CS 294-73
Software Engineering for Scientific Computing

Lecture 12

Sparse Matrix Class

```
#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;
```

Sparse Matrix Class Questions

Represent matrix using

```
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 ith element contains the description of the ith row the matrix with the zeros compressed out:

$A_{i,\texttt{m\_colIndex[i][j]}}$ = `m_data[i][j]` if that entry is nonzero; otherwise the entry is assumed to be zero.

Sparse Matrix Class Questions

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

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

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

```
v[i] = \sum_q m_data[i][q] w[colIndex[i][q]]
```

Why might this be an acceptable strategy from a performance standpoint ?

Recursion

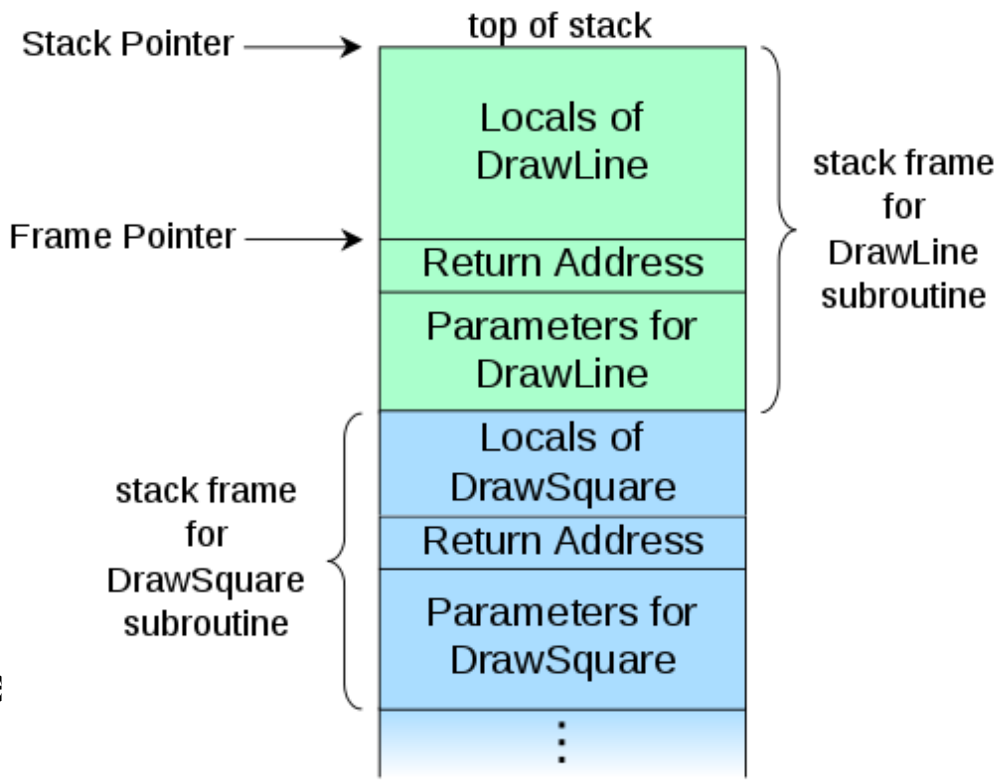
```
int factorial(int n)
{
    if (n > 1)
      {
        return n*factorial(n-1);
      }
      else if (n == 1)
      {
      return 1;
      }
      else
      {
            ... // error condition.
      }
};
```

Call Stack / Stack Frame

What happens when you call a function ?

- Jump to address of the function.
- Set up storage for local scope. This is put on the call stack
  - Values of the arguments.
  - Storage for local variables.

  Size is known at compile time.
- The reason this is a stack is that the corresponding values for the calling function are also stored there, ready to be used when the current function finishes and returns control to the calling function.
- The reason recursion works is that each successive call generates a new stack frame.



What about member functions of an object ?

