

## Chapter 4 - Fast Fourier Transformation

- Standard FT (without normalization): 
$$f(k) = \int_{-\infty}^{\infty} e^{-ikx} f(x) dx$$
- FT of periodic functions of period  $L$ :  
(In numerical representation we are limited to an interval,  $f$  has to be quasi-periodic) 
$$f(k) = \int_0^L e^{-ikx} f(x) dx$$
- Sampling of  $f$  on a (discrete) grid yields: 
$$f(k_n) = \Delta x \sum_{j=0}^{N-1} e^{-ik_n j \Delta x} f(x_j)$$
- $N$  grid points to cover interval of length  $L$  results in two limits:
  - Due to the finite length  $L$  there is a lowest wave-number  $k_{\min} = \Delta k = 2\pi/L$
  - *Sampling Theorem*: To uniquely identify an oscillation we need two sampling points per period, i.e.  $\lambda_{\min} = 2\Delta x$ , thus  $k_{\max} = \pi/\Delta x$

$$f(k_n) = \Delta x \sum_{j=0}^{N-1} e^{-ik_n j \Delta x} f(x_j) \longrightarrow f(k_n) = \Delta x \sum_{j=0}^{N-1} e^{-i \frac{2\pi}{N\Delta x} n j \Delta x} f_j$$

$$k_n = \frac{2\pi}{N\Delta x} n, \quad n = 0, \dots, N-1$$

- Forward DFT:  $f(k_n) = \Delta x \sum_{j=0}^{N-1} e^{-2\pi i n j / N} f_j$
- This is nothing but a projection of  $f(x_i)$  on a new basis:

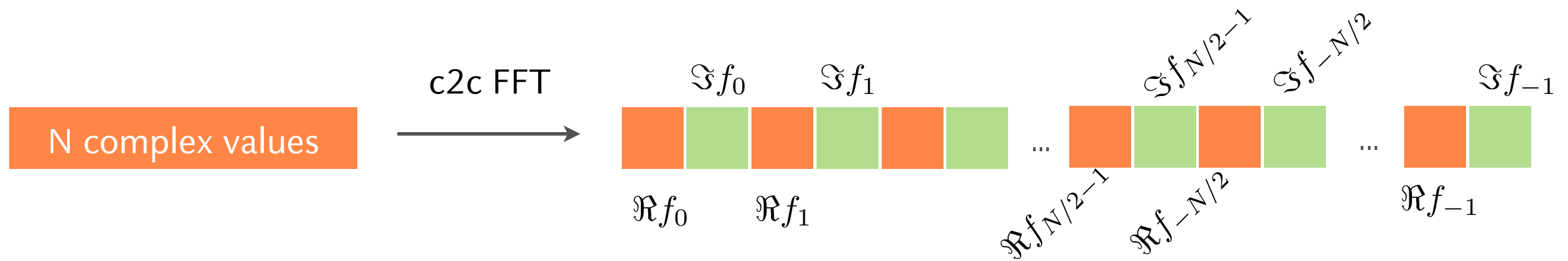
$$\hat{\mathbf{f}} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & & & & \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)^2} \end{pmatrix} \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N-1}) \end{pmatrix} \quad \omega = e^{-2\pi i / N}$$

- Fast Fourier Transformation is a clever way to compute this matrix-vector multiplication in  $\mathcal{O}(N \log(N))$  instead of  $\mathcal{O}(N^2)$

$$f(k_n) = \Delta x \sum_{j=0}^{N-1} e^{-2\pi i n j / N} f_j \quad \text{is periodic with period } N: f(k_{N-n}) = f(k_{-n})$$

- There are two ways in which we can identify the frequencies:
  - ▶ All frequencies are positive, ranging from  $k_{min}=0$  to  $k_{max}=k_{N-1}$
  - ▶ There are negative and positive frequencies, symmetric about 0:  $k_{min} = k_{-N/2}$  to  $k_{max} = k_{N/2-1}$

