Computation of the Fractional Fourier Transform

Adhemar Bultheel and Héctor E. Martínez Sulbaran[1]

Dept. of Computer Science, Celestijnenlaan 200A, B-3001 Leuven

Abstract

In this note we make a critical comparison of some matlab programs for the digital computation of the fractional Fourier transform that are freely available and we describe our own implementation that filters the best out of the existing ones. Two types of transforms are considered: First the fast approximate fractional Fourier transform algorithm for which two algorithms are available. The method is described in H.M. Ozaktas, M.A. Kutay, and G. Bozdağı. *Digital computation of the fractional Fourier transform. IEEE Trans. Signal Process.*, 44:2141–2150, 1996. There are two implementations: one is written by A.M. Kutay the other is part of package written by J. O'Neill. Secondly the discrete fractional Fourier transform algorithm described in the master thesis *C. Candan. The discrete fractional Fourier transform.* Bilkent Univ., 1998 and an algorithm described by S.C. Pei, M.H. Yeh, and C.C Tseng: *Digital fractional Fourier transform based on orthogonal projections IEEE Trans. Signal Process.*, 47:1335–1348, 1999.

Key words: Fractional Fourier transform

1 Introduction

The idea of fractional powers of the Fourier transform operator appears in the mathematical literature as early as 1929 [23,9,11]. Later on it was used in quantum mechanics [14,13] and signal processing [1], but it was mainly the optical interpretation and the applications in optics that gave a burst of publications since the 1990's that culminated in the book of Ozaktas et al [17].

The reason for its success in optical applications can be explained as follows. Consider a system which consists of a point light source on the left. The light illuminates an object after traversing a set of optical components like e.g. thin lenses. It is then well known that at certain points to the right of the object one may observe images that are the Fourier transform of the object image. Somewhat further it is the inverted object image, still further it becomes the inverted Fourier transform and still further it is the upright image etc. These images are obtained by the Fourier operator applied to the object image, its second power (the inverted image), the 3rd power (the

[1] The work of the first author is partially supported by the Fund for Scientific Research (FWO), projects "CORFU: Constructive study of orthogonal functions", grant #G.0184.02 and, "SMA: Structured matrices and their applications", grant G#0078.01, "ANCILA: Asymptotic analysis of the convergence behavior of iterative methods in numerical linear algebra", grant #G.0176.02, the K.U.Leuven research project "SLAP: Structured linear algebra package", grant OT-00-16, the Belgian Programme on Interuniversity Poles of Attraction, initiated by the Belgian State, Prime Minister's Office for Science, Technology and Culture. The scientific responsibility rests with the author.

inverted Fourier image) and the 4th power (the srcinal image), etc. The images in between are the result of intermediate (fractional) powers of the Fourier operator applied to the image object.

Like for the Fourier transform, there exists a discrete version of the fractional Fourier transform. It is based on an eigenvalue decomposition of the discrete Fourier transform matrix. If $F = E\Lambda E^{-1}$ is this decomposition then $F^a = E\Lambda^a E^{-1}$ is the corresponding discrete fractional F ourier transform.

As far as we know, there are not many public domain software routines available for the computation of the (discrete) fractional Fourier transform. We are aware of only two routines. There is the one that can be found on the web site [12] of the book [17] previously described in Ozaktas [16] and another one which is part of a package compiled by O'Neill. In this note we shall analyse the advantages and disadvantages of the two algorithms and propose some improvements.

The text is organized as follows. In Section 2 we briefly recall the mathematical background of the (discrete) fractional Fourier transform. Section 3 describes the algorithm of Candan and the implementation aspects are discussed in the subsequent section 4. Some modifications are proposed in section 5. The theory of the discrete transform is given on section 6 and the practical aspects are found in section 7. The performance in approximating the Gauss-Hermite functions are illustrated in section 8 and its approximation of the fast approximate transform in section 10. In section 9 alternative definitions of the discrete fractional Fourier transform are briefly discussed. Finally in section 11, an illustration of the algorithms on de two dimensional example is given.

2 Mathematical Background

In this section we introduce the definition of the continuous fractional Fourier transform (FrFT) and the discrete fractional Fourier transform.

Definition 1 (Continuous Fractional Fourier Transform) *The fractional Fourier transform $\mathcal{F}^a$ of order $a \in \mathbb{R}$ is a linear integral operator that maps a given function (signal) $f(x)$, $x \in \mathbb{R}$ onto $f_a(\xi)$, $\xi \in \mathbb{R}$ by*

$$f^a(\xi) = \mathcal{F}^a(\xi) = \int_{-\infty}^{+\infty} K^a(\xi, x) f(x) \mathrm{d}x$$

where the kernel is defined as follows. Set $\alpha = a\pi/2$ then

$$K_a(\xi, x) = C_\alpha \exp\left\{-i\pi\left(2\frac{x\xi}{\sin\alpha} - (x^2 + \xi^2)\cot\alpha\right)\right\},$$

with

$$C^\alpha = \sqrt{1 - i\cot\alpha} = \frac{\exp\quad i[\pi\mathrm{sgn}(\sin\alpha)/4\quad \alpha/2]}{\{-\quad \sqrt{/\sin\alpha/}\quad -\quad\}}.$$

For $k \in 2\mathbb{Z}$, limiting values are taken.

Note that for $a \in 4\mathbb{Z}$, the FrFT becomes the identity $f_{4k}(\xi) = f(\xi)$, hence the kernel is in that case

$$K_{4k}(\xi, x) = \delta(\xi - x), \quad k \in \mathbb{Z}$$

and for $a \in 2 + 4\mathbb{Z}$, this is the parity operator $f_{2+4k}(\xi) = f(-\xi)$, corresponding to the kernel

$$K_{2+4k}(\xi, x) = \delta(\xi + x), \quad k \in \mathbb{Z},$$

while for $a \in 1+4\mathbb{Z}$, $\mathcal{F}^a = \mathcal{F}^1$ is just the Fourier operator $\mathcal{F}$, and for $a \in 3+4\mathbb{Z}$, $\mathcal{F}^a = \mathcal{F}^3 = \mathcal{F}^2\mathcal{F}$, in other words, if $f_1(\xi) = \mathcal{F}f(x)$ is the Fourier transform, then $f_3(\xi) = f_1(-\xi)$.

This should make clear that $\mathcal{F}^a$ can be interpreted as the ath power of the Fourier transform which may be interpreted modulo 4. So, we have for example the well known properties $\mathcal{F}^a\mathcal{F}^b = \mathcal{F}^{a+b}$ and $\mathcal{F}^{-a}\mathcal{F}^a = \mathcal{I}$ is the identity.

In the theory of the fractional Fourier transform, a special role is played by a chirp function.

Definition 2 (Chirp function) *A chirp is a function that sweeps a certain frequency interval $[\omega^0, \omega^1]$ in a certain time interval $[t^0, t^1]$. If the sweep rate is linear, it has the form $\exp\{i\pi(\chi x+\gamma)x\}$ with χ the sweep rate.*

Note that $\mathcal{F}^a(\delta(x - \gamma)) = K_a(\xi, \gamma)$ is a chirp with sweep rate $\cot \alpha$.

Note also that

$$x^2 \cot \alpha - 2x\xi \csc \alpha + \xi^2 \cot \alpha = x^2(\cot \alpha - \csc \alpha) + (x - \xi)^2 \csc \alpha + \xi^2(\cot \alpha - \csc \alpha).$$

Thus the fractional Fourier transform can be seen as applying a chirp convolution with sweep rate $\csc \alpha$ in between two chirp multiplications with sweep rate $\cot \alpha$.

If the signal f consists of only a finite number of discrete samples, we have to define a corresponding discrete fractional Fourier transform. We shall introduce that via the classical discrete Fourier transform.

