CS 294-73
Software Engineering for Scientific Computing

Lecture 11: Fourier transforms; inheritance.
The Fourier Transform

Given a function \( f(x) = f : [0, 1] \to \mathbb{R} \), we can define its Fourier transform

\[
\mathcal{F}(f)_k = \int_{0}^{1} f(x) e^{-2\pi ikx} \, dx , \quad k = 0 \pm 1, \pm 2, \ldots
\]

and its inverse:

\[
\mathcal{F}^{-1}(\hat{f})(x) = \sum_{k=-\infty}^{\infty} \hat{f}_k e^{2\pi ikx} , \quad x \in [0, 1] , \quad \hat{f} : \mathbb{Z} \to \mathbb{R}
\]

This is useful in a number of settings: approximation theory, solving partial differential equations, signal processing.
The Fourier Transform

Examples:

• If $f$ is a $(q+1)$ – times differentiable, periodic function, then

$$f_M(x) - f(x) = O\left(\frac{1}{M^q}\right), \text{ where } f_M(x) = \sum_{k=-M}^{M} \hat{f}_k e^{2\pi i k x}, \hat{f} = \mathcal{F}(f)$$

• If $f$ is a smooth function, then

$$\frac{df}{dx} = \sum_{k=-\infty}^{\infty} i k \hat{f}_k e^{2\pi i k x}, \hat{f} = \mathcal{F}(f)$$

Taken together, these two examples provide a mechanism for computing very accurate approximations to $f$ and its derivatives.
The Fourier Transform

For dimensions greater than 1, can apply the 1D transform in each direction separately.

\[ \mathcal{F}(f)_{\mathbf{k}} = \int f(\mathbf{x}) e^{-2\pi i \mathbf{k} \cdot \mathbf{x}} d\mathbf{x}, \quad \mathbf{k} \in \mathbb{Z}^D, \quad f : [0, 1]^D \to \mathbb{R} \]

\[ = \int_0^1 \cdots \int_0^1 f(x_0, \ldots, x_{D-1}) e^{-2\pi i k_0 x_0} \times \cdots \times e^{-2\pi i k_{D-1} x_{D-1}} dx_0 \cdots dx_{D-1} \]

Then estimates and identities from the 1D case apply to the partial sums and to derivatives:

\[ \frac{\partial f}{\partial x_d} = \sum \nu k_d \hat{f}_k e^{2\pi i \mathbf{k} \cdot \mathbf{x}}, \quad \hat{f} = \mathcal{F}(f) \]
The Discrete Fourier Transform

Given a discrete set of function values on an equally-spaced grid, e.g.,

\[ f_j, \ j = 0, \ldots, N - 1, \ f_j \approx f(jh), \ h = \frac{1}{N}, \ f_N = f_0, \ N \text{ even} \]

we can define its discrete Fourier transform (DFT):

\[
\mathcal{F}_N(f)_k = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-2\pi i k j h}, \ k = -\frac{N}{2} + 1, \ldots, \frac{N}{2}
\]

\[
= \frac{1}{N} \sum_{j=0}^{N-1} f_j z^{kj}, \ z = e^{2\pi i h}
\]

\[
= \langle f, W^{(k)} \rangle, \ \langle q, r \rangle = \frac{1}{N} \sum_{j=0}^{N-1} q_j \bar{r}_j, \ W^{(k)}_j = z^{jk}
\]

We could have used any contiguous set of wave numbers

\[ k \in [k_0, \ldots, k_0 + N - 1] \]

and get back our original series using \( W^{(k)} = W^{(k+N)} (z^N = 1) \).
The Discrete Fourier Transform

The set of vectors \( \{ W^{(k)} \}_{k=1-N/2,\ldots,N/2} \) form an orthonormal basis for \( \mathbb{C}^N \):

\[
\langle W^k, W^{k'} \rangle = \frac{1}{N} \sum_{j=0}^{N-1} z^{(k-k')j}
\]

\[
= 1 \text{ if } k = k'
\]

\[
= \frac{(z^{(k-k')}N - 1)}{z^{(k-k')} - 1} = 0 \text{ if } k \neq k'
\]

So the DFT is invertible, and the inverse given by

\[
f_j = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} \langle f, W^{(k)} \rangle W_j^{(k)} = \frac{1}{N} \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} \mathcal{F}_N(f)_k z^{kj} = \frac{1}{N} \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} \mathcal{F}_N(f)_k e^{2\pi i k j h}
\]

The above relationship is a discrete analogue of Fourier expansion of a function described earlier.
The Discrete Fourier Transform

A discrete Fourier transform has analogous properties to the continuous one.

