Expression of Interest (EOI) Questionnaire

Name of the contact person for this submission: *DIMA Mihai-Octavian*

Institutions involved in this submission: *no institution involved (I am however employee of the Inst. For Nuclear Physics and Engr. – Romania (Computational Phys. Dept.), and Habil. Prof. in the Electronics Dept. of the Polytechnical Univ. – These institutions have rather lengthy procedures for formal involvement, but I can inquire if and the extent they would consider for involvement.)*

Items of interest for potential cooperation: *software*

Level of potential contribution: *in-kind: software development – see attached.*

Assumptions made as coming from the EIC Project/labs for items of interest: *none.*

Labor contribution for the EIC experimental equipment activities: *currently myself: 30% FTE, but am searching for grad students interested.*

Timing constraints to your submission: *none.*

Other information: *see attachment.*

Scientific software for EIC

Contact:

Mihai-Octavian Dima

Senior Researcher-I (PI)	Habilitated Professor
Computational Phys. Dept.	Electronics Dept. PhD School
http://dfcti.ifin.ro	www.sdettib.pub.ro/articole/28
e-mail: modima@cern.ch	
modima@nipne.ro	tel. : +40-724-502-557
http://cern.ch/modima	(handy)
Inst. for Nuclear Phys. & Engr.	Polytechnical Univ.
http://www.nipne.ro	http://www.upb.ro
Bucharest - Romania	Bucharest - Romania

Software

A number of proficient solutions that appear in science and engineering entail advanced objects or methods.

Often the "pen_on_paper" solution is simple, elegant and powerful.

However, implementing it in the computer can mean hundreds (more often thousands) of highly skilled man hours.

Example: *a polymorphic C++ class for complex numbers has 500+ afferent operator instantiations for operators in all combinations. Yet, these must be outside of class, for it to retain memory allocation speed, must have move semantix implemented (even without a resource), etc.*

My work in this direction has been that of fielding software for science and engineering to reach "pen_on_paper" capability in working with computers.

We do want very much those math objects because of their high power and do not want to go back to programming bit by bit.

Example: *solution to dispersive wave-equation with SU(2) scalars (see attached). Note (in blue) the simplicity of transcribing the propagator.*

Illustratively, the SU2 class has over 1100 operator instantiations, allowing it the polymorphic flexibility needed:

```
[sy]_su2<cpx<int>> * [sx + sz]_su2<dblx> = [isx - isz]_su2<cpx<dblx>>

SU2 >
```

With this capability, more advanced solutions can be built, like this show-case: *non-abelian Weyl-Wigner FFT (see attached).*

Description

The code represents an application of the SU2 suite for SU(2) spinors.

One showcase application is the solution to the transmission-line equation:

