One-dimensional and higher-dimensional arrays

Sampled data

Theoretical models of physical phenomena and experimental data from sensors are often represented in terms of **continuous functions** $y(x)$ of a coordinate x. To represent these functions in a computer we often use **sampled** values: samples of the function at a discrete set of coordinates.

In other cases, our models are naturally **discrete**, for example the spin at each lattice location in the Ising model.

This kind of data can be represented as a set of points (x_i, y_i) consisting of N values y_i at locations x_i. Of course, the positions and the values can have more than one dimension:

- A picture $I(x_i, y_i)$
- Position of a particle at different times $\mathbf{r}(t_i)$
- Magnetic field measured in 3-D space $\mathbf{B}(\mathbf{r}_i)$ — need 3 scalars for each of $\mathbf{B}$ and $\mathbf{r}_i$

Arrays

These data are naturally represented in computers as arrays: collections of objects (usually numbers) which can be accessed using an index.

We will restrict our attention to the case where the objects in a given array are of the **same type** (for example 64-bit floating-point numbers), and the index is a set of one or more **integers**.

One-dimensional arrays in Python

A single standard array type forms the basis of the **numpy** package

```
import numpy as np
a=np.array([0,1,2,3])
b=np.arange(4)
print(a,b)
print(a[1],a[:2],np.sqrt(a))
```

```
[0 1 2 3] [0 1 2 3]
1 [0 1] [ 0.          1.          1.41421356  1.73205081]
```

One-dimensional Arrays in C and C++

Statically defined C-style arrays are great for simple applications:

```
int main() {
  const int n = 10;
  double h[n];
  for (int i = 0; i < n; i++)
    h[i] = sqrt(i);
  std::cout << h << " " << &h[0] << "\n";
}
```

```
0x7fff5418d330 0x7fff5418d330
```

Note `h` is a pointer to the first element of the array; it is exactly the same as `&h[0]`

Dynamically-allocated arrays

Can also use dynamically allocated arrays: amount of memory used can be decided at run-time and the memory can be recycled when no longer needed.

```
#include <cmath>
void func(int n) {
  double *h = new double[n]; // allocate space for n doubles
  for (int i = 0; i < n; i++)
    h[i] = sqrt(i);
  delete[] h; // de-allocate space
}
int main() { func(10); }
```

Avoid if you can! Doing your own memory management with **`new`** and **`delete`** is error-prone. The STL template classes like **`vector`** do the work for you.

The C++ vector class

You are strongly recommended to use the `std::vector` container class of the C++ Standard Template Library (STL) for storing your arrays.

Using the C++ `vector` container we write:

```
#include <vector>
#include <cmath>
int main() {
  const int n = 10;
  std::vector<double> h(n);
  for (int i = 0; i < n; i++)
    h[i] = sqrt(i);
}
```

So far, this container object `h` looks rather like our normal C array. Note that the `vector` container can contain all sorts of different types of values. Here we choose double precision floats, but we could have also defined an array of integers via `std::vector<int> m(n)`.

But we can do much more with them!

```
  a[2] = 9;                       // a = 1,1,9
  a.push_back(8);                 // a = 1,1,9,8
  a.insert(a.begin(), 5);         // a = 5,1,1,9,8
  a.insert(a.begin() + 3, 3, 7); // a = 5,1,1,7,7,7,9,8
  a.clear();                      // a is empty
  a.resize(6);                    // a= 0 0 0 0 0 0
  // a[99]=0;                     // Bounds error, unhelpful crash!
  cout << a.at(99);               // Bounds error, throws useful exception
}
```

```
libc++abi.dylib: terminating with uncaught exception
of type std::out_of_range: vector
```

The `vector` container is very powerful.

1. dynamic, and looks after memory management for us
2. easy to change is size
3. implemented efficiently
4. data are stored contiguously in memory, so we can pass them to a function expecting a "normal" C or C++ function by taking the address of the first element, `&a[0]`
5. can write them out using stream operator `std::cout` with some persuasion
6. bounds checking possible using the `at` function to access your vector
7. If you define a new class, you can easily make a vector of that type which will behave in the ways you would expect

Computing with Multi-dimensional Arrays

Arrays in more than one dimension are common in physics: as well as matrices, we have for example images $I(x, y)$, or a 2-D array of spins in the Ising model. In matrix notation we denote position in the array with a 2-dimensional index

$$M = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \end{bmatrix}$$

This is straightforward to replicate in python

```
import numpy as np
M=np.array([[0,1,2,3],
            [4,5,6,7],
            [8,9,10,11]])
print(M[2,2])
```

```
10
```

Memory organisation of multi-dimensional arrays

Computers have a one-dimensional address space: every byte of memory has a unique single integer address. So we have to decide how to arrange our multi-dimensional data in memory.

