Lecture 3: A sample of Hybrid Monte Carlo

Bálint Joó

Jefferson Lab

INT Summer School, Lattice QCD and Applications

Seattle:

http://www.jlab.org/~bjoo/Lecture3.pdf

Goals

- We're going to write a Hybrid Monte Carlo Code
  - For Wilson Gauge Action + 2 Flavours of Unpreconditioned Wilson Fermions
- We'll work out a C++ class structure for Fields. HMC, MD integrators
  - Gauge action and 2 Flavor Fermion Action force terms
- Warning:
  - It is unlikely that you can write and test all this code from scratch in 90 minutes (took me 1 day or so)
  - This lecture may end up spilling over into Lecture 4.

Getting the Code

- You can get the code in the usual way from anonymous CVS

```
export CVSROOT=:pserver:anonymous@cvs.jlab.org:/group/lattice/cvsroot
cvs checkout seattle_tut/example3
```

- Remember, you need to change the CONFIG variable in **`Makefile`** and **`lib/Makefile`**
- **`make qcd`** will make the QCD HMC example
- **`make sho`** will make the HMC for a Harmonic Oscillator
- The classes discussed in this tutorial mostly live in **`lib/`** in header files.
  - **`abs_xxxx.*`** Abstract Classes (AbsIntegrator etc)
  - **`qcd_xxx.*`** QCD Classes
  - **`sho_xxx.*`** SHO Classes

The Basic Hybrid Monte Carlo Game:

1) Start off with a state: (p, q)

2) Refresh any pseudofermion fields in your Hamiltonian

3) Refresh the momenta

4) Save the state

5) Perform a Molecular Dynamics Trajectory (MD) of length τ

$$(p, q) \overset{MD(\tau)}{\longrightarrow} (p', q')$$

6) Compute energy change:

$$\delta H = H(p', q') - H(p, q)$$

7) Accept/ Reject (p',q') with probability:

$$P_{\text{acc}} = \min\left(1, e^{-\delta H}\right)$$

8) In case of rejection the new state is (p,q)

9) Go to step 1

The Hamiltonian

- We have our (fictitious) ***Hamiltonian*** (for MD) of the form:

$$H = \frac{1}{2}p^2 + S_1(q) + S_2(q) + \ldots$$

- We will refer to $S_1(q)$, $S_2(q)$ etc as ***Monomials***
  - The sum of the monomials makes up our ***Action***
- So our Hamiltonian is a collection of
  - the piece from the momenta
  - a collection of monomials
- The energy is likewise a sum of the momentum term + the sum of the actions from the monomials
- The MD force is just the sum of the forces from the monomials.

Design Issues

- We'd like a fairly generic framework
  - Just as easy to to Lattice QCD as a Simple Harmonic Oscillator
    - We'll use base classes, virtual functions, defaults & derivations to specify abstractions
    - We will use templates to cope with the variations in the types of the fields in the states
    - We will hide pseudofermion fields inside the fermionic monomials.

We already have some ideas for classes

- We will needs some class to hold the state (p,q)
  - Template this on the types of p and q
- We will need some classes for the monomials $S_i(q)$
  - To compute the action $S_i(q)$
  - To compute the force from $S_i(q)$
- We need a Hamiltonian to aggregate the monomials
- We need an integrator to do the MD
- We need an overall driver to do the rest of the HMC steps.

Abstract Classes: The Field State

```
template <typename P, typename Q>
class AbsFieldState
{
public:
  //! Virtual destructor
  virtual ~AbsFieldState<P,Q>() {}

  //! Clone the state
  virtual AbsFieldState<P,Q>* clone(void) const = 0;

  //! Read
  virtual const P& getP(void) const = 0;
  virtual const Q& getQ(void) const = 0;

  //! Write
  virtual P& getP(void) = 0;
  virtual Q& getQ(void) = 0;
};
```

Templates for "momenta" and "coordinates"

Discuss this later

Returns read/only (const) references "Accessors"

Returns writable references "Manipulators"

Abstract Classes: The Monomials

```
template<typename P, typename Q>
class AbsMonomial {
public:
    //! virtual destructor:
    virtual ~AbsMonomial() {}

    //! Compute Force for the system... Not specified how to actually do this
    // yet. s is the state, F is the computed force
    virtual void dsdq(P& F, const Q& s) const  = 0;

    //! Compute the total action
    virtual Double S(const AbsFieldState<P,Q>& s) const = 0;

    //! Refresh pseudofermion fields if any
    virtual
    void refreshInternalFields(const AbsFieldState<P,Q>& field_state) = 0;

};
```

Force is same type as momenta

DON'T PANIC

We'll return to this later

Abstract Classes: The Hamiltonian

```
template<typename P, typename Q>
class AbsHamiltonian
{
public:
  virtual ~AbsHamiltonian() {} // Virtual dsstructor

  //! get the number of monomials
  virtual int numMonomials(void) const = 0;

  //! get at a specific monomial (Read Only)
  virtual const AbsMonomial<P,Q>& getMonomial(int i) const = 0;

  //! get at a specific monomial (Read/Write)
  virtual AbsMonomial<P,Q>& getMonomial(int i) = 0;
  ...
```

These methods are accessors/manipulators. We haven't declared the storage yet. They'll allow defaults to work...

