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).
Method of Lines

For the case of

\[ \frac{d\vec{u}^h}{dt} = -\mathbb{P}^h \left( \sum_{d=0}^{D-1} D^d (u^h_d \vec{u}^h) \right) \]

we compute

\[ \vec{u}^* = \vec{u}^n + \frac{1}{6} (k_1 + k_2 + k_3 + k_4) \]
\[ \vec{u}^{n+1} = \mathbb{P}^h (\vec{u}^*) \]
Fourth-order Runge-Kutta

\[
\frac{d\tilde{u}^h}{dt} = -\mathbb{P}^h \left( \sum_{d=0}^{D-1} D^d (u_d^h \tilde{u}^h) \right) \iff \frac{dQ}{dt} = F(Q, t)
\]

\[
k_1 = F(Q^n, t^n)
\]

\[
Q^{n,(1)} = Q^n + \frac{\Delta t}{2} k_1, \quad k_2 = F(Q^{n,(1)}, t^{n+\frac{1}{2}})
\]

\[
Q^{n,(2)} = Q^n + \frac{\Delta t}{2} k_2, \quad k_3 = F(Q^{n,(2)}, t^{n+\frac{1}{2}})
\]

\[
Q^{n,(3)} = Q^n + \Delta t k_3, \quad k_4 = F(Q^{n,(3)}, t^{n+1})
\]

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

\[
\frac{1}{6} (k_1 + 2k_2 + 2k_3 + k_4) = \frac{1}{\Delta t} \int_{t^n}^{t^n + \Delta t} f(Q(t), t) dt + O(\Delta t)^4
\]
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);
template <class X, class F, class dX>
void RK4<X,F,dX>::advance(double a_time, double a_dt, X& a_state)
{
    m_delta.init(a_state); // Initialize dX classes from a_state.
    m_k.init(a_state);
    m_f(m_k, a_time, a_dt, a_state); // compute k1
    m_delta.increment(sixth, m_k); m_k*=half; // increment must not change m_k !!
    m_f(m_k, a_time+half*a_dt, a_dt, a_state); // compute k2
    m_delta.increment(third, m_k); m_k*=half;
    m_f(m_k, a_time+half*a_dt, a_dt, a_state); // compute k3
    m_delta.increment(third, m_k);
    m_f(m_k, a_time+a_dt, a_dt, a_state); // compute k4
    m_delta.increment(sixth, m_k);
    a_state.increment(m_delta);
}
Comments on RK4

- Interface class done purely with templates (as opposed to inheritance).
- \textit{x} comes in as an argument of \texttt{advance}, so it keeps track of any time-dependent context required to execute the various other operators.
- \texttt{dx} \leftrightarrow \texttt{k}, \texttt{delta} very lightweight: data holder for information required to increment the solution.
- \texttt{F} is a class, but is really just a function pointer.
Operator Class: ComputeEulerRHS

ComputeEulerRHS implements

\[- \mathbb{P}^h \left( \sum_{d=0}^{D-1} D^d (u_d^h \bar{\omega}^h) \right)\]

and conforms to the \(F\) template parameter interface.