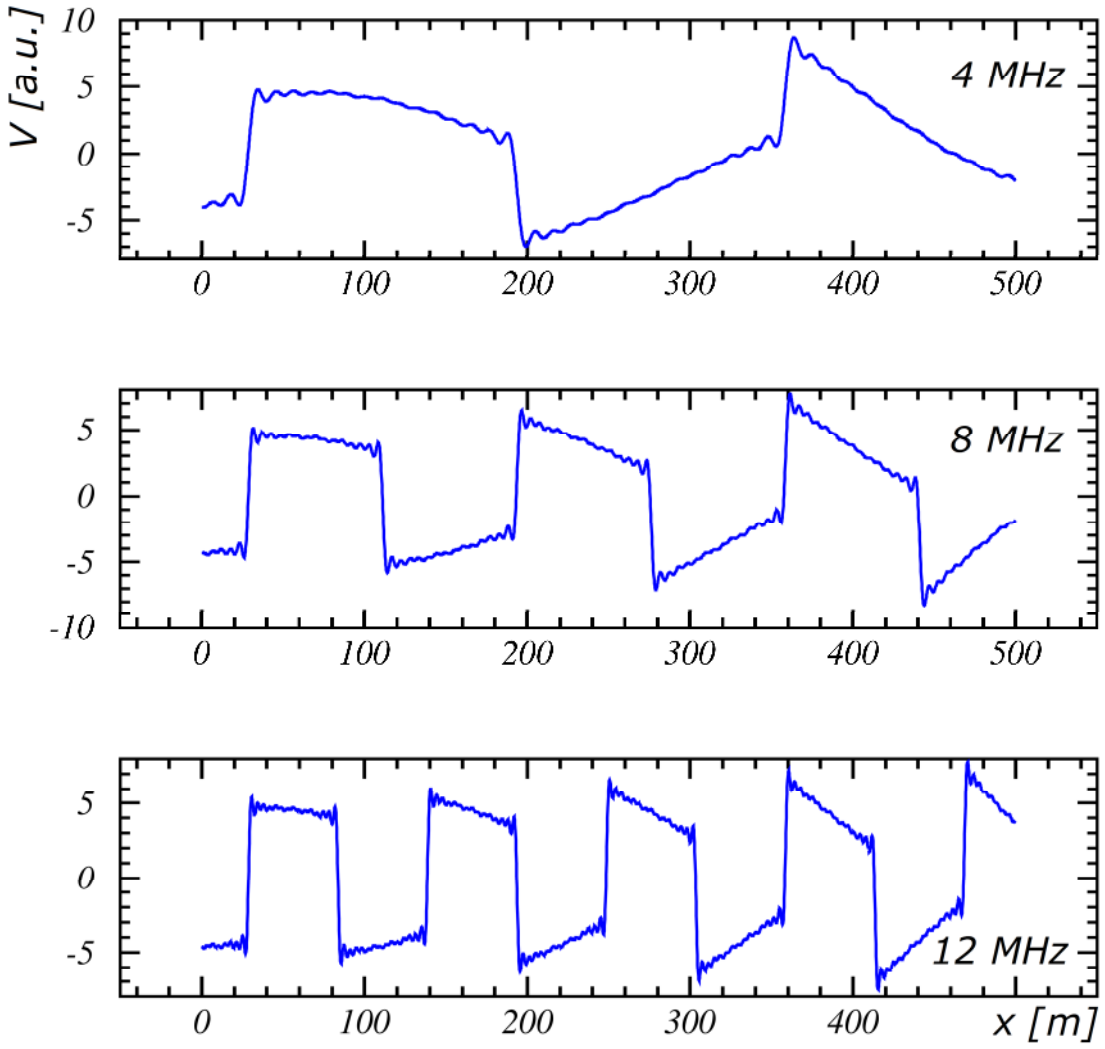
$$\partial_x + \sigma_x \partial_{ct} + \frac{j\sigma_y}{\lambda_d} = 0_{|\psi}$$

$$\psi(x,t) = e^{j\omega t - j(k\sigma_x + \frac{\sigma_y}{\lambda_d})x}{}_{|\psi_\omega}$$

$$= e^{-\frac{1+\sigma_x v/c}{1-v^2/c^2}\frac{j\sigma_y}{\lambda_d}(x-vt)}{}_{|\psi_v}$$

We will take the second embodiment of the solution, as the more physical of the two and implement it as a `propagator`.

We will add a few harmonics and let them propagate. The code computes both the voltage and the current.



