CS 294-73
Software Engineering for Scientific Computing

Lecture 14: PPPM for Molecular Dynamics; Final Project

Molecular Dynamics

We want to simulate a collection of *N* physical particles, evolving under Newton's laws of classical mechanics.

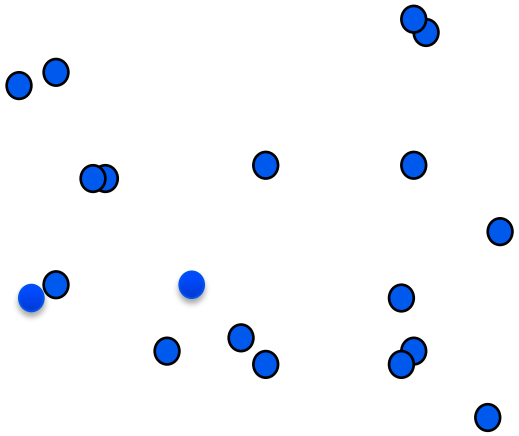
$$\{\boldsymbol{x}_k, \boldsymbol{v}_k, w_k\}_{k=1}^{N}$$

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

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

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

N-Body Calculation for Coulombic Systems



Collection of charged particles, each of which induces a contribution to the forces function in any point in space via Coulomb potential.

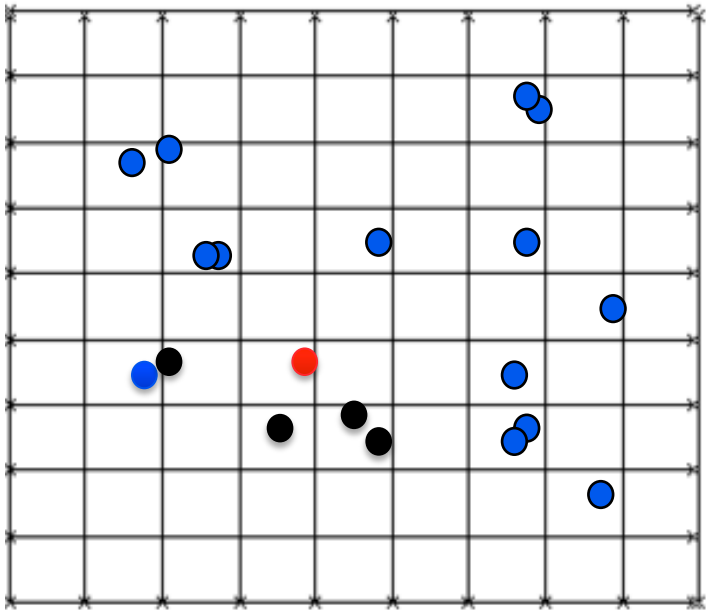
$$\vec{F}_k(\boldsymbol{x}) = q_k \nabla_{\boldsymbol{x}} G(\boldsymbol{x} - \boldsymbol{x}_k) \ , \ G(\boldsymbol{z}) = \frac{1}{4\pi|\boldsymbol{z}|} \ , \ \vec{F}(\boldsymbol{x}) = \sum_{k'} \vec{F}_{k'}(\boldsymbol{x})$$

Computing the forces on N particles is an O(N^2) calculation

$$\vec{F}(\boldsymbol{x}_k) = \sum_{k'} \vec{F}_{k'}(\boldsymbol{x}) \ , \ k = 1, \ldots, N$$

Can't just use nearby particles, due to long-range nature of the forces Can't use PIC, due to short-range nature of the forces..

Splitting into short-range and long-range



To compute the force on a particle (red), represent as a combination of

- Contributions from nearby particles (black), using (local) N-Body calculations.
- Contributions from far-away particles, using PIC (blue).

How do you do this without double-counting, or having many PIC solves ?
What is the error of the resulting method for an arbitrary distribution of a finite number of particles ?