```
void FFT::applyFFT(vector<complex<double> >& a_fhat,
            const vector<complex<double> >& a_f, int a_level)
{
...
applyFFT(fEven,fHatHalfEven,level-1);
applyFFT(fOdd,fHatHalfOdd,level-1);
...
}
```

- `applyFFT(...) <-> (*this).applyFFT(...)`
- Calling a member function is the same as calling a function with `this` as an additional argument (i.e. a pointer to the member data).
- Class = functions + C Struct ;

Hockney's method for discrete convolution (1D case)

Want to compute

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

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

We make two observations.
(1) Since we can replace the infinite sum with a finite sum: for any $b \geq N - 1$

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

(2) Again, if $b \geq N - 1$, we can replace f and G in the previous expression by their periodic extensions:

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

$$\begin{aligned} \tilde{q}_j &= q_j \ , \ j \in [-b, \dots N-1] \\ &= q_{mod(j+b, N+b)-b} \text{ otherwise} \end{aligned}$$

Hockney's method for discrete convolution

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

$$\hat{q}_k = \sum_{j'=0}^{N+b-1} q_{j'-b} z^{kj'} \ , \ q_{j'-b} = \frac{1}{N+b} \sum_{k=0}^{N+b-1} \hat{q}_k z^{-kj'}$$

$$(G * f)_{i'-b} = \sum_{j'=0}^{N+b-1} \tilde{G}_{i'-j'} \tilde{f}_{j'-b}$$

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

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

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

Hockney’s method for discrete convolution

What’s wrong with this ?

$$\hat{q}_k = \sum_{j'=0}^{N+b-1} q_{j'-b} z^{kj'} \; , \; q_{j'-b} = \frac{1}{N+b} \sum_{k=0}^{N+b-1} \hat{q}_k z^{-kj'}$$

If N = 64, the smallest possible b = 63, and N+b = 127, which is prime.

Solution: choose a slightly larger b that is more composite. In this case, b = 64, and N+b = 128.

Can use this idea to perform prime-radix FFTs (i.e. Rader’s method): convert to periodic convolution, then choose slightly larger radix for the FFTs.

FFT1D Class

```
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 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 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;
}
```

`FFTMD`: Multidimensional FFT Class

```
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;
};
```

Multidimensional FFT example

```
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.

FFT1D Base Class

```
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.

Class Derived from FFT1D

```
#include "FFT1D.H"
class FFT1DBRI:public FFT1D
{
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;
};
```

Multidimensional FFT example

```
#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

```
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_fft1dPtr->forwardFFTCC(fHat1d,f1d);
          for (int l = 0 ; l < m_N;l++){
              a_f[pt+edir*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. Specific to casting derived-class pointers to base-class pointers.

Other forms of type conversion via casting

- `reinterpret_cast<type_name>(expression)` – treat `expression` as if it were of type `type_name`.
- `const_cast<type_name>(expression)` returns a value of a new type that overrides `const` –ness. Doesn’t work on function pointers.

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 it's 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.

```
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).

Multidimensional FFT example

```
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;
};
```

