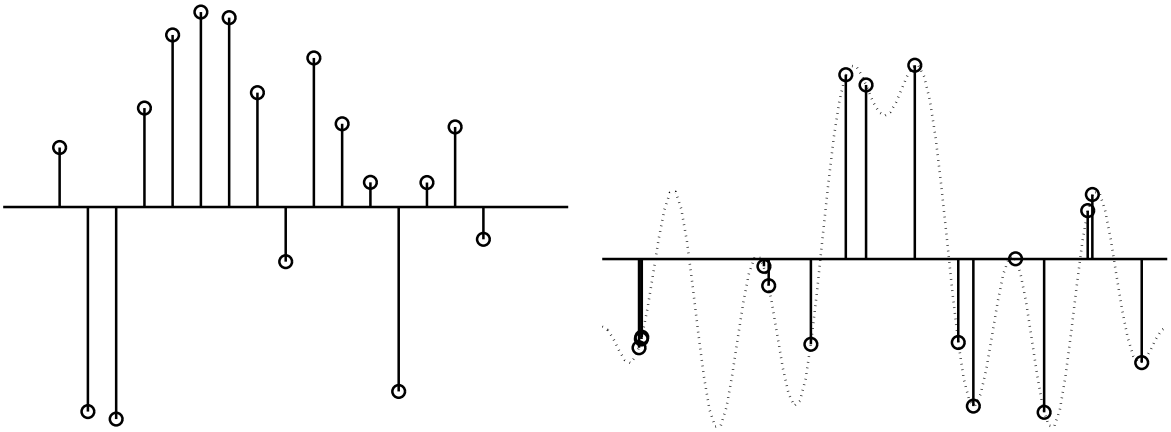
NFFT 3.0 - Tutorial

Jens Keiner* Stefan Kunis† Daniel Potts‡

http://www.tu-chemnitz.de/∼potts/nfft



*keiner@math.uni-luebeck.de,University of Lübeck, Institute of Mathematics, 23560 Lübeck
†kunis@mathematik.tu-chemnitz.de, Chemnitz University of Technology, Department of Mathematics, 09107 Chemnitz, Germany
‡potts@mathematik.tu-chemnitz.de, Chemnitz University of Technology, Department of Mathematics, 09107 Chemnitz, Germany

Contents

1 Introduction

This tutorial surveys the fast Fourier transform at nonequispaced nodes (NFFT), its generalisations, its inversion, and the related C library NFFT 3.0. The goal of this manuscript is twofold – to shed some light on the mathematical background, as well as to provide an overview over the C library to allow the user to use it effectively. Following this, this document splits into two parts:

In Sections **??**, **??** and **??**, we introduce the NFFT which, as a starting point, leads to several concepts for further generalisation and inversion. We focus on a mathematical description of the related algorithms. However, we complement the necessarily incomplete quick glance at the ideas by references to related mathematical literature. In Section **??**, we introduce the problem of computing the discrete Fourier transform at nonequispaced nodes (NDFT), along with the notation used and related works in the literature. The algorithms (NFFT) are developed in pseudo code. Furthermore, we turn for recent generalisations, including in particular NFFTs on the sphere and iterative schemes for inversion of the nonequispaced FFTs, in Section **??** and **??**, respectively.

Having laid the theoretical foundations, Section **??** provides an overview over the NFFT 3.0 C library and the general principles for using the available algorithms in your own code. Rather than describing the library interface in every detail, we restrict to simple recipes in order to familiarise with the very general concepts sufficient for most everyday tasks. For the experienced user, we provide full interface documentation (see `doc/html/index.html` in the package directory) which gives detailed information on more advanced options and parameter settings for each routine. Figure **??** gives an overview over the directory structure of the NFFT 3.0 package. Finally, Section **??** gives some simple examples for using the library and some more advanced applications are given in Section **??** along with numerical results.

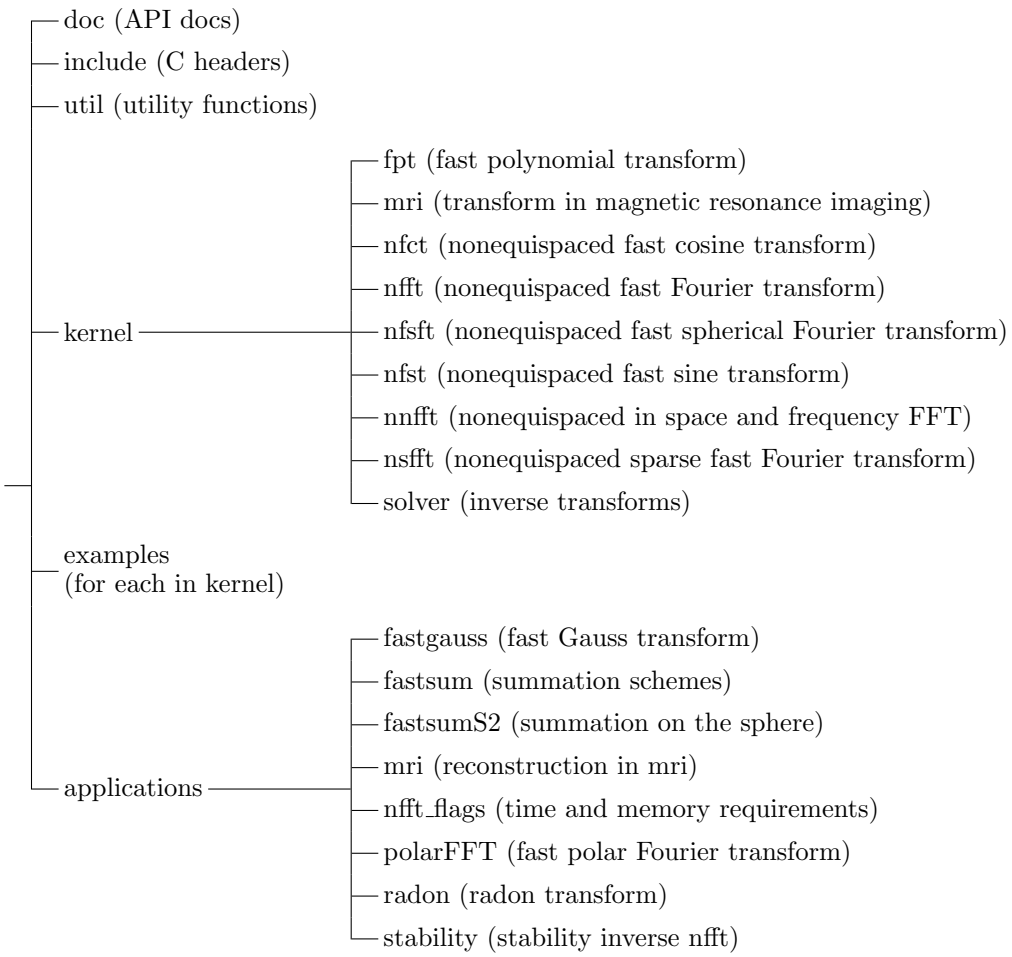


Figure 1.1: Directory structure of the NFFT 3.0 package

2 Notation, the NDFT, and the NFFT

This section summarises the mathematical theory and ideas behind the NFFT. Let the torus

$$\mathbb{T}^d := \left\{ \boldsymbol{x} = (x_t)_{t=0,\ldots,d-1} \in \mathbb{R}^d : -\frac{1}{2} \leq x_t < \frac{1}{2},\ t = 0,\ldots,d-1 \right\}$$

of dimension $d \in \mathbb{N}$ be given. It will serve as domain from which the nonequispaced nodes $\boldsymbol{x}$ are taken. Thus, the sampling set is given by $\mathcal{X} := \{\boldsymbol{x}_j \in \mathbb{T}^d :\ j = 0,\ldots,M-1\}$.

Possible frequencies $\boldsymbol{k} \in \mathbb{Z}^d$ are collected in the multi-index set

$$I_{\boldsymbol{N}} := \left\{ \boldsymbol{k} = (k_t)_{t=0,\ldots,d-1} \in \mathbb{Z}^d : -\frac{N_t}{2} \leq k_t < \frac{N_t}{2},\ t = 0,\ldots,d-1 \right\},$$

where $\boldsymbol{N} = (N_t)_{t=0,\ldots,d-1}$ is the **EVEN** multibandlimit, i.e., $N_t \in 2\mathbb{N}$. To keep notation simple, the multi-index $\boldsymbol{k}$ addresses elements of vectors and matrices as well, i.e., the plain index $\tilde{k} := \sum_{t=0}^{d-1}(k_t + \frac{N_t}{2}) \prod_{t'=t+1}^{d-1} N_{t'}$ is not used here. The inner product between the frequency index $\boldsymbol{k}$ and the time/spatial node $\boldsymbol{x}$ is defined in the usual way by $\boldsymbol{k}\boldsymbol{x} := k_0 x_0 + k_1 x_1 + \ldots + k_{d-1} x_{d-1}$. Furthermore, two vectors may be combined by the component-wise product $\boldsymbol{\sigma} \odot \boldsymbol{N} := (\sigma_0 N_0, \sigma_1 N_1, \ldots, \sigma_{d-1} N_{d-1},)^\top$ with its inverse $\boldsymbol{N}^{-1} := \left(\frac{1}{N_0}, \frac{1}{N_1}, \ldots, \frac{1}{N_{d-1}}\right)^\top$.

The space of all d-variate, one-periodic functions $f : \mathbb{T}^d \to \mathbb{C}$ is restricted to the space of d-variate trigonometric polynomials

$$T_{\boldsymbol{N}} := \operatorname{span}\left(\mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k} \cdot} :\ \boldsymbol{k} \in I_{\boldsymbol{N}} \right)$$

with degree N_t ($t = 0,\ldots,d-1$) in the t-th dimension. The dimension $\dim T_N$ of the space of d-variate trigonometric polynomials T_N is given by $\dim T_N = |I_{\boldsymbol{N}}| = \prod_{t=0}^{d-1} N_t$.

2.1 NDFT - nonequispaced discrete Fourier transform

The first problem to be addressed can be regarded as a matrix vector multiplication. For a finite number of given Fourier coefficients $\hat{f}_{\boldsymbol{k}} \in \mathbb{C}$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$, we consider the evaluation of the trigonometric polynomial

$$f(\boldsymbol{x}) := \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{f}_{\boldsymbol{k}} \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}} \tag{2.1}$$

at given nonequispaced nodes $\boldsymbol{x}_j \in \mathbb{T}^d$. Thus, our concern is the evaluation of

$$f_j = f(\boldsymbol{x}_j) := \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{f}_{\boldsymbol{k}} \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}_j}, \tag{2.2}$$

$j = 0,\ldots,M-1$. In matrix vector notation this reads

$$\boldsymbol{f} = \boldsymbol{A}\hat{\boldsymbol{f}} \tag{2.3}$$

where

$$\boldsymbol{f} := (f_j)_{j=0,\ldots,M-1}, \quad \boldsymbol{A} := \left(\mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}_j} \right)_{j=0,\ldots,M-1;\ \boldsymbol{k} \in I_{\boldsymbol{N}}}, \quad \hat{\boldsymbol{f}} := \left(\hat{f}_{\boldsymbol{k}} \right)_{\boldsymbol{k} \in I_{\boldsymbol{N}}}.$$

The straightforward algorithm for computing this matrix vector product, which is called NDFT, takes $\mathcal{O}(M|I_{\boldsymbol{N}}|)$ arithmetical operations.

A closely related matrix vector product is the adjoint NDFT

$$\hat{\boldsymbol{h}} = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{f}, \qquad \hat{h}_{\boldsymbol{k}} = \sum_{j=0}^{M-1} \hat{f}_j \mathrm{e}^{2\pi \mathrm{i}\boldsymbol{k}\boldsymbol{x}_j},$$

where $\boldsymbol{A}^{\mathsf{H}} = \overline{\boldsymbol{A}}^{\top}$ denotes the conjugate transpose of the nonequispaced Fourier matrix $\boldsymbol{A}$.

Equispaced nodes

For $d \in \mathbb{N},\ N_t = N \in 2\mathbb{N},\ t = 0, \ldots, d-1$ and $M = N^d$ equispaced nodes $\boldsymbol{x}_{\boldsymbol{j}} = \frac{1}{N}\boldsymbol{j},\ \boldsymbol{j} \in I_{\boldsymbol{N}}$ the computation of (**??**) is known as multivariate discrete Fourier transform (DFT). In this special case, the input data $\hat{f}_{\boldsymbol{k}}$ are called discrete Fourier coefficients and the samples $f_{\boldsymbol{j}}$ can be computed by the well known fast Fourier transform (FFT) with only $\mathcal{O}(|I_{\boldsymbol{N}}| \log |I_{\boldsymbol{N}}|)$ arithmetic operations. Furthermore, one has the inversion formula

$$\boldsymbol{A}\boldsymbol{A}^{\mathsf{H}} = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{A} = |I_{\boldsymbol{N}}|\boldsymbol{I},$$

which does **NOT** hold true for the nonequispaced case in general.

2.2 NFFT - nonequispaced fast Fourier transform

For the sake of simplicity, we explain the ideas behind the NFFT for the one-dimensional case $d = 1$ and the algorithm NFFT. The generalisation of the FFT is an approximative algorithm and has computational complexity $\mathcal{O}\left(N \log N + \log\left(1/\varepsilon\right) M\right)$, where ε denotes the desired accuracy. The main idea is to use standard FFTs and a window function φ which is well localised in the time/spatial domain $\mathbb{R}$ and in the frequency domain $\mathbb{R}$. Several window functions were proposed in [**?**, **?**, **?**, **?**, **?**].

