CS 294-73
Software Engineering for Scientific Computing

http://www.cs.berkeley.edu/~colella/CS294

Lecture 20: Final Project

Final Project

- Done in teams of 2-4 people.
- Final report due at 11:59PM, 12/16/11.
- We set a set of intermediate deadlines that
  - Define a process
  - Define the work product of your project.
- The intermediate work will not be graded, but read to provide you feedback and guidance.
- In this lecture, I will go through in detail the process and deadlines in terms of the undergraduate project. The artifacts I will show you will be provided to the undergraduates, and they will need to fill in the remainder. Graduate projects will need to write these artifacts as part of their project.

Intermediate milestones

- Mathematical description of the problem being solved. This can be in the form of a research article, combined with a concise summary. (11/8/11)
- A complete mathematical specification of the discretization methods and / or numerical algorithms to be used to solve this problem. A specification of what will constitute a computational solution to this problem. (11/15/11)
- 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. (11/22/11).

Undergraduate Project: Incompressible Euler Equations

- Mathematical specification: 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:

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

$$\vec{u}(0,x_1,t) = \vec{u}(1,x-1,t) \ , \ \vec{u}(x_0,0,t) = \vec{u}(x_0,0,t)$$

- 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.*

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

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

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

Projection formulation

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

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

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

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

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.

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

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

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

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

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

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.

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

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

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

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

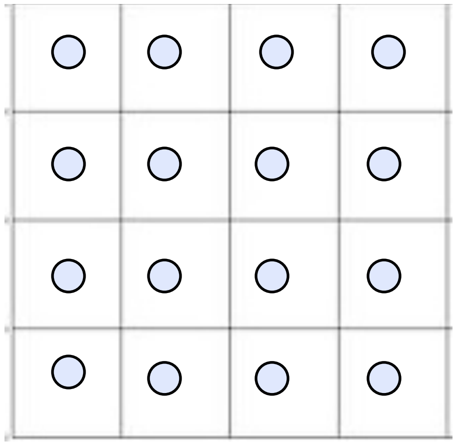
$$div(\vec{u})\big|_{t=0} = 0$$

Discretization in space

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

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

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



- Discretization of Projection operator

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

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

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

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

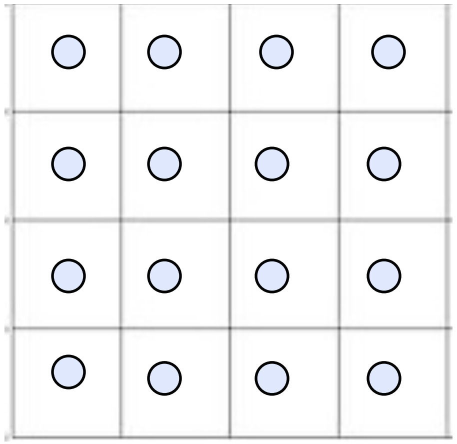
$$\text{note: } \Delta^h \neq div^h grad^h$$

Discretization in space

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

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

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



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

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

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

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.

FFT Solver for Poisson's equation

Given $f^h : \{0, \ldots, 2^M - 1\}^D \rightarrow \mathbb{R}$, 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).

Discretizing ODEs

Fourth-order Runge-Kutta:

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

$$
Q^{n+1} = Q^n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)
$$

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

Generalizes Simpson's rule for integrals.

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;
};
X <-> FieldData, F <-> ComputeEulerRHS, dX <->
 DeltaVelocity
```

For an expanded explanation of this class, see the handout for homework 4.

State Data

```
class FieldData
{public:
  FieldData();
  FieldData(Box a_grid,a_nComponent,int a_ghost,int a_M,
int a_N);
  ~FieldData();
  MDArray<double>& operator[](int a_component);
  void fillGhosts();
  const int& numComponents();
  void increment(const double& a_scalar,
                 const DeltaVelocity& a_fieldIncrement);
 ...
 private:
  int m_components;
  Box m_grid;
  int m_M,m_N,m_ghosts;
  MDArray<double>* m_data; /* need array of
MDArray<double>* of size m_components. */
}
```

State Data

```
class DeltaVelocity
{public:
  DeltaVelocity ();
  DeltaVelocity (Box a_grid,int a_nComponent);
  ~DeltaVelocity();
  MDArray<double>& operator[](int a_component);
  const int& numComponents();
  void increment(const double& a_scalar,
                 const DeltaVelocity& a_fieldIncrement);
 ...
 private:
  int m_components;
  Box m_grid;
  MDArray<double>* m_data; /* need array of
MDArray<double>* of size m_components. */
}
```

Operator Class: Projection

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

Operator Class: FFTPoissonSolver

```
class Projection
{public:
  FFTPoissonSolver();
  FFTPoissonSolver(int a_M);
  ~FFTPoissonSolver();
  void solve(MDArray<Real>& a_Rhs);
 ...
 private:
  int m_M,m_N;
  Box m_grid;
  FFTW1D* m_fft;
}
```

function: Advection

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

Operator Class: ComputeEulerRHS

```
Class ComputeEulerRHS
{public:
void operator()(
       DeltaVelocity& a_newDv,
       const double& a_time, const double& a_dt,
       const FieldData& a_velocity,
       DeltaVelocity& a_oldDv);
}
```

Operator Class: ComputeEulerRHS

`ComputeEulerRHS` implements the following algorithm.

$$\tilde{\vec{u}} = \vec{u}^{h,n} + \vec{\delta}^{in}$$

$$\vec{u}^{h,*} = \vec{u}^{h,n} - \Delta t D(\tilde{\vec{u}}\tilde{\vec{u}})$$

$$\vec{\delta}^{out} = \mathbb{P}^h\left(\vec{u}^{h,*}\right) - \vec{u}^{h,n}$$

Note that this is different from

$$\tilde{\vec{u}} = \vec{u}^{h,n} + \vec{\delta}^{in}$$

$$\vec{\delta}^{out} = -\Delta t \mathbb{P}^h(D(\tilde{\vec{u}}\tilde{\vec{u}}))$$