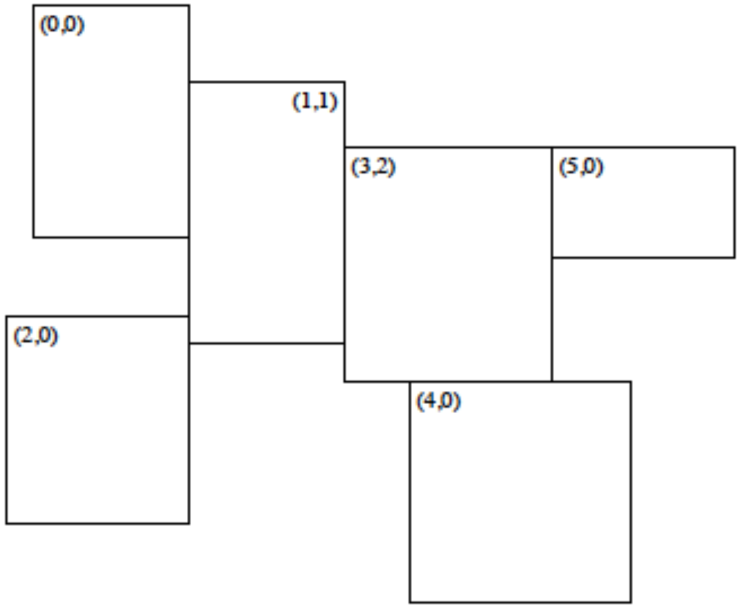
Casting Shared Pointers

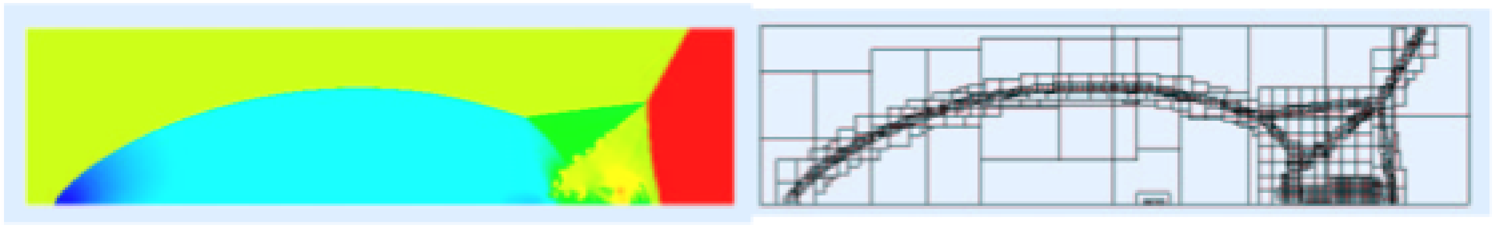
```
/* FFT1D* p_fft;
   FFT1DRecursive* p_fft1dR;
   FFT1DBRI* p_fft1dBRI; */

  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.
```

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.
};
```

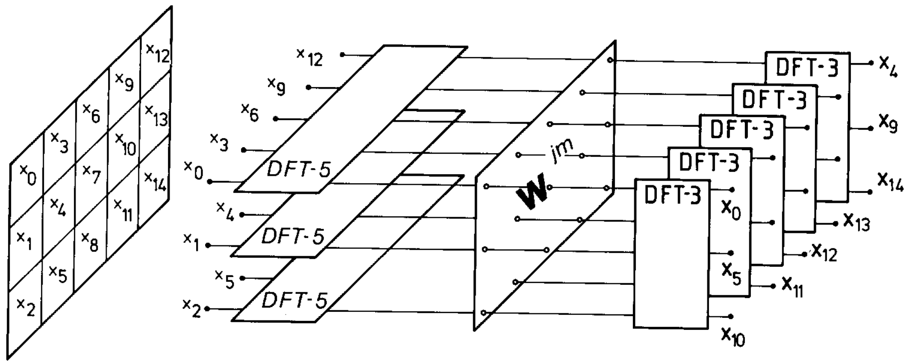
BoxLayout Constructor

```
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) ?

FFTW: http://www.fftw.org/

- Open-source tuned FFT.
- Based on split-radix formalism. with highly-optimized, machine-generated *codelets* written in C to implement short FFTs with non-unit stride.
- De-facto standard.
- Similar approach – Spiral (used mainly for DSP).