P^{3}M (Hockney and Eastwood)

Idea: express the potential as a sum.

$$G(\boldsymbol{z}) = G_{PP}(\boldsymbol{z}) + G_{PM}(\boldsymbol{z})$$

$$\begin{aligned} G_{PP}(\boldsymbol{z}) &= 0 \qquad \text{for } |\boldsymbol{z}| > \delta \\ &= G(\boldsymbol{z}) \text{ for } |\boldsymbol{z}| < \delta' < \delta \ , \ \delta = O(\delta') \end{aligned}$$

- $\sum_{k'} (\nabla G_{PP})(\boldsymbol{x} - \boldsymbol{x}'_k)$ Computed using N-body calculation.
- $\sum_{k'} (\nabla G_{PM})(\boldsymbol{x} - \boldsymbol{x}'_k)$ Approximated using PIC.

Cost of PP calculation: $O(N\delta^3)N = O(N^2\delta^3)$

Cost of PM calculation: $O(N) + O(N_g log N_g)$

P^3M (Hockney and Eastwood)

Issues:

- How to implement ?
  - Approach: basic particle representation from PIC doesn't change - only force calculation does.
- What is the error ?
  - This is only a question about the error in the PIC calculation - the decomposition and the PP calculation are exact.
  - Error analysis is tricky, since the number of particles is fixed.

PM Force Calculation

$$\rho_{\boldsymbol{i}}^{g} = \sum_{k} q_k \Psi(\boldsymbol{i}h_g - \boldsymbol{x}_k)$$

$$\phi_{\boldsymbol{i}}^{g} = \sum_{\boldsymbol{j}\in\mathbb{Z}^3} G_{PM}(\boldsymbol{i}h_g - \boldsymbol{j}h_g)\rho_{\boldsymbol{i}}^{g}$$

$$\vec{F}_{\boldsymbol{i}}^{g} = \nabla_g(\phi^g)_{\boldsymbol{i}}$$

$$\vec{F}_{PM,k} = \sum_{\boldsymbol{i}} \vec{F}_{\boldsymbol{i}}^{g} \Psi(\boldsymbol{x}_k - \boldsymbol{i}h_g)$$

Focus on the first two steps. We are making the approximation

$$\begin{aligned}
\sum_{k} q_k G_{PM}(\boldsymbol{x} - \boldsymbol{x}_k) &\approx \sum_{\boldsymbol{j}\in\mathbb{Z}^3} G_{PM}(\boldsymbol{x} - \boldsymbol{j}h_g)\rho_{\boldsymbol{i}}^{g} \\
&= \sum_{\boldsymbol{j}\in\mathbb{Z}^3} G_{PM}(\boldsymbol{x} - \boldsymbol{j}h_g) q_k \Psi(\boldsymbol{j}h_g - \boldsymbol{x}_k) \\
&= \sum_{k} q_k \sum_{\boldsymbol{j}\in\mathbb{Z}^3} G_{PM}(\boldsymbol{x} - \boldsymbol{j}h_g)\Psi(\boldsymbol{j}h_g - \boldsymbol{x}_k)
\end{aligned}$$

PM Force Calculation

In what sense does

$$G_{PM}(\boldsymbol{x}-\boldsymbol{x}_k) \approx \sum_{\boldsymbol{j}\in\mathbb{Z}^3} G_{PM}(\boldsymbol{x}-\boldsymbol{j}h_g)\Psi(\boldsymbol{j}h_g-\boldsymbol{x}_k)?$$

Taylor expansion of $G_{PM}(\mathbf{y}) = G_{PM}(\mathbf{x}-\mathbf{y})$ about $\mathbf{y} = \mathbf{x}_k$ (multi-index notation).

$$\sum_{\boldsymbol{j}\in\mathbb{Z}^3} G_{PM}(\boldsymbol{x}-\boldsymbol{j}h_g)\Psi(\boldsymbol{j}h_g-\boldsymbol{x}_k) = G_{PM}(\boldsymbol{x}-\boldsymbol{x}_k)\sum_{\boldsymbol{j}\in\mathbb{Z}^3}\Psi(\boldsymbol{j}h_g-\boldsymbol{x}_k)$$

