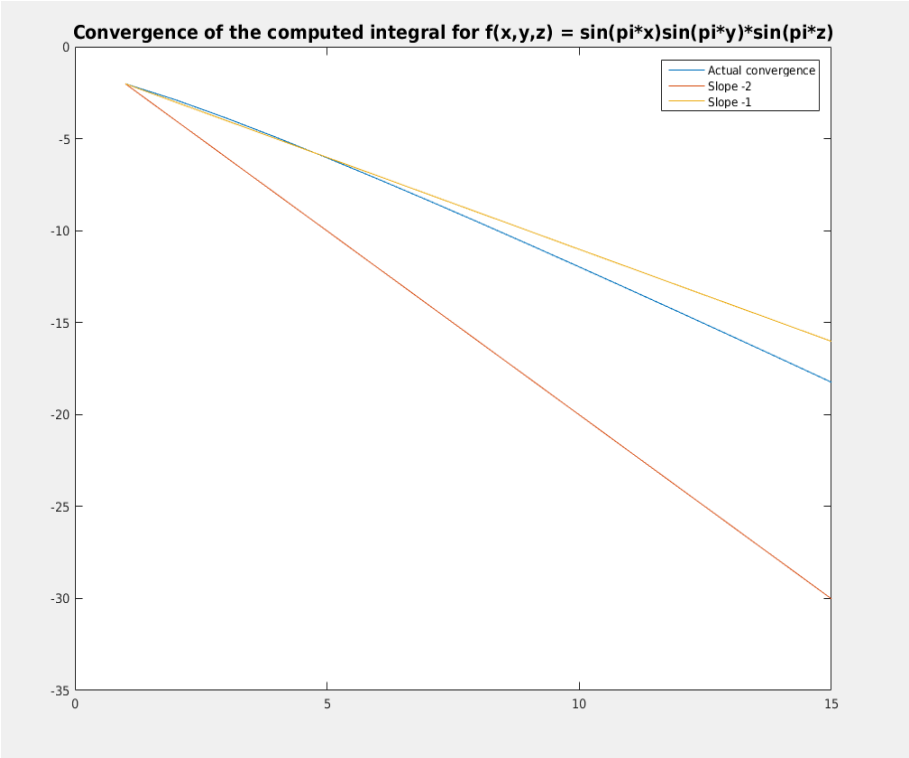


Figure 2.3: x-axis: Level of refinement, **y-axis**: Natural logarithm of the error

As we can see, in both of the functions studied above, we have somewhere between first and second order convergence. In the case of the second function, the convergence is worse because the problem is in 3D. As we increase the number of dimensions in the problem, the logarithmic factor in $O(h_n^2|\log h_n|^{d-1}$ becomes significant and hence the convergence worsens. Now, we will look at the behaviour of convergence in the case of adaptive grids.

Consider $I = \int_{[0,1]^2} xycos(\pi x)cos(\pi y)dxdy = 0.04106392901$

$-\log_{10} tol$(Tolerance for h.surplus)	Error($\lvert I(f_N) - 0.40528473456 \rvert$)
1	0.0071465818613635976
2	0.00016425772356082335
3	4.3290444147844886e-05
4	6.8042563001954881e-06
5	8.1112624634310793e-07
6	6.3675617646696825e-08

