# A two-phase implementation of the fractional Fourier transform

*Adhemar Bultheel*

*Report TW 588, March 2011*




Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# A two-phase implementation of the fractional Fourier transform

*Adhemar Bultheel*[*]

*Report TW 588, March 2011*

Department of Computer Science, K.U.Leuven

**Abstract**

We describe the implementation of a two-phase implementation of the filters used in the computation of the fractional Fourier transform. This implementation speeds up the classical code by an average factor from 2 to 4.

**Keywords :** Fractional Fourier transform, polyphase filters, numerical software, matlab.
**MSC :** Primary : 65R10, Secondary : 44-04.

[*]K.U.Leuven, Belgium

# A two-phase implementation of the fractional Fourier transform

A. Bultheel[a]

[a]*Dept. Computer Sc., K.U.Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium*

**Abstract**

We describe the implementation of a two-phase implementation of the filters used in the computation of the fractional Fourier transform. This implementation speeds up the classical code by an average factor from 2 to 4.

*Keywords:* Fractional Fourier transform, polyphase filters, numerical software, matlab.
*2000 MSC:* 65R10, 44-04

## 1. Introduction

The fractional Fourier transform is well known and has many applications. In this short report we shall only be interested in a particular implementation using a polyphase idea for the implementation of the filters involved. Therefore we restrict ourselves to the essentials of the theory. For more details, there is ample literature available. See for example the book [6] or the surveys [3] or [1] and the many references therein.

So we start by giving the definition:

**Definition 1. (fractional Fourier fransform)** The (continuous) fractional Fourier transform operator $\mathscr{F}^a$ depends on a real number $a$ and transforms a function $f$ into the function $f_a = \mathscr{F}^a(f)$ which is defined by an integral transform

$$f_a(\xi) = \int_{-\infty}^{+\infty} K_a(\xi, x) f(x) \mathrm{d}x$$

where the kernel is given by

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

and $C_\alpha$ is a normalizing constant

$$C_\alpha = \frac{-i[\pi \mathrm{sgn}(\sin\alpha)/4 - \alpha/2]}{\sqrt{|\sin\alpha|}}.$$

For $a \in 2\mathbb{Z}$ i.e., $\alpha \in \pi\mathbb{Z}$, limiting values should be taken.

*Email address:* adhemar.bultheel@cs.kuleuven.be (A. Bultheel)

First of all, the transform operation is 4-periodic in $a$ and using the fact that $\mathscr{F}^1$ is the classical Fourier transform and $\mathscr{F}^{-1}$ its inverse, and more generally that $\mathscr{F}^{-a} = (\mathscr{F}^a)^{-1}$, it is possible to reduce the computation of any $\mathscr{F}^a$ to the computation of $\mathscr{F}^a$ with $0.5 \leq a \leq 1.5$ (see e.g. [6] or [2]).

The formula can be conveniently rewritten using the formula

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

to give

$$f_a(\xi) = C_\alpha T_t(\xi) \int_{-\infty}^{+\infty} T_s(\xi - x)[T_t(x) f(x)]\mathrm{d}x$$

where we have denoted a chirp as $T_r(x) = \exp\{-i\pi r x^2\}$, where $r$ takes on the two values $t = -(\cot\alpha + \csc\alpha) = \tan(\alpha/2)$ and $s = -\csc(\alpha)$. It is clear from this formula that the operation can be decomposed into three basic steps:

1. A multiplication with a chirp $T_t$ (in the $x$-domain)
2. A convolution with a chirp $T_s$
3. A multiplication with a chirp $T_t$ (in the transform domain).

This leads to the simple scheme of figure 1.

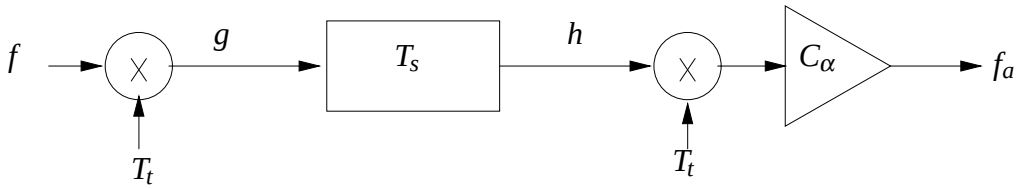



Figure 1: Simple 1-phase scheme for the frft.