Class ComputeEulerRHS // Corresponds to \(F\) in RK4.

```cpp
{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(const Box& 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;
        Box m_grid;
        int m_M, m_N, m_ghosts;
        BoxData<Real, DIM> m_data;
        Projection m_project;
State Data: \texttt{DeltaVelocity} (dX)

class DeltaVelocity
{
public:
    DeltaVelocity ();
    DeltaVelocity (const Box& a_grid);
    ~DeltaVelocity();
    BoxData<Real,DIM>& getVelocity();
    void increment(const double& a_scalar,
                   const DeltaVelocity& a_fieldIncrement);

...

private:
    Box m_grid;
    BoxData<double,DIM> m_data;
}
Operator Class: Projection

\[ \mathbb{P}^h_D \left( \sum_{d=0}^{D-1} D^d (u_d^h \mu^h) \right) \]

class Projection
{
 public:
  Projection();
  Projection(int a_M);
  ~Projection();
  void applyProjection(BoxData<Real,DIM>& a_velocity) const;
  void gradient(BoxData<Real,DIM>& a_vector,
                const BoxData<Real,DIM>& a_scalar);
  void divergence(RectMDArray<Real>& a_scalar,
                  const BoxData<Real,DIM>& a_vector);
 ...

 private:
  int m_M,m_N;
  Box m_grid;
  FFTPoissonSolver m_solver;
}
Operator Class: FFTPoissonSolver

class FFTPoissonSolver
{
    public:
        FFTPoissonSolver();
        FFTPoissonSolver(int a_M);
        ~FFTPoissonSolver();
        void solve(BoxData<Real>& a_Rhs);
    ...

    private:
        int m_M, m_N;
        Box m_grid;
        FFTMD m_fft;
}
function: Advection

\[ -P^h \left( \sum_{d=0}^{D-1} D^d (u^h_d u^h) \right) \]

void advectionOperator(
    deltaVelocity& a_divuu,
    const FieldData& a_velocity,
    const Box 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.
  - \( D(\text{vec}\{u\} == \text{const}) == 0. \)
  - \( L^h(u^h) = O(h^p) \) as \( h \to 0. \)
  - 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.
Homework 4

PIC methods for vorticity.
• Algorithmic issues
• Software / implementation.
Particle method for vorticity transport in 2D.

\[
\frac{dx^k}{dt} = \vec{U}(x^k(t)) \quad \omega^k = \omega(x^k(0), 0) h^D
\]

\[
\vec{U}(x) = \sum_k \omega^k \nabla_x \frac{1}{|x|} \int \! G(x - y) \zeta_\varepsilon(y) \, dy
\]

\[
= \sum_k \omega^k \vec{U}_\varepsilon(x - x_k)
\]

\[
U_\varepsilon(z) = \nabla_z \left( \int \! G(z - y) \zeta_\varepsilon(y) \, dy \right)
\]

\[
G(z) = \frac{1}{2\pi} \log(|z|)
\]

\[
\nabla_x \frac{1}{|x|} = \left( \frac{\partial}{\partial x_2}, -\frac{\partial}{\partial x_1} \right)
\]

\[
\zeta_\varepsilon(z) = \frac{1}{\varepsilon D} \zeta\left(\frac{z}{\varepsilon}\right)
\]

For epsilon = 0, we get a point vortex method, with a singular velocity field, so some sort of regularization (epsilon > 0) is required.
Regularized particle methods

\[
\zeta(z) = \frac{(-140(1 - |z|)^3 + 420(1 - |z|)^4 - 252(1 - |z|)^5)}{2\pi}
\]
Regularized particle methods

Can think of this as regularizing the potential induced by a collection of delta functions.

\[ \omega(x) = \sum_k \omega_k \delta(x - x_k) \]

\[ \Psi(x) = (G \ast \omega)(x) = \sum_k \omega_k G(x - x_k) , \]

\[ \Psi(x) = (G_\varepsilon \ast \omega)(x) = \sum_k \omega_k G_\varepsilon(x - x_k) \]

\[ U = \nabla^\perp \Psi \]
Start with a collection of particles, each of which induces a contribution to the stream function in any point in space via convolution.

**PIC:** Replace continuous convolution with discrete convolution on a rectangular grid:

- Replace \( \omega_k \delta(x - x_k) \rightarrow \omega_i^+ = \omega_k \delta^h(ih - x_k) \)
- Compute stream function on the grid as discrete convolution.
- Interpolate finite-difference approximation to velocity on the grid to particles.
Particle-in-Cell (PIC)

Given the particle locations, can compute a velocity field at those locations.

\[
\omega_i = \sum_k \omega^k \delta_h(ih - x^k), \quad \omega^k = \omega^k(x^k(0), 0)
\]

\[
\psi_i = \sum_j G(ih - jh) \omega_j \quad \text{G}(z) = \frac{1}{2\pi} \log(|z|)
\]

\[
\vec{U}_i = D^\perp(\psi)_i
\]

\[
(D_1 \psi) = \frac{\psi_{i+1,0} - \psi_{i-1,0}}{2h}, \quad (D_2 \psi) = \frac{\psi_{i+0,1} - \psi_{i-0,1}}{2h}
\]

\[
U(x^k) = \sum_i U_i \delta_h(x^k - ih) \quad \delta_h(z) = \delta\left(\frac{z_1}{h}\right)\delta\left(\frac{z_2}{h}\right)
\]

Deposition
Discrete convolution
Grid velocity
Particle velocity
Particle-in-Cell (PIC)

\[ U(x^k) = \sum_i U_i \delta_h(x^k - i) \]

\[ \delta_h(z) = \delta\left(\frac{z_1}{h}\right)\delta\left(\frac{z_2}{h}\right) \]