# Frequency mapping



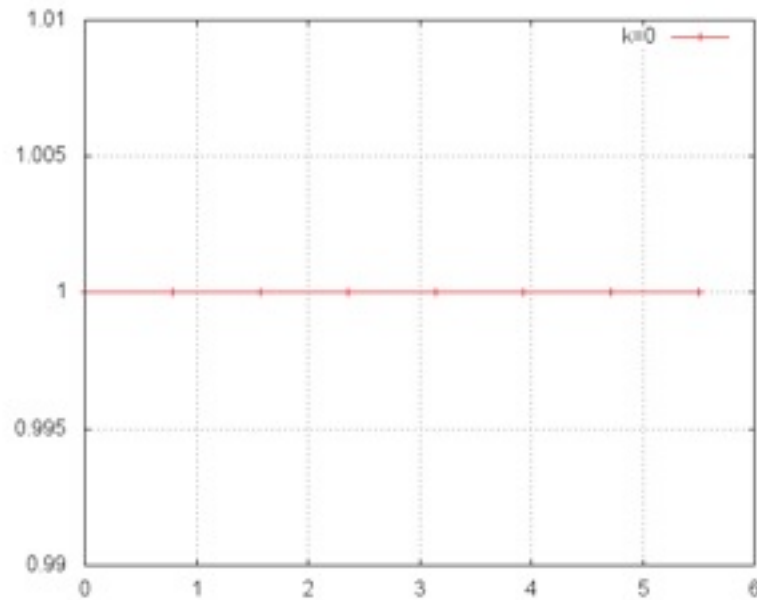
N complex values = 2N doubles

# Example: $N = 8$

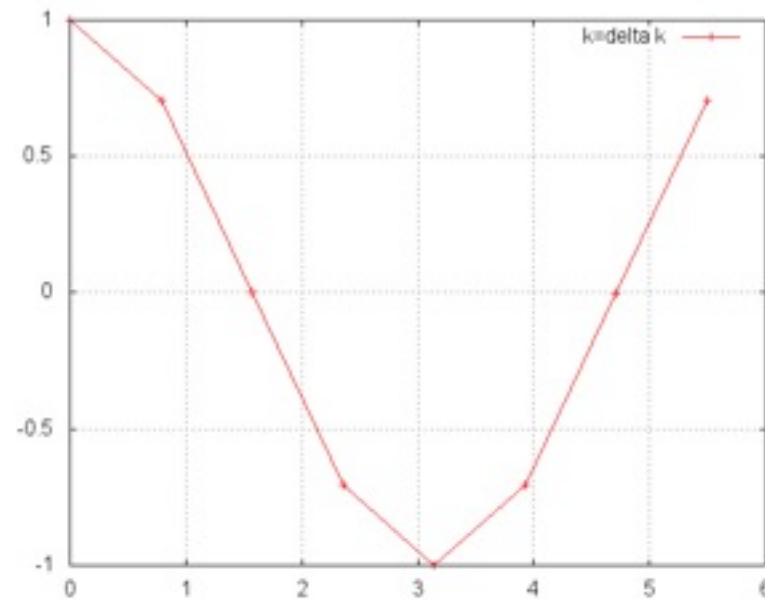
$$I = [0, 2\pi], \quad L = 2\pi, \quad N = 8, \quad \Delta k = \frac{2\pi}{L} = 1, \quad k_{max} = \frac{\pi}{\Delta x}$$

$$\Delta x = \frac{L}{N} = \frac{2\pi}{8} \longrightarrow x_{max} = 7\Delta x = \frac{7}{8}2\pi \neq 2\pi$$

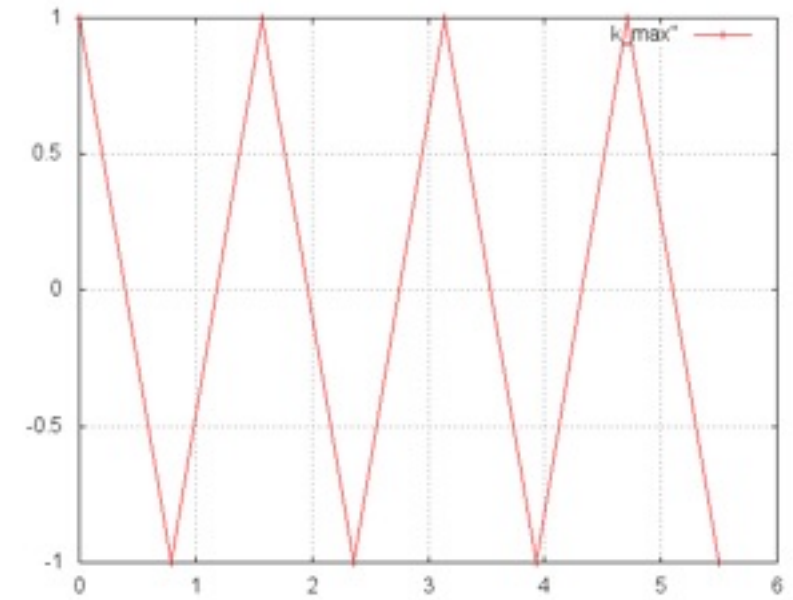
periodic grid: we do not store the redundant point at  $f(x=L)$ !