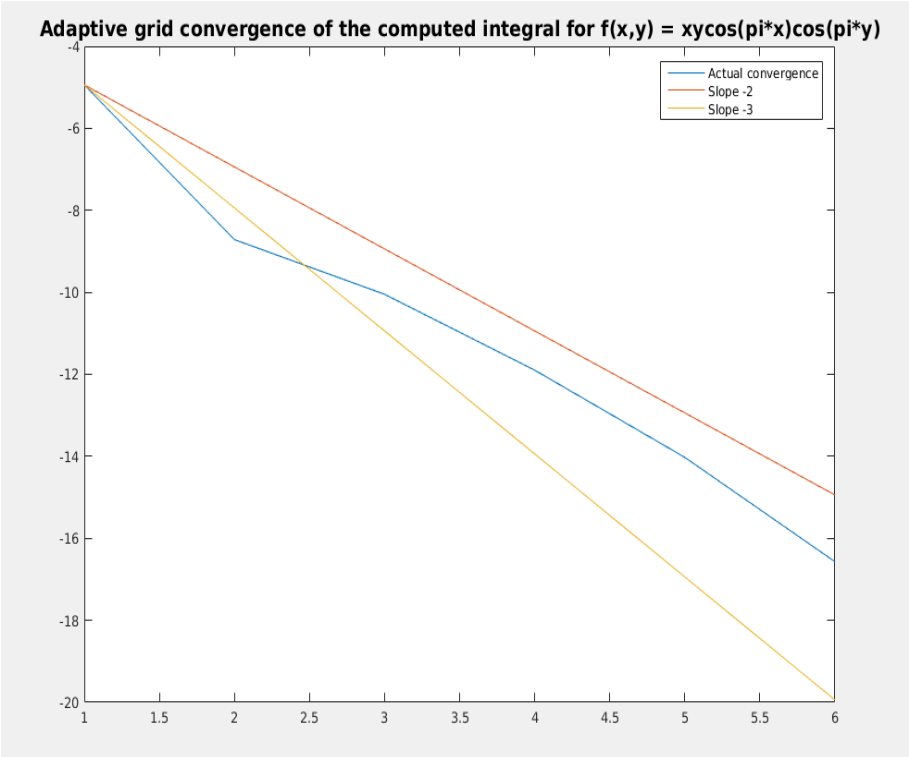


Figure 2.4: x-axis: -$\log_{10} tol$, **y-axis** : $log_e Error$

As of now, there is no result available in the literature that relates the tolerance for the magnitude of the hierarchical surplus to the order of convergence. However, if we observe the slopes in the graph, we see somewhere between second and third order convergence with respect to the order of tolerance i.e. $-\log_{10} tol$. We observe slightly different behaviour when the problem is in 3D.

Consider $I = \int_{[0,1]^3} xycos(\pi x)cos(\pi y)dxdy = -0.00832129178$

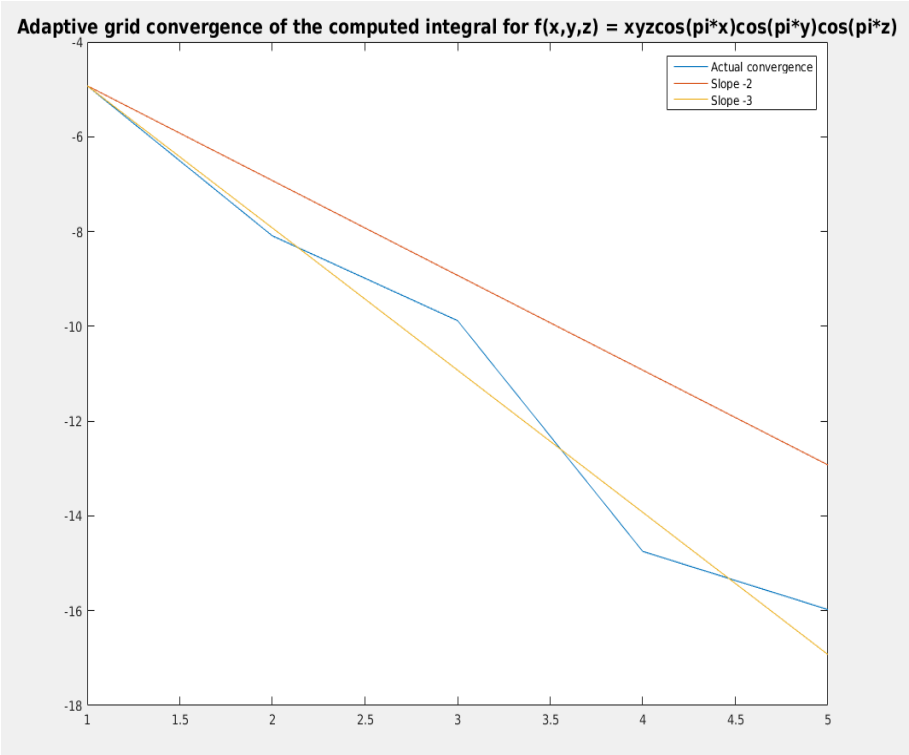


Figure 2.5: x-axis: -$\log_{10} tol$, **y-axis** : $log_e Error$

3

SOLVING PDES ON CURVILINEAR BOUNDED DOMAINS

In the previous chapters, we introduced sparse grids and explained how we can construct a sparse grid interpolant to perform numerical integration. In this chapter, we will see how we put this to use to build local stiffness matrices in an FEM solver for PDEs on curvilinear bounded domains. We will show how we transform the original variational formulation on the curvilinear domain to one on a unit cube. Then, we will use transfinite mapping and numerical integration using sparse grids to build the local stiffness matrices. Finally, we use a BiCGSTAB solver to solve the assembled system.