- Interpolation: if \( f_j = F(jh) \) where \( F \) is \((q+1)\)-times differentiable, then

\[
F(x) - f_N(x) = O\left(\frac{1}{N^q}\right), \quad \text{where} \quad f_N(x) = \frac{1}{N} \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} \mathcal{F}_N(f)_k e^{2\pi ikx}
\]

\[
F(jh) = f_N(jh)
\]
The Discrete Fourier Transform

• Diagonalizing difference operators: \( W_{j+1}^k = z^k W_j^k \)

\[
L(f)_j = \frac{f_{j+1} - 2f_j + f_{j-1}}{h^2}
\]

\[
= \frac{1}{N} \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} \Lambda(k) \mathcal{F}_N(f)_k e^{2\pi i k j h}
\]

\[
\Lambda(k) = \frac{(z^k - 2 + z^{-k})}{h^2}
\]

\[
L^{-1}(f)_j = \frac{1}{N} \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} \frac{1}{\Lambda(k)} \mathcal{F}_N(f)_k e^{2\pi i k j h}
\]
The Discrete Fourier Transform

- All of this extends to multiple dimensions by applying it one dimension at a time:

\[
\mathcal{F}_N(f)_k = \frac{1}{N^D} \sum_{j=0}^{N-1} f_j e^{-2\pi i k \cdot j h}, \quad k \in \left[-\frac{N}{2} + 1, \ldots, \frac{N}{2}\right]^D, \quad f : [0, \ldots, N-1]^D \to \mathbb{R}
\]

\[
= \frac{1}{N^D} \sum_{j_0=0}^{N-1} \cdots \sum_{j_{D-1}=0}^{N-1} f(j_0, \ldots, j_{D-1}) e^{-2\pi i k_0 j_0 h} \times \cdots \times e^{-2\pi i k_{D-1} j_{D-1} h}
\]

\[
f_j = \sum_{k \in \left[-\frac{N}{2} + 1, \ldots, \frac{N}{2}\right]^D} \mathcal{F}_N(f)_k e^{-2\pi i j \cdot k h}
\]

\[
= \sum_{k \in \left[-\frac{N}{2} + 1, \ldots, \frac{N}{2}\right]^D} \langle f, W^{(k)} \rangle W^{(k)}_j
\]

\[
W^{(k)}_j = \prod_{d=0}^{D-1} e^{2\pi i j_d k_d h}
\]
The Discrete Fourier Transform

A straightforward implementation of

\[ f_j = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} \langle f, W^{(k)} \rangle W_j^{(k)} = \frac{1}{N} \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} \mathcal{F}_N(f)_k z^{kj} = \frac{1}{N} \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} \mathcal{F}_N(f)_k e^{2\pi i k j h} \]

costs \(O(N^2)\) operations.

- Each inner product is \(O(N)\), and there are \(N\) of them;
- Computing the sum is also \(O(N)\) operations for each \(j\), and there are \(N\) of those.
- The DFT is multiplication an \(N \times N\) orthogonal matrix.
Fast Fourier Transform (Cooley and Tukey, 1965)

\[ \mathcal{F}_N(f)_k \equiv \sum_{j=0}^{N-1} f_j z^{jk}, \quad z = e^{-2\pi i/N}, \quad k = 0, \ldots, N - 1 \]

\[ = \sum_{j=0}^{N/2-1} f_{2\tilde{j}} z^{2\tilde{j}k} + z^k \sum_{\tilde{j}=0}^{N/2-1} f_{2\tilde{j}+1} z^{2\tilde{j}k} \]

We also have \( \mathcal{F}_N(f)_k = \mathcal{F}_N(f)_{(k+N)} \)

which allows us to obtain \( \mathcal{F}_N(f)_k, -\frac{N}{2} + 1 \leq k < 0 \) from the above sums.

