CS 294-73
Software Engineering for Scientific Computing

Lecture 13: Particle Methods;Homework 3.

Particle Methods

Numerical methods defined by the evolution of a finite collection of points in space. Each point carries both the position in space, along with other properties, which themselves may or may not evolve.

Example: a collection of *N* physical particles, evolving under Newton's laws of classical mechanics.

$$\{\boldsymbol{x}_k, \boldsymbol{v}_k, w_k\}_{k=1}^{N}$$

$$\frac{d\boldsymbol{x}_k}{dt} = \boldsymbol{v}_k$$

$$\frac{d\boldsymbol{v}_k}{dt} = \boldsymbol{F}(\boldsymbol{x}_k)$$

$$\boldsymbol{F}(x) = \sum_{k'} w_{k'} (\nabla\Phi)(\boldsymbol{x} - \boldsymbol{x}_{k'})$$

Particle Methods

Particle methods are also employed as discretizations of partial differential equations.

- Vorticity form of the incompressible Euler equations.

$$\frac{\partial \omega}{\partial t} + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} = 0$$

$$(u, v) = \vec{u} = \vec{u}(x, y, t)$$

$$u = \frac{\partial \psi}{\partial y} \ , \ v = -\frac{\partial \psi}{\partial x}$$

$$-\Delta\psi \equiv -\left(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}\right) = \omega$$

$$\omega = \omega(\boldsymbol{x}(t), t) \ , \ \frac{d\boldsymbol{x}}{dt} = \boldsymbol{u}(\boldsymbol{x}(t), t)$$

$$\frac{d}{dt}\big(\omega(\boldsymbol{x}(t), t)\big) = 0$$

$$\Longrightarrow$$

$$\omega(\boldsymbol{x}, t) \approx \sum_k \omega_k \zeta(|\boldsymbol{x} - \boldsymbol{x}_k(t)|)$$

$$\frac{d\boldsymbol{x}_k}{dt} = \sum_k \omega_k \vec{U}(\boldsymbol{x} - \boldsymbol{x}_k(t))$$

$$\vec{U} = \left(\frac{\partial \Psi}{\partial y}, -\frac{\partial \Psi}{\partial x}\right) \ , \ \Psi = \Psi(|\boldsymbol{x}|) = \Delta^{-1}\zeta$$

$$\vec{U}(x, y) = \frac{(y, -x)}{2\pi|\boldsymbol{x}|^2} \ , \ |\boldsymbol{x}| > \ \delta$$

Particle Methods

$$\boldsymbol{F}(x) = \sum_{k'} w_{k'} (\nabla \Phi)(\boldsymbol{x} - \boldsymbol{x}_{k'})$$

$$\frac{d\boldsymbol{x}_k}{dt} = \sum_{k} \omega_k \vec{U}(\boldsymbol{x} - \boldsymbol{x}_k(t))$$

To evaluate the fields for a single particle requires N evaluations of the field functions, leading to an $O(N^2)$ cost per time step: How do we reduce that cost ?

Short-Range Forces

- Short-range forces (e.g. Lennard-Jones potential).

$$\Phi(\boldsymbol{x}) = \frac{C_1}{|\boldsymbol{x}|^6} - \frac{C_2}{|\boldsymbol{x}|^{12}}$$

The forces fall off sufficiently rapidly that the approximation $\nabla\Phi(\boldsymbol{x}) \equiv 0 \text{ if } |\boldsymbol{x}| > \sigma$ introduces acceptably small errors for practical values of the cutoff distance σ.