3.1 Problem formulation

Consider that we have a curvilinear bounded domain $\mathbf{P} \subset \mathbb{R}^3$ and an elliptic PDE on $\mathbf{P}$ with the following variational formulation:

$$a(u,v) = \int_{\mathbf{P}} \nabla u A \nabla v^T + < \nabla u, b > v + cuvd\theta = \int_{\mathbf{P}} fvd\theta, \forall v \in H_0^1(\mathbf{P}) \qquad (3.1)$$

To move ahead, we need a mapping function that maps the unit cube $\mathbf{Q} = [0,1]^3$ to $\mathbf{P}$. Let us assume the existence of such a mapping function $\phi : \mathbf{Q} \rightarrow \mathbf{P}$, $\phi(\overline{x}) = \overline{\theta}, \overline{x} = (x,y)^T, \overline{\theta} = (\theta,\eta)^T$. We will show how we can construct such a function using the concept of transfinite mapping later. The mapping function should be bijective and smooth enough for its Jacobian to exist. This is because we need the Jacobian of ϕ to form the transformed variational formulation.

Applying the chain rule as shown in [2], we end up with the following transformed variational formulation on the unit cube $\mathbf{Q}$:

$$a^*(\widetilde{u},\widetilde{v}) = \int_{\mathbf{Q}} \nabla\widetilde{u}A^*\nabla\widetilde{v}^T + < \nabla\widetilde{u}, b^* > v + c^*\widetilde{u}\widetilde{v}d\overline{x} = \int_{\mathbf{Q}} f^*\widetilde{v}d\overline{x}, \forall\widetilde{v} \in H_0^1(\mathbf{Q}) \quad (3.2)$$

where

$$A^*(\overline{x}) = J^{-1}A(\phi(\overline{x}))J^{-T}|det(J)| \quad (3.3)$$

$$b^*(\overline{x}) = J^{-1}b(\phi(\overline{x}))|det(J)| \quad (3.4)$$

$$c^*(\overline{x}) = c(\phi(\overline{x}))|det(J)| \quad (3.5)$$

$$f^*(\overline{x}) = f(\phi(\overline{x})|det(J)| \quad (3.6)$$

$$J(\overline{x}) = \begin{bmatrix} \frac{\partial\phi_1}{\partial x} & \frac{\partial\phi_1}{\partial y} & \frac{\partial\phi_1}{\partial z} \\ \frac{\partial\phi_2}{\partial x} & \frac{\partial\phi_2}{\partial y} & \frac{\partial\phi_2}{\partial z} \\ \frac{\partial\phi_3}{\partial x} & \frac{\partial\phi_3}{\partial y} & \frac{\partial\phi_3}{\partial z} \end{bmatrix}$$

As we can see, to construct the local stiffness matrices, we need three essential ingredients:

- The mapping function ϕ.
- The Jacobian J.
- A method to perform numerical integration.

We already have the third ingredient from the previous chapters. For the first two ingredients, we need to construct the mapping function, which we do using transfinite mapping. We will first illustrate the concept of transfinite mapping for 2D and then extrapolate to 3D.

3.2 Transfinite mapping

The goal of transfinite mapping as shown in [5] and [2] is to contruct a bijective function ϕ that map a rectangular domain(unit cube in our case) $\mathbf{Q}$ to a curvilinear domain $\mathbf{P}$ to . It is important to note that there need not be a unique function for a specific pair $(\mathbf{P}, \mathbf{Q})$.

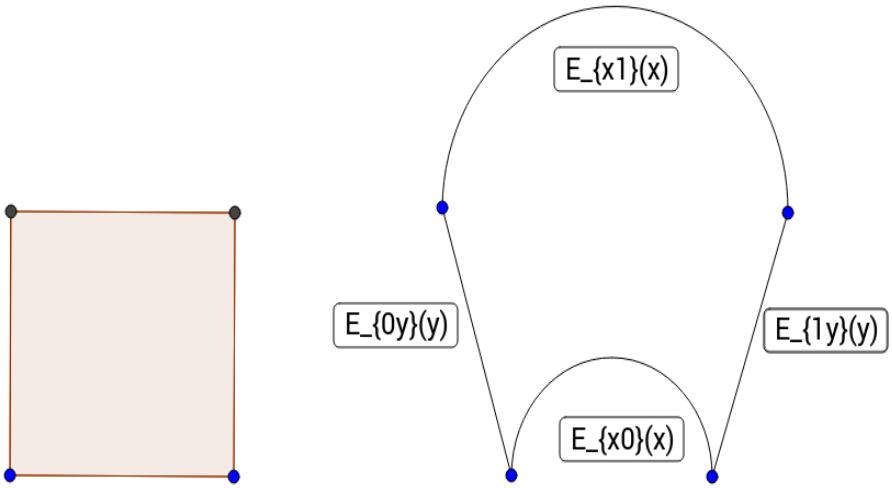


Figure 3.1: Mapping a unit square to a curvilinear domain

As shown in the figure above, let us consider we have a curvilinear domain P in 2D and the unit square $[0, 1]^2$ with the edges labelled according to the edges of the unit square that they correspond to. For example, the edge that corresponds to the edge $x = 0$ of the unit square is labelled as E_{0y}. We also assume that we are given the parametric equations of the edges of the curvilinear domain i.e. each of E_{0y}, E_{1y}, E_{x0}, and E_{x1} map $[0, 1]$ to ∂P, the boundary.