Definition 3 (Discrete Fourier Transform) *The discrete Fourier transform of a vector $f = [f(0),...,f(N-1)]^T$ is defined as the vector $f_1 = Ff$ where the $N \times N$ DFT matrix F has entries that are the Nth roots of unity: $F(k, n) = W^{kn}/\sqrt{N}$ with $W = e^{-i2\pi/N}$. Hence*

$$f_1(k) = \sum_{n=0}^{N-1} F(k, n)f(n) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} f(n)e^{-i\frac{2\pi k}{N}}, \quad k = 0,...,N-1.$$

The DFT matrix F satisfies $F^4 = I$ with I the identity matrix. It has the eigenvalues $\{1, -i, -1, i\} = \{e^{-ik\pi/2} : k = 0,...,N-1\}$. It has also N independent orthonormal eigenvectors that can be arranged as the columns of a matrix E, so that its eigenvalue decomposition is $F = E\Lambda E^T$. The definition of the discrete fractional Fourier transform is then easily given as a multiplication with a (fractional) power of the Fourier matrix. Note however that the choice of the set of eigenvectors

for the matrix F is not unique, even if we suppose them to be orthonormal. There are only fou eigenvalues and the choice of the (orthonormal) basisv ectors within each eigenspace is not unique. There are some arguments, that require a particular choice of these eigenvectors which impose certain symmetry conditions as we shall see later, and this will fix the eigenvectors uniquely.

Definition 4 (Discrete Fractional Fourier Transform) *A discrete fractional Fourier transform of a vector* $f = [f(0),...,f(N-1)]^T$ *is defined as the vector* $f_a = F^a f$*, i.e., the vector with components*

$$f_a(k) = F^a f = \sum_{n=0}^{N-1} F^a(k,n) f(n), \quad k = 0,...,N-1,$$

where $F^a = E\Lambda^a E^T$*, with* $F = E\Lambda E^T$ *an eigenvalue decomposition of the DFT matrix.*

Besides the ambiguity in the choice of the eigenvectors that we mentioned before, also the choice of the branch in the powers Λ^a for a real introduces another ambiguity which can be given an easy solution because the eigenvalus in the discrete case are $\{e^{-n\pi/2}\}$ with $n = 0, 1, 2, ...$, which is a cyclic repetition of $\{-i, -1, i, 1\}$, and then the powers can be defined as $e^{-ian\pi/2}$ for a real.

The discrete fractional Fourier transform can be used as an approximation of the continuous fractional Fourier transform when N is large as will be explained in section 8. But there is a more direct and faster way to compute an approximation of the continuous fractional Fourier transform, as will be explained now.

3 Fast computation of the continuous fractional Fourier transform algorithms

The algorithms considered here are algorithms that approximate the continuous fractional Fourier transform in the sense that they map samples of the signal to samples of the samples of the continuous fractional Fourier transform. Because it uses FFT techniques with complexity $N \log N$, it is a fast algorithm. However, it is not really the fractional analog of the Fast Fourier Transform (FFT) which is a particular fast implementation of the Discrete Fourier Transform (DFT). And although it implements just the FFT when the power $a = 1$ and the inverse FFT when $a = 3$, and although some authors have coined it fast fractional Fourier transforms, we deliberately avoid to use this term, and call it fast approximate fractional Fourier transform (FAFrFT). For the moment, the fast implementation of the discrete fractional Fourier transform which would be the genuine fast fractional Fourier transform (FFrFT), is yet unknown.

The continuous integral transform is approximated by a quadrature formula using samples of the srcinal function and computing only samples of the transformed function. We are aware of two matlab implementations of the fast approximate fractional Fourier transform.

First there is the matlab routine `fracF`, implementing the algorithm described in [16] (see also [17, Section 6.7]). It can we found on the web page [12].

The second one is another matlab code `fracft`, which is part of a software package developed by J. O Neill, now available at the mathworks website [15]. It also refers to the same paper but has some differences in implementation.

For completeness, we recall briefly the idea of the algorithm. As we mentioned before, the definition

is rewritten in the form of a convolution in between two chirp multiplications. Using $\cot\alpha - \csc\alpha = -\tan(\alpha/2)$, we have

$$f_a(\xi) = C_\alpha e^{-i\pi \tan(\alpha/2)\xi^2} \int_{-\infty}^{+\infty} e^{i\pi \csc\alpha\,(\xi-x)^2} [e^{-i\pi \tan(\alpha/2)\, x^2} f(x)]\mathrm{d}x.$$

A first approximation consists in assuming that the functions can be confined to the interval $[-\Delta/2, \Delta/2]$ in all time-frequency directions. This means that the Wigner distribution is essentially confined to a circle with radius $\Delta/2$ around the srcin of the time-frequency plane. As explained in the appendix A of [16]. The Wigner distribution of a chirp multiplication and of a chirp convolution for an f under consideration, will be compact and contained in a circle with radius Δ. That is twice the srcinal support. Therefore the integral can be restricted to the interval $[-\Delta, \Delta]$. Also, if we want to recover the result from discrete samples, then we should have samples at intervals $1/2\Delta$. Thus, assuming we have $N = \Delta^2$ samples of the srcinal f, then we need at least $2N$ samples of the convolution. So the integral sampled at $\xi_k = k/2\Delta$ can be approximated as

$$f_a(\xi_k) \approx \frac{C_\alpha}{2\Delta} e^{-i\pi\xi_k^2 \tan(\alpha/2)} \sum_{l=-N}^{N-1} g(k-l)h(l) \tag{1}$$

with $x_k = k/2\Delta$ and

$$g(k) = e^{i\pi x_k^2 \csc\alpha}, \quad h(k) = e^{-i\pi x_k^2 \tan(\alpha/2)} f(x_k).$$

This gives a discrete approximation of the continuous transform. The convolution has $O(N\log N)$ complexity when using FFT. Therefore it is called a fast approximate fractional Fourier transform.

4 Implementation details

A straightforward implementation of this algorithm does not lead to satisfactory results. Let us investigate the different steps in detail.

4.1 The length of the signal

Note that in the previous derivation it is assumed that the length of the signal is $2N$, hence even. The routine `fracF` is a direct implementation of these formulas and therefore requires the signal length to be even. However, this causes the summations in the previous formulas not to be completely symmetric. The lower bound is indeed $-N$ while the upper bound is $N-1$. To preserve symmetry, it seems much more convenient to assume that the signal length is odd. In that case, the summations can be made completely symmetric. That is why in the routine `fracft`, the signal length is assumed to be odd. In fact the formulas were chosen symmetric in [16] and nonsymmetric in [17, Section 6.7].

4.2 The FFT as a special case

Classically the FFT is defined as a transformation of a vector $[f(0),...,f(N-1)]$, and it is implemented in this way in the matlab routine `fft`. If we compare this with the definition of the fast approximate fractional Fourier transform given above, where the signal is assumed to be $[f(-N),...,f(N-1)]$, then it is obvious that the routine `fft` will not give the same result as an algorithm implementing (1) with $a = 1$. They will only match when the signal and its FFT are cyclically shifted over (approximately) half the signal length. For example, if the length N of the signal is even, then the FFT should be computed as $\mathcal{SFS}f$, where $\mathcal{S}$ is the cyclic shift over $N/2$ samples. For any N, we have implemented this as

```
shft = rem((0:N-1)+fix(N/2),N)+1;
SFSf(shft) = fft(f(shft))/sqrt(N);
```

The square root is needed for the normalization that we introduced in our definition. A similar observation holds for the inverse FFT: `fft` is replaced by `ifft` and the division by the square root is replaced by a multiplication by the square root.

In `fracF` this cyclic shift is avoided by computing the FFT and its inverse using the general routine, so that for $\mathcal{F}^{\pm 1}$ is computed in exactly the same way as for general $\mathcal{F}^{a}$.
In `fracft`, a cyclic shift is implemented which essentially corresponds to what we described above. Since the signal length is assumed to be odd, this operation introduces some asymmetry which may result in a shift of one sample when comparing the results of the two algorithms.

4.3 The reduction of the interval for a

Because we can compute the fractional Fourier transform for a value of a modulo 4, and by using further additivity properties of a, it can be reduced to an interval of length 2, with an additional (inverse) FFT.

