QMCPACK SCALAR ESTIMATOR IMPLEMENTATION

Introduction: Life of a Specialized QMCHamiltonianBase

Almost all observables in QMCPACK are implemented as specialized derived classes of the QMCHamiltonianBase base class. Each observable is instantiated in HamiltonianFactory and added to QMCHamiltonian for tracking. QMCHamiltonian tracks two types of observables: main and auxiliary. Main observables contribute to the local energy. These observables are elements of the simulated Hamiltonian such as kinetic or potential energy. auxiliary observables are expectation values of matrix elements that do not contribute to the local energy. These Hamiltonians do not affect the dynamics of the simulation. In the code, the main observables are labeled by "physical" flag, the auxiliary observables have "physical" set to false.

Initialization

When an `<estimator type="est_type" name="est_name" other_stuff="value"/>` tag is present in the `<hamiltonian/>` section, it is first read by HamiltonianFactory. In general, the `type` of the estimator will determine which specialization of QMCHamiltonianBase to be instantiated, and a derived class with `myName="est_name"` will be constructed. Then, the put() method of this specific class will be called to read any other parameters in the `<estimator/>` xml node. Sometimes these parameters will be read by HamiltonianFactory instead, because it can access more objects than QMCHamiltonianBase.

Evaluate

After the observable class (derived class of QMCHamiltonianBase) is constructed and prepared (by the put() method), it is ready to be used in a QMCDriver. A QMCDriver will call `H.auxHevaluate(W,thisWalker)` after every accepted move, where H is the QMCHamiltonian that holds all main and auxiliary Hamiltonian elements, W is a MCWalkerConfiguration and thisWalker is a pointer to the current walker being worked on. As shown below, this function goes through each auxiliary Hamiltonian element and evaluate it using the current walker configuration. Under the hood, observables are calculated and dumped

to the main particle set's property list for later collection.

```
// In QMCHamiltonian.cpp
// This is more efficient.
// Only calculate auxH elements if moves are accepted.
void QMCHamiltonian::auxHevaluate(ParticleSet& P, Walker_t& ThisWalker)
{
#if !defined(REMOVE_TRACEMANAGER)
  collect_walker_traces(ThisWalker,P.current_step);
#endif
  for(int i=0; i<auxH.size(); ++i)
  {
    auxH[i]->setHistories(ThisWalker);
    RealType sink = auxH[i]->evaluate(P);
    auxH[i]->setObservables(Observables);
#if !defined(REMOVE_TRACEMANAGER)
    auxH[i]->collect_scalar_traces();
#endif
    auxH[i]->setParticlePropertyList(P.PropertyList,myIndex);
  }
}
```

For estimators that contribute to the local energy (main observables), the return value of evaluate() is used in accumulating the local energy. For auxiliary estimators, the return value is not used (`sink` local variable above), only the value of Value is recorded property lists by the setObservables() method as shown in the above code snippet. By default, the setObservables() method will transfer `auxH[i]->Value` to `P.PropertyList[auxH[i]->myIndex]`. The same property list is also kept by the particle set being moved by QMCDriver. This list is updated by `auxH[i]->setParticlePropertyList(P.PropertyList,myIndex)`, where myIndex is the starting index of space allocated to this specific auxiliary Hamiltonian in the property list kept by the target particle set P.

Collection

The actual statistics are collected within the QMCDriver, which owns an EstimatorManager object. This object (or a clone in the case of multithreading) will be registered with each mover it owns. For each mover (such as VMCUpdatePbyP derived from QMCUpdateBase), an accumulate() call is made, which by default, makes an accumulate(walkerset) call to the EstimatorManager it owns. Since each walker has a property set, EstimatorManager uses that local copy to calculate statistics. The EstimatorManager performs block averaging

and file I/O.

Single Scalar Estimator Implementation Guide

Almost all of the defaults can be utilized for a single scalar observable. With any luck, only the put() and evaluate() methods need to be implemented. As an example, this section will present a step-by-step guide to implement a `SpeciesKineticEnergy` estimator that calculates the kinetic energy of a specific species instead of the entire particle set. For example, a possible input to this estimator can be:

```
<estimator type="specieskinetic" name="ukinetic" group="u"/>
<estimator type="specieskinetic" name="dkinetic" group="d"/>.
```

This should create two extra columns in the scalar.dat file that contains the kinetic energy of the up and down electrons in two separate columns. If the estimator is properly implemented,then the sum of these two columns should be equal to the default `Kinetic` column.