## 2. Available software

Several descriptions of algorithms for a digital implementation of this formula are available. For example [5], see also [6] with a matlab implementation available on the site www.ee.bilkent.edu.tr/~haldun/fracF.m. Another one is called `fracft` and is part of the matlab package `DiscreteTFDs` that was compiled by Jeffrey O'Neill in 1999. Currently, version 8.0 is available via the site www.koders.com. These two versions were compared in [2] and another 'the-best-of' implementation called `frft` was made available on the site nalag.cs.kuleuven.be/research/software/FRFT. On the same site also matlab implementation of the discrete fractional Fourier transform are available for algorithms described in the literature.

In this report we want to document a matlab implementation of the polyphase implementation proposed in [7] and first tested in a masters thesis [4]. In the previous implementations signals are upsampled during the computations and downsampling is taking place at the end of the computations. The polyphase, in this case a two-phase, implementation splits up the signals in even and odd samples and tries as much as possible to avoid the computation of samples that are thrown away at the end of the algorithm. In this way it is possible to reduce the computation time.

## 3. Initial steps

The first part of the program is identical to previous implementations, in particular the implementation called `frft` mentioned above. First the length of the input signal is defined and $a$ is defined modulo 4. The `shft` is a shift that places the central sample of the given vector $f$ at the origin. Here is the code.

```
f0 = f(:); N = length(f); sN = sqrt(N);
shft = rem((0:N-1)+fix(N/2),N)+1;
a = mod(a,4);
```

The next chunk deals with particular cases where $a$ is an integer, in which case the fractional Fourier transform can be approximated by an ordinary FFT. The `flipud` is in fact the application of $\mathscr{F}^2$, which just reverses the $x$-axis. Ths scalings with `sN`$=\sqrt{N}$ makes the transform unitary. That corresponds to the code

```
% do special cases
if (a==0), Faf = f0; return; end;
if (a==2), Faf = flipud(f0); return; end;
if (a==1), Faf(shft,1) = fft(f0(shft))/sN; return; end
if (a==3), Faf(shft,1) = ifft(f0(shft))*sN;return; end
```

The next part is the reduction of the interval $a \in [0,4]$ to $a \in [1/2,3/2]$. That is also exactly the same as in the previous implementations and makes use of the rule that $\mathscr{F}^a\mathscr{F}^b = \mathscr{F}^{a+b}$. The corresponding code is

```
% reduce to interval 0.5 < a < 1.5
if (a>2.0), a = a-2; f0 = flipud(f0); end
if (a>1.5), a = a-1; f0(shft,1) = fft(f0(shft))/sN; end
if (a<0.5), a = a+1; f0(shft,1) = ifft(f0(shft))*sN; end
```

The following parts of the algorithm do the actual computation in the generic case. These differ from the previous implementation, and we shall describe the arguments used to make the changes.

## 4. Discretization and sampling frequencies

For the digital implementation of this transform, the functions of the continuous variables $x$ and $\xi$ will have to be replaced by discrete samples and they will be of finite length. The number of samples needed to represent the intermediate signals of the computations properly will depend on their bandwidth. Suppose first that we have given $N$ samples of the signal $f$ in the $x$-interval length $[-\Delta_x/2, \Delta_x/2]$. We also assume that these samples are enough to represent the signal properly, i.e., we assume that its Fourier transform is essentially supported in the interval $[-\Delta_\xi/2, \Delta_\xi/2]$, such

that $\Delta_x/(N+1) = 1/\Delta_\xi$.[1] For simplicity we shall assume that the time-frequency plane has been properly scaled so that we may assume that $\Delta_x = \Delta_\xi = \Delta$, and thus that $N+1 = \Delta^2$. This may require a preprocessing step as explained for example in [5, p. 2143]. So we suppose that the signal $f$ that we want to transform is given in $N$ equidistant points $x_j = j/\Delta$ with[2] $j = -\frac{N-1}{2}, \ldots, \frac{N-1}{2}$ where $\Delta$ is the bandwidth of $f$.

For the intermediate signals appearing during the three steps of the algorithm we shall have to be careful at what sample frequency they need to be represented. I.e., we have to look at the bandwidths of these intermediate signals.

To analyse that we introduce the Wigner distribution of a general signal $f$.

**Definition 2. (Wigner distribution)** The Wigner distribution of $f$ is defined as

$$\mathscr{W}_f(x,\xi) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} f(x+\tau/2)\overline{f(x-\tau/2)} e^{-i\tau\xi} \mathrm{d}\tau.$$

The Wigner distribution gives an idea of the distribution of the energy of the signal in the $(x,\xi)$-plane [6]. By our scaling assumption we can assume that the Wigner distribution is mainly confined to a square $[-\Delta/2, \Delta/2]^2$ in the $(x,\xi)$-plane.

