Multidimensional Fourier Transform on Sparse Grids

Master of Science Thesis in Computational Mathematics

Sveinung Fjær

Department of Mathematics
University of Bergen

June 2, 2009

Forord

Arbeidet med masteren har bydd på mange interessante problemstillinger. Min forståelse av sparse grid (eller glisne grid om du vil) har endret seg gjennom hele prosessen, ny innsikt har stadig kommet gjennom programeringsprossesen og tatt form gjennom skriveprosessen.

Jeg vil takke min veileder Tor. Mang en ide har gått fra abstrakt form, til noe mer konkret etter å ha snakket med ham. Tor har også vært uvurderlig i skriveprosessen, og har stadig funnet ting som kunne forbedres. Christoffer fortjener også min takk, mangt et gjennombrudd har kommet etter å ha forklart hva jeg driver med til ham. En spesiell takk går til Kafé Edvard, som har gitt meg min daglige pause, og nydelig kaffe. Til og med en fysiker fortjener en takk - Sigurd som jeg har delt mangt et måltid med, koselige stunder i badstu og flere mil i basseng, det er i slike stunder masteren egentlig tar form. En siste takk går til Morten, som var elskverdig og leste gjennom min master og fant mangler hist og pist.

Contents

Chapter 1

Introduction

Quantum mechanics is the science governing the smallest building blocks in nature [34]. The Schrödinger equation, time independent (1.1) and time dependent (1.2), models the quantum mechanical systems [20].

$$E\Psi = \widehat{H}\Psi \tag{1.1}$$

$$i\hbar\frac{d\Psi}{dt} = \widehat{H}\Psi \tag{1.2}$$

The Hamilton operator, $\widehat{H}$, describes the system, and consists of the potential and kinetic energy operators. In computational quantum mechanics [2], the Schrödinger equation is solved numerically, as the majority of quantum mechanical systems cannot be solved analytically. But even numerically, we are not able to solve all quantum mechanical systems. Far from it. To model a general one particle system we need three dimensions. For n particles, we need $3n$ dimensions. E.g a helium atom (He) would in general be modeled in nine dimensions, three for each of the electrons, and three for the nucleus. Using the normal tensor product grid, the typical equidistant grid in d dimensions, we need N^d grid points to solve a d-dimensional problem, N being the number of points needed in the one dimensional problem. The number of grid points grows exponentially with dimension. This effect is known as *The curse of dimensionality* [5].

The physicists fight "the curse" by making approximations in their models, which let them reduce the numbers of dimensions [33]. This is not allways feasable, or enough to tame the curse. Another approach is to choose the grid points smartly, using another grid than the typical tensor product grid. The lattice grid is one such grid, using lattice rules to construct a grid, rather than the tensor product [24, 32]. Optimal grids are not found for arbitary dimensions, as finding optimal lattice rules in high dimensions is of a high computational cost [25].

In this thesis we will look at the sparse grid [6]. The sparse grid has numerous applications, e.g. data mining [10], image compression [28] and numerical integration [11]. In this thesis we will consider trigonometric interpolation on sparse grids. Rather than use the general sparse grid theory, which is used e.g. to make adaptive sparse grid [19], we choose to construct a set of definitions which lets us apply the Fourier transform effectively. Included in these definitions we get the classical sparse grid, where we can fight the curse of dimensionality, needing only $\mathcal{O}\left(N\left(\log N\right)^{d-1}\right)$ grid points in d dimensions. It is important to note that the classical sparse grid is suitable for only a certain family of problems. Opposed to the lattice grid, the sparse grid approach easily generalize to any dimension.

The aim of this thesis is to propose an algorithm that computes the Fourier transform effeciently on our grids. An algorithm is proposed in [16], using recursion, built on excisting sparse grid software. The algorithm presented in this thesis is constructed to invite to parallel computing. Testing have been done using a Cray XT4 [1].

Using Strang-splitting [12], we are able to split the Schrödinger equation in two parts. One with the kinetic energy and one with the potential energy. The kinetic energy operator (1.3) is basically the Laplace operator (with some physical constants bundled into a).

$$\widehat{H}_{kinetic} = a\nabla^2 = a\Delta = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2} \tag{1.3}$$

The potential energy operator varies from problem to problem, but is often nonlinear, but seldom contains any spatial derivatives. The computational effeorts will, after splitting, usually come from computing the Laplacian of the function. For this reason, we have chosen to work with the Poisson equation, as it represent the same difficulty, while making it easier to estimate the accuracy of our solution.

The thesis starts with a mathematical preliminaries chapter. Here we will give a short introduction to Hilbert spaces, discretization and spectral methods. Then we go on by defining our notion of sparse grids in Chapter 3. In Chapter 4 we introduce the hierarchical Fourier basis set, which is needed to compute the Fourier transform effectively on the sparse grids. Chapter 5 discuss how to do the Fourier transform on the sparse grids. In Chapter 6 our algorithm to do the transform is presented. In Chapter 7 the classical sparse grid is constructed and analyzed. At last, in Chapter 8, we present numerical results, comparing the classical sparse grid to the tensor product grid.

Chapter 2

Mathematical preliminaries

2.1 Hilbert spaces

The notion of Hilbert spaces is important in many fields within applied mathematics. In particular when solving differential equations. In the following, only a short introduction is presented. For a more thorough introduction the reader should turn to a textbook in functional analysis, like [9].

Definition 2.1.1 (Inner-product space)**.** Let $x, y, z \in X$ be elements in a vector space. Then $\langle x, y \rangle$ define an inner product if it has the following properties.

- $\langle x, y \rangle \in \mathbb{C}$
- $\langle x, y \rangle = \overline{\langle y, x \rangle}$
- $\langle \alpha x, y \rangle = \alpha \langle x, y \rangle \,, \;\; \alpha \in \mathbb{C}$
- $\langle x, x \rangle \geq 0$ and $\langle x, x \rangle = 0$ iff $x = 0$
- $\langle x + y, z \rangle = \langle x, z \rangle + \langle y, z \rangle$

X is then an inner-product space with inner product $\langle x, y \rangle$.

Definition 2.1.2 (Hilbert space)**.** H is a Hilbert space if it is a complete inner-product space. E.g. the limits of all Cauchy sequences are in H.

Definition 2.1.3 (Induced norm on a Hilbert space)**.** For $x \in H$, we can define the norm $||x|| = \sqrt{\langle x, x \rangle}$.

2.1.1 Examples of Hilbert spaces

The following Hilbert spaces are the ones which will be considered in this thesis.

Example 2.1.4 (Euclidian space)**.** *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be in* $\mathbb{R}^d$*. Then the Euclidean inner product is the following*

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{d} x_i y_i \tag{2.1}$$

Example 2.1.5 (L_2 space on R^d)**.** *Let* f *and* g *be scalar functions,* $\mathbb{R}^d \to \mathbb{C}$*, such that* $\int_{\mathbb{R}^d} |f(\mathbf{x})|^2 < \infty$*. Then the* L_2 *inner product is the following*

$$\langle f, g \rangle = \int_{\mathbb{R}^d} f(\mathbf{x}) \overline{g(\mathbf{x})} d\mathbf{x} \tag{2.2}$$

The last two Hilbert spaces are the ones we will solve differential equations on. The first is the space the actual problem is defined. The second is the subspace where the discretized problem is defined.

Example 2.1.6 (L_2 space on $[0, 2\pi)^d$)**.** *Let* f *and* g *be scalar functions,* $[0, 2\pi)^d \to \mathbb{C}$*, such that* $\int_{[0,2\pi)^d} |f(\mathbf{x})|^2 < \infty$*. Then the* L_2 *inner product is*

$$\langle f, g \rangle = \int_{[0,2\pi)^d} f(\mathbf{x}) \overline{g(\mathbf{x})} d\mathbf{x} \tag{2.3}$$

Example 2.1.7 (L_2 space on discretized $[0, 2\pi)$)**.** *Let* $f(x)$ *and* $g(x)$ *be defined for* $x \in [0, 2\pi)/h\mathbb{Z}$*. Then the inner product is the Euclidean inner product, summing over all possible values of* x

$$\langle f, g \rangle = \sum_{x \in [0,2\pi)/h\mathbb{Z}} f(x) \overline{g(x)} \tag{2.4}$$

2.1.2 Basis functions

Definition 2.1.8 (Basis set)**.** $\{\phi_i\} \in H$ is a basis set spanning H if for all $f \in H$ there exists a sequence $\{c_i\} \in \mathbb{R}$ such that

$$\left\| \sum_i c_i \phi_i - f \right\| = 0 \tag{2.5}$$

Definition 2.1.9 (Orthonormal basis set)**.** $\{\phi_i\} \in H$ is a orthonormal basis it it is a basis spanning H and have the orthogonality property

$$\langle \Phi_i, \Phi_j \rangle = \delta_{ij} \tag{2.6}$$

Theorem 2.1.10 (Representation in orthogonal basis)**.** *For an orthogonal basis we can find the coefficients using the inner product*

$$c_i = \langle f, \phi_i \rangle \tag{2.7}$$
$$f = \sum_i \langle f, \phi_i \rangle \phi_i \tag{2.8}$$

2.2 Discretization

When solving differential equations numerically, we cannot work directly with continuous data. The data must be defined on a discretized and bounded domain. The unbounded equidistant discretization in one dimension is denoted by $h\mathbb{Z}$, and are all points $x_j = jh$ for $j \in \mathbb{Z}$ [35, p. 9]. This could be generalized to include a shift, setting $x_j = jh + a$.

2.2.1 Equidistant grids

Unbounded grids can not be used directly when computing. If the problem is defined on $(-\infty, \infty)$, some kind of approximation must be done. The easiest way to solve this problem, is by "approximating" infinity by a sufficiently large number M, such that the truncation $(-\infty, \infty) \approx (-M, M)$ yields good results. Discretizing a bounded domain, $[a, b]$, is done by taking the intersection between $1h\mathbb{Z}$ and the domain [21, p. 617]. For convenience x_0 is usually set to a, and x_N to b. Giving the equidistant grid below.

$$x_j = a + jh \tag{2.9}$$
$$h = \frac{b-a}{N} \tag{2.10}$$
$$j \in 0, 1, ..., N \tag{2.11}$$

The periodic grid, which will be a main focus in this thesis is defined similarly, but since it is assumed that for all functions living on the grid have the periodic property $f(a) = f(b)$, the last point is omitted [35, p. 77]. Giving the discretization of $[a, b)$ below.

$$x_j = a + jh \tag{2.12}$$
$$h = \frac{b-a}{N} \tag{2.13}$$
$$j \in 0, 1, ..., N-1 \tag{2.14}$$

2.2.2 Non-equidistant grids

Non-equidistant grid can also be made, by defining a strict monotonic function g, such that $g(x_0) = a$ and $g(x_N) = b$. Where x_j is an element from an equidistant grid. An example is the grid defined by the maxima of a Chebyshev polynomial of degree N [35, p. 42].For $a = -1$ and $b = 1$, they are shown below.

$$x_j = \cos(jh\pi) \tag{2.15}$$
$$h = \frac{1}{N} \tag{2.16}$$
$$j \in 0, 1, ..., N-1 \tag{2.17}$$

2.2.3 Tensor product grid

In higher dimensions, the possibilities for discretizing $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ are many. The most used discretization is the tensor product grid. The tensor product grid in d dimensions can be represented by a set of d-dimensional vectors, where each element of the vector can take the values from a one dimensional grid [8, p. 98].

$$\mathbf{x_j} = (x_{j_1}, x_{j_2}, ..., x_{j_d}) \tag{2.18}$$
$$x_{j_i} = a_i + j_i h_i \tag{2.19}$$
$$h_i = \frac{b_i - a_i}{N_i} \tag{2.20}$$
$$j \in 0, 1, ..., N_i \tag{2.21}$$

2.3 Spectral methods

In spectral methods [8], the idea is to represent functions as weighted sums (or integrals) of a basis set.

$$f\left(\mathbf{x}\right)=\sum_{\mathbf{k}}c(\mathbf{k})\Phi_{\mathbf{k}}\left(\mathbf{x}\right) \tag{2.22}$$

If $\{\Phi_k\}$ is a orthonormal basis set, then $c\left(\mathbf{k}\right)$ is simply found as the projection of f onto each basis element.

$$c\left(\mathbf{k}\right)=\langle f,\Phi_{\mathbf{k}}\rangle \tag{2.23}$$

2.3.1 The dual space

Given a Hilbert space H and a basis set $\{\Phi_{\mathbf{k}}\left(\mathbf{x}\right)\}$, then the dual space D is the Hilbert space where $c(\mathbf{k})$ lives. There is a clear correspondence between the properties of H and D. If H is unbounded, then D is continuous, while a bounded H gives a discrete D. Likewise the other way around. If H is continuous, then D is unbounded, and if H is discrete, D is bounded [35, p. 18].

$$\begin{array}{ccc}\text{Primal} & & \text{Dual}\\ \text{Bounded} & \leftrightarrow & \text{Discrete}\\ \text{Continuous} & \leftrightarrow & \text{Unbounded}\end{array} \tag{2.24}$$

In numerics the bounded discrete version is of most interest, as that gives a finite number of elements.

2.4 The Fourier basis

The focus of this thesis is spectral methods using the Fourier basis [35, 14]. The Fourier basis is the trigonometric polynomials, in one dimension and d dimensions respectively, they are

$$\begin{array}{rcccl}\Phi_k\left(x\right) & = & e^{ikx} & = & \cos\left(kx\right)+i\sin\left(kx\right)\\ \Phi_{\mathbf{k}}\left(\mathbf{x}\right) & = & e^{i\mathbf{k}^T\mathbf{x}} & = & \prod_{j=1}^{d}\left(\cos\left(k_jx_j\right)+i\sin\left(k_jx_j\right)\right)\end{array} \tag{2.25}$$

As the trigonometric polynomials are smooth and periodic, spectral methods based on the Fourier basis typically best approximate smooth and periodic functions.

2.4.1 The Laplacian in Fourier space

In Fourier space, applying the Laplace operator is simple and cheap. The Laplace operator is defined as the sum of all the non-mixed double derivatives

$$\Delta = \nabla^2 = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2} \tag{2.26}$$

The Laplacian of a one dimensional trigonometric function is shown below [14, p. 223].

$$\Delta e^{ikx} = \frac{d^2}{dx^2} e^{ikx} = -k^2 e^{ikx}. \tag{2.27}$$

For d-dimensions this becomes

$$\Delta e^{i\mathbf{k}^T\mathbf{x}} = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2} e^{i\mathbf{k}^T\mathbf{x}} = -\sum_{i=1}^{d} k_i^2 e^{i\mathbf{k}^T\mathbf{x}} = -\left|\left|\mathbf{k}\right|\right|_2^2 e^{i\mathbf{k}^T\mathbf{x}} \tag{2.28}$$

2.5 Solving a PDE with a spectral method

The best way to show how spectral methods work, is to give an example. Consider the Poisson equation (2.29), in d dimensions, on $[0, 2\pi)^d$ with periodic boundary conditions.

$$\Delta u\left(\mathbf{x}\right) = f\left(\mathbf{x}\right) \tag{2.29}$$

$$u\left(\mathbf{x} + 2\pi\mathbf{e_i}\right) = u\left(\mathbf{x}\right),\ i \leq d \tag{2.30}$$

Where $\mathbf{e}_i$ is the ith unit vector. The equation is unique up to a constant.

Expand f and u in the Fourier basis,

$$u = \sum_{\mathbf{k}} c_{\mathbf{k}} e^{i\mathbf{k}^T\mathbf{x}} \tag{2.31}$$

$$f = \sum_{\mathbf{k}} d_{\mathbf{k}} e^{i\mathbf{k}^T\mathbf{x}}, \tag{2.32}$$

and substitute (2.29).

$$\Delta \sum_{\mathbf{k}} c_{\mathbf{k}} e^{i\mathbf{k}^T\mathbf{x}} = \sum_{\mathbf{k}} d_{\mathbf{k}} e^{i\mathbf{k}^T\mathbf{x}} \tag{2.33}$$

$$\sum_{\mathbf{k}} -\,||\mathbf{k}||_2^2\, c_{\mathbf{k}} e^{i\mathbf{k}^T\mathbf{x}} = \sum_{\mathbf{k}} d_{\mathbf{k}} e^{i\mathbf{k}^T\mathbf{x}} \tag{2.34}$$

Due to the orthogonality of the basis, this equation holds if and only if is valid term by term. Thus the coefficients can be found.

$$c_{\mathbf{k}} = -\frac{d_{\mathbf{k}}}{||\mathbf{k}||_2^2},\ \mathbf{k} \neq \mathbf{0} \tag{2.35}$$

$c_{\mathbf{0}}$ can be chosen freely.