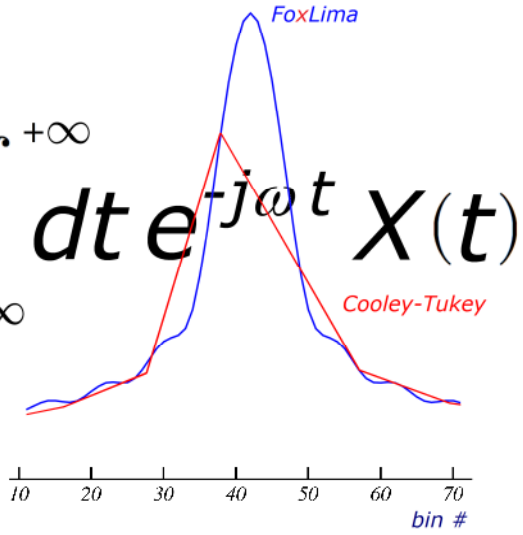
```
//  code    : test SU2 package
//
//  author  : M. Dima
//            Inst. for Nuclear Phys. & Engr.
//
//  date    : NXV4            / Sat Dec  3 17:13:57 CET 2016
//
//*************************************************************
//
#include <cmath>
#include <string>
#include <stdio.h>
#include <iostream>
#include <iomanip>

#include "cpx.hh"
#include "su2.hh"
#include "psi.hh"

using namespace std                                             ;
using namespace cpx4                                            ;
using namespace su24                                            ;
using namespace psi4                                            ;

 using dblx =      double                                       ;
 using real = long double                                       ;

 real     pi = 3.14159265358979323846264L                       ;
 real     e  = 2.71828182845904523536028L                       ;

 real    Ld  = 100.0                                            ;
 real    c   =   3.0E8 * 0.65                                   ;

 auto j  = cpx<real>(0,1)                                       ;

 auto sx = su2<cpx<real>>(0,  1,
                          1,  0)                                ;
 auto sy = su2<cpx<real>>(0, -j,
                          j,  0)                                ;
 auto sz = su2<cpx<real>>(1,  0,
                          0, -1)                                ;

auto propagator(real x, real t, real f)
 {
  real gamma = sqrt(1+f*f*Ld*Ld/c/c)                            ;
  real beta  = sqrt(gamma*gamma-1) / gamma                      ;

  return e^(-(1+sx*beta)*(j*sy)*(x-beta*c*t)*gamma*gamma/Ld);}
```

```
int main()
{

 int  N = 10000                                                    ;
 real L = 10000                                                    ;
 real f =     8.0e6                                                ;
 real V, I, x                                                      ;

 for (int i=0; i<N; i++) {
   x = i * L/N                                                     ;
   V = real (
       (psi<real>(1,0) | propagator(x,0,0.90*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,0.92*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,0.94*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,0.96*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,0.98*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,1.00*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,1.02*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,1.04*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,1.06*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,1.08*f) | psi<real>(0,1) )+
       (psi<real>(1,0) | propagator(x,0,1.10*f) | psi<real>(0,1) )
);
   I = real (
       (psi<real>(0,1) | propagator(x,0,0.90*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,0.92*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,0.94*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,0.96*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,0.98*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,1.00*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,1.02*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,1.04*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,1.06*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,1.08*f) | psi<real>(0,1) )+
       (psi<real>(0,1) | propagator(x,0,1.10*f) | psi<real>(0,1) )
);
   cout << x << " " << V << " " << I << endl                       ;}

}
```

FoxLima

non-abelian FFT

$$\mathcal{F}[X] = \int_{-\infty}^{+\infty} dt\, e^{-j\omega t} X(t)$$



NXV4 solutions
fftnxv4 @ gmail.com

Performance

- *71 log(N)* ..... *ns / sample* `Cooley-Tukey<double>`
- *0.083 $N^{0.79}$* ..... *ns / sample* `FoxLima<double>`

on: proc: *x86_64 Intel Xeon E312xx Sandy Bridge @ 2.6 GHz*
cache: *32k, 32k, 4096k*
op-sys: *Linux: 2.6.32-504.16.2.el6.x86_64*
gcc: *4.4.7 20120313 (Red Hat 4.4.7-11)*

Header file: `#include "fxl.hh"`

Libraries: `libfxl4.a` (static) `libfxl4.so` (dynamic)

linux-2.6.32-504.12.2.el6.x86_64
gcc-4.4.7 20120313

Class ... `fxl<T>` ..... std. T = `double, long double, cpx<double>, cpx<long double>`
extra T = `mtx<T>`

Member **variables** (public):

- `ddc / ddr = T` ..... pointers to *Np* long arrays of "imag" / "real" part (for `mtx<T> = A + iB`, A,B self-adjoint)
  `[ 0 ... +F −F ... 0]`
- `fzc / fzr = T` ..... pointers to *Np* long arrays of "imag" / "real" part (for `mtx<T> = A + iB`, A,B self-adjoint) after the FoxLima filter
  `[−F ... 0 ... +F]`
- `N = int` ..... # of Fourier bins = *Np*
- `p = int` ..... oversampling ratio
- `rms = double` ..... signal rms