Barebone

The first step is to create a barebone class structure for this simple scalar estimator. The goal is to be able to instantiate this scalar estimator with an xml node and have it print out a column of zeros in the scalar.dat file.

To achieve this, first create a header file "SpeciesKineticEnergy.h" in QMCHamiltonians folder, with only the required functions declared as follows:

```
// In SpeciesKineticEnergy.h
#ifndef QMCPLUSPLUS_SPECIESKINETICENERGY_H
#define QMCPLUSPLUS_SPECIESKINETICENERGY_H

#include <Particle/WalkerSetRef.h>
#include <QMCHamiltonians/QMCHamiltonianBase.h>

namespace qmcplusplus
{

class SpeciesKineticEnergy: public QMCHamiltonianBase
{
public:
```

```
  SpeciesKineticEnergy(ParticleSet& P):tpset(P){ };

  bool put(xmlNodePtr cur);          // read input xml node, required
  bool get(std::ostream& os) const; // class description, required

  Return_t evaluate(ParticleSet& P);
  inline Return_t evaluate(ParticleSet& P, std::vector<NonLocalData>& Txy)
  { // delegate responsity inline for speed
    return evaluate(P);
  }

  // pure virtual functions require overrider
  void resetTargetParticleSet(ParticleSet& P) { }                          // required
  QMCHamiltonianBase* makeClone(ParticleSet& qp, TrialWaveFunction& psi); // required

private:
  ParticleSet& tpset;

}; // SpeciesKineticEnergy

} // namespace qmcplusplus
#endif
```

Notice a local reference `tpset` to the target particle set `P` is saved in the constructor. The target particle set carries much information useful for calculating observables. Next, make "SpeciesKineticEnergy.cpp" and make vacuous definitions

```
// In SpeciesKineticEnergy.cpp
#include <QMCHamiltonians/SpeciesKineticEnergy.h>
namespace qmcplusplus
{

bool SpeciesKineticEnergy::put(xmlNodePtr cur)
{
  return true;
}

bool SpeciesKineticEnergy::get(std::ostream& os) const
{
  return true;
}

SpeciesKineticEnergy::Return_t SpeciesKineticEnergy::evaluate(ParticleSet& P)
{
  Value = 0.0;
```

```
  return Value;
}

QMCHamiltonianBase* SpeciesKineticEnergy::makeClone(ParticleSet& qp, TrialWaveFunction& psi)
{
  // no local array allocated, default constructor should be enough
  return new SpeciesKineticEnergy(*this);
}

} // namespace qmcplusplus
```

Now, head over to Hamiltonian Factory and instantiate this observable if an xml node is found requesting it. Look for "gofr" in HamiltonianFactory.cpp, for example, and follow the if block

```
// In HamiltonianFactory.cpp
#include <QMCHamiltonians/SpeciesKineticEnergy.h>
else if(potType =="specieskinetic")
{
  SpeciesKineticEnergy* apot = new SpeciesKineticEnergy(*target_particle_set);
  apot->put(cur);
  targetH->addOperator(apot,potName,false);
}
```

The last argument of addOperator(), i.e. the `false` flag, is **crucial**. This tells QMCPACK that the observable we implemented is not a physical Hamiltonian, thus it will not contribute to the local energy. Changes to the local energy will alter the dynamics of the simulation. Finally, add "SpeciesKineticEnergy.cpp" to HAMSRCS in "CMakeLists.txt" located in the QMCHamiltonians folder. Now, recompile QMCPACK and run it on an input that requests `<estimator type="specieskinetic" name="ukinetic"/>` in the `hamiltonian` block. A columns of zeros should appear in the scalar.dat file under the name "ukinetic".