Provided that f is sampled on an equidistant grid, we can compute d_k by a DFT. c_k is then found by (2.35). $\widetilde{u}$, a sampled approximation of u, is found by (2.31), using an inverse DFT.

Chapter 3

Introduction to sparse grids

When fighting the curse of dimensionality, it can not be fought for every problem at the same time. In our problem, the core computation is that of computing the Laplacian for the spectral approximation of a function. As always, for numerical computation an infinite expansion needs to be truncated. The idea is to truncate insignificant terms. In 1d the insignificant terms usually are $|k| > N$, for some N. In higher dimensions, there are lots of choices. In the tensor product grid the truncation is done by discarding $||k||_\infty > N$. The type of functions we consider in this thesis, are functions with bounded α-norm (3.1) [27, p. 2],[17, p. 108] (functions living in Korobov space, K^α).

$$||u||_\alpha = \sum_{\mathbf{k}\in\mathbb{Z}} \left|\widetilde{k_1} \cdot \widetilde{k_2} \cdot \ldots \cdot \widetilde{k_d}\right|^{2\alpha} |\widehat{u}\,(\mathbf{k})|^2 \tag{3.1}$$

$$\widetilde{k} = 1, \text{if } k = 0$$
$$\widetilde{k} = k, \text{otherwise.}$$

In this case it the insignificant terms usually are $\left|\widetilde{k_1} \cdot \widetilde{k_2} \cdot \ldots \cdot \widetilde{k_d}\right| > N$, which is the "$d$-ball" in the α-norm.

In Figures 3.1 and 3.2, the area defined by $\left|\widetilde{k_1} \cdot \widetilde{k_2} \cdot \ldots \cdot \widetilde{k_d}\right| > N$ is illustrated for 2- and 3-dimensions. Compared to the corresponding areas defined by the 1-,2- and max-norm Figure 3.3, we see that the alpha-norm area is hesitant to leave the axes. The corresponding area covered is also smaller. This effect becomes even more pronounced in higher dimensions. Thus, being able to exploit this structure when working with this kind of functions should yield huge computational saves.

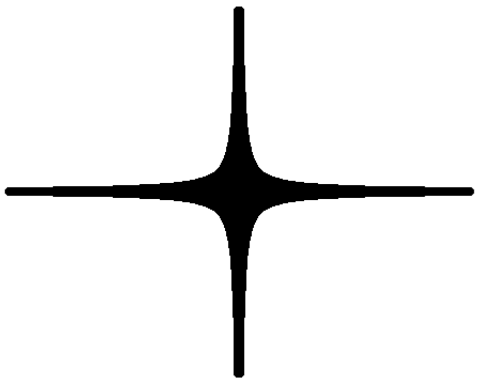


Figure 3.1: Two dimensional alpha norm area

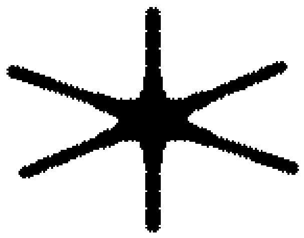


Figure 3.2: Three dimensional alpha norm area

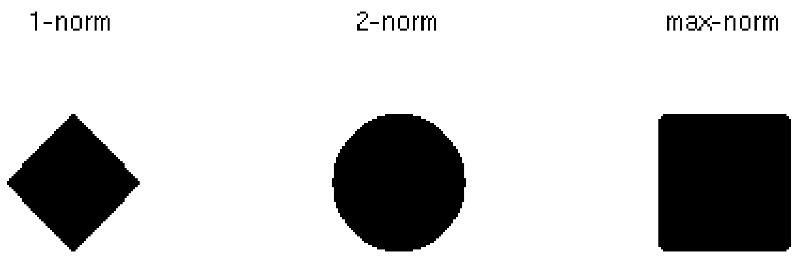


Figure 3.3: 1-,2- and max-norm area

In this and the next chapter, we have made a mathematical theory which presents the family of sparse grids where we can compute the Fourier transform effectively. In this chapter we construct a formal definition of the properties the grids must have.

3.1 The sparse Fourier grid

In this thesis we present a notion of sparse Fourier grids. A set of primal (or physical) grids and corresponding dual grids is proposed below. The grids have properties which makes it possible to efficiently compute the Discrete Fourier Transform (DFT) on them. This may be accomplished using the hierarchical basis set, which will be presented in the next chapter. All grids considered will be subsets of standard tensor product grids (full grid). The sparsity of the grids ranges from the full grid, to grids with elements only on the axes.

Definition 3.1.1 (Fourier sparse primal grid)**.** A Fourier sparse primal grid,S, is a grid with the following properties:

- S is a subset of a full grid of size $N_1 \times N_2 \times \ldots \times N_d$, discretizing $[0, 2\pi)^d$, where $N_i = 2^{n_i}$, for some $n_i \in \mathbb{N}_0$.
- Each primary axis of S have exactly N_i equidistant elements discretizing $[0, 2\pi)$ in the ith component.
- Every line in S parallel to the primary axis in dimension i have 2^k equidistant elements discretizing $[0, 2\pi)$ in the ith component. $k, n_i \in \mathbb{N}_0$ and $k \leq n_i$.

Definition 3.1.2 (Fourier sparse dual grid)**.** A Fourier sparse dual grid, $S^{'}$, is a grid with the following properties:

- $S^{'}$ is a subset of a full grid of size $N_1 \times N_2 \times \ldots \times N_d$, discretizing $\left[-\frac{N_1}{2}+1, \frac{N_1}{2}\right] \times \left[-\frac{N_2}{2}+1, \frac{N_2}{2}\right] \times \ldots \times \left[-\frac{N_d}{2}+1, \frac{N_d}{2}\right]$ equidistantly, where $N_i = 2^{n_i}$, for some $n_i \in \mathbb{N}_0$.
- Each primary axis of S' have exactly N_i elements discretizing $\left[-\frac{N_i}{2}+1, \frac{N_i}{2}\right]$ equidistantly in the ith component.
- Every line in $S^{'}$ parallel to the primary axis in dimension i have 2^k elements discretizing $\left[-\frac{2^k}{2}+1, \frac{2^k}{2}\right]$ equidistantly in the ith component. Where $k, n_i \in \mathbb{N}_0$ and $k \leq n_i$.

From this point on we will simply use the term sparse grid, rather than Fourier sparse grid. This is done for simplicity, but it should be noted that in general the term sparse grid contains many grids not covered by the definition above.

Notice that the fundamental difference between the primal and the dual sparse grid lies in the third part of the definitions. The primal sparse grid discretize the same area as the full primal grid would, but with a different spacing. The dual sparse grid have the same spacing as the full dual grid, but does not discretize the same area.

The choice of $N_i = 2^{n_i}$ is not arbitrary, but necessary to be able to use hierarchical basis, which is presented in the next chapter. This choice is also convenient computationally, as the Fast Fourier Transform algorithms usually works faster on vectors of length of a power of two.

In practice we also need grids which are partial primal and partial dual, these being used when doing a transform which takes a function from one grid to the other. This is done by combining the two definitions, being primal in some dimensions and dual in the other.

3.1.1 Examples of sparse grid pairs in 2d

To give the reader a feeling for what the sparse grid pairs may look like, some examples in two dimensions are presented. All the examples uses the same value of $N\,(=16)$. The three first examples shows a succession of grid getting denser, in Figures 3.4 to 3.6. The last example, Figure 3.7, shows the classical sparse grid, which will be grid to fight the curse of dimensionality with. It will be introduced in Chapter 7.

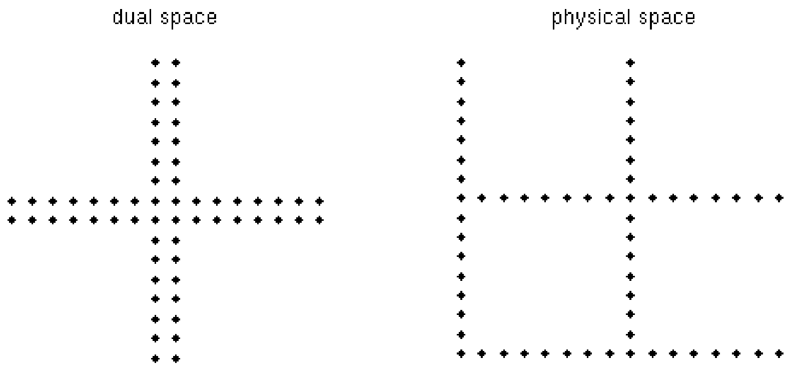


Figure 3.4: A rather sparse sparse grid

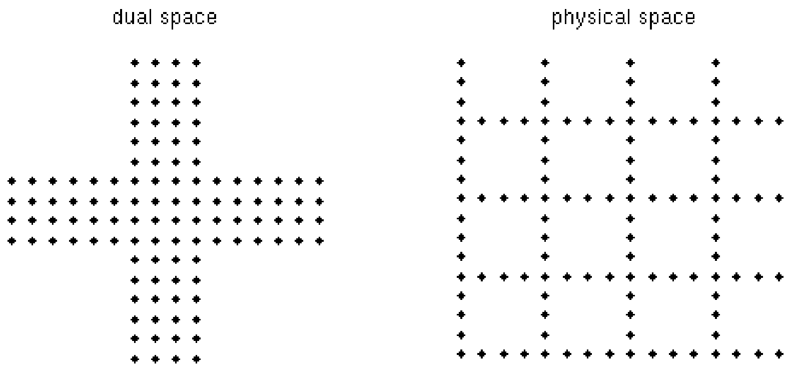


Figure 3.5: A bit denser sparse grid

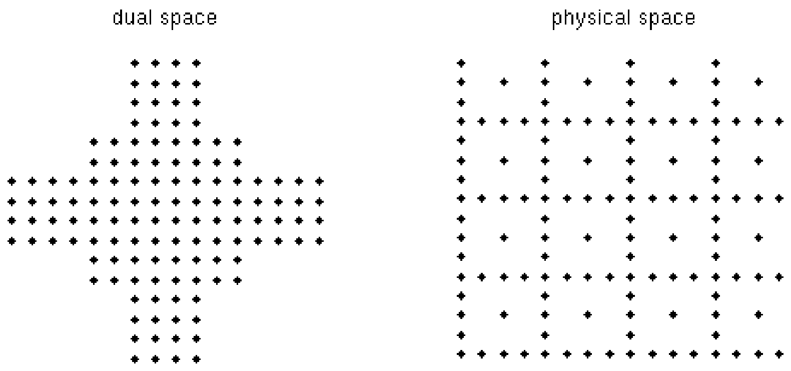


Figure 3.6: An even denser sparse grid

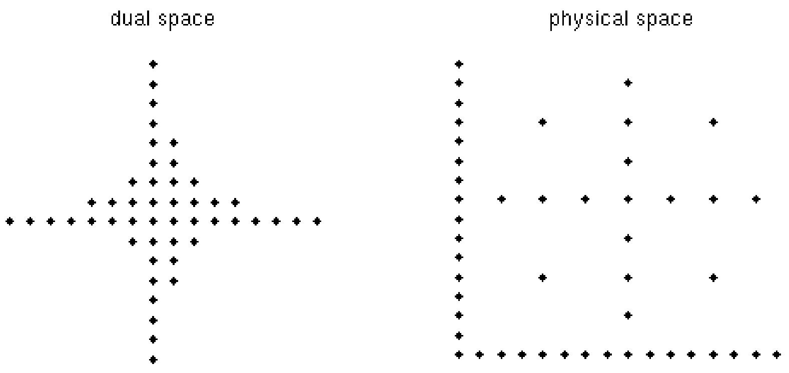


Figure 3.7: The classical sparse grid

Chapter 4

The hierarchical basis

Working with full grids, the normal Fourier basis (2.25) (now referred to as the "nodal basis") is usually all we need to efficiently apply the Fourier transform. On sparse grids though, life is not that easy. The hierarchical basis sets are needed as well. To explain what a hierarchical basis is, a few definitions are needed.

4.1 Some definitions

The level set helps us construct the hierarchy [16, p. 3].

Definition 4.1.1 (Level set)**.** Let I_n be the nth level set. Defined as

$$I_n := \begin{cases} \{0\}, & \text{if } n = 0 \\ \{2^{n-1}, 2^{n-1}+1, \dots, 2^n - 1\}, & \text{if } n > 0 \end{cases} \tag{4.1}$$

The zigzag function takes numbers from $\mathbb{N}_0$ to $\mathbb{Z}$, and is used to get the dual coordinate from the level set [16, p. 3].

Definition 4.1.2 (Zigzag function)**.**

$$\sigma(q) := \begin{cases} -\frac{q}{2}, & \text{if } q \text{ even} \\ \frac{q+1}{2}, & \text{if } q \text{ odd} \end{cases}, \; q \in \mathbb{N}_0 \tag{4.2}$$

The inverse zigzag is used e.g. when we go from a dual coordinate to a primal.

Definition 4.1.3 (Inverse zigzag)**.**

$$\sigma^{-1}(q) := \begin{cases} -2q, & q \leq 0 \\ 2q - 1, & q > 0 \end{cases}, \; q \in \mathbb{Z} \tag{4.3}$$

The modified level set simplifies notation when constructing the dual space.

Definition 4.1.4 (Modified level set)**.**

$$I_n^{'} = \{q \,:\, q = \sigma(k), k \in I_n\} \tag{4.4}$$

Definition 4.1.5 (Grid notation)**.** Let $G_n^{'}$ be the one dimensional dual grid.

$$G_n^{'} = \cup_{i=0}^{n} I_n^{'} \tag{4.5}$$

And let G_n be the one dimensional spatial grid.

$$G_n = \left\{ x \,:\, x = \frac{2\pi k}{2^n}, k \in G_n^{'} \right\} \tag{4.6}$$

$$= \left(\cup_{i=1}^{n} \cup_{q \in I_i} \left\{ \pi \frac{2q - 2^i + 1}{2^{i-1}} \right\} \right) \cup \{0\} \tag{4.7}$$

These sets might equally well be constructed without using the level sets [16, p. 3] (below).

$$G_n = \left\{ \, x : \, x = 2\pi q 2^{-n}, \, q \leq 2^n - 1, \;\; q \in \mathbb{N}_0 \right\} \tag{4.8}$$

$$G_n^{'} = \left\{ k \,:\, \; 1 - 2^{n-1} \leq k \leq 2^{n-1}, \; k \in \mathbb{Z} \right\} \tag{4.9}$$

$$G_0^{'} = \{0\} \tag{4.10}$$

But in our context it is preferable to use definitions (4.5) and (4.7), as they imply a hierarchy of the sets, which makes it easier to look at the intersection between two sets as well as a correspondence between a dual point and a primal point. This will be exploited algorithmically.

As a useful exercise, the definition of the sparse grids can be expressed using the grid notation. The third point in definition 3.1.1 could now be expressed as

- Every line in S parallel to the primary axis in dimension i has the elements from G_k, where $k, n_i \in \mathbb{N}_0$ and $k \leq n_i$.

and for definition 3.1.2

- Every line in $S^{'}$ parallel to the primary axis in dimension i has the elements from $G_k^{'}$. Where $k, n_i \in \mathbb{N}_0$ and $k \leq n_i$.

We need a function to determine from which level a spatial point x is from, e.g. $x \in G^n / G^{n-1}$ says that x is generated in the nth level.

Definition 4.1.6 (spatialFromLevel)**.** Let x be on the form

$$x = \pi \frac{2d+1}{2^{n-1}},\ d, n \in \mathbb{N} \tag{4.11}$$

then x is generated from level set n. Denoted by

$$\begin{array}{rl} spatialFromLevel\left(x\right) = & n,\ x \neq 0 \\ spatialFromLevel\left(0\right) = & 0 \end{array} \tag{4.12}$$

Likewise for k in $G_n^{'}/G_{n-1}^{'}$

Definition 4.1.7 (dualFromLevel)**.** Let $k \in I_n^{'}$, then

$$dualFromLevel\left(k\right) = n \tag{4.13}$$

By looking at (4.1.5), a natural correspondence between each physical and each dual point can be suggested. These functions are one to one from $G_n^{'}$ to G_n and from G_n to $G_n^{'}$ respectively. Making G_n and $G_n^{'}$ a primal-dual couple in one dimension.