Constructor (public)

```
fxl F(T*, T*, N,  q, str, ...)
      ^   ^   ^   ^    ^    ^
      |   |   |   |    |    |_ arg's for the command (see below)
      |   |   |   |    |
      |   |   |   |    |_ command
      |   |   |   |
      |   |   |   |_ 2^q FFT bins - allows oversampling, p = 2^q / N, recommended q:
      |   |   |                    q > 3 + log(N) / log(2)
      |   |   |
      |   |   |_ # i/p samples
      |   |
      |   |_ pointer "real"-part (or vec of matrices, symmetrical part)
      |
      |_ pointer "imag"-part (or vec of matrices, anti-symmetrical part)
```

command = "XX" , or "XX raw" – with XX = F0, F1, F2, F4, F8, F16, GX, FC, FX, FE, FW

- **F0** = Cooley-Tukey, -6dB/oct tails, p = 2^q / N
- **F1** = FoxLima-F1 , -6dB/oct tails, p = 2^q / N
- **F2** = FoxLima-F2 , -12dB/oct tails, p = 2^q / N, similar to Welch apodisation, but w/ exact cancellation of 1 / (f-f_0) tails
- **F4** = FoxLima-F4 , -24dB/oct tails, p = 2^q / N, suis-generis Welch-4 window, but w/ exact cancellation of 1 / (f-f_0)^k tails (for k = 1, 2, 3)
- **F8** = FoxLima-F8 , -48dB/oct tails, p = 2^q / N, suis-generis Welch-8 window
- **F16**= FoxLima-F16 , -96dB/oct tail, p = 2^q / N, suis-generis Welch-16
- **GX** = Gaussian(sigma), p = 2^q / N, gaussian apodisation
- **FC** = coherent chirp, p = 2^q with no windowing, σ size
- **FX** = coherent chirp, p = 2^q with Gaussian(σ) window
- **FE** = coherent chirp, p = 2^q with back-exp(τ) window
- **FW** = coherent chirp, p = 2^q with Welch(σ) window

For the following: λ_{-3} p = # bins at which smooth function falls -3 dB
λ_{-20} p = # bins at which smooth function falls -20 dB
condition: $1.472 < \lambda_{-20} / \lambda_{-3} < 4.684$
$2f_0 \Delta t = f_0 / FSR$ = chirp coherence-frequency
σ_{chirp} = chirp coherence-length # bins

case		
	F0, F1, F2, F4, F8, F16 raw	arg = void
	F0, F1, F2, F4, F8, F16 smooth	arg = λ_{-3}, λ_{-20}
	GX raw	arg = σ
	FC raw	arg = $2f_0 \Delta t$, σ_{chirp}
	FX raw	arg = $2f_0 \Delta t$, σ_{chirp}
	FE raw	arg = $2f_0 \Delta t$, τ_{chirp}
	FW raw	arg = $2f_0 \Delta t$, σ_{chirp}
	GX smooth	arg = σ, λ_{-3}, λ_{-20}
	FC smooth	arg = $2f_0 \Delta t$, σ_{chirp} , λ_{-3}, λ_{-20}
	FX smooth	arg = $2f_0 \Delta t$, σ_{chirp} , λ_{-3}, λ_{-20}
	FE smooth	arg = $2f_0 \Delta t$, τ_{chirp} , λ_{-3}, λ_{-20}
	FW smooth	arg = $2f_0 \Delta t$, σ_{chirp} , λ_{-3}, λ_{-20}

Parameters

- f_n = (n-Np/2) / Np / T ..... frequency of bin n
- $A_n = \sqrt{(ddr_n^2+ddc_n^2)}$ / Np ..... amplitude of bin n (Cooley-Tukey)
  $\sqrt{(fzr_n^2 + fzc_n^2)}$ / Np ..... - - '' - - (FoxLima)

Resolution

- df / f = 0.7 / int(f T) ..... frequency resolution
- dA / A = from i/p ..... amplitude resolution

Description

The ***fxl class*** is a suite of D-FFT methods.

