The PSI4 Programmer's Manual

T. Daniel Crawford,[a] C. David Sherrill,[b] Edward F. Valeev,[a]
Justin T. Fermann,[c] C. Brian Kellogg,[c], Andrew C. Simmonett[c]
and Justin M. Turney[c]

[a] *Department of Chemistry, Virginia Tech, Blacksburg, Virginia 24061*

[b] *Center for Computational Molecular Science and Technology, Georgia Institute of Technology, Atlanta, Georgia 30332-0400*

[c] *Center for Computational Quantum Chemistry, University of Georgia, Athens, Georgia 30602-2525*

PSI4 Version: 4.0.0-alpha
Created on: July 20, 2011
Report bugs to: psicode@users.sourceforge.net

Contents

1 Introduction **3**

1.1 introduction 3

1.2 History 3

1.3 License 4

2 Obtaining PSI4 **4**

2.1 PSI4 SVN Policies: Which Branch Should I Use? 5

2.2 Checking in altered PSI4 binaries or libraries 7

2.3 Adding entirely new code to the main PSI4 repository 8

2.4 Updating checked out code 9

2.5 Removing code from the repository 10

2.6 Checking out older versions of the code 10

2.7 Examining the revision history 11

2.8 The structure of the PSI4 Source Tree 11

3 Python and PSI4 **12**

4 Testing **13**

5 Sample Codes **14**

5.1 Integrals 14

5.2 MP2 With Density Fitting 19

1 Introduction

1.1 introduction

The purpose of this manual is to provide a reasonably detailed overview of the source code and programming philosophy of PSI4, such that programmers interested in contributing to the code will have an easier task. Section 2 gives a succint explanation of the steps required to obtain the source code from the main repository at Virginia Tech. (Installation instructions are given separately in the installation manual or in $PSI4/INSTALL.) **??** offers advice on appropriate programming style for PSI4 code, and section **??** describes the structure of the package's Makefiles. Section The appendices provide important reference material, including the currently accepted PSI4 citation and format information for some of the most important text files used by PSI4 modules.

There are many examples included in this document to provide sample input files and source files; these can be found in ASCII form in the PSI4 source itself. Each included file has a path, which is relative to $PSI4/doc/progman, as its title and this is where the unformatted file can be found. The examples described herein can even be compiled from the directories in which the source files are found.

1.2 History

The PSI suite of *ab initio* quantum chemistry programs is the result of an ongoing attempt by a cadre of graduate students, postdoctoral associates, and professors to produce code that is efficient but also easy to extend to new theoretical methods. Significant effort has been devoted to the development of libraries which are robust and easy to use. Some of the earliest contributions to what is now referred to as "PSI" include a direct configuration interaction (CI) program (Robert Lucchese, 1976, now at Texas A&M), the well-known graphical unitary group CI program (Bernie Brooks, 1977-78, now at N.I.H.), and the original integrals code (Russ Pitzer, 1978, now at Ohio State). From 1978-1987, the package was know as the BERKELEY suite, and after the Schaefer group moved to the Center for Computational Quantum Chemistry at the University of Georgia, the package was renamed PSI. Thanks primarily to the efforts of Curt Janssen (Sandia Labs, Livermore) and Ed Seidl (LLNL), the package was ported to UNIX systems, and substantially improved with new input formats and a C-based I/O system.

Beginning in 1999, an extensive effort was begun to develop PSI3 — a PSI suite with a completely new face. As a result of this effort, all of the legacy Fortran code was removed, and everything was rewritten in C and C++, including new integral/derivative integral, coupled cluster, and CI codes. In addition, new I/O libraries have been added, as well as an improved checkpoint file structure and greater automation of typical tasks such as geometry optimization and frequency analysis. The package has the capability to determine wavefunctions, energies, analytic gradients, and various molecular properties based on a variety of theories, including spin-restricted, spin-unrestricted, and restricted open-shell Hartree-Fock (RHF, UHF, and ROHF); configuration interaction (CI) (including a variety

of multireference CI's and full CI); coupled-cluster (CC) including CC with variationaly optimized orbitals; second-order Møller-Plesset perturbation theory (MPPT) including explicitly correlated second-order Møller-Plesset energy (MP2-R12); and complete-active-space self-consistent field (CASSCF) theory. By January 2008, all of the C code in `PSI3` was converted to C++ to enable a path toward more object-oriented design and a single-excecutable framework that will facilitate code reuse and ease efforts at parallelization. At this same time, all of the legacy I/O routines from `PSI2` were removed, greatly streamlining the `libciomr.a` library.

1.3 License

Mention the GPL and development policies...

2 Obtaining `PSI4`

The subversion control system (SVN) (`subversion.tigris.org`) provides a convenient means by which programmers may obtain the latest (or any previous) version of the `PSI4` source from the main repository or a branch version, add new code to the source tree or modify existing `PSI4` modules, and then make changes and additions available to other programmers by checking the modifications back into the main repository. SVN also provides a "safety net" in that any erroneous modifications to the code may be easily removed once they have been identified. This section describes how to use SVN to access and modify the `PSI4` source code. (Note that compilation and installation instructions are given in a separate document.)

The main repository for the `PSI4` Source code is currently maintained by the Crawford group at Virginia Tech. To check out the code, one must first obtain an SVN account by emailing `crawdad@vt.edu`. After you have a login-id and password, you are now ready to access the repository via a secure, SSL-based WebDAV connection, but first you must decide which version of the code you need.

The `PSI4` SVN repository contains three top-level directories:

- `trunk`: The main development area.
- `branches`: Release branches and private development branches are stored here.
- `tags`: Snapshots of the repository corresponding to public releases are stored here and should *never* be modified.

If you have a PSI4 SVN account, you can peruse these directories if you like by pointing web browser to:

`https://sirius.chem.vt.edu/svn/psi4/`

2.1 PSI4 SVN Policies: Which Branch Should I Use?

The PSI4 repository comprises a main trunk and several release branches. The branch you should use depends on the sort of work you plan for the codes:

1. For any piece of code already in the most recent release, bug fixes (defined as anything that doesn't add functionality — including documentation updates) should be made *only* on the most recent stable release branch.

2. The main trunk is reserved for development of new functionality. This allows us to keep new, possibly unstable code away from public access until the code is ready.

3. Code that you do not want to put into next major release of PSI4 should be put onto a separate branch off the main trunk. You will be solely responsible for maintenance of the new branch, so you should read the SVN manual before attempting this.

Fig. 1 provides a schematic of the SVN revision-control structure and branch labeling. Two release branches are shown, the current stable branch, named psi-3-4, and a planned future release, to be named psi-3-5. The tags on the branches indicate release shapshots, where bugs have been fixed and the code has been or will be exported for public distribution. The dotted lines in the figure indicate merge points: just prior to each public release, changes made to the code on the stable release branch will be merged into the main trunk.