The code `fracF` assumes that $-2 \leq a \leq 2$ and reduces this interval further by the following tests

- if $0 < a < 0.5$ then $\mathcal{F}^a = \mathcal{F}\mathcal{F}^{a-1}$
- if $-0.5 < a < 0$ then $\mathcal{F}^a = \mathcal{F}^{-1}\mathcal{F}^{a+1}$
- if $1.5 < a < 2$ then $\mathcal{F}^a = \mathcal{F}\mathcal{F}^{a-1}$
- if $-2 < a < -1.5$ then $\mathcal{F}^a = \mathcal{F}^{-1}\mathcal{F}^{a+1}$.

In this way we are left with values of a satisfying $0.5 < |a| < 1.5$.

In the code `fracft`, a can take any value and it is reduced to the interval $0.5 \leq a \leq 1.5$ by the following tests. First a is replaced by the residual of $a/4$ reducing a to the interval $[0, 4)$. This interval is reduced to $[0, 2]$ by using $\mathcal{F}^a = \mathcal{F}^2\mathcal{F}^{a-2}$ if $a > 2$. Note that $\mathcal{F}^2$ just reverses the input signal and is thus a trivial operation. Next by the following tests, it is further reduced to $0.5 \leq a \leq 1.5$.

- if $a > 1.5$ then $\mathcal{F}^a = \mathcal{F}\mathcal{F}^{a-1}$

- if $a < 0.5$ then $\mathcal{F}^a = \mathcal{F}^{-1}\mathcal{F}^{a+1}$.

Both approaches are equally good, although we think the second one is a bit simpler. Besides this reduction, the special cases of integer a, i.e. $a \in \{0, 1, 2, 3\}$ can be handled directly, and do not need the complicated computation.

In both cases one has to take care that if we compute $\mathcal{F}^a f$ for a ranging over the interval $[0\ , 4]$, then, depending on further implementation details, it may happen that for example the transition from a just smaller than 1.5 and a just larger than 1.5 is not very smooth. The reason being that just before $a = 1.5$, the transformation $\mathcal{F}^a$ is computed for the srcinal signal, while just after $a = 1.5$, the transformation $\mathcal{F}^{a-1}$ is computed for the FFT of the signal. And since we are dealing with a finite number of samples, and not with a continuous signal, all the computations (including the FFT) are only approximations of the actual transformation one wants to perform. Similar problems may arise for the transition from a value of a close to an integer to the integer value itself. That is probably the reason why in the code `fracF`, the FFT and inverse FFT operations are performed via the general routine for the FrFT. Thus actually using the formula (1) with $\alpha = \pm\pi/2$:

$$f_{\pm 1}(\xi_k) \approx \frac{C_{\pm\pi/2}}{2\Delta} e^{-i\pi x_k^2} \sum_{l=-N}^{N-1} e^{\pm i\pi x_{k-l}^2} e^{-i\pi x_l^2} f(x_l).$$

For a value of $a = 2.0001$, and $f = \sin(x)$ sampled in the interval $[0\ , \pi]$, the routine `fracF` gives an irregular shaped answer, while `fracft` produces a smooth answer that is approximately equal to the reversed signal $\mathcal{F}^2 f$. The reason is that $\mathcal{F}^{2.0001} f$ is computed as $\mathcal{F}^{1.0001}[\mathcal{F} f]$ and the FFT that is involved here is computed by the general routine instead of calling the built in routine `fft`.

4.4 The core of the routine

The heart of the routine consists of three steps:

- multiplication of f with a chirp function
- this result is convolved with a chirp (multiply their Fourier transforms)
- multiply with a chirp.

Thus if E_c represents the chirp function e^{icx^2}, then the approximation (1) can be written as

$$\mathcal{F}^a f \approx \frac{C_\alpha}{2\Delta}\ E_c\ \mathcal{F}^{-1}\{\mathcal{F}[E_d f] \cdot \mathcal{F}[E_c f]\} \tag{2}$$

where $c = \cot\ \alpha - \csc \alpha = -\tan(\alpha/2)$ and $d = \csc\ \alpha$. However, a direct implementation of this technique does not lead to accurate results. As explained in the appendix A of [16], the bandwidth after chirp multiplication can be doubled, so that we need to double the number of samples to avoid aliassing. A similar argument holds for the convolution. Therefore both routines `fracF` and `fracft` replace the signal f of length N by a longer signal obtained by interpolation.

In `fracF` if is assumed that f has an even length and using sinc interpolati on, its length is doubled. By padding it before and after by N zeros, it has length $4\ N$.

In `fracft`, it is assumed that f has an odd length and again using sinc interpolation, inserting $p-1$ values in between two successive sample values, it gets length $pN-p+1$. The author has chosen $p=3$. The result is padded with $N-1$ zeros before and after which results in a signal of length $5N-4$.

4.5 Subsampling of the result

Since the result has to be a signal of the same length as the srcinal signal, the long signal has to be subsampled. The reason for expanding the signal and padding it with zeros was mainly because the convolution of g and h in the formula (1) will not be contaminated by boundary effects in the middle of the convolution signal.

In `fracF`, the signals to be convolved are padded with zeros to a length that is the nearest power of 2, the convolution is then computed as the inverse FFT of the product of the FFTs of the two signals. As a result, the middle coefficients are selected.

In `fracft` however, the computation is more subtle. The main operation is a call of the matlab routine `lconv`, which does the same as described above, but before the convolution is computed, the signal is once more interpolated. The number of interpolation points inserted in between two values depends on the value of a in the formula (2), thus on the value of a. The subroutine is written with more general applications in mind because in our situation there is no interpolation necessary, because for all values of $a \in [0.5, 1.5]$ the formula always finds that the number of interpolation points needed is 0.

4.6 The interpolation algorithm

The interpolation can be performed in several ways. Sinc interpolation is popular in these applications. The sinc interpolant of f for the interpolation points $\{x_k = k/2\Delta : k = -N,...,N-1\}$, (here $\Delta = \sqrt{N}$) is given by

$$y(x) = \sum_{k=-N}^{N-1} f(x_k)\mathrm{sinc}(2\Delta(x - x_k)).$$

Thus it can be computed by either a convolution of the signal with a sinc function (as in `fracft`) or it can be applied explicitly using an FFT and an inverse FFT (as in `fracF`). In both cases it requires $O(M\log M)$ operations if the signal has length M.

If M is large, it might seem to be faster to use Lagrange interpolation. Indeed, assuming we want to insert one interpolation point in between two successive values, we can use an interpolating polynomial in $\{f(x_{k-1}), f(x_k), f(x_{k+1}), f(x_{k+2})\}$ and evaluate it in $x_{1/2}$, to get an interpolating value in $f(x^{k+1/2})$. For the boundaries, i.e., for $k = 1$ or $k = N-2$, we evaluate it in $x^{k-1/2}$, respectively in $x_{k+3/2}$ to obtain values for $f(x_{3/2})$ and $f(x_{N-1/2})$. This means that we obtain $f(x_{k+1/2})$ as

$$[-f(x_{k-1}) + 9f(x_k) + 9f(x_{k+1}) - f(x_{k+2})]/16 \text{ if } k = 2,...,N-3$$

while

$$f(x_{3/2}) = [5f(x_1) + 15f(x_2) - 5f(x_3) + f(x_4)]/16$$

and

$$f(x_{N-1/2}) = [f(x_{N-3}) - 5f(x_{N-2}) + 15f(x_{N-1}) + 5f(x_N)]/16.$$

Which is only $O(4M)$ operations. This would mean that if $M > 2^4$, the Lagrange interpolation is faster. However, matlab has a precompiled implementation of the FFT and the inverse FFT, which makes it faster, and some computer experiments revealed that sinc interpolation performed comparably, if not faster than the explicit Lagrange code, even when M was quite large.

5 Modification of the algorithm, allowing general a and of general length

From our tests on several examples, it turned out that `fracft` gives slightly better results than `fracF`. The problem with `fracft` is that only signals with odd length are allowed. The routine `fracF` allows only signals of even length and restricts the value of a to $[-2, 2]$. To write a general algorithm allowing a general a and signals of arbitrary length we did a slight rewriting of `fracft`.