COHERENT SIGNALS

F0 – is the classic Cooley-Tukey algorithm. It receives N time bins and outputs 2^q bins, where $p = 2^q / N$ is the (frequency)-oversampling ratio (recommended around x8). The **oversampling** solves the imprecision of Cooley-Tukey's algorithm which for certain frequencies (periodically in the spectrum) has large errors, missing their peaks in:

- frequency: $\delta f < 1 / Np\Delta t$,
- amplitude: $\delta A / A < 2p / \pi$,
- phase: $\delta\varphi < \pi / 2p$.

It is evident that for p = 1 the errors can be large.
The Cooley-Tukey algorithm has -6 dB/oct fall-off spectrum leakage tails. For certain peaks, with f = n / T, this is not visible as the sampling occurs exactly at the position of the zero's. For all else it is visible in various degrees. Metrologically, the favourable peaks do not represent the real performance of the algorithm.

F1 – is a filter to the Cooley-Tukey algorithm. It solves the problem of the Gibbs satellite-lobes, but has still the spectrum leakage tails, also -6 dB/oct fall-off. Its advantage is speed, requiring little CPU time.

F2 – is the best trade-off Cooley-Tukey filter, between thin peak width and Gibbs lobes' suppression. It has a -12 dB/oct fall-off of spectrum leakage tails. Suppression is exact with respect to digitisation – in order k = 1 of the $1 / (f-f_0)^k$ tails. It is loosely equivalent to Welch apodisation, however performed in frequency space rather than in temporal space.

F4 – has an advanced rejection of Gibbs lobes and -24 dB/oct fall-off of spectrum leakage tails. Suppression is exact with respect to digitisation – in orders k = 1, 2, 3 of the $1 / (f-f_0)^k$ tails. It is loosely equivalent to Welch-4 apodisation, however performed in frequency space rather than in temporal space.

F8 – has an advanced rejection of Gibbs lobes and -48 dB/oct fall-off of spectrum leakage tails. It is a Welch-8 apodisation, performed in temporal space. It is susceptible to spectrum distorsions due to inexact cancellations of digitisation terms when the signal has large phase noise, or it represents a short temporal sequence.

F16 – has an advanced rejection of Gibbs lobes and -96 dB/oct fall-off of spectrum leakage tails. It is a Welch-16 apodisation, performed in temporal space. It is susceptible to spectrum distorsions due to inexact cancellations of digitisation terms when the signal has large phase noise, or it represents a short temporal sequence.

GX – has an advanced rejection of Gibbs lobes and performant fall-off of spectrum leakage tails. It is a gaussian apodisation, performed in temporal space, of user given σ.

NOISE SIGNALS

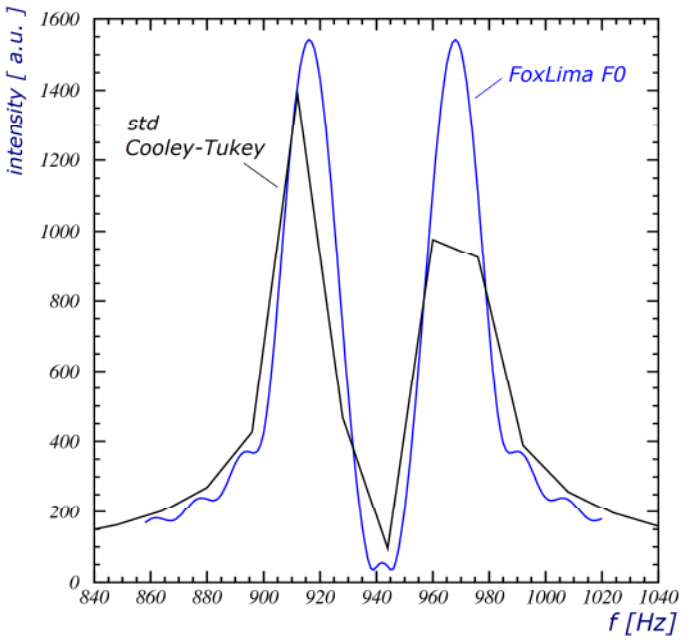
The signal below seems to have a well defined FFT peak in the spectrum, as being a single frequency.

