LibMints Overview

Dr. Justin M. Turney

Friday, January 24, 2014

LibMints

Provides an interface to compute one- and two-electron integrals with arbitrary angular momentum and substantial flexibility.

With minimal effort rapid implementation is possible, particularly with DF/RI methods.

LibMints

LibMints provides a number of C++ classes to the programmer.

For example, fundamental information about the molecular system — atomic coordinates, atomic numbers, symmetry, etc. — is encapsulated in a class called `Molecule`.

LibMints

A basis set class facilitates the use of multiple basis sets; this is of great utility when dealing with auxiliary basis set methods, such as R12 and DF/RI.

The various types of one- and two-electron integrals that can be computed are all available through a simple, user-friendly interface that is common to all integral types.

Synopsis

- One of the core libraries of Psi4
- Responsible for...
  - managing molecular geometry
  - loading of basis sets
  - computation of integrals
  - providing rudimentary matrix classes
- Interfaces to LIBINT and LIBERD available.

One-Electron Integrals Provided

Well Known One-Electron Integrals

OverlapInt $(\mathbf{a}||\mathbf{b})$
KineticInt $(\mathbf{a}|-\frac{1}{2}\nabla^2|\mathbf{b})$
PotentialInt $(\mathbf{a}|\frac{1}{|\mathbf{r}-\mathbf{C}|}|\mathbf{b})$

Property Integrals

AngularMomentumInt
DipoleInt
ElectricFieldInt
ElectrostaticInt
MultipoleInt $(\mathbf{a}|\mathfrak{M}(\mu)|\mathbf{b})$
NablaInt
PseudospectralInt
QuadrupoleInt

Two-Electron Integrals Provided

ERI
ErfComplementERI
ErfERI

Explicitly Correlated Integrals
F12
F12DoubleCommutator
F12G12
F12Squared

CorrelationFactor
FittedSlaterCorrelationFactor

Obtaining the molecule from Psi4

Psi4 automatically manages the active molecule for us. To access use the following code.

```
1 boost::shared_ptr<Molecule> molecule =
2         Process::environment.molecule();
```

Notice

From here on out, I will no longer say `boost::`
e.g. `boost::shared_ptr<>`

You will either need to add `boost::` yourself, or include the following near the top of your source file:

```
1 using namespace boost;
```

Loading a basis set

First, a basis set parser object needs to be created.

Second, using your molecule load the necessary basis sets.

```
1 shared_ptr<BasisSetParser>
2     parser(new Gaussian94BasisSetParser());
3
4 shared_ptr<BasisSet>
5     aoBasis =
6         BasisSet::construct(parser, molecule, "BASIS");
```

Information available from a `BasisSet`

`BasisSet` contains no symmetry information.

- Number of shells: `nshell()`
- Number of atomic orbitals: `nao()`
- Number of basis functions: `nbf()`

You can also obtain the data for individual shells using

```
shell(int shell)
```

or

```
shell(int center, int shell)
```

Obtaining symmetry information for a `BasisSet`

Symmetry information is obtained through an `SOBasisSet` object. To construct one, do the following:

```
1 shared_ptr<SOBasisSet>
2     soBasis(new SOBasisSet(aoBasis, integral));
```

Constructing an integral factory is discussed shortly.

Basis set symmetry information

You obtain symmetry blocking information from an `SOBasisSet` object:

```
1 const Dimension dimension = soBasis->dimension();
```

For example, for STO-3G H_2O:

`aoBasis->nbf()` $\rightarrow$ 7

`soBasis->dimension()` $\rightarrow$ [4, 0, 1, 2]

A `Dimension` object is a helper object for handling matrix dimensionality.

Creating an integral factory

A "factory" is responsible for creating objects.

An "integral factory" is responsible for creating integral objects.

```
1 shared_ptr<IntegralFactory>
2     integral(
3         new IntegralFactory
4             (aoBasis, aoBasis, aoBasis, aoBasis)
5     );
```

Creating one-electron integral objects

Once you have an integral factory creating one-electron integral objects is easy.

```
1 shared_ptr<OneBodySOInt>
2     sOBI(integral->so_overlap());
3
4 shared_ptr<OneBodySOInt>
5     tOBI(integral->so_kinetic());
6
7 shared_ptr<OneBodySOInt>
8     vOBI(integral->so_potential());
```

Creating one-electron integral objects

When you're computing one-electron integrals you'll need a matrix to store the data in. LibMints provides a `MatrixFactory` class to assist you.

Create a matrix factory to help you out and initialize it with the dimensions you obtained from the `SOBasisSet` object:

```
1 shared_ptr<MatrixFactory> factory(new MatrixFactory);
2 factory->init_with(dimension, dimension);
```

And then create a matrix:

```
1 SharedMatrix sMat = factory->create_shared_matrix("Overlap");
```

Compute the overlap integral matrix

Have our previously created integral object compute the integral:

```
1 // Compute all overlap integrals and store them into sMat
2 sOBI->compute(sMat);
3
4 // Print the matrix
5 sMat->print();
```

One-electron integrals available

Every entry below is a function available in the `IntegralFactory` object. Each returns an object that derives from either a `OneBodyAOInt` or `OneBodySOInt` class that provides a consistent interface for computing integrals.