It also allows us to compute \( \mathcal{F}_N(f)_k, k > N/2 \) from

\[ \mathcal{F}_{N/2}(\mathcal{E}(f)), \quad \mathcal{F}_{N/2}(\mathcal{O}(f)) \]

So the number of flops to compute \( \mathcal{F}_N(f)_k \) is 2N, given that you have

\[ \mathcal{F}_{N/2}(\mathcal{E}(f)), \quad \mathcal{F}_{N/2}(\mathcal{O}(f)) \]
Fast Fourier Transform

\[ \mathcal{F}_N(f)_k = \mathcal{F}_{N/2}(\mathcal{E}(f))_k + z_N^k \mathcal{F}_{N/2}(\mathcal{O}(f))_k \]

\[ z_N = e^{2\pi i / N}, \, 0 \leq k < \frac{N}{2} \]

\[ \mathcal{F}_N(f)_{k+N/2} = \mathcal{F}_{N/2}(\mathcal{E}(f))_k + z_N^{k+N/2} \mathcal{F}_{N/2}(\mathcal{O}(f))_k \]

\[ = \mathcal{F}_{N/2}(\mathcal{E}(f))_k - z_N^k \mathcal{F}_{N/2}(\mathcal{O}(f))_k \]

Can continue this process until computing \(2^{M-1}\) sets of \(\mathcal{F}_2\). At each level, the computational cost is \((4 \text{ multiplies } + 6 \text{ adds}) \times N/2\) (the factors \(z^k\) are precomputed once, at a cost of \(O(N)\)). So the total number of flops is \(O(M \times N) = O(N \log N)\). (actually, \(5N \log_2 N\)). (NB: only one complex multiply per pair).

\[(a + i\times b) \times (c + i\times d) = a\times c - b\times d + i\times (a\times d + b\times c)\]

This leads to the following implementation using recursion.
void FFT::applyFFT(vector<complex<double> >& a_fhat, const vector<complex<double> >& a_f, int a_level)
{
    if (level > 1){
        vector<complex<double> > fEven = evenPoints(a_f);
        vector<complex<double> > fOdd = oddPoints(a_f);
        int N = a_f.size(); a_fhat.resize(N);
        vector<complex<double> > fHatHalfEven, fHatHalfOdd;
        applyFFT(fEven, fHatHalfEven, level-1);
        applyFFT(fOdd, fHatHalfOdd, level-1);
        // loop to compute fHatEven[k] + z^k*FhatOdd[k]
    } else
        fHat[0] = a_f[0] + a_f[1]; fHat[1] = a_f[0] - a_f[1];
    // z = -1 at level 1.
};

(We’ll defer a deeper dive into recursion to a later time).
std::complex

(2) Complex type. This is an STL type, templated on the type of the real and imaginary parts (float, double, int). A complex number is stored as a pair (realpart, imaginarypart), in contiguous memory locations. To use it, you need the header

#include <complex>
Using namespace std:
...
complex<int> c(1,2);
complex<int> csq = c*c;
complex<float> c(1.,2.);
complex<float> csq = c*c;
Let’s look a little more closely at what is going on…

\[
\mathcal{F}_{N/2}(\mathcal{E} f)_k = \mathcal{F}_{N/4}(\mathcal{E}^2 f)_k + z^k \mathcal{F}_{N/4}(\mathcal{O}\mathcal{E} f)_k \\
\mathcal{F}_{N/2}(\mathcal{E} f)_{k+N/4} = \mathcal{F}_{N/4}(\mathcal{E}^2 f)_k - z^k \mathcal{F}_{N/4}(\mathcal{O}\mathcal{E} f)_k \\
\mathcal{F}_{N/2}(\mathcal{O} f)_k = \mathcal{F}_{N/4}(\mathcal{E} \mathcal{O} f)_k + z^k \mathcal{F}_{N/4}(\mathcal{O}^2 f)_k \\
\mathcal{F}_{N/2}(\mathcal{O} f)_{k+N/4} = \mathcal{F}_{N/4}(\mathcal{E} \mathcal{O} f)_k - z^k \mathcal{F}_{N/4}(\mathcal{O}^2 f)_k \\
z = e^{2\pi \nu (2/N)}
\]
Fast Fourier Transform

\[ F_{N/4}(\mathcal{E}^2 f)_{k} = F_{N/8}(\mathcal{E}^3 f)_{k} + z^k F_{N/8}(\mathcal{OE}^2 f)_{k} \]

\[ F_{N/4}(\mathcal{E}^2 f)_{k+N/8} = F_{N/8}(\mathcal{E}^3 f)_{k} - z^k F_{N/8}(\mathcal{OE}^2 f)_{k} \]

\[ F_{N/4}(\mathcal{EO} f)_{k} = F_{N/8}(\mathcal{E}^2 O f)_{k} + z^k F_{N/8}(\mathcal{O}E O f)_{k} \]

\[ F_{N/4}(\mathcal{EO} f)_{k+N/8} = F_{N/8}(\mathcal{E}^2 O f)_{k} - z^k F_{N/8}(\mathcal{O}E O f)_{k} \]