Now note that the Wigner distribution of a product is the convolution of the Wigner distributions of its factors since indeed

$$\begin{aligned}
\mathscr{W}_{gh}(x,\xi) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} g(x+\frac{\tau}{2})\overline{g(x-\frac{\tau}{2})} h(x+\frac{\tau}{2})\overline{h(x-\frac{\tau}{2})} e^{-i\tau\xi} \mathrm{d}\tau \\
&= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} g(x+\frac{\tau}{2})\overline{g(x-\frac{\tau}{2})} h(x+\frac{\sigma}{2})\overline{h(x-\frac{\sigma}{2})} e^{-i\tau\xi} \delta(\sigma-\tau) \mathrm{d}\tau \mathrm{d}\sigma \\
&= \frac{1}{2\pi} \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} g(x+\frac{\tau}{2})\overline{g(x-\frac{\tau}{2})} h(x+\frac{\sigma}{2})\overline{h(x-\frac{\sigma}{2})} e^{-i\tau\xi} e^{-i\rho(\sigma-\tau)} \mathrm{d}\tau \mathrm{d}\sigma \mathrm{d}\rho \\
&= \int_{-\infty}^{+\infty} \left[ \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} g(x+\frac{\tau}{2})\overline{g(x-\frac{\tau}{2})} e^{-i\tau(\xi+\rho)} \mathrm{d}\tau \right] \times \\
&\qquad \left[ \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} h(x+\frac{\sigma}{2})\overline{h(x-\frac{\sigma}{2})} e^{-i\rho\sigma} \mathrm{d}\sigma \right] \mathrm{d}\rho \\
&= \int_{-\infty}^{+\infty} \mathscr{W}_g(x,\xi+\rho)\mathscr{W}_h(x,\rho)\mathrm{d}\rho = \mathscr{W}_g(x,\xi) *_\xi \mathscr{W}_h(x,\xi).
\end{aligned}$$

Let us now look at the effect of a chirp multiplication $T_r(x) f(x)$.
Taking into account that [5]

$$\mathscr{W}_{T_r}(x,\xi) = \delta(\xi - rx),$$

[1] In the literature it is customary to use $N$ where we have written $N+1$. However if we place a sample on the left, on the right and in the middle of the interval, then we have 3 samples that are $\Delta_x/2$ apart to span the interval $[-\Delta_x/2, \Delta_x/2]$. In general one needs $N+1$ samples that are $\Delta_x/N$ apart to span the same interval. In practice $N$ is usually large so that the difference in using $N$ or $N+1$ is often negligible.

[2] With a notation like $j = -j_0, \ldots, j_1$ we shall always mean that the $j$-value jumps with steps of 1. Thus $j = j_0, j_0+1, j_0+2, \ldots, j_1-1, j_1$ where $j_0$ and $j_1$ can be both integer or both half-integer values. A similar convention holds if such an index range is used in summations.

it follows that in the $\xi$-direction the support will be multiplied by $1+|c|$ because of a shear in that direction. See figure 2.

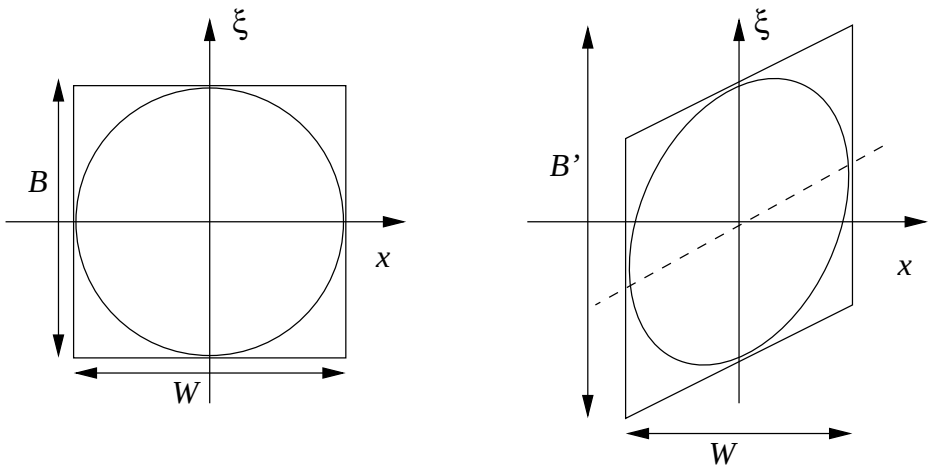



Figure 2: Left: the Wigner distribution of some signal. Right: the Wigner distribution of the signal multiplied by a chirp $T_r(x)$. The dashed line is $\xi = |r|x$. This results in $B' = B(1+|r|)$.