The considered problem is the fast evaluation of

$$f\left(x\right) = \sum_{k \in I_N} \hat{f}_k \mathrm{e}^{-2\pi \mathrm{i} k x} \tag{2.4}$$

at arbitrary nodes $x_j \in \mathbb{T},\ j = 0, \ldots, M-1$.

The ansatz

One wants to approximate the trigonometric polynomial f in (**??**) by a linear combination of shifted 1-periodic window functions $\tilde{\varphi}$ as

$$s_1\left(x\right) := \sum_{l \in I_n} g_l\ \tilde{\varphi}\left(x - \frac{l}{n}\right). \tag{2.5}$$

With the help of an oversampling factor $\sigma > 1$, the FFT length is given by $n := \sigma N$.

The window function

Starting with a reasonable window function $\varphi : \mathbb{R} \to \mathbb{R}$, one assumes that its 1-periodic version $\tilde{\varphi}$, i.e.

$$\tilde{\varphi}(x) := \sum_{r \in \mathbb{Z}} \varphi(x + r)$$

has an uniformly convergent Fourier series and is well localised in the time/spatial domain $\mathbb{T}$ and in the frequency domain $\mathbb{Z}$. The periodic window function $\tilde{\varphi}$ may be represented by its Fourier series

$$\tilde{\varphi}(x) = \sum_{k \in \mathbb{Z}} c_k(\tilde{\varphi})\, \mathrm{e}^{-2\pi \mathrm{i} k x}$$

with the Fourier coefficients

$$c_k(\tilde{\varphi}) := \int_{\mathbb{T}} \tilde{\varphi}(x)\, \mathrm{e}^{2\pi \mathrm{i} k x}\, \mathrm{d}x = \int_{\mathbb{R}} \varphi(x)\, \mathrm{e}^{2\pi \mathrm{i} k x}\, \mathrm{d}x = \hat{\varphi}(k), \quad k \in \mathbb{Z}.$$

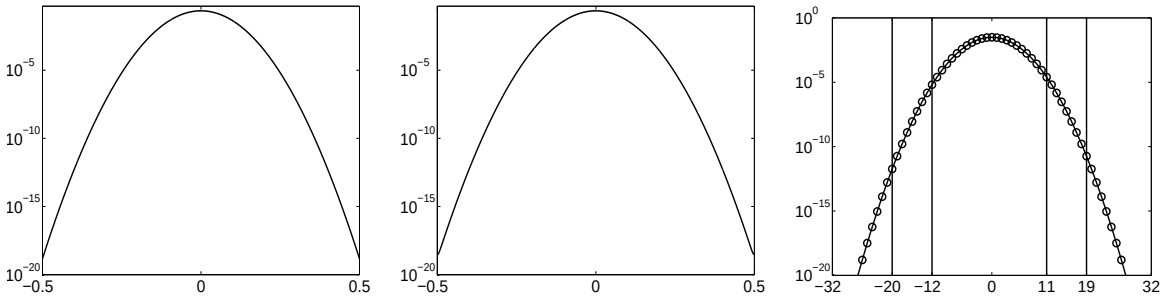


Figure 2.1: From left to right: Gaussian window function φ, its 1-periodic version $\tilde{\varphi}$, and the integral Fourier-transform $\hat{\varphi}$ (with pass, transition, and stop band) for $N = 24$, $\sigma = \frac{4}{3}$, $n = 32$.

The first approximation - cut-off in frequency domain

Switching from the definition (**??**) to the frequency domain, one obtains

$$s_1(x) = \sum_{k \in I_n} \hat{g}_k\, c_k(\tilde{\varphi})\, \mathrm{e}^{-2\pi \mathrm{i} k x} + \sum_{r \in \mathbb{Z} \setminus \{0\}} \sum_{k \in I_n} \hat{g}_k\, c_{k+nr}(\tilde{\varphi})\, \mathrm{e}^{-2\pi \mathrm{i}(k+nr)x}$$

with the discrete Fourier coefficients

$$\hat{g}_k := \sum_{l \in I_n} g_l\, \mathrm{e}^{2\pi \mathrm{i} \frac{kl}{n}}. \tag{2.6}$$

Comparing (**??**) to (**??**) and assuming $c_k(\tilde{\varphi})$ small for $|k| \geq n - \frac{N}{2}$ suggests to set

$$\hat{g}_k := \begin{cases} \frac{\hat{f}_k}{c_k(\tilde{\varphi})} & \text{for } k \in I_N, \\ 0 & \text{for } k \in I_n \setminus I_N. \end{cases} \tag{2.7}$$

Then the values g_l can be obtained from (**??**) by

$$g_l = \frac{1}{n} \sum_{k \in I_N} \hat{g}_k \, \mathrm{e}^{-2\pi \mathrm{i} \frac{kl}{n}} \qquad (l \in I_n),$$

a FFT of size n.

This approximation causes an aliasing error.

The second approximation - cut-off in time/spatial domain

If φ is well localised in time/space domain $\mathbb{R}$ it can be approximated by a function

$$\psi(x) = \varphi(x) \, \chi_{[-\frac{m}{n}, \frac{m}{n}]}(x)$$

with $\operatorname{supp} \psi \subset \left[-\frac{m}{n}, \frac{m}{n}\right]$, $m \ll n$, $m \in \mathbb{N}$. Again, one defines its one periodic version $\tilde{\psi}$ with compact support in $\mathbb{T}$ as

$$\tilde{\psi}(x) = \sum_{r \in \mathbb{Z}} \psi(x + r).$$

With the help of the index set

$$I_{n,m}(x_j) := \{l \in I_n : nx_j - m \leq l \leq nx_j + m\}$$

an approximation to s_1 is defined by

$$s(x_j) := \sum_{l \in I_{n,m}(x_j)} g_l \, \tilde{\psi}\left(x_j - \frac{l}{n}\right). \tag{2.8}$$

Note, that for fixed $x_j \in \mathbb{T}$, the above sum contains at most $(2m+1)$ nonzero summands.

This approximation causes a truncation error.

The case $d > 1$

Starting with the original problem of evaluating the multivariate trigonometric polynomial in (**??**) one has to do a few generalisations. The window function is given by

$$\varphi(\boldsymbol{x}) := \varphi_0(x_0) \, \varphi_1(x_1) \ldots \varphi_{d-1}(x_{d-1})$$

where φ_t is an univariate window function. Thus, a simple consequence is

$$c_{\boldsymbol{k}}(\tilde{\varphi}) = c_{k_0}(\tilde{\varphi}_0) \, c_{k_1}(\tilde{\varphi}_1) \ldots c_{k_{d-1}}(\tilde{\varphi}_{d-1}).$$

The ansatz is generalised to

$$s_1(\boldsymbol{x}) := \sum_{\boldsymbol{l} \in I_{\boldsymbol{n}}} g_{\boldsymbol{l}} \, \tilde{\varphi}\left(\boldsymbol{x} - \boldsymbol{n}^{-1} \odot \boldsymbol{l}\right),$$

where the FFT size is given by $\boldsymbol{n} := \boldsymbol{\sigma} \odot \boldsymbol{N}$ and the oversampling factors by $\boldsymbol{\sigma} = (\sigma_0, \ldots, \sigma_{d-1})^\top$. Along the lines of (**??**) one defines

$$\hat{g}_{\boldsymbol{k}} := \begin{cases} \frac{\hat{f}_{\boldsymbol{k}}}{c_{\boldsymbol{k}}(\tilde{\varphi})} & \text{for } \boldsymbol{k} \in I_{\boldsymbol{N}}, \\ 0 & \text{for } \boldsymbol{k} \in I_{\boldsymbol{n}} \backslash I_{\boldsymbol{N}}. \end{cases}$$

The values $g_{\boldsymbol{l}}$ can be obtained by a (multivariate) FFT of size $n_0 \times n_1 \times \ldots \times n_{d-1}$ as

$$g_{\boldsymbol{l}} = \frac{1}{|I_{\boldsymbol{n}}|} \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{g}_{\boldsymbol{k}} \, \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k} \left(\boldsymbol{n}^{-1} \odot \boldsymbol{l} \right)}, \quad \boldsymbol{l} \in I_{\boldsymbol{n}}.$$

Using the compactly supported function $\psi(\boldsymbol{x}) = \varphi(\boldsymbol{x}) \, \chi_{[-\frac{m}{n}, \frac{m}{n}]^d}(\boldsymbol{x})$, one obtains

$$s(\boldsymbol{x}_j) := \sum_{\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)} g_{\boldsymbol{l}} \, \tilde{\psi}\left(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l} \right),$$

where $\tilde{\psi}$ again denotes the one periodic version of ψ and the multi-index set is given by

$$I_{\boldsymbol{n},m}(\boldsymbol{x}_j) := \{ \boldsymbol{l} \in I_{\boldsymbol{n}} : \boldsymbol{n} \odot \boldsymbol{x}_j - m\boldsymbol{1} \leq \boldsymbol{l} \leq \boldsymbol{n} \odot \boldsymbol{x}_j + m\boldsymbol{1} \}.$$

The algorithm

In summary, the following Algorithm **??** is obtained for the fast computation of (**??**) with $\mathcal{O}(|I_{\boldsymbol{n}}| \log |I_{\boldsymbol{n}}| + mM)$ arithmetic operations.

Input: $d, M \in \mathbb{N}$, $\boldsymbol{N} \in 2\mathbb{N}^d$
$\boldsymbol{x}_j \in [-\frac{1}{2}, \frac{1}{2}]^d$, $j = 0, \ldots, M-1$, and $\hat{f}_{\boldsymbol{k}} \in \mathbb{C}$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$,

1: For $\boldsymbol{k} \in I_{\boldsymbol{N}}$ compute

$$\hat{g}_{\boldsymbol{k}} := \frac{\hat{f}_{\boldsymbol{k}}}{|I_{\boldsymbol{n}}| \, c_{\boldsymbol{k}}(\tilde{\varphi})}.$$

2: For $\boldsymbol{l} \in I_{\boldsymbol{n}}$ compute by d-variate FFT

$$g_{\boldsymbol{l}} := \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{g}_{\boldsymbol{k}} \, \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k} \left(\boldsymbol{n}^{-1} \odot \boldsymbol{l} \right)}.$$

3: For $j = 0, \ldots, M-1$ compute

$$f_j := \sum_{\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)} g_{\boldsymbol{l}} \, \tilde{\psi}\left(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l} \right).$$

Output: approximate values f_j, $j = 0, \ldots, M-1$.
Complexity: $\mathcal{O}(|\boldsymbol{N}| \log |\boldsymbol{N}| + M)$.

Algorithm 1: `NFFT`

Algorithm **??** reads in matrix vector notation as

$$\boldsymbol{A}\hat{\boldsymbol{f}} \approx \boldsymbol{B}\boldsymbol{B}\boldsymbol{B}\hat{\boldsymbol{f}},$$

where $\boldsymbol{B}$ denotes the real $M \times |I_{\boldsymbol{n}}|$ sparse matrix

$$\boldsymbol{B} := \left(\tilde{\psi}\left(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l} \right) \right)_{j=0,\ldots,M-1; \boldsymbol{l} \in I_{\boldsymbol{n}}},$$

where $\boldsymbol{B}$ is the Fourier matrix of size $|I_{\boldsymbol{n}}| \times |I_{\boldsymbol{n}}|$, and where $\boldsymbol{B}$ is the real $|I_{\boldsymbol{n}}| \times |I_{\boldsymbol{N}}|$ 'diagonal' matrix

$$\boldsymbol{B} := \bigotimes_{t=0}^{d-1} \left(\boldsymbol{O}_t \,|\, \operatorname{diag}\left(1/\, c_{k_t}\left(\tilde{\varphi}_t\right)\right)_{k_t \in I_{N_t}} \,|\, \boldsymbol{O}_t \right)^{\mathrm{T}}$$

with zero matrices $\boldsymbol{O}_t$ of size $N_t \times \frac{n_t - N_t}{2}$.

The corresponding computation of the adjoint matrix vector product reads as

$$\boldsymbol{A}^{\mathsf{H}} \hat{\boldsymbol{f}} \approx \boldsymbol{B}^{\top} \boldsymbol{B}^{\mathsf{H}} \boldsymbol{B}^{\top} \hat{\boldsymbol{f}}\,.$$

With the help of the transposed index set

$$I^{\top}_{\boldsymbol{n},m}\left(\boldsymbol{l}\right) := \left\{ j = 0, \ldots, M-1 : \boldsymbol{l} - m\boldsymbol{1} \leq \boldsymbol{n} \odot \boldsymbol{x}_j \leq \boldsymbol{l} + m\boldsymbol{1} \right\},$$

one obtains Algorithm **??** for the adjoint NFFT. Due to the characterisation of the nonzero

Input: $d, M \in \mathbb{N}$, $\boldsymbol{N} \in 2\mathbb{N}^d$
$\boldsymbol{x}_j \in [-\frac{1}{2}, \frac{1}{2}]^d$, $j = 0, \ldots, M-1$, and $f_j \in \mathbb{C}$, $j = 0, \ldots, M-1$,

1: For $\boldsymbol{l} \in I_{\boldsymbol{n}}$ compute

$$g_{\boldsymbol{l}} := \sum_{j \in I^{\top}_{\boldsymbol{n},m}(\boldsymbol{l})} f_j\, \tilde{\psi}\left(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l}\right).$$

2: For $\boldsymbol{k} \in I_{\boldsymbol{N}}$ compute by d-variate (backward) FFT

$$\hat{g}_{\boldsymbol{k}} := \sum_{\boldsymbol{l} \in I_{\boldsymbol{n}}} g_{\boldsymbol{l}}\, \mathrm{e}^{+2\pi \mathrm{i} \boldsymbol{k}\left(\boldsymbol{n}^{-1} \odot \boldsymbol{l}\right)}.$$

3: For $\boldsymbol{k} \in I_{\boldsymbol{N}}$ compute

$$\hat{h}_{\boldsymbol{k}} := \frac{\hat{g}_{\boldsymbol{k}}}{|I_{\boldsymbol{n}}|\, c_{\boldsymbol{k}}(\tilde{\varphi})}.$$

Output: approximate values $\hat{h}_{\boldsymbol{k}}$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$.
Complexity: $\mathcal{O}(|\boldsymbol{N}| \log |\boldsymbol{N}| + M)$.