In order for ϕ to be valid, it obviously needs to be valid at the corners. Using this fact, we guess:

$$\phi(x, y) = w_{x0}E_{x0}(x) + w_{x1}E_{x1}(x) + w_{0y}E_{0y}(y) + w_{1y}E_{1y}(y) \tag{3.7}$$

where w_{x0}, w_{x1}, w_{0y} and w_{1y} behave like boolean functions that get activated when $y = 0, y = 1, x = 0$ and $x = 1$ respectively. Our guess is incorrect without the inclusion of additional terms that take into account the validity of ϕ at the corners. The problem with the guess of 3.7 is that at the corners, we will have 2 edges that will get activated and

hence we need to subtract the contribution of one of those edges. Using this, we improve our guess to include terms for the corners and get the correct form of ϕ:

$$\begin{aligned}\phi(x,y) = w_{x0}E_{x0}(x) + w_{x0}E_{x0}(x) + w_{0y}E_{0y}(y) + w_{1y}E_{1y}(y) \\ - [w_{00}C(0,1) + w_{01}C(0,1) + w_{10}C(1,0) + w_{11}C(1,1)]\end{aligned}$$

where w_{00}, w_{01}, w_{10} and w_{11} behave like boolean functions to activate the corners. Our new guess has the correct form but we still need to find what our "boolean"-like functions are. They need to account for validity inside the domain. Hence, in the simplest case,

$$\begin{aligned}w_{x0}(y) &= 1 - y \\ w_{x1}(y) &= y \\ w_{0y}(x) &= 1 - x \\ w_{1y}(x) &= x \\ w_{00}(x,y) &= (1-x)(1-y) \\ w_{01}(x,y) &= (1-x)y \\ w_{10}(x,y) &= x(1-y) \\ w_{11}(x,y) &= xy\end{aligned}$$

We now extrapolate this concept to find the mapping function in 3D. Consider $\mathbf{P}$ to be a curvilinear bounded domain in 3D and $\mathbf{Q}$ to be the unit cube $[0,1]$. In 3D, we have 12 terms to account for interpolation along the edges and 8 terms for interpolation on the corners. In addition, we now need to subtract the contribution from the corner terms twice instead of once. This is because 3 edges intersect at the corners and hence, 3 edge terms get activated.

Taking into account all of the above, we have the mapping function in 3D as follows:

$$\begin{aligned}\phi(x,y,z) = &(1-x)(1-y)E_{00z}(z) + (1-x)yE_{01z}(z) + x(1-y)E_{10z}(z)\\ &+ xyE_{11z}(z) + (1-y)(1-z)E_{x00}(x) + (1-y)zE_{x01}(x)\\ &+ y(1-z)E_{x10}(x) + yzE_{x11}(x) + (1-x)(1-z)E_{0y0}(y)\\ &+ (1-x)zE_{0y1}(y) + x(1-z)E_{1y0}(z) + xzE_{1y1}(y)\\ \\ &- 2[(1-x)(1-y)(1-z)]C(0,0,0) + (1-x)(1-y)zC(0,0,1)\\ &+ x(1-y)(1-z)C(1,0,0) + (1-x)y(1-z)C(0,1,0)\\ &+ xy(1-z)C(1,1,0) + x(1-y)zC(1,0,1)\\ &+ (1-x)yzC(0,1,1) + xyzC(1,1,1)]\end{aligned}$$

