Dances With Solvers

Bálint Joó
Scientific Computing Group
Jefferson Lab

Goals

- Basic:
  - Meet the Wilson Dslash operator
  - Apply the Unpreconditioned Wilson Fermion Matrix,
  - Invert with MR solver
  - Compute a propagator
- More Advanced:
  - A bit of Object Orientation: Encapsulating Linear Ops.
  - Rewrite MR solver to make it more generic
  - Even Odd Preconditioning

The Quark Propagator

- To find the quark propagator we need:

$$\langle \bar{\psi}_a^\alpha(y)\psi_b^\beta(x)\rangle = \left[M^{-1}(x,y)\right]_{a,b}^{\alpha,\beta} = G_{a,b}^{\alpha,\beta}(x,y)$$

- We can use use translation invariance:

$$\langle \bar{\psi}_a^\alpha(x)\psi_b^\beta(0)\rangle = \left[M^{-1}(0,x)\right]_{a,b}^{\alpha,\beta} = G_{a,b}^{\alpha,\beta}(0,x)$$

N_{spin}xN_{color} "Sink" spin, color components

"Source" spin,color

- So we need to perform N_{spin} x N_{color} solutions of the system:

$$M_{a,b}^{\alpha,\beta}(0,x)\psi_a^\alpha(x) = \chi_b^\beta(0)$$

The Wilson Fermion Matrix

- Define the matrix as:

$$M_{x,y} = (N_d + M)\delta_{x,y} - \frac{1}{2}D_{x,y}$$

sometimes re-scaled as:

$$M_{x,y} = \delta_{x,y} - \kappa D_{x,y}$$

with

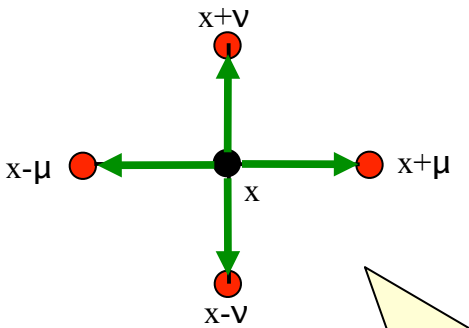
$$\kappa = \frac{1}{2(N_d + M)}$$

- D is the Wilson Dslash Term:

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

The Dslash Term

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



- Numerically Expensive: 1392 Flops/site
- Gauge Covariant Derivative
- $(1 \pm \gamma_\mu)$ are projectors (use this later)
- Can employ even-odd (red-black) preconditioning:

Note: "black sites" need information only from "red" sites in red/ black checkerboarding

$$D = \begin{bmatrix} 0 & D_{eo} \\ D_{oe} & 0 \end{bmatrix}$$

- The Dslash has a γ_5 hermiticity: $D^{\dagger} = \gamma_5 D \gamma_5$

The Spin Basis in QDP++

- QDP++ uses the DeGrand-Rossi Basis (same as MILC/CPS)

$$\gamma_0 = \begin{pmatrix} 0 & i\sigma^1 \\ -i\sigma^1 & 0 \end{pmatrix} = -\sigma^2 \otimes \sigma^1$$

$$\gamma_1 = \begin{pmatrix} 0 & -i\sigma^2 \\ i\sigma^2 & 0 \end{pmatrix} = \sigma^2 \otimes \sigma^2$$

$$\gamma_2 = \begin{pmatrix} 0 & i\sigma^3 \\ -i\sigma^3 & 0 \end{pmatrix} = -\sigma^2 \otimes \sigma^3$$

$$\gamma_3 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \sigma^1 \otimes 1$$

- This is a chiral basis: $\gamma_5 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \sigma^3 \otimes 1$

Gamma Matrices in QDP++

- We use the QDP++ function **`Gamma(N)`**
- `N` is a 4 bit long binary number: N = a b c d (binary)
- Then

$$\texttt{Gamma}(N) = \gamma_0^d \ \gamma_1^c \ \gamma_2^b \ \gamma_3^a$$

- So

$$\texttt{Gamma}(1) \Rightarrow \texttt{Gamma}(\texttt{0b0001}) \Rightarrow \gamma_0$$
$$\texttt{Gamma}(2) \Rightarrow \texttt{Gamma}(\texttt{0b0010}) \Rightarrow \gamma_1$$
$$\texttt{Gamma}(4) \Rightarrow \texttt{Gamma}(\texttt{0b0100}) \Rightarrow \gamma_2$$
$$\texttt{Gamma}(8) \Rightarrow \texttt{Gamma}(\texttt{0b1000}) \Rightarrow \gamma_3$$
$$\texttt{Gamma}(15) \Rightarrow \texttt{Gamma}(\texttt{0b1111}) \Rightarrow \gamma_0\gamma_1\gamma_2\gamma_3 = \gamma_5$$
$$\texttt{Gamma}(3) \Rightarrow \texttt{Gamma}(\texttt{0b0011}) \Rightarrow \gamma_0\gamma_1 = -\gamma_1\gamma_0$$

QDP++ Fermions and Expressions

- The Wilson Like Fermion Type: LatticeDiracFermion
- You can fill it, with noise or zero easily

```
LatticeDiracFermion x;
x=zero;
gaussian(x);
```

- You can do arithmetic with other scalars and vectors

```
LatticeDiracFermion x,y,z;
Real a = Real(0.5);
gaussian(x); gaussian(y);
x = a*x + y;
```

- You can multiply by a spin matrix:

```
LatticeDiracFermion x; gaussian(x);
y = Gamma(15)*x;   // Multiply by g_5
```

More Expressions

- You can take a 2-norm:

```
LatticeDiracFermion x;
gaussian(x);
Double y = norm2(x);
```