We can find out how Python does this using the `numpy.reshape()` function:

```
M1=np.reshape(M, [12])
print(M1)
print(np.reshape(M1, [4,3]))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```


Row-major vs column-major storage

Python and C++ use the same storage format for multi-dimensional arrays, so-called "row-major" storage. FORTRAN and MATLAB use "column-major storage", so the 2-dimensional array *M* above would appear in memory as `0, 4, 8, 1, 5, 9, 2, 6`….

The difference is not material except when transferring data between programs written in the different languages.

Implementing multi-dimensional arrays in C++

Static C-style arrays are simple and occasionally the right thing to use, but their sizes are fixed. For example `double cube[8][16][32];` creates $8 \times 16 \times 32$ contiguous numbers in memory.

In general, however, it is best to create a one-dimensional array big enough to hold your multi-dimensional array and do your own indexing. Many good libraries exist which implement such techniques. GSL does this using C structures and functions. Others use C++ classes, e.g. the BOOST libraries — recommended, but not for this course.

Statically-allocated arrays

We could represent our array **M**:

$$M = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \end{bmatrix}$$

Straightforwardly in C or C++ using:

```
int main() {
  const int n_rows = 3;
  const int n_cols = 4;
  int a[n_rows][n_cols] = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}};
  a[1][2] = 3; // example assignment
}
```

but it is fixed in size.

Multi-dimensional arrays in C++

A very good way is to use a `std::vector<double>` to store your multi-dimensional data. All you have to do then is do your own indexing into the 1-D array. For example:

```
#include <vector>
int main() {
  int n_rows = 3;
  int n_cols = 2;
  std::vector<double> a(n_rows * n_cols, 0.0);
  for (int row = 0; row < n_rows; row++)
    for (int col = 0; col < n_cols; col++)
      a[row * n_cols + col] = row + col; // note indexing
}
```

It's even better if you add a function to do calculate the index.

But this is a little awkward — we have to remember the number of columns whenever we use the array and calculate the index. We would like to be able to write code as simple as this:

```
int main() {
  int n_row = 3;
  int n_col = 2;
  Matrix M(n_row, n_col);
  for (int row = 0; row < n_row; row++)
    for (int col = 0; col < n_col; col++)
      M(row, col) = col + row; // or something more useful...
}
```

Do this with a class — effectively a new type:

```
#include <vector>
class Matrix {
private:
  std::vector<double> m_data;
  int m_rows;
  int m_cols;

public:
  Matrix(int n_rows, int n_cols, double init = 0.0)
      : m_data(n_rows * n_cols, init), m_rows(n_rows), m_cols(n_cols) {}

