Lecture 4: A sample of Hybrid Monte Carlo

Bálint Joó
Scientific Computing Group
Jefferson Lab

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:
  - This will be a long lecture.
  - I will focus on the class design, and skip over simple implementation methods where appropriate.

Getting the Code

- We'll be working with the code in **`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 t

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

6) Compute energy change:

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

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

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

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:

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

- 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 do 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"

lib/field_state.h

Abstract Classes: The Monomials

lib/abs_monomial.h

```
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;

};
```

- dsdq() - force term
- S()   - evalate action
- refreshInternalFields() - stub for monomials with p.f. fields

Abstract Classes: The Hamiltonian

lib/abs_hamiltonian.h

```
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;
}

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;
}

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

`lib/abs_hamiltonian.h`

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

lib/abs_hamiltonian.h

Abstract Classes: The Integrator

`lib/abs_integrator.h`

- Code this up as a function object:

$$MD : s \rightarrow s'$$

```
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;

};
```

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:


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

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

For use in defaults

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

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

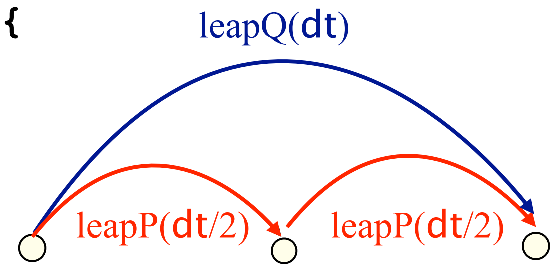
lib/abs_integrator.h

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
}
```



I have now written the Leapfrog Integrator logic for all actions and all field state combinations.

BUT : I will need to supply leapP() and leapQ() for each one.

This is an illustration of the principle of separation of concerns

A C++ Detour: References & Smart Pointers

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

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

$\times$ 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);

  s2 = s1;
 }
}
```

count = 1

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

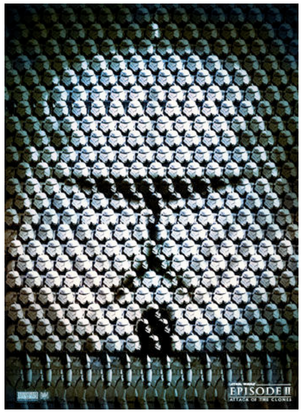
- Our "smart pointer" is implemented in lib/handle.h
- We use it extensively in chroma
- There are other kinds of smart pointer out there
  - eg: in the boost library.

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

HMC specific

lib/abs_hmc.h

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

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

Accept/Reject

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

lib/abs_hmc.h

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: **implementation**
    - However, these classes must supply 'tightly' defined interfaces
    - A lot of this is dull-code (implement get/set methods etc) -- we’ll skip over these. See them in the files.
    - Most of the hard work is encoded in our defaults.
      - We don't need to rewrite MD, or HMC ...

Concrete Classes: field state

- Our QCD state will consist of
  - `multi1d<LatticeColorMatrix>` for momenta
  - `multi1d<LatticeColorMatrix>` for the gauge fields
- Typing these involves a lot of finger exercise so we can make some abbreviations for shorthand:

```
namespace HMC {
  typedef multi1d<LatticeColorMatrix> GaugeP;
  typedef multi1d<LatticeColorMatrix> GaugeQ;

  class GaugeFieldState : public AbsFieldState<GaugeP,GaugeQ> {
  public:
   ...
  private:
     GaugeP p; // The momenta in this state
     GaugeQ q; // The "coordinates" in this state
  };
```

Shorthand

Gauge Field State is an "implemention" of AbsFieldState<P,Q> with P=GaugeP and Q=GaugeQ

`lib/qcd_field_state.h`

Constructing/Copying – C++ boilerplate

- In order to create and copy the GaugeState we need some constructors:

```
GaugeFieldState(const GaugeP& p_,       // Constructor
                const GaugeQ& q_) {
  p.resize(Nd); q.resize(Nd);           // Just copy p_ and q_
  for(int mu=0; mu < Nd; mu++) {        // To our internal storage
    p[mu] = p_[mu]; q[mu] = q_[mu];
  }
}

GaugeFieldState(const GaugeFieldState& s)  { // Copy
  p.resize(Nd); q.resize(Nd);
  for(int mu=0; mu < Nd; mu++) {
    p[mu] = s.p[mu]; q[mu] = s.q[mu];
  }
}

~GaugeFieldState() {}; // multi1d<>-s clean up automatically
```

lib/qcd_field_state.h

Now fulfill the rest of the interface

- We now need to supply the access methods and the clone() function

```
// Clone function -- covariant return type
GaugeFieldState* clone(void) const {
  return new GaugeFieldState(*this);
}

// Accessors
const GaugeP& getP(void) const { return p; }
const GaugeQ& getQ(void) const { return q; }