Lemma 4.1.8 (Conversion between dual and spatial grid)**.** *Let x be a point in G_n, and k be the corresponding point in $G_n^{'}$.*

$$\begin{array}{cc} x = \frac{\pi\left(2\sigma^{-1}(k)+1-2^{n}\right)}{2^{n-1}}, & n = dualFromLevel\left(k\right),\ k \neq 0 \\ x = 0, & k = 0 \end{array} \tag{4.14}$$

$$\begin{array}{cc} k = \sigma\left(\frac{\frac{x2^{n-1}}{\pi}-1}{2} + 2^{n-1}\right), & n = spatialFromLevel\left(x\right),\ x \neq 0 \\ k = 0, & x = 0 \end{array} \tag{4.15}$$

The transformations are one to one. $x \in G_n/G_{n-1}$ if and only if $k \in G_n^{'}/G_{n-1}^{'}$.

Proof. For $x = k = 0$ the lemma holds trivially. We see from (4.7) that $G_n/G_{n-1} = \cup_{q\in I_n}\left\{\frac{\pi(2\sigma^{-1}(k)+1-2^{n-1})}{2^n}\right\}$, and that (4.14) takes elements from $k \in I_n^{'}$ into this set. Doing some algebra, we show that (4.15) is (4.14), only solving for k.

$$\begin{aligned} x &= \frac{\pi\left(2\sigma^{-1}(k)+1-2^n\right)}{2^{n-1}} \\ \frac{2^{n-1}x}{\pi} &= 2\sigma^{-1}(k)+1-2^n \\ 2\sigma^{-1}(k) &= \frac{2^{n-1}x}{\pi}-1+2^n \\ k &= \sigma\left(\frac{\frac{2^{n-1}x}{\pi}-1}{2}+2^{n-1}\right) \end{aligned}$$

□

In d-dimensions these formulae are used on each component independently.

4.2 The hierarchical basis in one dimension

In this thesis two different discrete Fourier bases will be considered. The nodal has the property of orthogonality, as shown below.

$$\left\{e^{ikx} \,:\, \forall k \in G'_n\right\} \tag{4.16}$$

$$\left\langle e^{ikx}, e^{ilx}\right\rangle = 0,\ k \neq l \tag{4.17}$$

The other is the hierarchical basis. This basis should have properties which are useful when working on sparse grids. Consider a function interpolated by the hierarchical basis, χ_{mk},

$$u_n(x) = \sum_{k\in G'_n} c_{mk}\chi_{mk}(x),\ \ m = dualFromLevel(k) \tag{4.18}$$

The introduction of m in the index is not strictly necessary, as m is uniquely defined by k, but it simplifies notation when introducing the higher dimensional basis, as it keeps track of what the corresponding level set is.

A hierarchical basis has the property that when expanding the basis to G'_{n+1}, the coefficients from G'_n does not change, e.g.

$$u_n(x) = \sum_{k=G'_{n+1}} d_{mk}\chi_{mk}(x), \text{ where } d_k = c_k\ \forall k \in G'_n. \tag{4.19}$$

This is achieved by demanding that the new basis functions disappears on G_n

$$\chi_{(n+1)k}\left(x\right) = 0 \, \forall k \in G_{n+1}^{'}/G_n^{'} \, \forall x \in G_n, , \tag{4.20}$$

such that any inner product on G_n with $\chi_{(n+1)k}\left(x\right)$ will be zero. This means that the basis functions $\left\{\chi_{(n+1)k} \text{ for } k \in I_{n+1}^{'}\right\}$ must span the part of the function space which lives on G_{n+1}/G_n. This again demands that the hierarchical coefficients does not change when expanding the grid, as proposed in definition 4.19.

To get this property, the hierarchical basis takes the form from the definition below (the same, with another notation, as in [16, p. 4])

Definition 4.2.1 (Form of the hierarchical basis)**.**

$$\chi_{nk}\left(x\right) = e^{ikx} - e^{i(k - \text{ sign }(k)2^{n-1})x}, \; k \in G_n^{'}/G_{n-1}^{'} \tag{4.21}$$

$$\chi_{00}\left(x\right) = 1 \tag{4.22}$$

Theorem 4.2.2 (Hierarchical basis)**.** *A hierarchical basis function from level n, (4.2.1), disappears on G_{n-1} as proposed in (4.20).*

Proof.

$$\chi_{nk}\left(x\right) = e^{ikx} - e^{i(k - \text{ sign }(k)2^{n-1})} = e^{ikx}\left(1 - e^{i \text{ sign }(k)2^{n-1}x}\right) \tag{4.23}$$

It must thus be shown that $1 - e^{i \text{ sign }(k)2^{n-1}x} = 0$ for $x \in G_{n-1}$. This is equivalent to showing

$$2^{n-1}x \pmod{2\pi} = 0 \tag{4.24}$$

We know that all $x \in G_{n-1}$ can be written on the form

$$x = \pi\frac{2d+1}{2^m}, \; m, d \in \mathbb{N}_0, m < n \tag{4.25}$$

giving together with (4.24)

$$2^n\pi\frac{2d+1}{2^m} \pmod{2\pi} = 0 \tag{4.26}$$

which holds for $m < n$, and thus for $x \in G_{n-1}$. □

Theorem 4.2.3 (Equivalent bases)**.** *The first k basis function of the nodal and the hierarchical basis spans the same space.*

Proof. It is enough to show that there exist an invertible linear transform from $\mathbb{C}^d$ to $\mathbb{C}^d$ taking the hierarchical basis to the nodal basis. Then it follows from basic linear algebra that they span the same space [23, p. 274]. The hierarchical basis is in fact defined a such a linear transformation (4.21). $\square$

In Figure 4.1 the first six nodal and hierarchical basis functions are plotted (ordered hierarchically). Notice that the hierarchical basis functions have n zeroes (both imaginary and real part equal to zero), and that they are located at G_{n-1}.

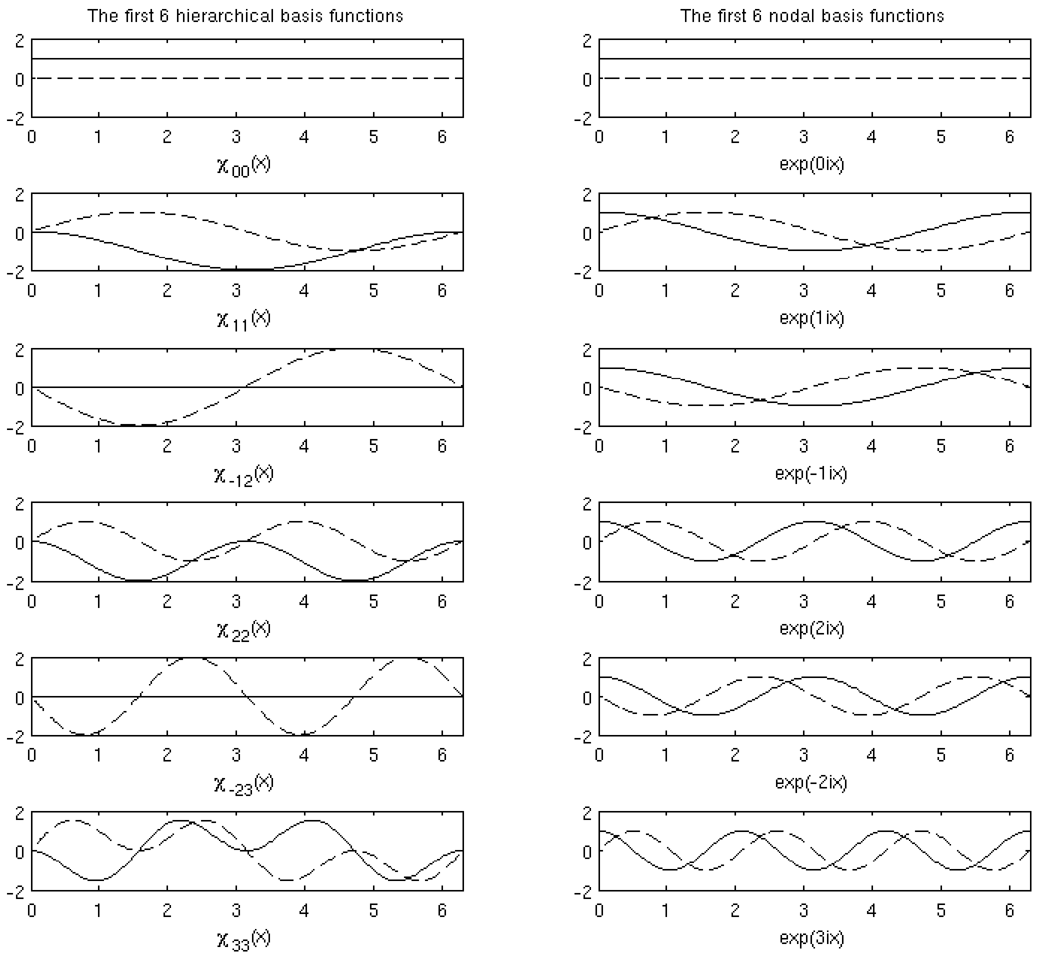


Figure 4.1: The six first nodal and hierarchical basis functions. Real part plotted in solid, imaginary in dotted.

4.3 The hierarchical basis in d dimensions

Generalizing the basis to higher dimensions is done by products of the one dimensional ones [16, pp. 4-5].

$$e^{i\mathbf{k}\cdot\mathbf{x}} = \prod_{l=1}^{d} e^{-ik_l x_l} \tag{4.27}$$

$$\chi_{\mathbf{nk}}\left(\mathbf{x}\right) = \prod_{l=1}^{d} \chi_{n_l k_l}\left(x_l\right) \tag{4.28}$$

Where $\mathbf{n}$ and $\mathbf{k}$ are multiindecies. Again, $\mathbf{n}$ is not strictly necessary, but useful to keep track of the level sets which are being used. A multiindex $\mathbf{n}$ defines uniquely a set of $\mathbf{k}$-multiindecies.

Definition 4.3.1 ($\mathbf{k}$ from $\mathbf{n}$)**.** Let $K\left(\mathbf{n}\right)$ define a set of multiindecies, such that

$$K\left(\mathbf{n}\right) = \left\{\mathbf{k} \in \mathbb{Z}^d \;:\; k_i \in I^{'}_{n_i}\right\} \tag{4.29}$$

Definition 4.3.2 (Partial ordering of $\mathbf{n}$)**.** Let $\mathbf{n}$ and $\mathbf{m}$ be vectors with d elements. Then $\mathbf{n} \geq \mathbf{m}$ iff

$$n_i \geq m_i \quad \forall \quad i \leq d \tag{4.30}$$

Definition 4.3.3 (Children set)**.** Let $\mathbf{n}$ be a multiindex. Then $\mathbf{n}$'s children set contains all multiindecies $\mathbf{m}$ where $\mathbf{m} \leq \mathbf{n}$

Notice that the "mother" $\mathbf{n}$ is a part of its own children set.

With this definition of partial ordering, it is possible to show which basis sets is usable with the sparse grid.

Definition 4.3.4 (Sparse multiindex set)**.** $mutliInd\left(S\right)$ is a sparse multiindex set defining a sparse grid pair (S,S') if it contains multiindecies of the same dimension, and contains all the indecies' children sets.

Stating it another way

Example 4.3.5 (Sparse multiindex set)**.** *If* $multiInd\left(S\right)$ *is a sparse multiindex set, and* $\mathbf{n} \in multiInd\left(S\right)$*. Then for any* $\mathbf{m}$ *such that* $\mathbf{m} \leq \mathbf{n}$, $\mathbf{m} \in multiInd\left(S\right)$.

Definition 4.3.6 (Cover of a sparse multiindex set)**.** Let $Cover\left(S\right)$ be the subset of $multiInd\left(S\right)$ where

$$Cover\left(S\right) = \left\{\mathbf{n} \;:\; \mathbf{n} \nleq \mathbf{m}\, \forall \mathbf{m} \in multiInd\left(S\right)\right\} \tag{4.31}$$

Definition 4.3.7 (Hierarcical basis set)**.** Let $multiInd\,(S)$ define a basis set Υ, where

$$\Upsilon = \left\{ \chi_{\mathbf{nk}} : \mathbf{k} \in \bigcup_{\mathbf{n} \in multiInd(S)} K\,(\mathbf{n}) \right\} \tag{4.32}$$

Υ can also be defined using $Cover\,(S)$, by first introducing a modified $K\,(\mathbf{n})$.

Definition 4.3.8. Let $K^{'}\,(\mathbf{n})$ define a set of multiindecies, such that

$$K^{'}\,(\mathbf{n}) = \left\{ \mathbf{k} \in \mathbb{Z}^{d} : \ k_i \in I^{'}_{n_i} \text{ for } i > 1. - 2^{n_1 1-1} + 1 \leq k_1 \leq 2^{n_1 - 1} \right\} \tag{4.33}$$

Definition 4.3.9 (Hierarcical basis set (using cover))**.** Let $Cover\,(S)$ define a basis set Υ.

$$\Upsilon = \left\{ \chi_{\mathbf{nk}} : \mathbf{k} \in \bigcup_{\mathbf{n} \in Cover(S)} K^{'}\,(\mathbf{n}) \right\} \tag{4.34}$$

The coordinates in dual space $S^{'}$ are now defined from $multiInd\,(S)$ or $Cover\,(S)$.

$$\mathbf{k} \in S^{'} \Leftrightarrow \mathbf{k} \in \bigcup_{\mathbf{n} \in multiInd(S)} K\,(\mathbf{n}) \Leftrightarrow \mathbf{k} \in \bigcup_{\mathbf{n} \in Cover(S)} K^{'}\,(\mathbf{n}) \tag{4.35}$$

We can use the dual space $S^{'}$ and (4.14) to define the coordinates in primal space.

$$\mathbf{x} \in S \Leftrightarrow x_i = \frac{\pi\,(2\sigma^{-1}\,(k_i) + 1 - 2^{n})}{2^{n-1}}, \mathbf{k} \in S^{'} \tag{4.36}$$

Now the primal sparse grid, S and the dual sparse grid, $S^{'}$, can be coupled using the basis set Υ. Υ is a basis set for functions on S, expanded by coefficients in $S^{'}$. The computational and algorithmically details will be presented in the coming chapters.

Chapter 5

Fourier transform on sparse grid

In the previous chapters two different Fourier bases have been introduced, as well as a primal sparse grid and corresponding dual sparse grid. This chapter will present the ideas and theory for a fast transform. The next will go into algorithmic details.

The goal will be to represent a function in the nodal Fourier basis, or get from a representation in the nodal basis to the primal representation.

5.1 Direct approach

The naïve first numerical scheme could be to use the inner product definition directly to find the Fourier coefficients (i.e. the representation in the nodal Fourier basis), by explicitly computing the sum (5.1) on the primal space, S, for each point, $\mathbf{k}$, in dual space, $S^{'}$.

$$\widehat{u}\left(\mathbf{k}\right) = \sum_{\mathbf{x}\in S} w_{\mathbf{x}} u\left(\mathbf{x}\right) e^{-i\mathbf{k}^T\mathbf{x}} \tag{5.1}$$

Where $w_{\mathbf{x}}$ is the weight corresponding to $\mathbf{x}$ for the Gaussian quadrature [15, pp. 301-307]. The weight depends on the sparse grid.

The usual way of finding coefficients for a basis is by solving a linear system. In the case of the Fourier transform, this will be a dense matrix. For the one dimensional case, the normal way of setting up the matrix is by setting $F_{i,j} = we^{-ik_ix_j}$, then finding $\widehat{f}$ from the matrix vector product $\widehat{\mathbf{f}} = F\mathbf{f}$. For a d-dimensional system, a similar system with $F_{i,j} = w_je^{-i\mathbf{k_i}^T\mathbf{x_j}}$ can be set up, given an ordering of $\mathbf{k}$ and $\mathbf{x}$. Finding the coefficients with these methods would be $\mathcal{O}\left(|S|^2\right)$, operations, where $|S|$ is the number of grid points. This should please no one, as the existence of the Fast Fourier Transform (FFT) would make $\mathcal{O}\left(|S|log\left(|S|\right)\right)$ something to hope for.

