CS 294-73
Software Engineering for Scientific Computing

Lecture 12
Doolittle’s Method for LU Decomposition

\[ A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & & & a_{NN} \end{pmatrix} = A^{(0)} \]

\[ A^{(1)} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ 0 & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{N2} & \cdots & a_{NN} \end{pmatrix} = L^{(1)}A^{(0)} , \quad L^{(1)} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ -l_{21}^{(1)} & 1 & \cdots & 0 \\ \vdots & 0 & \ddots & 0 \\ -l_{N1}^{(1)} & 0 & \cdots & 1 \end{pmatrix} , \quad l_{n1}^{(1)} = \frac{a_{n1}^{(0)}}{a_{11}^{(0)}} \]

\[ A^{(n)} = L^{(n)}A^{(n-1)} , \quad L^{(n)} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & 0 \\ 0 & \cdots & \cdots & \cdots \end{pmatrix} , \quad l_{kn}^{(n)} = \frac{a_{kn}^{(n-1)}}{a_{nn}^{(n-1)}} , \quad k > n \]
Doolittle’s Method for LU Decomposition

\[ U = A^{(N)} = L^{(N)} \ldots L^{(1)} A \]

\[ A = L U = (L^{(1)})^{-1} \ldots (L^{(N)})^{-1} U \]

\[
(L^{(n)})^{-1} = \begin{pmatrix}
1 & 0 & \cdots & 0 \\
0 & \ddots & \cdots & 0 \\
0 & \cdots & 1 \\
0 & \cdots & \cdots & 1 \\
\end{pmatrix}
\]

\[
L = \begin{pmatrix}
1 & 0 & \cdots & 0 \\
l_{21}^{(1)} & \ddots & \cdots & 0 \\
\vdots & \ddots & \ddots & \vdots \\
l_{N1}^{(1)} & \cdots & l_{Nn}^{(n)} & 1 \\
\end{pmatrix}
\]

This continues to work if we replace \( a_{ij} \) by bxb blocks. Then \[
\frac{a_{n1}}{a_{11}} \rightarrow a_{n1} a_{11}^{-1}
\]

etc.
Sparse Matrix Class

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`\textsuperscript{th} element contains the description of the `i`\textsuperscript{th} row the matrix with the zeros compressed out:

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

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

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 column tuple[1].

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

\[ v[i] = \sum_q m_{data}[i][q] \cdot w[colIndex[i][q]] \]

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

Long long factorial(int n)
{
    if (n > 1)
    {
        return n*factorial(n-1);
    }
    else if (n == 1)
    {
        return 1;
    }
    else
    {
        ... // error condition.
    }
}
What is a function?

Set of instructions:
• Operations on POD.
• Control flow, including calls to other functions.
• Local variables.
• Input arguments.
• Return value.

Old days: a single chunk of memory was allocated for a function that contained everything — instructions and local data.

However, you can’t perform recursive calls to a function using such an approach — the local variables get overwritten.

Solution: organize things so that the storage for local data is managed separately from the storage for the instructions.
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.
Call Stack / Stack Frame

Enables inheritance:
• Jump to address of the function (fft1d*).
• Set up storage for local scope.
  • Values of the arguments (depends on the base calls definition)
  • Storage for local variables (depends on the derived function, not known to the calling function). Size is known at compile time.

Enables debugging tools
• At any given point in the program, the entire (POD) data state of the program at that time is stored in the call stack. That includes the values of pointer variables so you can look at your memory-managed data. Traverse the calling stack using “up”, “down”.
• The calling stack can be quite deep, particularly when using libraries.
What about member functions of an object?

```cpp
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;
FFT: 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.
- Next-generation version based on CMU’s Spiral framework.
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
 */
...

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
\ell & -1 & -\ell & 1 \\
-1 & 1 & -1 & 1 \\
-\ell & -1 & \ell & 1
\end{bmatrix}
\]

\(5 n \log_2(n) = 40, 8n^2 = 128. \) 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
*/
...

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

(5 n \log_2(n) = 120, 8n^2 = 512. 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 log₂(n) = 241 , 8n² = 1352. About 650 LOC.)
Looking at codelets

nl_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 log₂(n) = 1920, 8n² = 32768. About 2950 LOC.)
Standard Template Library containers.

Predefined classes: aggregates that are templated on the type being held.

Example of a namespace. The names of these classes are `std::className`.

We use the command