\[ F_{N/4}(\mathcal{OE} f)_{k} = F_{N/8}(\mathcal{E}O E f)_{k} + z^k F_{N/8}(O^2 \mathcal{E} f)_{k} \]

\[ F_{N/4}(\mathcal{OE} f)_{k+N/8} = F_{N/8}(\mathcal{E}O E f)_{k} - z^k F_{N/8}(O^2 \mathcal{E} f)_{k} \]

\[ F_{N/4}(O^2 f)_{k} = F_{N/8}(\mathcal{EO}^2 f)_{k} + z^k F_{N/8}(O^3 f)_{k} \]

\[ F_{N/4}(O^2 f)_{k+N/8} = F_{N/8}(\mathcal{EO}^2 f)_{k} - z^k F_{N/8}(O^3 f)_{k} \]

\[ z = e^{2\pi i (4/N)} \]
Update-in-place Cooley-Tukey Implementation

$$\hat{f}_k^{(1)} = f_k + f_{k+\frac{N}{2}}$$
$$\hat{f}_{k+\frac{N}{2}}^{(1)} = f_k - f_{k+\frac{N}{2}}$$

0 ≤ k < $\frac{N}{2}$

Even: bit = 0
Odd: bit = 1
Let’s change how we store the function values in memory, based on reversing the order of the binary digits: \( g_j = \hat{f}_{\mathcal{R}(j,N)} \)

<table>
<thead>
<tr>
<th>Original Ordering</th>
<th>Values for reverse-ordered indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 001 010 011 100 101 110 111</td>
<td>000 100 010 110 001 101 011 111</td>
</tr>
<tr>
<td>000 001 010 011 100 101 110 111</td>
<td>000 100 010 110 001 101 011 111</td>
</tr>
<tr>
<td>000 001 010 011 100 101 110 111</td>
<td>000 100 010 110 001 101 011 111</td>
</tr>
<tr>
<td>000 001 010 011 100 101 110 111</td>
<td>000 100 010 110 001 101 011 111</td>
</tr>
</tbody>
</table>

No recursion - expressed as a loop instead. Entirely update in place, with a pair of complex numbers as temporaries. Memory locality?
Split-radix formulation

An FFT of size $N = N_1N_2$ can be expressed as a composition of two sets of FFTs, one set of length $N_1$, the other set of length $N_2$, with a set of $N_1N_2$ multiplications in between

$$\hat{f}_{k_2+N_2k_1} = \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} f_{j_2N_1+j_1} z_N^{(j_2N_1+j_1)k}, \quad k = k_2 + N_2k_1, \quad N = N_1N_2$$

$$= \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} f_{j_2N_1+j_1} z_N^{j_2N_1(k_2+N_2k_1)} z_N^{j_1k}$$

$$= \sum_{j_1=0}^{N_1-1} \left( \sum_{j_2=0}^{N_2-1} f_{j_2N_1+j_1} z_N^{j_2k_2} \right) z_N^{j_1k} \quad (z_N^{N_2} = z_N, z_N^{N_1N_2} = 1)$$

$$= \sum_{j_1=0}^{N_1-1} z_N^{j_1k_1} \left( \sum_{j_2=0}^{N_2-1} f_{j_2N_1+j_1} z_N^{j_2k_2} \right) z_N^{j_1k_2} \quad (z_N^{j_1(k_2+k_1N_2)} = z_N^{j_1k_2} z_N^{j_1k_1})$$

Split-radix formulation of FFT

N₂ = 4, N₁ = 2

N₂ = 2, N₁ = 4

N₂ = 5, N₁ = 3. Note the data motion! Also, connection with multidimensional FFTs
Performance Drivers

- $N_2$ transform accesses data with stride $N_1$; the $N_1$ transform accesses data with stride 1.
- How deeply to apply recursion?
- If the transforms are small enough, it may cost less to apply the matrix-multiplication form of the DFT.
- Some transforms require much less floating point than $5N\log_2(N)$.

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
l & -1 & -l & 1 \\
-1 & 1 & -1 & 1 \\
-l & -1 & l & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
\frac{i\sqrt{2}+\sqrt{2}}{2} & i & \frac{i\sqrt{2}-\sqrt{2}}{2} & -1 \\
\frac{-1}{2} & -1 & -l & 1 \\
\frac{-i\sqrt{2}+\sqrt{2}}{2} & -l & \frac{i\sqrt{2}+\sqrt{2}}{2} & -1 \\
\frac{-i\sqrt{2}-\sqrt{2}}{2} & l & \frac{-i\sqrt{2}+\sqrt{2}}{2} & 1 \\
\end{bmatrix}
\]

No flops at all!

\[
(2 \text{ adds } + 2 \text{ multiplies}) \times 16 = 64 \text{ flops} < 120 \text{ flops}
\]

- There is software written by pros out there, and you want to use it.
Hockney’s method for discrete convolution (1D case)

Want to compute

\[(G \ast f)_j \equiv \sum_{j \in \mathbb{Z}} G_{i-j} f_j, \ i \in [0, \ldots N - 1]\]

\[f_l \equiv 0 \text{ unless } l \in [0, \ldots, N - 1]\]

We make two observations.