5.2 Computing the DFT by the FFT on the full grid

On a full grid, the single sum can be split into d sums [26]. In the two dimensional case we get

$$\begin{aligned}
\widehat{f}(\mathbf{k}) &= \sum_{i=-\frac{N_1}{2}+1}^{\frac{N_1}{2}} \sum_{j=-\frac{N_2}{2}+1}^{\frac{N_2}{2}} f(x_i, y_j)\, e^{-i(k_1 x_i + k_2 y_j)} \\
&= \sum_{i=-\frac{N_1}{2}+1}^{\frac{N_1}{2}} e^{-ik_1 x_i} \sum_{i=-\frac{N_2}{2}+1}^{\frac{N_2}{2}} f(x_i, y_j)\, e^{-ik_2 y_j} \qquad (5.2) \\
&= \sum_{i=-\frac{N_1}{2}+1}^{\frac{N_1}{2}} e^{-ik_1 x_i} \widetilde{f}(x_i, k_2),
\end{aligned}$$

where $\widetilde{f}(x_i, k_2) = \sum_{i=-\frac{N_2}{2}+1}^{\frac{N_2}{2}} f(x_i, y_j)\, e^{ik_2 y_j}$ is partially in spatial and dual space.

Each of the sums (DFTs) can be computed using the Fast Fourier Transform. This recursively use of multiple $1d$ FFTs extends straight forward to higher dimensions.

On operator form, using the Kronecker product [31], the same operation can be written as

$$\widehat{f} = \left(\otimes_{i=1}^{d} F_i\right) f$$

Where F_i is the DFT applied along the ith dimension. The idea is to do these operations without making the matrices, but rather use FFT. This will cost $\mathcal{O}\left(dN^d \log(N)\right)$, which beats $\mathcal{O}\left(N^{2d}\right)$ by a landslide. We thus expect that for the sparse grid to offer any speed up, we must be able to use the FFT.

5.3 Hierarchical transform

As the nodal basis and the hierarchical basis spans the same space, there must be a linear transform which maps one into the other. These transforms will be presented in the next chapter, (alg. 3) and (alg. 4). They can be done separately in each dimension. Let F_i be the Fourier transformation in the ith dimension, and H_i be the linear transformation from nodal to hierarchical representation in the

ith dimension. Then the d-dimensional hierarchical Fourier transformation can be written on operator form as

$$\widehat{u}_{hier} = H_d F_d H_{d-1} F_{d-1} \cdots H_i F_i \cdots H_1 F_1 u. \tag{5.3}$$

And the linear transformation from hierarchical basis to nodal basis, let $D_i = H_i^{-1}$

$$\widehat{u}_{nodal} = D_d D_{d-1} \cdots D_i \cdots D_1 \widehat{u}_{hier}. \tag{5.4}$$

The hierarchical Fourier coefficients have the property that $\widehat{u}_{hier}(\mathbf{k})$ takes the same value independent of the sparse grid pair chosen, as long as $\mathbf{k}$ is in the sparse dual space. For example, let $S_N \subset S_M$ be two sparse grids. And let u_N and u_M be interpolated on S_N and S_M respectively, and $\widehat{u}_N$ and $\widehat{u}_M$ be the corresponding hierarchical coefficients, with coefficients from $S_N^{'}$ and $S_M^{'}$. Then

$$\widehat{u}_N(\mathbf{k}) = \widehat{u}_M(\mathbf{k}), \ \forall \mathbf{k} \in S_N^{'} \tag{5.5}$$

Chapter 6

Algorithm

The aim of the theory in the previous chapters is to give the mathematical structure for an effective algorithm to take the Fourier transform on sparse grids, using the Fast Fourier Transform (FFT) [7]. In [16], an algorithm based on recursion is proposed. Some ideas from that paper are used in the following, but we propose another algorithm. The idea is to get rid of the high level recursion, and make an algorithm which is easier to parallelize.

6.1 Data structure

Before presenting the algorithm, we propose a data structure. The motivation behind the data structure, is that it should not use unnecessarily much memory, and it should have properties making it possible to do a fast Fourier transform on a sparse grid.

It is in general not feasible to store a sparse grid in a $N \times N \times \ldots \times N$ grid. Only the function values/coefficients actually computed should be stored in a compressed data structure. This means storing all the entries in a long vector, together with some extra information enabling us to extract specific data elements when needed.

6.1.1 Sparse matrices

Sparse matrices can split into two categories. Matrices with some kind of structure and matrices with no apparent structure. In the second case, there is no other way to construct the data structure than saving additional info about what the corresponding indexes are to a specific data entry [29]. However, if the sparsity pattern has a regular structure, one might be able to construct appropriate formulae/mappings from the stored format to the actual indecies of the data. The way we have chosen to do it, is to have a one dimensional array which saves the data. A function should then be made to go between location in the array, and coordinate in the sparse grid (dual or primal).

The idea behind the data structure is to make it easy and fast to take the discrete Fourier transform in one of the dimensions. And then rearrange the data to take the transform in the other dimensions. We propose three properties for the data structure below.

6.1.2 Sorted in vectors

When taking the one dimensional transformations, each transformation works on single data vector. All the elements of such a vector should be stored consecutive. Such that for each vector that is to be transformed, all the elements from a position a to a position b is in the vector. If this is true for the data structure for the primal sparse grid, it will also be true for the dual sparse grid.

6.1.3 Ordering within vector

Within each vector, it would be convenient to have a the data ordered the way the FFT likes it. This means that the vectors should be ordered ascending in the primal coordinates. Having the ordering in physical space, it can be reordered in dual space, using (4.15).

6.1.4 Ordering of vectors

FFT codes like FFTW(Fastest Fourier Transform in the West) [3], can work even faster if they are able to work on many transformations of the same length. For this reason, it is a good idea to sort the segments in such a way that all the longest vectors comes first, then all the second longest, and so on. This also puts all the vectors with length 1 in the end of the array. Since all the transformations breaks down to the identity transformation for vectors with one element, it is not necessary to do anything to the tail of the array when transforming.

6.1.5 Data access

To work with this data structure, functions to go from coordinate in the sparse (primal or dual) grid to the array address is needed. For a coordinate point $\mathbf{x}$ and correspond array address pos, the functions `pos2Point` and `point2Pos` should uphold the following equalities

$$\texttt{pos2Point}\,(pos) = \mathbf{x} \tag{6.1}$$

$$\texttt{point2Pos}\,(\mathbf{x}) = pos \tag{6.2}$$

Another useful method for us is a function (or a table) to determine how many vectors of each length is in the grid.

$$\texttt{numVec}\,(n) = \text{number of vectors of length } 2^n \tag{6.3}$$

For these functions, special care must be taken when not working with sparse grids which are does not have a symmetric structure, where we can't be certain that if the point (a, b) exists, then (b, a) must exist. In these cases the program must keep track on what way the grid is transposed. In the following it will be assumed that the grid either have a symmetric structure, or that these functions are changed according to transponation.

6.2 Short overview of the algorithm

Given this data structure, there are different approaches to taking the Fourier transform, one could e.g. take the transform by using `point2Pos` to do jumps trough the array to do the transform for each dimensions (using different jump lengths), which is the normal way to do it when working on a full grid.

Our algorithm is based on transposing the data so that we can take the Fourier transform only along the first component. Doing the Fourier transform in place, and then transpose till the transform is done in all dimensions. On a single processor, there is no reason to believe that this should be faster than the idea stated above, but when parallelizing this approach is more convenient.

Let T_i be the operator which rearranges the data, such that dimension i becomes 1 and vice versa. E.g. $T_3 f\,(x_1, x_2, x_3, x_4) = f\,(x_3, x_2, x_1, x_4)$. (5.3) and (5.4) can be rewritten as

$$\widehat{u}_{nodal} = D_1 T_2 D_1 \cdots T_d D_1 H_1 F_1 T_d \cdots T_2 H_1 F_1 u \tag{6.4}$$

We notice that $T_i^{-1} = T_i$ and $D_1^{-1} = H_1$, and let F_1^{-1} be the inverse Fourier transform. Then we can take the inverse transform in the same manner

$$\widehat{u} = F_1 D_1 T_2 \cdots T_d F_1^{-1} D_1 H_1 T_d H_1 \cdots T_2 H_1 \widehat{u}_{nodal} \tag{6.5}$$

The algorithms are the equations (6.4) and (6.5). Written in algorithm style, they become the algorithms on the next page, (alg. 6.2) and (alg. 2)

Algorithm 1 Sparse Fast Fourier Transform (`SFFT`)

```
Let $f$ be a sampled function on a primal sparse grid
function SFFT($f$)
  for $i = 1$ to $dim$ do
    transpose($f, i$)
    manyFFT($f$)
    manyHira($f$)
  end for
  for $i = dim$ to 1 do
    transpose($f, i$)
    manyDehi($f$)
  end for
end function
```

Algorithm 2 Sparse Inverse Fast Fourier Transform (`SIFFT`)

```
Let $f$ be Fourier coefficients on a dual sparse grid
function SIFFT($f$)
  for $i = 1$ to $dim$ do
    transpose($f, i$)
    manyHira($f$)
  end for
  for $i = dim$ to 1 do
    transpose($f, i$)
    manyDehi($f$)
    manyIFFT($f$)
  end for
end function
```

`transpose`(f, i) does as T_i, rearrange the data in f, such that e.g.

$$\texttt{transpose}\left(f\left(x_1, x_2, x_3, x_4\right), 3\right) = f\left(x_3, x_2, x_1, x_4\right).$$

Notice that `transpose`(f, i) always will switch the first component with the ith. `manyFFT`(f) takes one-dimensional FFTs of different lengths along the first component of f. `manyIFFT` likewise, though taking IFFTs. `manyHira`(f) and `manyDehi`(f) does the same as H_1 and D_1, doing the linear transformation from nodal Fourier coefficients to hierarchical Fourier coefficients.

6.3 Transform between nodal and hierarchical coefficients

By exploring the definition of the hierarchical Fourier basis (4.21), a simple algorithm to go from nodal to hierarchical basis is found. For convenience we order the data hierarchically, such that the coefficient for basis function k is at position $\sigma^{-1}(k)$ in the array.

n	$\sigma^{-1}(k)$	k	e^{ikx}	$\chi_{nk}(x)$
0	0	0	1	1
1	1	1	e^{ix}	$e^{ix}-1$
2	2	-1	e^{-ix}	$e^{-ix}-e^{ix}$
	3	2	e^{2ix}	$e^{2ix}-1$
3	4	-2	e^{-2ix}	$e^{-2ix}-e^{2ix}$
	5	3	e^{3ix}	$e^{3ix}-e^{-ix}$
	6	-3	e^{-3ix}	$e^{-3ix}-e^{ix}$
	7	4	e^{4ix}	$e^{4ix}-1$
4	8	-4	e^{-4ix}	$e^{-4ix}-e^{4ix}$
	9	5	e^{5ix}	$e^{5ix}-e^{-3ix}$
	10	-5	e^{-5ix}	$e^{-5ix}-e^{3ix}$
	11	6	e^{6ix}	$e^{6ix}-e^{-2ix}$
	12	-6	e^{-6ix}	$e^{-6ix}-e^{2ix}$
	13	7	e^{7ix}	$e^{7ix}-e^{-ix}$
	14	-7	e^{-7ix}	$e^{-7ix}-e^{ix}$
	15	8	e^{8ix}	$e^{8ix}-1$

(6.6)

Table 6.6 shows the first 16 basis functions for the nodal and hierarchical Fourier basis, exactly from the definition in (4.21). From this definition, and the proposed ordering, the nodal to hierarchical (alg. 3) and hierarchical to nodal (alg. 4) algorithms are constructed, exactly like in [16, p. 12].

The algorithms are quite similar. At first glance the only difference is a sign. Note that to do the computation without temporary storage, the order of the outer loop is essential for correct overwriting. The direction of the inner loop is not important, as the operations does not depend on each other.

Both hira and dehi are both linear operators, and inverse of each other. The matrix representation for $n = 3$ is shown bellow [16, p. 12].

Algorithm 3 `hira`

```
Let $A$ be coefficients represented in nodal basis, with $2^n$ elements
function hira($A$)
  for $i = 1$ to $n$ do
    for $j = 2^i - 1$ to $2^{i-1}$ do
      $A\left(2^i - 1 - j\right) = A\left(2^i - 1 - j\right) + A\left(j\right)$
    end for
  end for
end function
```

Algorithm 4 `dehi`

```
Let $A$ be coefficients represented in hierarchical basis, with $2^n$ elements
function dehi($A$)
  for $i = n$ to $1$ do
    for $j = 2^{i-1}$ to $2^i - 1$ do
      $A\left(2^i - 1 - j\right) = A\left(2^i - 1 - j\right) - A\left(j\right)$
    end for
  end for
end function
```

$$\widehat{hira} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \widehat{dehi} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.7}$$

6.4 Taking consecutive transforms

Having the one dimensional transforms, a simple algorithm comes in to play to do the transforms on the whole data array. Assuming that the data has the structure proposed, and that the `numVec`(n) function is implemented, the `manyFFT`, `manyHira` and `manyDehi` have the same structure.

Here is the array notation from languages like Matlab used. Where $A\left(a : b\right)$ denotes the sub-array of A, starting at position a and ending on position b. Thus

Algorithm 5 `many1dTransform`

```
Let $A$ be an array containing several data vectors
function many1dTransform($A$)
    $startPos$ =numSegments(0)
    for $i = n$ to 1 do
        for $j = 1$ to numSegments($i$) do
            1dTransform($A\,(startPos : startPos + 2^i - 1)$)
            $startPos = startPos + 2^i$
        end for
    end for
end function
```

having $b - a + 1$ elements.

6.5 Transposing

Looking back on (alg. 6.2) and (alg. 2), only one function is left for it to be operable, the transpose function. If not considering memory usage, this can be done simply by making a new array, which is transposed. The idea behind the algorithm (alg. 6) is straight forward. For each point, transpose the data, according to their coordinates (i and 1 change places), using `pos2Point` and `point2Pos`.

Algorithm 6 `transpose` by copy

```
Let $A$ be the array containing data on a sparse grid
Let $B$ be an empty array of the same size as $A$
function transpose($A, transposeDim$)
    for $i = 0$ to length($A$) do
        $point$ =pos2Point($i$)
        $tmpCoor = point\,(1)$
        $point\,(1) = point\,(transposeDim)$
        $point\,(transposeDim) = tmpCoor$
        $newPos$ =point2Pos($point$)
        $B\,(newPos) = A\,(i)$
    end for
    $A = B$
end function
```

It is not preferable to use a copying algorithm though. For the symmetric case, it is simple to find an in place transpose algorithm (alg. 7). By noting that if the value at $A(i)$ is to be moves to $A(j)$, then the value at $A(j)$ is moved to $A(i)$. The convention used is that if $A(i)$ is to moved forward, $A(i)$ and $A(j)$ changes place, otherwise nothing is done.

Algorithm 7 `transpose` in place, symmetric

```
Let $A$ be the array containing data on a sparse grid
function transpose($A, transposeDim$)
  for $i = 0$ to length($A$) do
    $point = $ pos2Point($i$)
    $tmpCoor = point(1)$
    $point(1) = point(transposeDim)$
    $point(transposeDim) = tmpCoor$
    $newPos = $ point2Pos($point$)
    if $newPos > i$ then
      $tmpValue = A(newPos)$
      $A(newPos) = A(i)$
      $A(i) = tmpValue$
    end if
  end for
end function
```

In the case of non-symmetric sparse grids different changes to the transpose algorithm must be made to get an effective in place algorithm. As this depends on the grids, such algorithms are not presented.

6.6 Parallelization

The sparse grid method is used to fight the curse of dimensionality. When fighting a curse, it is useful to combine more weapons. By computing on a cluster, the size of problems that can be solved gets higher, given a method which can be paralleled effectively. With the algorithm presented here, this is quite doable. As has been seen earlier, most of the work is done on one dimensional segments, independently from each other. Thus, it matters little where the computation is done.

Two different schemes will be presented. One with a master node, using slave nodes to compute the transformations, (alg. 8) and (alg. 9) on the next page. This one keeps it easy to hold track of data, as all the data will be stored at the master node, but the effectively of this scheme is not as good, as the master node must send all data, and the transposing of the data must be done on that node as well. The size of problems which can be solved is also limited by the memory capacity of the master node. The second scheme (alg. 10) takes cares of these problems.

There is no master node, all nodes are equals, and the data is spread out trough all of them. In this scheme, though, the data exchange gets more complicated, but more effective and parallelizable.