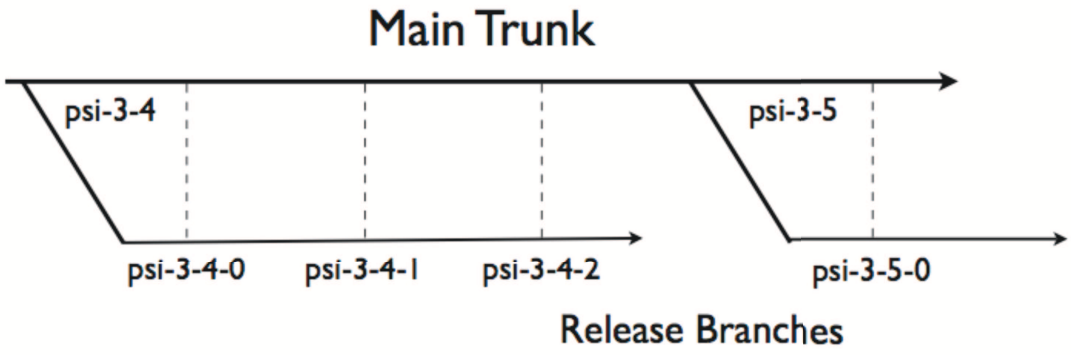

Figure 1: PSI4 SVN branch structure with examples of branch- and release-tag labelling.

A frequently encountered problem is what to do about bug fixes that are necessary for uninterrupted code development of the code on the main trunk. As Rule 1 of the above policy states, all bug fixes of the code already in the recent stable release must go on the corresponding branch, not on the main trunk. The next step depends on the severity of the bug:

1. If the bug fix is critical and potentially affects every developer of the code on the main trunk, then PSI4 administrators should be notified of the fix. If deemed necessary, appropriate steps to create a new patch release will be made. Once the next patch release is created then the bug fixes will be merged onto the main trunk. If the bug fix doesn't warrant an immediate new patch release, then you can incorporate the bug fix into your local copy of the main trunk code manually or using SVN merge features. This will allow you to continue development until next patch release is created and the bug fix is incorporated into the main trunk code in the repository. However you should *never* merge such changes into the main trunk yourself.

2. If the bug fix is not critical (e.g. a documentation update/fix), then you should wait until next patch release when it will be merged into the main trunk automatically.

The following are some of the most commonly used SVN commands for checking out and updating working copies of the PSI4 source code.

• To checkout a working copy of the head of the main trunk:

```
svn co https://sirius.chem.vt.edu/svn/psi4/trunk/ psi4
```

• To check out a working copy of the head of a specific release branch, e.g., the branch labelled psi-4-0:

```
svn co https://sirius.chem.vt.edu/svn/PSI4/branches/psi-4-0 psi4
```

Note that subsequent `svn update` commands in this working copy will provide updates only on the chosen branch. Note also that after you have checked out a fresh working copy of the code you must run the `autoconf` command to generate a `configure` script for building the code. (See the installation manual for configuration, compilation, and testing instructions.)

For each of the above commands, the working copy of your code will be placed in the directory psi4, regardless of your choice of branch. In this manual, we will refer to this directory from now on as $PSI4. Subsequent SVN commands are usually run within this top-level directory.

• To update your current working copy to include the latest revisions:

```
svn update
```

Notes: (a) This will update only the revisions on your current branch; (b) The old `-d` and `-P` flags required by CVS are not necessary with SVN.

• To convert your working copy to the head of a specific branch:

```
svn switch https://sirius.chem.vt.edu/svn/PSI4/branches/psi-4-0
```

• To convert your working copy to the head of the main trunk:

```
svn switch https://sirius.chem.vt.edu/svn/psi4/trunk/
```

• To find out what branch your working copy is on, run this in your top-level PSI4 source directory:

```
svn info | grep URL
```

This will return the SVN directory from which your working copy was taken, e.g.,

URL: `https://sirius.chem.vt.edu/svn/PSI4/branches/psi-4-0`

Some words of advice:

1. Most SVN commands are reasonably safe,
2. Unlike CVS, you shouldn't use `svn update` to see the status of your working copy. With SVN you should use `svn status` to see if you've modified any files or directories. If you want a direct comparison with the repository, you should use `svn status -u`.
3. Read the SVN manual. Seriously.

   `http://svnbook.red-bean.com/`

4. If you're about to start some significant development or bug-fixes, first update your working copy to the latest version on your branch. In addition, if you do development over a long period of time (say weeks to months) on a specific module or modules, be sure to run a `svn status -u` occasionally. In can be *very* frustrating to try to check in lots of changes, only to find out that the `PSI4` has changed dramatically since your last update.

2.2 Checking in altered `PSI4` binaries or libraries

If you have changes to Psi binaries or libraries which already exist, one of two series of steps is necessary to check these changes in to the main repository. The first series may be followed if all changes have been made only to files which already exist in the current version. The second series should be followed if new files must be added to the code in the repository.

- No new files need to be added to the repository. We will use `libciomr` as an example.
  1. `cd $PSI4/src/lib/libciomr`
  2. `svn ci -m ‘‘Put comments here.’’`
- New files must be added to the repository. Again, we use `libciomr` as an example. Suppose the new file is named `great_code.cc` .
  1. `cd $PSI4/src/lib/libciomr`
  2. `svn add great_code.cc`
  3. `svn ci -m ‘‘Put comments here.’’`

The `svn ci` command in both of these sequences will examine all of the code in the current `libciomr` directory against the current version of the code in the main repository. Any files which have been altered (and for which no conflicts with newer versions exist!) will be identified and checked in to the main repository (as well as the new file in the second situation).

SVN requires that you include a comment on your changes. However, unlike CVS, SVN prefers that you put your comments on the command-line rather than editing a text file. I prefer the CVS way, but this is a minor pain compared to all the advantages of SVN, in my opinion.

2.3 Adding entirely new code to the main PSI4 repository

If the programmer is adding a new executable module or library to the PSI4 repository, a number of important conventions should be followed:

1. Since such changes almost always involve additional functionality, new modules or libraries should be added only on the main SVN trunk. See section 2.1 for additional information.

2. The directory containing the new code should be given a name that matches the name of the installed code (e.g. if the code will be installed as `newcode`, the directory containing the code should be named `newcode`). New executable modules must be placed in `$PSI4/src/bin` and libraries in `$PSI4/src/lib` of the user's working copy.

3. The Makefile should be converted to an input file for the configure script (`Makefile.in` — see any of the current PSI4 binaries for an example) and should follow the conventions set up in all of the current PSI4 `Makefiles`. This includes use of `MakeVars` and `MakeRules`.

4. New binaries should be added to the list contained in `$PSI4/src/bin/Makefile.in` so that they will be compiled automatically when a full compilation of the PSI4 distribution occurs. This step is included in the sequence below.

5. A documentation page should be included with the new code (see section **??** for more information). As a general rule, if the code is not ready to have a documentation page, it is not ready to be installed in PSI4.

6. The `configure.ac` file must be altered so that users may check out copies of the new code and so that the `configure` script will know to create the Makefile for the new code. These steps are included in the sequence below.

