1 Introduction

FermiQCD is a software library for fast development of parallel Lattice QCD code. It includes examples and applications. The latest version of FermiQCD supports:

- Fully Object Oriented design
- Natural syntax
- Supports Wilson, Staggered and Domain-wall fermions
- Parallelization based on MPI is hidden to high level programmer
- SU3 operations optimized for P4 (SSE and SSE2 instructions)
- PSIM technology for emultaing parallel processes on a single processor machine and improving performance on multi-threaded processors
- Compile and runs on multiple architectures including Linux, Mac and Windows (with cygwin).

1.1 Installation

1. Download the file fermiqcd_4.0.tar.gz
2. Execute: gunzip fermiqcd_4.0.tar.gz
3. Execute: tar xvf fermiqcd_4.0.tar.gz

1.2 Files

Upon installation fermiqcd creates the following directory structure

```
\FermiQCD
\FermiQCD\Version_4.0
\FermiQCD\Version_4.0\Libraries
\FermiQCD\Version_4.0\Documentation
\FermiQCD\Version_4.0\Examples
\FermiQCD\Version_4.0\Converters
```

`\FermiQCD\Version_4.0\Tests`
`\FermiQCD\Version_4.0\Other`

The folder "Libraries" contains the software libraries. There is no need to precompile anything since all the code is in the header files. This is done to simplify usage and allow the compiler to perform better template optimizations.

The files starting with `mdp_` belong to the Matrix Distributed Processing (MDP) library, required by FermiQCD. The files starting with `fermiqcd_` are the proper FermiQCD files. They are distrubuted together but are covered by different licenses.

The folder "Documentation" contains licenses and documentation for MDP and FermiQCD. MDP and FermiQCD cannot be redistributed without this folder.

The folder "Examples" contains all of the examples described in these tutorial.

The folder "Converters" contains converters for starndard QCD file formats into the MDP file format used by FermiQCD.

You may ask: why another file format? Technically FermiQCD itself does not specify a file format but it defines fields of objects defined on a lattice. Field objects inherit parallel load/save operations from the underlying MDP library which specifies a single file format for any generic field (any number of lattice dimensions, any structure at the site) optimized for parallel IO. None of the previous file formats was general enough since they are all specific for a type of field and usually for 4 dimensions.

In any case FermiQCD can read UKQCD, MILC, CANOPY and many ASCII file formats. If you format is not supported email me and I will send you a converted in a week at no charge.

The folder "Tests" contains programs and libraries that I consider a work in progress. They are included with the official distribution to allow people to contribute.

The folder "Other" contains examples of applications in fields other than QCD, for example Cellular Authomata.

1.3 Compilation instructions

To compile all the example go into /FermiQCD/Version_4.0/Examples and type

```
make all
```

or compile any of the programs with

```
g++ prg.cpp -I../Libraries -o prg.exe -O3 [options]
```

You may want to edit the first few lines of the Makefile in order to enable some specific compiler options:

```
g++ prg.cpp -I../Libraries -o prg.exe --O3 -DLINUX
```

will compile for Linux and be able to measure running time

```
g++ prg.cpp -I../Libraries -o prg.exe -O3 -DUSE_DOUBLE_PRECISION
```

will compile the algorithms to use double precision

```
g++ prg.cpp -I../Libraries -o prg.exe -O3 -DSSE2
```

will compile the algorithms to use SSE/SSE2 optimizations (only on P4 processors)

```
g++ prg.cpp -I../Libraries -o prg.exe -O3 -DPARALLEL
```

will compile with MPI, otherwise it will compile with PSIM Parallel SIMulator (enables emulating parallel processing on a single processor PC, it is recommended on a multi-threaded processor where it will increase speed).

Compiler options can be combined.

1.4 A first program

1.5 General principles

More or less any FermiQCD the same structure:

```
#include ''fermiqcd.h''
int main(int argc, char** argv) {
   mdp.open_wormholes(argc,argv);
   // declare conventions
   // declare lattices
   // declare fields
   // declare variables
   // run algorithms
   mdp.close_wormholes();
   return 0;
}
```

Where open_wormholes and close_wormholes respectively start and stop communications.