  double &operator()(int i, int j) { return m_data[i * m_cols + j]; }
};
```


- This starts to look very elegant now — we have a new type `Matrix` which we can use to represent two-dimensional arrays.
- It is easy to declare and use — the main difference is that we use round brackets for access not square ones (because access is actually a function call). That is, we write `M( i, j ) = SomeFunction(...);` for example
- In addition, the object handles the memory allocation and deallocation automatically.
- See the `cav::Array2` and `cav::Complex_Array2` classes for a further example of this technique (look in `cavlib/arrays.hh`).
- This technique obviously generalises to more than 2 dimensions
- For serious work in the future, consider the BOOST libraries of C++.

1 and 2-D data structures in GSL

GSL defines its own structures to represent vectors and matrices. To use the routines, we either (1) use the same data structures for our arrays, or (2) use our own C++ arrays (`vector<double>` for example) and allow GSL to view and manipulate these arrays. The second approach is encouraged because C++ arrays are good things.

The Fast Fourier Transform

Fourier transform of a continuous function

$$H(f) = \int_{-\infty}^{\infty} h(t) \exp(-2\pi i f t) \mathrm{d}t$$
$$h(t) = \int_{-\infty}^{\infty} H(f) \exp(2\pi i f t) \mathrm{d}f$$

with h and H being continuous functions of t and f.

Recall that...

- the FT of a real signal has $H(f) = H^*(-f)$
- The power spectral density (PSD) of a signal is the power between f and $f + \mathrm{d}f$:

$$P(f) = |H(f)|^2 + |H(-f)|^2$$

For a real signal $h(t)$, the two terms are equal.

Fourier Series for a periodic signal

If our signal is measured over a period $0 \leq t \leq T$, and if we assume $h(t)$ is periodic, the Fourier transform now has components at discrete frequencies $f_n = n/T$. Fourier Series:

$$H_n = \int_0^T h(t) \exp(-2\pi i n t/T) \mathrm{d}t$$
$$h(t) = \frac{1}{T} \sum_{n=-\infty}^{\infty} H_n \exp(2\pi i n t/T)$$

i.e. an infinite number of discrete frequencies can represent a continuous periodic signal.

Sampling and the Nyquist frequency

- Consider sampling the signal $h(t)$ at N uniformly-spaced points:

$$t_k = k\Delta, k = 0, 1, \ldots (N-1)$$

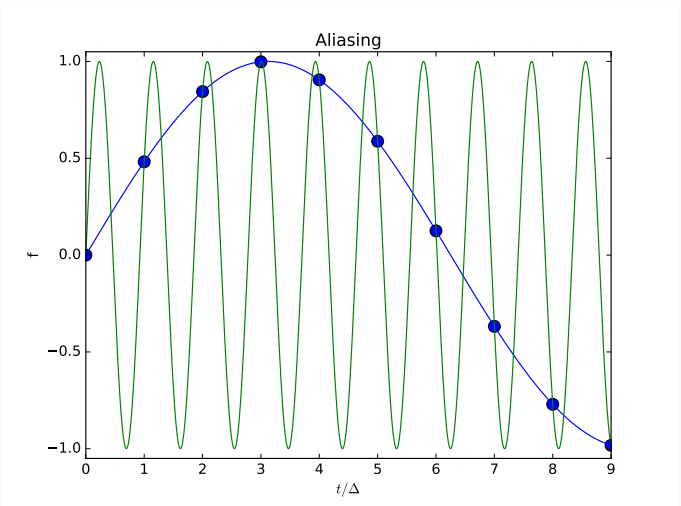
  with $T = N\,\Delta$
- The sampling rate is $1/\Delta$.
- There is a maximum representable frequency in such a signal, called the Nyquist critical frequency f_c, where we have two samples per cycle:

$$f_c = \frac{1}{2\Delta}$$

- Sampling Theorem: If $h(t)$ is band limited to frequencies $< f_c$ (i.e. if $H(f) = 0$ for $f > f_c$) then $h(t)$ is completely determined by h_n.

Aliasing

Higher frequencies than the Nyquist frequency are aliased into the range $-f_c < f < f_c$ because the frequency $f + 2f_c$ [i.e. $f + (1/\Delta)$] produces exactly the same samples as f:



Ideally we bandpass filter the signal before sampling to ensure it is bandwidth-limited and then no aliasing can occur.

Discrete Fourier Transform

Given our N samples h_k, we can construct N frequencies which approximate the continuous Fourier transform, with the highest frequency being the critical frequency f_c. It's simplest for now to assume that N is even. We define the discrete Fourier transform as

$$H_n = \sum_{k=0}^{N-1} h_k \, e^{-2\pi \, \mathrm{i} \, k \, n/N}$$

which maps N time-domain samples into N frequencies, which are

$$f_n = \frac{n}{N\Delta} = \frac{2n}{N} f_c$$

We now have a discrete signal in the time and frequency domains, with the functions periodic in both domains: $h_{k+N} = h_k$ and $H_{n+N} = H_n$.

The discrete frequencies are $f_n = n/(N\Delta) = 2nf_c/N$, with n running from $n = 0$ to $(N-1)$:

- $n = 0$ is zero frequency (sum of input values)
- $1 \leq n \leq (N/2)$ are positive frequencies, with $(N/2)$ being the highest (critical frequency f_c)