=1 (conservation of charge)

$$+\sum_{\boldsymbol{p}}\frac{(-1)^{|\boldsymbol{p}|_1}}{\boldsymbol{p}!}(\nabla^{\boldsymbol{p}}G_{PM})|_{(\boldsymbol{x}-\boldsymbol{x}_k)}\sum_{\boldsymbol{j}\in\mathbb{Z}^3}(\boldsymbol{j}h_g-\boldsymbol{x}_k)^{\boldsymbol{p}}\Psi(\boldsymbol{j}h_g-\boldsymbol{x}_k)$$

=0 (moment conditions)

$$+O\Big(\frac{h_g^P}{max(|\boldsymbol{x}-\boldsymbol{x}_k|,\delta)^{P+1}}\Big)$$

Remainder term in the Taylor expansion.

PM Force Calculation

Thus the error in the field induced by a single particle is

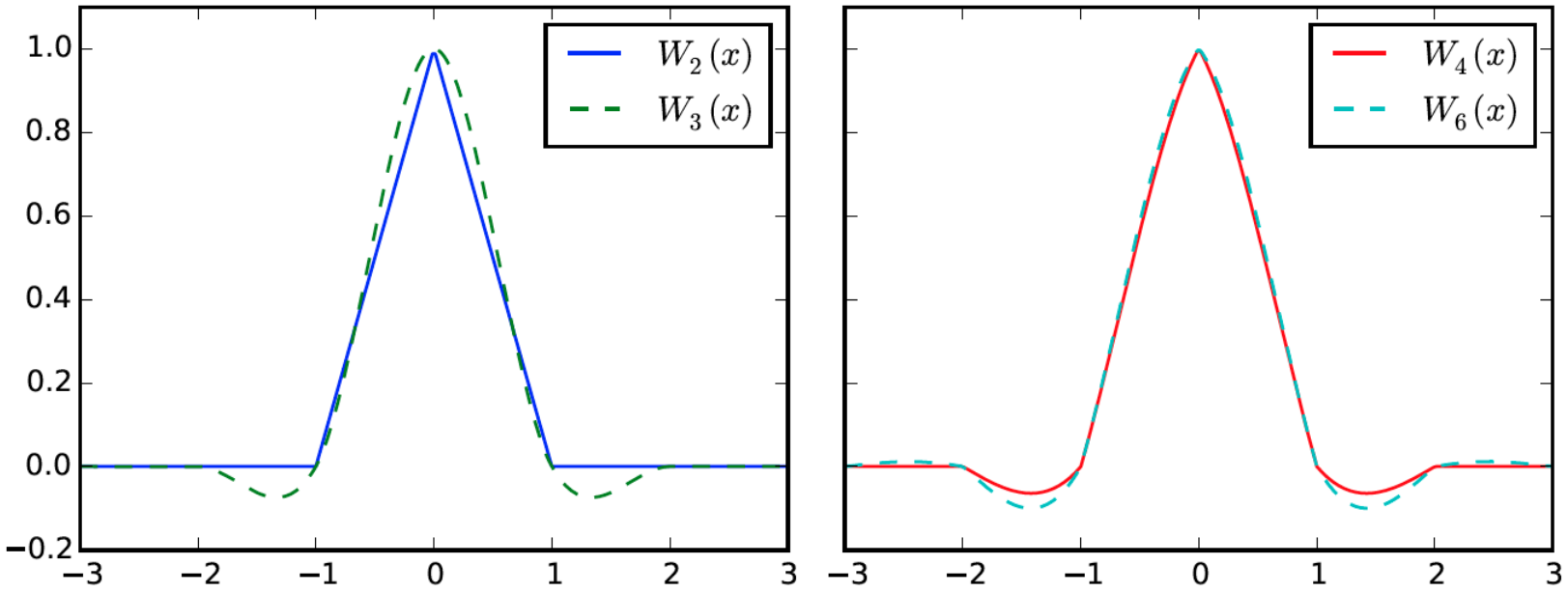
$$O\Big(\frac{h_g^P}{max(|\boldsymbol{x}-\boldsymbol{x}_k|,\delta)^{P+1}}\Big)$$