Algorithm 2: $\texttt{NFFT}^{\mathsf{H}}$

elements of the matrix $\boldsymbol{B}$, i.e.,

$$\bigcup_{j=0}^{M-1} j \times I_{\boldsymbol{n},m}\left(\boldsymbol{x}_j\right) = \bigcup_{\boldsymbol{l} \in I_{\boldsymbol{n}}} I^{\top}_{\boldsymbol{n},m}\left(\boldsymbol{l}\right) \times \boldsymbol{l}.$$

the multiplication with the sparse matrix $\boldsymbol{B}^{\top}$ is implemented in a 'transposed' way in the library, summation as outer loop and only using the multi-index sets $I_{\boldsymbol{n},m}\left(\boldsymbol{x}_j\right)$.

2.3 Available window functions and evaluation techniques

Again, only the case $d = 1$ is presented. To keep the aliasing error and the truncation error small, several functions φ with good localisation in time and frequency domain were proposed,

e.g. the (dilated) *Gaussian* [**?**, **?**, **?**]

$$\begin{aligned} \varphi(x) &= (\pi b)^{-1/2}\, \mathrm{e}^{-\frac{(nx)^2}{b}} \qquad \left(b := \frac{2\sigma}{2\sigma-1}\frac{m}{\pi}\right), \\ \hat{\varphi}(k) &= \frac{1}{n}\,\mathrm{e}^{-b\left(\frac{\pi k}{n}\right)^2}, \end{aligned} \tag{2.9}$$

(dilated) *cardinal central B–splines* [**?**, **?**]

$$\begin{aligned} \varphi(x) &= M_{2m}(nx), \\ \hat{\varphi}(k) &= \frac{1}{n}\mathrm{sinc}^{2m}(k\pi/n), \end{aligned} \tag{2.10}$$

where M_{2m} denotes the centred cardinal B–Spline of order $2m$,
(dilated) *Sinc functions* [**?**]

$$\begin{aligned} \varphi(x) &= \frac{N(2\sigma-1)}{2m}\mathrm{sinc}^{2m}\left(\frac{(\pi N x(2\sigma-1))}{2m}\right), \\ \hat{\varphi}(k) &= M_{2m}\left(\frac{2mk}{(2\sigma-1)N}\right) \end{aligned} \tag{2.11}$$

and (dilated) *Kaiser–Bessel functions* [**?**, **?**]

$$\begin{aligned} \varphi(x) &= \frac{1}{\pi}\begin{cases} \dfrac{\sinh\left(b\sqrt{m^2-n^2x^2}\right)}{\sqrt{m^2-n^2x^2}} & \text{for } |x| \le \frac{m}{n} \qquad \left(b := \pi\left(2-\frac{1}{\sigma}\right)\right), \\ \dfrac{\sin\left(b\sqrt{n^2x^2-m^2}\right)}{\sqrt{n^2x^2-m^2}} & \text{otherwise,} \end{cases} \\ \hat{\varphi}(k) &= \frac{1}{n}\begin{cases} I_0\left(m\sqrt{b^2-(2\pi k/n)^2}\right) & \text{for } k = -n\left(1-\frac{1}{2\sigma}\right),\ldots,n\left(1-\frac{1}{2\sigma}\right), \\ 0 & \text{otherwise,} \end{cases} \end{aligned} \tag{2.12}$$

where I_0 denotes the *modified zero–order Bessel function.* For these functions φ it has been proven that

$$|f(\boldsymbol{x}_j) - s(\boldsymbol{x}_j)| \le C(\sigma, m)\,\|\hat{\boldsymbol{f}}\|_1$$

where

$$C(\sigma,m) := \begin{cases} 4\,\mathrm{e}^{-m\pi(1-1/(2\sigma-1))} & \text{for } (\mathbf{??}), \\ 4\left(\frac{1}{2\sigma-1}\right)^{2m} & \text{for } (\mathbf{??}), \\ \frac{1}{m-1}\left(\frac{2}{\sigma^{2m}} + \left(\frac{\sigma}{2\sigma-1}\right)^{2m}\right) & \text{for } (\mathbf{??}), \\ 4\pi\left(\sqrt{m}+m\right)\sqrt[4]{1-\frac{1}{\sigma}}\,\mathrm{e}^{-2\pi m\sqrt{1-1/\sigma}} & \text{for } (\mathbf{??}). \end{cases}$$

Thus, for fixed $\sigma > 1$, the approximation error introduced by the NFFT decays exponentially with the number m of summands in (**??**). Using the tensor product approach the above error estimates can be generalised for the multivariate setting [**?**]. On the other hand, the complexity of the NFFT increases with m.

In the following, we suggest different methods for the compressed storage and application of the matrix $\boldsymbol{B}$ which are all available within our NFFT library by choosing particular flags in a simple way during the initialisation phase. These methods do not yield a different asymptotic performance but rather yield a lower constant in the amount of computation.

Fully precomputed window function

One possibility is to precompute all values $\varphi(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l})$ for $j = 0, \ldots, M-1$ and $\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)$ explicitly. Thus, one has to store the large amount of $(2m+1)^d M$ real numbers but uses no extra floating point operations during the matrix vector multiplication beside the necessary $(2m+1)^d M$ flops. Furthermore, we store for this method explicitly the row and column for each nonzero entry of the matrix $\boldsymbol{B}$. This method, included by the flag `PRE_FULL_PSI`, is the fastest procedure but can only be used if enough main memory is available.

Tensor product based precomputation

Using the fact that the window functions are built as tensor products one can store $\varphi_t((\boldsymbol{x}_j)_t - \frac{l_t}{n_t})$ for $j = 0, \ldots, M-1$, $t = 0, \ldots, d-1$, and $l_t \in I_{n_t,m}((\boldsymbol{x}_j)_t)$ where $(\boldsymbol{x}_j)_t$ denotes the t-th component of the j-th node. This method uses a medium amount of memory to store $d(2m+1)M$ real numbers in total. However, one has to carry out for each node at most $2(2m+1)^d$ extra multiplications to compute from the factors the multivariate window function $\varphi(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l})$ for $\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)$. Note, that this technique is available for every window function discussed here and can be used by means of the flag `PRE_PSI` which is also the default method within our software library.

Linear interpolation from a lookup table

For a large number of nodes M, the amount of memory can by further reduced by the use of lookup table techniques. For a recent example within the framework of gridding see [**?**]. We suggest to precompute from the even window function the equidistant samples $\varphi_t(\frac{rm}{Kn_t})$ for $t = 0, \ldots, d-1$ and $r = 0, \ldots, K$, $K \in \mathbb{N}$ and then compute for the actual node $\boldsymbol{x}_j$ during the NFFT the values $\varphi_t((\boldsymbol{x}_j)_t - \frac{l_t}{n_t})$ for $t = 0, \ldots, d-1$ and $l_t \in I_{n_t,m}((\boldsymbol{x}_j)_t)$ by means of the linear interpolation from its two neighbouring precomputed samples.

This method needs only a storage of dK real numbers in total where K depends solely on the target accuracy but neither on the number of nodes M nor on the multidegree $\boldsymbol{N}$. Choosing K to be a multiple of m, we further reduce the computational costs during the interpolation since the distance from $(\boldsymbol{x}_j)_t - \frac{l_t}{n_t}$ to the two neighbouring interpolation nodes and hence the interpolation weights remain the same for all $l_t \in I_{n_t,m}((\boldsymbol{x}_j)_t)$. This method requires $2(2m+1)^d$ extra multiplications per node and is used within the NFFT by the flag `PRE_LIN_PSI`. Error estimates for this approximation are given in [**?**].

Fast Gaussian gridding

Two useful properties of the Gaussian window function (**??**) within the present framework were recently reviewed in [**?**]. Beside its tensor product structure for $d > 1$, which also holds for all other window functions, it is remarkable that the number of evaluations of the form `exp()` can be greatly decreased. More precisely, for $d = 1$ and a fixed node x_j the evaluations of $\varphi(x_j - \frac{l'}{n})$, $l' \in I_{n,m}(x_j)$, can be reduced by the splitting

$$\sqrt{\pi b}\varphi\left(x_j - \frac{l'}{n}\right) = \mathrm{e}^{-\frac{(nx_j - l')^2}{b}} = \mathrm{e}^{-\frac{(nx_j - u)^2}{b}} \left(\mathrm{e}^{-\frac{2(nx_j - u)}{b}}\right)^l \mathrm{e}^{-\frac{l^2}{b}}.$$

where $u = \min I_{n,m}(x_j)$ and $l = 0, \ldots, 2m$. Note, that the first factor and the exponential within the brackets are constant for each fixed node x_j. Once, we evaluate the second expo-

nential, its l-th power can be computed consecutively by multiplications only. Furthermore, the last exponential is independent of x_j and these $2m+1$ values are computed only once within the NFFT and their amount is negligible. Thus, it is sufficient to store or evaluate $2M$ exponentials for $d = 1$. The case $d > 1$ uses $2dM$ storages or evaluations by using the general tensor product structure. This method is employed by the flags `FG_PSI` and `PRE_FG_PSI` for the evaluation or storage of $2d$ exponentials per node, respectively.

No precomputation of the window function

The last considered method uses no precomputation at all, but rather evaluates the univariate window function $(2m+1)^d M$ times. Thus, the computational time depends on how fast we can evaluate the particular window function. However, no additional storage is necessary which suits this approach whenever the problem size reaches the memory limits of the used computer.

2.4 Further NFFT approaches

Several papers have described fast approximations for the NFFT. Common names for NFFT are *non-uniform fast Fourier transform* [**?**], *generalised fast Fourier transform* [**?**], *unequally-spaced fast Fourier transform* [**?**], *fast approximate Fourier transforms for irregularly spaced data* [**?**], *non-equispaced fast Fourier transform* [**?**] or *gridding* [**?**, **?**, **?**].

In various papers, different window functions were considered, e.g. Gaussian pulse tapered with a Hanning window in [**?**], Gaussian kernels combined with Sinc kernels in [**?**], and special optimised windows in [**?**, **?**].

A simple but nevertheless fast scheme for the computation of (**??**) in the univariate case $d = 1$ is presented in [**?**]. This approach uses for each node $x_j \in [-\frac{1}{2}, \frac{1}{2})$ a m-th order Taylor expansion of the trigonometric polynomial in (**??**) about the nearest neighbouring point on the oversampled equispaced lattice $\{n^{-1}k - \frac{1}{2}\}_{k=0,\dots,n-1}$ where again $n = \sigma N$, $\sigma \gg 1$. Besides its simple structure and only $\mathcal{O}(N \log N + M)$ arithmetic operations, this algorithm utilises m FFTs of size n compared to only one in the NFFT approach, uses a medium amount of extra memory, and is not suited for highly accurate computations, see [**?**]. Furthermore, its extension to higher dimensions has not been considered so far.

Another approach for the univariate case $d = 1$ is considered in [**?**] and based on a Lagrange interpolation technique. After taking a N-point FFT of the vector $\hat{\boldsymbol{f}}$ in (**??**) one uses an exact polynomial interpolation scheme to obtain the values of the trigonometric polynomial f at the nonequispaced nodes x_j. Here, the time consuming part is the exact polynomial interpolation scheme which can however be realised fast in an approximate way by means of the fast multipole method. This approach is appealing since it allows also for the inverse transform. Nevertheless, numerical experiments in [**?**] showed that this approach is far more time consuming than Algorithm **??** and the inversion can only be computed in a stable way for almost equispaced nodes [**?**].

Furthermore, special approaches based on scaling vectors [**?**], based on minimising the Frobenius norm of certain error matrices [**?**] or based on min-max interpolation [**?**] are proposed. While these approaches gain some accuracy for the Gaussian or B-Spline windows, no reasonable improvement is obtained for the Kaiser-Bessel window function.

For comparison of different approaches, we refer to [**?**, **?**, **?**, **?**].

3 Generalisations and inversion

We consider generalisations of the NFFT for

3.1 NNFFT - nonequispaced in time and frequency fast Fourier transform

Now we are interested in the computation of

$$f_j = \sum_{k=0}^{N-1} \hat{f}_k \mathrm{e}^{-2\pi \mathrm{i}(\boldsymbol{v}_k \odot \boldsymbol{N}))\boldsymbol{x}_j}$$

for $j = 0, \ldots, M-1$, $\boldsymbol{v}_k, \boldsymbol{x}_j \in \mathbb{T}^d$, and $\hat{f}_k \in \mathbb{C}$, where $\boldsymbol{N} \in \mathbb{N}^d$ denotes the "nonharmonic" bandwidth. The corresponding fast algorithm is known as *fast Fourier transform for nonequispaced data in space* **and** *frequency domain* (NNFFT) [**?**, **?**, **?**] or as type 3 nonuniform FFT [**?**].

A straightforward evaluation of this sum with the standard NFFT is not possible, since the samples in neither domain are equispaced. However, an so-called NNFFT was first suggested in [**?**] and later studied in more depth in [**?**], which permits the fast calculation of the Fourier transform of a vector of nonequispaced samples at a vector of nonequispaced positions. It constitutes a combination of the standard NFFT and its adjoint see also [**?**].