Assume the new code is an executable module and is named `great_code`. The directory containing the new code must contain only those files which are to be checked in to the repository! Then the following steps will check in a new piece of code to the main repository:

1. `cd $PSI4/src/bin`

2. `svn add great_code`

3. `svn ci -m ''Put comments here.''`

4. `cd $PSI4`
5. Edit `configure.ac` and add `great_code` to the list.
6. `svn ci configure.ac -m ‘‘Put comments here.’’`
7. `autoconf`
8. `cd $PSI4/src/bin`
9. Edit `Makefile.in` and add `great_code` to the list.
10. `svn ci Makefile.in -m ‘‘Put comments here.’’`

At this point, all of the code has been properly checked in, but you should also test to make sure that the code can be checked out by other programmers, and that it will compile correctly. The following steps will store your personal version of the code, check out the new code, and test-compile it:

1. `cd $PSI4/src/bin`
2. `mv great_code great_code.bak`
3. `cd $PSI4/..`
4. `svn update`
5. `cd $objdir`
6. `$PSI4/configure --prefix=$prefix`
7. `cd src/bin/great_code`
8. `make install`

(Note that $prefix and $objdir to the installation and compilation directories defined in the `PSI4` installation instructions.) Your original version of the code remains under `great_code.bak`, but should be no longer necessary if the above steps work. Note that it is necessary to re-run `configure` explicitly, instead of just running `config.status`, because the latter contains no information about the new code.

2.4 Updating checked out code

If the code in the main repository has been altered, other users' working copies will of course not automatically be updated. In general, it is only necessary to execute the following steps in order to completely update your working copy of the code:

1. `cd $PSI4`

2. `svn update`

This will examine each entry in your working copy and compare it to the most recent version in the main repository. When the file in the main repository is more recent, your version of the code will be updated. If you have made changes to your version, but the version in the main repository has not changed, the altered code will be identified to you with an "M". If you have made changes to your version of the code, and one or more newer versions have been updated in the main repository, SVN will examine the two versions and attempt to merge them – this process often reveals conflicts, however, and is sometimes unsuccessful. You will be notified of any conflicts that arise (labelled with a "C") and you must resolve them manually.

If new directories have been added to the repository, the update above will automatically add them to your working copy. However, you may need to re-run `autoconf` and configure (`$objdir/config.status --recheck` is a convenient command) to be able to build the new code.

2.5 Removing code from the repository

If alterations of libraries or binaries under Psi involves the deletion of source code files from the code, these must be explicitly removed through SVN.

The following steps will remove a source code file named `bad_code.F` from a binary module named `great_code`:

1. `cd $PSI4/src/bin/great_code`
2. `svn remove bad_code.F`
3. `svn ci -m ‘‘Put comments here.’’`

2.6 Checking out older versions of the code

It is sometimes necessary to check out older versions of a piece of code. Assume we wish to check out an old version of `detci`. If this is the case, the following steps will do this:

1. `cd $PSI4/src/bin/detci`
2. `svn co --revision {2002-02-17}`

This will check the main repository and provide you with the code as it stood exactly on February 17th, 2002.

2.7 Examining the revision history

It can be very useful to use cvs to see what recent changes have been made to the code. Anytime one checks in a new version of a file, SVN requires the user to provide comments on the changes with the `-m` flag. These comments go into a log information that may be easily accessed through SVN. To see what changes have been made recently to the file `detci.cc`, one would go into the `detci` source directory and type

```
svn log detci.cc
```

Checking the log files is a very useful way to see what recent changes might be causing new problems with the code.

2.8 The structure of the PSI4 Source Tree

Your working copy of the PSI4 source code includes a number of important subdirectories:

- `$PSI4/lib` – Source files for OS-independent "library" data. This includes the main basis set data file (`pbasis.dat`) and the PSI4 program execution control file (`psi.dat`), among others. These files are installed in `$prefix/share`.
- `$PSI4/include` – Source files for OS-independent header files, including `physconst.h` (whose contents should be obvious from its name), `psifiles.h`, and `ccfiles.h`, among others. These files are installed in $prefix/include.
- `$PSI4/src/util` – Source code for the utility program `tocprint`. (Note that the `tmpl` module is no longer used and will eventually disappear.)
- `$PSI4/src/lib` – Source code for the libraries, including `libpsio`, `libipv1`, `libchkpt`, etc. The include files from the library source are used directly during the compilation of PSI to avoid problems associated with incomplete installations. Some include files are architecture-dependent and go in an include subdirectory of the compilation (object) directory.
- `$PSI4/src/bin` – Source code for the executable modules.

After compilation and installation, the `$prefix` directory contains the executable codes and other necessary files. **NB:** The files in this area should never be directly modified; rather, the working copy should be modified and the PSI4 `Makefile` hierarchy should handle installation of any changes. The structure of the installation area is:

- `$prefix/bin` – The main executable directory. This directory must be in your path in order for the driver program, PSI4, to find the modules.
- `$prefix/lib` – The PSI4 code libraries. (NB: The description of PSI4 `Makefiles` later in this manual will explain how to use the libraries.)

- `$prefix/include` – Header files. These are not actually used during the compilation of PSI but are useful for inclusion by external programs because they are all in the same directory.
- `$prefix/share` – OS-independent data files, including basis set information. (Do not edit this file directly; any changes you make can be overwritten by subsequent `make` commands.)
- `$prefix/doc` – `PSI4` documentation, including installation, programmer, and user manuals.

3 Python and `PSI4`

One of the most significant changes introduced in version 4 was the use of Python. The input file is actually a Python script, which interacts with a Psi Python module to perform computations. In order for this to happen, the C++ binding must be known to Python; this is all done in the `$PSI4/src/bin/psi4/python.cc` file. For example, we have an SCF module, with the C++ signature `PsiReturnType cscf::cscf(Options &options);` To allow Python to use this, we first define a little wrapper function

```
double py_psi_scf()
{
    if (scf::scf(Process::environment.options) == Success)
        return Process::environment.globals["CURRENT ENERGY"];
    else
        return 0.0;
}
```

This does a couple of things to automate things a) it passes the default options object into SCF automatically, so that the user doesn't have to, and b) checks the return value, and will return the energy, which is posted to the globals map, back to Python. Note that this is C++ code, within `PSI4` so it is aware of all global objects, such as PSIO, Chkpt and Options. Now we have this simple function call, we can tell Python about it:

```
def("scf",  py_psi_scf);
```

This binds the keyword "scf" to the newly created wrapper function, allowing the user to type "scf()" in their Python input file to fire up the SCF module. Similarly, the user might want to be able to call `Molecule`'s member functions directly from Python. This can also be done easily:

```
class_<Molecule, shared_ptr<Molecule> >("Molecule").
        def("print_to_output", &Molecule::print).
        def("nuclear_repulsion_energy", &Molecule::nuclear_repulsion_energy);
```

This first defines the keyword `Molecule` to refer to the C++ `Molecule`; the `shared_ptr<Molecule>` keyword tells Python to store it as a shared pointer, which ensures that the object will not be deleted until both C++ and Python have no more references to it. The member functions to be bound are then specified by a chained sequence of `def` calls (note the periods), terminated by a semicolon. Then, if the user had defined a molecule called "water", they could print its geometry simply with the command `water.print_to_output()`.