(1) Since we can replace the infinite sum with a finite sum: for any \(b \geq N\),

\[\sum_{j \in \mathbb{Z}} G_{i-j} f_j = \sum_{i=-b}^{N+b} G_{i-j} f_j, \ i \in [0, \ldots, N - 1]\]

(2) Again, if \(b \geq N\), we can replace \(f\) and \(G\) in the previous expression by their periodic extensions:

\[\sum_{i=-b}^{N+b} G_{i-j} f_j = \sum_{i=-b}^{N+b} \tilde{G}_{i-j} \tilde{f}_j, \ i \in [0, \ldots, N - 1]\]

\[\tilde{q}_j = q_j, \ j \in [-b, \ldots N + b]\]

\[= q_{\text{mod}(j+b,N+2b+1)-b} \text{ otherwise}\]
Hockney’s method for discrete convolution

(3) We can now transform the calculation to computing three FFTs.

\[
\hat{q}_k = \sum_{j'=0}^{N+2b} \tilde{q}_{j'-b} z^{k j'}, \quad \tilde{q}_{j'-b} = \frac{1}{N + 2b + 1} \sum_{k=0}^{N+2b} \hat{q}_k z^{-k j'}
\]

\[
(G \ast f)_{i'-b} = \sum_{j'=0}^{N+2b} \tilde{G}_{i'-j'} \tilde{f}_{j'-b}
\]

\[
= \frac{1}{N + 2b + 1} \sum_{j=0}^{N+2b} \sum_{k=0}^{N+2b} \hat{G}_k z^{-(i'-j')k} f_{j'-b}
\]

\[
= \frac{1}{N + 2b + 1} \sum_{k=0}^{N+2b} \sum_{j'=0}^{N+2b} z^{-i'k} \sum_{j'=0}^{N+2b} z^{j'k} f_{j'-b}
\]

\[
= \frac{1}{N + 2b + 1} \sum_{k=0}^{N+2b} z^{-i'k} \hat{G}_k \hat{f}_k
\]

This extends easily to any number of dimensions. Also, \( b \) becomes a tuning parameter.
Final Comments on the mathematics

- Sometimes you need to transform back to centered and normalized form

\[ \mathcal{F}_N(f)_k = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-2\pi i k j h}, \quad k = -\frac{N}{2} + 1, \ldots, \frac{N}{2} \]

\[ \mathcal{F}_N(f)_k = \mathcal{F}_N(f)_{(k+N)}, \quad -\frac{N}{2} + 1 \leq k < 0 \]

A similar manipulation is required for the inverse transform. This form is customary in order to obtain a real interpolant from real data: \( \hat{f}_0, \frac{N}{2} \) are real, as are their corresponding discrete Fourier modes, and

\[ \tilde{f}_k = \hat{f}_{-k}, \quad k = 1, \ldots, \frac{N}{2} - 1 \]