```cpp
using namespace std;
```

in global scope to tell compiler to look for functions of the form `std::className`. Some authorities view this as bad form.

http://www.cplusplus.com/

NB: C++11 standard.
Various choices in container templates

Container templates in the STL

- C arrays as first-class objects (array),
- dynamic arrays (vector),
- queues (queue),
- stacks (stack),
- heaps (priority_queue),
- linked lists (list),
- trees (set),
- associative arrays (map)

• They are distinguished by the kinds of access they provide and the complexity of their operations.

• To use these, you need to include the appropriate header file, e.g.
  \#include <array>
**std::vector<T>**

- unsigned int size();
- push_back(const T&);
- pop_back(const T&);
- T& back();
- T& front();
- operator[](int);

Vector<T>::iterator begin()
- Looks like a 1D array: can index any element by an integer less than size().
- Can add, delete elements at the end of an array.
- Fast access: data stored in contiguous locations in memory (just as if you had used new. In fact, you can access the underlying contiguous storage as an ordinary 1D array.
How do remove an element from a vector?

• Can do this at the end easily, but in general
  - find the element you wish to remove
  - make a whole new vector 1 smaller than the original
  - copy all but the excluded object to the new vector

• But we have already been doing something almost as awful with the push_back function of vector
  - grow vector length by one
  - copy all elements to the new vector with length+=1
  - copy the new element on the end
  - (in reality vector is doing a version of doubling it’s size when it runs of of room and keeps track of it’s “real” size and it’s size() )

• Vectors are good at:
  - Accessing individual elements by their position index (constant time).
  - Iterating over the elements in any order (linear time).
  - Add and remove elements from its end (constant amortized time).
list<T>

- std::list provides the following features
  - Efficient insertion and removal of elements anywhere in the container (constant time).
  - Efficient moving elements and block of elements within the container or even between different containers (constant time).
  - Iterating over the elements in forward or reverse order (linear time).

- What list is not good at is random access.
  - ie. if you wanted to access the 35th entry in a list, you need to walk down the linked list to the 35th entry and return it.
list<T>

unsigned int size();

push_back(const T&);

pop_back(const T&);

T& front();

T& back();

insert(list<T>::iterator , const T&);

erase(list<T>::iterator);

list<T>::iterator begin();

But no indexing operator! However, insertion / deletion is cheap once you find the location you want to insert or delete at.
Why list instead of vector?

- erase, insert, splice, merge are O(1) complexity
- remove, unique are O(linear) complexity.

```cpp
void removeBoundary(std::list<Node>& a_nodes, std::list<Node>& a_boundary)
{
    std::list<Node>::iterator it;
    for(it=a_nodes.begin(); it!=a_nodes.end(); ++it)
    {
        if(!it->isInterior())
        {
            a_boundary.splice(a_boundary.start(), a_nodes, it);
        }
    }
    Executes in linear time, and Node is never copied.
}
<map> : an associative container

- Stores elements formed by the combination of a key value and a mapped value.

- You index into a map with the key, you get back out the value.
  - You could consider vector a simple map where the key is an unsigned integer, and the value is the template class, but that imposes the constraint that they keys are the continuous interval [0,size-1]
  - but what if your keys don’t have this nice simple property?

- map take two template arguments, one for the key, and one for the value

- The key class needs to implement the operator<
  - Strict Weak Ordering
    - if a<b and b<c, then a<c
    - if a<b then !(b<a)
    - if !(a<b) and !(b<a) then a == b
Map<Key,T>

unsigned int size();

insert(pair<Key,T>);
insert(map<Key,T>::iterator,const T&);

erase(map<Key,T>::iterator);
erase(const Key K&);

T& front();
T& back();

map<Key,T>::iterator begin();
operator[](const Key K&);
A simple dictionary object

```cpp
#include <map>
#include <string>
#include <iostream>

using namespace std;

void fillDictionary(map<string, string>& a_dictionary, const string& filename);

int main(int argc, char* argv[]) {
    map<string, string> dictionary;
    string key;
    fillDictionary(dictionary, argv[1]);
    while(true) {
        cout << "query : ";
        cin >> key;
        if(key.size() == 0) return 0;
        map<string, string>::iterator val = dictionary.find(key);
        if(val == dictionary.end())
            cout << "\n did not find that word in the dictionary " << endl;
        else
            cout << "\n" << val->second << endl;
    }
}
```
parse a simple input file

void fillDictionary(map<string,string>& a_dictionary, const string& filename)
{
    ifstream f(filename.c_str()); string key, value; char buffer[2048];
    bool next=true; char token[] = ":";
    while(f.getline(buffer,2048, token[0]))
    {
        if(next)
        {
            key = string(buffer);  next = !next;
        }
        else
        {
            value = string(buffer);  a_dictionary[key]=value; next=!next;
        }
    }
}
std::unordered_map

- Same Interface as std::map, but this is a hash table

- Optimized for fast lookup
  - std::map is $O(\log N)$ insertion and lookup, std::unordered_map is $O(\log N)$ insertion and $O(1)$ lookup (average).

- Implementation generally uses more memory to speed up lookup (one or two levels of binning)

- Relies on the concept of hashing
  - Turn Key type into a size_t integer.
    ```cpp
    std::size_t h = std::hash<Bob>(myBob);
    ```
  - A good hash has few if any collisions
  - A good container hash even density

- Not a good choice if your access pattern is visiting every member
Hierarchical use of BoxData

map<Point,int> m_getContents;
int k = m_getContents[pt];

RectMDArray<bool> m_bitmap;

vector<list<Particle > > m_particles;
vector<BoxData<T,N> > m_refGrids;
Hierarchical use of BoxData

/// Boolean-valued BoxData. Defines which Boxes are members of the domain.

BoxData<bool> m_bitmap;

/// Vector of Points each of which is associated with a data in the region defined by a Point: a refined patch of grid, a collection of particles.
vector<T> m_stuff;

/// Maps Points to an index into m_stuff.
map<Point, int > m_getPatches;

Why do we use this extra level of indirection?
• Indexing [.] creates a new entry if one is not there.
• We could get rid of m_bitmap by using map::find. (Warning: average cost smaller than worst-case).

Why don’t we store the elements of m_stuff directly in the map? Answer: one copy of metadata for multiple data holders.