Conventions are declared using the command

```
declare_base_matrices(''FERMIQCD'');
```

which basically declares gamma matrices (Gamma[], Gamma5, and Gamma1). Other conventions supported include "UKQCD".

A lattice, for examle a 4D lattice 16×8^3 called `mylattice` can be declared as follows

```
int box[]={16,8,8,8};
mdp_lattice lattice(4,box);
```

The constructor of the class can take optional parameters that are discussed in the next section. The optional parameters specify how the lattice object has to be partionioned over the parallel processes (default: by timeslices), specify the lattice topology (default: mesh) and the size of the buffer used for parallel communications (default: optimized for Wilson and Clover actuions).

A **lattice object** contains a parallel random number generator. To ask every site to print a uniform random number:

```
#include ''fermiqcd.h''
int main(int argc, char** argv) {
   mdp.open_wormholes(argc,argv);
   declare_base_matrices(''FERMIQCD'');
```

```
   int box[]={16,8,8,8}
   mdp_lattice mylattice(4,box);
   site x(mylattice);
   forallsites(x)
      cout<<mylattice.random(x).plain()<<endl;
   mdp.close_wormholes();
   return 0;
}
```

Class **site** represents a site of the lattice; `forallsites(x)` is a parallel loop over all lattice sites using the site `x` as looping variable and

```
mylattice.random(x)
```

is the **random number generator** associated to the site `x`. `plain()` is a method of the random generator that returns a uniform random number in (0,1). Other methods include

```
mylattice.random(x).SU(n)
```

that generates a random SU(n) matrix using Cabibbo-Marinari.

To declare a **field** of floating point numbers in single precision called `myfield` on our lattice

```
mdp_field<float> myfield(mylattice);
```

To **save** or **load** a field[1]

```
string filename;
myfield.save(filename);
myfield.load(filename);
```

The following code creates a filed of floating point numbers, initialize them at random and saves them

[1]This functions may crash if you there is not enough memory to allocate buffers for parallel IO. If this occurs pass a second `int` argument to the load/save functions with a value below 1024. The smoller the value, the smaller the required buffer size.

```
#include ''fermiqcd.h''
int main(int argc, char** argv) {
   mdp.open_wormholes(argc,argv);
   declare_base_matrices(''FERMIQCD'');
   int box[]={16,8,8,8}
   mdp_lattice mylattice(4,box);
   mdp_field<float> myfield(mylattice);
   site x(mylattice);
   forallsites(x)
      myfield(x)=mylattice.random(x).plain();
   myfield.update();
   myfield.save(''myfield.mdp'');
   mdp.close_wormholes();
   return 0;
}
```

Note the function **update()**. It is the single and most important function in MDP and FermiQCD. It must be called every time a field changes and before it is used. It instruct the parallel nodes to keep the copies of the lattice sites and field variables therein sinchronized.

1.6 Notes

To loop over all sites of a given parity (EVEN or ODD)

```
forallsitesofparity(x,EVEN)
```

To loop over all local sites and local copies of sites stored on other processors

```
forallsitesandcopies(x)
```

(This is used when looping to initialize a field with a local expression that does not require the parallel random number generator, in order to avoid a susequent call to the function update; if not sure, do not use it.)

To loop over all local even sites and local copies of even sites stored on other processors:

```
forallsitesandcopiesofparity(x,EVEN)
```

(Same as above.)
To print the time components of a site variable x

```
for(int k=0;k<x.lattice().ndim;k++)
   cout << x[k] << endl;
```

Where `x.lattice().ndim` reads as the number of dimensions (`ndim`) of the lattice on which the site `x` was declared. All fields and site objects have a method lattice() to obtain a reference to the underlying lattice. `x[k]` reads as the `k`-th coordinate of the site `x`. If the lattice was 4D x has four components 0,1,2, and 3. We adopt the convention of calling coordinate 0 the TIME coordinate and 1,2,3 the space coordinates.