Direct interaction with the Psi module from Python requires function calls that look like `PsiMod.call_some_function()`. This is not very friendly to your average user, so a preprocessor checks for known Psi syntax and turns it into valid Python, before handing it off for excecution. This preprocessor is purely Python, and lives in `$PSI4/lib/python/input.py`. For example, the following text

```
set scf {
    SCF_TYPE DIRECT
    BASIS cc-pVDZ
    RI_BASIS_SCF cc-pVDZ-HF
    guess core
}
```

is converted to the following text

```
PsiMod.set_default_options_for_module("SCF")
PsiMod.set_option("SCF_TYPE", "DIRECT")
PsiMod.set_option("BASIS", "cc-pVDZ")
PsiMod.set_option("RI_BASIS_SCF", "cc-pVDZ-HF")
PsiMod.set_option("GUESS", "core")
```

which can be handled by Python.

There are a number of other utilities, which are entirely Python, located in `$PSI4/lib/python`. These provide convenient functions to the user, such as `table`.

4 Testing

The `PSI4` test suite is designed to maximize code reuse and provide testing in $prefix before the `PSI4` executables have been installed. The configure script in $PSI4 will take all the necessary files in $PSI4/tests with the .in stub: Makefile.in, MakeRules.in, MakeVars.in, and runtest.pl.in, replace variables with system specific parameters, and copy/create the testing files and directories in $prefix/tests. The tests should be run in the object directory before installation.

If you have just added a new module for performing, say multireference coupled cluster, and you would like to add a test case to the current test suite, here is what you should do.

1. Copy one of the existing test case directories to an appropriately named directory for the new test case.

2. Create an appropriate input file for running the new module. Then, if your program produced the correct data, rename the output files to *.ref. Follow the convention of the existing test cases. Make sure you add a descriptive comment to the input file, stating what the calculation type is. Use the special comment marker "%!" to do this, so that the comment is inserted into the user's manual.

3. If the test case is small, add the directory name to the list in $PSI4/tests/Makefile.in. If the test is particularly tricky, see the psi_start or rhf-stab test cases as an example.

4. All the testing functionality is located in the perl library `runtest.pl.in`. If you are testing for a quantity that is not searched for currently, then add a function to the library following the format of the functions already available. If you have added functionality to the `PSI4` driver, make sure to update the appropriate functions in `runtest.pl.in`.

5. Add the location of the Makefile for the new test case to the configure script in $PSI4.

Please contact one of the authors of `PSI4` before making any major changes or if you have a problem adding a new test case. Remember, if all else fails, read the source code.

5 Sample Codes

In this chapter we demonstrate how to write code for `PSI4` using some simple examples. It can be quite daunting to develop code in an unfamiliar programming environment, so a good starting point is to write code that requires little knowledge of the existing code structure. In this regard, the modular nature of `PSI3` made it a perfect development platform, with each "module" existing as a standalone code. With the transition to a single executable paradigm in `PSI4`, developing new code is still a simple process; in fact the availability of library functions to perform most tasks makes it even easier to get started with programming in `PSI4`, as we will demonstrate.

5.1 Integrals

As noted previously, we want to start from a code that's not too tightly integrated with the `PSI4` code itself, so we begin with a `Makefile` that will allow us to write a standalone code that includes all requisite `PSI` libraries. We're going to write a small sample code that generates integrals, which involves two just two source files. We begin by defining a `Makefile` that will include all of the `PSI4` libraries and header files, so that we can take full advantage of the wide range of features implemented without having to worry about the details of their implementation.

sample-codes/integrals/Makefile

```
1 # Name of the executable you want to generate
2 TARGET = integrals
```

```
3
4  # List of source files to be compiled
5  CXXSRC = main.cc integrals.cc
6
7  # Location of your PSI4 source
8  psi_top_srcdir = /Users/andysim/programming/workspace/psi4
9
10 # Location of your PSI4 install, by default as listed
11 psi_top_objdir = /Users/andysim/programming/workspace/psi4_obj_debug
12
13 # Include the BLAS and LAPACK linear algebra libraries
14 BLAS   = -lblas
15 LAPACK = -llapack
16
17 # The name of the C++ compiler. Must match the compiler used to
18 # compile PSI4
19 CXX = g++
20
21 # C++ Compiler flags, in this case: no optimization and
22 # enable native GDB debugging, respectively
23 CXXFLAGS = -O0 -g $(INCLUDES)
24 #CXXFLAGS = -O2 $(INCLUDES)
25
26
27 #************ None of the following should be modified *********#
28 # Name of the object intermediates that make up the
29 # executable
30 OBJS = $(CXXSRC:%.cc=%.o)
31 # Include the PSI4 include files (this needs to be cleaned up)
32 INCLUDES = -I$(psi_top_srcdir)/include\
33 -I$(psi_top_objdir)/include\
34 -I$(psi_top_srcdir)/src/lib\
35 -I$(psi_top_objdir)/src/lib
36 # Include the PSI4 libraries
37 LIBS = -L$(psi_top_objdir)/lib -lPSI_mints -lPSI_ccenergy -lPSI_ccsort\
38 -lPSI_input3 -lPSI_cscf -lPSI_cints -lPSI_transqt2 -lPSI_psiclean -lPSI_dpd\
39 -lPSI_chkpt -lPSI_iwl -lPSI_qt -lPSI_psio -lPSI_ciomr -lPSI_ipv1\
40 -lPSI_options -lPSI_deriv -lPSI_int -lPSI_util -lPSI_parallel $(BLAS) $(LAPACK)
41 # Execute the compilation for the specified object, linking
42 # with all the libraries specified
43 $(TARGET): $(OBJS)
44     $(CXX) -o $(TARGET) $(OBJS) $(LIBS)
45
46 # Execute the compilation for all objects of the specified
47 # target
48 all:
49     $(TARGET)
50
51 # Erase all compiled intermediate files
52 clean:
53     rm -f $(OBJS) $(TARGET)
```

Only a few lines of this makefile need to be modified to utilize it for other programming projects; we'll concentrate on them. On the second line, we define the name of the executable to be generated, in this example we opt for the unimaginative title of `integrals`. Line 4 provides the list of source files that the project comprises; these will be detailed below. The top source directory for the `PSI4` installation and the top object directory (where `PSI4` was compiled) should be provided on lines 6 and 8, respectively. Lines 10 and 11 describe the flags needed to link in the `BLAS` and `LAPACK` libraries and might need a combination of "`-L folder_name`" and "`-l library_name`", depending on your system's setup. Finally, the compiler and flags are detailed on lines 12–17. It's a good idea to use the flags described

on line 16 for development; they speed up code compilation and provide lots of information for standard debugging tools. As noted in the `Makefile` itself, nothing below line 17 should require modification for any other `PSI4` project.