- You can take an inner product: $\langle \phi, \chi \rangle = \phi_i^\dagger \ \chi_i$

```
LatticeDiracFermion phi,chi;
DComplex prod = innerProduct(psi,chi);
```

- You can multiply by a compliant type e.g. **`LatticeColorMatrix`** (multiply each 3-vector in a spinor on a site by an SU(3) matrix)

```
LatticeFermion phi; gaussian(phi);
LatticeColorMatrix u = Real(1); //Free
LatticeFermion chi = u * x;
```

Spin Projection Trick

- Consider Use of Spin Projection in the Dslash in the 3-direction

$$(1+\gamma_3)U \begin{bmatrix} x \\ y \\ z \\ t \end{bmatrix} = \begin{bmatrix} U(x+z) \\ U(y+t) \\ \hline U(x+z) \\ U(y+t) \end{bmatrix} \begin{matrix} \} \text{ upper half vector} \\ \} \text{ lower half vector} \end{matrix}$$

- Break down: projection – multiplication- reconstruction

$$\begin{bmatrix} x \\ y \\ z \\ t \end{bmatrix} \xrightarrow{SpinProject} \begin{bmatrix} x+z \\ y+t \end{bmatrix} \xrightarrow{Multiply\ U} \begin{bmatrix} U(x+z) \\ U(y+t) \end{bmatrix} \xrightarrow{SpinReconstruct} \begin{bmatrix} U(x+z) \\ U(y+t) \\ \hline U(x+z) \\ U(y+t) \end{bmatrix}$$

- Save two SU(3) multiplications per site

Spin Projection in QDP++

	Projection:	Reconstruction:
$(1-\gamma_\mu)\phi$	`spinProjectDirXMinus(phi)`	`spinReconstructDirXMinus(phi)`
$(1+\gamma_\mu)\phi$	`spinProjectDirXPlus(phi)`	`spinReconstructDirXPlus(phi)`

X is the direction, i.e.: m so one of 0,1,2,3

- The projected Half Fermion has type: LatticeHalfFermion

```
LatticeFermion x;
gaussian(x);
LatticeHalfFermion x_proj = spinProjectDir0Plus(x);
LatticeFermion y = spinReconstructDir0Plus(x_proj);
QDPIO::cout << "|| y - x ||=" << sqrt(norm2(y-x))
            << endl;
```

Checkerboarding

- On a red/black checkerboarded lattice we need the sites on one checkerboard for evaluating the dslash on the other checkerboard.
- In QDP++ we can restrict operations to a "subset" of the lattice (eg. the red sites) by using a subset index.
- Red/black checkerboarding provides 2 subsets
  - the red sites or the black sites.
- In QDP++ we can specify subsets of a lattice using the **[]** operator
  - **`foo[ rb[0] ]`** – on one checkerboard (say "red")
  - **`foo[ rb[1] ]`** – on the other checkerboard (say "black")
- Red/Black checkerboarding is used so often that QDP++ pre-defines it (others include **`all, rb3, cb32`**)
- We'll show how to define custom "subsets" later on.

OK Let's Look at the Code:

```
bash$ cd seattle_tut/example2
```

- Remember to edit the Makefile and change the CONFIG macro to point to your own one (also in lib/Makefile)
- Run 'make' to check things still work

The Dslash Routine

- Look in **examples2/lib/dslashm_w.cc** :

```
switch (isign) {
case 1:
{
  LatticeHalfFermion tmp, tmp2;
  // Dir 0 FORWARD
  tmp[rb[otherCB]]  = spinProjectDir0Minus(psi);
  tmp2[rb[cb]] = shift(tmp, FORWARD, 0);
  chi[rb[cb]] = spinReconstructDir0Minus(u[0]*tmp2);

  // Dir 0 BACKWARD
  tmp[rb[otherCB]]  = adj(u[0])*spinProjectDir0Plus(psi);
  tmp2[rb[cb]] = shift(tmp, BACKWARD, 0);
  chi[rb[cb]] += spinReconstructDir0Plus(tmp2);
...
```

This is a subset index
rb[0] = even
rb[1] = odd

And so forth for all directions. There is also a case for the daggered operator

Unpreconditioned Wilson Op.

- It is now straightforward to apply the Unpreconditioned Wilson Operator (code in: lib/unprec_wilson_w.cc)

```
void M_unprec_wils(LatticeDiracFermion& result,
                   const LatticeDiracFermion& phi,
                   const multi1d<LatticeColorMatrix>& u,
                   int isign,
                   const Real Mass)
{
  Real mass_term=Real(Nd)+Mass;
  Real half = Real(0.5);
  // (Nd + M) phi
  result=mass_term*phi;
  LatticeDiracFermion  tmp;

  // Dslash phi on both checkerboards
  dslash(tmp, u, phi, isign, 0);
  dslash(tmp, u, phi, isign, 1);

  result -= half*tmp; // (Nd + M ) phi – 0.5 Dslash phi
}
```

The function needs u and Mass

MR Solver

- We have an operator, now we need a solver. The simplest one is MR. The algorithm solves the system $M\psi = \chi$

Algorithm: MR

Start with: $\psi^0 = \text{Initial Guess}$, $r^0 = \chi - M\psi^0$

for m=1, 2, 3 ... (until convergence or maximum iterations) :

$$p^m \leftarrow M r^{m-1}$$

$$\alpha_m \leftarrow \omega \frac{\langle p^m, r^{m-1} \rangle}{\langle p^m, p^m \rangle}$$

$$\psi^m \leftarrow \psi^{m-1} + \alpha_m r^{m-1}$$

$$r^m \leftarrow r^{m-1} - \alpha_m p^m$$

ω is an overrelaxation parameter

MR Implementation

- The full implementation is in lib/invmr.cc
- Look at it at your own leisure
- Consider its definition:

```
//! Minimal-residual (MR) algorithm for a Unprec Wilson Linear Operator
  void InvUnprecWilsonMR(const LatticeDiracFermion& chi,
                         LatticeDiracFermion& psi,
                         const Real& MRovpar,
                         const Real& RsdMR,
                         int MaxMR,
                         const multi1d<LatticeColorMatrix>& u, // For Wilson M
                         const Real& Mass,  // For Wilson M
                         int isign,  //
                         int& n_count,   // No of iters gets written here
                         Double& resid);    // True residuum gets written here
```