	<u>Re</u>	<u>Im</u>
$f_0$	8	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0



	<u>Re</u>	<u>Im</u>
	0	0
$f_1$	4	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
$f_{-1}$	4	0



	<u>Re</u>	<u>Im</u>
	0	0
	0	0
	0	0
	0	0
$f_{-N/2}$	8	0
	0	0
	0	0
	0	0
	0	0

- FFTW provides three interfaces:
  - ▶ Basic interface (we will use this one)
  - ▶ Advanced interface
  - ▶ Guru interface
- Results from FFTW are usually without normalization, i.e. one might need to normalize results by  $1/N$  to obtain agreement with analytical results

# C2C transformation with FFTW

```
#include <fftw3.h>
#include <cmath>
#include <iostream>
//-----
using namespace std;
//-----
int main()
{
    const int N=8;

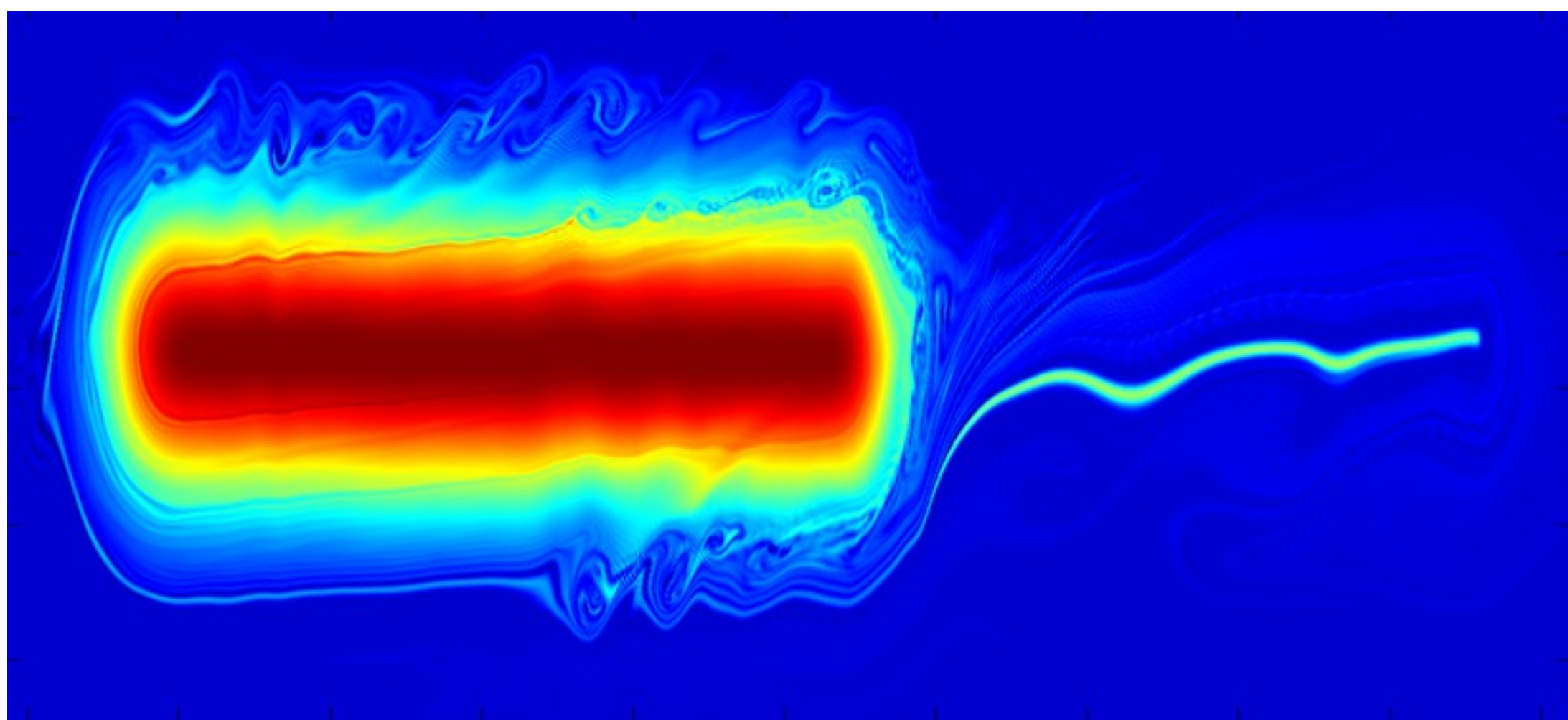
    fftw_plan fw, fw2;

    fftw_complex* inC = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    fftw_complex* outC = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    const double L=2*M_PI;
    const double dx=L/N;
    double x;

    // inC = exp(ii*x)
    for(int i=0; i<N; i++){
        x=i*dx;
        inC[i][0]= cos(x); // Re
        inC[i][1]= sin(x); // Im
    }
    fw = fftw_plan_dft_1d(N, inC, inC, FFTW_FORWARD, FFTW_ESTIMATE);

    fftw_execute(fw);
    (...)
}
```