// Manipulators
GaugeP& getP(void)  { return p; }
GaugeQ& getQ(void)  { return q; }
```

see all of this in the file lib/qcd_field_state.h

`lib/qcd_field_state.h`

Now the Hamiltonian

- Again, we need to add constructors

```
class QCDHamiltonian : public AbsHamiltonian<GaugeP, GaugeQ>
{
public:

  //! virtual descructor:
  ~QCDHamiltonian() {}

  //! Constructor
  QCDHamiltonian(multi1d< Handle<AbsMonomial<GaugeP, GaugeQ> > >& m_) {
    monomials.resize(m_.size());
    for(int i=0; i < monomials.size(); i++) {
      monomials[i] = (m_[i]);
    }
  }

  ...
private:
  multi1d< Handle< AbsMonomial<GaugeP, GaugeQ> > > monomials;
};
```

Array of Handles of Monomials

Copy to internal monomial list

`lib/qcd_hamiltonian.h`

Fulfilling the Interface

- Then we just fulfill the interface that has no defaults (field refreshment, accessors, etc)

```
int numMonomials(void) const {
  return monomials.size();
}

const AbsMonomial<GaugeP, GaugeQ>& getMonomial(int i) const {
  return *(monomials[i]);
}

AbsMonomial<GaugeP, GaugeQ>& getMonomial(int i) {
  return *(monomials[i]);
}
```

The * “de-references” the Handle<>

- NOTE: The cool bit! ***Everything else*** is already done for us in the AbsHamiltonian. .

Next Low Hanging Fruit: Leapfrog

- Here we need to do a bit of work but let's do the easy part first: Constructors etc.

```
class QCDLeapfrog : public AbsLeapfrogIntegrator<GaugeP, GaugeQ> {
public:
  ~QCDLeapfrog(void) {}    // Destructor

  // Constructor
  QCDLeapfrog( AbsHamiltonian<GaugeP, GaugeQ>& H_, int n_steps_) : H(H_),
                                                             n_steps(n_steps_)  {}

 int getNumSteps(void) const { return n_steps; }

protected:
   // leapP and leapQ on next slides
private:
  int n_steps;
  AbsHamiltonian<GaugeP,GaugeQ>& H;
};
```

`lib/qcd_leapfrog.h`

LeapP

- This is the step in the leapfrog where we update the momenta:

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

- For QCD, the q are the SU(3) link matrices U
- For an action S, the force is defined as:

$$F(U) = T\left[U\mu \frac{\partial S(U)}{\partial U_\mu} \right]$$

- T[U] is the ***traceless anti-hermitian projection*** back into the Lie algebra su(3):

$$u = \frac{1}{2}\left[\left(U - U^\dagger\right) - \frac{i}{N_c}\mathrm{Tr}\ \left(U - U^\dagger\right) I_{N_c} \right]$$

LeapP

- We don't need to implement the T[] in the forces themselves, but only on the sum of forces in the leapP. We would need to put it in the forces , if we want to monitor them.
- The code for T[] is simple (lib/taproj.[h,cc]) :

```
void taproj(LatticeColorMatrix& a)
{
  LatticeColorMatrix aux_1 = a;
  a -= adj(aux_1);
  if (Nc > 1) {
    // tmp = Im Tr[ a ]
    LatticeReal tmp = imag(trace(a));
    tmp *= (Real(1)/Real(Nc));
    LatticeColorMatrix aux = cmplx(0, tmp);
    a -= aux;
  }
  a *= (Real(1)/Real(2));
}
```

LeapP()

- With this in mind we have the following simple code for the SU(3) leapP:

```
protected:
   void leapP(AbsFieldState<GaugeP,GaugeQ>& s, Real dt) const {
     GaugeP F(Nd);
     H.dsdq(F, s.getQ()); // Get the total force for H

     for(int mu =0; mu < Nd; mu++) {
       // p <- p + dt* T[ F ]
       // 1) project the force
       Example::taproj( F[mu] );

        // 2) Update the momenta.
       (s.getP())[mu] += dt * F[mu];

     }
   }
```

`lib/qcd_leapfrog.h`

LeapQ

- This is where we update the gauge fields:

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

- For QCD, the momenta are in the LieAlgebra su(3). We need to
  - exponentiate them into the group:

$$P = e^{i\delta\tau p}$$

  - then "add" them to the "q" with SU(3) group addition (matrix multiplication):

$$U \leftarrow U \bigoplus P = UP$$

An exact way to exponentiate su(3) elements

- Cayley – Hamilton:
  - For a traceless antihermitian 3x3 matrix

$$e^{iQ} = f_1 I + f_2 Q + f_3 Q^2$$

- In the eigenbasis of Q:

$$Q = M \Lambda_Q M^{-1} \quad \Lambda_Q = \begin{bmatrix} q_1 & 0 & 0 \\ 0 & q_2 & 0 \\ 0 & 0 & q_3 \end{bmatrix}$$

- The coefficients f_i are the solutions of:

$$\begin{bmatrix} 1 & q_1 & q_1^2 \\ 1 & q_2 & q_2^2 \\ 1 & q_3 & q_3^2 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} e^{iq_1} \\ e^{iq_2} \\ e^{iq_3} \end{bmatrix}$$

su(3) exponentiation

- The system of equations can be solved in various ways:
    - Our implementation follows hep-lat/0311018 by Morningstar and Peardon
  - The code is in **lib/expmat.[h,cc]**
  - The routine is