Long-Range Forces

- Coulomb / Newtonian potentials

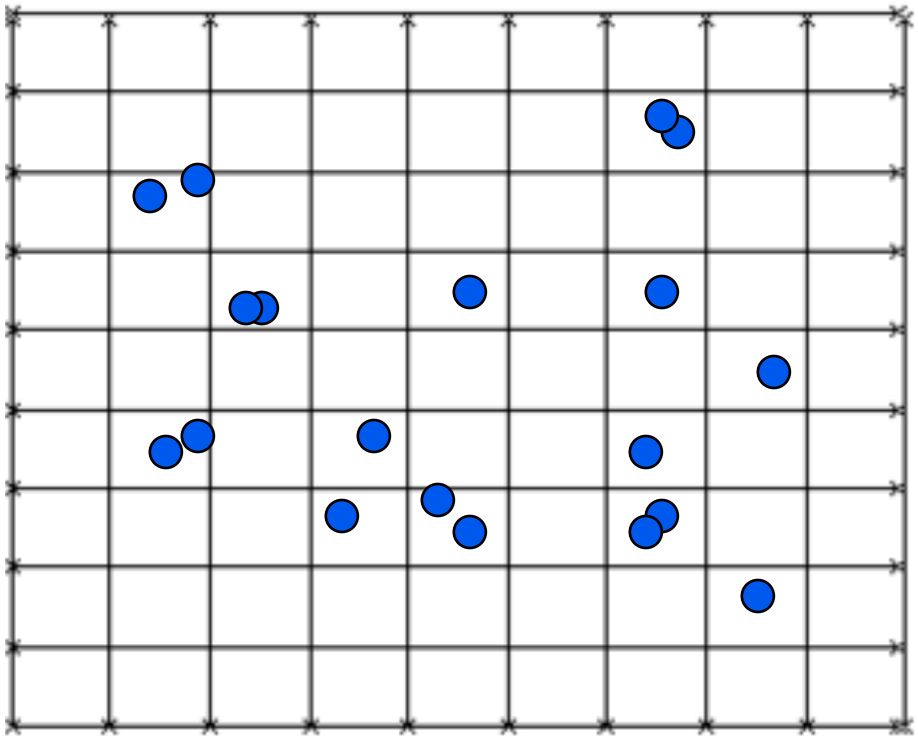
$$\Phi(r) \sim \frac{1}{2\pi} log(r) \ , \ D = 2$$
$$\sim \frac{1}{4\pi r} \ , \ D = 3 \qquad\qquad r > \delta$$

cannot be localized by cutoffs without an unacceptable loss of accuracy. But for this special case, we can take advantage of the fact, that the potential is a localized solution to Poisson's equation, for $r > \delta$,

$$-\Delta\Phi = 0$$

Bin Sorting

For both long-range and short-range forces, need to sort particles to determine which ones are near / far from a particle. The easiest way to do this is with bin sorting.