6.6.1 Master - slave approach

In this approach, the master node initialize the data, just as would be done in the single processor method. The only difference is that when the transformations is done, instead of doing them locally, they are sent to the slaves who do the transformation and send the result back. How to best balance the work between the slaves is an optimization problem, which is not solved here. An easy approach is to send vectors of equal size to each node, and try to share the work fairly.

Algorithm 8 `many1dTransform` on master node

```
Let $A$ be the array containing data on a sparse grid
function many1dTransformMother($A$)
  for $i = 1$ to $nrOfSlaves$ do
    Send segments of $A$ to node $i$
  end for
  for $i = 1$ to $nrOfSlaves$ do
    Receive transformed segments of $A$ from node $i$
  end for
  end function
```

Algorithm 9 `many1dTransform` on slave node

```
function many1dTransformChild()
  Receive work segments from master node
  Do many1dTransform on segments
  Send transformed data back to master
end function
```

6.6.2 Distributed approach

In this parallelizing approach, there is no mother node to hold all the data. The data is rather distributed among all the nodes. This makes it possible to solve much larger systems, by just adding more computing nodes. Each node keeps the data representing some coordinates in the sparse grid, always whole segments. The one dimensional transforms are done on each node. Upon transponation, data is exchanged. This is done in a cycle, first sending data one step “to the right”, then two steps, and so on [22, p. 98]. Figuring out what data is going to which node is not trivial, and depends on how the data is distributed. One way is to distribute the data according to the data structure. The first node works with the data in the first vectors, the second continues where the first stops, and so on. Then the data array is spread out on the nodes, where each node contains all elements between a given position a and b in the data structure. These values can be saved in a table, which then can be used to figure out where data should be sent.

Algorithm 10 Distributed `transpose`

```
$thisNode$ = id number for this node
for $i = 1$ to $nrOfNodes - 1$ do
    Find data to send using pos2Point and point2Pos
    Send data to $thisNode + i$ mod $nrOfNodes$
    Receive data from $thisNode - i$ mod $nrOfNodes$
end for
```

Chapter 7

The classical sparse grid

Finally we are ready to fight the curse of dimensionality. We have the Hilbert space we want to work in (the functions with finite α-norm). The sparse grid construction have been suggested to approximate functions with specific structures in the dual space, and we have an algorithm to do the transform there. Now the sparse grid must be combined with the α-norm Hilbert space. We shall use the name "the classical sparse grid" for this grid [16, p. 5].

7.1 From alpha-norm to the classical sparse grid

As presented in chapter 3, we have to do a cut off in the dual space, such that we only work with coefficients where $|k_1 \cdot k_2 \cdot \ldots \cdot k_d| \leq M$. Assuming $M = 2^N$ and $\alpha = \frac{1}{2}$, we can rewrite this as a sum of logarithms

$$|k_1 k_2 \cdots k_d| \leq 2^N \tag{7.1}$$

$$\log_2(k_1) + \log_2(k_2) + \cdots + \log_2(k_d) \leq N \tag{7.2}$$

Using this observation, we can construct a sparse grid with the wanted properties.

Definition 7.1.1 (The classical sparse grid)**.** Let S_N^d be the classical sparse grid in d dimensions, sub-grid of the $\left(2^N\right)^{\times d}$ tensor product grid, defined by the multiindex set $multInd\left(S_N^d\right)$

$$multInd\left(S_N^d\right) = \left\{\mathbf{n} \in \mathbb{N}_0^d \ : \ ||\mathbf{n}||_1 \leq N\right\} \tag{7.3}$$

Where the dual and primal coordinates in S_n^d is defined as in (4.35) and (4.35) respectively.

The classical grid is made such that S_N^d is a sub-grid of S_{N-1}^d. This is illustrated in Figure 7.1.

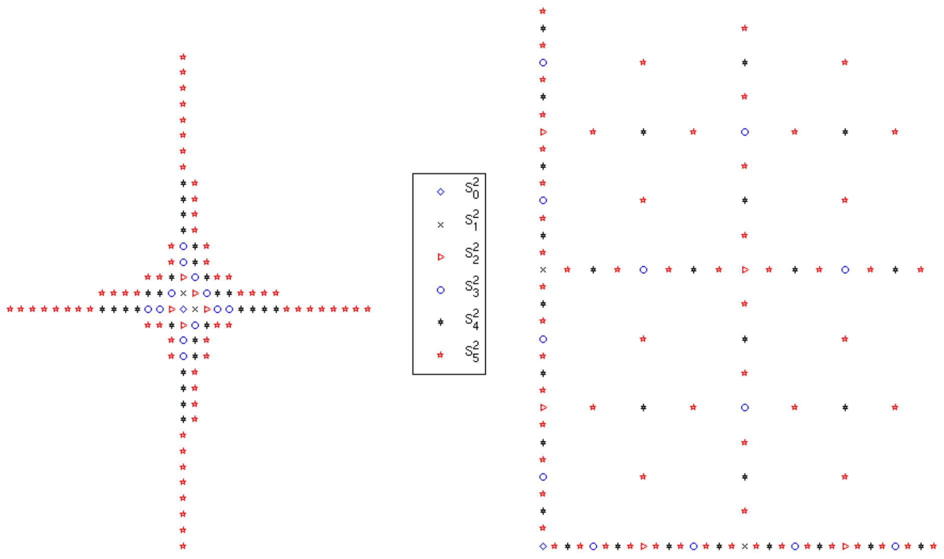


Figure 7.1: Dual and primal classical sparse grid, built up hierarchical

As we see in Figure 7.2 on the next page, the classical sparse dual grid does not fill its corresponding α-norm level set ($|k_1 \cdot k_2 \cdot \ldots \cdot k_d| \leq M$). This is because I'_{n+1} is not symmetric around 0. As $-2^n + 1$ is the largest negative number and 2^n as the largest positive.

7.2 Complexity analysis

For the complexity analysis, we can get nicer formulae using $Cover\left(S_n^d\right)$ rather than $multiInd\left(S_n^d\right)$. The cover is similar to definition 7.1.1, for $multiInd\left(S_n^d\right)$, except using "$=$" rather than "$\geq$".

$$Cover\left(S_N^d\right) = \left\{\mathbf{n} \in \mathbb{N}_0^d \ : \ ||\mathbf{n}||_1 = N\right\} \tag{7.4}$$

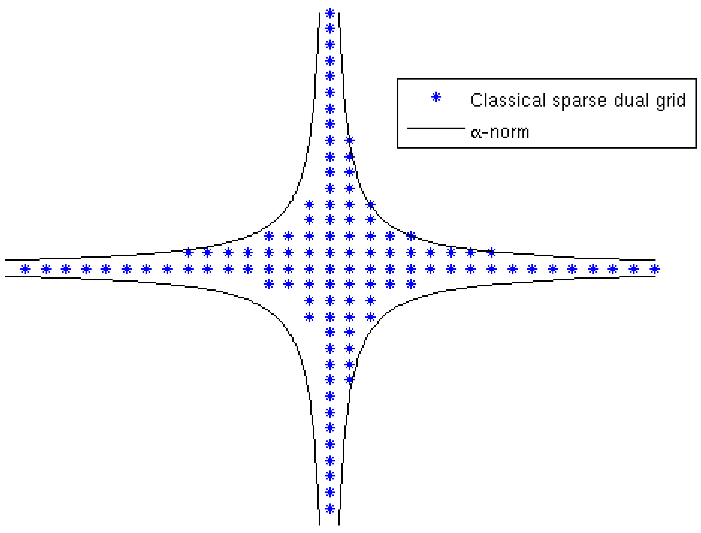


Figure 7.2: The classical sparse dual grid S_5^d compared to the α-norm level curve for $N = 32$

7.2.1 Number of grid points

The full grid will have 2^{dN} grid points. For the classical sparse grid the expression is more complex.

$$\begin{aligned} gridPoints\left(S_N^d\right) &= \sum_{\mathbf{n}\in multiInd\left(S_N^d\right)} 2^{\sum_{i=1}^{d}\max(n_i-1,0)} \\ &= d \sum_{\mathbf{n}\in Cover\left(S_N^d\right)} 2^{\sum_{i=1}^{d} n_i} \end{aligned} \tag{7.5}$$

Where we notice that $2^{\max(n-1,0)}$ is the number of elements in I_n, and 2^{n_1} the number of elements in G_n. (7.5) can be simplified for specific values of d [4],

$$\begin{aligned} gridPoints(S_N^1) &= 2^N \\ gridPoints(S_N^2) &= \left(\tfrac{N}{2}+1\right)2^N \\ gridPoints(S_N^3) &= \left(\tfrac{N^2+7N+8}{8}\right)2^N \\ gridPoints(S_N^4) &= \left(\tfrac{N^3}{48}+\tfrac{5N^2}{16}+\tfrac{7N}{6}+1\right)2^N \end{aligned} \tag{7.6}$$

$N \backslash d$	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	4	8	13	19	26	34	43	53
3	8	20	38	63	96	138	190	253
4	16	48	104	192	321	501	743	1059
5	32	112	272	552	1002	1683	2668	4043
6	64	256	688	1520	2972	5336	8989	14407
7	128	576	1696	4048	8472	16172	28814	48639

Table 7.1: Number of elements in S_N^d for the first 7 values of N and the 8 first dimensions

The $\mathcal{O}\left(N^{d-1}2^N\right)$ complexity holds for $d > 4$ too, with exact formulae becoming increasingly more complex. For fixed d, we get

$$\begin{array}{rcl} \left|S_1^d\right| & = & d+1 \\ \left|S_2^d\right| & = & \frac{d^2+5d+2}{2} \\ \left|S_3^d\right| & = & \frac{(d+5)^3}{6} - \frac{(d+5)^2}{2} - \frac{8(d+5)}{3} + 6 \end{array} \tag{7.7}$$

The $\mathcal{O}\left(d^N\right)$ complexity hold for $N > 3$ too, but with increasing complexity of the formulae.

For practical problems, the "big $\mathcal{O}$" notation might not be all that interesting. There are considerable constants which dampens the effect of the higher order terms. It might thus be more interesting to look at actual values for smaller values of N and d, as in Table 7.1.

7.2.2 Function evaluations for FFT

A one dimensional FFT needs $\mathcal{O}\left(M \log\left(M\right)\right)$ function evaluations, and a corresponding full d-dimensional FFT $\mathcal{O}\left(dM^d \log\left(M\right)\right)$. In the special case of the "radix 2 FFT" the number of operations needed is approximated to $5M\log_2\left(M\right) - 3M = 5N2^N - 3 \cdot 2^N$ in one dimension [13]. In the sparse grid case, the counting becomes more difficult. We will give two formulae for the "radix 2 FFT", one using $multiInd\left(S_N^d\right)$ and one using only the cover, $Cover\left(S_N^d\right)$. When deriving the full multiindex formula, we want to consider how much each hierarchical sub-vector adds to the cost of the FFT. To do this, we need the following identity, which represent the $N2^N$ cost as a sum over the hierarchical building blocks of a

$N \backslash d$	1	2	3	4	5	6	7	8
1	4	11	18	25	32	39	46	53
2	28	76	141	223	322	438	571	721
3	96	300	636	1131	1812	2706	3840	5241
4	272	976	2328	4584	8037	13017	19891	29063
5	704	2864	7584	16344	31044	54051	88246	137071
6	1728	7872	22896	53520	109284	203172	352293	578619
7	4096	20672	65472	164816	359584	710004	1301308	2250323

Table 7.2: Cost of FFT on S_N^d for the first 7 values of N and the 8 first dimensions

vector.

$$N2^N = \sum_{i=0}^{N-1} (i+2)2^i, \tag{7.8}$$

This then says that the cost of each hierarchical sub-vector (as defined by the multiindecies) will be $5\,(i+2)\,2^i - 3i$. Using the cover, we traverse a set of parallel vectors which define the sparse grid. The total cost is then the sum of the cost on these times d (since we must take the transform in all directions).

$$\begin{aligned} fftCost\left(S_N^d\right) &= \sum_{\mathbf{n}\in multiInd\left(S_N^d\right)} \left(\sum_{i=1}^{d} 5 \cdot 2^{\sum_{j=1,j\neq i}^{d} \max(n_j-1,0)} 2^{n_i}\left(n_i+2\right) - 3 \cdot 2^{\sum_{i=1}^{d} \max(n_i-1,0)} \right) \\ &= d \sum_{\mathbf{n}\in Cover\left(S_N^d\right)} 2^{\sum_{i=1}^{d} n_i} \left(5n_1 - 3\right) \end{aligned} \tag{7.9}$$

The first cost values are found in Table 7.2.

7.2.3 Classical sparse grid compared to the tensor product grid

The differences between the number of grid points used in the full tensor product grid and the classical sparse grid are huge when going to higher dimensions. In Figure 7.3 a comparison between the classical sparse grid and the tensor product grid is shown. It shows how many dimensions the classical sparse grid could have computed with, using fewer grid points, than a corresponding M^3 tensor product grid. It says, i.e., that the classical sparse grid S_6^{16} (64 elements on the primary axes, in 16 dimensions) uses fewer than the corresponding full tensor product grid in 3 dimensions $(64 \times 64 \times 64)$.

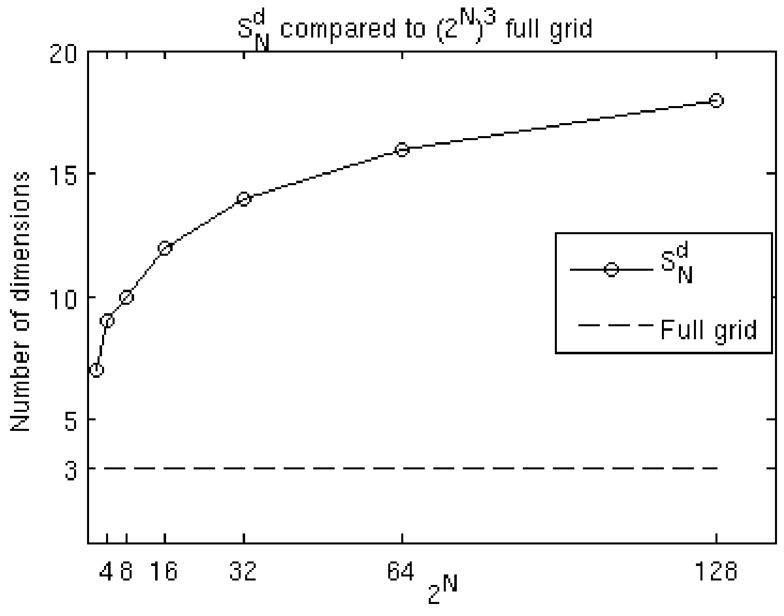


Figure 7.3: Shows the highest dimensional classical sparse grid S_N^d for different values of N, where the number of elements in the sparse grid is fewer than the corresponding full tensor product grid,$|S_N^d| < (2^n)^3$

7.3 Data structure for the classical sparse grid

Recall from chapter 6.2 the three assumptions(6.2.2, 6.2.3 and 6.2.4) on the ordering of the data structure. For fixed indexes $2, ..., d$ data should be stored consecutive by their first index. The longest vectors should be saved first. One more assumption must be made to have a unique ordering of the data array. The assumption about how vectors of the same length should be ordered.

In the following we will use the cover. Each multiindex in the cover represents a set of vectors along the first component. This structure is implied in (7.9), where the cost of the FFT is calculated using the cover. $\mathbf{n}$ defines $2^{\sum_{i=2}^{d} n_i}$ vectors of length 2^{n_1} in primal space. Specifically the vectors below.

Definition 7.3.1 (Vector set)**.** Let $vecSet\left(\mathbf{n}\right)$ be ordered set of vectors

$$\omega = \left(\mathbf{x^{(1)}}, \mathbf{x^{(2)}}, ..., \mathbf{x^{(2^{n_1})}}\right) \tag{7.10}$$

$$x_1^{(j)} = 2\pi\frac{j-1}{2^{n_1}} \tag{7.11}$$

$$x_i^{(j)} = 2\pi\frac{a_i-1}{2^{n_i}} \tag{7.12}$$

$$a_i \in I_{n_i}^{'} \tag{7.13}$$

for all combinations of a_i. Where the lexicographic ordering is used based on a_i, for $i \geq 2$.

Now we have to order $Cover\left(S_N^d\right)$, and we will have a total ordering of the classical sparse grid. We notice that the lexicographic ordering works well here as well. This ascertains that the vectors are ordered descending in length. The position for the dual space is found by converting $\mathbf{k}$ to the corresponding $\mathbf{x}$, using 4.15. Table 7.3 shows the ordering for S_3^3.