1.7 Other fields

FermiQCD comes with a set of predefined fields

```
gauge_field
fermi_field
fermi_propagator
staggered_field
staggered_propagator
dw_fermi_field
dw_fermi_propagator
```

and algorithms to create and used them. All the FermiQCD algorithms work with any SU(n) gauge group although they are highly optimized for SU(3). The staggered algorithms also work for any even-dimensional space. Some other algorithms require a four dimensional space because of optimizations related to the gamma matrix conventions.

2 Matrix Distributed Processing

3 Short Lattice QCD tutorial and Notation

3.1 Modern Physics

Modern Physics states that:

- Any system can be uniquely descrived by a set of variables (ψ, A_μ, etc.) that represents its degrees of freedom and by a function of these variables called Action that we indicate with the symbol $S_W(...)$.
- Any observable on the system can be associated to an expression, function of the variables, that we indicate with $O(...)$.
- We cannot predict the outcome of every experiment but we can compute the expectation value of any observable by computing the path integral
$$E[O(...)] = \int [d\psi][dA_\mu] O(...) e^{iS(...)}$$
where the integral covers the domain of all possible states of the system (i.e. one integral for each degree of freedom).
- Because information travels at a finite speed, $S(...)$ must be local in space-time variables therefore it can be expressed as $S(...) = \int L(...) d^4x$ where L is called Lagrangian density and it is also function of the degrees of freedom of the system.

3.2 Quantum Chromo Dynamics

If the system we are describing is QCD, i.e. the model that describes quarks and gluons we have that:

- $\psi^f_\alpha(x)$ represent the degrees of freedom associated to quarks.
- $A_\mu(x)$ represent the degrees of freedom associated to gluons
- The Action for QCD is
$$S_{QCD}(\psi, A) = \int d^4x \left[-\frac{1}{4g^2} G_{\mu\nu} G^{\mu\nu} + \overline{\psi}^f_\alpha D^{\alpha\beta}_f[A] \psi^f_\beta \right]$$
where
$$G_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu + [A_\mu, A_\nu]$$
$$D^{\alpha\beta}_f[A] = \gamma^{\alpha\beta}_\mu (\partial_\mu + A_\mu) + 1^{\alpha\beta} m_f$$
Note that A_μ and $G_{\mu\nu}$ are 3×3 complex matrices (i.e. in adjoint representation) while ψ^f_α for every f and α is a 3 components complex vector (i.e. fundamental representation).

- The most general path integral looks like

$$E[O(...)] = \int [d\psi][dA_\mu] O(...\psi...A_\mu) P_{QCD}[\psi, A]$$

with

$$P_M[\psi, A] = e^{i\int d^4x \left[-\frac{1}{4g^2} G_{\mu\nu}G^{\mu\nu} + \overline{\psi}^f_\alpha D_f^{\alpha\beta}[A]\psi^f_\beta\right]}$$

3.3 From QCD to Lattice QCD

Lattice QCD provides a technique to define (i.e. regularize) the above integral and compute the path integral analytically. This is achieved in four steps:

1. Perform a Wick rotation $x_0 \to ix_0$ so that

$$P_{QCD}[\psi, A] \to P[\psi, A] = e^{-\int d^4x \left[-\frac{1}{4g^2} G_{\mu\nu}G_{\mu\nu} + \overline{\psi}^f_\alpha D_f^{\alpha\beta}[A]\psi^f_\beta\right]}$$

become real.

2. Perform a Wick contraction of all spinors in $O(...)$ and replace

$$\begin{aligned} ...\overline{\psi}^f_\alpha(x)...\psi^f_\beta(y)... &\to S_{\alpha\beta}(A, x, y) \\ ...\psi^f_\alpha(x)...\overline{\psi}^f_\beta(y)... &\to S_{\beta\alpha}(A, y, x) \end{aligned}$$

Note that $S = 1/D[A]$ and it is called a propagator.