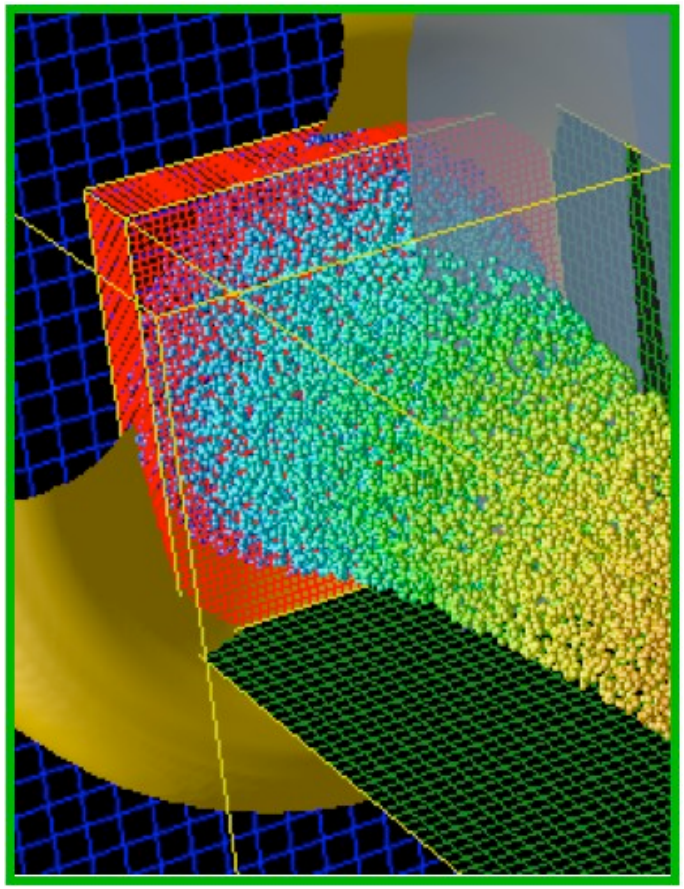


$$\boldsymbol{i}_k = \left\lfloor \frac{1}{h} \boldsymbol{x}_k \right\rfloor$$

- Cost of sorting into bins: O(1) per particle.
- Cost of computing which bins are close: O(1) per bin.
- Cost of computing which particles are separated by at least some fixed distance: O(N) + O(B) (why not $O(B^2)$).

Bin Sorting

Can use locally-refined grids / to maintain number of particle / bin fixed.



Coulomb Forces

For short-range forces, choose $h \sim \sigma$, and to determine which particles are close are close enough to each other to require evaluation of the force. However, we still have an $O(N (h/L)^D)^2 \times (L/h)^D = O(N^2) (h/D)^D$ calculation for a uniformly-distributed collection of particles, i.e. we've reduced the number of pair-potential calculations by the number of bins.

For Coulomb forces, need to take advantage of the smoothness of the far field, and the relationship of the Coulomb potential to Poisson's equation to reduce the work required to compute the potential of distantly-separated particles.

- Multipole methods / tree methods.
- Particle-in-cell (PIC) methods / Particle-Particle Particle-Mesh(PPPM, P^3M) methods.

Particle-In-Cell Methods

Uses bin-sorting grid to compute the solution to Poisson's equation to represent the field induced by the particles. This is the most common approach for problems in which the particles are used to discretize a solution to a partial differential equation.

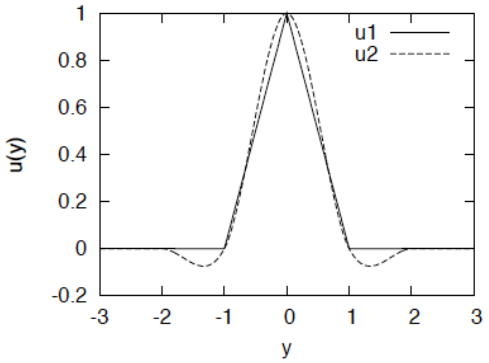
- Deposit charges onto the grid.
- Compute potential on the grid as a discrete convolution: $\phi = G^h * q$ Can do this fast using Hockney's method.
- Compute fields using finite differences on the grid.
- Interpolate fields to the particles.

Examples charge-deposition functions in 1D:

$$q_i = \sum_k q_k u(ih - x_k)$$

Both of these functions conserve total charge:

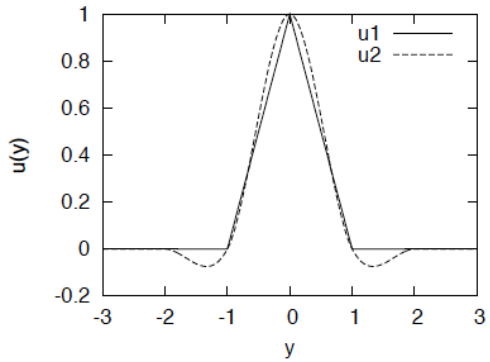
$$\sum_i q_i = \sum_k q_k$$



Particle-In-Cell Methods

Difficulties with particle-in-cell methods.

1. Not the right answer for “real” particles (e.g. molecular dynamics).
2. Accumulation of numerical error for long-time integrations.
3. Solutions to 2:
   I. Introduce a smoothing of the particles that scales more slowly than the mesh spacing.
   II. Remap the particles onto a fixed grid every few time steps. Accuracy vs. positivity.



Method of Local Corrections (a P^{3}M method)

Idea: compute far field effects with particle-in-cell, effect of nearby particles with N-body calculation. For this approach, it is conceptually simpler to work directly with the fields / forces.