ω

The main MR loop

- The main loop (lib/invmr.cc) is as below

Control iterations

Apply the matrix

```
while( (k < MaxMR) && (toBool(cp > rsd_sq)) )
{
      ++k;
      M_unprec_wils(Mr, r, u, isign, Mass); // Mr = M*r

      c = innerProduct(Mr, r); // c = < M.r, r >
      d = norm2(Mr);           // d = | M.r | ** 2
      a = c / d;               // alpha = <M.r,r> / <Mr,Mr>
      a *= Mrovpar;            // alpha *= 'omega'

      psi += a*r;              // psi <- psi + a*r
      r -= a*Mr;               // r <- r - a*Mr
      cp = norm2(r);           // ||r^2|| for termination
}
```

- looks just like the "Algorithm“

Using the MR Solver In the Code

- in example2/example2_1.cc:

```
multi1d<LatticeColorMatrix> u(Nd);
// ... Startup the field somehow

// Make a random (gaussian) source
LatticeDiracFermion psi,chi;  gaussian(chi);
psi=zero;  // Initial Guess
Real Mass = Real(0.1);
Real MRovpar=Real(1.1); // Omega
Real RsdMR = Real(1.0e-6); // Target residuum
int MaxMR = 1000; // Maximum iters
int isign=1;  // Want to solve with matrix not its dagger
int n_count;  // How many iterations it really took
Double resid;   // What the true absolute residuum is
Example::InvUnprecWilsonMR(chi, psi, Mrovpar, RsdMR, MaxMR,
                           u, Mass, isign, n_count,resid);
```

Unphysical for illustration only

It is good practice to check results:

```
LatticeDiracFermion Msolution;

 Example::M_unprec_wils(Msolution,
                        psi,
                        u,
                        isign,
                        Mass);

 LatticeDiracFermion our_resid;

 our_resid = chi - Msolution;

 QDPIO::cout << "Our absolute residuum is: "
             << sqrt(norm2(our_resid)) << endl;
 QDPIO::cout << "Our relative residuum is: " <<
          << sqrt(norm2(our_resid)/norm2(b)) << endl;
```

Multiply back:
M * solution

compute real
chi - M*solution

Let Us Pause and Reflect

- So we can now invert the Wilson Fermion matrix on a random source. Great!
- But there is something not just right:
  - Our MR algorithm doesn't care about M
  - But the code has Unpreconditioned Wilson Fermion matrix hardwired in. It is specific to these fermions.
  - Extra parameters (u, and Mass) 'pollute' solver interface
- It doesn't have to be this way
  - Encapsulate Fermion Matrix and Parameters
    - C++ Function objects (a.k.a functors)
  - Template Fermion Types in Solver

A Bit of Object Oriented Design

- First a Base Class for a LinearOperator Interface

operator() will make it look like a function call ie:
M(result, source, +/-1);
+/-1 => dagger or not.

```
// T is the type for the fermions...
 template<typename T>
   class LinearOperator {
   public:
     // Automatic cleanup
     virtual ~LinearOperator() {}

     // This is what makes it look like a 'function'
     // allow inheriting classes to override this by making it
     // virtual
     virtual void operator()(T& result, const T& source, int isign) const = 0;

     // The subset on which the lattice acts
     virtual Subset& subset() const = 0;
 };
```

A Bit of Object Oriented Design

- Next – a subclass for LinearOperators acting on the whole lattice

UnprecLinearOperator<T> implements at least the public methods of LinearOperator<T>

```
template<typename T>
class UnprecLinearOperator : public LinearOperator<T> {
public:
  // Correct Cleanup
  virtual ~UnprecLinearOperator() {}

  // This is what makes it look like a 'function'
  // allow inheriting classes to override this by making it
  // virtual
  virtual void operator()(T& result, const T& source, int isign) const = 0;

  // The subset on which the lattice acts
  // In QDP++ all means on every part of the lattice.
  Subset& subset() const { return all; }
};
```

Unpreconditioned linear operators act on the whole lattice.

A Bit Of Object Oriented Design

- Finally a concrete class (implementation) for Wilson:

UnprecWilsonLinOp implements at least the public methods of UnprecLinearOperator<T> with T=LatticeDiracFermion

```
class UnprecWilsonLinOp : public UnprecLinearOperator<LatticeDiracFermion> {
 public:
   ~UnprecWilsonLinOp() { }

   // Constructor. This is where we package up the gauge field and the mass.
   UnprecWilsonLinOp(const multi1d<LatticeColorMatrix>& u_,
                     const Real& Mass_) : u(u_), Mass(Mass_) {}

   // supply body in .cc file
   void operator()(LatticeDiracFermion& result,const LatticeDiracFermion& source,
                   int isign) const;

   // Subset function is inherited.
 private:
   multi1d<LatticeColorMatrix> u; // My packaged gauge field
   Real Mass;                     // Mass
 };
```

A bit of Object Oriented Design

- The body of the operator() is:

The actual implementation

```
void
UnprecWilsonLinOp::operator()(LatticeDiracFermion& result,
                    const LatticeDiracFermion& source,
                    int isign) const
{
  Real mass_term=Real(Nd)+Mass;
  Real half = Real(0.5);

  result=mass_term*source;                  // (Nd + M) source (all sites)

  LatticeDiracFermion  tmp;
  Example::dslash(tmp, u, source, isign, 0);   // red sites
  Example::dslash(tmp, u, source, isign, 1);   // black sites
  result -= half*tmp;                          // all sites
}
```

Now we can create a Linear Op.

