Lecture 4: More samplings of HMC

Bálint Joó
Jefferson Lab

INT Summer School – Lattice QCD and its Applications

http://www.jlab.org/~bjoo/Lecture4.pdf

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

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

6) Compute energy change:

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

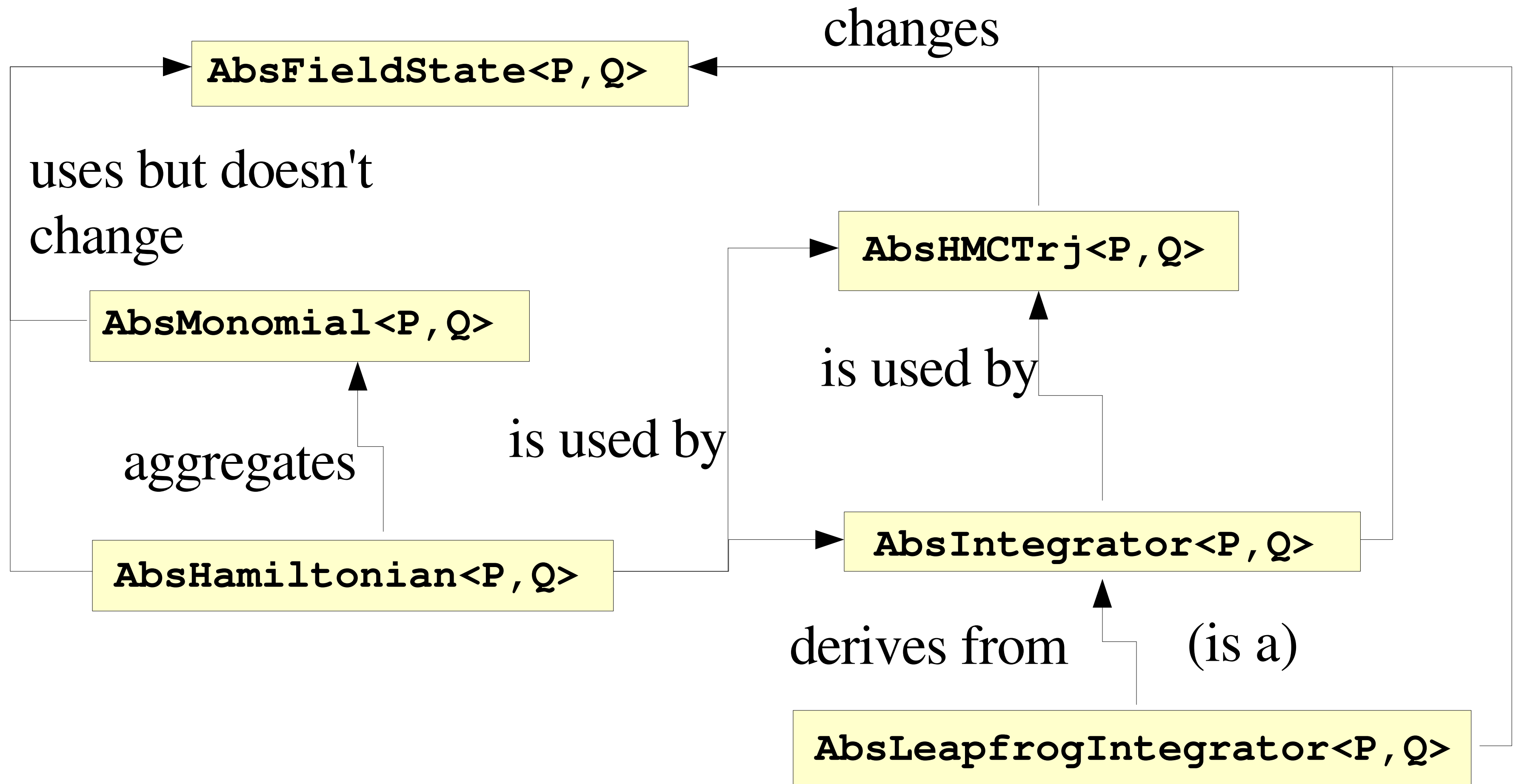
7) Accept/ Reject (p',q') with probability:

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

8) In case of rejection the new state is (p,q)
9) Go to step 1

Recap from Last Lecture

- We created an abstract class structure for HMC:



AbsFieldState<P,Q>

- Packages our coordinates and momenta
- Defines abstract accessors/manipulators to the data:
  - getP() & getQ()   functions
    - both read only and read/write
- Defines abstract clone() function
  - so we can copy derived classes

AbsMonomial<P,Q>

- Responsible for
  - Computing a piece of action on a state:
    - Defines abstract function S():
  - Computing the Molecular Dynamics Force on a state:
    - Defines abstract function dsdq()
  - Refreshing any internal fields, given a state:
    - Defines abstract function refreshInternalFields()