```
void expmat(LatticeColorMatrix & iQ)
```

- While examining the code is instructive, it is too long a distraction here... see the paper and the code together. The file is quite short < 100 lines.

Leap back to leapQ

- With a matrix exponentiator thus handy, the code for leapQ is quite straightforward:

```
void leapQ(AbsFieldState<GaugeP,GaugeQ>& s, Real dt) const {

  LatticeColorMatrix tmp_1;
  LatticeColorMatrix tmp_2;

  for(int mu = 0; mu < Nd; mu++) {
    tmp_1 = dt*(s.getP())[mu];       // Exponentiation.
    Example::expmat(tmp_1);

    tmp_2 = tmp_1*(s.getQ())[mu];    // Group addition
    (s.getQ())[mu] = tmp_2;

    // Reunitarize u[mu]
    Example::reunit((s.getQ())[mu]);
  }
}
```

lib/qcd_leapfrog.h

Now for the HMC

- Essentially the HMC for QCD turns out to be mostly just a collector for the Hamiltonian, integrator and the trajectory length:

```
class QCDHMCTrj : public AbsHMCTrj<GaugeP,GaugeQ> {
  public:
    ~QCDHMCTrj() {};
    QCDHMCTrj(Handle< AbsHamiltonian<GaugeP,GaugeQ> > H_,
              Handle< AbsIntegrator<GaugeP,GaugeQ> > integrator_,
              const Real& MD_traj_length_) :
                H(H_),  the_integrator(integrator_),
                MD_traj_length(MD_traj_length_) {}
  protected:
    // fulfill obligations here
  private:
    Handle< AbsHamiltonian<GaugeP,GaugeQ> > H;
    Handle< AbsIntegrator<GaugeP,GaugeQ> > the_integrator;
    Real MD_traj_length;
  };
```

`lib/qcd_hmc.h`

Refreshing Momenta

- We must supply a routine to refresh our momenta
  - Our momenta have too large a variance for our SU(3) generators. To match them up we must multiply the momenta by $\sqrt{\frac{1}{2}}$

```
void refreshP(AbsFieldState<GaugeP,GaugeQ>& state) const {
   for(int mu=0; mu < Nd; mu++) {
     gaussian(state.getP()[mu]);                // Fill with noise
     state.getP()[mu] *= sqrt(Real(0.5));  // normalisation
     Example::taproj(state.getP()[mu]);
   }
}
```

Accept or Reject?

- We want to reuse our Accept/Reject test in several HMC classes (eg in SHO). So we isolate it in its own files:
  - lib/global_metropolis_accrej.[h,cc]

```
bool globalMetropolisAcceptReject(const Double& DeltaH)
{
  bool ret_val;
  if ( toBool( DeltaH <= Double(0)) ) {
    ret_val = true;
  }
  else {
    Double AccProb = exp(-DeltaH);
    Double uni_dev; random(uni_dev);

    if( toBool( uni_dev <= AccProb ) ) { ret_val = true; }
    else { ret_val = false;}
  }
  return ret_val;
}
```

If dH <=0 then always accept

Get uniform deviate pseudo random number

accept if random number is less than acceptance probability

Accept/Reject

- With this small factoring in place, supplying the accept reject function for QCDHMCTrj is very simple:

```
bool acceptReject(const Double& DeltaH) const {
    globalMetropolisAcceptReject(DeltaH);
  }
```

- And our HMC is done except for the Monomials...

The Wilson Gauge Monomial

- We need constructor, destructor, S() and Force Term:
- Our declarations are in lib/wilson_gauge_monomial.h:

```
class WilsonGaugeMonomial : public AbsMonomial<GaugeP,GaugeQ> {
public:
  ~WilsonGaugeMonomial() {}
  WilsonGaugeMonomial(const Real& beta_) : beta(beta_) {}

  //! Compute dsdq for the system... Not specified how to actually do this
  void dsdq(GaugeP& F, const GaugeQ& q) const;

  //! Compute the total action
  Double S(const AbsFieldState<GaugeP,GaugeQ>& s) const;

  //! Refresh pseudofermion fields if any
  void refreshInternalFields(const AbsFieldState<GaugeP,GaugeQ>& s)  {}
private:
  Real beta;
};
```

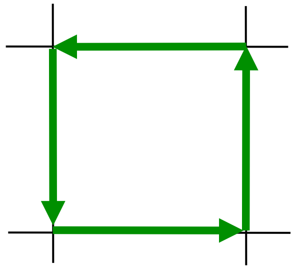
lib/wilson_gauge_monomial.h

The Wilson Gauge Action

- The action (lib/wilson_gauge_monomial.cc) is just our plaquette routine from the first exercise, multiplied by: $\frac{\beta}{N_c}$