5.1 Reduction of the interval for a

Here we followed the second strategy of section 4.3 and reduced the interval first to $[0, 4)$ and subsequently to $[0.5, 1.5)$ avoiding unnecessary transformations for the special cases $a \in \{0, 1, 2, 3\}$. For this reduction we also had to take care that the FFT was performed with the necessary cyclic shifts as explained in section 4.2.

5.2 The interpolation algorithm

Because we did not see a considerable improvement with the oversampling factor $p = 3$ instead of the factor $p = 2$ (interpolate 2 instead of 1 interpolation point), we took $P = 2$ for the interpolation. We have already mentioned that although theoretically Lagrange interpolation should be faster than sinc interpolation, it is slower in practice. Therefore Lagrange interpolation seems not advisable. However, there is yet another reason why Lagrange interpolation is not the best choice. We illustrate this with an example. Suppose we want to compute $\mathcal{F}^{0.0001}(f)$, where f is the vector containing the samples of e^{ix} computed in the interval $x \in [0, 2\pi]$ with steps 0.01. Recall that this is computed as $\mathcal{F}^{1.0001}(\mathcal{F}^{-1}(f))$. Thus the first step is to compute $f_{-1} = \mathcal{F}^{-1}(f)$ which should be a vector containing the samples of a delta-function. Next, the general routine is applied to compute the transform $f_{0.0001} = \mathcal{F}^{1.0001}(f_{-1})$. This is approximately the inverse transform, and one would expect the absolute value of $f_{0.0001}$ to be approximately equal to the constant function 1. In this process, f_{-1} is upsampled by interpolation. When Lagrange interpolation is used, the nearly delta-function f_{-1} is interpolated well and it remains an approximate delta-function. When sinc interpolation is used however, the energy is somewhat leaking and the transform looks like in Figure 1. The result is that when Lagrange interpolation is used (and hence the delta-function

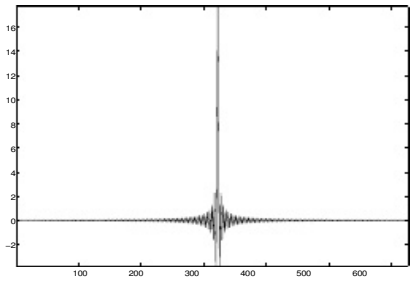

Fig. 1. sinc interpolation of a delta-function

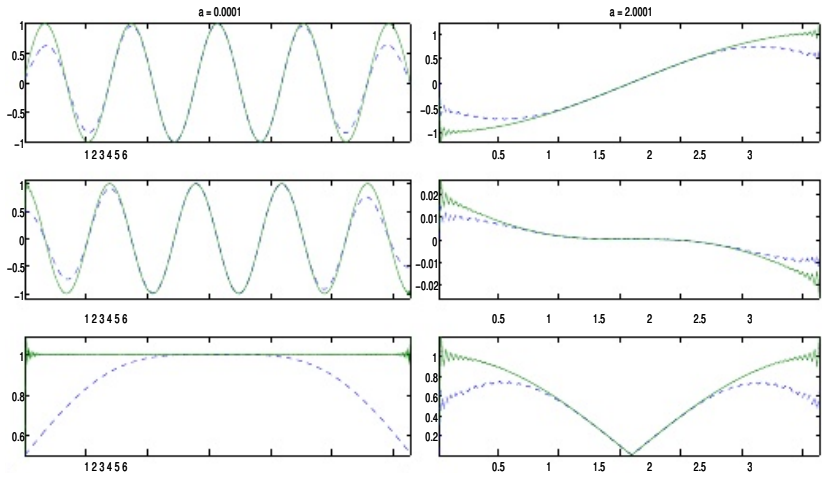

Fig. 2. Real part, imaginary part and absolute value of $\mathcal{F}^{0.0001}(e^{-i(4.5x+\pi/2)})$, $x \in [0, 2\pi]$ (left) and of $\mathcal{F}^{2.0001}(\cos(x))$, $x \in [0, \pi]$ (right). Solid line for sinc interpolation, dashed line for Lagrange interpolation.

is kept better concentrated in the middle during the interpolation process), then the fractional transform will also concentrate most of its energy in the middle and deteriorate near the boundary of the interval as is clearly seen in Figure 2.

Since the special cases $a \in \{0, 1, 2, 3\}$ are treated before the interpolation, we do not see the oscillating effect for those special values. In the code `fracF` however, the FFT and inverse FFT is computed using the general routine for the fractional Fourier transform which results in a transform $\mathcal{F}^2(\cos(x))$ that looks much like the transform using sinc interpolation that is shown on the right of Figure 2, whereas this should be simply obtained as the srcinal signal in reversed order.

6 Discrete fractional Fourier transform

For the definition and the algorithm of the discrete fractional Fourier transform, we refer to [7,8,10]. The main point is to construct the eigenvalue decomposition of the discrete Fourier transform matrix F. The eigenvectors are discrete analogs of the Gauss-Hermite eigenfunctions of the continuous transform. In this approximation process, first order approximations or higher order approximations are possible. It can be shown that as the length N of the signal goes to infinity, the discrete eigenvectors are sample values of the continuous eigenfunctions. However, since there will

be finitely many eigenvectors, only a finite number of eigenvectors can be constructed of orders $0, 1, ..., N-1$, and as the order is closer to N, the approximation becomes worse.

Since the eigenvalues of the $N \times N$ DFT matrix F are known to be $\{(-i)^n : n = 0, ..., N-1\}$ if N is even and $\{(-i)^n : n = 0, ..., N-2, N\}$ if N is odd, the diagonal part in the matrix for the discrete fractional Fourier transform $F^a = E\Lambda^a E^H$ is readily computed. Thus the remaining problem is to compute the matrix E of eigenvectors.

These eigenvectors should approximate the Gauss-Hermite functions

$$\psi_n(x) = \frac{2^{1/4}}{\sqrt{2^n n!}} H_n(\sqrt{2\pi}\, x) e^{-\pi x^2}$$

with $H_n(x)$ the nth Hermite polynomial which satisfies

$$H_0 = 1, \quad H_1(x) = 2x, \quad H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x), \ n = 1, 2, ...$$

Since there are only 4 different eigenvalues with an appropriate multiplicity, the choice of the eigenvectors is not unique. We do want them to be orthogonal, and besides that, it is natural to construct the eigenvectors such that they have the same symmetry properties as the corresponding Gauss-Hermite functions. Because $\psi_n(x)$ is even for n even and odd for n odd, also the eigenvector E_n corresponding to the eigenfunction for the eigenvalue $(-i)^n$ should be even or odd depending on n being even or odd. The ingenious trick that was developed in this context is to design a real symmetric matrix H with distinct eigenvalues (hence with orthogonal eigenvectors) that commutes with F (hence whose eigenvectors coincide with the eigenvectors of F).

For the background of this construction we refer to the cited references [7,8,19]. The result is that, given the symmetry properties of the eigenvectors, it can be reduced to a block diagonal, with one block for the even eigenvectors and one block for the odd eigenvectors. We have

$$VHV^T = \begin{bmatrix} Ev & \\ & Od \end{bmatrix}$$

where

$$V = \frac{1}{\sqrt{2}} \left[\begin{array}{c|cc} \sqrt{2} & & \\ \hline & I_r & J_r \\ & J_r & -I_r \end{array}\right], \quad r = (N-1)/2, \quad \text{if } N \text{ is odd} \qquad (3)$$

and

$$V = \frac{1}{\sqrt{2}} \left[\begin{array}{c|cc|c} \sqrt{2} & & & \\ \hline & I_r & J_r & \\ & 1 & & \\ & J_r & I_r & \\ \hline & & - & \end{array}\right], \quad r = (N-2)/2, \quad \text{if } N \text{ is even}. \tag{4}$$

I_r is the $r \times r$ unit matrix (1's on the main diagonal) and J_r is the $r \times r$ anti-unit matrix (with 1's on the main anti-diagonal). Thus we have to compute the eigenvalue decompositions of the symmetric matrices Ev and Od

$$Ev = V_e \Lambda_e V_e^T \quad \text{and} \quad Od = V_o \Lambda_o V_o^T,$$

so that

$$EHE^T = \left[\begin{array}{c|c} \Lambda_e & \\ \hline & \Lambda_o \end{array}\right], \quad E = V \left[\begin{array}{c|c} V_e & \\ \hline & V_o \end{array}\right]. \tag{5}$$

In other words the columns of E are the eigenvectors, which have the desired symmetry properties by construction. The first ones are the evens, the trailing ones are the odds. It remains to interlace them appropriately.