The problem however is that its second part is phased 180^o behind the first. As such, when they are added, the FFT signal will be zero.

This may be mathematically correct, however engineering-wise unsatisfactory: there is a definite frequency in the spectrum, which needs to be flagged accordingly.

Similarly, certain sources emit "chirps", of valid FFT-spectrum, however (usually) at random time intervals (same $\Delta\varphi/\omega$ for all frequencies), and possibly also at various phase lags among frequencies.

For a given frequency the input sample can be divided into smaller samples and for each the FFT be computed. The results can be then summed as absolute values. For the frequency asked this will give the maximum amplitude attainable if all samples are coherent.



The problem is that this procedure has poor resolution. For any FFT the thinness of the peaks is given by the sum $\Sigma_{k=1,N-1} \exp(2\pi i(\omega-\omega_0)kT)$ which gives an "Airy"-function of frequency resolution $\delta f = 1 / NT$.

To overcome this, a phase-jump estimate at f_0 is performed, then this is applied to all other frequencies. This should be adjusted for frequency drift, however for those around f_0 it is sufficient (and other distant frequencies would have entailed too much noise from the extrapolation anyway).

This preserves the main mechanism of FFT computation, the summation of the above mentioned sum, and hence the resolution of signals – in spite of the small sample partition (and poor afferent σ_f apodisation).

FC – is without windowing of the smaller samples. User given σ and f_0.

FX – is a gaussian apodisation of the smaller samples. User given σ and f_0.

FE – is a back-exponential apodisation of the smaller samples. User given τ and f_0.

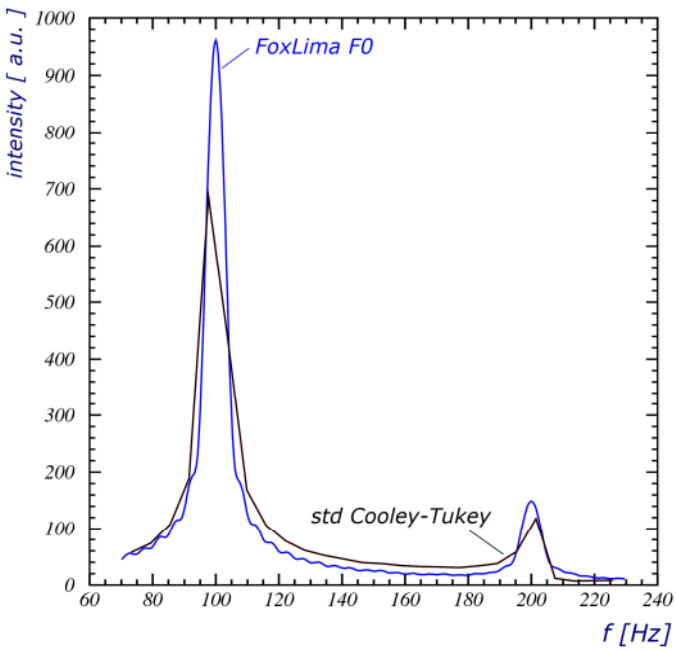
FW – is a Welch apodisation of the smaller samples. User given σ and f_0.

Performance

The main advantages of the FoxLima suite are:

1. *exact signal metrics* – due to the possibility to oversample in frequency-space, FoxLima algorithms provide exact peak resolution over Cooley-Tukey, that can register errors of up to:
   - $\Delta f = 1 / 2N\Delta t$ in frequency
   - $\Delta A = 36\%$ A in amplitude
   - $\Delta\varphi = 90^o$ in phase