Evaluate

The evaluate() method is where we perform the calculation of the desired observable. In a first iteration, we will simply hard-code the name and mass of the particles

```
// In SpeciesKineticEnergy.cpp
#include <QMCHamiltonians/BareKineticEnergy.h> // laplaician() defined here
SpeciesKineticEnergy::Return_t SpeciesKineticEnergy::evaluate(ParticleSet& P)
{
  std::string group="u";
```

```
  RealType minus_over_2m = -0.5;

  SpeciesSet& tspecies(P.getSpeciesSet());

  Value = 0.0;
  for (int iat=0; iat<P.getTotalNum(); iat++)
  {
    if (tspecies.speciesName[ P.GroupID(iat) ] == group)
    {
      Value += minus_over_2m*laplacian(P.G[iat],P.L[iat]);
    }
  }
  return Value;

  // Kinetic column has:
  // Value = -0.5*( Dot(P.G,P.G) + Sum(P.L) );
}
```

Voila, you should now be able to compile QMCPACK, rerun, and see that the values in the "ukinetic" column are no longer zero. Now, the only task left to make this basic observable complete is to read in the extra parameters instead of hard-coding them.

Parse Extra Input

The preferred method to parse extra input parameters in the xml node is to implement the put() function of our specific observable. Suppose we wish to read in a single string that tells us whether to record the kinetic energy of the up electron (group="u") or the down electron (group="d"). This is easily achievable using the OhmmsAttributeSet class

```
// In SpeciesKineticEnergy.cpp
#include <OhmmsData/AttributeSet.h>
bool SpeciesKineticEnergy::put(xmlNodePtr cur)
{
  // read in extra parameter "group"
  OhmmsAttributeSet attrib;
  attrib.add(group,"group");
  attrib.put(cur);

  // save mass of specified group of particles
  SpeciesSet& tspecies(tpset.getSpeciesSet());
  int group_id  = tspecies.findSpecies(group);
  int massind   = tspecies.getAttribute("mass");
  minus_over_2m = -1./(2.*tspecies(massind,group_id));
```

```
  return true;
}
```

where we may keep "group" and "minus_over_2m" as local variables to our specific class.

```
// In SpeciesKineticEnergy.h
private:
  ParticleSet& tpset;
  std::string  group;
  RealType minus_over_2m;
```

Notice, the above operations are made possible by the saved reference to the target particle set. Last but not least, compile and run a full example (i.e. a short DMC calculation) with the following xml nodes in your input

`<estimator type="specieskinetic" name="ukinetic" group="u"/>`

`<estimator type="specieskinetic" name="dkinetic" group="d"/>`.

Make sure the sum of the "ukinetic" and "dkinetic" columns is **exactly** the same as the Kinetic columns at **every block**.