To understand the notation, we can simply extrapolate from the notation for 2D. Hence, E_{00z} maps the line corresponding to $(x = 0, y = 0)$ to the boundary and the corresponding "boolean"-like with which it is multiplied is $w_{00z}(x,y) = (1-x)(1-y)$. The following code snippet shows how we use the convenient notation to implement the mapping function:

```
D3vector IntegratorPoissonCurved::Map(double s, double t, double u) const{
  D3vector ret (0.0,0.0,0.0) ;

  // Interpolation along the edges of the domain
  for(int b1 = 0; b1 <=1; b1 ++){
    for(int b2 = 0; b2 <= 1; b2 ++){
      ret = ret +
        bitInterp(b1,t)*bitInterp(b2,u)*transformEdge[edIndex(0,b1,b2)](s);
      ret = ret +
        bitInterp(b1,s)*bitInterp(b2,u)*transformEdge[edIndex(1,b1,b2)](t);
      ret = ret +
        bitInterp(b1,s)*bitInterp(b2,t)*transformEdge[edIndex(2,b1,b2)](u);
    }
  }

  // Interpolation along the corners of the domain
  for(int b1 = 0; b1 <=1; b1 ++){
    for(int b2 = 0; b2 <= 1; b2 ++){
      for(int b3 = 0; b3 <= 1; b3 ++){
        ret = ret -
          2*bitInterp(b1,s)*bitInterp(b2,t)*bitInterp(b3,u)*corner[coIndex(b1,b2,b3)];
      }
    }
  }
```

```
    return ret;
}
```

In the code above, $bitInterp(bit, x)$ returns either x or $1 - x$ depending on whether bit is 1 or 0. $transformEdge[]$ is an array of function pointers of size 12 that contains pointers to functions for edge transformation. $edIndex()$ returns the index of this array that corresponds to the edge represented by its inputs.

We also need the expressions for partial derivatives of ϕ in order to form the Jacobian. Using the expression for the mapping function, we have,

$$\begin{aligned}\frac{\partial\phi(x,y,z)}{\partial x} &= -(1-y)E_{00z}(z) - yE_{01z}(z) + (1-y)E_{10z}(z) \\ &+ yE_{11z}(z) + (1-y)(1-z)E'_{x00}(x) + (1-y)zE'_{x01}(x) \\ &+ y(1-z)E'_{x10}(x) + yzE'_{x11}(x) - (1-z)E_{0y0}(y) \\ &- zE_{0y1}(y) + (1-z)E_{1y0}(z) + zE_{1y1}(y) \\ \\ &- 2[-(1-y)(1-z)]C(0,0,0) - (1-y)zC(0,0,1) \\ &+ (1-y)(1-z)C(1,0,0) - y(1-z)C(0,1,0) \\ &+ y(1-z)C(1,1,0) + (1-y)zC(1,0,1) \\ &- yzC(0,1,1) + yzC(1,1,1)]\end{aligned}$$

We can similarly have expressions for the other partial derivatives. The implementation of the Jacobian is similar to that of the mapping function:

```
D3vector IntegratorPoissonCurved::MapDeriv(int dim,D3vector& x){

  D3vector ret (0.0,0.0,0.0);

  double s1 = x[dim], s2 = x[(dim+1)%3], s3 = x[(dim+2)%3];
  int ed1 = dim*4, ed2 = ((dim+1)%3)*4, ed3 = ((dim+2)%3)*4;

  //Terms related to edge interpolation
  for(int b1=0; b1<=1; b1++){
    for(int b2=0; b2<=1; b2++){
      ret = ret +
        bitInterp(b1,s2)*bitInterp(b2,s3)*transDerivs[edIndexDeriv(ed1,b1,b2,dim)](s1);

      ret = ret +
        bitSign(b1)*bitInterp(b2,s3)*transformEdge[edIndexDeriv(ed2,b1,b2,dim)](s2);

      ret = ret +
        bitSign(b1)*bitInterp(b2,s2)*transformEdge[edIndexDeriv(ed3,b1,b2,dim)](s3);
```

```
        }
    }

    //Terms related to corner interpolation
    for(int b1 = 0; b1 <=1; b1 ++){
        for(int b2 = 0; b2 <= 1; b2 ++){
            for(int b3 = 0; b3 <= 1; b3 ++){
                ret = ret - 2*interpOrSign(b1,0,dim,x)*interpOrSign(b2,1,dim,x)\
                        *interpOrSign(b3,2,dim,x)*corner[coIndex(b1,b2,b3)];
            }
        }
    }

    return ret;
}
```

Here, $transDerivs[]$ is an array of function pointers that contains pointers to functions that return the derivatives of the edge transformation functions and $edIndexDeriv()$ returns the index along the array that corresponds to the edge represented by its inputs.

3.3 Test Problem and Results

In this section, we will combine all the concepts discussed until now to solve the Poisson problem in 3D with Dirichlet boundary conditions:

$$-\Delta u(\overline{\theta}) = f, \overline{\theta} \in \mathbf{P} \tag{3.8}$$

$$u(\partial P) = 0 \tag{3.9}$$

f is chosen such that $u(x,y,z) = xyz(z-0.5)cos(\frac{\pi}{2}(x^2+y^2))sin(\frac{\pi}{2}(x^2+y^2))$ is the solution to the PDE.