- sine and cosine transforms. These are useful, particularly when solving PDE, in order to allow you to use Dirichlet (sine) or Neumann (cosine) boundary conditions.
class FFT1D {
Public:
    FFT1D();
    // Constructor. argument a_M specifies number of points is N= 2^{a_M}
    FFT1D(int a_M){m_M = a_M; m_N = Power(2,m_M);};
    // Forward FFT: a_fHat[k] = \sum_{j=0}^{N-1} a_f[j] z^{j \cdot k},
    // z = e^{-2 \pi \iota / m_N}
    void forwardFFTCC(vector<complex<double> > & a_fHat, const vector<complex<double> >& f) const;
    // inverse FFT: a_f[j] = \sum_{k=0}^{N-1} a_fHat[k] z^{j \cdot k},
    // z = e^{2 \pi \iota / m_N}
    void inverseFFTCC(vector<complex<double> > & a_f, const vector<complex<double> > & a_fHat) const;
    // Access functions.
    const int& getN(){return m_N;};
    const int& getM(){return m_M;}
protected:
    int m_M, m_N;
}
class FFTMD
{
public:
    FFTMD();
    FFTMD(int a_M);
    void forwardCC(RectMDArray<complex<double> > & a_f) const;
    void inverseCC(RectMDArray<complex<double> > & a_fHat) const;
    const int& getN() const;
    const int& getM() const;
private:
    int m_N;
    int m_M; // m_N = 2^m_M
    FFT1D m_fft1d;
};
void FFTMD::forwardCC(RectMDArray<complex<double> > & a_f) const
{
    int low[DIM], high[DIM];
    vector<complex<double> > f1d(m_N);
    vector<complex<double> > fHat1d(m_N);
    for (int dir = 0; dir < DIM; dir++) {
        for (int dir2 = 0; dir2 < DIM; dir2++) {
            low[dir2] = 0;
            high[dir2] = m_N - 1;
        }
        Point edir = getUnitv(dir);
        high[dir] = 0;
        Box base(low, high);
        for (Point pt = base.getLowCorner(); base.notDone(pt); base.increment(pt)) {
            for (int l = 0; l < m_N; l++) {
                f1d[l] = a_f[pt + edir * l];
            }
            m_fft1d.forwardFFTCC(fHat1d, f1d);
            for (int l = 0; l < m_N; l++) {
                a_f[pt + edir * l] = fHat1d[l];
            }
        }
    }
}
Pointers to Functions -> Inheritance

- You would like to use different functions that do the same thing: FFTW1D, FFT1DBRI, FFT1DRecursive. Want to do that in something other than a text-editing fashion.

- In C and Fortran, you can hand around pointers to functions. Makes sense from a low-level execution standpoint (a function call hands off control to an address), and from a low-level programming standpoint (addresses are of fixed length, so call-by-value on addresses makes sense).

- Want to make this type-safe, e.g. make sure function signatures conform (in fact we get more).

- This leads to inheritance.
  - Base class: FFT1D defines the interface, i.e. function signatures.
  - Derived classes: FFTW1D, FFT1DBRI, FFT1DRecursive provide implementations that conform to these signatures.
class FFT1D{
public:
    // Interface class for complex-to-complex power-of-two FFT on the unit interval.
    FFT1D();
    // Constructor. argument a_M specifies number of points is N = 2^{a_M}
    FFT1D(int a_M){m_M = a_M; m_N = Power(2, m_M);};
    // Forward FFT: a_fHat[k] = \sum_j=0^{N-1} a_f[j] z^{j k},
    // z = e^{-2 \pi \iota / m_N}
    virtual void forwardFFTCC(vector<complex<double> > & a_fHat,
                                const vector<complex<double> > & f) const = 0;
    // inverse FFT: a_f[j] = \sum_{k=0}^{N-1} a_fHat[k] z^{j k},
    // z = e^{2 \pi \iota / m_N}
    virtual void inverseFFTCC(vector<complex<double> > & a_f,
                                const vector<complex<double> > & a_fHat) const = 0;
    // Access functions.
    const int& getN(){return m_N;};
    const int& getM(){return m_M;}
protected:
    int m_M, m_N;
}

virtual: denotes a member function that can be (re)defined by the derived class.
virtual ... =0: denotes a member function that must be defined by the derived class.
#include "FFT1D.H"

class FFT1DBRI:

    public FFT1DBRI:
    {
        public:
            FFT1DBRI();
            FFT1DBRI(const int& a_M);
            virtual void forwardFFTCC(vector<complex<double> > & a_fHat, const vector<complex<double> >& f) const;
            virtual void inverseFFTCC(vector<complex<double> > & a_f, const vector<complex<double> > & a_fHat) const;
    };
#include "FFT1D.H"

class FFTMD
{
public:
    FFTMD();
    FFTMD(FFT1D* a_fft1dPtr); // new constructor.
    void forwardCC(RectMDArray<complex<double> > & a_f) const;
    void inverseCC(RectMDArray<complex<double> > & a_fHat) const;
    const int& getN() const;
    const int& getM() const;