- We can make a linear operator specific to a gauge field and mass:

```
// Create an operator.
Example::UnprecWilsonLinOp   M(u,Mass);
```

- We should of course test it:

```
Example::UnprecWilsonLinOp   M(u,Mass);

// Old Way
Example::M_unprec_wils(solution, source, u, isign, Mass);

M(Msolution, source, isign); // Use Function Object version

our_resid = Msolution - solution;  // Take the difference
QDPIO::cout << "The difference between function and class is: "
            << sqrt(norm2(our_resid)) << endl;
```

Now for the solver

- We can pass a Reference to the base class:

```
//! Minimal-residual (MR) algorithm for a generic Linear Operator
 void InvMR(const LinearOperator<LatticeDiracFermion>& M,
            const LatticeDiracFermion& source,
            LatticeDiracFermion& target,
            const Real& MRovpar,
            const Real& RsdMR,
            int MaxMR,
            int isign,  // solve with matrix or dagger
            int& n_count,   // No of iters gets written here
            Double& resid);    // True residuum gets written here
```

u and Mass parameters are GONE!

- Function Body looks cleaner too:

```
k = 0;
while( (k < MaxMR) && (toBool(cp > rsd_sq)) )
{
  ++k;
  M(Mr, r,isign);     // Mr = M r
  ...
```

Now for the Rest...

- Template the LatticeDiracFermion throughout the invmr:

```
template<typename T>
void InvMR_a(const LinearOperator<T>& M,
             const T& chi,
             T& psi,
             const Real& MRovpar,
             const Real& RsdMR,
             int MaxMR,
             int isign,
             int& n_count,
             Double& resid)
{
  T Mr;
  ...
```

T is the template parameter, instead of LatticeFermion

Subtlety about templates

- With GCC at least, one cannot declare a template in one file and put the body in the other (linkage issues). So I can't do in invmr.h:

```
template<typename T>
void InvMR(const LinearOperator<T>& M, ...)
```

- and then put the body in invmr.cc as:

```
template<typename T>
void InvMR(const LinearOperator<T>& M, ...) { ... }
```

- This will compile, but not link (Unresolved Symbol errors) with gcc.

Workaround for Subtlety

- Declare templated file in the .cc file only.
- Put in a specialization (with no templates) to wrap it for the desired types.
- Only put the specialization into the .h
- So in the .cc file:

```
template<typename T>
void InvMR_a(const LinearOperator<T>&
             const T& chi,
             T& psi,
             const Real& MRovpar,
             const Real& RsdMR,
             int MaxMR,
             int isign,
             int& n_count,
             Double& resid)
{
  T Mr; ...
```

InvMR_a has file scope. Only exists in .cc file

Workaround of Subtlety

- Then later on in the .cc file we have a wrapper

```
void InvMR(const LinearOperator<LatticeDiracFermion>& M,
           const LatticeDiracFermion& source,
           LatticeDiracFermion& target,
           const Real& MRovpar,
           const Real& RsdMR,
           int MaxMR,
           int isign,  // solve with matrix or dagger
           int& n_count,   // No of iters gets written here
           Double& resid)     // True residuum gets written here
 {
   InvMR_a(M,source,target,MRovpar,RsdMR,MaxMR,isign,n_count,resid);
 }
```

Wrap InvMR_a in non-templated function

- For a new Fermion type, one needs a new wrapper function
- The wrapper function gets declared in the .h file making it visible to all who include the .h file

Subsets 1

- Remember the subset() function?
- A Subset can be used to identify a subset of sites on our lattice eg:
  - red sites vs. black sites in red-black preconditioning
  - timeslices (see later).
- In our LinearOperator we put in a subset() function to tell us which subset of sites the operator acts on. We can use this in our solver eg:
  - assignments to (target) subsets: `psi[s] += r * a;`
  - inner products over subsets: `c = innerProduct(Mr, r, s);`
  - norms over subsets: `norm2(r,s);`
- We add these changes in to our solver so we can precondition later without rewriting the solver

subset index only on the 'target'

We now have:

- An unpreconditioned Wilson operator Function Object
- A generic, reusable MR solver
- You can find the code for all of this in the example2 directory:
  - include/linop_class.h
  - include/unprec_wilson_2_w.h
  - include/invmr2.h
  - lib/unprec_wilson_2_w.cc
  - lib/invmr2.cc

Next Step: Creating a source

- Point source on the origin. This is not easy to do in a data parallel way since it refers to a concrete site, spin, color.
- QDP++ provides functions to access sites, spins, colors:
  - `pokeSite(dst, src, coords);`
  - `pokeColor(dst, src, color);`
  - `pokeSpin(dst, src, spin);`
  - `result = peekSite(src, coords);`
  - `result = peekColor(src, color);`
  - `result = peekSpin(src, spin);`
- NB: These are data parallel, so the result of a 'peek' gets broadcast to all nodes. Likewise the poke functions get called on all nodes.

Creating a source with Poke and Peek

- Here is a way to create a point source at the origin

```
void makePtSourceOrigin(LatticeFermion& src, int spin, int color)
{
  Complex cone=cmplx(Real(1),0);  // Complex 1
  ColorVector tmp_cvec = zero;
  pokeColor(tmp_cvec,cone, color); // Put into the color component of a vector

  Fermion tmp_ferm=zero;
  pokeSpin(tmp_ferm,tmp_cvec,spin); // Put color vec into spinor

  multi1d<int> coords(Nd);
  coords[0] = 0; coords[1]=0; coords[2]=0; coords[3]=0;

  src=zero;
  pokeSite(src, tmp_ferm, coords); // Inject spinor into a source
}
```

- There are other ways but this is most 'portable'.