- $(N/2) + 1 \leq n \leq (N-1)$ can be thought of as negative frequencies: we can subtract $2f_c = 1/\Delta$ from them and they are the same because H_n is periodic

There is an exact inverse:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \, e^{2\pi \, \mathrm{i} \, k \, n/N}$$

The inverse is the same as forward transform except change the sign in the exponential and divide by N. Note it does not contain Δ in the definition. If we want the true FT values, we have $H(f_n) \approx \Delta H_n$.

Fast Fourier Transform (FFT)

- The FFT is an efficient method for calculating the Discrete Fourier Transform (DFT). It is important as it has revolutionised signal processing in many fields.
- How much computation is involved in a DFT? We can write

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{2}$$

where

$$W \equiv \exp(-2\pi \mathrm{i}/N) \tag{3}$$

- This looks like a matrix multiplication with a square matrix W whose $N \times N$ elements W_{nk} multiply the vector h_k of length N. This is an $\mathcal{O}(N^2)$ process i.e. its compute time is dominated by a number of complex multiplications proportional to N^2

FFT

- In fact, the FFT algorithm can do the same job in $\mathcal{O}(N \log_2 N)$ operations.
- Time saving is huge: for $N = 10^6$, $N^2/(N \log_2 N) \approx 50,000$. (50,000s is 14 hours)
- "Discovered" by Danielson & Lanczos (1942), computer discovery by Cooley and Tukey (1965), but original ideas goes back at least as far as Gauss (1802).

A DFT of length N can be written as the sum of two DFTs of length $N/2$. So if N is a power of 2, we can apply this theorem over and over again, $\log_2(N)$ times in fact, until we end up with $N = 1$. Split the DFT into odd and even terms: consider the expression for H_n which we can write

$$\sum_k^{\text{even}} h_k \exp(-2\pi \mathrm{i} kn/N) + \sum_k^{\text{odd}} h_k \exp(-2\pi \mathrm{i} kn/N) =$$

$$\sum_{m=0}^{\mathrm{N}/2-1} h_k \exp(-2\pi \mathrm{i}(2m)n/N) + \sum_{m=0}^{\mathrm{N}/2-1} h_k \exp(-2\pi \mathrm{i}(2m+1)n/N) =$$

$$\sum_{m=0}^{\mathrm{N}/2-1} h_k \exp(-2\pi \mathrm{i} mn/(N/2)) + \exp(-2\pi \mathrm{i} n/N) \sum_{m=0}^{\mathrm{N}/2-1} h_k \exp(-2\pi \mathrm{i} mn/(N/2))$$

Because $2(N/2)^2 < N^2$ this is $\sim 2\times$ quicker to calculate. And further more we can apply repeatedly until N is small.

- Best case: If N is a power of two, get very efficient FFTs. Highly recommended to use 2^N if efficiency important. Either design your experiment correctly, or pad with zeros until your data blocks have length 2^N!
- But fast techniques also exist when N can be factorised.
- Worst case: N is prime.

But computer are so fast these days you may never need to worry about this for simple applications...

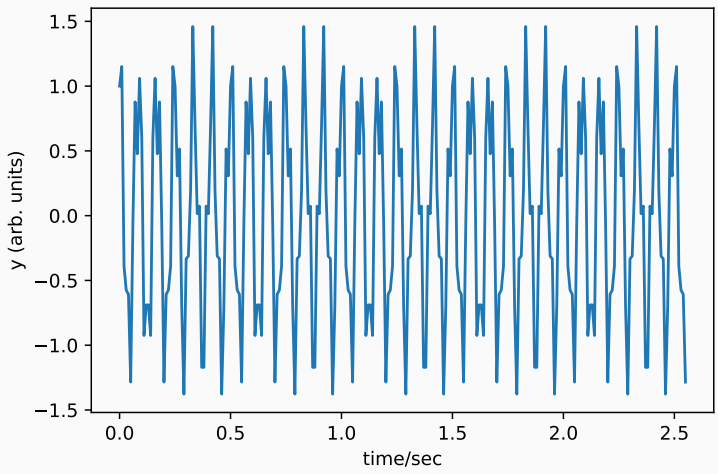
Ideas extend to more than 1-dimension, so that e.g. fast 2-D transforms are possible. Most common example is perhaps image processing by Fourier filtering.

FFT Applications

1. Convolution of two signals: FFT each, multiply the FFTs, then inverse FFT back. For example, smooth an image using a Gaussian kernel.
2. Filtering a signal – closely related to convolution. We take a signal, FFT it, multiply the FFT by a function, then FFT back, e.g. low- or high-pass filtering.
3. Crystallography
4. Find the power spectrum (PSD) $|H_n|^2$
5. Optics: Fraunhofer (and Fresnel) diffraction
6. Optics: the spatial/temporal coherence function is equal to the Fourier transform of the brightness distribution/power spectrum.
7. Signal processing. e.g. Freeview signals are transmitted using FFT: the data are cut into 8192-piece chunks, ($8192 = 2^{13}$), FFT'ed, transmitted, inverse FFT'ed on reception. In fact, FFTs are central algorithm in Digital Signal Processing (DSP)

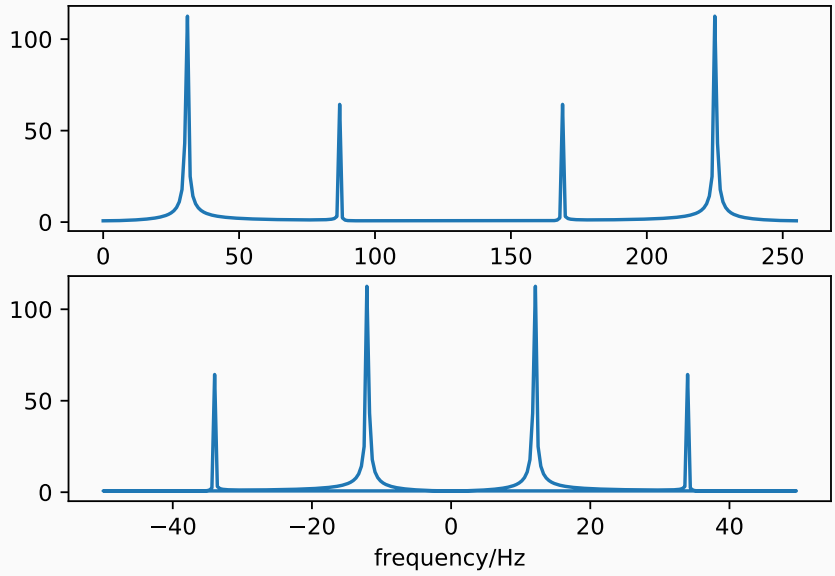
FFTs in Python