The functions `pos2Point` and `point2Pos` must be implemented for the data structure to be operatable. Proof of concept codes, including these functions, where made during the work with this thesis. An algorithm for `point2Pos` is presented in Appendix A. The c++ library (made for Cray XT4) can be found online [30].

adress	k_1	k_2	k_3	$\frac{x_1}{\pi}$	$\frac{x_3}{\pi}$	$\frac{x_3}{\pi}$	n_1	n_2	n_3
1	0	0	0	0	0	0	3	0	0
2	-2	0	0	0.25	0	0	3	0	0
3	-1	0	0	0.5	0	0	3	0	0
4	3	0	0	0.75	0	0	3	0	0
5	1	0	0	1	0	0	3	0	0
6	-3	0	0	1.25	0	0	3	0	0
7	2	0	0	1.5	0	0	3	0	0
8	4	0	0	1.75	0	0	3	0	0
9	0	0	1	0	0	1	2	0	1
10	-1	0	1	0.5	0	1	2	0	1
11	1	0	1	1	0	1	2	0	1
12	2	0	1	1.5	0	1	2	0	1
13	0	1	0	0	1	0	2	1	0
14	-1	1	0	0.5	1	0	2	1	0
15	1	1	0	1	1	0	2	1	0
16	2	1	0	1.5	1	0	2	1	0
17	0	0	-1	0	0	0.5	1	0	2
18	1	0	-1	1	0	0.5	1	0	2
19	0	0	2	0	0	1.5	1	0	2
20	1	0	2	1	0	1.5	1	0	2
21	0	1	1	0	1	1	1	1	1
22	1	1	1	1	1	1	1	1	1
23	0	-1	0	0	0.5	0	1	2	0
24	1	-1	0	1	0.5	0	1	2	0
25	0	2	0	0	1.5	0	1	2	0
26	1	2	0	1	1.5	0	1	2	0
27	0	0	-3	0	0	1.25	0	0	3
28	0	0	-2	0	0	0.25	0	0	3
29	0	0	3	0	0	0.75	0	0	3
30	0	0	4	0	0	1.75	0	0	3
31	0	1	-1	0	1	0.5	0	1	2
32	0	1	2	0	1	1.5	0	1	2
33	0	-1	1	0	0.5	1	0	2	1
34	0	2	1	0	1.5	1	0	2	1
35	0	-3	0	0	1.25	0	0	3	0
36	0	-2	0	0	0.25	0	0	3	0
37	0	3	0	0	0.75	0	0	3	0
38	0	4	0	0	1.75	0	0	3	0

Table 7.3: The data structure for S_3^3

Chapter 8

Numerical results

The aim of the numerical tests is to use different test functions to show how well the classical sparse grid works, compared to the tensor product grid.

8.1 Decay in dual space

When considering which functions will be approximated well on the classical sparse grid, their behavior in dual space is of interest. All functions in L_2 will have Fourier coefficients converging to 0 as $||\mathbf{k}||$ goes to infinity [14, p. 77]. The behavior of this decay depends on the function. We will consider functions which decay as the level set of a given norm. That is to say, the Fourier coefficients $\widehat{u}\left(\mathbf{k}\right)$ are close to constant along the level set given by the "ball" in the norm. The level sets for the α-norm this are defined as the coefficients $\left\{\mathbf{k} \,:\, \prod_{i=1}^{d} \widetilde{k_i} = a\right\}$. For the 1 and 2 norm, the corresponding level sets are $\{\mathbf{k} \,:\, ||\mathbf{k}||_i = a\}$, for i equal to 1 and 2 respectively.

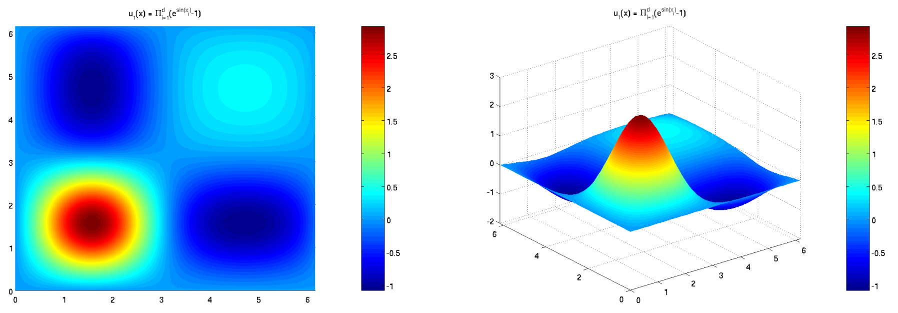
8.2 Test functions

In the numerical experiments, three test functions will be considered. In the trigonometric interpolation, they will be used directly. In the Poisson equation, they will be the solution to the equation. The test functions are chosen such that the Fourier coefficients decay as the 1-norm, 2-norm and α-norm respectively [27].

8.2.1 Test function 1

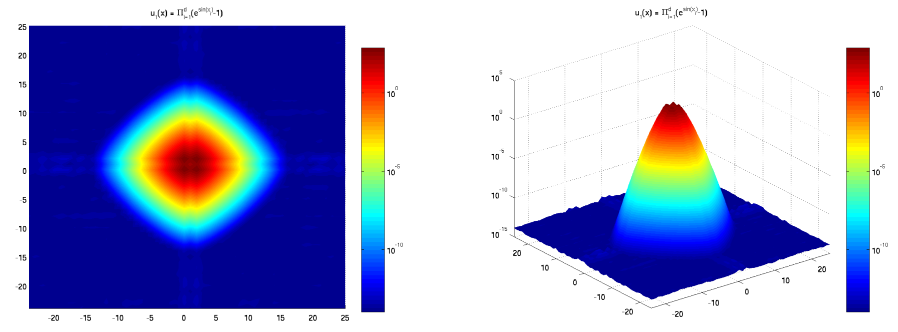
The first test function, u_1, is 2π-periodic and infinitely many times differentiable ($u_1 \in C^\infty$). This makes it ideal for the Fourier spectral methods. But the decay in dual space goes as the 1-norm, so we do not expect it to be an ideal candidate for the classical sparse grid.

$$u_1(\mathbf{x}) = \prod_{i=1}^{d} \left(e^{\sin(x_i)} - 1\right) \tag{8.1}$$



(a) 2d view (b) 3d view

Figure 8.1: Test function 1 in primal space



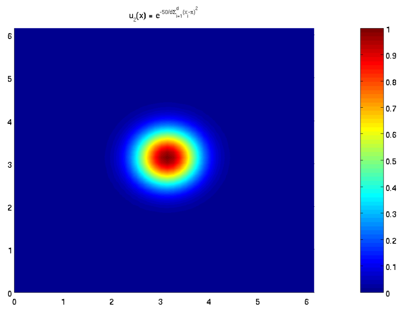
(a) 2d view (b) 3d view

Figure 8.2: Test function 1 in dual space

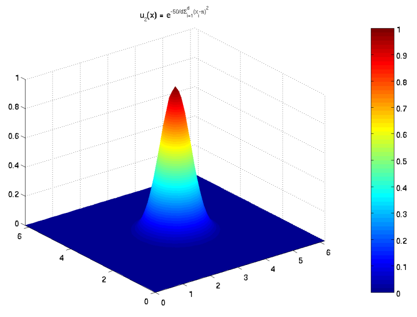
8.2.2 Test function 2

The second test function, u_2 is a Gaussian distribution. It is infinitely many times differentiable ($u_2 \in C^\infty$) , but is not periodic. It has a rapid decay to zero, so by scaling the Gaussian it can be approximated well on a periodic domain. The decay goes as the 2-norm, which still is not good for the classical sparse grid. It is included as a test function because it is an interesting function in quantum mechanics as the ground state solution to the harmonic potential [20, pp. 51-54]. It has also been used in related papers [16, 17, 27].

$$u_2(\mathbf{x}) = e^{-\frac{50}{d}\sum_{i=1}^{d}(x_i-\pi)^2} \tag{8.2}$$

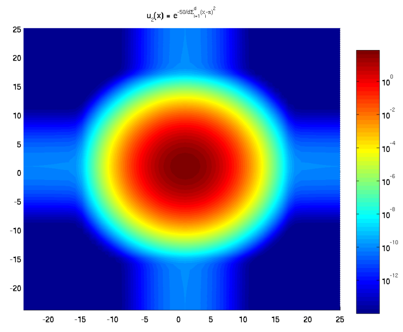


(a) 2d view

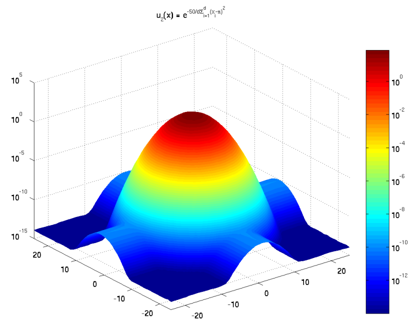


(b) 3d view

Figure 8.3: Test function 2 in primal space



(a) 2d view



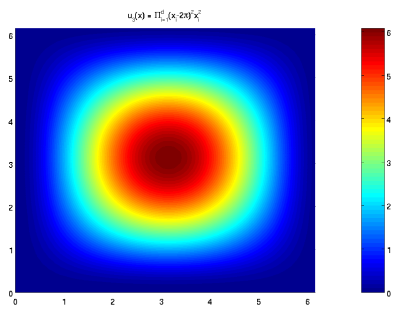
(b) 3d view

Figure 8.4: Test function 2 in dual space

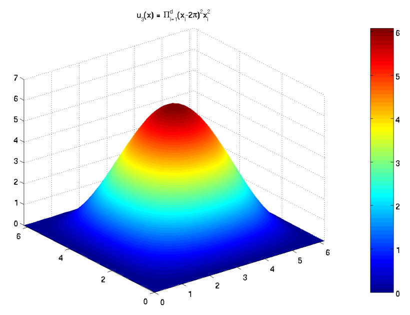
8.2.3 Test function 3

The third test function, u_3, is a polynomial. It is 2π-periodic, but only two times differentiable ($u_3 \in C^2$). In dual space it decays as the α-norm. This test function seem like a good candidate for the classical sparse grid, and we expect to see better result for this than the previous two when using the classical sparse grid.

$$u_3(\mathbf{x}) = \prod_{i=1}^{d} (x_i - 2\pi)^2 x_i^2 \tag{8.3}$$

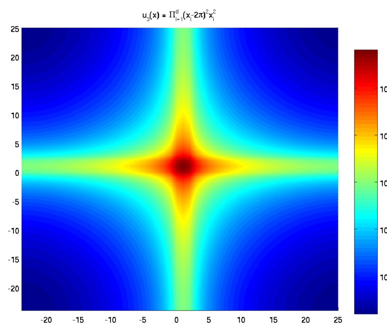


(a) 2d view

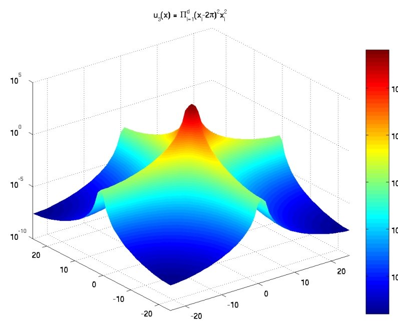


(b) 3d view

Figure 8.5: Test function 3 in primal space



(a) 2d view



(b) 3d view

Figure 8.6: Test function 3 in dual space

8.3 Trigonometric interpolation

The first numerical tests will be to look at how well the functions are interpolated by the Fourier basis on the classical sparse grid. We will use the L_2 norm as a measure on how good the interpolation is. The L_2-norm is evaluated using the identity below [14, p. 221].

$$\begin{aligned} ||u_n||_{L2}^2 &= \int\limits_{[0,2\pi]^d} |u_n(\mathbf{x})|^2 \, d\mathbf{x} \\ &= \int\limits_{[0,2\pi]^d} \sum_{\mathbf{k}\in S'_n} \left|\widehat{u}_n(\mathbf{k}) e^{i\mathbf{x}^T\mathbf{k}}\right|^2 d\mathbf{x} \\ &= \sum_{\mathbf{k}\in S'_n} |\widehat{u}_n(\mathbf{k})|^2 \\ &= ||\widehat{u}_n||_2^2 \end{aligned} \tag{8.4}$$

The interpolation error is estimated taking the difference in calculated L_2-norm between the given grid and the finest grid used, similar to [16, p. 14].

$$error = \left| \frac{||\widehat{u}_N||_2^2 - ||\widehat{u}_{max}||_2^2}{||\widehat{u}_{max}||_2^2} \right| \tag{8.5}$$

For test function 1 the numerical results, Figure 8.7, show that the full grid performs much better than the classical sparse grid. The full grid typically needing an order of magnitude fewer points to get the same accuracy. The same conclusions are drawn for test function 2 in Figure 8.8. For test function 3, the sparse grid performs better. In Figure 8.9 we see that the classical sparse grid gives and accuracy of 10^{-13} using approximately 10^4 grid points, while the full grid needs 10^7 grid points to get the same accuracy. For low accuracy, the full grid needs fewer grid points than the classical sparse grid. In Figure 8.10 we see that the full grid beats the classical sparse grid on accuracy $3 \cdot 10^{-3}$, they need equally many points for accuracy for $3 \cdot 10^{-5}$, and the classical sparse grid performs best when the target accuracy is $3 \cdot 10^{-7}$. This can be explained by noticing that test function 3 has highest magnitude of its Fourier coefficients near the origin, and that the "α-norm" behavior is not as pronounced there. The full grid will thus cover the high magnitude coefficients using fewer grid points than the classical sparse grid. But getting further away from the origin, the α-norm behavior gets more pronounced, and classical sparse grid comes to its right.

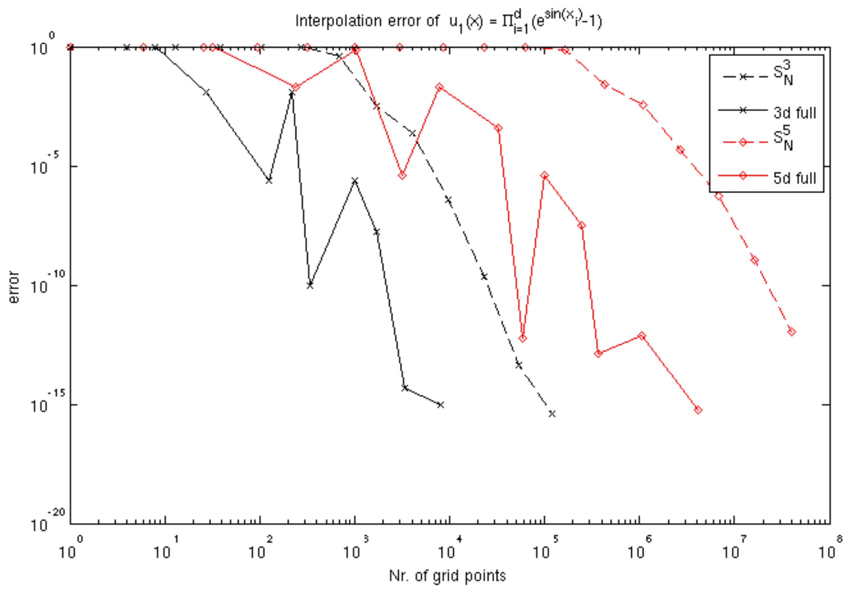


Figure 8.7: Error when interpolating test function 1, in 3 and 5 dimensions. Number of grid points on the x-axis and L_2 error on the y-axis. Full grid plotted in solid, classical sparse grid in dashed. Tthe full grid performs best.