- Deposit fields on the grid.

$$\vec{D}_{\boldsymbol{i}} \equiv \sum_k w_k (\Delta^h \vec{F}^{(k)})_{\boldsymbol{i}}^{trun,k}$$

$$\vec{F}_{\boldsymbol{i}}^{(k)} = (\nabla\Phi)(\boldsymbol{i}h - \boldsymbol{x}_k)$$

$$W_{\boldsymbol{i}}^{trun,k} = W_{\boldsymbol{i}} \text{ if } ||\boldsymbol{i} - \boldsymbol{i}_k|| \le C \ , \ = o \text{ otherwise}$$

- Compute discrete convolution to get forces:

$$\vec{\mathcal{F}}^h = G^h * \vec{D}^h$$

C = ∞, original force calculation evaluated at grid points. Why can we take C to be finite with only a small error ?

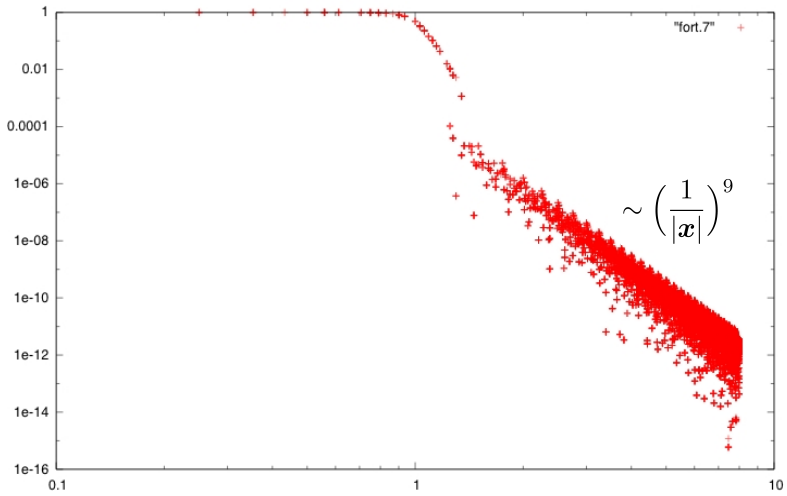
- Compute fields at particle locations:

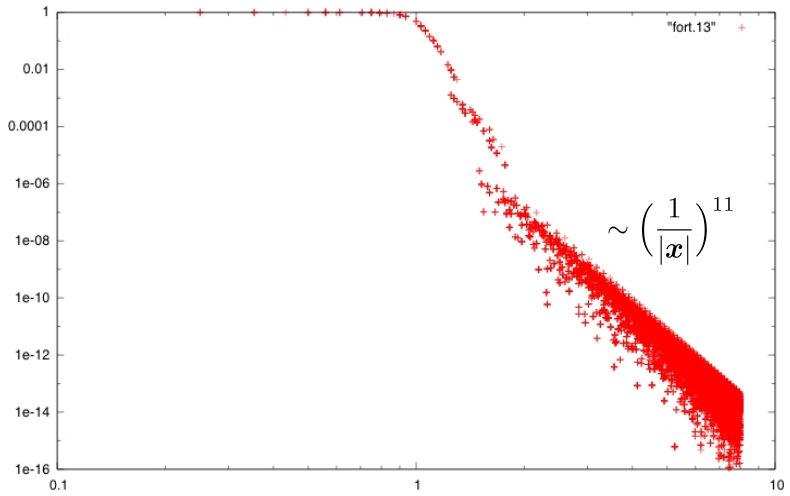
$$\vec{F}_k = \vec{F}^{local}(\boldsymbol{x}_k) + \mathcal{I}(\{\vec{\mathcal{F}}_{\boldsymbol{i}}^h - \vec{F}^{local}(\boldsymbol{i}h)\})(\boldsymbol{x}_k)$$

$$\vec{F}^{local}(\boldsymbol{x}) = \sum_{k' : ||\boldsymbol{i}_k - \boldsymbol{i}_{k'}|| \le C} w_{k'} \nabla\Phi(\boldsymbol{x} - \boldsymbol{x}_{k'})$$

Finite-Difference Localization

Decay of truncation error:





Plot of $\Delta^H(G * f)$, $\frac{R}{H} = 4$ for 27-point operator (q = 6) left; 53-point operator (q=8), right.