AbsHamiltonian<P,Q>

- Aggregates Monomials for us and deals with momenta
- Responsible for:
  - Computing Energies from Momenta and Monomials
  - Computing Forces due to Monomials
- Defines abstract functions
  - numMonomials()
  - getMonomial() (read only & read/write)
  - mesKE(), mesPE(), mesE()
  - dsdq(), refreshInternalFields()
- Provides default implementations for:
  - mesKE(), mesPE(), mesE()
  - dsdq(), refreshInternalFields()

AbsIntegrator<P,Q>

- Responsible for integrating a state for a given trajectory length with Molecular Dynamics
- Defines abstract function:
  - operator() - do the integration

AbsLeapfrogIntegrator<P,Q>

- Is an AbsIntegrator<P,Q> that performs the leapfrog algorithm for the MD integration
- Defines extra abstract functions (apart from operator())
  - getNumSteps()
  - leapP()
  - leapQ()
- Provides a default implementation for
  - operator() (encodes the leapfrog algorithm)

AbsHMCTrj<P,Q>

- Responsible for doing the HMC
- Defines abstract functions:
  - refreshP()
  - acceptReject()
  - getMCHamiltonian()
  - getMDIntegrator()
  - getMDTrajLength()
- Provides an implemened function:
  - operator() - Perform the HMC algorithm

We need to Implement in derived classes

- AbsFieldState<P,Q>
  - getP() (r & r/w)
  - getQ() (r & r/w)
  - clone()
- AbsMonomial<P,Q>
  - S()
  - dsdq()
  - refreshInternalFields()
- AbsHamiltonian<P,Q>
  - numMonomials()
  - getMonomial() (r&r/w)
- AbsLeapfrogIntegrator<P,Q>
  - getNumSteps()
  - leapP()
  - leapQ()
- AbsHMCTrj<P,Q>
  - refreshP()
  - acceptReject()
  - getMCHamiltonian()
  - getMDIntegrator()
  - getMDTrajLength()
- C++ Salad (class bodies, data members, etc)

legend: standard (C++) Physics

The State of QCD

- 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:

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
  };
```

Encapsulated data

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);
  for(int mu=0; mu < Nd; mu++) {
    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
```

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

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

Fulfilling the Interface

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

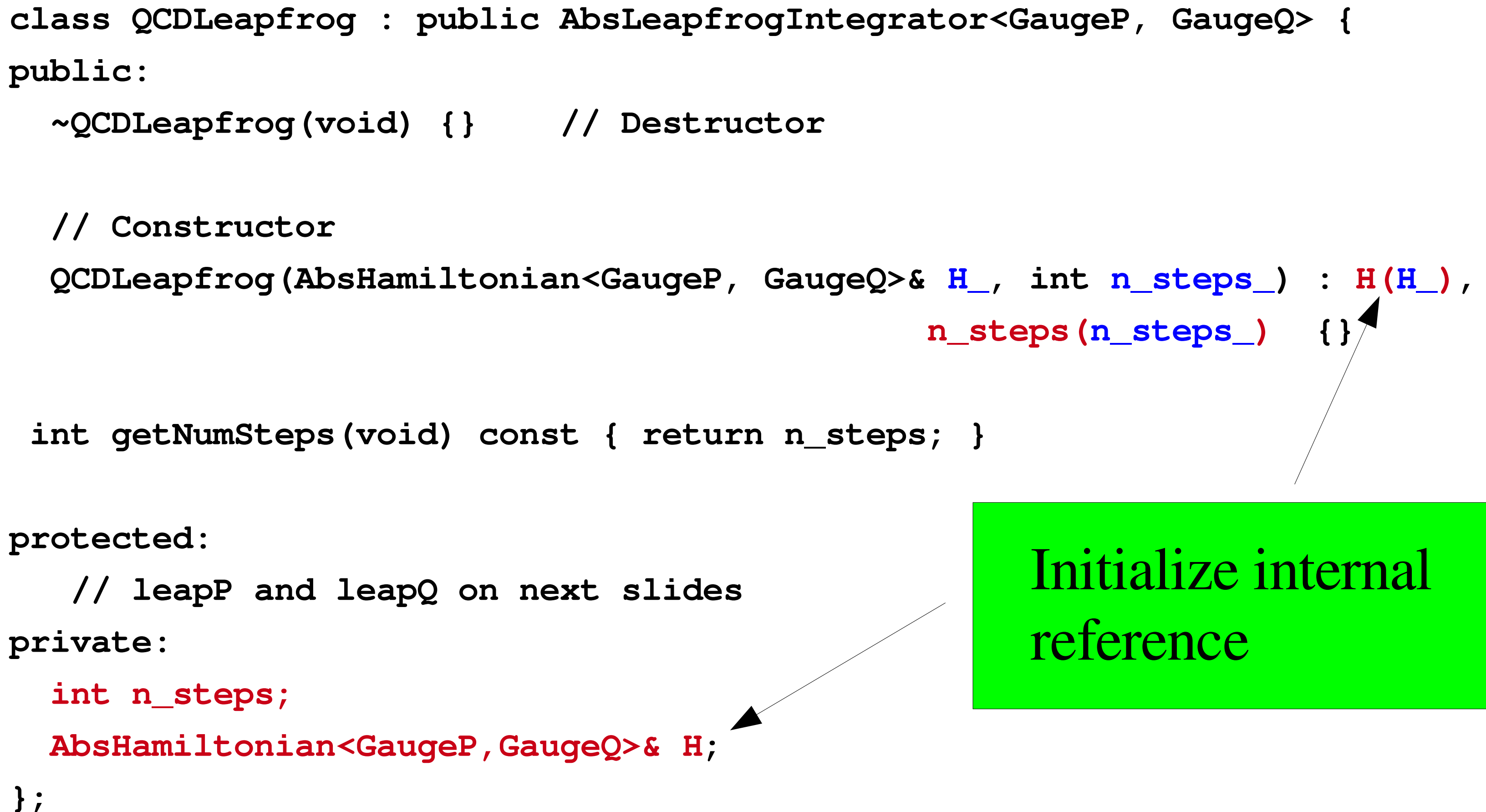
```
int numMonomials(void) const {
  return monomials.size();
}

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;
};
```