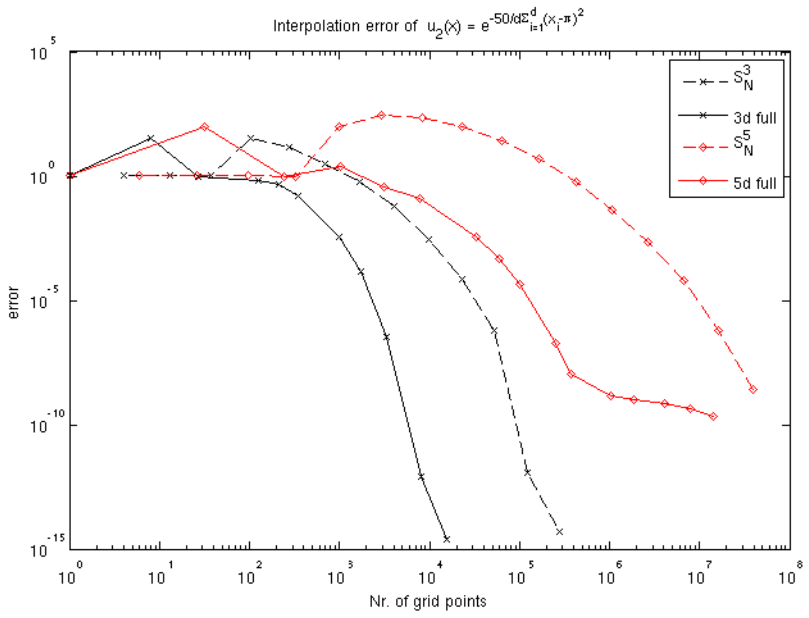


Figure 8.8: Error when interpolating test function 2, in 3 and 5 dimensions. Number of grid points on the x-axis and L_2 error on the y-axis. Full grid plotted in solid, classical sparse grid in dashed. The full grid performs best.

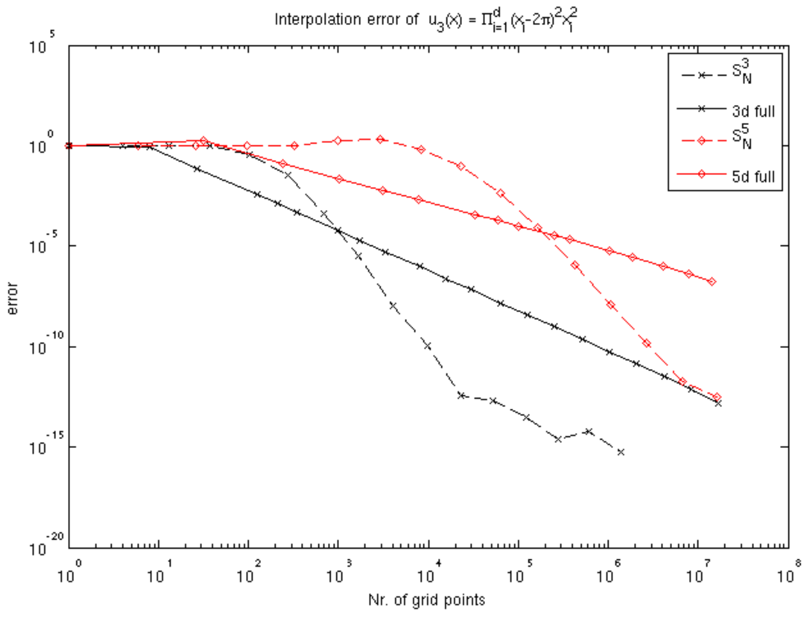


Figure 8.9: Error when interpolating test function 3, in 3 and 5 dimensions. Number of grid points on the x-axis and L_2 error on the y-axis. Full grid plotted in solid, classical sparse grid in dashed. For high accuracy, the classical sparse grid perform better than the full grid.

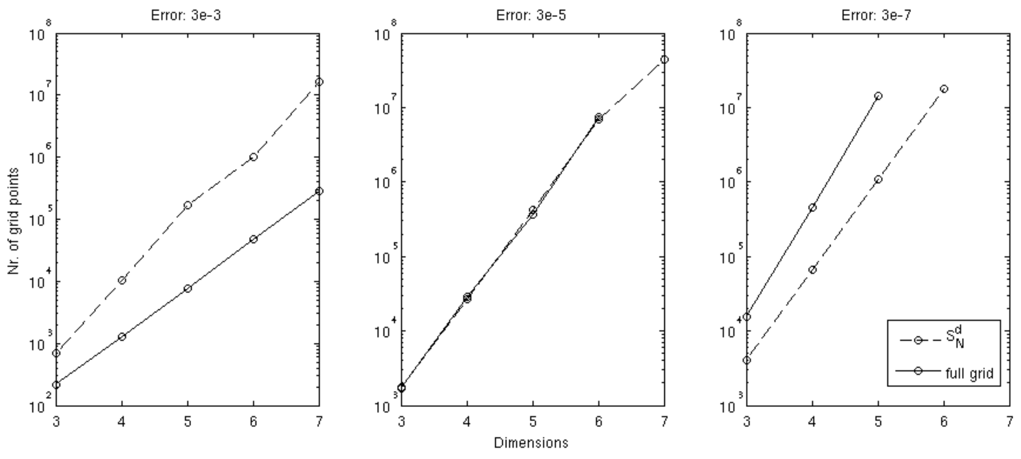


Figure 8.10: Number of grid points needed to achieve an error less than $3 \cdot 10^{-3}$, $3 \cdot 10^{-5}$ and $3 \cdots 10^{-7}$ respectively, for test function 3. The full grid is plotted in solid and the classical sparse grid in dashed. Notice that the lines overlap in the middle plot. For low accuracy, the full grid is better than the classical sparse grid, independent of dimensions. For higher accuracy the classical sparse grid is best, inependent of dimensions.

8.4 The Poisson equation

The Poisson equation, as presented in (2.29), is a simple partial differential equation, which is ideal to see how the spectral Fourier method performs on the classical sparse grid. The right hand side of the equation, (8.6), is chosen so that the solution are the test functions (8.1),(8.2) and (8.3) respectively. They are found by calculating the Laplacian of the test functions.

$$\begin{aligned} f_1(\mathbf{x}) &= \sum_{i=1}^{d} \left(\cos^2(x_i) - \sin(x_i)\right) e^{\sin(x)} \prod_{j=1, j\neq i}^{d} \left(e^{\sin(x_j)} - 1\right) \\ f_2(\mathbf{x}) &= \sum_{i=1}^{d} \left(\frac{100^2}{d^2}(x_i - \pi)^2 - \frac{100}{d}\right) e^{-\frac{50}{d}\sum_{j=1}^{d}(x_j-\pi)^2} \\ f_3(\mathbf{x}) &= \sum_{j=1}^{d} \left(x_j^2 + 4x_j(x_j - 2\pi) + (x_j - 2\pi)^2\right) \prod_{i=1}^{d} (x_i - 2\pi)^2 x_i^2 \end{aligned} \tag{8.6}$$

We know the exact solution for the Poisson equation, by construction they are the test functions. We calculate the relative error using the 2-norm of the difference between the estimated solution and the exact solution on the grid points.

$$error = \frac{||u - u_{est}||_2}{||u||_2} \tag{8.7}$$

The numerical results are similar to that of the interpolating test. The classical sparse grid is not suitable for the two first test functions, but for test function 3 it shows great improvements over the full grid. In Figure 8.11 and Figure 8.14, and Figure 8.12 and Figure 8.15 we see that the two first test functions needs more than an order of magnitude more grid points to converge on the classical sparse grid than the full grid. When going to higher dimensions we see that for fixed number of grid points, the full grid gives a good approximation of test function 1, where the classical sparse grid is not even close to approximating the function. For test function 3, the classical sparse grid is the best choice. In Figure 8.13 we see that the convergence is much more rapid for the classical sparse grid than the full grid. The full grid needs orders of magnitude more grid points for the same accuracy, and the order of magnitude increases when needing higher accuracy. To reach an accuracy of 10^{-1}, we see in Figure 8.17 that the classical sparse grid can achieve this in 7 dimensions using 10^6 grid points, while already in 5 dimensions, the full grid needs over 10^7 grid points. Figure 8.16 shows that for fixed number of grid points $5 \cdot 10^4$, the classical sparse grid achieves 4 orders of accuracy more than the full grid for dimensions 3 to 7.

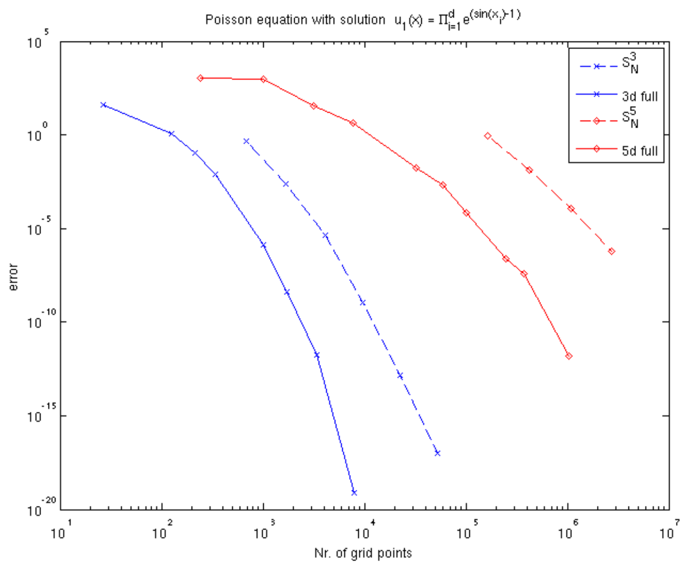


Figure 8.11: Error plots for the Poisson equation for test function 1, in 3 and 5 dimensions. Number of grid points on the x-axis and error on the y-axis. Full grid plotted in solid, classical sparse grid in dashed. The full grid performs best.

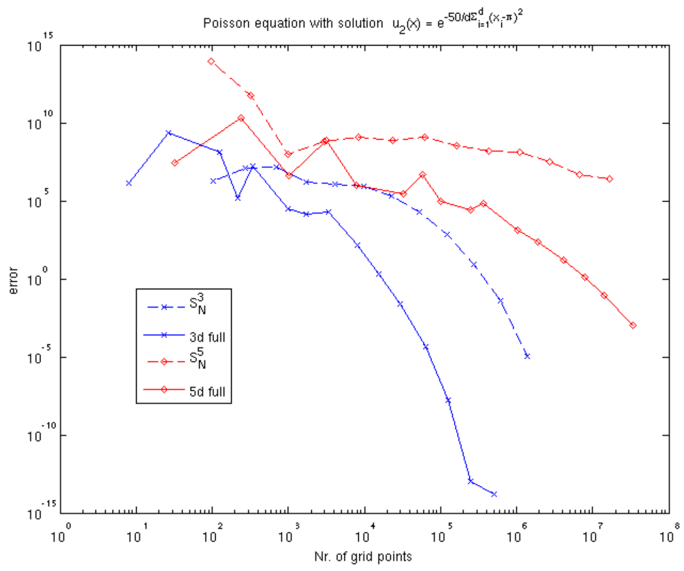


Figure 8.12: Error plots for the Poisson equation for test function 2, in 3 and 5 dimensions. Number of grid points on the x-axis and error on the y-axis. Full grid plotted in solid, classical sparse grid in dashed. The full grid perform best.

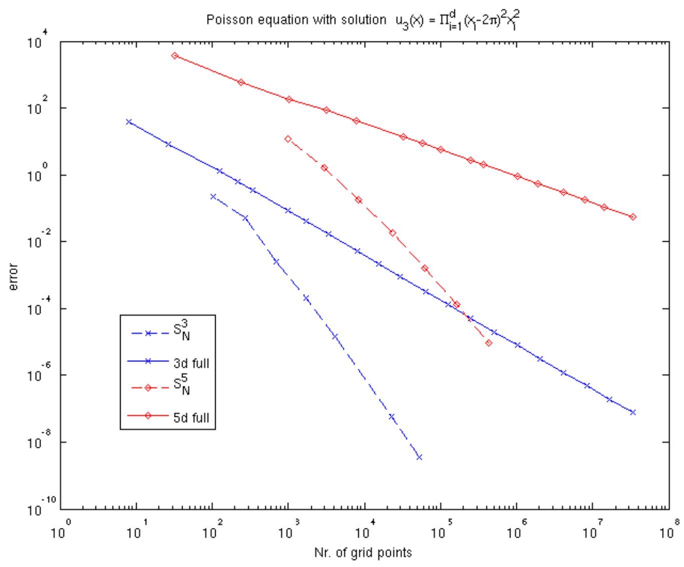


Figure 8.13: Error plots for the Poisson equation for test function 3, in 3 and 5 dimensions. Number of grid points on the x-axis and error on the y-axis. Full grid plotted in solid, classical sparse grid in dashed. The classical sparse grid performs much better than the full grid.

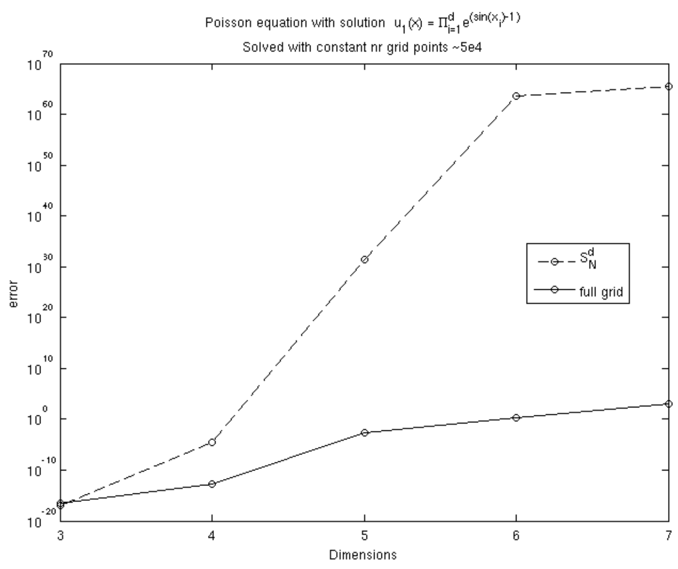


Figure 8.14: Plot of the error for the Poisson equation for fixed number of grid points ($\approx 5 \cdot 10^5$) for test function 1. Dimensions along the x-axis and error along the y-axis.

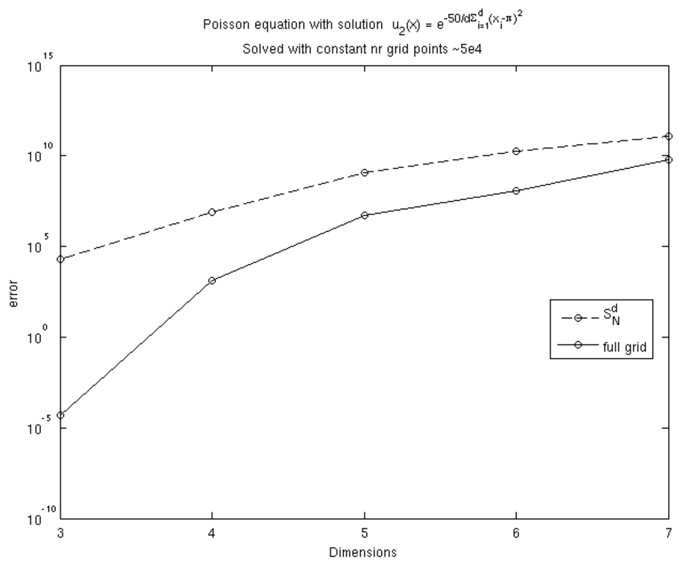


Figure 8.15: Plot of the error for the Poisson equation for fixed number of grid points ($\approx 5 \cdot 10^5$) for test function 2. Dimensions along the x-axis and error along the y-axis.

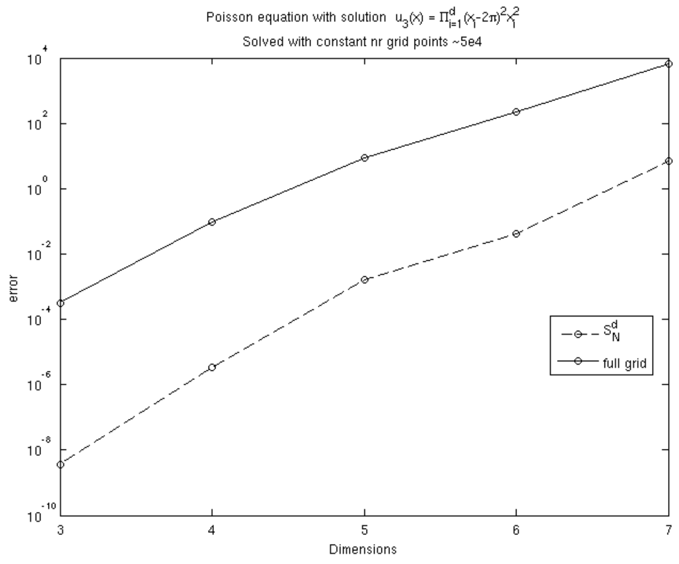


Figure 8.16: Plot of the error for the Poisson equation for fixed number of grid points ($\approx 5 \cdot 10^5$) for test function 3. Dimensions along the x-axis and error along the y-axis. The classical sparse grid have 4 order of magnitude more accuracy than the full grid approach.