Moving a LatticeFermion to and fro

- QDP++ supplies a propagator type: **LatticePropagator**
- This holds the full NsxNc dimensional matrix per site.
- Useful for computing correlation functions
- We supply some functions to move fermions with source spin and source color components to and from propagators (in the include/transf.h lib/transf.cc files):

```
void Example::PropToFerm(LatticeFermion& ferm,
                         const LatticePropagator& prop,
                         int spin, int color)
```

Extract spin color component from 'prop' into 'ferm'.

```
void Example::FermToProp(LatticePropagator& prop,
                         const LatticeFermion& ferm,
                         int spin, int color);
```

Insert source spin color component fermion from 'ferm' into 'prop'.

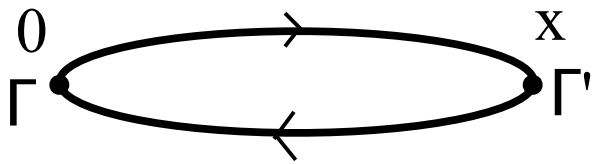
The Main Propagator Loop

```
LatticePropagator result  = zero;              // The propagator itself
for(int spin=0; spin < Ns; spin++) {           // Loop over source spin and color
   for(int color=0; color < Nc; color++) {

     LatticeFermion pt_source=zero;                          // Make the source
     Example::makePtSourceOrigin(pt_source, spin,color);

     LatticeFermion soln = zero;                             // Initial Guess

     isign=1;
     Example::InvMR(M,  pt_source, soln,  Mrovpar,  RsdMR,  MaxMR,  isign,
                    n_count, resid);                         // Inversion

     QDPIO::cout << "Solver took " << n_count << " iterations" << endl;
     QDPIO::cout << "Solver claims residuum is " << resid << endl;

     Example::FermToProp(result, soln,spin,color);      // Move to the propagator
   }
 }
```

Now Let us Make a Correlation Function

- We can easily Compute the Zero Momentum Meson correlation function:

$$C(x)_{\Gamma,\Gamma'} = \mathrm{Tr}\ [\gamma_5 G(x,0)\gamma_5 \Gamma G(x,0)\Gamma']$$



- We will use $\Gamma=\Gamma'=\gamma_5$ by way of example:

```
LatticePropagator anti_prop = Gamma(15)*result*Gamma(15);

 // Compute the pion
 LatticeComplex correlation_fn = trace(adj(anti_prop)*Gamma(15)
                                       *result*Gamma(15));
```

Recall: 15=0b1111 Gamma(15)=$\gamma_0\gamma_1\gamma_2\gamma_3=\gamma_5$

Timeslice Summing.

- We need to define a Set function object to specify timeslices:

```
class TimeSliceFunc : public SetFunc
{
public:
  TimeSliceFunc(int dir): mu(dir) {}

  int operator()(const multi1d<int>& coord) const
  {return coord[mu];}

  // The number of subsets is the length of the lattice
  // in direction mu
  int numSubsets() const {return Layout::lattSize()[mu];}

private:
  int mu;// Time direction
};
```

Given a coordinate (of a site) return its subset (timeslice)

How many subsets are there ?

Using the Set

- First we must create an instance of our Set:

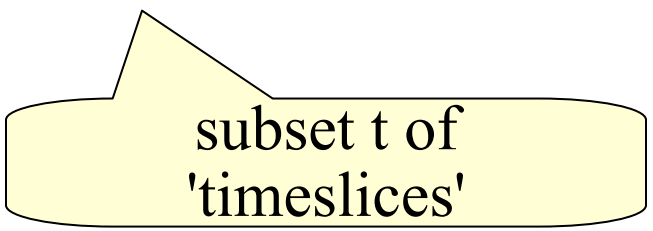
```
Set timeslices;
timeslices.make(TimeSliceFunc(3)); // Make the timeslice in direction 3
```

- Second, we create space for the summed correlation fn:

```
multi1d<DComplex> hsum( timeslices.numSubsets() );
```

- Finally, we perform the sum over each timeslice:

```
for(int t=0; t < timeslices.numSubsets(); t++) {
  hsum[t] = sum(correlation_fn, timeslices[t]);
  QDPIO::cout << "t= " << t << " Pion(t) = " << hsum(t) << endl;
}
```



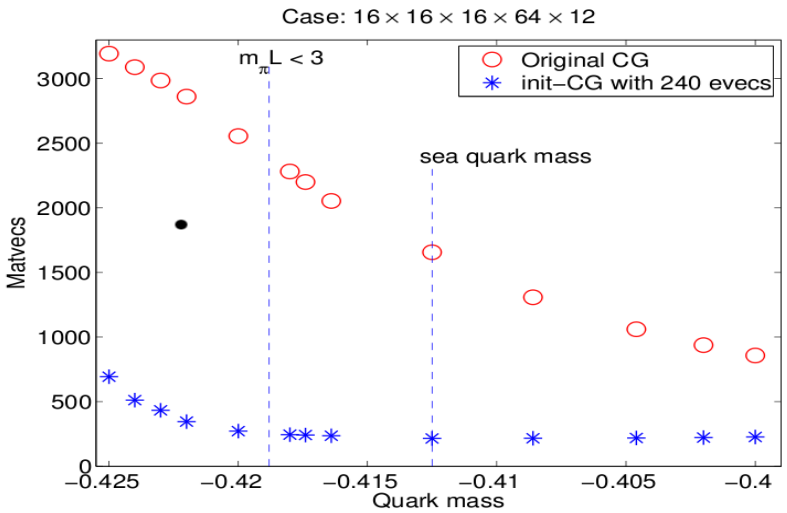
Main Goal Reached

- We have reached our main goal, of computing a propagator and the zero momentum pion correlator on a random gauge field.
- In the code, you can find pretty much everything discussed here in the **lib/** and **include/** directories as well as in the **example2_model.cc** file
- In principle you should now be capable of a lot
  - Inverting on noisy sources
    - Noisy estimators: $\langle \bar{\psi}\psi \rangle = \langle \mathrm{Tr}\ M^{-1} \rangle_U = \langle \phi_i^\dagger M^{-1} \phi_i \rangle_{\phi_i, U}$
  - All mesons at zero momentum
    - ie meson masses