LeapP

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

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

Now we see why F is same type as p

- 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*F
       // 1) project the force
       Example::taproj( F[mu] );

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

     }
   }
```

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):

$$U \leftarrow U \oplus P = UP$$

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]);
  }
}
```

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;
  };
```

Accessors/Manipulators

- We need to get at the encapsulated Integrator, Hamiltonian and trajectory length:

```
// Get at the Exact Hamiltonian
AbsHamiltonian<GaugeP,GaugeQ>& getMCHamiltonian(void) { return *H; }

// Get at the Integrator
AbsIntegrator<GaugeP,GaugeQ>& getMDIntegrator(void) {
  return *the_integrator;
}

// Get at the trajectory length
Real getMDTrajLength(void) const {
  return MD_traj_length;
}
```

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]);   // Project back into algebra
   }
}
```

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:
  - 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 <= the acceptance proability

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

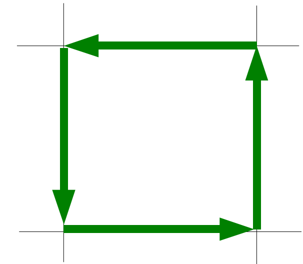
- We need constructor, destructor, S() and Force Term:
- Our declarations are in lib/wilson_gauge_monomial.h:

```
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;
};
```

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;
}
```



No normalization by volume & no of planes

Wilson Gauge Force

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

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

- For a given Uμ in a plaquette

$$\frac{\partial}{\partial \rightarrow} \square = \sqcap$$

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

from $U_{\mu\nu}$ from $U^\dagger_{\mu\nu}$

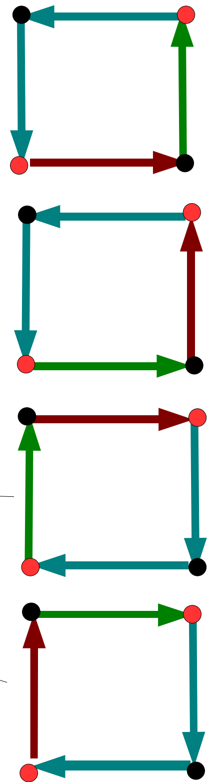
$+ \quad + \quad +$

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

Wilson Gauge Force

```
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;
   }
 }
```

tmp_0 = blue arrows



Two flavours of Wilson Fermions

$S = \langle \phi, X \rangle$

- 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:

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

- We define, for later convenience

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

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

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

    - 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:

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

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;
};
```

For Fermion

For Solver

Do our solve, and get X for us.

Our Pseudofermion

Fresh Fields.

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

Gaussian with variance 1/2

Transformation: $\Rightarrow \phi = M^\dagger \eta$

```
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

effect the transformation: -1 => dagger

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:

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

with Conjugate Gradients

$$M^{\dagger}M$$

is manifestly Hermitian & positive definite

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;
  }
```

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.

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;
};
```

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

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

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);
    LatticeColorMatrix temp_mat;

    // 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()

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

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);
  }
}
```

Call dslash_deriv()
on each
checkerboard

Multiply by -1/2
prefactor for Dslash

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

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

And We Are Done!

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

```
// Hot start for the gauge field
 // Fill momenta with Traceless Antihermitian Projected gaussian noise
 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);
```

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 1 term.
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;
}
```

Do 1 HMC update

Measure something

Summary

- Good Class Design can help us a lot:
  - Clear Structure & Class responsibilities
  - Sensible defaults (for HMC etc.)
- Thanks to our design to code a new HMC all we need to write are the:
  - Field State "client"
    - Hamiltonian "client"
    - leapP and leapQ for the system (leapfrog client)
    - an HMC "client"
  - Monomials (Force Term + Energy)
- Our HMC in total takes < 2700 lines (including Makefiles & Simple Harmonic Oscillator Classes too)
- Writing an HMC is NOT MAGIC! It is in fact relatively simple.

By "client" I mean that primarily construction and access functions need to be implemented

May not even need these if FieldState is unchanged

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:

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

- 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:

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

- 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:

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

- 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