3. Integrate out $[d\psi]$ and obtain

$$E[O(...)] = \int [dA_\mu] O(...S_{\alpha\beta}...A_\mu) P[A]$$

$$P[A] = e^{-\int d^4x \left[-\frac{1}{4g^2} G_{\mu\nu}G_{\mu\nu}\right] - \sum_f \log\det D_f[A]}$$

4. Change variable from $A_\mu(x)$ to $U_\mu(x) = e^{iaA_\mu(x)}$ and approximate the integral $[dA_\mu]$ with a sum over $U^{[k]}_\mu(x)$

$$E[O(...)] = \sum_k O(...S_{\alpha\beta}...U^{[k]}_\mu) P[U^{[k]}_\mu]$$

where

$$\begin{aligned} P[U_\mu^{[k]}] &= e^{-S_{Gauge}-S_{Fermi}} \\ S_{Gauge} &= \beta \sum_x \left[\sum_{\mu\nu} P_{\mu\nu}(x) \right] \\ S_{Fermi} &= \sum_f \log\det D_f[U] \end{aligned}$$

$\beta = 1/g^2$ is a regularized coupling

$$P_{\mu\nu}(x) = U_\mu(x) U_\nu(x+\mu) U_\mu^H(x+\nu) U_\nu^H(x) = -\frac{a^4}{4} G_{\mu\nu} G_{\mu\nu}$$

is called a plaquette.

$\beta = 1/g^2$ is the only physical parameter of any lattice computation other than quark masses. Because it sets the value of the regularized coupling constant g, by virtue dimensional transmutation, it sets the lattice scale in the computation.

Remember that computer programs deal with dimensionless number and not dimensionful quantities. By chaning the value of β one changes the resolution of the computation. Typically a choice of $\beta = 5.7$ is equivalent to resolution of $1GeV^{-1}$. This relation is not linear and depends on the discretization of the action.

3.4 Lattice QCD computations

Any Lattice QCD computation proceeds in the following steps:

- Identify the expression $O(...)$, in terms of $S_{\alpha\beta}$ and U that corresponds to the quantity to compute. In FermiQCD $S_{\alpha\beta}$ and U are implemented as C++ objects.
- Generate datasets $U_\mu^{[k]}$ with probability given by $P[U_\mu^{[k]}]$. This is achieved by a Markov Chain Monte Carlo and the datasets (also called gauge configurations) are generated in succession starting from a random one. In FermiQCD the markov chain is created by the `heatbath` algorithm.

- Evaluate the integrand $O(...)$ on each configuration and average the results. Given a configuration U, $S_{\alpha\beta}$ is computed by inverting $D_f^{\alpha\beta}[U]$ numerically. In FermiQCD this is done by the `generate` algorithm.

- Study the distribution of the results to estimate the statistical error in the average. In FermiQCD this is done by the `JackBoot` object.

Note that there are different ways to discretize the Action of QCD and they affect the corvence rate of the numerical algorithm. FermiQCD include various discretizations and it is easy to add more. There are also many numerical techniques to invert $D_f^{\alpha\beta}[U]$ and compute $S_{\alpha\beta}$, FermiQCD implements the minimum residure and the stabilized biconjugate gradient.

3.5 On Fermions

Often the term S_{Fermi} is ignored in the computation. In a perturbative language this is equivalent to ignore virtual quark loops. The effect of this approximaiton, called quanching, is unclear but certainly non-negligible.

Whether Fermions are treated dynamically or not one has to discretize $D_f^{\alpha\beta}[U]$ in order to invert it numerically. There are five general prescriptions to do it:

- Naive fermions. They do not work because lattice artifacts introduce un-physical zeros in the propagator S.

- Wilson fermions. The most common and natural prescription. The only problem is that it break the chiral symmetry and therefore Wilson fermions cannot be masseless.

- Staggered (Kogut-Susskind) fermions. They do not break the chiral symmetry completely and are more efficient than Wilson fermions but introduce 15 additional quark flavors for each physical flavor. One can deal with them but it is tricky.

- Domain-wall fermions. By introducing an unphysical extra-dimension one can recover chirality and have almost massless quarks. They are extreamly computationally expensive.