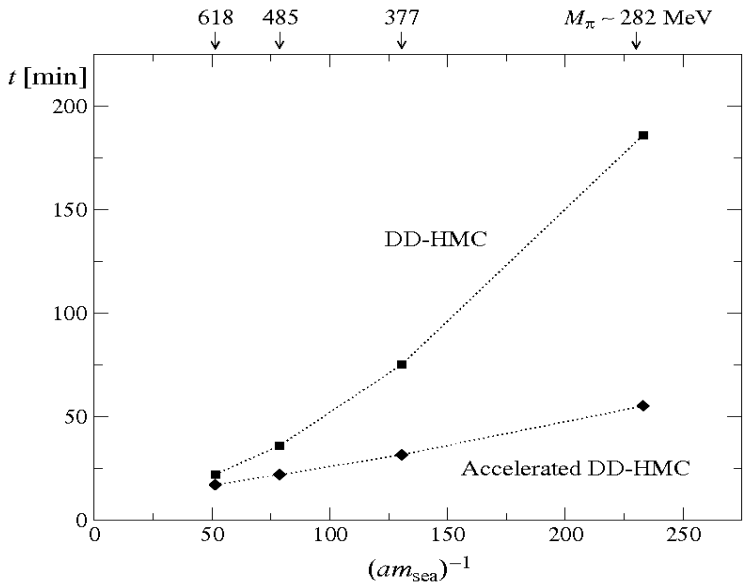
Advanced solver topics

- We have touched on the simplest of solvers here (MR)
  - In the exercises, you will also meet ConjugateGradients (CG)
  - In practice solvers make up a very rich field of research
    - see references next slides
  - Solvers for multiple right hand sides
    - Deflation
      - project out low lying eigenspace responsible for slow convergence
    - Multi-Grid
      - sample successively lower eigenmodes with reduced degrees of freedom
  - Architecture aware solvers
    - Multiple-precision solvers
      - do most work in reduced precision, but maintain desired precision in the result
    - Reduced communication solvers
      - useful when communications is a bottleneck

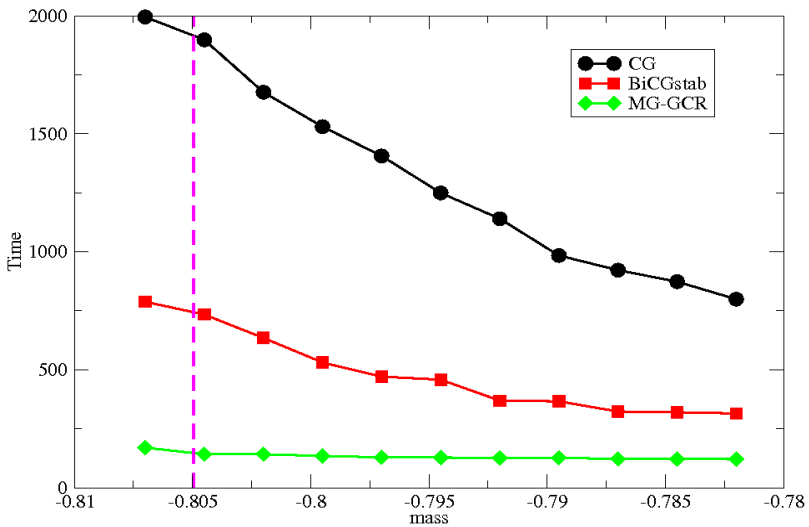
Slowing Down the Slowing Down



EigCG Deflation
(Orginos, Stathopoulos)
arXiv:0707.0131 [hep-lat]



Deflated DD-HMC
(Luscher et al)
arXiv:0710.5417[hep-lat]
JHEP0712:011,2007



Adaptive Multigrid
(Clark et al)
arXiv:0811.4331 [hep-lat]

Solver Research References

- Schwarz preconditioner
  - (SAP+GCR) Lüscher, Comput.Phys.Commun. 156(2004) 209-220
  - (RAS+ flex. BiCGStab) Osaki, Ishikawa, PoS(Lattice2010), 036
  - (DD+GCR) Babich et. al.Proceedings of SC'11, 2011
- Multi-Grid:
  - Babich et. al., Phys.Rev.Lett.105:201602,2010
  - Osborn et. al., PoS Lattice2010:037,2010
- Deflation:
  - Lüscher, JHEP 0707:081,2007, JHEP 0712:011,2007
  - Stathopoulos & Orginos: SIAM J. Sci. Comput. 32, pp. 439-462
- Multi Precision etc:
  - Clark, et. al., Comp. Phys. Commun. 181:1517-1528, 2010

High Performance Solver Libraries

- Solvers are typically implemented in libraries
  - QUDA - GPU solver library (M. Clark et. al.)
    - http://lattice.github.com/quda/
  - MDWF - Domain Wall Fermion Library (A. Pochinsky)
    - http://www.mit.edu/~avp/mdwf/index.html
  - QOPQDP - Staggered & Wilson Solvers and Forces (J. Osborn)
    - http://usqcd.jlab.org/usqcd-docs/qopqdp/
  - BFM - BlueGene/Q Fermion matrix library (P. Boyle)
    - http://www2.ph.ed.ac.uk/~paboyle/bagel/Bagel.html

- So called 'Level 3' libraries (in USQCD SciDAC layers)
- Large packages like Chroma integrate/wrap such libraries.
- These libraries are typically highly optimized to target machine
  - We'll discuss issues and optimization in the next lecture.