Homework 3: Matrix Multiply, fast and slow.

- You will implement dense general matrix matrix multiply for two column-wise stored dense matrices A and B. This data looks like the data layout we described for the float* indexing in Homework 1, except we will now be using double precision (double not float). The result is stored into another Matrix C. These will be square matrices, and a few dozen matrix sizes will be executed and then checked for correctness. A and B are initialized with random data.

1. implement your own triply-nested loop version of dense matrix multiply and put it in the file `dgemm-naive.cpp`. it will contain one function declared as

   ```
   void square_dgemm( int n, double *A, double *B, double *C )
   ```

   compile the 'naive' makefile target and execute and capture the output in a file named '`naive.out`' which you check into the repo

2. change the compiler flags from the default '`-g -Wall` flags to '-O3'. i.e., turn on compiler optimization. build 'naive' again, and produce a 'naive_opt.out' output.

3. implement a version of this function in a file named `dgemm-blas.cpp` with a call to `cblas` third party library. Build the `'blas'` makefile target. run the code and create a `'blas.out'` output for your problem to submit.

Homework 3: FFT, fast and slow

- You will write classes `FFT1DRecursive, FFT1DW,` derived from FFT1D. You will build them with the FFT1D class, and time them using the Unix `time` shell command. We will provide and implementation of the bit-reversed algorithm (BRI). Time them with both the debug and optimized versions.
- Once you have your three implementations running, you will test them inside the 3D driver, timing them for the sizes indicated.

FFTW (From Section 2.1 of http://www.fftw.org/fftw3_doc/)

```
#include “fftw3.h”
...
{
        fftw_complex *in, *out;
        fftw_plan p; ... in = (fftw_complex*)
        fftw_malloc(sizeof(fftw_complex) * N);
        out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
        p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE); ...
        fftw_execute(p);
        /* repeat as needed */ ...
        fftw_destroy_plan(p);
        fftw_free(in);
        fftw_free(out);
}
```

- Need to go to the fftw site and install fftw on your own machine.
- You won’t use this approach from the FFTW document. fftw needs to have its own view of the data – you will alias your own complex data to fftw’s data, as indicated in the last lecture.

Homework #3 as seen through the makefiles

- `hw3/GNUmakefile` – contains common makefile definitions.
- `hw3/dgemmTest/GNUmakefile` – build rules for matrix-multiply tests.
- `hw3/fftTest/GNUmakefile` – build rules for FFT tests.
- `hw3/fftTools/GNUmakefile` – build rules for FFT libraries.

Building and Using Libraries

Makefile for building FFT1D test.

In `$(HOME)/GNUmakefile:`

```
LIBS_LOCAL = $(HOME)/lib
LIB_FLAGS:= -L$(LIBS_LOCAL) -L$(FFTW_HOME)/lib -lfftw3 -lfft1D

test1d: FFT1DTest.cpp GNUmakefile $(LIBS_LOCAL)/libfft1D.a
    $(CXX) $(CFLAGS) FFT1DTest.cpp $(LIB_FLAGS) -o test1d.exe
```

What is `libfft1D.a` ?