Abstract Classes: Hamiltonian defaults

- Aggregate Energies (still within class body...)

```
virtual Double mesKE(const AbsFieldState<P,Q>& s) const {
   Double KE=norm2(s.getP());
   return KE;
}
```

Kinetic energy default: suitable for QCD

```
virtual Double mesPE(const AbsFieldState<P,Q>& s) const {
  Double PE;
  PE = getMonomial(0).S(s);
  for(int i=1; i < numMonomials(); i++) { PE += getMonomial(i).S(s); }
  return PE;
}
```

Delegate computation to the Monomials

```
virtual void  mesE(const AbsFieldState<P,Q>& s, Double& KE, Double& PE)
const {
   KE = mesKE(s);
   PE = mesPE(s);
}
```

Abstract Classes: Hamiltonian Defaults

```
  void dsdq(P& F, const Q& s) const {
    P F_tmp;
    getMonomial(0).dsdq(F,s);
    for(int i=1; i < numMonomials(); i++) {
      (getMonomial(i)).dsdq(F_tmp, s);
      F += F_tmp;
    }
  }
  //! Refresh pseudofermsions (if any)
  virtual void refreshInternalFields(const AbsFieldState<P,Q>& s) {
       getMonomial(0).refreshInternalFields(s);
       for(int i=1; i < numMonomials(); i++) {
         getMonomial(i).refreshInternalFields(s);
       }
   }
} ; // End Class AbsHamiltonian
```

Aggregate Forces

Call the field refreshment on every monomial

Abstract Classes: The Integrator

- This is essentially just a function object:

```
template<typename P, typename Q>
class AbsIntegrator {
public:
  //! Virtual destructor
  virtual ~AbsIntegrator(void) {}

  //! Do an integration of length n*delta tau in n steps.
  virtual void operator()(AbsFieldState<P,Q>& s,
                          const Real traj_length) const = 0;

};
```

$$MD : s \rightarrow s'$$

Here I just define an interface! No details of the integration yet.

A Leapfrog Integrator:

```
template<typename P, typename Q>
class AbsLeapfrogIntegrator : public AbsIntegrator<P,Q>{
public:
  virtual ~AbsLeapfrogIntegrator(void) {} // Virtual destructor

  // operator() on next slide

  virtual int getNumSteps(void) const = 0;
protected:
```

For use in defaults

Symplectic Updates:

```
virtual void leapP(AbsFieldState<P,Q>& s,
                   const Real dt) const =0;

  virtual void leapQ(AbsFieldState<P,Q>& s,
                   const Real dt) const=0;
};
```

$$p \leftarrow p + \delta\tau \; F(q)$$

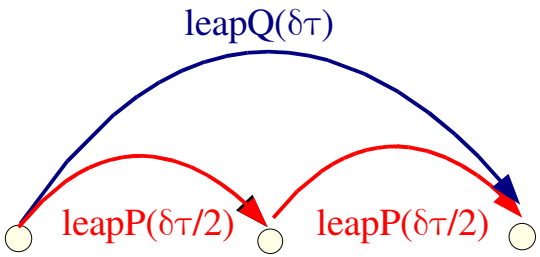
$$q \leftarrow q + \delta\tau p$$

A Leapfrog Integrator

```
// Default Implementation
virtual void operator()(AbsFieldState<P,Q>& s,
                        const Real traj_length) const {

  int n_steps = getNumSteps();
  Real dt = traj_length / Real(n_steps);
  Real dtby2 = dt/Real(2);

  leapP(s, dtby2);  // First Half Step
  leapQ(s, dt);     // First Full Step
  for(int i=0; i < n_steps-1; i++) {
     leapP(s, dt);
     leapQ(s, dt);
  }
  leapP(s, dtby2); // Last Half Step
}
```



A C++ Detour: References & Smart Pointers

- We can't create an instance of a class with undefined virtual functions.

```
AbsFieldState<Real, Real> state;
```

Not allowed!

- We could create a reference but only if we refer to something.

```
SHOFieldState sho_state(p,q);
AbsFieldState<Real, Real>& state=sho_state;
```

- Just creating a reference without referring to anything i.e:

```
AbsFieldState<Real, Real>& state;
```

  - is an uninitialized reference and is defined in C++ as a programming error.