```
import numpy as np
import matplotlib.pyplot as plt
dt=0.01
fftsize=256
t=np.arange(fftsize)*dt
#Generate some fake data at 12 Hz and 34 Hz
y=np.cos(2*np.pi*12*t)+0.5*np.sin(2*np.pi*34*t)
plt.plot(t,y)
```



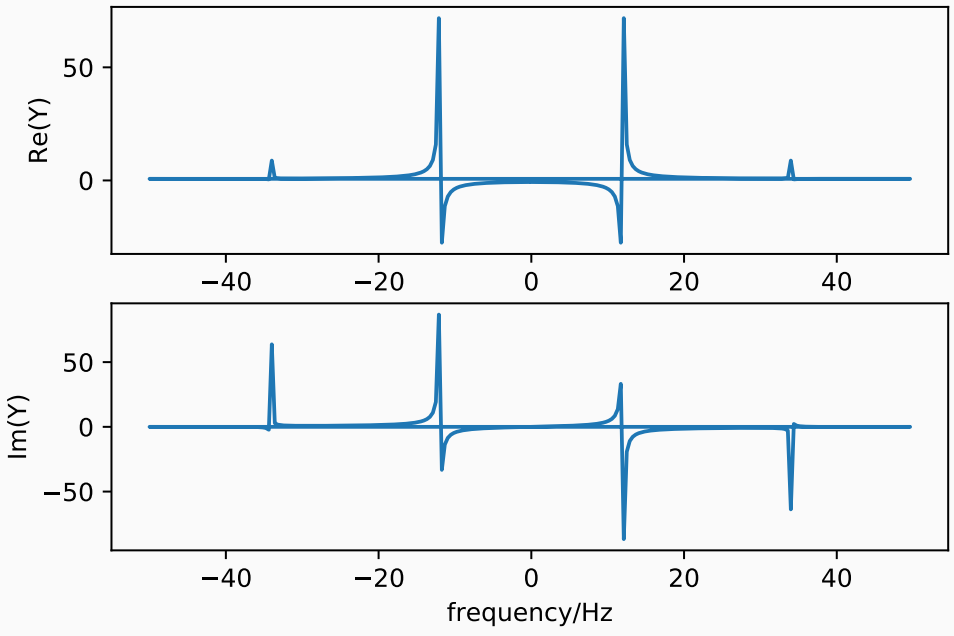
Negative frequencies come *after* positive frequencies