The `PSI4` driver program provides a lot of functionality that we forgo in writing a standalone code; this is instead emulated in the `main.cc` file, shown below.

sample-codes/integrals/main.cc

```
1  #include <libpsio/psio.h>
2  #include <libipv1/ip_lib.h>
3  #include <libciomr/libciomr.h>
4
5  #include "psi4-def.h"
6
7  using namespace psi;
8
9  namespace psi {
10     FILE *infile;
11
12     namespace integrals{
13         PsiReturnType integrals(Options &options, int argc, char **argv);
14     }
15
16     int
17     read_options(std::string name, Options &options)
18     {
19         ip_cwk_clear();
20         ip_cwk_add(":BASIS");
21         ip_cwk_add(":DEFAULT");
22         ip_cwk_add(":PSI");
23         ip_set_uppercase(1);
24         options.clear();
25         if(name == "INTEGRALS") {
26             ip_cwk_add(":INTEGRALS");
27             /*- The amount of information printed
28                 to the output file -*/
29             options.add_int("PRINT", 1);
30             /*- Whether to compute two-electron integrals -*/
31             options.add_bool("DO_TEI", false);
32         }
33         options.read_ipv1();
34
35         return true;
36     }
37 }
38
39
40 int
41 main(int argc, char *argv[], char *envp[])
42 {
43     int num_unparsed, i;
44     char *argv_unparsed[100];
45
46     for (i=1, num_unparsed=0; i<argc; ++i)
47         argv_unparsed[num_unparsed++] = argv[i];
48
49     // Setup the environment
50     Process::arguments.init(argc, argv);
51     Process::environment.init(envp);
52
53     // Initialize local communicator
54     Communicator::world = shared_ptr<Communicator>(new LocalCommunicator);
55
56     psi_start(&infile,&outfile,&psi_file_prefix,num_unparsed, argv_unparsed, 0);
```

```
57      psio_init();
58      psio_ipv1_config();
59      Options options;
60
61      read_options("INTEGRALS", options);
62      psi::integrals::integrals(options, argc, argv);
63
64      psi_stop(infile, outfile, psi_file_prefix);
65      return (EXIT_SUCCESS);
66  }
```

All modules in PSI4 must have the argument list and return type shown on line 13. The possible return types, defined by an enumeratable constant are documented in psi4-dec.h, which lives in $PSI4/include. Notice that all of the code must live in it's own namespace within the psi namespace, in this case it's in the psi::integrals namespace. Without this nesting, functions belonging to different parts of the code, but having the same name, would cause conflicts. The read_options function is responsible for setting up the Options object, which contains the list of user-provided options. Lines 25–32 are important - these provide the list of keywords expected by the code, their types, and their default values (if any). This part of the code will be inserted into the PSI4 driver when the module is ready for merging with the PSI4 distribution; this process will be detailed later in the chapter. Notice the special format of the comments on lines 27 and 30. These are still valid C++ comments, but the extra hyphens inside are essential in this context. Whenever adding any options for any module, you must comment them as shown - this will ensure that the keywords are automatically inserted into the PSI4 users' manual. The main function does a little setting up of the PSI input and output environments, before calling the module code we're developing (on line 53) and shutting down the PSI4 I/O systems.

The module we're developing is in the following source file.

sample-codes/integrals/integrals.cc

```
1   #include "psi4-dec.h"
2   #include <liboptions/liboptions.h>
3   #include <libmints/mints.h>
4   #include <libpsio/psio.h>
5
6   namespace psi{ namespace integrals{
7
8   PsiReturnType
9   integrals(Options &options, int argc, char *argv[])
10  {
11      int print = options.get_int("PRINT");
12      int doTei = options.get_bool("DO_TEI");
13      // This will print out all of the user-provided options for this module
14      options.print();
15      // Set up some essential arrays in the Wavefunction object
16      Wavefunction::initialize_singletons();
17      // Make a (reference-counted) psio object for I/O operations and set the parser up with
            it
18      shared_ptr<PSIO> psio(new PSIO);
19      psiopp_ipv1_config(psio);
20      // Open the existing checkpoint object by creating a Chkpt object
21      shared_ptr<Chkpt> chkpt(new Chkpt(psio, PSIO_OPEN_OLD));
22      // The matrix factory can create matrices of the correct dimensions...
23      shared_ptr<MatrixFactory> factory(new MatrixFactory);
24      // ...it needs to raid the checkpoint file to find those dimensions.
```

```
25      factory->init_with_chkpt(chkpt);
26      // The basis set is also created from the information stored in the checkpoint file
27      shared_ptr<BasisSet> basis(new BasisSet(chkpt));
28      // The integral factory oversees the creation of integral objects
29      shared_ptr<IntegralFactory> integral(new IntegralFactory
30              (basis, basis, basis, basis));
31
32      // Form the one-electron integral objects from the integral factory
33      shared_ptr<OneBodyAOInt> sOBI(integral->overlap());
34      shared_ptr<OneBodyAOInt> tOBI(integral->kinetic());
35      shared_ptr<OneBodyAOInt> vOBI(integral->potential());
36      // Form the one-electron integral matrices from the matrix factory
37      shared_ptr<Matrix> sMat(factory->create_matrix("Overlap"));
38      shared_ptr<Matrix> tMat(factory->create_matrix("Kinetic"));
39      shared_ptr<Matrix> vMat(factory->create_matrix("Potential"));
40      shared_ptr<Matrix> hMat(factory->create_matrix("One Electron Ints"));
41
42      // Compute the one electron integrals, telling each object where to store the result
43      sOBI->compute(sMat);
44      tOBI->compute(tMat);
45      vOBI->compute(vMat);
46
47      if(print > 5){
48          sMat->print();
49      }
50      if(print > 3){
51          tMat->print();
52          vMat->print();
53      }
54      // Form h = T + V by first cloning T and then adding V
55      hMat->copy(tMat);
56      hMat->add(vMat);
57      hMat->print();
58
59      if(doTei){
60          // Here's an example of an exception - we throw it if there are more than 99 basis
61          // functions, because only 2 digits are used in the print function below
62          if(chkpt->rd_nso() > 99)
63              throw PsiException("This code can only handle fewer than 100 basis functions",
                    __FILE__, __LINE__);
64          // Now, the two-electron integrals
65          shared_ptr<TwoBodyAOInt> eri(integral->eri());
66          // The buffer will hold the integrals for each shell, as they're computed
67          const double *buffer = eri->buffer();
68          // The iterator conveniently lets us iterate over functions within shells
69          AOShellCombinationsIterator shellIter = integral->shells_iterator();
70          int count=0;
71          for (shellIter.first(); shellIter.is_done() == false; shellIter.next()) {
72              // Compute quartet
73              eri->compute_shell(shellIter);
74              // From the quartet get all the integrals
75              AOIntegralsIterator intIter = shellIter.integrals_iterator();
76              for (intIter.first(); intIter.is_done() == false; intIter.next()) {
77                  int p = intIter.i();
78                  int q = intIter.j();
79                  int r = intIter.k();
80                  int s = intIter.l();
81                  fprintf(outfile, "\t(%2d %2d | %2d %2d) = %20.15f\n",
82                      p, q, r, s, buffer[intIter.index()]);
83                  ++count;
84              }
85          }
86          fprintf(outfile, "\n\tThere are %d unique integrals\n\n", count);
87      }
88
89      return Success;
90  }
91
```

```
92  }} // End Namespaces
```