For easy reference, the complete list of changes is summarized bellow

```
// In HamiltonianFactory.cpp
#include "QMCHamiltonians/SpeciesKineticEnergy.h"
else if(potType =="specieskinetic")
{
        SpeciesKineticEnergy* apot = new SpeciesKineticEnergy(*targetPtcl);
        apot->put(cur);
        targetH->addOperator(apot,potName,false);
}

// In SpeciesKineticEnergy.h
#include <Particle/WalkerSetRef.h>
#include <QMCHamiltonians/QMCHamiltonianBase.h>

namespace qmcplusplus
{

class SpeciesKineticEnergy: public QMCHamiltonianBase
{
public:

  SpeciesKineticEnergy(ParticleSet& P):tpset(P){ };

  // xml node is read by HamiltonianFactory, eg. the sum of following should be equivalent to Kinetic
  // <estimator name="ukinetic" type="specieskinetic" target="e" group="u"/>
```

```
  // <estimator name="dkinetic" type="specieskinetic" target="e" group="d"/>
  bool put(xmlNodePtr cur);          // read input xml node, required
  bool get(std::ostream& os) const; // class description, required

  Return_t evaluate(ParticleSet& P);
  inline Return_t evaluate(ParticleSet& P, std::vector<NonLocalData>& Txy)
  { // delegate responsity inline for speed
    return evaluate(P);
  }

  // pure virtual functions require overrider
  void resetTargetParticleSet(ParticleSet& P) { }                          // required
  QMCHamiltonianBase* makeClone(ParticleSet& qp, TrialWaveFunction& psi); // required

private:
  ParticleSet&       tpset; // reference to target particle set
  std::string        group; // name of species to track
  RealType   minus_over_2m; // mass of the species !! assume same mass
  // for multiple species, simply initialize multiple estimators

}; // SpeciesKineticEnergy

} // namespace qmcplusplus
#endif

// In SpeciesKineticEnergy.cpp
#include <QMCHamiltonians/SpeciesKineticEnergy.h>
#include <QMCHamiltonians/BareKineticEnergy.h> // laplaician() defined here
#include <OhmmsData/AttributeSet.h>

namespace qmcplusplus
{

bool SpeciesKineticEnergy::put(xmlNodePtr cur)
{
  // read in extra parameter "group"
  OhmmsAttributeSet attrib;
  attrib.add(group,"group");
  attrib.put(cur);

  // save mass of specified group of particles
  int group_id  = tspecies.findSpecies(group);
  int massind   = tspecies.getAttribute("mass");
  minus_over_2m = -1./(2.*tspecies(massind,group_id));

  return true;
}
```

```
bool SpeciesKineticEnergy::get(std::ostream& os) const
{ // class description
  os << "SpeciesKineticEnergy: " << myName << " for species " << group;
  return true;
}

SpeciesKineticEnergy::Return_t SpeciesKineticEnergy::evaluate(ParticleSet& P)
{
  Value = 0.0;
  for (int iat=0; iat<P.getTotalNum(); iat++)
  {
    if (tspecies.speciesName[ P.GroupID(iat) ] == group)
    {
      Value += minus_over_2m*laplacian(P.G[iat],P.L[iat]);
    }
  }
  return Value;
}

QMCHamiltonianBase* SpeciesKineticEnergy::makeClone(ParticleSet& qp, TrialWaveFunction& psi)
{ //default constructor
  return new SpeciesKineticEnergy(*this);
}

} // namespace qmcplusplus
```

Multiple Scalars

It is fairly straight-forward to created more than one column in the scalar.dat file with a single observable class. For example, if we want a single SpeciesKineticEnergy estimator to record the kinetic energies of all species in the target particle set simultaneously, we only have to write two new methods: addObservables() and setObservables(), then tweak the behavior of evaluate(). Firstly, we will have to override the default behavior of addObservables() to make room for more than one column in the scalar.dat file as follows

```
// In SpeciesKineticEnergy.cpp
void SpeciesKineticEnergy::addObservables(PropertySetType& plist, BufferType& collectables)
{
  myIndex = plist.size();
  for (int ispec=0; ispec<num_species; ispec++)
  { // make columns named "$myName_u", "$myName_d" etc.
    plist.add(myName + "_" + species_names[ispec]);
```

```
  }
}
```

where "num_species" and "species_name" can be local variables initialized in the constructor. We should also initialize some local arrays to hold temporary data

```
// In SpeciesKineticEnergy.h
private:
  int num_species;
  std::vector<std::string> species_names;
  std::vector<RealType> species_kinetic,vec_minus_over_2m;

// In SpeciesKineticEnergy.cpp
SpeciesKineticEnergy::SpeciesKineticEnergy(ParticleSet& P):tpset(P)
{
  SpeciesSet& tspecies(P.getSpeciesSet());
  int massind = tspecies.getAttribute("mass");

  num_species = tspecies.size();
  species_kinetic.resize(num_species);
  vec_minus_over_2m.resize(num_species);
  species_names.resize(num_species);
  for (int ispec=0; ispec<num_species; ispec++)
  {
    species_names[ispec] = tspecies.speciesName[ispec];
    vec_minus_over_2m[ispec] = -1./(2.*tspecies(massind,ispec));
  }
}
```

Next, we need to override the default behavior of setObservables() to transfer multiple values to the property list kept by the main particle set, which eventually go into the scalar.dat file

```
// In SpeciesKineticEnergy.cpp
void SpeciesKineticEnergy::setObservables(PropertySetType& plist)
{ // slots in plist must be allocated by addObservables() first
  copy(species_kinetic.begin(),species_kinetic.end(),plist.begin()+myIndex);
}
```