What happens for many particles - some nearby, some farther away ?

$$\begin{aligned}\epsilon &= \sum_k q_k O\Big(\frac{h_g^P}{max(|\boldsymbol{x}-\boldsymbol{x}_k|,\delta)^{P+1}}\Big)\\ &= \sum_r O\Big(\frac{h^P}{(r\delta)^{P+1}}\bar{q}\delta(r\delta)^2\Big)\\ &= O\Big(\frac{h_g^P}{\delta^{P-2}}\sum_{r=1}^{R}\frac{1}{r^{P-1}}\Big) = O\Big(h_g^2\Big(\frac{h_g}{\delta}\Big)^{P-2}\Big)\end{aligned}$$

(Missing a factor of log(R) if P=2).

Different deposition functions



Final Project

- Done in teams of 2-4 people.
- Final report due at 11:59PM, 12/15/17.
- We have set of intermediate deadlines that
  - Define a process
  - Define the work product of your project.
- The intermediate work will not be graded at the various deadlines, but read to provide you feedback and guidance.They are still required by the end of the project.
- You should be using git to track / submit your intermediate work, and to communicate within the team. Don't just submit everything at the end.
- In this lecture, I will go through in detail the process and deadlines in terms of a project I have used in the past.

Intermediate milestones

- Identification of the teams, and of the topic for the project. A mathematical description of the problem being solved. This can be in the form of a research article, combined with a concise summary. (Required by 10/30/17. We will set up shared git repositories for each project / team.)
- A complete mathematical specification of the discretization methods and / or numerical algorithms to be used to solve the problem. A specification of what will constitute a computational solution to this problem. (11/13/17)
- A top-level design of the software used to solve this problem, in the form of header files for the major classes, the mapping of those classes to the algorithm specification given above, and the specification of a testing process for each class. (11/27/17).

Mathematical Specification

Incompressible Euler Equations

- Bell, Colella, Glaz, J. Comput. Phys. 1989.

$$\frac{\partial u_d}{\partial t} + \sum_{d'=0}^{D-1} \frac{\partial (u_d u_{d'})}{\partial x_{d'}} = -\frac{\partial p}{\partial x_d}$$

$$\sum_{d'=0}^{D-1} \frac{\partial u_d}{\partial x_d} = 0$$

$$(u_0(x_0, x_1, t), u_1(x_0, x_1, t)) = \vec{u} : [0,1]^D \times [0,T] \rightarrow \mathbb{R}^D \ , \ D = 2$$

- Initial conditions on velocity specified, periodic boundary conditions:

$$\vec{u}(\boldsymbol{x}, 0) = \vec{u}_0(\boldsymbol{x})$$

$$\vec{u}(\boldsymbol{x} + (k,l)) = \vec{u}(\boldsymbol{x}) \ , \ k, l \in \mathbb{Z}$$

- Mathematically unfamiliar: combination of evolution equations and constraints.

Pressure-Poisson formulation

- Eliminate the constraint by differentiating it with respect to time, obtaining an equation for *p.*

$$\sum_{d=0}^{D-1} \frac{\partial}{\partial x_d}\left(\frac{\partial u_d}{\partial t} + \sum_{d'=0}^{D-1} \frac{\partial (u_d u_{d'})}{\partial x_{d'}} + \frac{\partial p}{\partial x_d}\right) = 0$$

$$\sum_{d=0}^{D-1} \frac{\partial}{\partial x_d}\left(\sum_{d'=0}^{D-1} \frac{\partial (u_d u_{d'})}{\partial x_{d'}}\right) = -\sum_{d=0}^{D-1} \frac{\partial^2 p}{\partial x_d^2}$$

$$\nabla p = \left(\frac{\partial p}{\partial x_0}, \frac{\partial p}{\partial x_1}\right) \text{ periodic } \Rightarrow p \text{ periodic}$$