$\mathbf{P}$ is a quarter of a ring in 3D with thickness = 0.5

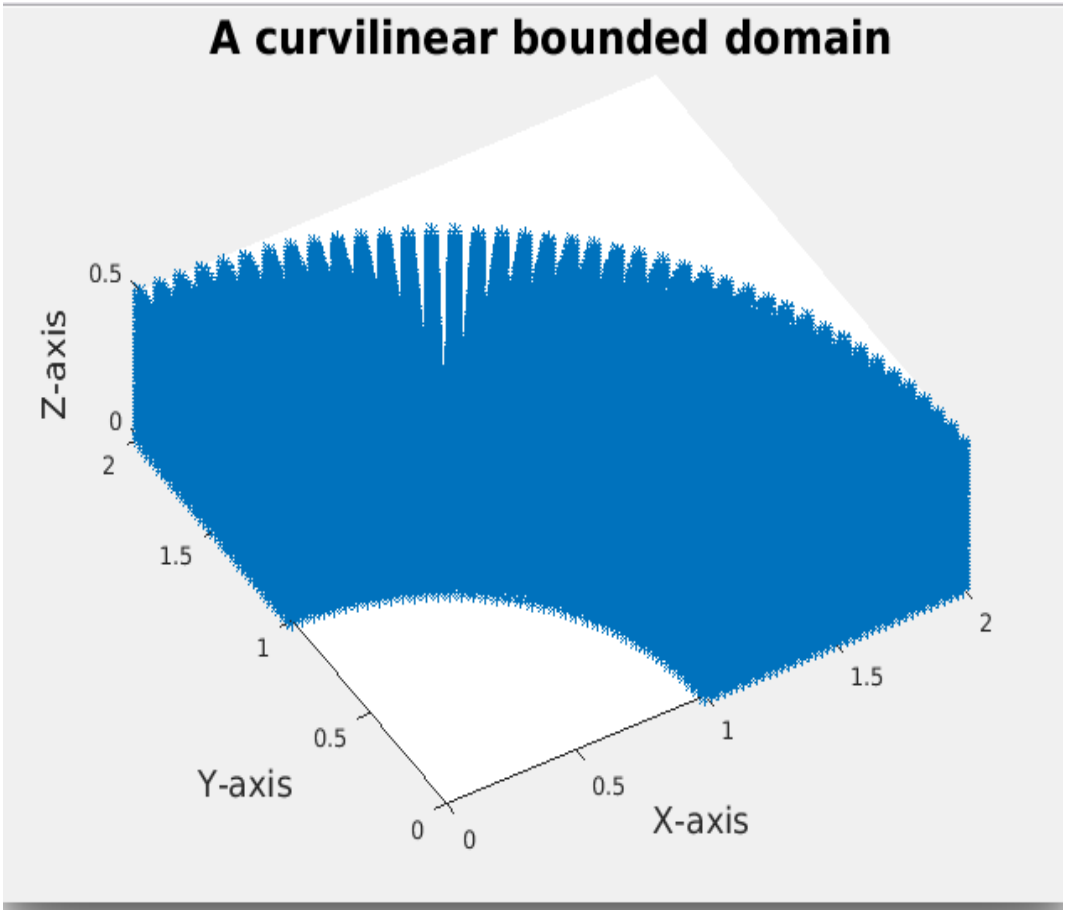


Figure 3.2: Our test domain P

To solve this problem, we use a FEM solver with trilinear elements that uses sparse grid integration for construction of the local stiffness matrices and after assembly, uses a BiCGSTAB solver to solve the resulting linear system. We will now show how the solution converges as we refine our outer grid keeping the sparse grid refinement level as a constant(10). We set our tolerance for the BiCGSTAB solver as 10^{-6} and the maximum number of iterations $= 1000$.

N(Number of elements = N^3)	L^1 norm of error
10	0.00111134
15	0.00050606
20	0.00030031
25	0.00020480
30	0.00015307
40	0.00009833
50	0.00007098
60	0.00005505

We can see from the table above that when we double N, the error gets becomes approximately 0.3333 smaller. This suggests an order of convergence between 1 and 2.