For step one, we multiply $f$ with $T_t$ where $t = -(\cot\alpha + \csc\alpha) = \tan\frac{\alpha}{2} = \tan\frac{a\pi}{4}$. So that the bandwidth $B' = B(1+|t|)$ will be larger than $B$. Since we have restricted $a$ to the range $[1/2, 3/2]$, the amplification factor $1+|t|$ lies in $[\sqrt{2}, 2+\sqrt{2}] \approx [1.41, 3.41]$. Thus we might expect to need an upsampling with a factor 4.

In step 2, the convolution corresponds to a multiplication of the Fourier transforms. In other words the effect described in step 1 is now mirrored. The multiplication with a chirp $T_s$ in the $\xi$-domain will cause a widening of the support in the $x$-domain. The Fourier transform of a chirp with rate $s$ is a chirp with rate $1/s$. In our case $s = \csc\alpha$, thus $1/s = \sin(a\pi/2)$ and because $a \in [1/2, 3/2]$, the amplification factor $1+1/|s| \in [(1+\sqrt{2})/2, 2]$ will be at most 2. Thus this factor of 2 compensates for half of the upsampling by 4, needed in the previous step.

The conclusion is that already in step 1, an upsampling factor of 2 will suffice. In the literature this factor 2 is commonly used. In [5] $a$ was restricted to $[-1/2, 1/2]$, and then $1+|r| \in [1, \sqrt{2}]$, so that doubling the sampling frequency is certainly good enough. However in [6], the authors assume that $a \in [1/2, 3/2]$ like we do here, yet also there the authors propose to double the sampling frequency. This can be justified on the following grounds. The effect of the multiplication with $T_t(x) = \exp\{i\pi x^2(\cot\alpha + \csc\alpha)\}$ can be split into the factors $\exp\{i\pi x^2 \cot\alpha\}$ and $\exp\{i\pi x^2 \csc\alpha\}$ and the latter is compensated by the factor $\exp\{-i\pi x^2 \csc\alpha\}$ of the convolution chirp $T_s(\xi - x) = \exp\{i\pi(\xi - x)^2 \csc\alpha\}$, just like the other quadratic factor $\exp\{-i\pi\xi^2 \csc\alpha\}$ of $T_s(\xi - x)$ cancels against the same factor in the final chirp multiplication $T_t(\xi)$. Thus, looking at the formula

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

it is seen that the $(\csc\alpha)$-part in $T_t$ drops out and it is only $(\cot\alpha)$-part of the chirp multiplication

that gives an amplifying effect. For $a \in [1/2, 3/2]$ that effect is $1+|\cot\alpha| = 1+\cot(a\pi/2) \in [1,2]$, thus of a factor at most 2. This argument can also be found in [7].

Define $g(x) = T_t(x)f(x)$ then we know that we have to compute it at double the frequency of $f(x)$. The missing intermediate values for $f$ can be obtained by sinc interpolation[3]. Set $y_l = l/(2\Delta)$, $l = -(N-1), \ldots, N-1$, then we define

$$f(y_l) = f\left(\frac{l}{2\Delta}\right) = \sum_{j=-\frac{N-1}{2}}^{\frac{N-1}{2}} f(x_j)\mathrm{sinc}\left(2\Delta(y_l - x_j)\right) = \sum_{j=-\frac{N-1}{2}}^{\frac{N-1}{2}} f\left(\frac{j}{\Delta}\right)\mathrm{sinc}\left(l-2j\right).$$

Thus to compute discrete samples of the transform $f_a$ in the points $\xi_k = \frac{k}{2\Delta}$, $k = -N, \ldots, N$, the discretized transform looks as

$$f_a(\xi_k) = \frac{C_\alpha}{2\Delta}T_t(\xi_k)\sum_{l=-(N-1)}^{N-1} T_s\left(\xi_k - y_l\right)g(y_l) = \frac{C_\alpha}{2\Delta}T_t\left(\frac{k}{2\Delta}\right)\sum_{l=-(N-1)}^{N-1} T_s\left(\frac{k-l}{2\Delta}\right)g\left(\frac{l}{2\Delta}\right),$$

However, since we want the transformed signal to have the same length as the original signal, and we have now computed samples for a double length, we shall have to downsample again.

## 5. A polyphase implementation

Since we are upsampling and downsampling during this procedure, it is advisable to just compute those samples that are really needed, so that we shall not have to throw away half the work that has been done. This is exactly what one tries to achieve with a polyphase (in this case a two-phase) implementation of the algorithm as described in [7] and in [4].