3.2 NFCT/NFST - nonequispaced fast (co)sine transform

Let nonequispaced nodes $\boldsymbol{x}_j \in [0, \frac{1}{2}]^d$ and frequencies $\boldsymbol{k}$ in the index sets

$$I_{\boldsymbol{N}}^C := \left\{ \boldsymbol{k} = (k_t)_{t=0,\ldots,d-1} \in \mathbb{Z}^d : 0 \le k_t < N_t,\ t = 0, \ldots, d-1 \right\},$$
$$I_{\boldsymbol{N}}^S := \left\{ \boldsymbol{k} = (k_t)_{t=0,\ldots,d-1} \in \mathbb{Z}^d : 1 \le k_t < N_t,\ t = 0, \ldots, d-1 \right\}$$

be given. For notational convenience let furthermore $\cos(\boldsymbol{k} \odot \boldsymbol{x}) := \cos(k_0 x_0) \cdot \ldots \cdot \cos(k_{d-1} x_{d-1})$ and $\sin(\boldsymbol{k} \odot \boldsymbol{x}) := \sin(k_0 x_0) \cdot \ldots \cdot \sin(k_{d-1} x_{d-1})$.

The *nonequispaced discrete cosine* and *sine transforms* are given by

$$f(\boldsymbol{x}_j) = \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}^C} \hat{f}_{\boldsymbol{k}} \cos(2\pi(\boldsymbol{k} \odot \boldsymbol{x}_j)),$$
$$f(\boldsymbol{x}_j) = \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}^S} \hat{f}_{\boldsymbol{k}} \sin(2\pi(\boldsymbol{k} \odot \boldsymbol{x}_j)),$$

for $j = 0, \ldots, M-1$ and real coefficients $\hat{f}_{\boldsymbol{k}} \in \mathbb{R}$, respectively.

The straight forward algorithm of this matrix vector product, which is called ndct and ndst, takes $\mathcal{O}(M|I_{\boldsymbol{N}}^C|)$ and $\mathcal{O}(M|I_{\boldsymbol{N}}^C|)$ arithmetical operations. For these real transforms the adjoint transforms coincide with the ordinary transposed matrix vector products. Our fast approach is based on the NFFT and seems to be easier than the Chebyshev transform based derivation in [**?**] and faster than the algorithms in [**?**] which still use FFTs. Instead of FFTs we use fast algorithms for the discrete cosine transform (DCT–I) and for the discrete sine transform (DST–I). For details we refer to [**?**].

3.3 NSFFT - nonequispaced sparse fast Fourier transform

We consider the fast evaluation of trigonometric polynomials from so-called hyperbolic crosses. In multivariate approximation one has to deal with the so called 'curse of dimensionality', i.e., the number of degrees of freedom for representing an approximation of a function with a prescribed accuracy depends exponentially on the dimensionality of the considered problem. This obstacle can be circumvented to some extend by the interpolation on sparse grids and the related approximation on hyperbolic cross points in the Fourier domain, see, e.g., [**?**, **?**, **?**].

Instead of approximating a Fourier series on the standard tensor product grid $I_{(\boldsymbol{N},\ldots,\boldsymbol{N})}$ with $\mathcal{O}(N^d)$ degrees of freedom, it can be approximated with only $\mathcal{O}(N \log^{d-1} N)$ degrees of freedom from the hyperbolic cross

$$H_N^d := \bigcup_{\boldsymbol{N}\in\mathbb{N}^d,\, |I_{\boldsymbol{N}}|=N} I_{\boldsymbol{N}},$$

where $N = 2^{J+2},\ J \in \mathbb{N}_0$ and we allow $N_t = 1$ in the t-th coordinate in the definition of $I_{\boldsymbol{N}}$. The approximation error in a suitable norm (dominated mixed smoothness) can be shown to deteriorate only by a factor of $\log^{d-1} N$, cf. [**?**].

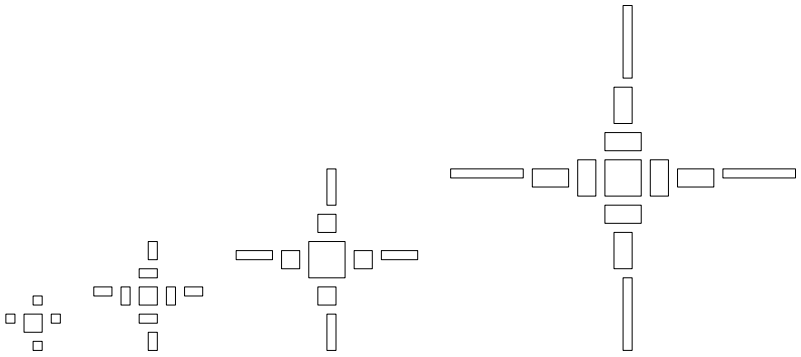


Figure 3.1: Hyperbolic cross points for $d = 2$ and $J = 0, \ldots, 3$.

The *nonequispaced sparse discrete Fourier transform* (NSDFT) is the evaluation of

$$f(\boldsymbol{x}_j) = \sum_{\boldsymbol{k}\in H_N^d} \hat{f}_{\boldsymbol{k}} \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}_j}$$

for given Fourier coefficients $\hat{f}_{\boldsymbol{k}} \in \mathbb{C}$ and nodes $\boldsymbol{x}_j \in \mathbb{T}^d$. The number of used arithmetical operations is $\mathcal{O}(MN \log^{d-1} N)$. This is reduced by our fast schemes to $\mathcal{O}(N \log^2 N + M)$ for $d = 2$ and $\mathcal{O}(N^{3/2} \log N + M)$ for $d = 3$, see [**?**] for details.

3.4 FPT - fast polynomial transform

A *discrete polynomial transform* (DPT) is a generalisation of the DFT from the basis of complex exponentials $\mathrm{e}^{\mathrm{i}kx}$ to an arbitrary systems of algebraic polynomials satisfying a three-term recurrence relation. More precisely, let $P_0, P_1, \ldots : [-1, 1] \to \mathbb{R}$ be a sequence of polynomials that satisfies a three-term recurrence relation

$$P_{k+1}(x) = (\alpha_k x + \beta_k) P_k(x) + \gamma_k P_{k-1}(x),$$

with $\alpha_k \neq 0$, $k \geq 0$, $P_0(x) := 1$, and $P_{-1}(x) := 0$. Clearly, every P_k is a polynomial of exact degree k and typical examples are the classical orthogonal Jacobi polynomials $P_k^{(\alpha,\beta)}$.

Now, let $f : [-1, 1] \to \mathbb{R}$ be a polynomial of degree $N \in \mathbb{N}$ given by the finite linear combination

$$f(x) = \sum_{k=0}^{N} a_k P_k(x).$$

The discrete polynomial transform (DPT) and its fast version, the *fast polynomial transform* (FPT), are the transformation of the coefficients a_k into coefficients b_k from the orthogonal expansion of f into the basis of Chebyshev polynomials of the first kind $T_k(x) := \cos(k \arccos x)$, i.e.,

$$f(x) = \sum_{k=0}^{N} b_k T_k(x).$$

A direct algorithm for computing this transformation needs $\mathcal{O}(N^2)$ arithmetic operations. The FPT algorithm implemented in the NFFT library follows the approach in [**?**] and is based on the idea of using the three-term-recurrence relation repeatedly. Together with a method for fast polynomial multiplication in the Chebyshev basis and a cascade-like summation process, this yields a method for computing the polynomial transform with $\mathcal{O}(N \log^2 N)$ arithmetic operations. For more detailed information, we refer the reader to [**?**, **?**, **?**, **?**, **?**] and the references therein.

3.5 NFSFT - nonequispaced fast spherical Fourier transform

Fourier analysis on the sphere has, despite other fields, practical relevance in tomography, geophysics, seismology, meteorology and crystallography. In analogy to the complex exponentials $\mathrm{e}^{\mathrm{i}kx}$ on the torus, the spherical harmonics form the orthogonal Fourier basis with respect to the usual inner product on the sphere.

Spherical coordinates

Every point in $\mathbb{R}^3$ can be described in spherical coordinates by a vector $(r, \vartheta, \varphi)^\top$ with the radius $r \geq 0$ and two angles $\vartheta \in [0, \pi]$, $\varphi \in [0, 2\pi)$. We denote by $\mathbb{S}^2$ the two-dimensional unit sphere embedded into $\mathbb{R}^3$, i.e.

$$\mathbb{S}^2 := \left\{ \boldsymbol{x} \in \mathbb{R}^3 : \ \|\boldsymbol{x}\|_2 = 1 \right\}$$

and identify a point from $\mathbb{S}^2$ with the corresponding vector $(\vartheta, \varphi)^\top$. The spherical coordinate system is illustrated in Figure **??**.

Legendre polynomials and associated Legendre functions

The Legendre polynomials $P_k : [-1, 1] \to \mathbb{R}$, $k \geq 0$, as classical orthogonal polynomials are given by their corresponding Rodrigues formula

$$P_k(x) := \frac{1}{2^k k!} \frac{\mathrm{d}^k}{\mathrm{d}x^k} \left(x^2 - 1\right)^k.$$

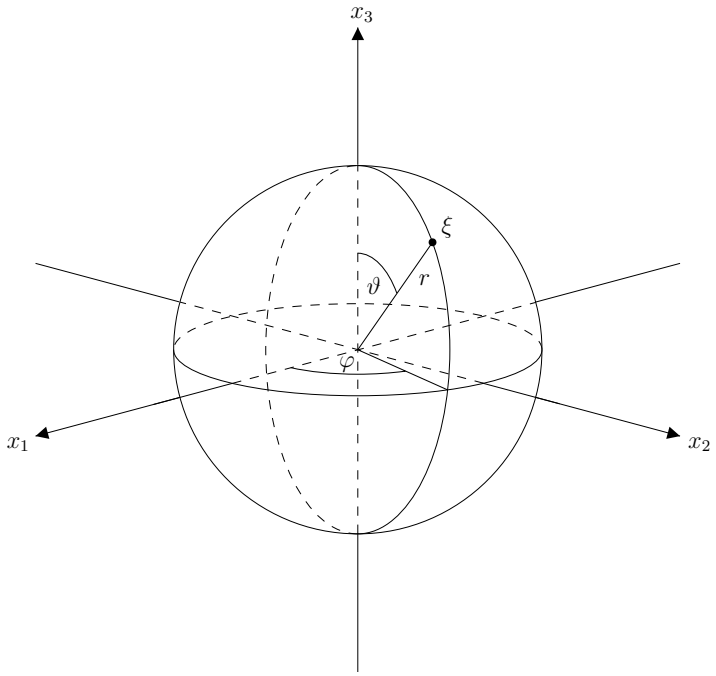


Figure 3.2: The spherical coordinate system in $\mathbb{R}^3$: Every point $\boldsymbol{\xi}$ on a sphere with radius r centred at the origin can be described by angles $\vartheta \in [0, \pi]$, $\varphi \in [0, 2\pi)$ and the radius $r \in \mathbb{R}^+$. For $\vartheta = 0$ or $\vartheta = \pi$ the point $\boldsymbol{\xi}$ coincides with the North or the South pole, respectively.

The associated Legendre functions $P_k^n : [-1, 1] \to \mathbb{R}$, $k \geq n \geq 0$ are defined by

$$P_k^n(x) := \left(\frac{(k-n)!}{(k+n)!}\right)^{1/2} \left(1 - x^2\right)^{n/2} \frac{\mathrm{d}^n}{\mathrm{d}x^n} P_k(x).$$

For $n = 0$, they coincide with the Legendre polynomials $P_k = P_k^0$. The associated Legendre functions P_k^n obey the three-term recurrence relation

$$P_{k+1}^n(x) = \frac{2k+1}{((k-n+1)(k+n+1))^{1/2}} x P_k^n(x) - \frac{((k-n)(k+n))^{1/2}}{((k-n+1)(k+n+1))^{1/2}} P_{k-1}^n(x)$$

for $k \geq n \geq 0$, $P_{n-1}^n(x) = 0$, $P_n^n(x) = \frac{\sqrt{(2n)!}}{2^n n!} \left(1 - x^2\right)^{n/2}$. For fixed n, the set $\{P_k^n : k \geq n\}$ forms a set of orthogonal functions, i.e.,

$$\langle P_k^n, P_l^n \rangle = \int_{-1}^{1} P_k^n(x) P_l^n(x) \mathrm{d}x = \frac{2}{2k+1} \delta_{k,l}.$$

Again, we denote by $\bar{P}_k^n = \sqrt{\frac{2k+1}{2}} P_k^n$ the orthonormal associated Legendre functions. In the following, we allow also for $n < 0$ and set $P_k^n := P_k^{-n}$ in this case.

Spherical harmonics

The spherical harmonics $Y_k^n : \mathbb{S}^2 \to \mathbb{C}$, $k \geq |n|$, $n \in \mathbb{Z}$, are given by

$$Y_k^n(\vartheta, \varphi) := P_k^n(\cos \vartheta) \, \mathrm{e}^{\mathrm{i}n\varphi}.$$

They form an orthogonal basis for the space of square integrable functions on the sphere, i.e.,

$$\langle Y_k^n, Y_l^m \rangle = \int_0^{2\pi} \int_0^{\pi} Y_k^n(\vartheta, \varphi) \overline{Y_l^m(\vartheta, \varphi)} \sin \vartheta \, \mathrm{d}\vartheta \, \mathrm{d}\varphi = \frac{4\pi}{2k+1} \delta_{k,l} \delta_{n,m}.$$

The orthonormal spherical harmonics are denoted by $\bar{Y}_k^n = \sqrt{\frac{2k+1}{4\pi}} Y_k^n$.