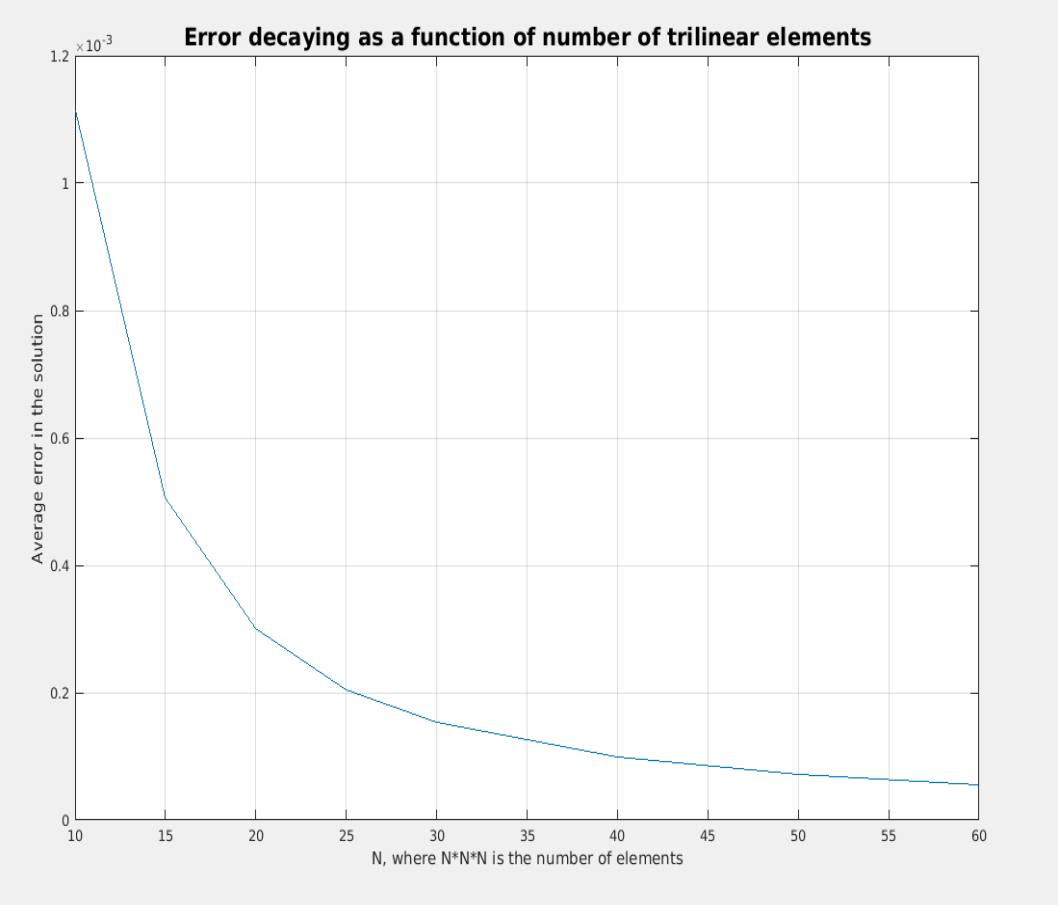


Figure 3.3: X-axis: N, Y-axis: Average Error

Now, we keep the number of trilinear elements as a constant($= 25^3$) and we refine the inner sparse grid to see how the error decays. The parameters for the BiCGSTAB solver are the same as before.

N(Level of refinement for sparse grid integration)	Average Error($\|u - u_N\|$)
2	0.008008194
4	0.005955755
6	0.003903318
8	0.001850884
10	0.00020486
12	0.00225399

The following figure shows how our solution looks like on the unit cube for $z = 0.25$. Note that x and y coordinates vary from 0 to 1, and the z-axis varies from -0.04 to 0.06.