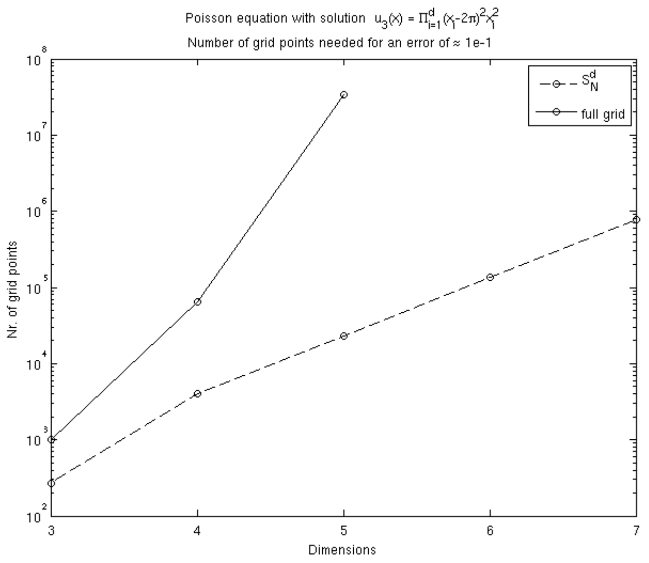


Figure 8.17: Points needed for a error $\approx \cdot 10^{-1}$ for test function 3. Dimensions on the x-axis, number of grid points on the y-axis. The classical sparse grid can solve higher dimensional problems with far fewer grid points than the full grid.

8.5 A problem with many dimensions

The last example is a problem where the tensor product grid is outperformed by the classical sparse grid.

We have a function, f, which is represented as a sum of trigonometric functions and is on the form

$$f(\mathbf{x}) = \sum_{m=1}^{d} \sum_{n=1}^{d} c_{mn} e^{i(x_i + x_j)} \tag{8.8}$$

$$c_{nm} = c_{mn} \; \forall m, n \tag{8.9}$$

We want to determine the coefficients c_{mn} by sampling f on a grid. The degree of the trigonometric polynomials are 2 at most, and none with negative exponents. A $3 \times 3 \times \ldots \times 3$ grid is sufficient to find the exact coefficients using the tensor product grid. For the classical sparse grid approach, we must look at the mixed trigonometric polynomials. At most two components are mixed, we thus need $N = 2$ to cover all coefficients, so S_2^d is sufficient to find the exact coefficients using the classical sparse grid. Looking back on (7.7) we find the number of grid points for the classical sparse grid in this case. The grid points needed for the tensor product grid and the classical sparse grid i respectively

$$|3 \times 3 \times \cdots \times 3| = 3^d \tag{8.10}$$

$$\left|S_2^d\right| = \frac{d^2 + 5d + 2}{2} \tag{8.11}$$

Here the tensor product grid has an exponential growth of grid points needed, while the classical sparse grid have a quadratic (polynomial) growth. The number of unknown coefficients is $\frac{d^2+d}{2}$, so in this case the classical sparse grid only use $2d + 1$ more grid points than the lower bound. For the case $d = 250$, the sparse grid needs 31876 grid points to find the exact coefficients. For the same problem, the tensor product grid would need 10^{120} grid points. It is of course not feasible to approach this high dimensional problem with the tensor product grid (we would need more than a googolbyte of memory to compute), while using the classical sparse grid solves the problem easily (using some kilobytes of memory).

This problem is chosen, as it shows extreme differences between the classical sparse grid and the tensor product grid. The moral is that one should choose a method which fits the problem.

Chapter 9

Conclusion and further work

In this thesis we have presented a class of sparse grids, and an algorithm which can compute the Discrete Fourier Transform effectively on it. We have considered the special case of the classical sparse grid. In this case we have seen that huge computational savings can be gained, if the functions we are working with have the right type of behavior in Fourier space - the coefficients decay as the level sets of the α-norm. For other functions we are better off with the tensor product grid.

Further work is needed to make an effective data structure, so that the method can be used on the huge problems for which it is intended. An effort should also be done, together with physicists, to identify what physical problems this method would be of use. In [18], interesting sparse grids for working with particles in 3d are presented. Further work could be done to implement the Fourier transform on such grids, using the theories from this thesis.

Appendix A

Implementation of `point2Pos`

In this appendix we will present an algorithm for the `point2Pos` method. It should be noted that this algorithm is used for "proof of concept", and that further work should be done to find more effective algorithms.

To make good data access codes, we need to analyze the properties of the data structure. In this section we will revisit and explore the definitions given in Section 6.1 and Section 7.3. Specifically, we are most interested in how many vectors with given properties there are in different parts of the data array. We will consider the classical sparse grid S_n^d, which is a sub-grid of the d-dimensional tensor product grid $[2^n \times 2^n \times \ldots \times 2^n]$.

The data is structured such that vectors along the first dimension is saved consecutively. Where a vector along the first dimensions is defined as coordinates where the last $d-1$ components are fixed, and the first component varies. The longest vectors are stored in the top of the array, then vectors with half the length, and then vectors half of that length again, and so on till we have vectors of length 1 in the end of the array. Recall from Definition 4.3.8, that a multiindex $\mathbf{n}$ define a set of coordinates $K^{'}(\mathbf{n})$.

We will let the multiindecies $\mathbf{k} \in K^{'}(\mathbf{n})$ denote the coordinates in the dual sparse grid. The primal coordinates can be found using (4.14).

We can now present some properties of $K^{'}(\mathbf{n})$. $K^{'}(\mathbf{n})$ contains vectors of length 2^{n_1} along the the first dimension, as each $\mathbf{k} \in K^{'}(\mathbf{n})$ takes all values from 0 to $2^{n_1}-1$ for each combination of $\{k_i\}_{i=2}^{d}$. $K^{'}(\mathbf{n})$ contains $\prod_{i=2}^{d} 2^{\max(n_i-1,0)}$. This because $K^{'}(\mathbf{n})$ have all combinations of $\{k_i\}_{i=2}^{d}$, where k_i can take $|I_{n_i}| = 2^{\max(n_i-1,0)}$ different values.

We can now denote all coordinates in the classical sparse grid using $K^{'}$.

$$\mathbf{k} \in S_n^d \Leftrightarrow \mathbf{k} \in \bigcup_{\mathbf{n} \in Cover\left(S_n^d\right)} K^{'}(\mathbf{n}) \tag{A.1}$$

In Section 6.1.5 we propose a function $\texttt{numVec}(m)$ that tells us how many vectors of length 2^m there are. We find this for S_n^d by finding all $\mathbf{n} \in Cover\left(S_n^d\right)$ where $n_1 = m$, and count how many vectors they contain.

$$\texttt{numVec}_n^d(m) = \sum_{\widetilde{\mathbf{n}} \in A_{n-m}^{d-1}} \prod_{i=1}^{d-1} 2^{\max(\widetilde{n}_i - 1, 0)} \tag{A.2}$$

$$\left\{\widetilde{\mathbf{n}} \in A_m^d : \sum_{i=1}^{d} \widetilde{n}_i = m\right\} \tag{A.3}$$

The values of $\texttt{numVec}_n^d(m)$ will in practice be precomputed to a look up table for needed values of n,d and m. Noticing that in practice only $n - m$ is used in the computation, this becomes a two dimensional array.

The $\texttt{point2Pos}(\mathbf{k})$ algorithm consists of two steps. First figure out where in the array the first element of $K'(\mathbf{n})$ is located (for $\mathbf{k} \in K'(\mathbf{n})$). Then figure out where in $K'(\mathbf{n})$ $\mathbf{k}$ is located. The first part is done by finding the first element in $Cover\left(S_n^d\right)$ where the ith component is equal to n_i, starting with the first component, then find the first with first and second component is correct, and so on until we find $\mathbf{n}$. To find the first multiindex, we just figure out how many vectors there are of length greater than 2^{n_1}, and sum up the space they need in the data array. After this, all vectors we will consider are of length 2^{n_1}. The next steps will now be done using `numVec`. Now get combinatorial problems on the form, "how many ways can we add up to $n - n_1 - n_2 - ... - n_{i-1} - k$ with $d - i$ integers?", and how many vectors does this correspond to. Luckily $\texttt{numVec}_n^{d-i}(n_1 + n_2 + ... + n_{i-1} + k)$. This must be done for $0 \leq k < n_i$. This gives us the first part of the algorithm.

```
$pos = 0$
for $l = n_1 + 1$ to $n$ do
    $pos = pos + 2^l \texttt{numVec}_n^d(l)$
end for
for $i = 2$ to $d$ do
    for $k = 0$ to $n_i - 1$ do
        $pos = pos + 2^{n_1} \texttt{numVec}_n^{d-i}(n_1 + n_2 + ... + k)$
    end for
end for
```

In the second part of the algorithm we look at the vectors within $K'(\mathbf{n})$. The ideas are similar to the first part, only now we try to find where the vector containing $\mathbf{k}$ begins within $K'(\mathbf{n})$. Then we find $\mathbf{k}$'s position in the vector. This time we are working with the vectors where the $i > 1$ component can take the values $k_i \in I_{n_i}$. We introduce a new function, which keeps track of the ordering within the level sets. We proposed in Chapter 7 that the ordering should ascending for $\sigma(k_i)$. The ordering of e.g. I_3 would be $\{6,4,5,7\}$. Our new function `posInSortedLevelSet` will give the position in the level set, given the above ordering. E.g. `posInSortedLevelSet`$(4) = 2$. Now we can once again find the correct vector by fixing first the second component of $\mathbf{k}$, then the second and so on. We do not have to solve the same combinatorial problem to figure out how many vectors we traverse in each step. We know that the last i elements of $\mathbf{n}$ tells us it represent $\prod_{j=i}^{d} 2^{\max(n_j-1,0)}$ vectors of length 2^{n_1} elements. We can thus find the start of the vector containing $\mathbf{k}$ in the second part of the algorithm.

```
**for** $i = 2$ to $d$ **do**
    $pos = pos + 2^{n_i}\left(\texttt{posInSortedLevelSet}\left(k_i\right) - 1\right)\prod_{j=i+1}^{d} 2^{\max(n_j-1,0)}$
**end for**
```

The last part is to find $\mathbf{k}$ within the vector. In the first component the values are ordered ascending in the primal coordinate. Let `posInPrimal`(k,n) give this position. Then the last part of the algorithm becomes

$pos = pos + \texttt{posInPrimal}\left(k_1, n_1\right)$

Giving the complete algorithm for `point2Pos` below.

Algorithm 11 `point2Pos`

```
**function** point2Pos $(\mathbf{k})$
    $pos = 0$
    **for** $l = n_1 + 1$ to $n$ **do**
        $pos = pos + 2^l \texttt{numVec}_n^d\left(l\right)$
    **end for**
    **for** $i = 2$ to $d$ **do**
        **for** $k = 0$ to $n_i - 1$ **do**
            $pos = pos + 2^{n_1}\texttt{numVec}_n^{d-i}\left(n_1 + n_2 + ... + k\right)$
        **end for**
    **end for**
    **for** $i = 2$ to $d$ **do**
        $pos = pos + 2^{n_1}\left(\texttt{posInSortedLevelSet}\left(k_i\right) - 1\right)\prod_{j=i+1}^{d} 2^{\max(n_j-1,0)}$
    **end for**
    $pos = pos + \texttt{posInPrimal}\left(k_1, n_1\right)$
**end function**
```

The `pos2Point` algorithm used is based on much the same ideas. But it is more complex, using a bit of trial and error in finding the point at a given position. The algorithmic details are omitted.

Both `point2Pos` and `pos2Point` uses $\mathcal{O}(dn)$ operations. In the "proof of concept" code [30], an early version of this algorithm is used, and is the main bottleneck in the program, where the two codes takes approximately 99% of the computing time.

Bibliography

[1] Guide to the cray xt4. http://www.parallab.uib.no/resources/hexagon, 2008.

[2] Bergen computational quantum physics group. http://cqp.uib.no/, 2009.

[3] Fftw homepage. http://www.fftw.org/, 2009.

[4] The on-line encyclopedia of integer sequences. http://www.research.att.com/˜njas/sequences/, 2009.

[5] Richard Ernest Bellman. *Adaptive Control Processes: a Guided Tour*. Princeton University Press, Princeton, N.J., 1961.

[6] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. *Acta Numerica*, pages 1–123, 2004.

[7] Phil Burk, Larry Polansky, Douglas Repetto, Mary Roberts, and Dan Rockmore. History of the fft. http://eamusic.dartmouth.edu/ book/MATCpages/chap.3/chap3.pops/3.4.pop.xb1.html, 2002.

[8] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Cang. *Spectral Methods: Fundementals in Single Domains*. Springer, 2006.

[9] Ward Cheney. *Analysis for Applied Mathematics*. Springer, 2001.

[10] K. Cios, W. Pedrycz, and R. Swiniarski. *Data mining methods for knowledge discovery*. Springer, 1998.

[11] R. Cools and B. Maerten. Experiments with smolyak's algorithm for integration over a hypercube. *Technical Report*, 1997.

[12] Time dependent Schrodinger, P. C. Moan, England Cb Ew, S. Blanes, and S. Blanes. Splitting methods for the time-dependent schrÃűdinger equation, 1999.

[13] James W. Cooley et. al. A subspace tracking algorithm using the fast fourier transform. *IEEE signal processing letters*, 11:30–32, 2004.

[14] Gerald B. Folland. *Fourier analysis and its applications*. Brooks/Cole, 1992.

[15] Curtis F. Gerald and Patric O. Wheatly. *Applied Numerical Analysis*. Pearson Education, 2004.

[16] V. Gradinaru. Fourier transform on sparse grids: Code design and the time dependent schrodinger equation. *Computing*, 80:1–22, 2007.

[17] V. Gradinaru. Strang splitting for the time-dependent schrodinger equation on sparse grids. *Siam J. Numer. Anal.*, 46(1):103–123, 2007.

[18] M. Griebel and J. Hamaekers. Sparse grids for the Schrödinger equation. *Mathematical Modelling and Numerical Analysis*, 41(2):215–247, 2007. Special issue on Molecular Modelling. Also as INS Preprint No. 0504.

[19] Michael Griebel and Frank Koster. Adaptive wavelet solvers for the unsteady incompressible navier-stokes equations. In *Advanced Mathematical Theories in Fluid Mechanics*, pages 67–118. Springer, 2000.

[20] Per Christian Hemmer. *Kvantemekanikk*. Tapir, 2005.

[21] David Kincaid and Ward Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole, 2002.

[22] K. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing*. Addison Wesley, 2003.

[23] David C. Lay. *Linear Algebra and its applications*. Addison Wesley, 2003.

[24] J. N. Lyness. An introduction of lattice rules and their generator matrices. *Journal of Numerical Analysis*, pages 405–419, 1988.

[25] J. N. Lyness and T. Sørevik. A search program for finding optimal integration lattices. *Siam Computing*, 47(2):103–120, 1990.

[26] Dave Marshall. The discrete fourier transform (dft). http://www.cs.cf.ac.uk/Dave/Multimedia/node228.html, 2001.

[27] Hans Munthe-Kaas and Tor Sørevik. Multidimensional pseudo-spectral methods on lattice grids. pages 1–14, 2008.

[28] A. Paul. *Kompression von Bildfolgen mit hierarchischen Basen*. PhD thesis, Institut für Informatik, TU München, 1995.

[29] Udo W. Pooch and Al Nieder. A survey of indexing techniques for sparse matrices. *ACM Comput. Surv.*, 5(2):109–133, 1973.

[30] Sveinung Fjær. Sparse fourier transform source files. http://www.student.uib.no/ sfj097/sparseGrid/, 2009.

[31] Kathrin Schäcke. On the kronecker product, 2004.

[32] Ian H. Sloan and J. N. Lyness. The representation of lattice quadrature rules as multiple sums. *Mathematics of Computation*, 52(185):81–94, 1989.

[33] K.T. Taylor, J.S. Parker, D. Dundas, K.J. Meharg, B.J. S. Doherty, D.S. Murphy, and J.F. McCann. Multiphoton double ionization of atoms and molecules by fel xuv light. *Journal of Electron Spectroscopy and Related Phenomena*, 144-147:1191 – 1196, 2005.

[34] Paul A. Tipler and Ralph A. Llewellyn. *Modern Physics*. Freeman, 2003.

[35] Lloyd N. Trefethen. *Spectral Methods in Matlab*. Siam, 2000.