It remains to define this mysterious matrix H. A theoretical analysis [8] shows that H is the discrete equivalent of the operator $\mathcal{H} = \pi(\mathcal{U}^2 + \mathcal{D}^2)$ where $\mathcal{D}$ is the differentiation operator $(\mathcal{D}f)(x) = (i2\pi)^{-1}\mathrm{d}f(x)/\mathrm{d}x$, and $\mathcal{U}$ is the shift operator ($\mathcal{U}f)(x) = xf(x)$, which may be written as $\mathcal{U} = \mathcal{F}\mathcal{D}\mathcal{F}^{-1}$.

An approximation for the second derivative can be given by the second order central difference operator (indices are taken modulo N)

$$\mathcal{D}^2 \approx \delta^2 = \mathcal{S}^{-1} - 2I + \mathcal{S}, \quad \mathcal{S}f\,(k) = f(k+1), \quad k = 0,...,N-1.$$

On the other hand, the time domain shift $\mathcal{U}^2 = \mathcal{F}\mathcal{D}^2\mathcal{F}^{-1}$ corresponds to a frequency domain multiplication, which, based on the same approximation of $\mathcal{D}^2$ results in a multiplication operator

$$\mathcal{U}^2 \approx F\delta^2 F^{-1} = M^{-1} - 2I + M, \quad Mf\,(k) = e^{ik\frac{2\pi}{N}} f(k), k = 0,...,N-1.$$

These add up to give the approximation for H as follows

$$(Hf)(k) \approx f(k-1) + 2[\cos(k\frac{2\pi}{N}) - 2]f(k) + f(k+1),$$

in other words

$$H = \begin{bmatrix} 2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & 2\cos(\frac{2\pi}{N}) & 1 & \cdots & 0 & 0 \\ 0 & 1 & 2\cos(2\frac{2\pi}{N}) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 0 & 0 & \cdots & 1 & 2\cos((N-1)\frac{2\pi}{N}) \end{bmatrix} - 4I_N$$

However, it is possible to get better approximations for $\mathcal{D}^2$ (and hence also for $\mathcal{U}^2$). It is shown in [7, p.58] that higher order approximations are given by

$$\mathcal{D}^2 \approx \sum_{p=1}^{m} (-1)^{p-1} \frac{[(p-1)!]^2}{(2p)!} (\delta^2)^p.$$

Note that $(\delta^2)^p = d_p(S)$ where d_p is the trigonometric polynomials $d_p(x) = (x - 2 + x^{-1})^p$. The coefficients of d_p can be computed by p successive convolutions of the vector $[1, -2, 1]$. This $d_p(S)$ is represented by a (symmetric) circulant matrix C_p in the sense that the diagonal elements of C_p correspond to the constant term c_0 in the polynomial $d_p(x)$.

Similarly, the operator approximating $\mathcal{U}^2$ is the matrix

$$\mathcal{U}^2 \approx \sum_{p=1}^{m} (-1)^{p-1} \frac{[(p-1)!]^2}{(2p)!} F(\delta^2)^p F^{-1}.$$

Note that $F(\delta^2)^p F^{-1} = d_p(M)$ which is a diagonal matrix whose kth element is the real part of the DFT of the coefficients of the polynomial d_p. So we may conclude that H is represented by the matrix

$$H = \sum_{p=1}^{m} (-1)^{p-1} \frac{[(p-1)!]^2}{(2p)!} (\hat{C}_p + \hat{D}_p + c_0 I_N)$$

where $\hat{C}_p$ is the circulant matrix C_p whose diagonal is removed (and written separately as $c_0 I_N$) and $\hat{D}_p$ is the diagonal matrix whose elements are given by Re($FFT(d_p)$). Since we are interested in the eigenvectors of H, and the constant diagonal $c_0 I_N$ will influence the eigenvalues, but not the eigenvectors, it can be removed for the computations and we only take into account the matrix

$$\hat{H} = \sum_{p=1}^{m} (-1)^{p-1} \frac{[(p-1)!]^2}{(2p)!} (\hat{C}_p + \hat{D}_p). \tag{6}$$

Since the previous formula with m terms, corresponds to a finite difference approximation of the derivative up to the order h^{2m}, it is said to be of order $2m$.

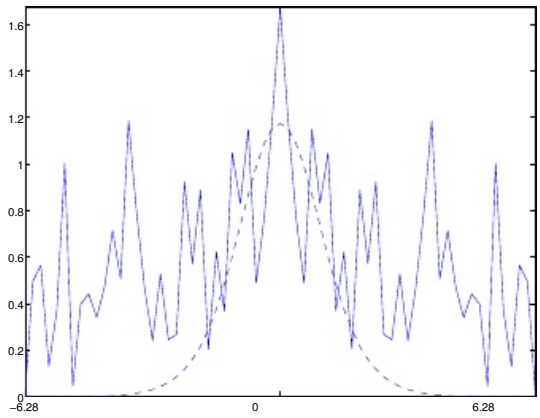

Fig. 3. The absolute value of the discrete fractional Fourier transform $F^{0.5}f$ where f contains 65 equidistant samples in $[-2\pi, 2\pi]$ for the function $\exp(-x^2)$. In solid line for $F^{0.5}$ computed by matlab's built in power routine; in dashed line when the routine described above is used.

The result is an algorithm that computes the eigenvectors of the DFT matrix. It can be summarized as follows (see [8, p. 212]).

(1) Compute the matrix $\hat{H}(6)$ for some order m
(2) Compute the transformation matrix V (3,4)
(3) Compute blocks Ev and Od from $V\hat{H}V^T$
(4) Compute the eigenvectors V_e of Ev and the eigenvectors V_o of Od
(5) Transform these with the matrix V to the eigenvectors in E (5)
(6) Interlace the eigenvectors to correspond to the ordering of the eigenvalues $(-i)^n$.
(7) Compute the matrix $F^a = E\Lambda^a E^T$.

Note: Of course the most simple way of computing the discrete fractional Fourier transform is by putting all the burden upon matlab. We can set up the matrix F and just write `fa=F^a*f`, letting matlab do the computation of F^a. Matlab claims that F^a is computed via eigenvalues and eigenvectors, but does not give any details. Because the method is a built-in routine for matlab, it is relatively fast for small matrices. But the chance that the eigenvectors and the powers of the eigenvalues are computed in exactly the same way as the ones that were chosen here for our implementations is practically zero. So, just writing `F^a` in matlab will almost certainly not generate the particular way in which this has been computed in the algorithms we discussed. A simple test shows that is indeed the case and completely different transforms are computed for the same data. And even if the natlab choice were taken as the definition for the discrete fractional Fourier transform, then for larger matrices, computation time becomes totally unacceptable. On figure 3 we see the absolute value of the discrete fractional Fourier transform for $\exp(-x^2)$ with 65 equidistant samples in the interval $[-2\pi, 2\pi]$. The discrete transform for $a = 0.5$ is computed. In solid line with the matlab functionality, and in dashed line using the routine for which the implementation details are given in the next section.

7 Implementation aspects

We consider here the only freely available matlab code for the construction of F^a that we know of. It is described in [7,8,17] and can be found on the web [6]. This code is called `dFRT`.

First of all, to compare the result of the discrete fractional Fourier transform, we multiply the given vector f with the transform matrix SF^aS where S represents here the cyclic shift matrix that can be implemented as described in section 4.2.