- Overlap (Neuberger) fermions. Have similar properties as Domain-wall fermions but the extra dimension is integrated out analytically. They are also very computationally expensive.

At this time FermiQCD suppors all the above types of fermions but the Overlap. Domain-wall fermions have never been properly tested. Wilson and Naive are implemented for both isotropic and un-isotropic lattices. Staggered fermions are implemented for any even number of space-time dimensions. All fermions and fermionic algorithms work for any SU(n) gauge group.

3.6 On the observables

Because of the Wick rotation the numerical computation of expectation values of observables $E[O(...)]$ that require an analytical continuation is difficult. This makes difficult for example to compute scettering phases.

Nevertheless some observable can be parametrized in terms of quantity that are not affected by the Wick rotation and can be computed numercally. They include masses of composite particle and the modulus of most matrix elements.

Note that only gauge invariant quantities can be observable (remember the Aaronov-Bohm effect). In the Lattice QCD language this means that any observable $O(...)$ that is not gauge invariant will have a zero expectation value. Non-trivial observables $O(...)$ are function of closed loops of quark propagators S and products of links U.

We can classify observables $O(...)$ in the following typical categories:

- Pure glue that does not contain a length scale. They measure topological properties of QCD vacuum.
- Pure glue that does contain a length scale such as the correlation between two or more loops of links. They measure masses and matrix elements of glueballs
- Product of two propagators S. They contain information about meson masses.
- Product of three propagators S contracted at the same two points. They contain information about baryon masses.

- Other product of three or more propagators S contracted in at least three points. They contain information about matrix elements between mesons, baryons or both.

Complete example programs in each category will be provided in the fourth chapter.

4 Quantum Chromo Dynamics

4.1 Pure gauge

4.1.1 Creating a hot gauge configuration

```
#include ''fermiqcd.h''
int main(int argc, char** argv) {
   mdp.open_wormholes(argc,argv);
   declare_base_matrices(''FERMIQCD'');
   int box[]={16,8,8,8}
   mdp_lattice mylattice(4,box);
   int nc=3;
   gauge_field U(mylattice,nc);
   set_hot(U);
   U.save(''gauge.0000.mdp'');
   mdp.close_wormholes();
   return 0;
}
```

the function set_hot is already implemented but could have been implemented easily as

```
void set_hot(gauge_field &U) {
   site x(U.lattice());
   forallsites(x)
      for(int mu=0; mu<U.ndim; mu++)
         U(x,mu)=U.lattice().random(x).SU(U.nc);
   U.update();
}
```

4.1.2 Creating a cold gauge configuration

```
#include ''fermiqcd.h''
int main(int argc, char** argv) {
   mdp.open_wormholes(argc,argv);
   declare_base_matrices(''FERMIQCD'');
   int box[]={16,8,8,8}
   mdp_lattice mylattice(4,box);
   int nc=3;
   gauge_field U(mylattice,nc);
   set_cold(U);
   U.save(''gauge.0000.mdp'');
   mdp.close_wormholes();
   return 0;
}
```

the function set_hot is already implemented but could have been implemented easily as

```
void set_cold(gauge_field &U) {
   site x(U.lattice());
   forallsites(x)
      for(int mu=0; mu<U.ndim; mu++)
         U(x,mu)=1;
   U.update();
}
```

Note how each site initializes `U(x,mu)` with 1 (interpreted as 3×3 identity matrix). Since every site variable is initialized with a constant and it does not depend on the random number generator it would be more efficient to ask each process to initialize also the local copies of remote site variables and avoid calling update

```
void set_cold(gauge_field &U) {
   site x(U.lattice());
   forallsitesandcopies(x)
      for(int mu=0; mu<U.ndim; mu++)
         U(x,mu)=1;
}
```

This is how set_cold is implemented in practice and it requires no parallel communication.