Projection formulation

- Use the Helmholtz projection operator to make this manifestly an evolution equation without constraints.

$$\mathbb{P} \equiv \mathbb{I} - grad(\Delta^{-1})div$$

$$div(\vec{w}) = \sum_{d=0}^{D-1} \frac{\partial w_d}{\partial x_d} \ , \ grad(\phi) = \left(\frac{\partial p}{\partial x_0}, \ldots, \frac{\partial p}{\partial x_{D-1}} \right) \ , \ \Delta\phi = div(grad(\phi))$$

$$\mathbb{P}(grad(\phi)) = 0 \ , \ \mathbb{P}(\vec{u}) = \vec{u} \text{ if } div(\vec{u}) = 0$$

Starting from

$$\frac{\partial \vec{u}}{\partial t} + \sum_{d'=0}^{D-1} \frac{\partial (u_{d'} \vec{u})}{\partial x_{d'}} + grad(p) = 0$$

$$div(\vec{u}) = 0$$

We apply the projection operator

Projection formulation

- Use the Helmholtz projection operator to make this manifestly an evolution equation without constraints.

$$\mathbb{P} \equiv \mathbb{I} - grad(\Delta^{-1})div$$

$$div(\vec{w}) = \sum_{d=0}^{D-1} \frac{\partial w_d}{\partial x_d} \ , \ grad(\phi) = \left(\frac{\partial p}{\partial x_0}, \ldots, \frac{\partial p}{\partial x_{D-1}} \right) \ , \ \Delta\phi = div(grad(\phi))$$

$$\mathbb{P}(grad(\phi)) = 0 \ , \ \mathbb{P}(\vec{u}) = \vec{u} \text{ if } div(\vec{u}) = 0$$

$$\mathbb{P}\left(\frac{\partial \vec{u}}{\partial t} + \sum_{d'=0}^{D-1} \frac{\partial (u_{d'} \vec{u})}{\partial x_{d'}} + grad(p) \right) = 0$$

$$div(\vec{u}) = 0$$

And using the fact that the projection annihilates gradients, leaves divergence-free fields unchanged

Projection formulation

- Use the Helmholtz projection operator to make this manifestly an evolution equation without constraints.

$$\mathbb{P} \equiv \mathbb{I} - grad(\Delta^{-1})div$$

$$div(\vec{w}) = \sum_{d=0}^{D-1} \frac{\partial w_d}{\partial x_d} \ , \ grad(\phi) = \left(\frac{\partial p}{\partial x_0}, \ldots, \frac{\partial p}{\partial x_{D-1}} \right) \ , \ \Delta\phi = div(grad(\phi))$$

$$\mathbb{P}(grad(\phi)) = 0 \ , \ \mathbb{P}(\vec{u}) = \vec{u} \text{ if } div(\vec{u}) = 0$$

$$\frac{\partial \vec{u}}{\partial t} + \mathbb{P}\left(\sum_{d'=0}^{D-1} \frac{\partial (u_{d'} \vec{u})}{\partial x_{d'}} \right) = 0$$

$$div(\vec{u})\big|_{t=0} = 0$$

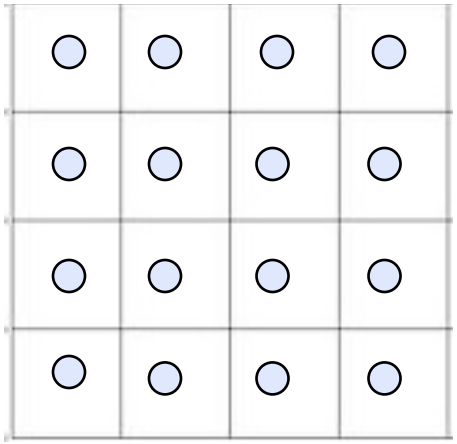
Discretization

Discretization in space

- Cell-centered discretization on a power-of-two grid.