- Collection of binaries (`.o` files), assembled into an archive (`.a`)
- Unix utilities for building archives: `ar` (Linux), `libTool` (Mac).
- You've already been accessing such archives (part of the compilation system).
  - `/usr/lib , /usr/local/lib`
- Compiler / linker told where to look for them through `-L , -l` flags.
  - `-L<dir_name>` : search this directory for `.a` files.
  - `-l<root>` : look for library name of the form `lib<root>.a`

Building and Using Libraries

In `fftTools/GNUmakefile:`

```
1DFFTOBJS = FFT1DBRI.o PowerItoI.o FFTCTBRI.o FFT1DRecursive.o FFTW1D.o
libfft1D.a: GNUmakefile $(1DFFTOBJS)
    $(LIBTOOL) libfft1D.a $(1DFFTOBJS)
    mkdir -p ../lib;mv libfft1D.a $(LIBS_LOCAL)
```

In `hw3/GNUmakefile:`

```
$(LIBS_LOCAL)/libfft1D.a:$(wildcard $(FFT_HOME)/*.{H,cpp} ) GNUmakefile
    cd $(FFT_HOME);make clean;make libfft1D.a DIM=1 CXX=$(CXX)
```

Test Programs for Assignment 3

```
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(unsigned int a_M){m_M = a_M; m_N = Power(2,m_M);}
  virtual ~FFT1D() { }
  // 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 unsigned int& getN(){return m_N;};
  const unsigned int& getM(){return m_M;}
protected:
  unsigned int m_M, m_N;
};
```

Test Programs for Assignment 3

```
int main(int argc, char* argv[])
{
  int M;
  int inputMode;
  string fft_string;

  cout << "input log_2(N), N = number of points" << endl;
  cin >> M ;
  cout << "input test mode < N" << endl;
  cin >> inputMode;
```

Test Programs for Assignment 3

```
  cout << "input FFT being tested: Recursive, BRI, FFTW" << endl;
cin >> fft_string;
shared_ptr<FFT1D> p_fft;

if (fft_string == "Recursive")
  {
    // Uncomment and delete the abort when ready to test.
    // shared_ptr<FFT1DRecursive> p_fft1dR
    // =shared_ptr<FFT1DRecursive>(new FFT1DRecursive(M));
    // p_fft =  dynamic_pointer_cast<FFT1D >(p_fft1dR);
    cout << "this one is for the students to do" << endl;
    abort();
  }
else if (fft_string == "BRI")
  {
    shared_ptr<FFT1DBRI> p_fft1dBRI =
      shared_ptr<FFT1DBRI>(new FFT1DBRI(M));
    p_fft =  dynamic_pointer_cast<FFT1D >(p_fft1dBRI);
  }
```

Test Programs for Assignment 3

```
  else if (fft_string == "BRI")
     {
       shared_ptr<FFT1DBRI> p_fft1dBRI =
      shared_ptr<FFT1DBRI>(new FFT1DBRI(M));
       p_fft =  dynamic_pointer_cast<FFT1D >(p_fft1dBRI);
     }
else if (fft_string == "FFTW")
     {
       // Uncomment and delete the abort when ready to test.
       // p_fft =
      // dynamic_pointer_cast<FFT1D >(shared_ptr<FFTW1D>(new FFTW1D(M)));

       cout << "this one is for the students to do" << endl;
       abort();
     }
   else
     {cout << "invalid input - should use BRI, Recursive or FFTW as name fo
  FFT implementation to be tested" << endl;
      abort();}
```

Test Programs for Assignment 3

```
double error = test1(p_fft);
int mode = test2(p_fft,inputMode);
cout << fft_string<< ": test 1: error in Gaussian  = " << error << endl;
cout << fft_string << ": test 2: reproducing input Fourier mode " <<
inputMode <<
  " , output mode " << mode <<endl;
cout << "The input mode number and the output mode number should match"
<< endl;
cout << "if multiple modes have non-roundoff amplitude, this is an error
and the output mode is set to -1" << endl;
```

Test Programs for Assignment 3

```
int main(int argc, char* argv[])
{
  int M;
  double time;

  sscanf(argv[1],"%d",&M);
  dynamic_pointer_cast<FFT1D >(shared_ptr<FFTW1D>(new FFTW1D(M)));
  FFTMD fftmd(p_fft);
  double error = test(fftmd,time);
  cout << "test 1: error in Gaussian  = " << error << endl;
  cout << "time in FFTMD = " << time << " seconds" << endl;
};
```