4.1.3 Performing Wilson heatbath steps

Code to generate 100 gauge configurations $U_\mu^{[k]}(x)$ and save them.

```
#include ''fermiqcd.h''
int main(int argc, char** argv) {
   mdp.open_wormholes(argc,argv);
   declare_base_matrices(''FERMIQCD'');
   int box[]={16,8,8,8}
   mdp_lattice mylattice(4,box);
   int nc=3;
   gauge_field U(mylattice,nc);
   set_cold(U);
   coefficients gauge;
   gauge[''beta'']=6.0;
   int niter=10;
   for(int k=0; k<100; k++) {
      WilsonGaugeAction::heatbath(U,gauge,niter);
      U.save(string(''gauge.'')+tostring(k)+string(''.mdp''));
   }
   mdp.close_wormholes();
   return 0;
}
```

Note that gauge is a variable of type **coefficients**, basically a hash table that associates a floating point to any string. Variables of type coefficients are used to pass parameters (or coefficients) to algorithms that implement a physical action.

`WilsonGaugeAction::heatbath` is the **hetbath** algorithm using the Wilson Gauge Action (who would have guessed?). It's first argument is the gauge field is acts upon (reads and writes it), the second argument is the set of coefficients (the only one it needs it "beta", i.e. the lattice spacing), the third argument is the number of iterations before returning. Noe that the number of interations technically is not a coefficient of the action therefore it not passed as a variable in the gauge object.

The line

```
string(''gauge.'')+tostring(k)+string(''.mdp'')
```

simply builds a filename without messing around with pointers since they are unsafe.

4.1.4 Improved actions

All actions in FermiQCD are implemented as pure static classes (i.e. classes with no member variables and only static methods). All gauge actions have a static method heatbath that mush follow the same prototype.

For the Wilson gauge action

```
coefficients gauge;
gauge[''beta'']=6.0;
WilsonGaugeAction::heatbath(U,gauge,niter);
```

For the MILC improved gauge action

```
coefficients gauge;
gauge[''beta'']=6.0;
gauge[''zeta'']=1.0;
gauge[''u_t'']=1.0;
gauge[''u_s'']=1.0;
string model=''MILC'';
ImprovedGaugeAction::heatbath(U,gauge,niter,model);
```

For the Morningstar improved gauge action

```
coefficients gauge;
gauge[''beta'']=6.0;
gauge[''zeta'']=1.0;
gauge[''u_t'']=1.0;
gauge[''u_s'']=1.0;
string model=''Morningstar'';
ImprovedGaugeAction::heatbath(U,gauge,niter,model);
```

4.1.5 Average plaquette

Code to compute

$$\frac{1}{6}\frac{1}{N_V}\frac{1}{N_c}\,\mathrm{Re}\,Tr\sum_{x,\mu>\nu}P_{\mu\nu}(x)$$

where N_V is the lattice volume and N_c is the number of colors.

```
#include ''fermiqcd.h''
int main(int argc, char** argv) {
   mdp.open_wormholes(argc,argv);
   declare_base_matrices(''FERMIQCD'');
   int box[]={16,8,8,8}
   mdp_lattice mylattice(4,box);
   int nc=3;
   gauge_field U(mylattice,nc);
   set_cold(U);
   coefficients gauge;
   gauge[''beta'']=6.0;
   int niter=10;
   for(int k=0; k<100; k++) {
      WilsonGaugeAction::heatbath(U,gauge,niter);
      U.save(string(''gauge.'')+tostring(k)+string(''.mdp''));
      mdp << average_plaquette(U) << endl;
   }
   mdp.close_wormholes();
   return 0;
}
```

The function average_plaquette computes the average plaquette on the gauge field U. The output is a float number. Sending the output to mdp rather than cout makes sure only process 0 prints the average plaquette even if all processes contribute to the computation.