$$\vec{u}_{\boldsymbol{i}}^{h}(t) \approx \vec{u}((\boldsymbol{i}+\frac{1}{2}\boldsymbol{u})h,t)\ ,\ \phi_{\boldsymbol{i}} \approx \phi(\boldsymbol{i}+\frac{1}{2}\boldsymbol{u})$$

$$(i_0,\ldots,i_{D-1}) = \boldsymbol{i} \in \{0,\ldots,2^M-1\}^D\ ,\ \boldsymbol{u} = (1,1,\ldots,1)$$



- Discretization of $\dfrac{\partial}{\partial x_d}(u_d\vec{u})$:

$$\vec{u}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}} \equiv \frac{1}{2}(\vec{u}_{\boldsymbol{i}} + \vec{u}_{\boldsymbol{i}+\boldsymbol{e}^d})$$

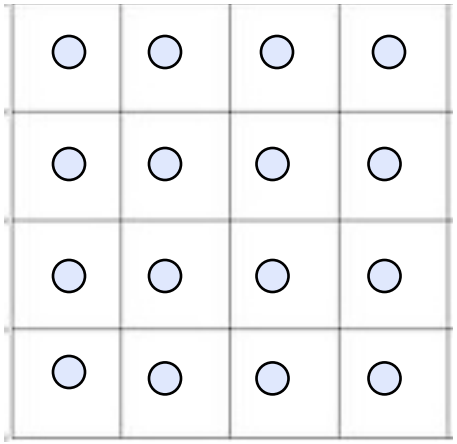
$$\left.\frac{\partial}{\partial x_d}(u_d\vec{u})\right|_{(\boldsymbol{i}+\frac{1}{2}\boldsymbol{u})h} \approx D^d(u_d\vec{u})_{\boldsymbol{i}} \equiv \frac{(u_d)_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d}\vec{u}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} - (u_d)_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^d}\vec{u}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^d}}{h}$$

Discretization in space

- Cell-centered discretization on a power-of-two grid.

$$\vec{u}_{\boldsymbol{i}}^{h}(t) \approx \vec{u}((\boldsymbol{i}+\frac{1}{2}\boldsymbol{u})h, t)\ ,\ \phi_{\boldsymbol{i}} \approx \phi(\boldsymbol{i}+\frac{1}{2}\boldsymbol{u})$$

$$(i_0, \ldots, i_{D-1}) = \boldsymbol{i} \in \{0, \ldots, 2^M - 1\}^D\ ,\ \boldsymbol{u} = (1, 1, \ldots, 1)$$



- Discretization of the projection operator

$$\mathbb{P}^h = \mathbb{I} - grad^h(\Delta^h)^{-1}div^h$$

$$grad^h(\phi^h)_{\boldsymbol{i}} = \left(\frac{1}{2h}(\phi_{\boldsymbol{i}+\boldsymbol{e}^0} - \phi_{\boldsymbol{i}-\boldsymbol{e}^0}), (\phi_{\boldsymbol{i}+\boldsymbol{e}^1} - \phi_{\boldsymbol{i}-\boldsymbol{e}^1})\right)$$

$$div^h(w^h)_{\boldsymbol{i}} = \frac{1}{2h}((u_0)_{\boldsymbol{i}+\boldsymbol{e}^0} - (u_0)_{\boldsymbol{i}-\boldsymbol{e}^0}) + \frac{1}{2h}((u_1)_{\boldsymbol{i}+\boldsymbol{e}^1} - (u_1)_{\boldsymbol{i}-\boldsymbol{e}^1}))$$

$$\Delta^h(\phi^h)_{\boldsymbol{i}} = \frac{1}{4h^2}\Big(-4\phi_{\boldsymbol{i}} + \phi_{\boldsymbol{i}+\boldsymbol{e}^0} + \phi_{\boldsymbol{i}-\boldsymbol{e}^0} + \phi_{\boldsymbol{i}+\boldsymbol{e}^1} + \phi_{\boldsymbol{i}-\boldsymbol{e}^1}\Big)$$

$$\text{note: } \Delta^h \neq div^h grad^h$$

Solvers

We require two solver algorithms: one for solving Poisson's equation on a periodic grid, plus a time-discretization algorithm.