```
Y=np.fft.fft(y)
# Plot FFT modulus versus array index
plt.subplot(2,1,1); plt.plot(abs(Y))
# Now use the correct frequency coordinates
f=np.fft.fftfreq(fftsize,dt)
plt.subplot(2,1,2); plt.plot(f,abs(Y))
```



Negative frequencies are not always needed

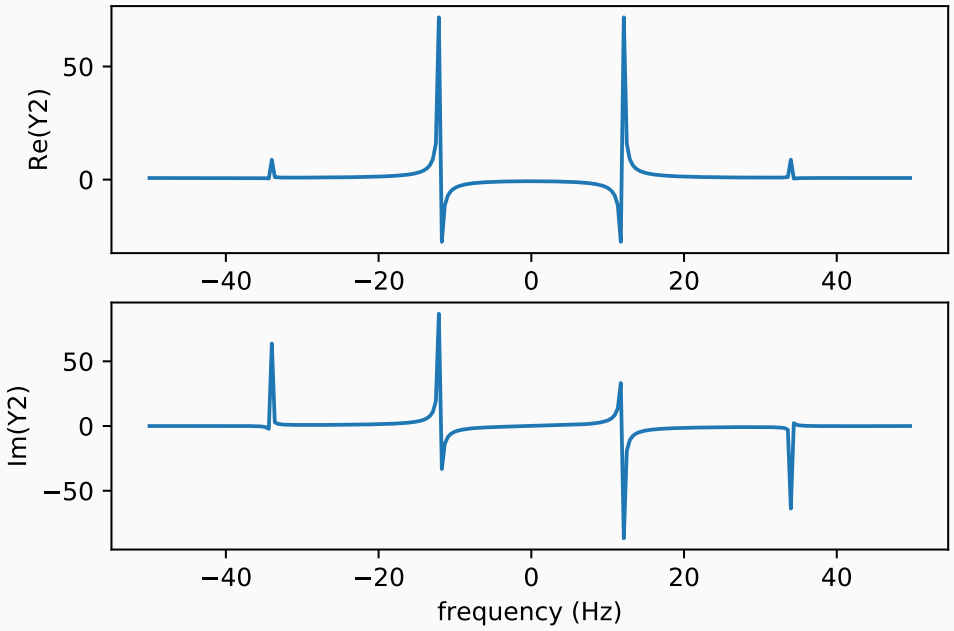
```
plt.subplot(2,1,1); plt.plot(f,Y.real)
plt.subplot(2,1,2); plt.plot(f,Y.imag)
```



Recall that the FT of a real signal has $H(f) = H^*(-f)$

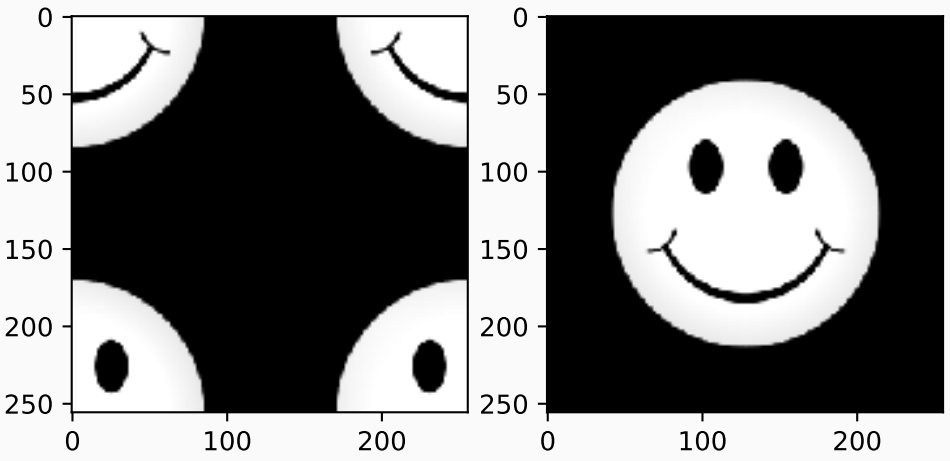
Re-ordering the array makes plots tidier