## Chapter 5 - Libraries

## No need to re-invent the wheel...

- Libraries are a collection of functions or classes
- Many scientific libraries are open-source, i.e. we can get the source-code, look into the code, compile our own version of the library and use it for free
- There is a huge amount of interesting scientific/mathematical libraries out there. Examples are:
  - ▶ FFTW (Fourier Transform)
  - ▶ BLAS/LAPACK (Linear Algebra)
  - ▶ GSL (Collection of many scientific mathematical functions, ODE solvers,...)
  - ▶ PETSc ( Linear Algebra / Non-linear Newton methods, fully parallel)
  - ▶ Blitz++ (Vector/Matrix library)
- Libraries that do not provide ready mathematical functions, but help us to implement parallel algorithms. Examples:
  - ▶ OpenMP (Multi-threading)
  - ▶ MPI (Communication between processes running on multiple machines)

## From scratch to working library

- We will go through the necessary steps how to get a working library from its sources and use FFTW as an example. The steps for other libraries are very similar.
- Note: With many Linux distributions you can also install libraries via the packet manager from the repositories. Usually you want to have full control about what version of the library you are working with and the precise settings that have been used generating it. Hence, we generate the library ourselves.
- *1. Step:* Download the sources.  
Sources usually come in a file with filename extension \*.tgz on Linux. \*.tgz means: This is a tar-ball (usuall ending: \*.tar) but it is additionally zipped (i.e. compressed) using GZip.
- *2. Step:* Unpack the sources  
If the file from step 1 has the name fftw.tgz, unpack with the tar command in the console:  
`tar xvfz fftw.tgz`

## From scratch to working library

- *Step 3: Prepare to compile the library*

Once you have unpacked the tar-file, you have all source code of the library readily available. Still, it is of no real use to us. We need to compile the sources into a binary object, the library. Compilation often requires a setup which defines what compiler options will be used, where the final output will be stored and sets library depending options.

This setup process is handled via the configure tool. In almost all unpacked sources you will find an executable script called configure. Run `configure --help` to see all available options.

A configure call for FFTW might look like:

```
./configure --prefix=$HOME/libs/fftw --enable-sse2 --disable-fortran
```

The `--prefix` option is very common and almost always present. It tells the compiler to where the eventually generated library object will be copied.

Once configure is done Makefiles are generated. These are the files which contain all recipes the compiler has to know to compile the library.

## From scratch to working library

- *Step 4: Compiling the library*

Just type "make" (get some coffee, call some friends, check facebook, or sleep)

Type "make -j N" to compile on N cores in your machine (less time for coffee,...)

- *Step 5: Install the library*

Type "make install" to have the library copied to the place we specified with the --prefix flag of configure.

- *Step 6: Done!*

Check out the install directory and you will find at least two sub-folders in it. One is called "lib", the other "include". In the lib directory you will find the binary object which is the actual library. For FFTW this is the file "libfftw3.a". In the include directory you will find all the header files you need to use the library.

## Using the library in our project

- When we use the library in our project, there are two places we have to pay attention:
- In our source code we will have to include the header-file(s) from the library.
  - ▶ When we compile our source code, using functions from the library, the compiler will have to check if there are any functions around with these names and the appropriate interfaces.

The compiler will read the header file of the library and be satisfied. We will have to tell the compiler where to look for the header files of the library.
- Once our own program is compiled the linker has to generate the final executable. Therefore, the compiler will take the object generated out of our program and the library object and "melt" them together.

We just have to tell the compiler where to find the library and that it has to be included in the "melting" process.

## Using the library in our project

- A short program which calculates a 1D complex to complex FFT using FFTW

```
#include <fftw3.h>
```

```
int main(){  
    fftw_complex *in, *out;  
    fftw_plan p_fwd, p_bck;  
    const int N = 1024;  
  
    in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);  
    out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);  
  
    p_fwd = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);  
    p_bck = fftw_plan_dft_1d(N, in, out, FFTW_BACKWARD, FFTW_ESTIMATE);  
  
    fftw_execute(p_fwd); /* repeat as needed */  
    fftw_execute(p_bck);  
  
    fftw_destroy_plan(p_fwd);  
    fftw_destroy_plan(p_bck);  
    fftw_free(in);  
    fftw_free(out);  
}
```

## Using the library in our project

- When compiling our program we have to add to the compiler options
  - ▶ the linker option to link `libfftw3.a` FFTW to our final file
  - ▶ the `include directory` to the include path
  - ▶ the `library directory` to the library path

```
g++ fftw_test.cxx -o fftw_test -L$HOME/libs/fftw/lib -I$HOME/libs/fftw/includes -lfftw3
```

- When linking libraries there is a convention for the name of the libraries. A library file name starts always with `lib...` (like `libfftw3.a`). In the compiler command line we always have to leave away the `lib...` and the extension.
- Real life is a bit more tricky. Usually there are *static* and *shared libraries*. We used a static library so far. Static libraries have a `.a` (for *archive*) extension, while dynamic libraries have `.so` (for *shared object*) as extension.