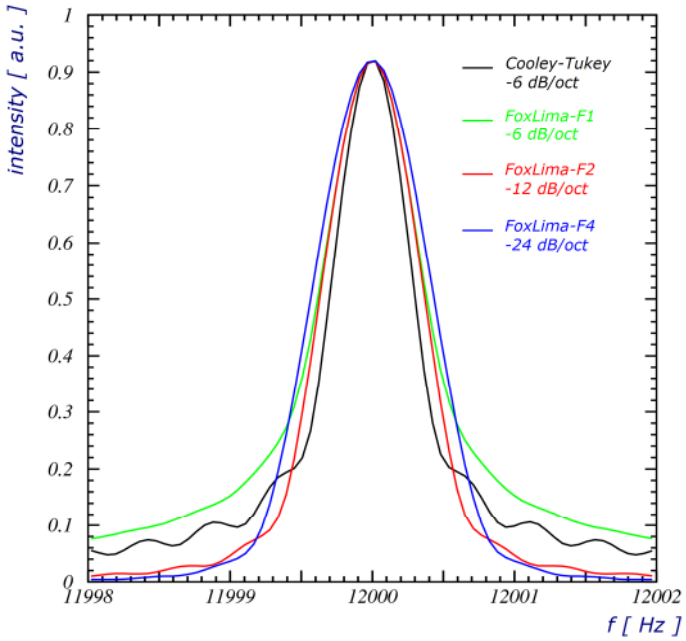
   These aspects can be seen in the figure above, and the figure to the right, where the sub-sampling problem of the Cooley-Tukey algorithm are evidentiated as peak truncation, or central-frequency imprecision.

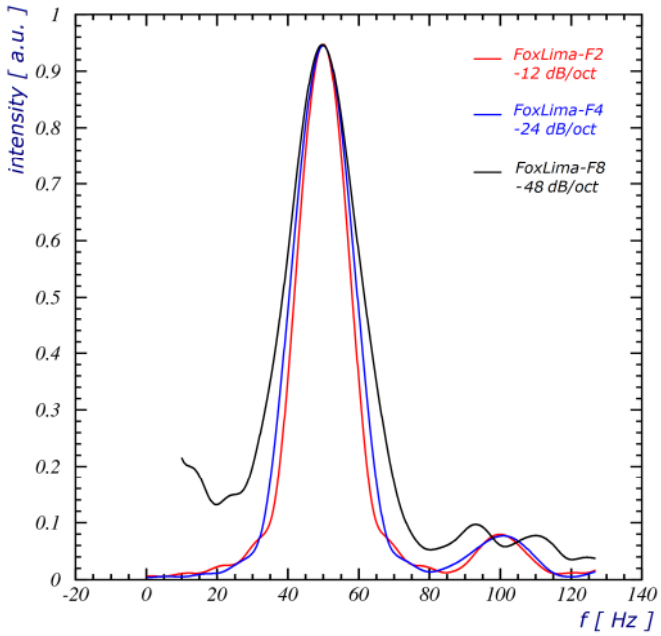


2. *exact leakage elimination* – due to its Fourier-space apodisation, algorithms F1, F2 and F4 eliminate exactly the $1 / (f-f_0)^k$ spectrum-leakage tails for:
   - k = 0 (F1 algorithm)
   - k = 1 (F2 algorithm) and
   - k = 1,2,3 (F4 algorithm).

   The figure to the right shows smoothed versions of the algorithms for width and tail fall-off.

   The 3 Fourier-space apodisation algorithms have fall-offs of:
   - -6 dB/oct (F1 algorithm)
   - -12 dB/oct (F2 algorithm)
   - -24 dB/oct (F4 algorithm).





It is important to note, that time-space apodisation does not take into account discretisation components exactly as does Fourier-space apodisation.

Therefore signals with phase-noise, or a short time data-acquisition window are not described as exact by time-space apodisation algorithms.

This is shown in the figure here to the left – on a short time-window (of 6 x 50 Hz periods).

The F8 algorithm has significant deviations beyond the central-peak, whereas F2 and F4 are both very precise.

3. *Weyl-Wigner FFT* – as mentioned above, random-chirp signals may deceivingly add destructively showing little (or no signal at all) at certain frequencies.

   The Weyl-Wigner algorithms add coherently such signals around a (user given) coherence frequency (f_0), of sampling window (the approximate size of the chirp, σ).

   A comparison with the F2 algorithm (on a noise signal) is given in the figure here below.

A small presence in F2, with a large one in WW denotes significant cancellation (or phase noise).

The apodisation over the σ window is:

- **FC** – plain Cooley-Tukey
- **FX** – gaussian
- **FW** – Welch
- **FE** – back-exponential

The apodisation used in the figure here is gaussian.