C++ Detour: References & Smart Pointers

- We can dynamically create the derived state:

```
AbsFieldState<Real,Real>*  state;
state = new SHOFieldState(p,q);
```

  - This is OK. But now, we have to remember to call `delete` when we are done with the state or we'll suffer a **MEMORY LEAK.**
- What we need is a "smart pointer" that
  - can wrap the pointer returned by new,
  - keep track of "live references" to the object pointed to
  - call `delete` when the object has no further references to it
- The `Handle<>` class provides such a reference counting smart pointer

Handle<> from Stroustrup

- The **Handle** is templated, so we can wrap any pointer with it

```
Handle< AbsFieldState<Real,Real> > s = new SHOState(p,q);
```

  - s keeps a reference count (1) that increases to 2 when we make a copy of the pointer:

```
Handle< AbsFieldState<Real,Real> > s2 = s;
```

  - Now **s** can go out of scope. The reference count falls back to 1, so **s2** is not deleted
  - Then **s2** can go out of scope. The reference count decreases again. It reaches 0. Now **s2** is deleted:

Reference Counting...

```
{
 Handle< AbsFieldState<Real, Real> > s2;
 {
  Handle< AbsFieldState<Real, Real> > s1=new SHOState(p,q);
```

count = 1

```
   s2 = s1;
```

count = 2

s1 disappears here => count = 1 state not **delete**-d

```
 }
```

s2 disappears here => count = 0 => Destructor of Handle<> calls **delete**

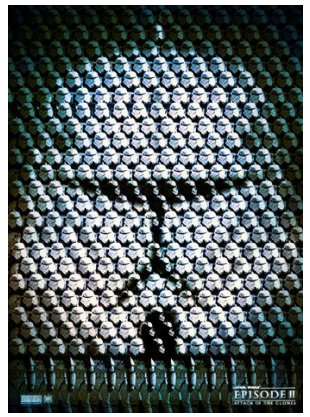
```
}
```

Final word on Smart Pointers

- Our “smart pointer” is implemented in lib/handle.h
- We use it extensively in chroma
- There are other kinds of smart pointer out there
  - eg: in the boost library.

More C++-isms: Attack of the clones!

- We can't create an abstract class, we cannot copy it either.
- What if we want to save a copy?
  - Base class defines a virtual function



No arguments

```
virtual AbsFieldState<P,Q>* clone() = 0;
```

  - Inheriting class implements this e.g:

```
SHOFieldState* clone() { return new SHOFieldState(...); }
```

  - Then we can call the `clone()` function from the abstract class.

Use constructor here, rather than =

```
Handle< AbsFieldState<P,Q> >  s_old(s.clone());
```

- Inheriting/Derived class MUST have enough information to clone itself...

Abstract Classes: The HMC

```
template<typename P, typename Q>
 class AbsHMCTrj {
 public:
   virtual ~AbsHMCTrj() {};
   // operator() on next slide
 protected:
   // Get at the Exact Hamiltonian
   virtual AbsHamiltonian<P,Q>& getMCHamiltonian(void) = 0;

   // Get at the Integrator
   virtual AbsIntegrator<P,Q>& getMDIntegrator(void)  = 0;

   // Get at the MD traj length
   virtual Real getMDTrajLength(void) const =0;

   virtual void refreshP(AbsFieldState<P,Q>& state) const = 0;
   virtual bool acceptReject(const Double& DeltaH) const = 0;
 };
```

Functions so we can write the default operator()

Access to encapsulated information

Monte Carlo related

HMC: The Real Meat & Potatoes!

```
virtual void operator()(AbsFieldState<P,Q>& s, const bool WarmUpP) {
  AbsIntegrator<P,Q>& MD = getMDIntegrator();
  AbsHamiltonian<P,Q>& H_MC = getMCHamiltonian();

  refreshP(s);
  H_MC.refreshInternalFields(s);

 Handle< AbsFieldState<P,Q> >  s_old(s.clone());

  Double KE_old, PE_old, KE, PE;
  H_MC.mesE(s, KE_old, PE_old);
  MD(s, getMDTrajLength());
  H_MC.mesE(s, KE, PE);

  Double DeltaKE = KE - KE_old; Double DeltaPE = PE - PE_old;
  Double DeltaH  = DeltaKE + DeltaPE;
  Double AccProb = where(DeltaH < 0.0, Double(1), exp(-DeltaH));
  QDPIO::cout << "AccProb=" << AccProb << endl;
  if( ! WarmUpP ) {
    bool acceptTestResult = acceptReject(DeltaH);
    QDPIO::cout << "AcceptP=" << acceptTestResult << endl;
    if ( ! acceptTestResult ) {
      s.getQ() = s_old->getQ();
      s.getP() = s_old->getP();
    }
  }
}
```

Field Refreshment

Save (p,q)

MD, compute energies before and after

$$P_{\mathrm{acc}} = \min\left(1, e^{-\delta H}\right)$$

Accept/Reject

If we don't accept the new state is the old one

And we're done?

- Sadly not. We have a good framework but:
  - These classes are abstract. We cannot 'create' instances of them.
    - We need derived (client) classes appropriate to the system we are simulating
    - However, these classes must supply 'tightly' defined interfaces
    - Most of the hard work is encoded in our defaults.
      - We don't need to rewrite MD, or HMC ...
- Due to recapping previous lectures and answering questions, we've run out of time today. We will write the concrete QCD implementing classes in Lecture 4.