```
Y2=np.fft.fftshift(Y)
f2=np.fft.fftshift(f)
plt.subplot(2,1,1); plt.plot(f2,Y2.real)
plt.subplot(2,1,2); plt.plot(f2,Y2.imag)
```



In 2-d this helps to visualise Fourier results

```
plt.subplot(1,2,1)
plt.imshow(smiley,cmap="gray")
plt.subplot(1,2,2)
plt.imshow(np.fft.fftshift(smiley),cmap="gray")
```



Fast Fourier Transforms in C++

Key points

- Use a library. `FFTW` is good.
- Understand how the data are stored in memory.

Usually a complex number is just represented by two consecutive foating-point numbers, being the real and imaginary parts. Most libraries expect this! So if you want to FFT N numbers, just create an array of $2N$ values, and store the numbers sequentially as

$$[R_0, I_0, R_1, I_1, \ldots R_{N-1}, I_{N-1}]]$$

where (R_i, I_i) are the real and imaginary parts of the $i'th$ number in the series.

Typical FFT Storage Order for $N = 8$

memory offset	type	time	freq
0	real	$t = 0$	$f = 0$
1	imag	$t = 0$	$f = 0$
2	real	$t = \Delta$	$f = 1/(8\Delta)$
3	imag	$t = \Delta$	$f = 1/(8\Delta)$
4	real	$t = 2\Delta$	$f = 2/(8\Delta)$
5	imag	$t = 2\Delta$	$f = 2/(8\Delta)$
6	real	$t = 3\Delta$	$f = 3/(8\Delta)$
7	imag	$t = 3\Delta$	$f = 3/(8\Delta)$
8	real	$t = 4\Delta$	$f = 4/(8\Delta) = 1/(2\Delta) = f_c$
9	imag	$t = 4\Delta$	$f = 4/(8\Delta) = 1/(2\Delta) = f_c$
10	real	$t = 5\Delta$	$f = -3/(8\Delta)$
11	imag	$t = 5\Delta$	$f = -3/(8\Delta)$
12	real	$t = 6\Delta$	$f = -2/(8\Delta)$
13	imag	$t = 6\Delta$	$f = -2/(8\Delta)$
14	real	$t = 7\Delta$	$f = -1/(8\Delta)$
15	imag	$t = 7\Delta$	$f = -1/(8\Delta)$

Simple FFT example

```
#include <vector>
#include <iostream>
#include <cmath>
#include <fftw3.h>
int main() {
  const int n = 8;
  std::vector<double> inp(2 * n, 0);
  std::vector<double> out(2 * n, 0);
  // Set the real parts to something quasi-random:
  inp[0] = -4; inp[2] = 0; inp[4] = -3; inp[6] = 6;
  inp[8] = -2; inp[10] = 9; inp[12] = -6; inp[14] = 5;
  // FFTW wants the addresses of the input and output arrays, but has
  // its own own complex type (actually a typedef). Use a cast:
  fftw_complex *finp = (fftw_complex *)&inp[0];
  fftw_complex *fout = (fftw_complex *)&out[0];