Finally, we need to change the behavior of evaluate() to fill the local vector "species_kinetic" with appropriate observable values

```
SpeciesKineticEnergy::Return_t SpeciesKineticEnergy::evaluate(ParticleSet& P)
{
  std::fill(species_kinetic.begin(),species_kinetic.end(),0.0);

  for (int iat=0; iat<P.getTotalNum(); iat++)
  {
    int ispec = P.GroupID(iat);
```

```
    species_kinetic[ispec] += vec_minus_over_2m[ispec]*laplacian(P.G[iat],P.L[iat]);
  }

  Value = 0.0; // Value is no longer used
  return Value;
}
```

That's it! SpeciesKineticEnergy estimator no longer needs the "group" input and will automatically output the kinetic energy of every species in the target particle set in multiple columns. You should now be able to run with `<estimator type="specieskinetic" name="skinetic"/>` and check that the sum of all columns that starts with "skinetic" is equal to the default "Kinetic" column.

HDF5 Output

If we desire an observable that will output hundreds of scalars per simulation step (eg. SkEstimator), then it is preferred to output to the stat.h5 file instead of the scalar.dat file for better organization. A large chunk of data to be registered in the stat.h5 file is called a "Collectable" in QMCPACK. In particular, if a QMCHamiltonianBase object is initialized with `UpdateMode.set(COLLECTABLE,1)` then the "Collectables" object carried by the main particle set will be processed and written to the stat.h5 file, where "UpdateMode" is a bit set (i.e. a collection of flags) with the following enumeration

```
// In QMCHamiltonianBase.h
///enum for UpdateMode
enum {PRIMARY=0,
  OPTIMIZABLE=1,
  RATIOUPDATE=2,
  PHYSICAL=3,
  COLLECTABLE=4,
  NONLOCAL=5,
  VIRTUALMOVES=6
};
```

As a simple example, to put the two columns we produced in the previous section into the stat.h5 file, we will first need to declare that our observable uses "Collectables"

```
// In constructor add:
hdf5_out = true;
UpdateMode.set(COLLECTABLE,1);
```

Then make some room in the "Collectables" object carried by the target particle set.

```
// In addObservables(PropertySetType& plist, BufferType& collectables) add:
if (hdf5_out)
{
  h5_index = collectables.size();
  std::vector<RealType> tmp(num_species);
  collectables.add(tmp.begin(),tmp.end());
}
```

Next make some room in the stat.h5 file by overriding the registerCollectables() method

```
// In SpeciesKineticEnergy.cpp
void SpeciesKineticEnergy::registerCollectables(std::vector<observable_helper*>& h5desc, hid_t gid) const
{
  if (hdf5_out)
  {
    std::vector<int> ndim(1,num_species);
    observable_helper* h5o=new observable_helper(myName);
    h5o->set_dimensions(ndim,h5_index);
    h5o->open(gid);
    h5desc.push_back(h5o);
  }
}
```

Finally, edit evaluate() to use the space in the "Collectables" object.

```
// In SpeciesKineticEnergy.cpp
SpeciesKineticEnergy::Return_t SpeciesKineticEnergy::evaluate(ParticleSet& P)
{
  RealType wgt = tWalker->Weight; // MUST explicitly use DMC weights in Collectables!
  std::fill(species_kinetic.begin(),species_kinetic.end(),0.0);

  for (int iat=0; iat<P.getTotalNum(); iat++)
  {
    int ispec = P.GroupID(iat);
    species_kinetic[ispec] += vec_minus_over_2m[ispec]*laplacian(P.G[iat],P.L[iat]);
    P.Collectables[h5_index + ispec] += vec_minus_over_2m[ispec]*laplacian(P.G[iat],P.L[iat])*wgt;
  }

  Value = 0.0; // Value is no longer used
  return Value;
}
```

That should be it. There should now be a new entry in the stat.h5 file containing the same columns of data as the scalar.dat file. After this check, we should clean up the code by

1. make "hdf5_out" and input flag by editing the put() method
2. disable output to scalar.dat when "hdf5_out" flag is on