Advanced: Even-Odd Preconditioning

- We can perform a Schur Decomposition of M based on whether the Matrix elements connect even(red) or odd (black) sites

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

- So we have:

$$
\begin{aligned}
M\ \phi &= \chi \\
\Rightarrow L\tilde{M}U\phi &= \chi \\
\Rightarrow \tilde{M}\phi' &= \chi', \quad \chi' = L^{-1}\chi, \quad \phi = U^{-1}\phi'
\end{aligned}
$$

- What is more:

$$
L^{-1} = \begin{bmatrix} 1 & 0 \\ -M_{oe}M_{ee}^{-1} & 1 \end{bmatrix} \quad U^{-1} = \begin{bmatrix} 1 & -M_{ee}^{-1}M_{eo} \\ 0 & 1 \end{bmatrix}
$$

Even Odd Preconditioning

- We write our sources and solutions in even-odd form:

$$\chi = \begin{bmatrix} \chi_e \\ \chi_o \end{bmatrix} \qquad \psi = \begin{bmatrix} \psi_e \\ \psi_o \end{bmatrix}$$

Group even (odd) sites together

- Prepare the source:

$$\chi' = L^{-1}\chi = \begin{bmatrix} \chi_e \\ \chi_o - M_{oe}M_{ee}^{-1}\chi_e \end{bmatrix}$$

- Solve: $\tilde{M}\phi' = \chi'$

$$\phi' = M_{ee}^{-1}\chi'_e \quad \text{trivially}$$

$$\left(M_{oo} - M_{oe}M_{ee}^{-1}M_{eo}\right)\phi'_o = \chi'_o \quad \text{with solver}$$

must be **easy** to apply M_{ee}^{-1}

- Reconstruct solution:

$$\phi = U^{-1}\phi' = \begin{bmatrix} \phi'_e - M_{ee}^{-1}M_{eo}\phi'_o \\ \phi'_o \end{bmatrix}$$

Even More Even-Odd Preconditioning

- For Wilson Fermions:

$$M_{oo} = M_{ee} = (N_d + M) \quad M_{oo}^{-1} = M_{ee}^{-1} = \frac{1}{(N_d + M)}$$

$$M_{eo} = -\frac{1}{2} D_{eo}$$

Definitely easy to apply

$$\tilde{M} = (N_d + M) - \frac{1}{4(N_d + M)} D_{oe} D_{eo}$$

Even Odd Prec. LinOp Class Design

- Add methods for: $M_{ee}, M_{eo}, M_{oe}, M_{oo}, M_{ee}^{-1}$
- operator() now applies: $\tilde{M} = M_{oo} - M_{oe} M_{ee}^{-1} M_{eo}$
- Can code operator() in terms of: $M_{ee}, M_{eo}, M_{oe}, M_{oo}, M_{ee}^{-1}$
  - This produces a default implementation
  - The default may be wasteful: make it virtual so implementations can override it.

The code...

```
template<typename T>
class SchurEvenOddLinearOperator : public LinearOperator<T> {
public:
  virtual ~SchurEvenOddLinearOperator() {}
  virtual void evenEvenLinOp(T& result, const T& source, int isign) const=0;
  virtual void evenOddLinOp(T& result, const T& source, int isign) const=0;
  virtual void oddEvenLinOp(T& result, const T& source, int isign) const=0;
  virtual void oddOddLinOp(T& result, const T& source, int isign) const=0;
  virtual void evenEvenInvLinOp(T& res, const T& source, int isign) const=0;

  virtual void operator()(T& result, const T& source, int isign) const {
    oddOddLinOp(result, source, isign);
    T tmp,tmp2;
    evenOddLinOp(tmp, source, isign);    // M_{eo}
    evenEvenInvLinOp(tmp2, tmp, isign);  // M^{-1}_{ee}
    oddEvenLinOp(tmp, tmp2, isign);      // M_{oe} M^{-1}_{ee} M_{eo}
    result[ subset() ] -= tmp;
  }

  virtual const Subset& subset() const = 0;
};
```

The default
(waste a function call)

The implementation for Wilson

```
class PrecWilsonLinOp : public SchurEvenOddLinearOperator<LatticeFermion> {
  public:
    ~PrecWilsonLinOp() {}
    PrecWilsonLinOp(multi1d<LatticeColorMatrix>& u_, Real Mass_) :u(u_) {
        mass_term = Nd + Mass_ ;
    }

    // The various bits:
    void evenEvenLinOp(LatticeFermion& result, const LatticeFermion& source, int
isign) const {
      result[ rb[0] ] = mass_term * source;
    }

    void evenOddLinOp(LatticeFermion& result, const LatticeFermion& source, int
isign) const {
      Real half=Real(-0.5);
      dslash(result, u, source,isign,  0);  result[rb[0]] *= half;
    }

    // odd-even and odd-odd proceed similarly ...

    void evenEvenInvLinOp(LatticeFermion& res, const LatticeFermion& source, int
isign) const {
      Real inv_mass_term = Real(1)/mass_term;
      res[rb[0]]= inv_mass_term * source;
    }
```