```
Double WilsonGaugeMonomial::S(const AbsFieldState<GaugeP,GaugeQ>& s) const
{
  Double S = zero;
  const GaugeQ& u = s.getQ();

  for(int mu=1; mu < Nd; ++mu) {
    for(int nu=0; nu < mu; ++nu) {
      S += sum(real(trace(u[mu]
                          *shift(u[nu],FORWARD,mu)
                          *adj(shift(u[mu],FORWARD,nu))
                          *adj(u[nu]))));
    }
  }
  S *= Double(-beta)/Double(Nc);
  return S;
}
```



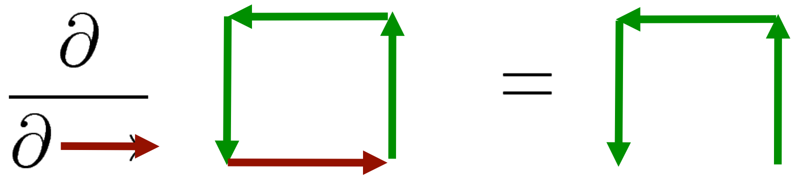
`lib/wilson_gauge_monomial.cc`

Wilson Gauge Force

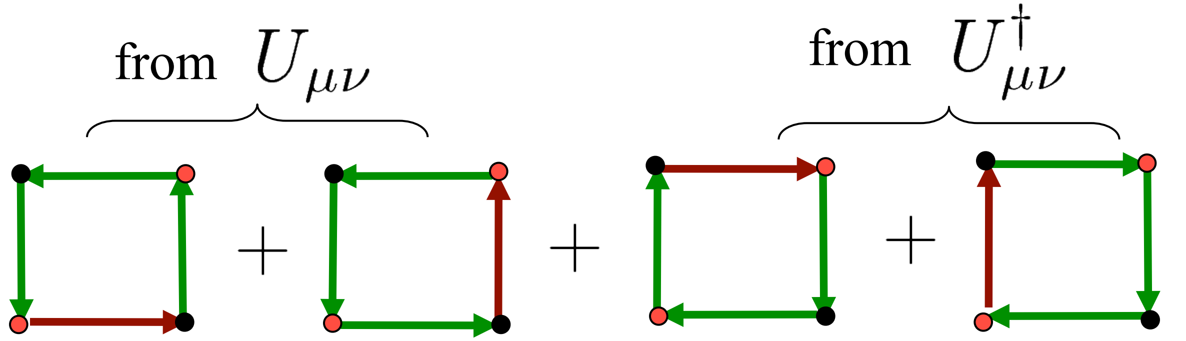
- Using the fact that $\dfrac{\partial U_\mu}{\partial U_\mu} = 1$

- For a given U_μ in a plaquette

$$\boxed{\mathrm{ReTr}\ U_{\mu\nu} = \frac{1}{2}\mathrm{Tr}\ \left[U_{\mu\nu} + U^\dagger_{\mu\nu}\right]}$$



- A plaquette then gives the following force contributions to the links it contains:



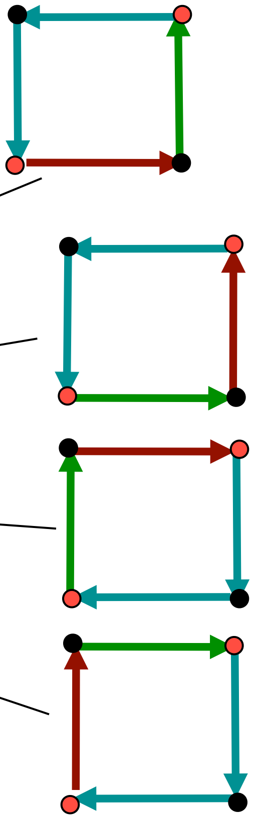
$+$ hermitian conjugate from variation w.r.t $U^\dagger_\mu$

Wilson Gauge Force

lib/wilson_gauge_monomial.cc

```
void WilsonGaugeMonomial::dsdq(GaugeP& F, const GaugeQ& u) const
 {
   F.resize(Nd);