private:
    int m_N;
    int m_M; // m_N = 2^m_M
    FFT1D* m_fft1dPtr;
};
Multidimensional FFT example

```cpp
void FFTMD::forwardCC(RectMDArray<complex<double> > & a_f) const
{
    int low[DIM], high[DIM];
    vector<complex<double> > f1d(m_N);
    vector<complex<double> > fHat1d(m_N);
    for (int dir = 0; dir < DIM; dir++){
        for (int dir2 = 0; dir2 < DIM; dir2++){
            low[dir2] = 0;
            high[dir2] = m_N-1;
        }
        Point edir = getUnitv(dir);
        high[dir]=0;
        Box base(low, high);
        for (Point pt=base.getLowCorner(); base.notDone(pt); base.increment(pt)){
            for (int l = 0 ; l < m_N; l++)
            {
                f1d[l] = a_f[pt+mdir*l];
            }
            m_fft1dPtr->forwardFFTCC(fHat1d, f1d);
            for (int l = 0 ; l < m_N; l++)
            {
                a_f[pt+mdir*l] = fHat1d[l];
            }
        }
    }
}
```
Using Derived Classes and Casting

int main(int argc, char* argv[]) {
...
    FFT1D* p_fft;
    FFT1DRecursive* p_fft1dR;
    FFT1DBRI* p_fft1dBRI;
    ...
    else if (fft_string == "BRI") {
        p_fft1dBRI = new FFT1DBRI(M);
        p_fft = dynamic_cast<FFT1D*>(p_fft1dBRI);
    }
    ...
    FFTMD foo(p_fft);
    // Now you’re good to go.
    ...
}

• (Pointers to) base class and derived class are different types.
• Use `dynamic_cast` to perform type conversion.
Why is this powerful?

- FFT1d can be used without recoding.
  - FFT1D defines an interface for the classes that interact with it.

- In fancier terms, this is a mechanism for “Programming by Contract”. The Base Class defines the contract that the object users expect to be able to call.

- You have already been programming to a contract in your homeworks: Our header files have declared classes that you have been asked to define. Also, class template parameters typically define a contract.
What is the language of these contracts?

**virtual declaration** (this keyword is only used in declarations, not definitions)
- keyword indicates that the derived class *can* override this function with its own implementation.
- “=0;” syntax means the derived class *must* override this function, as the base class does not provide a default definition.
  - classes with at least one “=0;” virtual function are referred to as “pure virtual” class or an “abstract class”.
  - You cannot instantiate an abstract class.

**protected:**
- added member declaration keyword. (now have public, private, protected)
- derived classes can see and manipulate public and protected data.
- derived classes cannot access private member data.
Templates versus inheritance

• Both provide mechanisms for reuse.

• Templates: (text editing) + (type checking). Everything is done at compile time.

• Inheritance: uses pointers. Mixture of compile time (checking function signatures, base vs. derived classes) and run time (pointers defined at run time). More general and powerful, e.g. derivation chains, multiple inheritance. Interaction with the type system is subtle.

• Conservative rule-of-thumb: templates for containers, inheritance for function pointers / interfaces.
Managing pointered data.

```cpp
DerivedClass* dptr = new DerivedClass(...);
BaseClass* bptr = dynamic_cast<BaseClass> dptr;
UserClass(bptr ...);
...
Class UserClass(BaseClass* a_bptr, ...)
{
    UserClass(BaseClass* a_bptr){m_bptr = a_bptr;...};
`~UserClass(){delete m_bptr; ...};
private:
    m_bptr;
}
Really dangerous, and unavoidably so: new is called outside the class definition that is going to use the derived class. Who should call delete?
Pointer Safety

• STL provides a mechanism for “safe” pointers, i.e. ones for which memory management is automatic.

• Main example: `shared_ptr<T>`.
  - Created once.
  - Assignment increments a counter.
  - Going out of scope, reassignment decrements a counter.
  - Counter = 0 calls `delete`.

• Example: metadata holder for unions of rectangles.
  - Want to create one, but use it in multiple contexts.
  - Too big to allow large numbers of copies floating around (really a parallel computing issue).
class FFTMD
{
public:
    FFTMD();
    FFTMD(shared_ptr<FFT1D> a_fft1dPtr);
    void forwardCC(RectMDArray<complex<double> > & a_f) const;
    void inverseCC(RectMDArray<complex<double> > & a_fHat) const;
    const int & getN() const;
    const int & getM() const;
private:
    int m_N;
    int m_M;
    shared_ptr<FFT1D> m_fft1dPtr;
};
Derived Classes and Casting

/* FFT1D*  p_fft;
 FFT1DRecursive* p_fft1dR;
 FFT1DBRI* p_fft1dBRI; */

shared_ptr<FFT1D> p_fft;

...

else if (fft_string == "BRI")
{
/* p_fft1dBRI = new FFT1DBRI(M);
 p_fft = dynamic_cast<FFT1D*>(p_fft1dBRI); */

    p_fft1dBRI = shared_ptr<FFT1DBRI>(new FFT1DBRI(M));
    p_fft = dynamic_pointer_cast<FFT1D>(p_fft1dBRI);
}
...

FFTMD foo(p_fft);
// Now you’re good to go.
#ifndef _SPARSEMATRIX_H_
define _SPARSEMATRIX_H_

#include <vector>
#include <cassert>
#include <cmath>
using namespace std;
class SparseMatrix
{
public:
    /// set up an M rows and N columns sparse matrix with all
    /// values of zero (no non-zero elements)
    SparseMatrix();
    SparseMatrix(int a_M, int a_N);

    /// Matrix Vector multiply. a_v.size()==N,
    /// returns vector of size M
    vector<double> operator*(const vector<double>& a_v) const;
}
Represent matrix using