Hence, any square integrable function $f : \mathbb{S}^2 \to \mathbb{C}$ has the expansion

$$f = \sum_{k=0}^{\infty} \sum_{n=-k}^{k} \hat{f}(k,n) \bar{Y}_k^n,$$

with the spherical Fourier coefficients $\hat{f}(k,n) = \left\langle f, \bar{Y}_k^n \right\rangle$. The function f is called *bandlimited*, if $\hat{f}(k,n) = 0$ for $k > N$ and some $N \in \mathbb{N}$. In analogy to the NFFT on the Torus $\mathbb{T}$, we define the sampling set

$$\mathcal{X} := \left\{ (\vartheta_j, \varphi_j) \in \mathbb{S}^2 : \, j = 0, \ldots, M-1 \right\}$$

of nodes on the sphere $\mathbb{S}^2$ and the set of possible "frequencies"

$$\mathcal{I}_N := \left\{ (k,n) : \, k = 0, 1, \ldots, N; \; n = -k, \ldots, k \right\}.$$

The *nonequispaced discrete spherical Fourier transform* (NDSFT) is defined as the evaluation of the finite spherical Fourier sum

$$f_j = f\left(\vartheta_j, \varphi_j\right) = \sum_{(k,n) \in \mathcal{I}_N} \hat{f}_k^n \, Y_k^n\left(\vartheta_j, \varphi_j\right) = \sum_{k=0}^{N} \sum_{n=-k}^{k} \hat{f}_k^n \, Y_k^n\left(\vartheta_j, \varphi_j\right) \tag{3.1}$$

for $j = 0, \ldots, M-1$. In matrix vector notation, this reads $\boldsymbol{f} = \boldsymbol{Y} \, \hat{\boldsymbol{f}}$ with

$$\begin{aligned} \boldsymbol{f} &:= (f_j)_{j=0}^{M-1} \in \mathbb{C}^M, \;\; f_j := f\left(\vartheta_j, \varphi_j\right), \\ \boldsymbol{Y} &:= \left(Y_k^n\left(\vartheta_j, \varphi_j\right)\right)_{j=0,\ldots,M-1;\; (k,n) \in \mathcal{I}_N} \in \mathbb{C}^{M \times (N+1)^2}, \\ \hat{\boldsymbol{f}} &:= \left(\hat{f}_k^n\right)_{(k,n) \in \mathcal{I}_N} \in \mathbb{C}^{(N+1)^2}. \end{aligned}$$

The corresponding *adjoint nonequispaced discrete fast spherical Fourier transform* (adjoint NDSFT) is defined as the evaluation of

$$\hat{h}_k^n = \sum_{j=0}^{M-1} f\left(\vartheta_j, \varphi_j\right) \overline{Y_k^n\left(\vartheta_j, \varphi_j\right)}$$

for all $(k,n) \in \mathcal{I}_N$. Again, in matrix vector notation, this reads $\hat{\boldsymbol{h}} = \boldsymbol{Y}^{\mathsf{H}} \boldsymbol{f}$.

The coefficients $\hat{h}_k^n$ are, in general, not identical to the Fourier coefficients $\hat{f}(k,n)$ of the function f. However, provided that for the sampling set $\mathcal{X}$ a quadrature rule with weights w_j, $j = 0, \ldots, M-1$, and sufficient degree of exactness is available, one might recover the Fourier coefficients $\hat{f}(k,n)$ by evaluating

$$\hat{f}_k^n = \int_0^{2\pi} \int_0^{\pi} f(\vartheta, \varphi) \overline{Y_k^n(\vartheta, \varphi)} \sin \vartheta \, \mathrm{d}\vartheta \, \mathrm{d}\varphi = \sum_{j=0}^{M-1} w_j f\left(\vartheta_j, \varphi_j\right) \overline{Y_k^n\left(\vartheta_j, \varphi_j\right)}$$

Input: $N \in \mathbb{N}_0$, $M \in \mathbb{N}$, spherical Fourier coefficients $\hat{\boldsymbol{f}} = (\hat{f}_k^n)_{(k,n)\in\mathcal{I}_N} \in \mathbb{C}^{(N+1)^2}$,
a sampling set $\mathcal{X} = (\vartheta_j, \varphi_j)_{j=0}^{M-1} \in ([0,\pi] \times [0,2\pi))^M$.

for $n = -N, \ldots, N$ **do**
Compute the Chebyshev coefficients $(b_k^n)_{k=0}^N$ of g^n by a fast polynomial transform.
Compute the coefficients $(c_k^n)_{k=-N}^N$ from the coefficients $(b_k^n)_{k=0}^M$.
end for
Compute the function values $(f(\vartheta_j, \varphi_j))_{j=0}^{M-1}$ by evaluating the Fourier sum using a fast two-dimensional NFFT.

Output: The function values $\boldsymbol{f} = (f(\vartheta_j, \varphi_j))_{j=0}^{M-1} \in \mathbb{C}^M$.
Complexity: $\mathcal{O}\left(N^2 \log^2 N + M\right)$.

Algorithm 3: Nonequispaced fast spherical Fourier transform (NFSFT)

Input: $N \in \mathbb{N}_0$, $M \in \mathbb{N}$, a sampling set $\mathcal{X} = (\vartheta_j, \varphi_j)_{j=0}^{M-1} \in ([0,\pi] \times [0,2\pi))^M$,
values $\tilde{\boldsymbol{f}} = (\tilde{f}_j)_{j=0}^{M-1} \in \mathbb{C}^M$.

Compute the coefficients $(\tilde{c}_k^n)_{k,n=-N}^N$ from the values $(f_j)_{j=0}^{N-1}$.
for $n = -N, \ldots, N$ **do**
Compute the coefficients $\left(\tilde{b}_k^n\right)_{k=0}^N$ from the coefficients $(\tilde{c}_k^n)_{k=-N}^N$.
Compute the coefficients $(\tilde{a}_k^n)_{k=|n|}^N$ from the coefficients $\left(\tilde{b}_k^n\right)_{k=0}^N$ by a fast transposed polynomial transform.
end for

Output: Coefficients $\boldsymbol{h} = \left(\tilde{h}_k^n\right)_{(k,n)\in\mathcal{I}_N} \in \mathbb{C}^{(N+1)^2}$.
Complexity: $\mathcal{O}\left(N^2 \log^2 N + M\right)$.

Algorithm 4: Adjoint nonequispaced fast spherical Fourier transform (adjoint NFSFT)

for all $(k,n) \in \mathcal{I}_N$.

Direct algorithms for computing the NDSFT and adjoint NDSFT transformations need $\mathcal{O}(MN^2)$ arithmetic operations. A combination of the fast polynomial transform and the NFFT leads to approximate algorithms with $\mathcal{O}(N^2 \log^2 N + M)$ arithmetic operations. These are denoted *NFSFT* and *adjoint NFSFT*, respectively. The NFSFT algorithm using the FPT and the NFFT was introduced in [**?**] while the adjoint NFSFT variant was developed in [**?**].

3.6 Solver - inverse transforms

In the following, we describe the inversion of the NFFT, i.e., the computation of Fourier coefficients from given samples $(\boldsymbol{x}_j, y_j) \in \mathbb{T}^d \times \mathbb{C}$, $j = 0, \ldots, M-1$. In matrix vector notation, we aim to solve the linear system of equations

$$\boldsymbol{A}\hat{\boldsymbol{f}} \approx \boldsymbol{y} \tag{3.2}$$

for the vector $\hat{\boldsymbol{f}} \in \mathbb{C}^{|I_{\boldsymbol{N}}|}$. Note however that the nonequispaced Fourier matrix $\boldsymbol{A}$ can be replaced by any other Fourier matrix from Section **??**.

Typically, the number of samples M and the dimension of the space of the polynomials $|I_{\boldsymbol{N}}|$ do not coincide, i.e., the matrix $\boldsymbol{A}$ is rectangular. There are no simple inverses to nonequispaced Fourier matrices in general and we search for some pseudoinverse solution $\hat{\boldsymbol{f}}^{\dagger}$, see e.g. [**?**, p. 15]. However, we conclude from eigenvalue estimates in [**?**, **?**, **?**] that the matrix $\boldsymbol{A}$ has full rank if

$$\max_{0\le t<d} N_t < c_d \delta^{-1}, \qquad \delta := 2 \max_{\boldsymbol{x}\in\mathbb{T}^d} \min_{j=0,\dots,M-1} \operatorname{dist}_\infty(\boldsymbol{x}_j, \boldsymbol{x}_l),$$

or

$$\min_{0\le t<d} N_t > C_d q^{-1}, \qquad q := \min_{0\le j<l<M} \operatorname{dist}_\infty(\boldsymbol{x}_j, \boldsymbol{x}_l).$$

In what follows, we consider a weighted least squares solution for the over-determined case and an optimal interpolation for the consistent underdetermined case. Both cases are solved by means of iterative algorithms where Algorithm **??** and **??** are used for the matrix vector multiplication with $\boldsymbol{A}$ and $\boldsymbol{A}^{\mathsf{H}}$, respectively.

Least squares

For a comparative low polynomial degree $|I_{\boldsymbol{N}}| < M$ the system (**??**) is over-determined, so that in general the given data $y_j \in \mathbb{C},\ j = 0, \dots, M-1$, will be only approximated up to the *residual* $\boldsymbol{r} := \boldsymbol{f} - \boldsymbol{A}\hat{\boldsymbol{f}}$. One considers the *weighted approximation problem*

$$\|\boldsymbol{y} - \boldsymbol{A}\hat{\boldsymbol{f}}\|_{\boldsymbol{W}} = \left(\sum_{j=0}^{M-1} w_j |y_j - f(x_j)|^2 \right)^{1/2} \xrightarrow{\hat{\boldsymbol{f}}} \min,$$

which may incorporate weights $w_j > 0$, $\boldsymbol{W} := \operatorname{diag}(w_j)_{j=0,\dots,M-1}$, to compensate for clusters in the sampling set $\mathcal{X}$. This least squares problem is equivalent to the *weighted normal equation of first kind*

$$\boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{A}\hat{\boldsymbol{f}} = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{y}. \tag{3.3}$$

Applying the Landweber (also known as Richardson, ...), the steepest descent, or the conjugate gradient scheme to (**??**) yields the following Algorithms **??**-**??**.

Interpolation

For a comparative high polynomial degree $|I_{\boldsymbol{N}}| > M$ one expects to interpolate the given data $y_j \in \mathbb{C},\ j = 0, \dots, M-1$, exactly. The (consistent) linear system (**??**) is under-determined. One considers the *damped minimisation problem*

$$\|\hat{\boldsymbol{f}}\|_{\hat{\boldsymbol{W}}}^{-1} = \left(\sum_{\boldsymbol{k}\in I_{\boldsymbol{N}}} \frac{|\hat{f}_{\boldsymbol{k}}|^2}{\hat{w}_{\boldsymbol{k}}} \right)^{1/2} \xrightarrow{\hat{\boldsymbol{f}}} \min \quad \text{subject to} \quad \boldsymbol{A}\hat{\boldsymbol{f}} = \boldsymbol{y},$$

which may incorporate 'damping factors' $\hat{w}_{\boldsymbol{k}} > 0$, $\hat{\boldsymbol{W}} := \operatorname{diag}(\hat{w}_{\boldsymbol{k}})_{\boldsymbol{k}\in I_{\boldsymbol{N}}}$. A smooth solution is favoured, i.e., a decay of the Fourier coefficients $\hat{f}_{\boldsymbol{k}},\ \boldsymbol{k} \in I_{\boldsymbol{N}}$, for decaying damping factors $\hat{w}_{\boldsymbol{k}}$. This interpolation problem is equivalent to the *damped normal equation of second kind*

$$\boldsymbol{A}\hat{\boldsymbol{W}}\boldsymbol{A}^{\mathsf{H}}\tilde{\boldsymbol{f}} = \boldsymbol{y}, \qquad \hat{\boldsymbol{f}} = \hat{\boldsymbol{W}}\boldsymbol{A}^{\mathsf{H}}\tilde{\boldsymbol{f}}. \tag{3.4}$$

Applying the conjugate gradient scheme to (**??**) yields the following Algorithm **??**.