For discrete signals, the idea is to split the samples into the even ones and the odd ones and do all the operations on these two parts separately, but taking care that only the necessarily samples are computed. In our notation, it will depend on $N$ being even or odd whether the given samples $f(\frac{k}{\Delta})$ for $k = -\frac{N-1}{2}, \ldots, \frac{N-1}{2}$ will get an index ranging over integer values or half- integer values for $k$. Now imagine that we store these $f(\frac{k}{\Delta})$ values at the even index entries of a vector and that we call these samples the 'even' samples. There are $N$ of them. The samples halfway in between, are those that we need to obtain by interpolation. We shall call them the 'odd' samples. Note that there are $N-1$ odd samples interlacing with the $N$ even samples. The adjectives 'even' and 'odd' can be justified by the fact that they correspond to the values $f(\frac{p}{2\Delta})$ with an even or an odd value for $p$ respectively.

So we have to compute the odd samples by sinc interpolation (i.e., convolution with a sinc function). We denote by $f_0$ the vector with samples $f_0(k) = f(\frac{k}{\Delta}) = f(\frac{2k}{2\Delta})$, $k = -\frac{N-1}{2}, \ldots, \frac{N-1}{2}$. Let $f_1$ be the vector of the odd samples: $f_1(l) = f(y_{2l+1}) = f(\frac{2l+1}{2\Delta})$, $l = -\frac{N-1}{2}, \ldots, \frac{N-3}{2}$. Sinc interpolation gives

$$f_1(l) \;=\; f(y_{2l+1}) = f\left(\frac{2l+1}{2\Delta}\right) = \sum_{j=-\frac{N-1}{2}}^{\frac{N-1}{2}} f\left(\frac{j}{\Delta}\right)\mathrm{sinc}\left(2l+1-2j\right)$$

[3]That sinc interpolation is a better choice than Lagrange interpolation has been illustrated in [3].

$$= \sum_{j=-\frac{N-1}{2}}^{\frac{N-1}{2}} f_0(j)\,\mathrm{sinc}\,(2(l-j)+1),\ \ l=-\frac{N-1}{2},\ldots,\frac{N-3}{2}.$$

This means that we have to evaluate the sinc function in the points $-\frac{2N-3}{2},\ldots,\frac{2N-3}{2}$. The interpolation is then obtained by a discrete convolution, which is suitably obtained as a product of the FFT's followed by an inverse FFT. The convolution doubles the length of the signal, so that we have to take out the middle part, i.e., the $N-1$ samples $N,\ldots,2N-2$. This explains the polyphase sinc interpolation part of the matlab program:

```
f0 = f(:); N = length(f);
...
% sinc interpolation
f1 = fconv(f0,sinc([-(2*N-3):2:(2*N-3)]'/2),1);
f1 = f1(N:2*N-2);
```

Now we arrive at the multiplication with the chirp $T_t$. This is relatively simple since we have to compute $T_t(\frac{k}{2\Delta})$ (i.e., $\exp\{-i\pi\frac{k^2}{4\Delta^2}\tan\frac{\alpha}{2}\}$ for $k=-N+1,\ldots,N-1$ with $\Delta^2=N+1$). To get the even samples of the product $T_t(x)f(x)$ we have to multiply the even samples of the two functions and for the odd samples we have to multiply the odd samples of the factors. Thus if $l_0$ is the vector with the even samples of $T_t$ (i.e., the values $T_t(\frac{2k}{2\Delta})$) and $l_1$ the vector with the odd samples (i.e., the values $T_t(\frac{2k+1}{2\Delta})$) then we have to multiply the vectors $f_0$ and $l_0$ componentwise for the even samples and the vectors $f_1$ and $l_1$ componentwise for the odd samples of $g=T_t f$. This explains[4]

```
t = pi/(N+1)*tan(alpha/2)/4;
...
% chirp multiplication
chrp = exp(-i*t*(-N+1:N-1)'.^2);
l0 = chrp(1:2:end);  l1 = chrp(2:2:end);
f0 = f0.*l0          f1 = f1.*l1;
```

Now we arrive at step 2 of the algorithm which does the convolution $T_s * g$ with $g=T_t f$, whose even and odd samples have just been computed: $N$ even samples and $N-1$ odd samples, a total of $2N-1$ samples in the points $y_l=\frac{l}{2\Delta}$, with $l=-(N-1),\ldots,N-1$. So, eventually, we have to compute $f_a\left(\frac{k}{\Delta}\right)=f_a\left(\frac{2k}{2\Delta}\right)$ for $k=-\frac{N-1}{2},\ldots,\frac{N-1}{2}$. Splitting the convolution in even and odd parts, this gives

$$\begin{aligned} f_a\left(\frac{2k}{2\Delta}\right) &= \frac{C_\alpha}{2\Delta}T_t\left(\frac{2k}{2\Delta}\right)\sum_l T_s\left(\frac{2k-l}{2\Delta}\right)g\left(\frac{l}{2\Delta}\right) \\ &= \frac{C_\alpha}{2\sqrt{N+1}}l_0(k)\left[\sum_l T_s\left(\frac{2(k-l)}{2\Delta}\right)g\left(\frac{2l}{2\Delta}\right)+\sum_l T_s\left(\frac{2(k-l)-1}{2\Delta}\right)g\left(\frac{2l+1}{2\Delta}\right)\right] \end{aligned}$$