- We will use an FFT solver for Poisson's equation.
- We will use RK4 for time discretization, with a slight variation specific to the kind of projection discretization we are using.

Method of Lines

This leads to a system of ordinary differential equations for the discretized variables.

$$\frac{d\vec{u}_{\boldsymbol{i}}}{dt} = -\mathbb{P}^h \left(\sum_d D^d (u_d^h \vec{u}^h)_{\boldsymbol{i}} \right)$$

To solve a system of ordinary differential equations of the form

$$\frac{dQ}{dt} = F(Q, t)$$

Fourth-order Runge-Kutta

$$
\begin{aligned}
k_1 &= F(Q^n, t^n) \\
Q^{n,(1)} = Q^n + \frac{\Delta t}{2} k_1 \ , \quad k_2 &= F(Q^{n,(1)}, t^{n+\frac{1}{2}}) \\
Q^{n,(2)} = Q^n + \frac{\Delta t}{2} k_2 \ , \quad k_3 &= F(Q^{n,(2)}, t^{n+\frac{1}{2}}) \\
Q^{n,(3)} = Q^n + \Delta t k_3 \ , \quad k_4 &= F(Q^{n,(3)}, t^{n+1}) \\
Q^{n+1} = Q^n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)
\end{aligned}
$$

$$
\frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) = \frac{1}{\Delta t} \int_{t^n}^{t^n + \Delta t} f(Q(t), t) dt + O(\Delta t)^4
$$

Generalizes Simpson's rule for integrals.

Method of Lines

For the case of

$$\frac{d\vec{u}_{\boldsymbol{i}}}{dt} = -\mathbb{P}^h\left(\sum_d D^d(u_d^h \vec{u}^h)_{\boldsymbol{i}}\right)$$

we compute

$$\vec{u}^* = \vec{u}^n + \frac{1}{6}(k_1 + k_2 + k_3 + k_4)$$
$$\vec{u}^{n+1} = \mathbb{P}^h(\vec{u}^*)$$

FFT Solver for Poisson's equation

Given $f^h : \{0, \ldots, 2^M - 1\}^D \to \mathbb{R}$, we compute $(\Delta^h)^{-1} f^h$ as follows.

- Compute the normalized complex forward FFT in 2D of

$$\mathcal{F}_N(f)_{k_0,k_1} \ , \ (k_0, k_1) \in \{-N/2 + 1, \ldots, N/2\}^D$$

- Compute the Fourier coefficients of $(\Delta^h)^{-1} f^h$:

$$\hat{g}_{k_0,k_1} = \frac{1}{2cos(2\pi k_0 h) + 2cos(2\pi k_1 h) - 4} \mathcal{F}_N(f)_{k_0,k_1} \ , \ (k_0, k_1) \neq (0, 0)$$

$$(f_{0,0} = 0)$$

- Compute the inverse FFT to obtain $(\Delta^h)^{-1} f^h$:

$$(\Delta^h)^{-1}_{i_0,i_1} = \mathcal{F}_N^{-1}(\hat{g})_{i_0,i_1} \ , \ (i_0, i_1) \in \{0, \ldots, N - 1\}^D$$

Software Design

Use the mathematical structure of your algorithm as a basis for your software design

- Hierarchical structure that represents the hierarchy of abstractions in the description of your algorithm.
- Classes for data containers, and low-level operations on them.
- Classes or functions for operators (depending on whether the operator has state, or needs to be used as a template parameter).