```cpp
unsigned int m_m, m_n;
double m_zero;
vector<vector<double> > m_data;
vector<vector<int> > m_colIndex;
```

`m_data` and `m_colIndex` are of length `m_m`; the \( i \)th element contains the description of the \( i \)th row the matrix with the zeros compressed out:

\[
A_{i,m\_colIndex[i][j]} = m\_data[i][j] \quad \text{if that entry is nonzero; otherwise the entry is assumed to be zero.}
\]
Sparse Matrix Class Questions

```cpp
unsigned int m_m, m_n;
double m_zero;
vector<vector<double> > m_data;
vector<vector<int> > m_colIndex;
```

\[ A[tuple] = \ldots (\text{non-const indexing operator}). \]
If there is an entry for \( \text{tuple}[0], \text{tuple}[1] \), returns a reference to the correct element of \( m\_data[\text{tuple}[0]] \);
If not, add a new element to \( m\_data[\text{tuple}[0]] \) and \( m\_colIndex[\text{tuple}[0]] \)
Using `push_back`. In either case, you need to search through \( m\_colIndex[\text{tuple}[0]] \) to see whether you have a nonzero in column \( \text{tuple}[1] \).

Note that the columns are not sorted in any particular order. This is ok, because matrix multiplication is given by

\[ v[i] = \sum_q m\_data[i][q] \times w[\text{colIndex}[i][q]] \]

Why might this be an acceptable strategy from a performance standpoint?
Another use case: reusing metadata.

Go back to our structured grid example, but now we want to define data on unions of rectangles. Data: the values of the array. Metadata: the skeleton key of the data that allows you to access it.

- For a single rectangle, checking to see whether the boxes are the same is cheap, and storing the box is a small overhead.
- For unions of rectangles, the corresponding information regarding unions of rectangles is more expensive to compute, store, and answer questions about. On multiprocessor systems, you need one copy per processor (or distribute your metadata, which is really complicated.).
BoxLayout.H (fixed-size boxes)

...
#include <memory>
Using namespace std;
...
{
...
private:
  Box m_domain; ///< Box representing the physical domain.

  Box m_bitbox; ///< Each Point in m_bitbox corresponds to a block.

  int m_blockPower; ///< N = 2^m_blockPower.

  int m_blocksize; ///< Size of a single block.

  shared_ptr<RectMDArray<bool>> m_bitmap; ///< Boolean, scalar valued RectMDArray.

  shared_ptr<vector<Point>> m_patchlocs; ///< Which blocks are in the domain.

  shared_ptr<map<Point, int>> m_getPatches; ///< Where is a Point stored in m_patchLocs?
  // Needed to build vector<RectMDArray<T>> of dataholders.
};
BoxLayout Constructor

BoxLayout::BoxLayout(int a_domainExp, const vector<Point>& a_points)
{
    ...
    m_domain = m_bitbox.refine(m_blocksize);
    m_bitmap = shared_ptr<RectMDArray<bool>> (new RectMDArray<bool>(m_bitbox));
    m_patchlocs = shared_ptr<vector<Point>> (new vector<Point> );
    m_getPatches = shared_ptr<map<Point,int>> (new map<Point,int> );
    m_bitmap->setVal(false); // Initialize m_bitmap as False everywhere.
    *m_patchlocs = a_points;
    int counter = 0;

    // Iterate through all Points in a_points.
    // Set corresponding values in m_bitmap to True.
    // Store a related index in m_getPatches with key equal to the associated Point.

    for (auto it = m_patchlocs->begin(); it != m_patchlocs->end(); ++it)
    {
        (*m_bitmap)[*it] = true;
        int index = it - m_patchlocs->begin();
        (*m_getPatches)[*it] = index;
    }
}

What happens to this object on assignment (when all member data are copied using the assignment operator)?