[4] If $g_0$ and $g_1$ are the vectors with the even, respectively the odd samples of $g$, then we can store these in the vectors $f_0$ and $f_1$ since these are not needed anymore.

$$= \frac{C_\alpha}{2\sqrt{N+1}} l_0(k) \left[ \sum_l e_0(k-l) g_0(l) + \sum_l e_1(k-l) g_1(l) \right].$$

where we have denoted by $e_0$ the vector with the even samples of $T_s$ and by $e_1$ the vector with the odd samples. As before, the vector $l_0$ contains the even samples of the chirp $T_t$.

The boundaries for the summations are left out on purpose. Let us have a closer look. In $e_0(k-l)$, the $k-l$ ranges over $-(N-1),\ldots,(N-1)$ while in $e_1(k-l)$ the $k-l$ ranges over $-\frac{2N-3}{2},\ldots,\frac{2N-3}{2}$. Thus we need the samples $T_s(\frac{p}{2\Delta})$ for $p=-(2N-3),\ldots,(2N-3)$ That means that the number of elements in $e_0$ will be $2N-1$ and the number of elements in $e_1$ will be $2N-2$. Since $g_0$ has $N$ elements and $g_1$ has $N-1$ elements, the convolution of the even parts would have length $3N-2$ and the length of the convolution of the odd parts would be $2N-4$, so that these do not add properly. This can be solved by computing $T_s(\frac{q}{2\Delta})$ for $q=-(2N-1),\ldots,(2N-1)$ instead. Taking for $e_0$ the odd subsamples of $T_s(\frac{q}{2\Delta})$ is the same as taking the even samples of the previous values $T_s(\frac{p}{2\Delta})$ where we took $p=-(2N-3),\ldots,(2N-3)$. The still gives a convolution $e_0 * g_0$ with length $3N-2$. However, the vector $e_1$ of even subsamples for this new sequence has length $2N$, so that the convolution $e_1 * g_1$ has now length $2N+(N-1)-1=3N-2$ as well and this allows to add the two convolved vectors.

Convolution is conveniently computed as before. For example $e_0 * g_0$ takes an FFT of $e_0$ and an FFT of $g_0$, multiplies the transforms and brings everything back with an inverse FFT. Similarly for $e_1 * g_1$. That are 4 FFT's for vectors of length $O(2N)$ and 2 inverse FFT's for vectors of length $O(3N)$. Some computer effort is saved if the FFT's of $e_i$ and $g_i$ are multiplied for $i=0,1$, but then adding these two first before an inverse FFT is applied to the sum. That saves one (inverse) FFT. To do this we have added an extra parameter $c$ to the convolution routine. If $c \neq 0$, then the usual convolution is done. That includes the inverse FFT after the multiplication of the two FFT's. If $c=0$, then the inverse FFT is skipped. So we end up with the following piece of code for the convolution.

```
s = pi/(N+1)/sin(alpha)/4;
...
% chirp convolution
chrp = exp(i*s*[-(2*N-1):(2*N-1)]'.^2);
e1 = chrp(1:2:end);  e0 = chrp(2:2:end);
f0 = fconv(f0,e0,0); f1 = fconv(f1,e1,0);
h0 = ifft(f0+f1);
```

The vector $h_0$ has $3N-2$ elements, of which we should take out $N$ samples in the middle to get the relevant (i.e. the even) samples of the signal $h = T_s * g$, namely $h(\frac{k}{\Delta})$, $k=-\frac{N-1}{2},\ldots,\frac{N-1}{2}$.

All what is left to do to obtain the samples $f_a(\frac{k}{\Delta})$ is to multiply elementwise with the vector $l_0$, the even component of $T_t$ and to scale with the factor $C_s = C_\alpha/(2\sqrt{N+1})$. Which is done in the following code.

```
Cs = sqrt(s/pi)*exp(-i*(1-a)*pi/4);
...
Faf = Cs*l0.*h0(N:2*N-1);
```

The whole two-phase scheme is summarized in Figure 3.