There is a similar function

```
average_plaquette(U,mu,nu)
```

that computes the average plaquette considering the mu-nu plane only

where mu and nu are integers

$$\frac{1}{N_V}\frac{1}{N_c}\,\mathrm{Re}\,Tr\sum_x P_{\mu\nu}(x)$$

The function average plaquette is already implemented as

```
myreal average_plaquette(gauge_field &U,int mu,int nu) {
 myreal tmp=0;
 site x(U.lattice());
 forallsites(x)
   tmp+=real(trace(plaquette(U,x,mu,nu)));
 mdp.add(tmp);
 return tmp/(U.lattice().nvol_gl*U.nc);
}
```

where `plaquette(U,x,mu,nu)` is defined as

```
U(x,mu)*U(x+mu,nu)*hermitian(U(x,nu)*U(x+nu,mu));
```

mdp.add(tmp) adds the values tmp computed by the parallel processes (always to be called after summing a variable inside a forallsites loop), and `U.lattice().nvol_gl` reads as the total number of lattice sites (`nvol_gl`) in the lattice for `U`.

4.1.6 Average path

Note how the most general gluonic gauge invariant observable has the form

$$\oint_C e^{iA_\mu dx^\mu}$$

where C is a generic path. Here is the code to compute

$$\frac{1}{N_V}\frac{1}{N_c}\,\mathrm{Re}\,Tr\sum_x \oint_C e^{iA_\mu dx^\mu}$$

```
#include ''fermiqcd.h''
int main(int argc, char** argv) {
   mdp.open_wormholes(argc,argv);
   declare_base_matrices(''FERMIQCD'');
```

```
   int box[]={16,8,8,8}
   mdp_lattice mylattice(4,box);
   int nc=3;
   gauge_field U(mylattice,nc);
   set_cold(U);
   coefficients gauge;
   gauge[''beta'']=6.0;
   int niter=10;
   int mu=0, nu=1;
   int path[6][2]={{+1,mu},{+1,mu},{+1,nu},{-1,mu},{-1,nu},{-1,nu}};
   for(int k=0; k<100; k++) {
      WilsonGaugeAction::heatbath(U,gauge,niter);
      U.save(string(''gauge.'')+tostring(k)+string(''.mdp''));
      mdp << average_path(U,6,path) << endl;
   }
   mdp.close_wormholes();
   return 0;
}
```

The function `average_path` is a computes the average path on the gauge field U. where the path C is a spacified by a 2D array `d` of links and each each link is a verse (+1 or -1) and a direction (mu, nu=0,1,2,3,...). Note that because a path may be highly non-local the implementation of the funciton average_path requires, in general, quite some communication.

This function is already implemented and looks like the following:

```
mdp_complex average_path(gauge_field &U, int length, int d[][2]) {
   mdp_matrix_field psi(U.lattice(),U.nc,U.nc);
   mdp_site x(U.lattice());
   mdp_complex sum=0;
   for(int i=0; i<length; i++) {
     if(i==0) forallsites(x) psi(x)=U(x,d[i][0],d[i][1]);
     else forallsites(x) psi(x)*=U(x,d[i][0],d[i][1]);
     if(i<length-1) psi.shift(d[i][0],d[i][1]);
     else forallsites(x) sum+=trace(psi(x));
   }
   return sum/(U.lattice().nvol_gl*U.nc);
}
```

Note tha call to the shift operator defined so that

```
psi.shift(+1,mu)
```

assignes $\psi(x-\mu)$ to $\psi(x)$.

4.2 Chromo-electro-magnetic field

The Chromo-electro-magnetic field $P_{\mu\nu} = a^2 G_{\mu\nu}$ has its own class as a field of vectors of matrices but one never really needs to declare it since it is uniquely associated to a aguge field. Given a gauge field `U` just call:

```
compute_em_field(U);
```

and to obtain the plaquette $P_{\mu\nu}(x)$ just call

```
U.em(x,mu,nu)
```

Therefore the average plaquette could also have been computes as

```
mdp_real sum=0.0;
compute_em_field(U);
forallsites(x)
   for(int mu=0; mu<U.ndim; mu++)
      for(int nu=mu+1; nu<U.ndim; nu++)
         sum+=real(trace(U.em(x,mu,nu));
retur sum/(U.lattice().nvol_gl*U.ndim*(U.ndim-1)/2*U.nc);
```