AO version	SO version
ao_overlap	so_overlap
ao_kinetic	so_kinetic
ao_potential	so_potential
ao_pseudospectral	so_pseudospectral
ao_dipole	so_dipole
ao_quadrupole	so_quadrupole
ao_multipole	so_multipole
ao_traceless_quadrupole	so_traceless_quadrupole
ao_nabla	so_nabla
ao_angular_momentum	so_angular_momentum

Two-electron integrals

Psi4 provides the ability to compute AO (no symmetry) and SO (with symmetry) two-electron integrals.

The way to access them is different so let's cover the AO version first.

AO two-electron integrals

Using the integral factory create an ERI object.

```
1 // Now, the two-electron integrals
2 shared_ptr<TwoBodyAOInt> eri(integral->eri());
```

We need access to the buffer that the object will put the integrals into:

```
3 // buffer will hold the integrals for each shell, as they're computed
4 const double *buffer = eri->buffer();
```

AO two-electron integrals

LibMints provides a couple of handy objects for iterating through only the unique integrals.

```
5  // The iterator conveniently lets us iterate over functions within shells
6  AOShellCombinationsIterator shellIter = integral->shells_iterator();
7
8  // Let's keep count how many integrals we computed.
9  unsigned long count=0;
10
11 // Use the iterator to loop through the unique shell combinations.
12 for (shellIter.first(); shellIter.is_done() == false; shellIter.next()) {
```

AO two-electron integrals

Now to compute the integrals.

```
11 // Use the iterator to loop through the unique shell combinations.
12 for (shellIter.first(); shellIter.is_done() == false; shellIter.next()) {
13
14     // Compute quartet
15     eri->compute_shell(shellIter);
```

AO two-electron integrals

Work our way through the integrals that we just computed.

```
16      // From the quartet get all the integrals
17      AOIntegralsIterator intIter = shellIter.integrals_iterator();
18      for (intIter.first(); intIter.is_done() == false; intIter.next()) {
19          int p = intIter.i(); int q = intIter.j();
20          int r = intIter.k(); int s = intIter.l();
21          fprintf(outfile, "\t(%2d %2d | %2d %2d) = %20.15f\n",
22              p, q, r, s, buffer[intIter.index()]);
23          ++count;
24      }
25  }
26  fprintf(outfile, "\n\tThere are %d unique integrals\n\n", count);
```

AO two-electron integrals

That little bit of code was enough to compute all the AO two-electron integrals.

What you do with them is up to you.

SO two-electron integrals

The technique used to construct the symmetrized two-electron integrals is different from that of the AO.

We use the C++ technique known as functors.

A *functor* (or *function object*) is an C++ class that acts like a function. Functors can be called using the familiar function call syntax, and can yield values and accept parameters just like regular functions.

SO two-electron integrals

Let's first define out functor. Our functor will simply print the integrals out just like our AO integral code before.

```
1  class ERIPrinter
2  {
3  public:
4      void operator() (int pabs, int qabs, int rabs, int sabs,
5                       int pirrep, int pso,
6                       int qirrep, int qso,
7                       int rirrep, int rso,
8                       int sirrep, int sso,
9                       double value)
10     {
11         fprintf(outfile, "\t(%2d %2d | %2d %2d) = %20.10lf\n",
12                 pabs, qabs, rabs, sabs, value);
13     }
14 };
```

SO two-electron integrals

Computing SO two-electron integrals takes only a few steps.

First obtain an AO integral object:

```
1 // 1. Obtain an object that knows how to compute two-electron AO
2 // integrals.
3 shared_ptr<TwoBodyAOInt> tb(integral->eri());
```

Create a `TwoBodySOInt` object with your AO integral object.

```
4 // 2. Create an object that knows how to convert any two-body AO
5 // integral to SO.
6 shared_ptr<TwoBodySOInt> eri(new TwoBodySOInt(tb, integral));
```

SO two-electron integrals

Initialize your functor.

```
7  // 3. We to create an instance of our ERIPrinter
8  ERIPrinter printer;
```

Iterator through the shells producing unique SO integrals.

```
9  // 4. Create an SOShellCombintationsIterator to step through the
10 // necessary combinations
11 SOShellCombinationsIterator shellIter(soBasis, soBasis, soBasis, soBasis);
12 for (shellIter.first(); shellIter.is_done() == false; shellIter.next()) {
13     // 5. Call the TwoBodySOInt object to compute integrals giving
14     // it the
15     // instance to our functor.
16     eri->compute_shell(shellIter, printer);
17 }
```

Two-electron integrals available

Every entry below is a function available in the `IntegralFactory` object. Each returns an object that derives from the `TwoBodyAOInt` class and provides a consistent interface for computing integrals.

AO version
eri
erd_eri ← Only here for testing purposes.
erf_eri
erf_complement_eri
f12
f12_squared
f12g12
f12_double_commutator

We've seen examples of using LibMints to obtain the current molecule from Psi4 and computing one- and two- electron integrals.

What else can LibMints do?

What else can LibMints do?

- Construct Cartesian displacement SALCs used in finite difference calculations.
- Point group and character tables.
- Integral derivatives.
- Property analysis.
- Also provides a `Wavefunction` class for theory modules to derive from.
- Various writers for interfacing with other programs. For example, Molden and NBO.

Interface with Python

Almost all objects in LibMints are interfaced with Python. This allows for methods for some pretty extraordinary input files. There are a few examples of SCF codes being written in Psi4 input files.

The End.