Input: $\boldsymbol{y} \in \mathbb{C}^M$, $\hat{\boldsymbol{f}}_0 \in \mathbb{C}^{|I_{\boldsymbol{N}}|}$, $\alpha > 0$

$\boldsymbol{r}_0 = \boldsymbol{y} - \boldsymbol{A}\hat{\boldsymbol{f}}_0$

$\hat{\boldsymbol{z}}_0 = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_0$

for $l = 0, \ldots$ **do**

  $\hat{\boldsymbol{f}}_{l+1} = \hat{\boldsymbol{f}}_l + \alpha\hat{\boldsymbol{W}}\hat{\boldsymbol{z}}_l$

  $\boldsymbol{r}_{l+1} = \boldsymbol{y} - \boldsymbol{A}\hat{\boldsymbol{f}}_{l+1}$

  $\hat{\boldsymbol{z}}_{l+1} = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_{l+1}$

end for

Output: $\hat{\boldsymbol{f}}_l$

Algorithm 5: Landweber

Input: $\boldsymbol{y} \in \mathbb{C}^M$, $\hat{\boldsymbol{f}}_0 \in \mathbb{C}^{|I_{\boldsymbol{N}}|}$

$\boldsymbol{r}_0 = \boldsymbol{y} - \boldsymbol{A}\hat{\boldsymbol{f}}_0$

$\hat{\boldsymbol{z}}_0 = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_0$

for $l = 0, \ldots$ **do**

  $\boldsymbol{v}_l = \boldsymbol{A}\hat{\boldsymbol{W}}\hat{\boldsymbol{z}}_l$

  $\alpha_l = \frac{\hat{\boldsymbol{z}}_l^{\mathsf{H}}\hat{\boldsymbol{W}}\hat{\boldsymbol{z}}_l}{\boldsymbol{v}_l^{\mathsf{H}}\boldsymbol{W}\boldsymbol{v}_l}$

  $\hat{\boldsymbol{f}}_{l+1} = \hat{\boldsymbol{f}}_l + \alpha_l\hat{\boldsymbol{W}}\hat{\boldsymbol{z}}_l$

  $\boldsymbol{r}_{l+1} = \boldsymbol{r}_{l+1} - \alpha_l\boldsymbol{v}_l$

  $\hat{\boldsymbol{z}}_{l+1} = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_{l+1}$

end for

Output: $\hat{\boldsymbol{f}}_l$

Algorithm 6: Steepest descent

Input: $\boldsymbol{y} \in \mathbb{C}^M$, $\hat{\boldsymbol{f}}_0 \in \mathbb{C}^{|I_{\boldsymbol{N}}|}$

$\boldsymbol{r}_0 = \boldsymbol{y} - \boldsymbol{A}\hat{\boldsymbol{f}}_0$

$\hat{\boldsymbol{z}}_0 = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_0$

$\hat{\boldsymbol{p}}_0 = \hat{\boldsymbol{z}}_0$

for $l = 0, \ldots$ **do**

  $\boldsymbol{v}_l = \boldsymbol{A}\hat{\boldsymbol{W}}\hat{\boldsymbol{p}}_l$

  $\alpha_l = \frac{\hat{\boldsymbol{z}}_l^{\mathsf{H}}\hat{\boldsymbol{W}}\hat{\boldsymbol{z}}_l}{\boldsymbol{v}_l^{\mathsf{H}}\boldsymbol{W}\boldsymbol{v}_l}$

  $\hat{\boldsymbol{f}}_{l+1} = \hat{\boldsymbol{f}}_l + \alpha_l\hat{\boldsymbol{W}}\hat{\boldsymbol{p}}_l$

  $\boldsymbol{r}_{l+1} = \boldsymbol{r}_l - \alpha_l\boldsymbol{v}_l$

  $\hat{\boldsymbol{z}}_{l+1} = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_{l+1}$

  $\beta_l = \frac{\hat{\boldsymbol{z}}_{l+1}^{\mathsf{H}}\hat{\boldsymbol{W}}\hat{\boldsymbol{z}}_{l+1}}{\hat{\boldsymbol{z}}_l^{\mathsf{H}}\hat{\boldsymbol{W}}\hat{\boldsymbol{z}}_l}$

  $\hat{\boldsymbol{p}}_{l+1} = \beta_l\hat{\boldsymbol{p}}_l + \hat{\boldsymbol{z}}_{l+1}$

end for

Output: $\hat{\boldsymbol{f}}_l$

Algorithm 7: Conjugate gradients for the normal equations, Residual minimisation (CGNR)

Input: $\boldsymbol{y} \in \mathbb{C}^M$, $\hat{\boldsymbol{f}}_0 \in \mathbb{C}^{|I_{\boldsymbol{N}}|}$

$\boldsymbol{r}_0 = \boldsymbol{y} - \boldsymbol{A}\hat{\boldsymbol{f}}_0$

$\hat{\boldsymbol{p}}_0 = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_0$

for $l = 0, \ldots$ **do**

  $\alpha_l = \frac{\boldsymbol{r}_l^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_l}{\hat{\boldsymbol{p}}_l^{\mathsf{H}}\hat{\boldsymbol{W}}\hat{\boldsymbol{p}}_l}$

  $\hat{\boldsymbol{f}}_{l+1} = \hat{\boldsymbol{f}}_l + \alpha_l\hat{\boldsymbol{W}}\hat{\boldsymbol{p}}_l$

  $\boldsymbol{r}_{l+1} = \boldsymbol{r}_l - \alpha_l\boldsymbol{A}\hat{\boldsymbol{W}}\hat{\boldsymbol{p}}_l$

  $\beta_l = \frac{\boldsymbol{r}_{l+1}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_{l+1}}{\boldsymbol{r}_l^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_l}$

  $\hat{\boldsymbol{p}}_{l+1} = \beta_l\hat{\boldsymbol{p}}_l + \boldsymbol{A}^{\mathsf{H}}\boldsymbol{W}\boldsymbol{r}_{l+1}$

end for

Output: $\hat{\boldsymbol{f}}_l$

Algorithm 8: Conjugate gradients for the normal equations, Error minimisation (CGNE)

4 Library

The library is completely written in C and uses the FFTW library [**?**] which has to be installed on your system. The library has several options (determined at compile time) and parameters (determined at run time).

4.1 Installation

Download from `www.tu-chemnitz.de/~potts/nfft` the most recent version `nfft3.x.x.tgz`. In the following, we assume that you use a `bash` compatible shell. Uncompress the archive.

- `tar xfvz nfft3.x.x.tar.gz`

Change to the newly created directory.

- `cd nfft3.x.x`

Optional: Set environment variables (see also below). Leave this step out, if you are unsure what to do. Example:

- `export CPPFLAGS="-I/path/to/include/files"`

- `export LDFLAGS="-L/path/to/libraries"`

Run the configure script.

- `./configure`

Compile the sources.

- `make`

This builds the NFFT library and all examples and applications. You can run `make install` to install the library on your system. If for example `/usr` is your default installation directory, `make install` will copy the NFFT 3 library to `/usr/lib`, the C header files to `/usr/include/nfft` and the documentation together with all examples and applications to `/usr/share/nfft`.

For more information on fine grained control over the installation directories, run the `configure` script with the option `--help`, i.e. `./configure --help`. For example, to see what gets installed, run `./configure --prefix=some/path/to/a/temporary/directory` in the line of commands above. After `make` and `make install`, you will find all installed files in the directory structure created under the temporary directory.

The following options can also be used in conjunction with the configure script: By default, NFFT 3 routines use the Kaiser-Bessel window function (see Appendix **??**). You can change the window function with the option `--with-window=ARG`, where `ARG` is replaced by `kaiserbessel` (Kaiser-Bessel), `gaussian` (Gaussian), `bspline` (B-spline), `sinc` (sinc power) or `delta` (Dirac delta) to yield the window function in parenthesis.

The NFFT transforms can be configured to measure elapsed time for each step of Algorithm **??** and **??**. This should help customise the library to one's needs. You can enable this behaviour with the options `--enable-measure-time` and/or `--enable-measure-time-fftw`.

4.2 Procedure for computing an NFFT

One has to follow certain steps to write a simple program using the NFFT library. A complete example is shown in Section **??**. The first argument of each function is a pointer to a application-owned variable of type `nfft_plan`. The aim of this structure is to keep interfaces small, it contains all parameters and data.

Initialisation

Initialisation of a plan is done by one of the `nfft_init`-functions. The simplest version for the univariate case $d = 1$ just specifies the number of Fourier coefficients N_0 and the number of nonequispaced nodes M. For an application-owned variable `nfft_plan my_plan` the function call is

```
nfft_init_1d(&my_plan,N0,M);
```

The first argument should be uninitialised. Memory allocation is completely done by the init routine.

Setting nodes

One has to define the nodes $\boldsymbol{x}_j \in \mathbb{T}^d$ for the transformation in the member variable `my_plan.x`. The t-th coordinate of the j-th node $\boldsymbol{x}_j$ is assigned by

```
my_plan.x[d*j+t]= /* your choice in [-0.5,0.5] */;
```

Precompute $\tilde{\psi}$

The precomputation of the values $\tilde{\psi}\left(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l}\right)$, i.e. the entries of the matrix $\boldsymbol{B}$, depends on the choice for `my_plan.x`. If one of the proposed precomputation strategies (`PRE_LIN_PSI`, `PRE_FG_PSI`, `PRE_PSI`, or `PRE_FULL_PSI`) is used, one has to call the appropriate precomputation routine **AFTER** setting the nodes. For simpler usage this has been summarised by

```
if(my_plan.nfft_flags & PRE_ONE_PSI) nfft_precompute_one_psi(&my_plan);
```

Note furthermore, that

1. `FG_PSI` and `PRE_FG_PSI` rely on the Gaussian window function,
2. `PRE_LIN_PSI` is actual node independent and in this case the precomputation can be done before setting nodes,
3. `PRE_FULL_PSI` asks for quite a bit of memory.

Doing the transform

Algorithm **??** is implemented as

```
nfft_trafo(&my_plan);
```

takes `my_plan.f_hat` as its input and overwrites `my_plan.f`. One only needs one plan for several transforms of the same kind, i.e. transforms with equal initialisation parameters. For comparison, the direct calculation of (**??**) is done by `ndft_trafo`. The adjoint transforms are given by `nfft_adjoint` and `ndft_adjoint`, respectively.

All data with multi-indices is stored plain, in particular the Fourier coefficient $\hat{f}_{\boldsymbol{k}}$ is stored in `my_plan.f_hat[k]` with `k` $:= \sum_{t=0}^{d-1}(k_t + \frac{N_t}{2}) \prod_{t'=t+1}^{d-1} N_{t'}$. Data vectors might be exchanged by

```
NFFT_SWAP_complex(my_plan.f_hat,new_f_hat);.
```

Finalisation

All memory allocated by the init routine is deallocated by

```
nfft_finalize(&my_plan);
```

Note, that almost all (de)allocation operations of the library are done by `fftw_malloc` and `fftw_free`. Additional data, declared and allocated by the application, have to be deallocated by the user's program as well.

Data structure and functions

The library defines the structure `nfft_plan`, the most important members are listed in Table **??**. Moreover, the user functions for the NFFT are collected in Table **??**. They all have return type `void` and their first argument is of type `nfft_plan*`.

Type	Name	Size	Description
`int`	`d`	1	Spatial dimension d
`int*`	`N`	d	Multibandwidth $\boldsymbol{N}$
`int`	`N_total`	1	Number of coefficients $\lvert I_{\boldsymbol{N}}\rvert$
`int`	`M_total`	1	Number of nodes M
`double complex*`	`f_hat`	$\lvert I_{\boldsymbol{N}}\rvert$	Fourier coefficients $\hat{\boldsymbol{f}}$ or adjoint coefficients $\hat{\boldsymbol{h}}$
`double complex*`	`f`	M	Samples $\boldsymbol{f}$
`double*`	`x`	dM	Sampling set $\mathcal{X}$

Table 4.1: Interesting members of `nfft_plan`.

4.3 Generalisations and nomenclature

The generalisations to the nonequispaced FFT in time and frequency domain, the nonequispaced fast (co)sine transform, and the nonequispaced sparse FFT are based on the same procedure as the NFFT. In the following we give some more details for the nonequispaced fast spherical Fourier transform.

Name	Additional arguments
ndft_trafo	
ndft_adjoint	
nfft_trafo	
nfft_adjoint	
nfft_init_1d	int N0, int M
nfft_init_2d	int N0, int N1, int M
nfft_init_3d	int N0, int N1, int N2, int M
nfft_init	int d, int* N, int M
nfft_init_guru	int d, int* N, int M, int* n, int m, unsigned nfft_flags, unsigned fftw_flags
nfft_precompute_one_psi	
nfft_check	
nfft_finalize	

Table 4.2: User functions of the NFFT.

NFSFT

Compared to the NFFT routines, the NFSFT module shares a similar basic procedure for computing the transformations. It differs, however, slightly in data layout and function interfaces.

The NFSFT routines depend on initially precomputed global data that limits the maximum degree `N` that can be used for computing transformations. This precomputation has to be performed only once at program runtime regardless of how many individual transform plans are used throughout the rest. This is done by

```
nfsft_precompute(N,1000,0U,0U);
```

Here, `N` is the maximum degree that can be used in all subsequent transformations, `1000` is the threshold for the FPTs used internally, and `0U` and `0U` are optional flags for the NFSFT and FPT. If you are unsure which values to use, leave the threshold at `1000` and the flags at `0U`.

Initialisation of a plan is done by calling one of the `nfsft_init`-functions. We need no dimension parameter here, as in the NFFT case. The simplest version is a call to `nfsft_init` specifying only the bandwidth N and the number of nonequispaced nodes M. For an application-owned variable `nfsft_plan my_plan` the function call is

```
nfsft_init(&my_plan,N,M);
```

The first argument should be uninitialised. Memory allocation is completely done by the init routine.