Algorithmic design considerations

- Initialization: particles deposited on a grid with spacing $h_p$.
- What is the relationship between epsilon, $h$, $h_p$? Our choices: $h_p = h/n_p$, $n_p$ a positive integer, epsilon = $h$.
- There is a good convergence theory for vortex methods (i.e. no grid). The grid introduces its own short-range regularization. Can we eliminate that, while still retaining the (fast) grid representation of the long-range effects? (Answer: yes - PPPM).
We will provide an implementation of Hockney’s algorithm for computing

\[ \psi_i = \sum_j G_\epsilon (ih - jh) \omega_j \]

as well as the templated RK4 class.

You will implement the rest of the particle method:
- Deposition of charge and calculation of velocity field (template class \( F \)).
- The template classes \( x, \ dx \) appropriate to the PIC method.
template <class X, class F, class dX> class RK4
{
    public: void advance(double a_time, double a_dt, X& a_state);
    private:
        dX m_k;
        dX m_delta;
        F m_f;
};
template <class X, class F, class dX>
void RK4<X,F,dX>::advance(double a_time, double a_dt, X& a_state)
{
    m_delta.init(a_state);  // Initialize dX classes from a_state.
    m_k.init(a_state);
    m_f(m_k, a_time, a_dt, a_state); // compute k1
    m_delta.increment(sixth, m_k); m_k*=half; // increment must not change m_k !!
    m_f(m_k, a_time+half*a_dt, a_dt, a_state); // compute k2
    m_delta.increment(third, m_k);  m_k*=half;
    m_f(m_k, a_time+half*a_dt, a_dt, a_state); // compute k3
    m_delta.increment(third, m_k);
    m_f(m_k, a_time+a_dt, a_dt, a_state); // compute k4
    m_delta.increment(sixth, m_k);
    a_state.increment(m_delta);
}
Particle Classes

class DX // displacement.
{
    public:
        DX()
        { ... }    
        array<Real, DIM> m_x;
        inline void increment(double a_scale, const DX& a_rhs)
        { ... }
        inline void operator*=(double a_scale){ ... };    
    
}
Particle Classes

class Particle
{
public:

    array<Real, DIM> m_x;
    Real strength;

    // (*this).m_x -> (*this).m_x + a_shift.m_x;
    inline void increment(const DX& a_shift)
    {
        ...
    }
};
Particle Classes

class ParticleSet; // Forward declaration.  
                   // Needed to avoid circular dependency.

class ParticleShift // Corresponds to dX in RK4.
{
    public:
        vector<DX> m_particles;
        void init(const ParticleSet& a_particles);
        void increment(double a_scale, const ParticleShift& a_rhs);
        void operator*=(double a_scale);
        void setToZero();
    }

All that the ParticleShift declaration is allowed to know about ParticleSet is that it is a class. Pointers, references, but not values (why?).
Particle Classes

class ParticleSet

// Corresponds to template class X in RK4.
{
  Public:
    ParticleSet(shared_ptr<ConvKernel>& a_kerptr,
                Box& a_box,
                double& a_h,
                array<Real, DIM>& a_lowCorner,
                int a_M);
    ParticleSet(){};
    vector<Particle> m_particles;
    double m_h;
    Box m_box;
    array<Real, DIM> m_lowCorner;
    Hockney m_hockney;

    void increment(const ParticleShift& a_shift);
};
Particle Classes

class ParticleVelocities // Implements template class F
{
public:
    ParticleVelocities();
    void operator()(ParticleShift& a_k, 
                    const Real& a_time, 
                    const Real& dt, 
                    ParticleSet& a_state);
};

ParticleVelocities has no state, only a default constructor. All objects required to evaluate the velocity must come in via a_state. This is one of the reasons why all of the member data of ParticleSet are public (alternative: friend classes).
Hockney Class

class Hockney
{
    public:

    Hockney();
    Hockney(shared_ptr<ConvKernel>& a_kerPtr, const double& a_h, int a_M);
    void define(shared_ptr<ConvKernel>& a_kerPtr, const double& a_h, int a_M);
    void convolve(RectMDArray<Real>& a_rhs);
    ~Hockney(){};

    protected:

    ...
};

You will have to define your Hockney in the ParticleSet constructor.
a_kerptr, a_M, and a_h are passed through to you via the arguments.
m_hockney.convolve(...) called in ParticleVelocities.
template <class X, class F, class dX>
void RK4<X,F,dX>::advance(double a_time, double a_dt, X& a_state) {
    m_delta.init(a_state);  // Initialize dx classes from a_state.