   LatticeColorMatrix tmp_0; // Temporaries
   F = zero;
   // Cycle through all the plaquettes
   for(int mu = 0; mu < Nd; mu++) {
     for(int nu=mu+1; nu < Nd; nu++) {
       tmp_0 = adj(shift(u[mu], FORWARD,nu))*adj(u[nu]);
       F[mu] += shift(u[nu], FORWARD, mu)*tmp_0;
       F[nu] += shift(tmp_0*u[mu], BACKWARD, mu);
       tmp_0 = adj(shift(u[nu], FORWARD, mu))*adj(u[mu]);
       F[mu] += shift( tmp_0*u[nu], BACKWARD, nu);
       F[nu] += shift(u[mu],FORWARD,nu)*tmp_0;
     }
     tmp_0 =  Real(-beta)/(Real(2*Nc))*F[mu];
     F[mu] = u[mu]*tmp_0;
   }
 }
```



Two flavours of Wilson Fermions

- To simulate the fermion determinant, we use pseudofermions:

$$\det(M^\dagger M) = \int d\phi^\dagger \; d\phi \; e^{-\phi^\dagger (M^\dagger M)^{-1} \phi}$$

- This gives us an action:

$$S = \phi^\dagger \left(M^\dagger M\right)^{-1} \phi$$

- The variation of the action with respect to the gauge fields:

$$\boxed{\frac{\delta S}{\delta U} = -\phi^\dagger \left(M^\dagger M\right)^{-1} \left[\frac{\delta M^\dagger}{\delta U} M + M^\dagger \frac{\delta M}{\delta U}\right] \left(M^\dagger M\right)^{-1} \phi}$$

- We define, for later convenience

$$\boxed{X = \left(M^\dagger M\right)^{-1} \phi, \quad Y = MX} \qquad S = \langle \phi, X \rangle$$

The Wilson Fermion Monomial

- Much like the other monomials but:
  - Monomial will now store pseudofermion fields (phi)
    - Our **`refreshInternalFields()`** method will not be empty
  - We will add a getX() function to compute X
    - This needs to solve

$$\boxed{\left(M^\dagger M\right) X = \phi}$$

    - so we will need a CG Solver
    - and we will need to store its parameters.
- We will need to modify our LinearOperator to allow us to compute:

$$\boxed{X^\dagger \frac{\delta M^\dagger}{\delta U} Y} \qquad \boxed{Y^\dagger \frac{\delta M}{\delta U} X}$$

The Easy Bits First

```
class TwoFlavorWilsonFermMonomial : public AbsMonomial<GaugeP,GaugeQ> {
public:
  ~TwoFlavorWilsonFermMonomial() {}
  TwoFlavorWilsonFermMonomial(const Real& Mass_,
                              const Real& RsdCG_,
                              int MaxCG_
                              ) : Mass(Mass_), RsdCG(RsdCG_), MaxCG(MaxCG_) {}

  void dsdq(GaugeP& F, const GaugeQ& q) const;

  Double S(const AbsFieldState<GaugeP,GaugeQ>& s) const;

  //! Refresh pseudofermions
  void refreshInternalFields(const AbsFieldState<GaugeP,GaugeQ>& s) ;
private:
  void getX(LatticeDiracFermion& X, const GaugeQ& u) const;
  Real Mass;
  Real RsdCG;
  int MaxCG;

  LatticeDiracFermion phi;
};
```

Do our solve, and get X for us.

Our Pseudofermion

lib/wilson_ferm_two_flavor_monomial.h

Fresh Fields.

$$e^{-\phi^\dagger (M^\dagger M)^{-1} \phi} = \boxed{e^{-\eta^\dagger \eta}} \longleftarrow$$

Gaussian with variance 1/2

Transformation: $\boxed{\Rightarrow \phi = M^\dagger \eta}$

```
void
TwoFlavorWilsonFermMonomial::refreshInternalFields(
                    const AbsFieldState<GaugeP,GaugeQ>& s)  {
  const GaugeQ& u=s.getQ();
  UnprecWilsonLinOp M(u, Mass);

  LatticeDiracFermion eta;
  gaussian(eta);
  eta *= sqrt(0.5);

  M(phi, eta, -1);
}
```

fill with noise and reset width

do the transformation (-1 => dagger)

lib/wilson_ferm_two_flavor_monomial.cc

Getting X

- This is a simple matter of invoking your solver. Should be familiar from session 2 exercises (you'll need your CG solver)

```
void TwoFlavorWilsonFermMonomial::getX(LatticeFermion& X, const GaugeQ& u) const
{

   UnprecWilsonLinOp M(u, Mass);
   Real RsdCGOut;
   int n_count;
   InvCG(M,
         phi,
         X,
         RsdCG,
         MaxCG,
         RsdCGOut,
         n_count);

 }
```

Just solve:

$$\boxed{\left(M^\dagger M\right) X = \phi}$$

with Conjugate Gradients

$$M^\dagger M$$

is manifestly Hermitian & positive definite

`lib/wilson_ferm_two_flavor_monomial.cc`

Computing S

- Once we have X, computing the action is easy since:

$$\phi\left(M^{\dagger}M\right)^{-1}\phi = \langle \phi, X\rangle$$

- The code is straightforward:

```
//! Compute the total action
  Double TwoFlavorWilsonFermMonomial::S(const AbsFieldState<GaugeP,GaugeQ>& s)
const {
    const GaugeQ& u=s.getQ();
    LatticeFermion X=zero;
    getX(X,u);
    Double result=real(innerProduct(phi,X));
    return result;
  }
```

lib/wilson_ferm_two_flavor_monomial.cc

Computing the force

- We need X, and $X^\dagger \frac{\delta M^\dagger}{\delta U} Y$
- We will delegate the matrix derivative to our linear operator
  - Will allow us to generalise our Wilson Monomial to any two flavour monomial.
- We *extend* our **LinearOperator** class to a new class

  DiffLinearOperator

  - This class can evaluate our derivative with a new function:

```
void deriv(P& F,  const T& X, const T& Y, int isign)
```

  - The **isign** decides whether we do the derivative of M or its conjugate (+1 or -1 respectively)

The Extended Linear Operator Class

```
template<typename P, typename T>
class DiffLinearOperator : public LinearOperator<T> {
public:
   virtual ~DiffLinearOperator() {}

   // Make sure derived classes can override the operator() method
   virtual void operator()(T& result, const T& source, int isign)
        const = 0;

   // Make sure derived classes can override the subset method
   // The subset on which the lattice acts
   virtual const Subset& subset() const = 0;

   // Now a derivative term of the form X^\dagger \dot(M) Y
   virtual void deriv(P& F, const T& X, const T& Y, int isign) const = 0;
};
```

`lib/linop_class.h`

The Derivative Of M

Since $M = (N_d + M) - \frac{1}{2}D \quad \Rightarrow \quad \frac{\delta M}{\delta U_\mu} = -\frac{1}{2}\frac{\delta D}{\delta U_\mu}$

Recall that:

$$D_{x,y} = \sum_\mu \left[(1-\gamma_\mu)U_{x,\mu}\delta_{x+\hat{\mu},y} + (1+\gamma_\mu)U^\dagger_{x-\hat{\mu},\mu}\delta_{x-\hat{\mu},y}\right]$$

So we have: $$\frac{\delta D}{\delta U_\mu} = (1-\gamma_\mu)\,\delta_{x+\hat{\mu},y}$$

And thus:

This is just a trace identity

$$\boxed{X\frac{\delta D}{\delta U_\mu}Y = X^\dagger(1-\gamma_\mu)Y_{x+\hat{\mu}} = \mathrm{Tr}_s\left[(1-\gamma_\mu)Y_{x+\hat{\mu}} \otimes X^\dagger\right]}$$

Implementation

- We add a derivative routine to **`dslashm_w.cc`**

```
void dslash_deriv( multi1d<LatticeColorMatrix>& F,
                   const LatticeDiracFermion& X,
                   const LatticeDiracFermion& Y,
                   int isign, int cb)
{
  F.resize(Nd);
  for(int mu = 0; mu < Nd; ++mu) {
   LatticeDiracFermion temp_ferm1;
    LatticeHalfFermion tmp_h;

    switch (isign) {
    case 1:
      // Undaggered: Minus Projectors
      {

        switch(mu) {
        case 0:
          tmp_h[rb[1-cb]] = spinProjectDir0Minus(Y);
          temp_ferm1[rb[1-cb]] = spinReconstructDir0Minus(tmp_h);
          break;
          ...   // other mu values and isign
```

Evaluate
temp_ferm= $(1 - \gamma_\mu)$ Y
like in session2 with
projector/reconstructor

Now shift and trace

```
    ...
    LatticeDiracFermion temp_ferm2 = shift(temp_ferm1, FORWARD, mu);


    // This step supposedly optimised in QDP++
    F[mu][rb[cb]] = traceSpin(outerProduct(temp_ferm2,X));
    F[mu][rb[1-cb]] = zero;
  }
}
```

$$(1-\gamma_\mu)Y_{x+\hat{\mu}}$$

QDP++ supplies traceSpin() & outerProduct()

$$X\frac{\delta D}{\delta U_\mu}Y = X^\dagger(1-\gamma_\mu)Y_{x+\hat{\mu}} = \mathrm{Tr}_s\left[(1-\gamma_\mu)Y_{x+\hat{\mu}}\otimes X^\dagger\right]$$

lib/dslashm_w.cc

Now back to the Unprec Wilson LinOp

```
void
UnprecWilsonLinOp::deriv(multi1d<LatticeColorMatrix>& F,
                         const LatticeDiracFermion& X,
                         const LatticeDiracFermion& Y,
                         int isign) const
{

    // Dslash Derivatives
  F.resize(Nd);
  for(int mu=0; mu < Nd; mu++) { F[mu]=zero; }

  multi1d<LatticeColorMatrix> F_tmp(Nd);
  dslash_deriv(F, X, Y, isign, 0);
  dslash_deriv(F_tmp, Y, X, isign, 1);
  F += F_tmp;

  for(int mu = 0; mu < Nd; ++mu) {
    F[mu] *= Real(-0.5);
  }
}
```

lib/unprec_wilson_w.cc

And back to the monomial force:

```
void TwoFlavorWilsonFermMonomial::dsdq(GaugeP& F, const GaugeQ& u) const
 {
   UnprecWilsonLinOp M(u,Mass);
   LatticeDiracFermion X,Y;

   getX(X,u);  // (M^\dag M) X = \phi
   M(Y,X,1);   //   Y = M X

   GaugeP F_tmp;
   M.deriv(F_tmp, X, Y, -1);
   M.deriv(F, Y, X, +1);
   for(int mu=0; mu < Nd; mu++) {
     F_tmp[mu] += F[mu];
     F_tmp[mu] *= Real(-1);
   }
   // Now multiply by U
   for(int mu=0; mu < F.size(); ++mu) {
     F[mu] = u[mu]*F_tmp[mu];
   }
 }
```

$$X^\dagger \frac{\delta M^\dagger}{\delta U} Y$$

$$Y^\dagger \frac{\delta M}{\delta U} X$$

Accumulate
add - sign

$$U_\mu \frac{\delta S}{\delta U_\mu}$$

lib/wilson_ferm_two_flavor_monomial.cc

and we are done

- all that is needed now is a driver for main() - see next slides
- Recap:
  - We defined abstract class structure needed for HMC:
    - field state, integrator, hamiltonian, monomials, HMC
    - these classes provided interface functions and default behaviour
  - We presented concrete implementations
    - GaugeFieldState, QCDHamiltonian, QCDLeapfrog, QCDHMCTraj
    - Presented Gauge and Fermion Monomials
- Additional Exercises and background material follow:
  - Details of main() to set up the classes for use
  - Omelyan's integrator
  - Even-Odd (red-black) preconditioning in HMC.

Highlights of the Driver

- All we need is a main program to drive it all
  - example3/qcd.cc
- Highlights: Starting up the state

```
Seed seed = 27;
RNG::setrn(seed);

for(int mu=0; mu < Nd; mu++)
  gaussian(initial_q[mu]);
  reunit(initial_q[mu]);

  gaussian(initial_p[mu]);
  initial_p[mu] *= sqrt(Real(0.5));
  taproj(initial_p[mu]);
}

// Create a field
GaugeFieldState s(initial_p, initial_q
```

Reseed RNG

Usual Disordered Start

A momentum refresh...

Create State

One main() to drive it all...

- Setting up the Monomials and Hamiltonian & Integrator

HMC params

```
Real beta=Real(5.4);                // Gauge Coupling
Real Mass=0.02;                     // Quark Mass
int MaxCG=500;                      // Max no of solver iterations
Real RsdCG=Real(1.0e-8);            // Desired Solver Tolerance
int n_steps = 16;                   // No of steps over a trajectory
Real traj_length=1;                 // Length of the MD trajectory

// Create a monomial list of 2 terms.
multi1d< Handle< AbsMonomial<GaugeP,GaugeQ> > > monomials(2);

monomials[0] =new WilsonGaugeMonomial(beta);

monomials[1] = new TwoFlavorWilsonFermMonomial(Mass, RsdCG, MaxCG);

// Group Monomials into a Hamiltonian
Handle<AbsHamiltonian<GaugeP,GaugeQ> > H(new QCDHamiltonian(monomials));
Handle<AbsIntegrator<GaugeP,GaugeQ> > integrator(new QCDLeapfrog( *H,n_steps ));
```

Handles to Abstact classes

Dynamically allocate concrete instances

Setting Up and Running the HMC

Create HMC function object

```
QCDHMCTrj hmc( H, integrator, traj_length );

for(int i=0; i < 1000; i++) {
  hmc(s, false);

  Double plaquette; Example::MeasPlq(s.getQ(), plaquette);
  QDPIO::cout << "i=" << i << " Plaquette= " << plaquette << endl;
}
```

Measure something

Exercise: Omelyan's integrator

- De Forcrand and Takaishi suggest the use of an Omelyan Integrator in Phys. Rev. E73(2006) 036706 hep-lat/0505020
- Algorithm (per timestep dt):