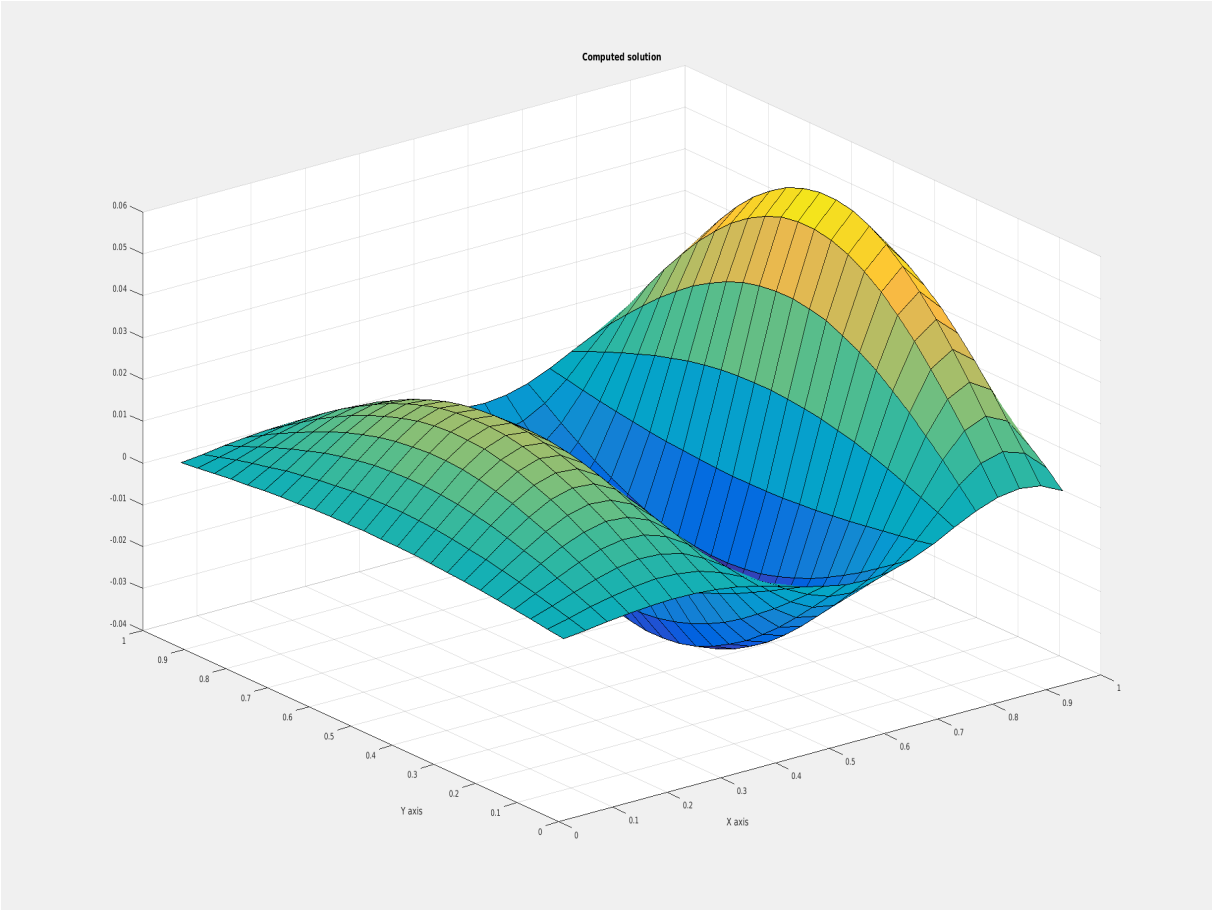


Figure 3.4: Computed solution at $z = 0.5$, 25^3 elements

3.4 Conclusion

In this thesis, we have studied sparse grid integration in detail and applied it towards solving PDEs on curvilinear bounded domains. Sparse grids are inexpensive but can be hard to implement. We can improve a lot upon our current implementation. For example, instead of using a queue for storing grid points to implement adaptivity, we could use better data structures like binary trees and hash maps to ensure fast search of grid points. Also, instead of using regular trilinear elements for the main grid in the FEM solver, we could use a sparse grid discretization, as in [3]. However, this would make the assembly of local stiffness matrices difficult, and would present new challenges like big differences in refinement levels along different dimensions for certain elements, which would further affect the accuracy and convergence of the computed solution.

Bibliography

[1] Christoph Zenger, Sparse Grids,. Parallel Algorithms for Partial Differential Equations, W. Hackbusch, ed., Notes Number Fluid Mech. 31, Braunschweg, Germany, Vieweg, 1991, pp. 241–251.

[2] T.Dornseifer and C.Pflaum, Discretization of Elliptic Differential Equations on Curvilinear Bounded Domains with Sparse Grids . Computing 56, 197-213 (1996)

[3] Rainer Hartmann and Christoph Pflaum, A Sparse Grid Discretization with Variable Coefficients in High Dimensions Numerical Linear Algebra with Applications, 2015;00:1-19

[4] Dirk Pflueger, Spatially Adaptive Sparse Grids for High-Dimensional problems. http://www5.in.tum.de/pub/pflueger10spatially.pdf

[5] W.J.Gordon and L.C.Thiel, Transfinite Mappings and their application to grid generation. Department of Mathematical Sciences, Drexel University

[6] J.D.Jakeman and S.G.Roberts, Local and Dimension Adaptive Sparse Grid Interpolation and Quadrature https://arxiv.org/abs/1110.0010

[7] Thomas Gerstner, Sparse Grid Quadrature Methods for Computational Finance http://wissrech.iam.uni-bonn.de/research/pub/gerstner/gerstner$_h abil.pdf$

[8] Jochen Garcke, Sparse Grids in a nutshell Sparse Grids and Applications, 2012