After initialising a plan, one defines the nodes $(\vartheta_j, \varphi_j) \in [0, 2\pi) \times [0, \pi]$ in spherical coordinates in the member variable `my_plan.x`. For consistency with the other modules and the conventions used there, we use swapped and scaled spherical coordinates

$$\tilde{\varphi} := \begin{cases} \frac{\varphi}{2\pi}, & \text{if } 0 \leq \varphi < \pi, \\ \frac{\varphi}{2\pi} - 1, & \text{if } \pi \leq \varphi < 2\pi, \end{cases} \qquad \tilde{\vartheta} := \frac{\vartheta}{2\pi}$$

and identify a point from $\mathbb{S}^2$ with the vector $(\tilde{\varphi}, \tilde{\vartheta}) \in [-\frac{1}{2}, \frac{1}{2}) \times [0, \frac{1}{2}]$. The angles $\tilde{\varphi}_j$ and $\tilde{\vartheta}_j$ for the j-th node are assigned by

```
my_plan.x[2*j]   = /* your choice in [-0.5,0.5] */;
my_plan.x[2*j+1] = /* your choice in [0,0.5] */;
```

After setting the nodes, the same node-dependent (see the exception above) precomputation as for the NFFT has to be performed. This is done by

```
nfsft_precompute_x(&my_plan);
```

which itself calls `nfft_precompute_one_psi(&my_plan)` as explained above. You might modify the precomputation strategy by passing the appropriate NFFT flags to one of the more advanced `nfsft_init` routines. Setting the spherical Fourier coefficients $\hat{f}_k^n$ in `my_plan.f_hat` should be done using the helper macro `NFSFT_INDEX`. This reads

```
for (k = 0; k <= N; k++)
  for (n = -k; n <= k; n++)
    my_plan.f_hat[NFSFT_INDEX(k,n,&plan)] = /* your choice */;
```

For the NFSFT, spherical Fourier coefficients are stored in a fashion different from the NFFT. You should always use the helper macro `NFSFT_INDEX` for setting the spherical Fourier coefficients. If, in place of a NFSFT transformation, you would like to perform an adjoint NFSFT transformation, you set the values $f(\vartheta_j, \varphi_j)$ in `my_plan.f` as follows:

```
for (j = 0; j < M; j++)
  my_plan.f[j] = /* your choice */;
```

The actual transforms are computed by calling either `nfsft_trafo(&my_plan)` for the NFSFT (see Algorithm **??**) or `nfsft_adjoint(&my_plan)` for the adjoint NFSFT (see Algorithm **??**). Remember, that you must make sure that `my_plan.x` and `my_plan.f_hat` or `my_plan.f` have been initialised appropriately prior to calling the corresponding transformation routine. On execution, `nfsft_trafo` overwrites `my_plan.f` while `nfsft_adjoint` overwrites `my_plan.f_hat`. One only needs one plan for several transforms of the same kind, i.e. transforms with equal initialisation parameters. For comparison, the direct calculation of (**??**) and (**??**) are done by `ndsft_trafo` and `ndsft_adjoint`, respectively.

All memory allocated by the init routine is deallocated by

```
nfsft_finalize(&my_plan);
```

while the memory allocated by `nfsft_precompute` gets freed by calling

```
nfsft_forget();
```

Additional data, declared and allocated by the application, has to be deallocated by the user's program as well.

The library defines the structure `nfsft_plan`, the most important members are listed in Table **??**. The structure contains, public read-only (r) and public read-write (w) members. The user functions for the NFSFT are collected in Table **??**.

Some more things should be kept in mind when using the NFSFT module:

Type	Name	Size	Type	Description
`int`	`N`	1	r	Bandwidth N
`int`	`N_total`	1	r	Set to $4\lvert\mathcal{I}_N\rvert = 4(N+1)^2$, quadruple of the number of Fourier coefficients
`int`	`M_total`	1	r	Total number of nodes M
`double complex*`	`f_hat`	$4\lvert\mathcal{I}_N\rvert$	w	Fourier coefficients $\hat{\boldsymbol{f}}$ or adjoint coefficients $\hat{\boldsymbol{h}}$
`double complex*`	`f`	M	w	Samples $\boldsymbol{f}$
`double*`	`x`	M	w	Sampling set $\mathcal{X}$

Table 4.3: Interesting members of `nfsft_plan`.

Name	Additional arguments
`nfsft_precompute`	`int N, double threshold, unsigned int nfsft_flags, unsigned int fpt_flags`
`nfsft_init`	`int N, int M`
`nfsft_init_advanced`	`int N, int M, unsigned int nfsft_flags`
`nfsft_init_guru`	`int N, int M, unsigned int nfsft_flags, int nfft_flags, int nfft_cutoff`
`nfsft_precompute_x`	
`ndsft_trafo`	
`ndst_adjoint`	
`nfsft_trafo`	
`nfsft_adjoint`	
`nfsft_finalize`	
`nfsft_forget`	

Table 4.4: User functions of the NFSFT.

- The bandwidth $N_{\max}$ up to which global precomputation is performed when calling `nfsft_precompute` is always chosen as the next power of two with respect to the specified maximum bandwidth N.
- By default, the NDSFT and NFSFT transformations, `ndsft_trafo` and `nfsft_trafo`, are allowed to destroy the input `f_hat` while the input `x` is preserved. On the contrary, the adjoint NDSFT and NFSFT transformations, `ndsft_adjoint` and `nfsft_adjoint`, do not destroy the input `f` and `x` by default. The desired behaviour can be assured by using the `NFSFT_PRESERVE_F_HAT`, `NFSFT_PRESERVE_X`, `NFSFT_PRESERVE_F` and `NFSFT_DESTROY_F_HAT`, `NFSFT_DESTROY_X`, `NFSFT_DESTROY_F` flags.

4.4 Inversion and solver module

The flow chart in Figure **??** shows how to use the inverse transforms as implemented in the solver module of the library. There is no general stopping rule implemented, since this task is highly dependent on the particular application.

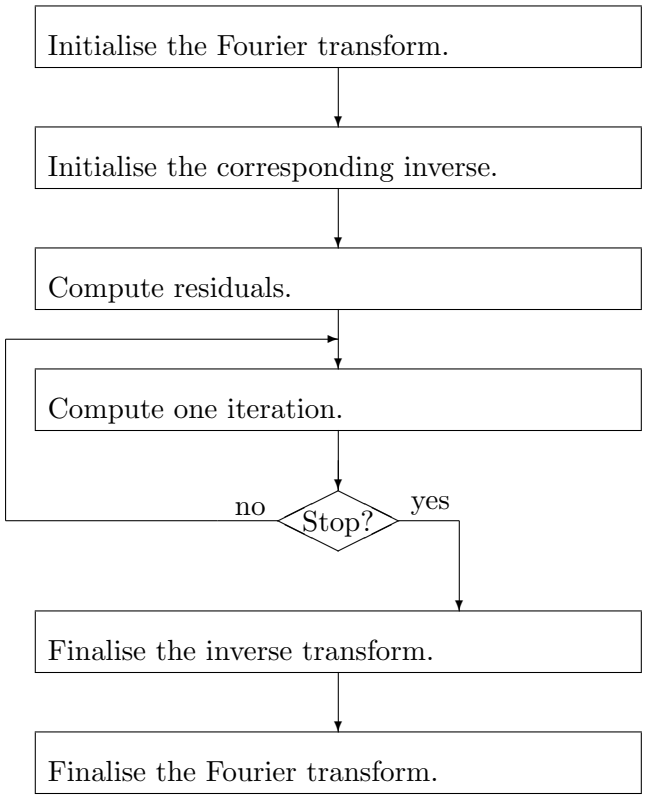


Figure 4.1: Flow chart of the inverse transforms.

Each inverse transform basically wraps an already initialised Fourier transform. The user specifies one of the Algorithms **??**-**??** by setting one of the flags (`LANDWEBER`, `STEEPEST_DESCENT`, `CGNR`, `CGNE`). Weights and/or damping factors are used if the flags `PRECOMPUTE_WEIGHT`, `PRECOMPUTE_DAMP` are set and one has to initialise the members `my_iplan.w`, `my_iplan.w_hat` in these cases. Default flags are `CGNR`.

In case of the NFFT, the library defines the structure `infft_plan`. The members of this plan are given by Table **??**. The user functions for the inverse NFFT are collected in Table **??**. They all have return type `void` and their first argument is of type `infft_plan*`. Replacing `nfft` by any other Fourier transform gives the appropriate inverse to this transform.

Type	Name	Size	Description
`double*`	`w`	`M_total`	weights $\boldsymbol{w}$
`double*`	`w_hat`	`N_total`	damping factors $\hat{\boldsymbol{w}}$
`(FLT_TYPE)*`	`y`	`M_total`	right hand side $\boldsymbol{y}$
`(FLT_TYPE)*`	`f_hat_iter`	`N_total`	actual solution
`(FLT_TYPE)*`	`r_iter`	`M_total`	residual vector $\boldsymbol{r}_{l+1}$
`double`	`dot_r_iter`	1	$\|\boldsymbol{r}_{l+1}\|_{\boldsymbol{W}}^2$

Table 4.5: More important members of each inverse plan, where `(FLT_TYPE)` is `double` for the NFCT and NFST and `double complex` in all other cases.

Name	Additional arguments
`infft_init`	`nfft_plan *mv`
`infft_init_advanced`	`nfft_plan *mv, unsigned infft_flags`
`infft_before_loop`	
`infft_loop_one_step`	
`infft_finalize`	

Table 4.6: User functions of the inverse NFFT.

5 Examples

The library was tested on a AMD Athlon(tm) XP 2700+, 1GB memory, SuSe-Linux, kernel 2.4.20-4GB-athlon, gcc version 3.3. In all tests with random input the nodes $\boldsymbol{x}_j$ and the Fourier coefficients $\hat{f}_{\boldsymbol{k}}$ are chosen pseudo randomly with $\boldsymbol{x}_j \in [-0.5, 0.5]^d$ and $\hat{f}_{\boldsymbol{k}} \in [0,1] \times [0,1]$i.

5.1 Computing your first transform

The following code summarises the steps of Section **??** and computes a univariate NFFT from 14 Fourier coefficients and 19 nodes. Note that this routine is part of `simple_test.c` in `examples/nfft/` and uses additional routines as defined `include/util.h` to set up and show vectors.

```
void simple_test_nfft_1d()
{
  nfft_plan p;
  int N=14;
  int M=19;

  nfft_init_1d(&p,N,M);

  nfft_vrand_shifted_unit_double(p.x,p.M_total);

  if(p.nfft_flags & PRE_ONE_PSI)
    nfft_precompute_one_psi(&p);

  nfft_vrand_unit_complex(p.f_hat,p.N_total);
  nfft_vpr_complex(p.f_hat,p.N_total,"given Fourier coefficients, f_hat");

  ndft_trafo(&p);
  nfft_vpr_complex(p.f,p.M_total,"ndft, f");

  nfft_trafo(&p);
  nfft_vpr_complex(p.f,p.M_total,"nfft, f");

  nfft_finalize(&p);
}
```

5.2 Computation time vs. problem size

The program `nfft_times` in the same directory compares the computation time of the FFT ([**?**], `FFTW_MEASURE`), the straightforward evaluation of (**??**), denoted by NDFT, and the NFFT for increasing total problem sizes $|I_{\boldsymbol{N}}|$ and space dimensions $d = 1, 2, 3$, where $\boldsymbol{N} = (N, \ldots, N)^\top$, $N \in \mathbb{N}$. While the nodes for the FFT are restricted to the lattice $\boldsymbol{N}^{-1} \odot I_{\boldsymbol{N}}$, we choose $M = N^d$ random nodes for the NDFT and the NFFT. Within the latter, we use the oversampling factor $\sigma = 2$, the cut-off $m = 4$, and the Kaiser-Bessel window function

(PRE_PSI, PRE_PHI_HUT). This results in a fixed accuracy of $E_\infty := \|\boldsymbol{f} - \boldsymbol{s}\|_\infty / \|\hat{\boldsymbol{f}}\|_1 \approx 10^{-8}$ for $d = 1, 2, 3$.

l_N	FFT	NDFT	NFFT	l_N	FFT	NDFT	NFFT
$d = 1$				$d = 2$			
3	$1.3e-07$	$8.7e-06$	$4.6e-06$	6	$9.9e-07$	$5.7e-04$	$3.2e-04$
4	$2.0e-07$	$3.5e-05$	$8.7e-06$	8	$4.4e-06$	$9.2e-03$	$1.3e-03$
5	$4.0e-07$	$1.4e-04$	$1.7e-05$	10	$2.1e-05$	$1.5e-01$	$5.2e-03$
6	$8.9e-07$	$5.6e-04$	$3.6e-05$	12	$1.2e-04$	$2.4e+00$	$2.3e-02$
7	$2.2e-06$	$2.2e-03$	$7.2e-05$	14	$1.7e-03$	$3.8e+01$	$1.5e-01$
8	$4.8e-06$	$9.0e-03$	$1.4e-04$	16	$2.1e-02$	*	$6.8e-01$
9	$1.1e-05$	$3.6e-02$	$2.9e-04$	18	$8.4e-02$	*	$2.8e+00$
10	$2.4e-05$	$1.4e-01$	$6.0e-04$	20	$3.2e-01$	*	$1.2e+01$
11	$5.7e-05$	$5.8e-01$	$1.4e-03$	22	$1.4e+00$	*	$5.3e+01$
12	$1.5e-04$	$2.3e+00$	$3.2e-03$	$d = 3$			
13	$5.5e-04$	$9.4e+00$	$8.2e-03$	9	$1.0e-05$	$3.7e-02$	$2.5e-02$
14	$1.7e-03$	$3.8e+01$	$2.0e-02$	12	$1.1e-04$	$2.4e+00$	$2.5e-01$
15	$3.8e-03$	$1.5e+02$	$4.9e-02$	15	$3.4e-03$	$1.5e+02$	$2.4e+00$
16	$8.2e-03$	*	$1.2e-01$	18	$5.2e-02$	*	$2.1e+01$
17	$1.9e-02$	*	$2.4e-01$	21	$9.0e-01$	*	$1.8e+02$
18	$4.5e-02$	*	$3.6e-01$				
19	$9.2e-02$	*	$9.8e-01$				
20	$1.9e-01$	*	$2.1e+00$				
21	$4.2e-01$	*	$4.2e+00$				
22	$1.0e-00$	*	$9.5e+00$				

Table 5.1: Computation time in seconds with respect to $l_N = \log_2 |I_{\boldsymbol{N}}|$. Note that we used accumulated measurements in case of small times and the times (*) are not displayed due to the large response time in comparison to the FFT time.

We conclude the following: The FFT and the NFFT show the expected $\mathcal{O}(|I_{\boldsymbol{N}}| \log |I_{\boldsymbol{N}}|)$ time complexity, i.e., doubling the total size $|I_{\boldsymbol{N}}|$ results in only slightly more than twice the computation time, whereas the NDFT behaves as $\mathcal{O}(|I_{\boldsymbol{N}}|^2)$. Note furthermore, that the constant in the $\mathcal{O}$-notation is independent of the space dimension d for the FFT and the NDFT, whereas the computation time of the NFFT increases considerably for larger d.