Given the extensive documentation within the code, we'll not describe this file line-by-line; however, some points warrant elaboration. Notice that the entire module is encapsulated in the `psi::integrals` namespace (lines 6 and 92). This simple example has only one function body, which lives in a single source file - if more functions and/or source files were added, these too would have to live in the `psi::integrals` namespace. On lines 29 and 31 of `main.cc` we told the parser which keywords to expect, and provided default values in case the user omited them from the input. This makes retrieving these options very clean and simple (*c.f.* lines 11 and 12 of `integrals.cc`). Each `PSI4` module will have to initialize its own local `PSIO` and `Chkpt` objects to perform I/O and to retrieve information from previously run modules. Notice that these objects are created within smart pointers (see section XXX for more information) so that they are automatically deleted when they go out of scope, thus reducing the burden on the programmer. Likewise, the basis sets, matrices and integral objects are allocated using smart pointers.

The code described above can be built by simply typing "make" on the command line. To run this code, you must first run the `input` module to read in the basis set information. A `PSI` input for this code should look some thing like the following:

sample-codes/integrals/input.dat

```
1  psi:(
2     label      = "blah"
3     print      = 6
4     do_tei     = true
5     basis      = sto-3g
6     zmat       = (
7        0
8        H 1 0.9
9        H 1 0.9 2 105.0
10    )
11 )
```

5.2 MP2 With Density Fitting

Here's a more realistic example of a `PSI4` module - a reasonably efficient DF-MP2 code. The major advantage of density fitting methods is the replacement of four-index integrals with products of three-index integrals. With the `libmints` class, `PSI4` can compute these integrals easily and efficiently, as we will now demonstrate. The Makefile for this project is

sample-codes/df-mp2/Makefile

```
1  # Name of the executable you want to generate
2  TARGET = df-mp2
3
4  # List of source files to be compiled
5  CXXSRC = main.cc df-mp2.cc df-mp2-formJ.cc
6
7  # Location of your PSI4 source
```

```
psi_top_srcdir = /Users/andysim/programming/workspace/psi4

# Location of your PSI4 install, by default as listed
psi_top_objdir = /Users/andysim/programming/workspace/psi4_obj_debug

# Include the BLAS and LAPACK linear algebra libraries
BLAS   = -lblas
LAPACK = -llapack

# The name of the C++ compiler
CXX=g++

# C++ Compiler flags, in this case: no optimization and
# enable native GDB debugging, respectively
CXXFLAGS = -O0 -g $(INCLUDES)
#CXXFLAGS = -O2 $(INCLUDES)


#************ None of the following should be modified *********#
# Name of the object intermediates that make up the
# executable
OBJS = $(CXXSRC:%.cc=%.o)
# Include the PSI4 include files (this needs to be cleaned up)
INCLUDES = -I$(psi_top_srcdir)/include\
-I$(psi_top_objdir)/include\
-I$(psi_top_srcdir)/src/lib\
-I$(psi_top_objdir)/src/lib
# Include the PSI4 libraries
LIBS = -L$(psi_top_objdir)/lib -lPSI_mints -lPSI_ccenergy -lPSI_ccsort\
-lPSI_input3 -lPSI_cscf -lPSI_cints -lPSI_transqt2 -lPSI_psiclean -lPSI_dpd\
-lPSI_chkpt -lPSI_iwl -lPSI_qt -lPSI_psio -lPSI_ciomr -lPSI_ipv1\
-lPSI_options -lPSI_deriv -lPSI_int -lPSI_util -lPSI_parallel $(BLAS) $(LAPACK)
# Execute the compilation for the specified object, linking
# with all the libraries specified
$(TARGET): $(OBJS)
    $(CXX) -o $(TARGET) $(OBJS) $(LIBS)

# Execute the compilation for all objects of the specified
# target
all:
    $(TARGET)

# Erase all compiled intermediate files
clean:
    rm -f $(OBJS) $(TARGET)
```

As with the integral code in section 5.1, we make a simple `main` routine to set up the `PSI` I/O routines and call our new module. Notice that the hyphens are absent from the namespace definition, as they are not allowed in C++, although underscores are valid.

sample-codes/df-mp2/main.cc

```
#include <libpsio/psio.h>
#include <libipv1/ip_lib.h>
#include <libciomr/libciomr.h>

#include "psi4-def.h"
#include "libmints/mints.h"

using namespace psi;

namespace psi{
    FILE *infile;
    int read_options(std::string name, Options &options)
```

```
13      {
14          ip_cwk_clear();
15          ip_cwk_add(":BASIS");
16          ip_cwk_add(":DEFAULT");
17          ip_cwk_add(":PSI");
18          ip_set_uppercase(1);
19          options.clear();
20          if(name == "DF-MP2") {
21              ip_cwk_add(":DF-MP2");
22              /*- The amount of information printed
23                  to the output file -*/
24              options.add_int("PRINT", 1);
25              /*- Whether to compute the SCS energy -*/
26              options.add_bool("DO_SCS", true);
27              /*- Whether to compute the SCS-N energy -*/
28              options.add_bool("DO_SCS-N", true);
29              /*- The name of the orbital basis set -*/
30              options.add_str("BASIS", "");
31              /*- The name of the auxilliary basis set -*/
32              options.add_str("RI_BASIS", "");
33              /*- The opposite-spin scale factor for the SCS energy -*/
34              options.add_double("SCALE_OS", 6.0/5.0);
35              /*- The same-spin scale factor for the SCS energy -*/
36              options.add_double("SCALE_SS", 1.0/3.0);
37          }
38          options.read_ipv1();
39      }
40      namespace dfmp2{
41          PsiReturnType dfmp2(Options &options, int argc, char **argv);
42
43          void title()
44          {
45              fprintf(outfile, "\t\t\t*************************\n");
46              fprintf(outfile, "\t\t\t*                       *\n");
47              fprintf(outfile, "\t\t\t*         DF-MP2        *\n");
48              fprintf(outfile, "\t\t\t*                       *\n");
49              fprintf(outfile, "\t\t\t*************************\n");
50              fflush(outfile);
51          }
52      } // End Namespace df-mp2
53  } // End namespace psi
54
55
56  int
57  main(int argc, char *argv[], char *envp[])
58  {
59      int num_unparsed, i;
60      char *argv_unparsed[100];
61
62      for (i=1, num_unparsed=0; i<argc; ++i)
63          argv_unparsed[num_unparsed++] = argv[i];
64
65      // Setup the environment
66      Process::arguments.init(argc, argv);
67      Process::environment.init(envp);
68
69      // Initialize local communicator
70      Communicator::world = shared_ptr<Communicator>(new LocalCommunicator);
71
72      psi_start(&infile, &outfile, &psi_file_prefix,
73                  num_unparsed, argv_unparsed, 0);
74      psio_init();
75      psio_ipv1_config();
76      Options options;
77
78      psi::dfmp2::title();
79      read_options("DF-MP2", options);
80      psi::dfmp2::dfmp2(options, argc, argv);
```

```
81
82      psi_stop(infile, outfile, psi_file_prefix);
83      return (EXIT_SUCCESS);
84  }
```

The code for the Module itself is a little more complex than before, and a few extra features are used that were absent from the `integrals` code. On lines 31 and 36, instead of throwing a generic `PsiException` we throw a more specific `InputException`. The full list of exceptions available can be found on in the `Doxygen` documentation or in the `exceptions.h` file, which is in $PSI4/include. The default scale factors are defined in `main.cc`, where the options are declared and not where the options are read in on lines 51 and 52.