To compute the discrete fractional Fourier transform, we have to compute the transformation matrix as described in the previous section, and then multiply the given vector with this matrix. The computational complexity is $O(n^2)$, which is higher than for the fast approximate fractional Fourier transform. The larger part of the computation time goes to the construction of the eigenvectors. These eigenvectors depend only on the size N and the order of approximation used. Thus if we want to compute the discrete fractional Fourier transform F^af for a sequence of values of a, then we have to compute the vectors E only once because $F^a = E\Lambda^a E^T$. In the routine `dFRT` this is solved in an ingenious way. The matrix of eigenvectors E and the approximation order p are stored as global variables. If the new size N of f and the new approximation order are equal to the size of the global matrix E and the global order p respectively, then the global matrix E is used again and is not recomputed.

When very high orders of approximation are used, like say $m = N/2$ and N large, then the computation of the factor $(k!)^2/(2k)!$ may cause overflow when the results $(k!)^2$ and $(2k)!$ are computed separately. That can be avoided when this coefficient is evaluated as

$$\frac{1}{1}\cdot\frac{1}{2}\cdot\frac{2}{3}\cdot\frac{2}{4}\cdots\frac{k}{2k-1}\cdot\frac{k}{2k}.$$

However, as the experiments show, taking these very high orders does not pay the effort.

The rest of the algorithm is a straightforward implementation of the method described in the previous section. In `dFRT`, some of the steps are implemented in different function subroutines which are mostly avoided in our version. We also use the matlab `toeplitz` to construct the circulant matrices.

Also for the computation of the eigenvalue decomposition of the blocks Ev and Od, the matlab routine `eig` is used. This returns eigenvalues (and hence also the eigenvectors) in reverse order of what is needed for the matrix E. A simple `fliplr` will place the vectors in the correct order. The interlacing operation to give the ordered columns of the matrix E can be done in the first place on the indices as well.

One final remark. If you want to compute the DFRFT as $f_a = F^af = E\Lambda^a E^T f$, then the mutiplication of the $N \times N$ matrix F^a with the vector f requires $O(N^2)$ operations. However, computing $F^a = E\Lambda^a E^T$, given E and Λ^a requires $O(N^3)$ operations. Thus if only one DFRFT has to be computed, it is more efficient to compute $f_a = E(\Lambda^a(E^T f))$. Multiplication with E^T requires $O(N^2)$ operations, multiplication with Λ^a another $O(N)$ and finally the multiplication with E is again $O(N^2)$, which is cheaper than first evaluating F^a.

8 Discrete vs. continuous eigenvectors

One way of measuring how well the discrete transform approximates the continuous transform is by comparing the continuous Gauss-Hermite functions ψ_n with the corresponding eigenvectors of F. A careful mathematical analysis for which we refer to the literature [7,8,18,17] reveals that when ordered appropriately, the interlacing in step 6 of our algorithm in section 6 corresponds to

ordering the eigenvectors according to their number of zero crossings. That implies for example that the evens and odds will interlace. However, in view of the size of the blocks Ev and Od, it turns out that for N even, there is no eigenvector with $N-1$ zero crossings and for N odd, there is no eigenvector with N zero crossings. Since the Gauss-Hermite function ψ_n has exactly n zero crossings, it is clear which eigenvector should approximate which ψ_n. More precisely, one can prove the following

Theorem 5 *Let ψ_n be the Gauss-Hermite functions and let E_n be the eigenvector with n zero crossings for the $N \times N$ discrete Fourier transform matrix F.*
For N even, define the vector with $N+1$ entries as

$$e_n = [E_n(N/2+2),...,E_n(N), E_n(1),...,E_n(N/2+2)]^T.$$

Then, with proper normalization, it will contain approximations of the sample values in the vector

$$\Psi_n = [\psi_n(x_k) : k = -N/2,...,N/2]^T, \quad x_k = k\sqrt{2\pi/(N+1)}.$$

For N odd, define the vector with N entries

$$e^n = [E^N((N+3)/2),...,E^N(N), E^N(1),...,E^N((N+1)/2)]^T.$$

Then, with proper normalization, it will contain approximations of the sample values in the vector

$$\Psi_n = [\psi_n(x_k) : k = -(N-1)/2,..., (N-1)/2]^T, \quad x_k = k\sqrt{2\pi/N}.$$

The term approximation means that the vectors e_n will converge to the vectors Ψ_n as $N \to \infty$.

For a more precise treatment see [4,3].
Note that the eigenvectors are supposed to be normalized, but even then, they are only defined up to a sign. For the definition of the discrete fractional Fourier transform, this sign is of no importance because $F^a = E\Lambda^a E^T$, and the sign does not matter.

The approximation becomes worse as n approaches N. We have illustrated this in figure 4. The functions ψ_n are plotted in solid lines, the sample values Ψ_n are indicated by a cross and the approximate vectors e_n are plotted with circles (and joined by a dashed line). In this figure, the approximation order used for the matrix H is 2 ($m = 1$). For a larger N, the order does make a difference. For example in figure 5 we have plotted the case $N = 100$, and $n = 30$, on the left for $m = 1$ and on the right for $m = 20$. It can be observed that the eigenvector of F has extreme values for $m = 1$ at the beginning and the end, while for $m = 20$, these values are pulled towards zero, so that they give better approximants.

9 Other definitions of the DFRFT

Many other definitions of the discrete fractional Fourier transform do exist [2,21,22,5], but they have several theoretical disadvantages. For example the fast approximate fractional Fourier transform that we have discussed before is not a unitary operator like the discrete transform as defined above is.

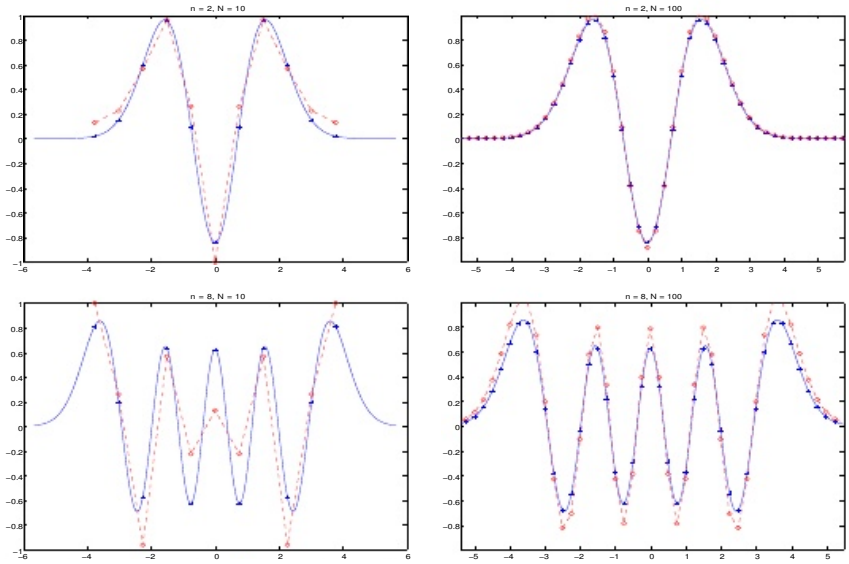

Fig. 4. The continuous Gauss-Hermite functions (solid line) sampled at equidistant points (+) and the eigenvectors (o) of the discrete FRFT matrix. On the left for 10 samples, on the right for 100 samples. On top for $\psi 2$, at the bottom for $\psi 8$.

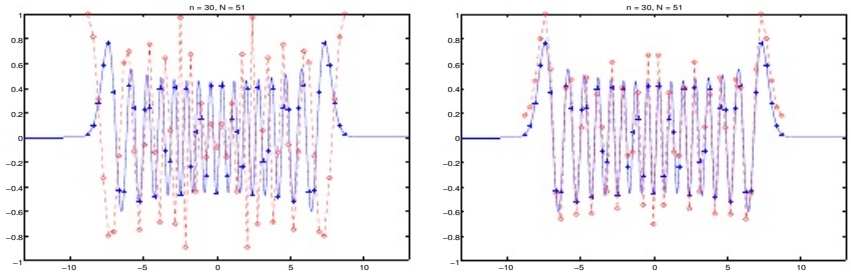

Fig. 5. The continuous Gauss-Hermite functions (solid line) sampled at equidistant points (+) and the eigenvectors (o) of the discrete FRFT matrix for 100 samples, on the right for 100 samples. On the left for $m = 1$, and on the right for $m = 20$.