Operator Class: RK4

```
template <class X, class F, class dX>
class RK4
{
public:
  void advance(double a_time, double a_dt, X& a_state);
protected:
  dX m_k;
  dX m_delta;
  F m_f;
};
```

In advance, the operator m_f(...) is called to compute increments of the solution in RK4:

```
m_f(dX& a_dx, double& a_time, double& a_dt, X& a_x,
    dX& a_oldDx);
```

Comments on RK4

- Interface class done purely with templates (as opposed to inheritance).
- `X` comes in as an argument of `advance`, so it keeps track of any time-dependent context required to execute the various other operators.
- `dX <-> k` very lightweight: data holder for information required to increment the solution.
- `F` is a class, but is really just a function pointer.

Operator Class: ComputeEulerRHS

`ComputeEulerRHS` implements

$$-\mathbb{P}^h\left(\sum_d D^d(u_d^h \vec{u}^h)_{\boldsymbol{i}}\right)$$

and conforms to the `F` template parameter interface.

Operator Class: ComputeEulerRHS

```
Class ComputeEulerRHS // Corresponds to F in RK4.
{public:
void operator()(
DeltaVelocity& a_newDv,
const Real& a_time, const Real& a_dt,
const FieldData& a_velocity,
DeltaVelocity& a_oldDv);
}
```

State Data : FieldData (X)

```
class FieldData
{public:
  FieldData();
  FieldData(DBox a_grid,a_nComponent,int a_ghost,int a_M,
int a_N);
  ~FieldData();
  void fillGhosts();
  void increment(const Real& a_scalar,
                 const DeltaVelocity& a_fieldIncrement);
  void imposeConstraint();
  int m_components;
  DBox m_grid;
  int m_M,m_N,m_ghosts;
  RectMDArray<Real,DIM> m_data;
```

State Data: `DeltaVelocity (dX)`

```
class DeltaVelocity
{public:
  DeltaVelocity ();
  DeltaVelocity (DBox a_grid);
  ~DeltaVelocity();
  RectMDArray<Real,DIM>& getVelocity();
  void increment(const double& a_scalar,
                 const DeltaVelocity& a_fieldIncrement);
 ...
 private:
  DBox m_grid;
  RectMDArray<double,DIM> m_data;
}
```

Operator Class: `Projection`

```
class Projection
{public:
  Projection();
  Projection(int a_M);
  ~Projection();
  void applyProjection(RectMDArray<Real,DIM>& a_velocity)
const;
  void gradient(RectMDArray<Real,DIM>& a_vector,
          const RectMDArray<Real,DIM>& a_scalar);
  void divergence(RectMDArray<Real>& a_scalar,
      const RectMDArray<Real,DIM>& a_vector);
 ...
 private:
  int m_M,m_N;
  DBox m_grid;
  FFTPoissonSolver m_solver;
}
```

Operator Class: `FFTPoissonSolver`

```
class FFTPoissonSolver
{public:
  FFTPoissonSolver();
  FFTPoissonSolver(int a_M);
  ~FFTPoissonSolver();
  void solve(RectMDArray<Real>& a_Rhs);
 ...
 private:
  int m_M,m_N;
  Box m_grid;
  FFTMD m_fft;
}
```

function: Advection

```
void advectionOperator(
deltaVelocity& a_divuu,
const FieldData& a_velocity,
const DBox m_grid,
const double& a_h);
```

Organizing Your Project

MyProject/

- Code/
  - src/ (or src1/ , src2/ , ... ; or subdirectories of src)
    - <*.H, *.cpp files>
    - unitTests
    - o/ , d/
  - exec/ (or exec1/ , exec2/ , ...)
  - lib/
  - include/
- Documents/
  - designDocument/
  - doxygenDocument/
  - FinalReport/

Everything should be buildable by invoking "`make all`" in `Code/exec/` , possibly by invoking `make` from other directories.

What are Unit Tests ?

- For each class, want to test member functions to see whether the various member functions are correctly implemented.
  - (D(\vec{u} == const) == 0.)
  - $L^h(u^h) = O(h^P)$ as h-> 0.
  - Known analytic behavior for small systems.
- Unit tests should have their own makefile, make targets.
- Unit tests grow in time. As you identify fix bugs, you want to be sure they stay fixed.