Using FFTW

FFTW uses an explicit mapping of the addresses in the inputs / outputs.
The information is stored in “plans”, that are explicitly created / destroyed.
Two modes: estimated tuning, and run-time tuning.

```
“include fftw3.h”
fftw_plan forward,reverse;
fftw_complex* in; fftw_complex* out;
vector<complex<double> > vec_in(N), vec_out(N);
in  = reinterpret_cast<fftw_complex*>(&(vec_in[0]));
out = reinterpret_cast<fftw_complex*>(&(vec_out[0]));

forward=
fftw_plan_dft_1d(N, in, out, FFTW_FORWARD,  FFTW_ESTIMATE);
inverse=
fftw_plan_dft_1d(N, in, out, FFTW_BACKWARD, FFTW_ESTIMATE);
...
fftw_execute(forward);
fftw_execute(reverse);
...
fftw_destroy_plan(forward);
fftw_destroy_plan(reverse);
```

Looking at codelets

- `cd codelets`
- `ls`
- n1_10.c n1_11.c n1_12.c n1_13.c n1_14.c ...
- t1_10.c t1_12.c t1_15.c t1_16.c t1_2.c t1_20.c t1_25.c …

Looking at codelets

```
n1_4.c

...
/*
 * This function contains 16 FP additions, 0 FP multiplications,
 * (or, 16 additions, 0 multiplications, 0 fused multiply/add),
 * 13 stack variables, 0 constants, and 16 memory accesses
 */
...
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ \iota & -1 & -\iota & 1 \\ -1 & 1 & -1 & 1 \\ -\iota & -1 & \iota & 1 \end{bmatrix}$$

(5 n log_2(n) = 40, n^2 = 16. About 100 LOC).

Looking at codelets

```
n1_8.c

...
/*
 * This function contains 52 FP additions, 8 FP multiplications,
 * (or, 44 additions, 0 multiplications, 8 fused multiply/add),
 * 36 stack variables, 1 constants, and 32 memory accesses
 */
...
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \frac{\iota\sqrt{2}+\sqrt{2}}{2} & \iota & \frac{\iota\sqrt{2}-\sqrt{2}}{2} & -1 & -\frac{\iota\sqrt{2}+\sqrt{2}}{2} & -\iota & -\frac{\iota\sqrt{2}-\sqrt{2}}{2} & 1 \\ \iota & -1 & -\iota & 1 & \iota & -1 & -\iota & 1 \\ \frac{\iota\sqrt{2}-\sqrt{2}}{2} & -\iota & \frac{\iota\sqrt{2}+\sqrt{2}}{2} & -1 & -\frac{\iota\sqrt{2}-\sqrt{2}}{2} & \iota & -\frac{\iota\sqrt{2}+\sqrt{2}}{2} & 1 \\ -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ -\frac{\iota\sqrt{2}+\sqrt{2}}{2} & \iota & -\frac{\iota\sqrt{2}-\sqrt{2}}{2} & -1 & \frac{\iota\sqrt{2}+\sqrt{2}}{2} & -\iota & \frac{\iota\sqrt{2}-\sqrt{2}}{2} & 1 \\ -\iota & -1 & \iota & 1 & -\iota & -1 & \iota & 1 \\ -\frac{\iota\sqrt{2}-\sqrt{2}}{2} & -\iota & -\frac{\iota\sqrt{2}+\sqrt{2}}{2} & -1 & \frac{\iota\sqrt{2}-\sqrt{2}}{2} & \iota & \frac{\iota\sqrt{2}+\sqrt{2}}{2} & 1 \end{bmatrix} \qquad (1)$$

(5 n log_2(n) = 120 , n^2 = 64. About 200 LOC)

Looking at codelets

```
n1_13.c

...
/*
 * This function contains 176 FP additions, 114 FP multiplications,
 * (or, 62 additions, 0 multiplications, 114 fused multiply/add),
 * 87 stack variables, 25 constants, and 52 memory accesses
 */
...
```

(5 n log_2(n) = 241 , n^2 = 169. About 650 LOC.)

Looking at codelets

```
n1_64.c

...
/*
 * This function contains 912 FP additions, 392 FP multiplications,
 * (or, 520 additions, 0 multiplications, 392 fused multiply/add),
 * 202 stack variables, 15 constants, and 256 memory accesses
 */
...
```

(5 n log_2(n) = 2560 , n^2 = 4096. About 2950 LOC.)