Another approach similar to the previous one is described in [18]. The idea is the following. Since the eigenvectors of the discrete Fourier transform matrix F are approximated by the samples of the Gauss-Hermite eigenfunctions, it is proposed here that the vectors Ψ_n that were introduced in the previous sections are projected onto the corresponding eigenspaces. There are only 4 eigenspaces: $\mathcal{E}_k,\ k = 0, 1, 2, 3$ corresponding to the eigenvalues $(-i)^k,\ k = 0, 1, 2, 3$. Because the eigenspaces corresponding to different eigenvalues will be orthogonal, it suffices to orthogonalize the projections within their eigenspaces. Some special care has to be taken in the case of a signal length N that is even, because then there is some jump in the sequence of eigenvalues, because the eigenvalues are $(-i)^k,\ k = 0,...,N-2, N-1$ for N odd and $(-i)^k,\ k = 0,...,N-2, N$ for N even. So we use the eigenvectors E_k with k zero crossings as defined in the routine `dFRFT` and the vectors of

corresponding sample values Ψ_k for example for $N = 10$, the spaces are spanned by the vectors

$\mathcal{E}_0$	E_0, E_4, E_8
$\mathcal{E}_1$	E_1, E_5
$\mathcal{E}_2$	E_2, E_6, E_{10}
$_3$	E_3, E_7

$\mathcal{E}$

On the other hand, the following vectors are to be projected on the eigenspaces indicated

$\mathcal{E}_0$	Ψ_0, Ψ_4, Ψ_8
$\mathcal{E}_1$	Ψ_1, Ψ_5
$\mathcal{E}_2$	$\Psi_2, \Psi_6, \Psi_{10}$
$\mathcal{E}_3$	Ψ_3, Ψ_7

Our implementation uses similar tricks as in the case of `dFRFT`. We remark that the straightforward computation of the Gauss-Hermite functions in their unnormalized form easily leads to overflow since the hermite polynomials are growing very fast. Therefore a renormalization was implemented to avoid overflow and underflow.

Clearly, the extra projections and orthogonalizations needed require some extra computer time. On the other hand, the resulting eigenvectors that are used are much closer to the sample values of the continuous Gauss-Hermite functions. We give in figure 6 the analogs of figure 4, which shows that there is a better correspondence.

Also the MIMO system implementation discussed in [10] is just a general setting to split the signal into M disjunct blocks and process these blocks in parallel by a filter bank so that on a multiprocessor machine, the transform is computed faster, but this does not essentially change the result.

Yet another implementation is proposed in [24]. However, this requires the precomputation of all the fractional transforms $x_n = F^{bn} f$ with $b = 4/N$ if the signal has length N. This is only justified in very particular applications, but rather inefficient for a general purpose routine.

10 Discrete vs. fast approximate transform

In figure 7 we see the absolute value of the fractional Fourier transform of $\cos(\quad x)$ where $x = 0 : 0.02 : 2\ \pi$. The discrete transform is drawn in solid line, the fast approximate transform is drawn in dashed line. This figure illustrates a general observation: the fast approximate transform oscillates more near the boundaries, while the discrete transform oscillates more in the middle of the interval if the approximation order is too small (i.e.: if $\quad \pi n$ is small). However for a larger %, the approximation of the fast approximate transform is much better, as can be observed in the right plot. As could have been predicted from the previous section, the approximation of the Gauss-Hermite functions by the eigenvectors of the discrete transform matrix not being very good near the boundaries when the number of zero crossings is high, it is also generally true that for

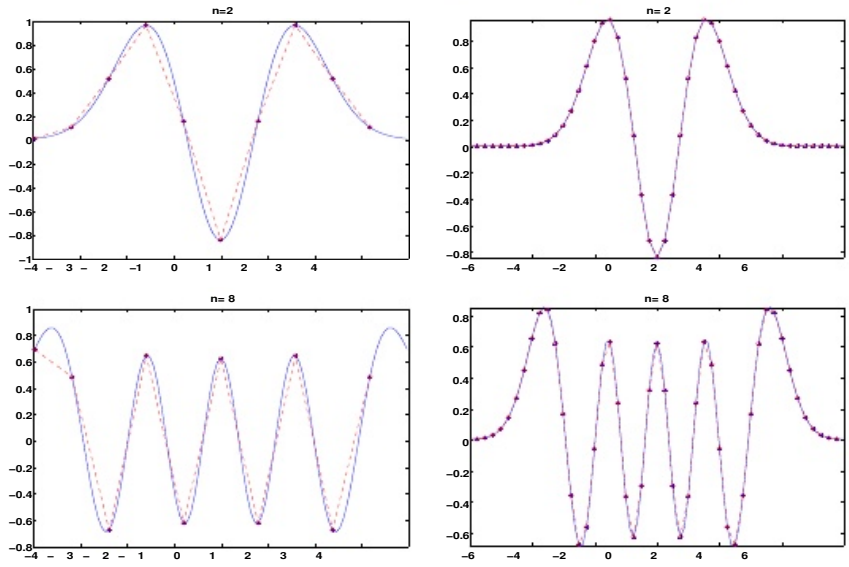

Fig. 6. The continuous Gauss-Hermite functions (solid line) sampled at equidistant points (+) and the eigenvectors (o) of the discrete FRFT matrix using `DFPei`. On the left for 10 samples, on the right for 100 samples. On top for $\psi 2$, at the bottom for $\psi 8$.

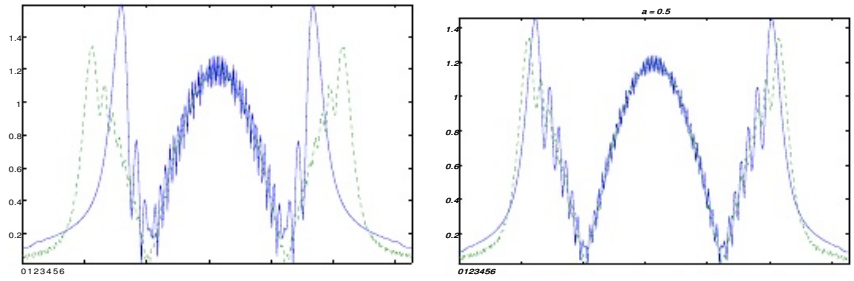

Fig. 7. Absolute value of $\mathcal{F}^{0.5}(\cos(x))$ (dashed) and of $F^{0.5}(\cos(x))$ (solid), where $x = 0:0\ .02:2\,\pi$. On the left with $m = 2$, on the right with $m = 30$.

most values of a, the discrete fractional Fourier transform does not approximate very well the fast approximate fractional Fourier transform near the boundaries. If N is large, then a high order of approximation, i.e., taking relatively large values of m will help.

The discrete transform algorithm is considerably slower than the fast approximate transform. The computation of the eigenvectors is the most time consuming. Therefore the trick of saving the eigenvectors avoiding recomputation for different values of a when N and m remain the same is a considerable saving of computer time.

A plot of the DFRFT with the Pei or Candan algorithms gives only minor differences. See figure 8.

Note the irregular behaviour of the Pei approximant near the boundary.