  fftw_plan plan_forward =
      fftw_plan_dft_1d(n, finp, fout, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute(plan_forward);
```

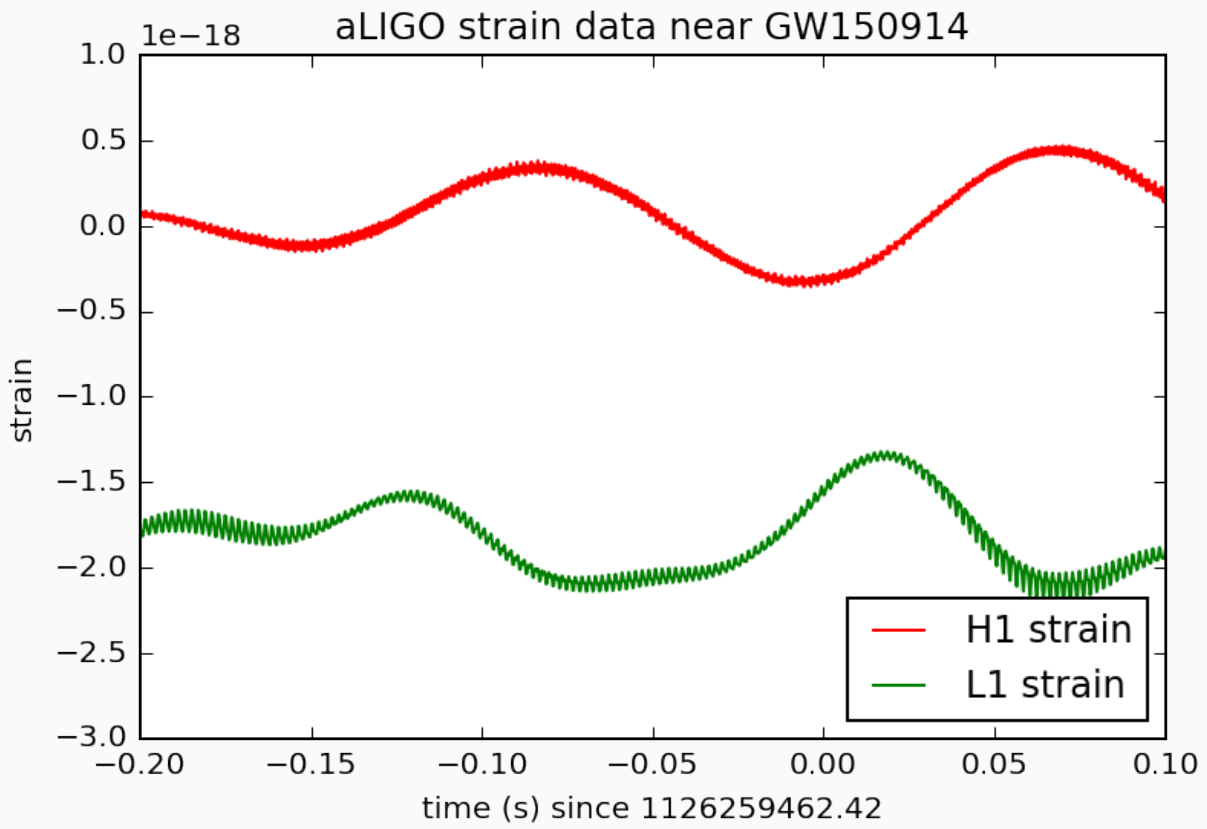

Simple FFT example (cont'd)

```
  std::cout.precision(3);
  for (int i = 0; i < n; i++)
    std::cout << "(" << inp[2 * i] << "," << inp[2 * i + 1] << ")  ("
              << out[2 * i] << ", " << out[2 * i + 1] << ")\n";
  return 0;
}
```

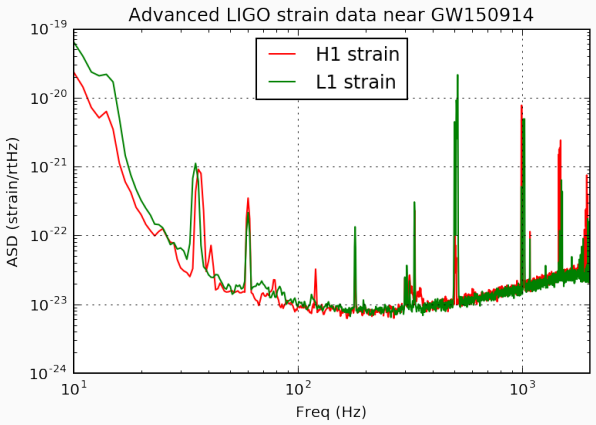
```
(-4,0)  (5, 0)
(0,0)  (-9.07, 2.66)
(-3,0)  (3, 2)
(6,0)  (5.07, 8.66)
(-2,0)  (-35, 0)
(9,0)  (5.07, -8.66)
(-6,0)  (3, -2)
(5,0)  (-9.07, -2.66)
```

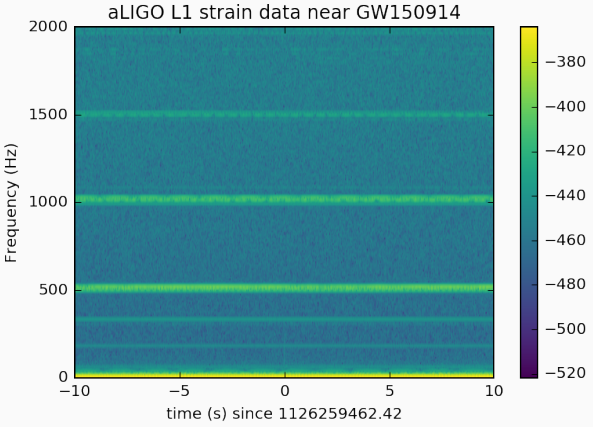
See `fft2.cc` and `fft2f.gp` in the examples on the MCS systems

Signal Processing Example - LIGO



LIGO data in the Fourier domain





Fourier filtered data