This code shows off the flexibility of the `libmints` module, which can accept arbitrary basis sets for each index in the four index integral. Passing a special `zero_basis_set` object for one index allows us to compute three-center integrals efficiently - see the `IntegralFactory` creation on lines 174 and 175.

sample-codes/df-mp2/df-mp2.cc

```
1   #include "psi4-dec.h"
2   #include <liboptions/liboptions.h>
3   #include <libmints/mints.h>
4   #include <libpsio/psio.h>
5   #include <libciomr/libciomr.h>
6   #include <libqt/qt.h>
7
8   #define TIME_DF_MP2 1
9
10  namespace psi{ namespace dfmp2{
11
12  extern void formInvSqrtJ(double **&J_mhalf, shared_ptr<BasisSet> basis,
13                    shared_ptr<BasisSet> ribasis, shared_ptr<BasisSet> zero);
14
15  PsiReturnType
16  dfmp2(Options &options, int argc, char *argv[])
17  {
18      // Required for libmints, allocates and computes:
19      // ioff, fac, df, bc
20      Wavefunction::initialize_singletons();
21      shared_ptr<PSIO> psio(new PSIO);
22      psiopp_ipv1_config(psio);
23      shared_ptr<Chkpt> chkpt(new Chkpt(psio, PSIO_OPEN_OLD));
24      shared_ptr<MatrixFactory> factory(new MatrixFactory());
25      factory->init_with_chkpt(chkpt);
26      shared_ptr<BasisSet> basis(new BasisSet(chkpt));
27
28      int print = options.get_int("PRINT");
29      std::string ri_basis = options.get_str("RI_BASIS");
30      if(ri_basis == ""){
31          throw InputException(std::string("Keyword not specified"),
32                  std::string("RI_BASIS"), __FILE__, __LINE__);
33      }
34      std::string orbital_basis = options.get_str("BASIS");
35      if(orbital_basis == ""){
36          throw InputException(std::string("Keyword not specified"),
37                  std::string("BASIS"), __FILE__, __LINE__);
38      }
39      fprintf(outfile, "Using the %s basis set for the orbitals, with the %s RI basis\n\n",
40                      orbital_basis.c_str(), ri_basis.c_str());
41
42      shared_ptr<BasisSet> ribasis(new BasisSet(chkpt, "DF_BASIS"));
```

```
43
44      if(print > 5){
45          ribasis->print();
46      }
47      shared_ptr<BasisSet> zero(BasisSet::zero_basis_set());
48
49      bool doSCSN = options.get_bool("DO_SCS-N");
50      bool doSCS  = options.get_bool("DO_SCS");
51      double SCSScaleOS = options.get_double("SCALE_OS");
52      double SCSScaleSS = options.get_double("SCALE_SS");
53      if(doSCS){
54          fprintf(outfile,"\tSpin-Component Scaled RI-MP2 requested\n"
55                          "\tOpposite-spin scaled by %10.4lf\n"
56                          "\tSame-spin scaled by     %10.4lf\n", SCSScaleOS, SCSScaleSS);
57      }
58      // The integrals code below does not use symmetry, so we need to accumulate the
59      // orbital info for each irrep
60      int nirreps = chkpt->rd_nirreps();
61      int *clsdpi = chkpt->rd_clsdpi();
62      int *orbspi = chkpt->rd_orbspi();
63      int *frzcpi = chkpt->rd_frzcpi();
64      int *frzvpi = chkpt->rd_frzvpi();
65      int ndocc = 0;
66      int nvirt = 0;
67      int nfocc = 0;
68      int nfvir = 0;
69      int norbs = 0;
70      int nact_docc = 0;
71      int nact_virt = 0;
72      for(int h=0; h < nirreps; ++h){
73          nfocc     += frzcpi[h];
74          nfvir     += frzvpi[h];
75          ndocc     += clsdpi[h];
76          nact_docc += clsdpi[h] - frzcpi[h];
77          nvirt     += orbspi[h] - clsdpi[h];
78          nact_virt += orbspi[h] - frzvpi[h] - clsdpi[h];
79          norbs     += orbspi[h];
80      }
81
82      fprintf(outfile, "\n\t\t================================================\n");
83      fprintf(outfile, "\t\t #ORBITALS #RI  FOCC DOCC AOCC AVIR VIRT FVIR \n");
84      fprintf(outfile, "\t\t------------------------------------------------\n");
85      fprintf(outfile, "\t\t  %5d  %5d  %4d %4d %4d %4d %4d %4d\n",
86              norbs,ribasis->nbf(),nfocc,ndocc,nact_docc,nact_virt,nvirt,nfvir);
87      fprintf(outfile, "\t\t================================================\n");
88
89      // Read in MO coefficients
90      shared_ptr<SimpleMatrix> C_so(factory->
91          create_simple_matrix("MO coefficients (SO basis)"));
92
93      double *orbital_energies = chkpt->rd_evals();
94      double **vectors = chkpt->rd_scf();
95
96      if (vectors == NULL) {
97          throw PsiException("Could not find MO coefficients. Run cscf first.",
98                  __FILE__, __LINE__);
99      }
100     C_so->set(vectors);
101     free_block(vectors);
102     double **Umat = chkpt->rd_usotbf();
103     shared_ptr<SimpleMatrix> U(factory->
104             create_simple_matrix("SO->BF"));
105     U->set(Umat);
106     free_block(Umat);
107     // Transfrom the eigenvectors from the SO to the AO basis
108     shared_ptr<SimpleMatrix> C(factory->
109             create_simple_matrix("MO coefficients (AO basis)"));
110     C->gemm(true, false, 1.0, U, C_so, 0.0);
```

```
111
112     double** Co   = block_matrix(norbs, nact_docc);
113     double** Cv   = block_matrix(norbs, nact_virt);
114     double** half = block_matrix(nact_docc, norbs);
115     double* epsilon_act_docc = new double[nact_docc];
116     double* epsilon_act_virt = new double[nact_virt];
117     int*    docc_sym = new int[nact_docc];
118     int*    virt_sym = new int[nact_virt];
119     int offset = 0;
120     int act_docc_count  = 0;
121     int act_virt_count  = 0;
122     for(int h=0; h<nirreps; ++h){
123         // Skip over the frozen core orbitals in this irrep
124         offset += frzcpi[h];
125         // Copy over the info for active occupied orbitals
126         for(int i=0; i<clsdpi[h]-frzcpi[h]; ++i){
127             for (int mu=0; mu<norbs; ++mu){
128                 Co[mu][act_docc_count] = C->get(mu, offset);
129             }
130             epsilon_act_docc[act_docc_count] = orbital_energies[offset];
131             docc_sym[act_docc_count] = h;
132             ++act_docc_count;
133             ++offset;
134         }