Since the discrete fractional sine and cosine transform algorithms as described in [20] are essentially obtained by taking as eigenvectors half of the even or odd eigenvectors of the discrete fractional Fourier transform matrix that has twice the size of the signal, it should be clear that implementa-

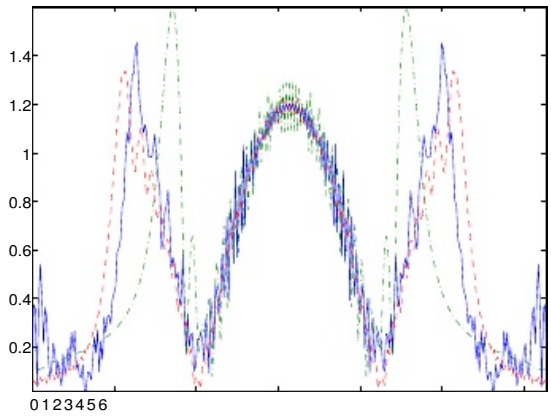

Fig. 8. The fast approximate fractional Fourier transform $\mathcal{F}^{0.5}(\cos x)$ where $x = 0 : 0 \quad .02 : 2\ \pi$, i.e., $N = 315$ (dash-dotted line), and the discrete transform corresponding to the `DFpei` code (solid line) and the discrete transform corresponding to the code `Disfrft` (dashed line). The order of approximation used is $p = N/2$.

Fig. 9. On the left, an image that is contaminated by a chirp which has sweep rate 0.6 in the x-direction and 0.3 in the y-direction. When the separable two-dimensional discrete fractional Fourier transform is applied with the appropriate orders, then the chirp will be transformed in a delta-function. It is then easily identified and removed. After back transforming the clean image is reconstructed as on the right.

tions for these transforms are directly obtained from the implementation of the discrete fractional Fourier transform that we have discussed.

11 Two-dimensional transform

Although there exists a definition of an non-separable two-dimensional fractional Fourier transform, the easiest application is a tensor product type of two-dimensional fractional Fourier transform by applying subsequently the one-dimensional transform to the rows and the columns of the image. As an example, we superposed a chirp noise on an image as can be seen in the left of figure 9. After the appropriate one-dimensional discrete fractional Fourier transforms on the rows and the columns, the chirp is transformed to a delta-function. This can be seen in figure 10, where we have plotted a mesh for the rows and columns in the neighborhood of the delta-function. Even though the noise is hardly seen in the image, the peak clearly stands out and this identifies

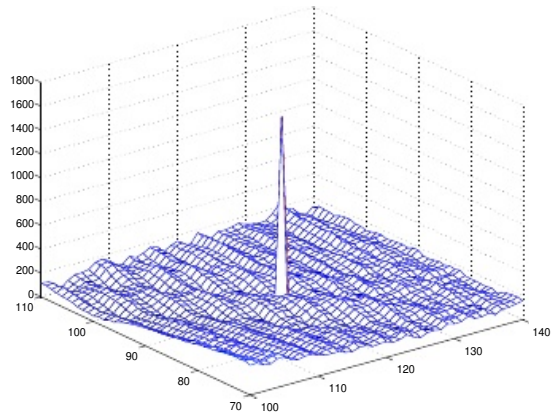

Fig. 10. After the appropriate one-dimensional transforms on the rows and the columns of the image the chirp is transformed into a delta-function. The plot shows the neighborhood of the peak.

that the appropriate transformation has been made. After removal by replacing the peak by the average on the neighboring pixels, the image is back transformed and a clean image is found.

12 Conclusion

We have compared two existing routines for the computation of the fast approximate fractional Fourier transform and one routine for the discrete fractional Fourier transform. We studied in detail the different steps of the implementation and propose our own implementaion as an alternative that overcomes some of the restrictions. We also test an implementation for the discrete fractional Fourier transform, compare the results of the discrete and the fast approximate transforms and describe our own implementation. It is compared with another algorithm of Pei which is baed on orthogonal projections. The extra computational effort did not seem to be worthwhile in general. Adaptations of our implementation for obtaining discrete sine or cosine transforms are easy.

Matlab versions of our implementations are available on the website

`www.cs.kuleuven.ac.be/~nalag/research/software/FRFT/`

References

[1] L.B. Almeida. The fractional Fourier transform and time-frequency representation. *IEEE Trans. Sig. Proc.*, 42:3084–3091, 1994.

[2] N.M. Atakishiyev, L.E. Vicent, and K.B. Wolf. Continuous vs. discrete fracti onal Fourier trans form. *J. Comput. Appl. Math.* , 107:73–95, 1999.

[3] L. Barker. The discrete fractional Fourier transform and Harper's equation. *Mathematika*, 47(1-2):281–297, 2000.

[4] L. Barker, Ç. Candan, T. Hakioğlu, M.A. Kutay, and H.M. Ozaktas. The discrete harmonic oscillator, Harper's equation, and the discrete Fractional Fourier Transfom. *J. Phys. A* , 33:2209–2222, 2000.

[5] G. Cariolaro, T. Erseghe, P. Kraniauskas, and N. Laurenti. Multiplicity of fractional Fourier transforms and their relationships. *IEEE Trans. Sig. Proc.*, 48(1):227–241, 2000.

[6] Ç. Candan. `dFRT`: The discrete fractional Fourier transform, 1996. A matlab program `www.ee.bilkent.edu.tr/~haldun/dFRT.m`

[7] Ç. Candan. *The discrete Fractional Fourier Transform*. MS Thesis, Bilkent University, Ankara, 1998.

[8] C. Candan, M.A. Kutay, and H.M. Ozaktas. The discrete Fractional Fourier Transform. *IEEE Trans. Sig. Proc.*, 48:1329–1337, 2000.

[9] E.U. Condon. Immersion of the Fourier transform in a continuous group of functional transformations. *Proc. National Academy Sciences*, 23:158–164, 1937.

[10] D.F. Huang and B.S. Chen. A multi-input-multi-output system approach for the computation of discrete fractional Fourier transform. *Signal Processing*, 80:1501–1513, 2000.

[11] H. Kober. Wurzeln aus der Hankel- und Fourier und anderen stetigen Transformationen. *Quart. J. Math. Oxford Ser.*, 10:45–49, 1939.

[12] M.A. Kutay. `fracF`: Fast computation of the fractional Fourier transform, 1996. `www.ee.bilkent.edu.tr/~haldun/fracF.m`

[13] A.C. McBride and F.H. Kerr. On Namias's fractional Fourier transforms. *IMA J. Appl. Math.*, 39:159–175, 1987.

[14] V. Namias. The fractional order Fourier transform and its application in quantum mechanics. *J. Inst. Math. Appl.*, 25:241–265, 1980.

[15] J. O'Neill. `DiscreteTFDs`: a collection of matlab files for time-frequency analysis, 1999. `ftp.mathworks.com/pub/contrib/v5/signal/DiscreteTFDs/`

[16] H.M. Ozaktas, M.A. Kutay, and G. Bozdağı. Digital computation of the fractional Fourier transform. *IEEE Trans. Sig. Proc.*, 44:2141–2150, 1996.

[17] H.M. Ozaktas, Z. Zalevsky, and M.A. Kutay. *The fractional Fourier transform*. Wiley, Chichester, 2001.

[18] S.-C. Pei, M.-H. Yeh, and C.-C. Tseng. Discrete fractional Fourier-transform based on orthogonal projections. *IEEE Trans. Sig. Proc.*, 47(5):1335–1348, 1999.

[19] S.C. Pei and M.H. Yeh. Improved discrete fractional Fourier transform. *Optics Letters*, 22:1047–1049, 1997.

[20] S.C. Pei and M.H. Yeh. The discrete fractional cosine and sine transforms. *IEEE Trans. Sig. Proc.*, 49:1198–1207, 2001.

[21] M.S. Richman, T.W. Parks, and R.G. Shenoy. Understanding discrete rotations. In *Proc. IEEE Int. Conf. Acoust. Speech, Signal Process.*, 1997.

[22] B. Santhanam and J.H. McClellan. The discrete rotational Fourier transform. *IEEE Trans. Sig. Proc.*, 44:994–998, 1996.

[23] N. Wiener. Hermitian polynomials and Fourier analysis. *J. Math. Phys.*, 8:70–73, 1929.

[24] M.H. Yeh and S.C. Pei. A method for the discrete fractional Fourier transform computation. *IEEE Trans. Sig. Proc.*, 51(3):889–891, 2003.