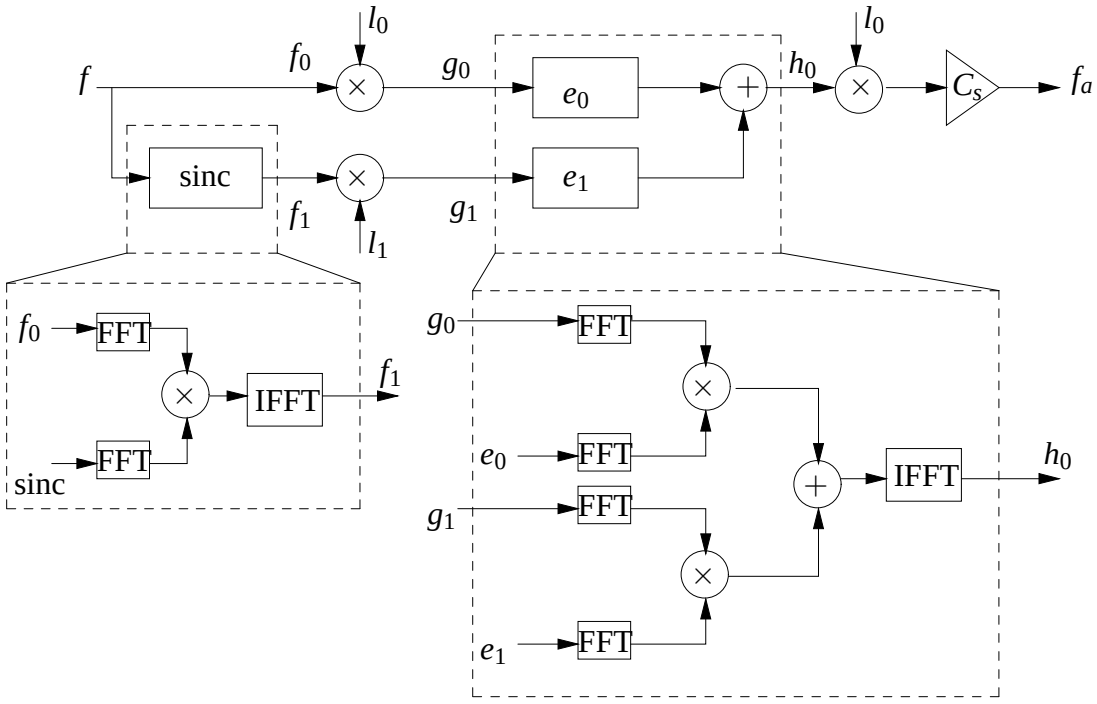



Figure 3: Resulting 2-phase scheme for the frft.

## 6. Performance

There is no visual difference between the the fractional Fourier transforms obtained with the routine `frft` described in [3] and with the new implementation (wich is made available under the name `frft2` at nalag.cs.kuleuven.be/research/software/FRFT as well). Some tests for `frft` are described in [3], and the results are the same for `frft2`. There is however a considerable difference in timing as shown in Figure 4. For that graph we have computed the fractional Fourtier transforms of signals of length $2^k$ for $k = 5,6,\ldots,20$. For each of these signals the transform is computed for 10 equidistant values of $a$ in the interval $[0,4]$. The CPU-time for each transform is approximated with the matlab routine `cputime` and the average of these 10 transforms is then taken as the execution time for that particular $k$. In Figure 4 we have plotted the values of $\log_2(\text{cpu-time})$ against the value of $k$. The solid line (blue) is the timing for the new routine `frft2` described in this report. The dashed line (red) is the timing for the routine `frft` that was a previous implementation. On the average there is a speed-up by a factor 4.2. These timings were obtained under Linux Ubuntu on a Dell desktop computer with an Intel Pentium 4 processor of 3.20 GHz and 2047 kB cache using matlab 7.11.0 (R2010b). Part of the speed-up is caused by a reduction in swapping that was needed because the vectors are not as long in `frft2` as in `frft`. For smaller values of $N$ where swapping was less needed, the speed-up was more like a factor 2.