```
leapQ( λ dt )
leapP( dt / 2)
leapQ( (1 – 2 λ) dt )
leapP ( dt / 2)
leapQ( λdt )
```

- Write an abstract class for this integrator following leapfrog as an example. Write a QCD Implementation. Use in qcd.cc instead of leapfrog

HMC And Even Odd Preconditioning

- Remember even Odd Preconditioning from Lecture 2?

$$\begin{aligned} M &= \begin{bmatrix} M_{ee} & M_{eo} \\ M_{oe} & M_{oo} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ M_{oe}M_{ee}^{-1} & 1 \end{bmatrix} \begin{bmatrix} M_{ee} & 0 \\ 0 & M_{oo} - M_{oe}M_{ee}^{-1}M_{eo} \end{bmatrix} \begin{bmatrix} 1 & M_{ee}^{-1}M_{eo} \\ 0 & 1 \end{bmatrix} \\ &= L\tilde{M}U \end{aligned}$$

- How does this impact HMC?
  - Way 1: Preconditioning system reduces cost of solving

$$\left(M^{\dagger}M\right)X = \phi$$

  - Way 2: By reformulating our Hamiltonian in term of
    - Can reduce solver costs AND MD Force

HMC and Even Odd Preconditioning

- Recall that fundamentally we are trying to simulate the determinant by our pseudofermion games:

$$\det(M^\dagger M) = \int d\phi^\dagger \; d\phi \; e^{-\phi^\dagger \left(M^\dagger M\right)^{-1} \phi}$$

- With preconditioning we can play determinant games:

$$\begin{aligned}
\det\left(M^\dagger M\right) &= \det\left(\left[U^\dagger \tilde{M}^\dagger L^\dagger\right] \; \left[L \tilde{M} U\right]\right) \\
&= \det\left(\tilde{M}^\dagger \tilde{M}\right) \qquad \text{since}\, \det(L) = \det(U) = 1 \\
&= \det\left(\tilde{M}^\dagger_{ee} \tilde{M}_{ee}\right) \; \det\left(\tilde{M}^\dagger_{oo} \tilde{M}_{oo}\right)
\end{aligned}$$

HMC and Even-Odd Preconditioning

- For Wilson Fermions $\tilde{M}_{ee} = 1$ and so:

$$\begin{aligned}\det\left(M^\dagger M\right) &= \det\left(\tilde{M}^\dagger_{oo}\tilde{M}_{oo}\right) \\ &= \int d\phi^\dagger_o\, d\phi_o\; e^{-\phi^\dagger_o\left(\tilde{M}^\dagger_{oo}\tilde{M}_{oo}\right)^{-1}\phi_o}\end{aligned}$$

- NB: This is not true for all fermions. Some have $\tilde{M}_{ee} \neq 1$
  - In this case we must deal with $\tilde{M}_{ee}$
  - This can perhaps be done explicitly (eg: in Clover Fermions)

$$\det\left(\tilde{M}^\dagger_{ee}\tilde{M}_{ee}\right) = e^{\ln\det\left(\tilde{M}^\dagger_{ee}\tilde{M}_{ee}\right)} = e^{\mathrm{Tr}\ln\left(\tilde{M}^\dagger_{ee}\tilde{M}_{ee}\right)}$$

HMC And Even Odd Preconditioning

- Preconditioned Action:

$$S = \phi_o^\dagger \left(\tilde{M}_{oo}^\dagger \tilde{M}_{oo} \right)^{-1} \phi_o - 2 \ \mathrm{Tr} \ln \det \left| \tilde{M}_{ee} \right|$$

- For Wilson Fermions force stays same as before except for:

$$X^\dagger \frac{\delta \tilde{M}}{\delta U} Y = \frac{-1}{4(N_d + M)} X^\dagger \frac{\delta}{\delta U} \left[D_{oe} D_{eo} \right] Y$$

$$\begin{aligned} X^\dagger \frac{\delta}{\delta U} \left[D_{oe} D_{eo} \right] Y &= X^\dagger \frac{\delta D_{oe}}{\delta U} \boxed{D_{eo} Y} + \boxed{X^\dagger D_{oe}} \frac{\delta D_{eo}}{\delta U} Y \\ &= X^\dagger \frac{\delta D_{oe}}{\delta U} \boxed{\tilde{Y}} + \boxed{\tilde{X}^\dagger} \frac{\delta D_{oe}}{\delta U} Y \end{aligned}$$

- NOTE: Force still acts on ALL of the lattice

HMC And Preconditioning: Key Points

- Preconditioning can be done in 2 ways
  - Way 1: as a trick to speed up the solver
  - Way 2: it can be used to rewrite
    - The Action/Hamiltonian
    - The Force Terms

  in terms of the preconditioned matrices
  - The magnitude of forces varies with the condition number of the matrices in the force term (ie Way 2).
    - Better conditioned matrices => Smaller forces
    - Smaller forces => One can take LONGER steps
    - => Multiple time scale integrators and most recent HMC algorithmic tricks...

Advanced Exercises

- Extend the Even-Odd Preconditioned Linear Operator from Session 2, with a derivative function()
  - To be completely general you'll need a derivative for both the even-even, even-odd, odd-even and odd-odd parts
  - You can then code the full deriv() as a default in terms of these functions
- Extend the Even-Odd Wilson Operator with a derivative function
  - Because your even-even term is trivial you may wish to override the derivative in the base class you've just written

Advanced Exercises

- Code a Monomial for 2 flavours of Even Odd Wilson Fermions.
  - field refreshment over just the odd subset now
  - Use the subset in the inner product for the action
  - force should not change, except for the kind of matrix you use.
- Replace the unpreconditioned Wilson monomial with your new preconditioned one in the qcd.cc code
- Without changing anything else run the HMC code
  - What happens to your iteration counts?
  - What happens to your acceptance rate