    m_k.init(a_state);
    m_f(m_k, a_time, a_dt, a_state);  // compute k1
    // F::operator()(dX&,double&,double&,X&);
    m_delta.increment(sixth, m_k);  // increment must not change m_k !!
    m_k*=half;
    m_f(m_k, a_time+half*a_dt, a_dt, a_state);  // compute k2
    m_delta.increment(third, m_k);  // compute k3
    m_k*=half;
    m_f(m_k, a_time+half*a_dt, a_dt, a_state);
    m_delta.increment(third, m_k);
    m_f(m_k, a_time+a_dt, a_dt, a_state);  // compute k4
    m_delta.increment(sixth, m_k);
    a_state.increment(m_delta);
}
Test Problems

\[ T = \begin{array}{ccc}
0.00000 & 2.50000 & 5.00000 \\
10.00000 & 15.00000 & 20.00000 \\
27.50000 & 37.50000 & 50.00000 \\
\end{array} \quad \begin{array}{ccc}
Eff = 0.752 & 0.794 & 0.768 \\
0.852 & 0.825 & 0.773 \\
0.755 & 0.823 & 0.872 \\
\end{array} \]
Back to organization / make process

Top-level directory structure:

src/
  src/fftTools/  -  Contains only FFT1D, FFT1DW, FFTMD
  src/Particles/  -  .H only – you will put your .cpp files here
  src/Hockney/
utils/
  utils/ Writers/
  utils/ timer/
exec/
exec/

exec/GNUmakefile
exec/vortexTest.cpp
exec/o2d/
exec/d2d/

Makefile will create o2d/, d2d/ directories.
- o2d/ will contain the .o files generated in the build process.
- d2d/ will contain the .d files (dependencies) generated in the build process.
GNUmakefile

HOME = ./..
WRITERS = $(HOME)/utils/Writers
PROTO = $(PROTO_DIR)/include
FFT = $(HOME)/src/fftTools
CONV = $(HOME)/src/Hockney
PARTICLES = $(HOME)/src/Particles
TIMER = $(HOME)/utils/timer

VPATH= . $(HOME) $(PARTICLES) $(PROTO) $(FFT) $(CONV) $(TIMER) $(WRITERS)  

Tells Gmake where to search for file dependencies.

DIM=2

#CXX=g++
CXX=clang++

FFTWDIR = /usr/local

#CFLAGS = -g -Wall
CXXFLAGS = -O3

CFLAGS += -std=c++11 -I/. -I$(PARTICLES) -I$(PROTO) -I$(FFT) -I$(CONV) -I$(TIMER) -I$(WRITERS) -I$(FFTWDIR)/include

Tells the compiler where to look for #include files

CXXFLAGS += -D DIM=$(DIM)
GNUmakefile

odir = ./o.$(DIM)d
ddir = ./d.$(DIM)d

LIBS:= -L$(FFTWDIR)/lib -lfftw3

SRCFILES:= $(notdir $(wildcard $(TIMER)/.*.cpp $(RMDA)/.*.cpp $(Writers)/.*.cpp $(FFT)/.*.cpp ./*.cpp $(CONV)/.*.cpp $(Particles)/.*.cpp))
This defines our .cpp sources. notdir strips off the directory part of each element in the list.

OBJS:=$(patsubst %.cpp, $(odir)/%.o, $(SRCFILES))
This defines our object files. With this definition, make will look for them in ./$(odir)

DEPS:=$(patsubst $(odir)/%.o, $(ddir)/%.d, $(OBJS))
This defines our dependency files to be stored in $(ddir). More on them shortly.
GNUmakefile

odir = ./o.$(DIM)d
ddir = ./d.$(DIM)d

LIBS:= -L$(FFTWDIR)/lib -lfftw3

SRCFILES:= $(notdir $(wildcard $(TIMER)//*.cpp $(RMDA)/*.cpp $(Writers)//*.cpp $(FFT)//*.cpp ./*.cpp $(conv)//*.cpp $(Particles)//*.cpp))
This defines our .cpp sources. notdir strips off the directory part of each element in the list.

OBJBS:=$(patsubst %.cpp, $(odir)/%.o, $(SRCFILES))
This defines our object files. With this definition, make will look for them in ./$(odir)

DEPS:=$(patsubst $(odir)/%.o, $(ddir)/%.d, $(OBJBS))
This defines our dependency files to be stored in $(ddir). More on them shortly.
Rule for making .o files.

$(odir)/%.o:%.cpp GNUmakefile
<TAB> mkdir -p $(odir); \
    $(CXX) -c $(CFLAGS) $< -o $@
<TAB> mkdir -p $(ddir); \
    $(CXX) -MM $(CFLAGS) $< \
    | sed '1s/^/o.$(DIM)d//' > $*.d \
    mv $*.d $(ddir)

For each file of the form %.cpp accessible in VPATH, it makes a target of the form $(odir)/%.o. $(CXX) -c: compile only, output object file to argument following -o. In this case, it is $@.

This rule also makes a file of the form $*.d and moves it to $(ddir). ($* is the stem of %.cpp).

mkdir -p creates a directory only if there isn’t one.

Questions:
What do the .d files do for us?
What’s that stuff in the middle?
Rule for making .o files.

\[-\text{include } $(DEPS) \] — includes all of the files in $(DEPS). In this case, that is all files of the form \texttt{d.2d/*.d}. Placed at the end of the makefile. Each file is a makefile, with one rule in it.

Let’s look at \texttt{d.2d/Box.d}:

\[
o.2d/DBox.o: \ldots/src/RectMDArray/DBox.cpp \ldots/src/RectMDArray/DBox.H \ldots/src/RectMDArray/Point.H \ldots/src/RectMDArray/PointImplem.H
\]

It is a rule for making \texttt{o.2d/DBox.o}. But it has no recipes. The effect of this is to add dependencies to already-existing ones (in this case, for \texttt{o.2d/DBox.o}).
Rule for making .o files.