5.3 Accuracy vs. window function and cut-off parameter m

The accuracy of the Algorithm **??**, measured by

$$E_2 = \frac{\|\boldsymbol{f} - \boldsymbol{s}\|_2}{\|\boldsymbol{f}\|_2} = \left(\sum_{j=0}^{M-1} |f_j - s(\boldsymbol{x}_j)|^2 / \sum_{j=0}^{M-1} |f_j|^2 \right)^{\frac{1}{2}}$$

and

$$E_\infty = \frac{\|\boldsymbol{f} - \boldsymbol{s}\|_\infty}{\|\hat{\boldsymbol{f}}\|_1} = \max_{0 \le j < M} |f_j - s(\boldsymbol{x}_j)| / \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} |\hat{f}_{\boldsymbol{k}}|$$

is shown in Figure **??**.

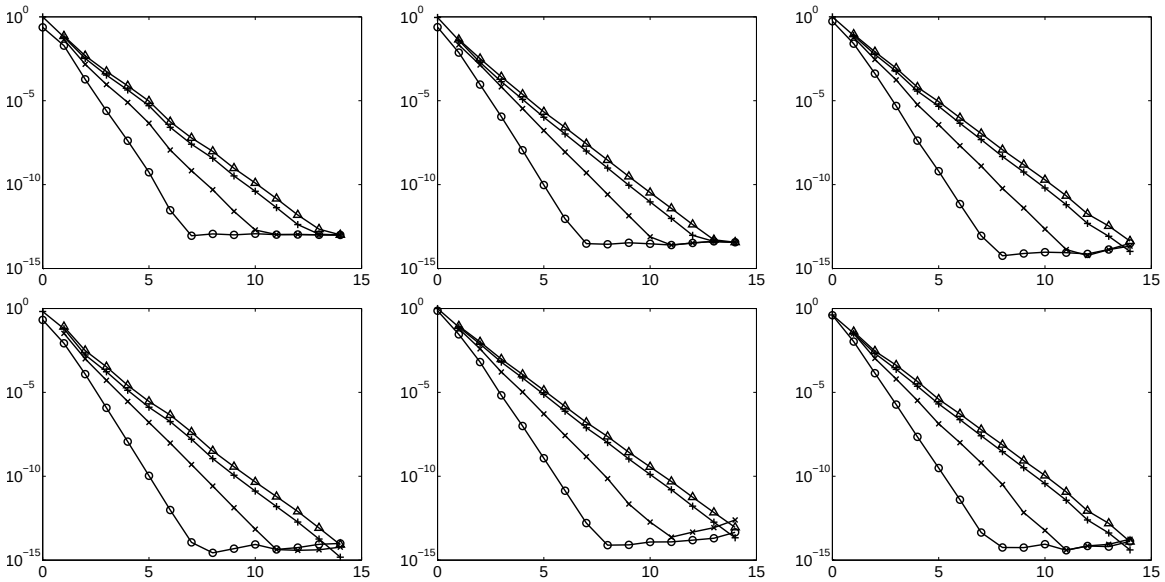


Figure 5.1: The error E_2 (top) and E_∞ (bottom) with respect to m, from left to right $d = 1, 2, 3$ ($N = 2^{12}, 2^6, 2^4$, $\sigma = 2$, $M = 10000$), for Kaiser Bessel- (circle), Sinc- (x), B-Spline- (+), and Gaussian window (triangle).

5.4 Computing an inverse transform

The usage of the inverse NFFT is demonstrated by `simple_test` in `examples/solver`. Executing the MATLAB script `glacier.m` in the same directory produces the following two plots. Note that the corresponding C-file `glacier.c` is called from the MATLAB script.

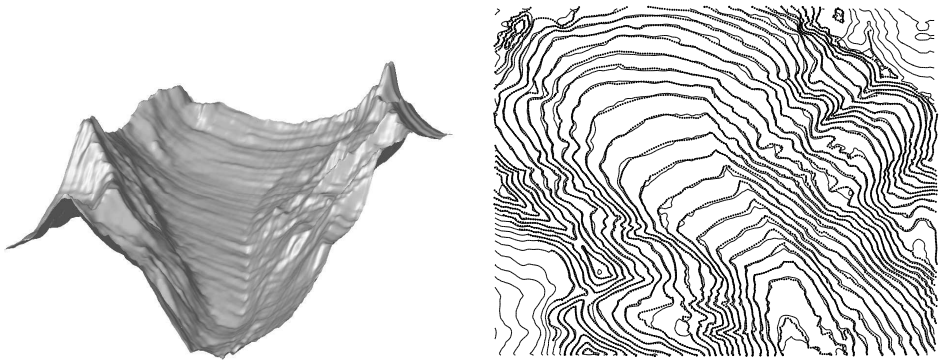


Figure 5.2: Reconstruction of the glacier from samples at $M = 8345$ nodes (`vol87.dat` from [**?**]) with $N_0 = N_1 = 256$ and 40 iterations.

6 Applications

In this section we describe important applications which are based on the NFFT kernel. One can find these programs in the directory `applications`.

6.1 Summation of smooth and singular kernels

We are interested in the fast evaluation of linear combinations of radial functions, i.e. the computation of

$$g\left(\boldsymbol{y}_j\right) := \sum_{k=1}^{N} \alpha_k K\left(\left\|\boldsymbol{y}_j - \boldsymbol{x}_k\right\|_2\right)$$

for $j = 1, \ldots, M$ and nodes $\boldsymbol{x}_k, \boldsymbol{y}_j \in \mathbb{R}^d$. For smooth kernels K with an additional parameter $c > 0$, e.g. the Gaussian $K(x) = \mathrm{e}^{-x^2/c^2}$, the multiquadric $K(x) = \sqrt{x^2 + c^2}$ or the inverse multiquadric $K(x) = 1/\sqrt{x^2 + c^2}$ our algorithm requires $\mathcal{O}(N + M)$ arithmetic operations. In the case of singular kernels K, e.g.,

$$\frac{1}{x^2},\ \frac{1}{|x|},\ \log|x|,\ x^2\log|x|,\ \frac{1}{x},\ \frac{\sin(cx)}{x},\ \frac{\cos(cx)}{x},\ \cot(cx)$$

an additional regularisation procedure must be incorporated and the algorithm has the arithmetic complexity $\mathcal{O}(N \log N + M)$ or $\mathcal{O}(M \log M + N)$ if either the target nodes $\boldsymbol{y}_j$ or the source nodes $\boldsymbol{x}_k$ are "reasonably uniformly distributed".

Note that the proposed fast algorithm [**?**, **?**, **?**] generalises the diagonalisation of convolution matrices by Fourier matrices to the setting of arbitrary nodes. In particular, this yields nearly the same arithmetic complexity as the FMM [**?**] while allowing for an easy change of various kernels. A recent application in particle simulation is given in [**?**]. The directory `applications/fastsum` contains C and MatLab programs that show how to use the fast summation method.

6.2 Fast Gauss transform

This is a special case of the fast summation method, we compute approximations of the following sums. Given complex coefficients $\alpha_k \in \mathbb{C}$ and source nodes $x_k \in [-\frac{1}{4}, \frac{1}{4}]$, our goal consists in the fast evaluation of the sum

$$g\left(y\right) = \sum_{k=1}^{N} \alpha_k \mathrm{e}^{-\sigma|y - x_k|^2}$$

at the target nodes $y_j \in [-\frac{1}{4}, \frac{1}{4}]$, $j = 1, \ldots, M$, where $\sigma = a + \mathrm{i}b$, $a > 0$, $b \in \mathbb{R}$, denotes a complex parameter. For details see [**?**] and the related paper [**?**] for applications. All numerical examples of [**?**] are produced by the programs in `applications/fastgauss`.

6.3 Summation of zonal functions on the sphere

Given $M, N \in \mathbb{N}$, arbitrary source nodes $\boldsymbol{\eta}_k \in \mathbb{S}^2$ and real coefficients $\alpha_k \in \mathbb{R}$, evaluate the sum

$$g\left(\boldsymbol{\xi}\right) := \sum_{k=1}^{N} \alpha_k K\left(\boldsymbol{\eta}_k \cdot \boldsymbol{\xi}\right)$$

at the target nodes $\boldsymbol{\xi}_j \in \mathbb{S}^2$, $j = 1, \ldots, M$. The naive approach for evaluating this sum takes $\mathcal{O}(MN)$ floating point operations if we assume that the zonal function K can be evaluated in constant time or that all values $K(\boldsymbol{\eta}_k \cdot \boldsymbol{\xi}_j)$ can be stored in advance.

In contrast, our scheme is based on the nonequispaced fast spherical Fourier transform, has arithmetic complexity $\mathcal{O}(M + N)$, and can be easily adapted to such different kernels K as

1. the Poisson kernel $Q_h : [-1, 1] \to \mathbb{R}$ with $h \in (0, 1)$ given by

$$Q_h(x) := \frac{1}{4\pi} \frac{1 - h^2}{(1 - 2hx + h^2)^{3/2}},$$

2. the singularity kernel $S_h : [-1, 1] \to \mathbb{R}$ with $h \in (0, 1)$ given by

$$S_h(x) := \frac{1}{2\pi} \frac{1}{(1 - 2hx + h^2)^{1/2}},$$

3. the locally supported kernel $L_{h,\lambda} : [-1, 1] \to \mathbb{R}$ with $h \in (-1, 1)$ and $\lambda \in \mathbb{N}_0$ given by

$$L_{h,\lambda}(x) := \begin{cases} 0 & \text{if } -1 \leq x \leq h, \\ \frac{\lambda+1}{2\pi(1-h)^{\lambda+1}} (x - h)^\lambda & \text{if } h < x \leq 1, \end{cases}$$

   or

4. the spherical Gaussian kernel $G_\sigma : [-1, 1] \to \mathbb{R}$ with $\sigma > 0$

$$G_\sigma(x) := \mathrm{e}^{2\sigma x - 2\sigma}.$$

For details see [**?**], all corresponding numerical examples can be found in `applications/fastsumS2`.

6.4 Iterative reconstruction in magnetic resonance imaging

In magnetic resonance imaging (MRI) the raw data is measured in k-space, the domain of spatial frequencies. Methods that use a non-Cartesian sampling grid in k-space, e.g. a spiral, are becoming increasingly important. Reconstruction is usually performed by resampling the data onto a Cartesian grid and the usage of the standard FFT - often denoted by gridding. Another approach, the inverse model, is based on an implicit discretisation. Both discretisations are solved efficiently by means of the NFFT and the inverse NFFT, respectively. Furthermore, a unified approach to field inhomogeneity correction has been included, see [**?**, **?**] for details.

6.5 Computation of the polar FFT

The polar FFT is a special case of the NFFT, where one computes the Fourier transform on particular grids. Of course, the polar as well as a so-called pseudo-polar FFT can be computed very accurately and efficiently by the NFFT. Furthermore, the reconstruction of a $2d$ signal from its Fourier transform samples on a (pseudo-)polar grid by means of the inverse nonequispaced FFT is possible under certain density assumptions. For details see [**?**] and for further applications [**?**].

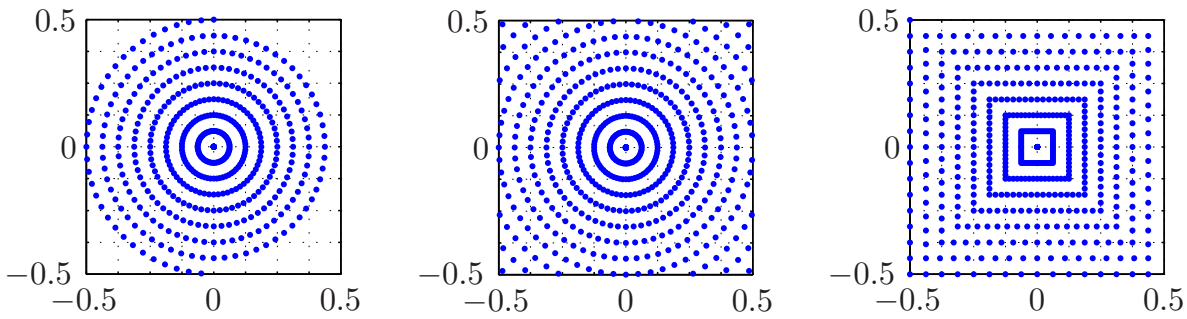


Figure 6.1: Left to right: polar, modified polar, and linogram grid of size $R = 16$, $T = 32$.

6.6 Radon transform, computer tomography, and ridgelet transform

We are interested in efficient and high quality reconstructions of digital $N \times N$ medical images from their Radon transform. The standard reconstruction algorithm, the filtered backprojection, ensures a good quality of the images at the expense of $\mathcal{O}(N^3)$ arithmetic operations. Fourier reconstruction methods reduce the number of arithmetic operations to $\mathcal{O}(N^2 \log N)$. Unfortunately, the straightforward Fourier reconstruction algorithm suffers from unacceptable artifacts so that it is useless in practice. A better quality of the reconstructed images can be achieved by our algorithm based on NFFTs. For details see [**?**, **?**, **?**] and the directory `applications/radon`.

Another application of the discrete Radon transform is the discrete Ridgelet transform, see e.g. [**?**]. A simple test program for denoising an image by hard thresholding the ridgelet coefficients can be found in `applications/radon`. It uses the NFFT-based discrete Radon transform and the translation-invariant discrete Wavelet transform based on MatLab toolbox WaveLab850 [**?**]. See [**?**] for details.