The Implementation For Wilson

```
   const Subset& subset(void) const {
     return rb[1];
   }

   void operator()(LatticeFermion& result, const LatticeFermion& source, int isign)
const {
    Real inv_term = Real(1)/(Real(4)*mass_term);
    result[ rb[1] ] = mass_term*source;

     LatticeFermion tmp1, tmp2;
     dslash(tmp1, u, source,isign, 0);
     dslash(tmp2, u, tmp1,  isign, 1);
     result[ rb[1] ] -= inv_term*tmp2;
   }


 private:
   Real mass_term;
   multi1d<LatticeColorMatrix> u;

 };
```

Override default with more efficient version since we have trivial M_{ee}^{-1}

Preparing the Source

Example::PrecWilsonLinOp tildeM(u, Mass);

```
// Loop over source spin, color
for(int spin=0; spin < Ns; spin++) {
  for(int color=0; color < Nc; color++) {

    QDPIO::cout << "Solving on (spin,color) component
("<<spin<<","<<color<<") "
                << endl;

    // Make the source normally
    LatticeFermion pt_source=zero;
    Example::makePtSourceOrigin(pt_source, spin,c

    // Prepare the source for preconditioning
    LatticeFermion prep_source;
    prep_source  = pt_source; // Both checkerboards

    LatticeFermion tmp,tmp2;
    tildeM.evenEvenInvLinOp(tmp, pt_source, 1);
    tildeM.oddEvenLinOp(tmp2, tmp, 1);
    prep_source[rb[1]] -= tmp2;
```

From Diagonal Part of L^{-1}

From off-diagonal part of L^{-1}

Doing the Solve

- This is essensially just the same as before, thanks to the solver design:

```
// Invert on the odd part using source
// set isign to 1 : ie use matrix, not its dagger

isign=1;
Example::InvMR(tildeM,
               prep_source,
               prec_soln,
               MRovpar,
               RsdMR,
               MaxMR,
               isign,
               n_count,
               resid);


QDPIO::cout << "Solver took " << n_count << " iterations" << endl;
QDPIO::cout << "Solver claims residuum is " << resid << endl;
```

Since we considered subsets in the solver, we can reuse it directly, with the preconditioned M

Reconstructing The Solution

```
    // soln U^{-1}
     LatticeFermion soln = prec_soln;

     tildeM.evenOddLinOp(tmp, prec_soln,1);
     tildeM.evenEvenInvLinOp(tmp2, tmp, 1);
     soln[rb[0]] -= tmp2;

     // Now put the result into our lattice pi
     Example::FermToProp(result, soln,spin,color);
   }
}
```

From diagonal part of U^{-1}

From off-diagonal part of U^{-1}

And we're done...

Exercise 1: Different Linear Operator

- Write a parity breaking (twisted mass) Wilson Linear Operator. This involves adding a $-i\mu\gamma_5$ term to the Wilson Operator
  - Hints: **timesI(x)** can be used to multiple x by i
  - You should make this a function object like the Wilson case. You'll need an extra parameter for the constructor
  - Remember that the i switches sign on daggering.
  - First write an unpreconditioned one.
  - Then think about the Schur even-odd preconditioned one
    - Is M_{ee}^{-1}still trivial?

Exercise 2: A different solver

- The Conjugate Gradients (CG solver) to solve $M^\dagger M\phi = \chi$ is ($\phi_0 = \phi$ is an Initial Guess):

1. Compute $r_0 = \chi - M^\dagger M\phi_0,\ p_0 = r_0$
2. For $j = 0, 1, \ldots$ until convergence:
3. $\alpha_j = \dfrac{\langle r_j, r_j\rangle}{\langle Mp_j, Mp_j\rangle}$
4. $\phi_{j+1} = \phi_j + \alpha_j p_j$
5. $r_{j+1} = r_j - \alpha_j \left(M^\dagger M\right) p_j$
6. $\beta_j = \dfrac{\langle r_{j+1}, r_{j+1}\rangle}{\langle r_j, r_j\rangle}$
7. $p_{j+1} = r_{j+1} + \beta_j p_j$
8. End For

Exercise 2: A different solver

- Implement this solver along the lines of the MR solver
  - With a Templated Fermion Type
  - Expecting a Linear Operator
  - Allowing an arbitrary subset
  - Test it by multiplying the results with $M^\dagger M$
- Propagators with Conjugate Gradients:
  - To solve:

$$M\phi = \chi$$

- We turn to Conjugate Gradients on the Normal Equations (CGNE) – solve:

$$M^\dagger M \ \phi = M^\dagger \phi$$

- We need to modify the source by hitting it with $M^\dagger$

The Conjugate Gradients Solver

- There is an implementation of this in chroma. You can look at it online at:

http://usqcd.jlab.org/usqcd-software/chroma/chroma/docs/doxygen/html/invcg2_8cc-source.html

- CG is "optimal" in some sense for Hermitian Positive Definite matrices
  - Can outperform MR,
  - Definitely needed for forces where one is solving: $M^\dagger M \phi = \chi$
- You can find out more about the wonderful world of solvers from Yousef Saad's book:
  - Iterative Methods for Sparse Linear Systems
    - get it free online at:

http://www-users.cs.umn.edu/~saad/books.html

Final Thoughts

- We haven't explored
  - non-zero momenta and the Fourier Transform
  - the complicated quark contractions for baryons
  - Fermion Boundary conditions (eg: antiperiodic)
- Most of these come pre-written for you in frameworks like Chroma and would have served as a distraction here.
- More introductory material on solvers:
  - Saad's book is excellent
  - Also Henk van der Vorst's notes: http://www.math.uu.nl/people/vorst/lecture.html
  - G. Golub & C. van Loan (1996), Matrix computations, third edition, The Johns Hopkins University Press
  - Multiple shift (mass) solvers: B. Jegerlehner arxiv:hep-lat/9612014