```
<TAB> mkdir -p $(ddir);$(CXX) -MM $(CFLAGS) $< | sed '1s/^/o. $(DIM)d\//' > $*.d;mv $*.d $(ddir)

- | : take output from stdout and stream it to stdin for the next program.
- sed : streaming editor for Unix systems. Uses similar commands to vi.
- > : take output from stdout and send it to file on rhs.
- mv : move the file.
```
Rule for building vortex2D.exe

vortex2D: GNUmakefile $(OBJS)
    $(CXX) $(CFLAGS) $(OBJS) $(LIBS) -o vortex$(DIM)D.exe

- It depends on the makefile and everything in the $(OBJS) list.
- The rule is to run the compiler with no .cpp files, so all it is doing is linking the object files (so main had better be there).
Housekeeping, utilities.

clean:
<TAB>  rm -r *.exe $(odir) $(ddir)

listsnc:
<TAB>  @echo $(SRCFILES)

listobj:
<TAB>  @echo $(OBJJS)

listdep:
<TAB>  @echo $(DEPS)
Results Demo
Visit for Particles

From the directory you ran vortex2d.exe:

• Visit
  - open -> PART*.vtk (database)
  - controls -> view -> 2D -> set viewport, window to 0 1 0 1, apply, dismiss
  - add -> mesh -> mesh
  - draw
  - For small number of particles: click on mesh, change point type to sphere, pixels to 10