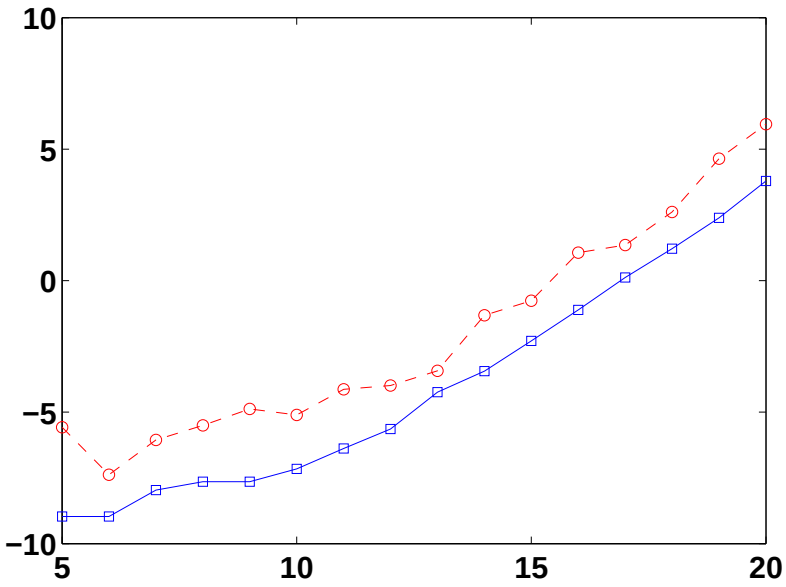



Figure 4: Timings of the computation. The $\log_2$ of the CPU-time is plotted against $k$ where $2^k$ is the length of the signal being transformed. The solid line (blue) is the timing for the new routine `frft2` described in this report. The dashed line (red) is the timing for the routine `frft` that was a previous implementation.

## References

[1] A. Bultheel, H. Martínez, A shattered survey of the fractional Fourier transform, Technical Report TW337, Department of Computer Science, K.U.Leuven (April 2002).

[2] A. Bultheel, H. Martínez Sulbaran, Computation of the fractional Fourier transform, Appl. Comput. Harmonic Anal. 16 (3) (2004) 182–202.

[3] A. Bultheel, H. Martínez Sulbaran, An introduction to the fractional Fourier transform and friends, Cubo Matemática Educacional 7 (2) (2005) 201–221.

[4] M. Grauwels, Nieuwe implementatie van de fractionele Fourier transformatie, Master's thesis, K.U.Leuven, Dept. Computer Schience (2010).

[5] H. Ozaktas, O. Arıkan, M. Kutay, G. Bozdaği, Digital computation of the fractional Fourier transform, IEEE Trans. Sig. Proc. 44 (9) (1996) 2141–2150.

[6] H. Ozaktas, Z. Zalevsky, M. Kutay, The fractional Fourier transform, Wiley, Chichester, 2001.

[7] R. Tao, G. Liang, X.-H. Zhao, An efficient FPGA-based implementation of fractional Fourier transform algorithm, J. Signal Proc. Syst. 60 (1) (2010) 47–58.

## Appendix: The matlab code

```
function [Faf] = frft2(f,a)
error(nargchk(2, 2, nargin));

f0 = f(:); N = length(f); sN = sqrt(N);
shft = rem((0:N-1)+fix(N/2),N)+1;
a = mod(a,4);

% do special cases
if (a==0), Faf = f0; return; end;
if (a==2), Faf = flipud(f0); return; end;
if (a==1), Faf(shft,1) = fft(f0(shft))/sN; return; end
if (a==3), Faf(shft,1) = ifft(f0(shft))*sN;return; end

% reduce to interval 0.5 < a < 1.5
if (a>2.0), a = a-2; f0 = flipud(f0); end
if (a>1.5), a = a-1; f0(shft,1) = fft(f0(shft))/sN; end
if (a<0.5), a = a+1; f0(shft,1) = ifft(f0(shft))*sN; end

% precompute some parameters
alpha = a*pi/2;
s = pi/(N+1)/sin(alpha)/4;
t = pi/(N+1)*tan(alpha/2)/4;
Cs = sqrt(s/pi)*exp(-i*(1-a)*pi/4);

% sinc interpolation
f1 = fconv(f0,sinc([-(2*N-3):2:(2*N-3)]'/2),1);
f1 = f1(N:2*N-2);

% chirp multiplication
chrp = exp(-i*t*(-N+1:N-1)'.^2);
l0 = chrp(1:2:end); l1 = chrp(2:2:end);
f0 = f0.*l0;        f1 = f1.*l1;

% chirp convolution
chrp = exp(i*s*[-(2*N-1):(2*N-1)]'.^2);
e1 = chrp(1:2:end);  e0 = chrp(2:2:end);
f0 = fconv(f0,e0,0); f1 = fconv(f1,e1,0);
h = ifft(f0+f1);

Faf = Cs*l0.*h(N:2*N-1);

function z = fconv(x,y,c)
% convolution by fft
N = length([x(:);y(:)])-1;
P = 2^nextpow2(N);
z = fft(x,P) .* fft(y,P);
if c >0,
    z = ifft(z);
    z = z(1:N);
end
```