135         // Copy over the info for active virtual orbitals
136         for(int a=0; a<orbspi[h]-clsdpi[h]-frzvpi[h]; ++a){
137             for (int mu=0; mu<norbs; ++mu){
138                 Cv[mu][act_virt_count] = C->get(mu, offset);
139             }
140             epsilon_act_virt[act_virt_count] = orbital_energies[offset];
141             virt_sym[act_virt_count] = h;
142             ++offset;
143             ++act_virt_count;
144         }
145         // Skip over the frozen virtual orbitals in this irrep
146         offset += frzvpi[h];
147     }
148
149     if(print > 5){
150         fprintf(outfile, "Co:\n");
151         print_mat(Co, norbs, nact_docc, outfile);
152         fprintf(outfile, "Cv:\n");
153         print_mat(Cv, norbs, nact_virt, outfile);
154     }
155
156     double **J_mhalf;
157     formInvSqrtJ(J_mhalf, basis, ribasis, zero);
158
159     double **mo_p_ia = block_matrix(ribasis->nbf(),nact_docc*nact_virt);
160
161     // find out the max number of P's in a P shell
162     int maxPshell = 0;
163     for (int Pshell=0; Pshell < ribasis->nshell(); ++Pshell) {
164         int numPshell = ribasis->shell(Pshell)->nfunction();
165         maxPshell = numPshell > maxPshell ? numPshell : maxPshell;
166     }
167     double*** temp = new double**[maxPshell];
168     for (int P=0; P<maxPshell; P++) temp[P] = block_matrix(norbs, norbs);
169
170 #ifdef TIME_DF_MP2
171     timer_on("Form mo_p_ia");
172 #endif
173
174     shared_ptr<IntegralFactory>
175             rifactory(new IntegralFactory(ribasis, zero, basis, basis));
176     shared_ptr<TwoBodyAOInt> eri(rifactory->eri());
177     const double *buffer = eri->buffer();
178
```

```
179     for(int Pshell=0; Pshell < ribasis->nshell(); ++Pshell){
180         int numPshell = ribasis->shell(Pshell)->nfunction();
181         for(int P=0; P<numPshell; ++P){
182             zero_mat(temp[P], norbs, norbs);
183         }
184         for(int MU=0; MU < basis->nshell(); ++MU){
185             int nummu = basis->shell(MU)->nfunction();
186             for(int NU=0; NU <= MU; ++NU){
187                 int numnu = basis->shell(NU)->nfunction();
188                 eri->compute_shell(Pshell, 0, MU, NU);
189                 for(int P=0, index=0; P < numPshell; ++P){
190                     for(int mu=0; mu < nummu; ++mu) {
191                         int omu = basis->shell(MU)->function_index() + mu;
192                         for(int nu=0; nu < numnu; ++nu, ++index){
193                             int onu = basis->shell(NU)->function_index() + nu;
194                             // (oP | omu onu) integral
195                             temp[P][omu][onu] = buffer[index];
196                             // (oP | onu omu) integral
197                             temp[P][onu][omu] = buffer[index];
198                         }
199                     }
200                 } // end loop over P in Pshell
201             } // end loop over NU shell
202         } // end loop over MU shell
203         // now we've gone through all P, mu, nu for a given Pshell
204         // transform the integrals for all P in the given P shell
205         for(int P=0, index=0; P < numPshell; ++P){
206             int oP = ribasis->shell(Pshell)->function_index() + P;
207             // Do transform
208             C_DGEMM('T', 'N', nact_docc, norbs, norbs, 1.0, Co[0],
209             nact_docc, temp[P][0], norbs, 0.0, half[0], norbs);
210             C_DGEMM('N', 'N', nact_docc, nact_virt, norbs, 1.0, half[0],
211             norbs, Cv[0], nact_virt, 0.0, mo_p_ia[oP], nact_virt);
212         }
213     } // end loop over P shells; done with forming MO basis (P|ia)'s
214
215     for(int P=0; P<maxPshell; P++) free_block(temp[P]);
216
217 #ifdef TIME_DF_MP2
218     timer_off("Form mo_p_ia");
219     timer_on("Form B_ia^P");
220 #endif
221
222     // mo_p_ia has integrals
223     // B_ia^P = Sum_Q (i a | Q) (J^-1/2)_QP
224     double **B_ia_p = block_matrix(nact_docc * nact_virt, ribasis->nbf());
225
226     C_DGEMM('T','N',nact_docc*nact_virt,ribasis->nbf(),ribasis->nbf(),
227     1.0, mo_p_ia[0], nact_docc*nact_virt, J_mhalf[0], ribasis->nbf(),
228     0.0, B_ia_p[0], ribasis->nbf());
229
230     free_block(mo_p_ia);
231     free_block(J_mhalf);
232
233 #ifdef TIME_DF_MP2
234     timer_off("Form B_ia^P");
235     timer_on("Compute EMP2");
236 #endif
237
238     double *I = init_array(nact_virt * nact_virt);
239     double emp2 = 0.0;
240     double os_mp2 = 0.0, ss_mp2 = 0.0;
241
242     // loop over i>=j pairs
243     for(int i=0; i < nact_docc; ++i){
244         for(int j=0; j <= i; ++j){
245             int ijsym = docc_sym[i] ^ docc_sym[j];
246             // get the integrals for this pair of occupied orbitals
```

```
247 #ifdef TIME_DF_MP2
248             timer_on("Construct I");
249 #endif
250             int ia, jb;
251             for(int a=0,ab=0; a < nact_virt; ++a){
252                 ia = nact_virt*i + a;
253                 for(int b=0; b < nact_virt; ++b, ++ab){
254                     jb = nact_virt*j + b;
255                     int absym = virt_sym[a] ^ virt_sym[b];
256                     if (ijsym == absym)
257                     I[ab] = C_DDOT(ribasis->nbf(), B_ia_p[ia], 1, B_ia_p[jb], 1);
258                     else
259                     I[ab] = 0.0;
260                     // I[ab] = (ia|jb) for the fixed i,j
261                     // note (I[ab] = (ia|jb)) != (I[ba] = (ib|ja))
262                 }
263             } // end loop over a
264 #ifdef TIME_DF_MP2
265             timer_off("Construct I");
266 #endif
267             double iajb, ibja, tval, denom;
268             int ab, ba;
269             for(int a=0,ab=0; a < nact_virt; ++a){
270                 for(int b=0; b < nact_virt; ++b,ab++){
271                     int ba = b * nact_virt + a;
272                     iajb = I[ab];
273                     ibja = I[ba];
274                     denom = 1.0 /
275                     (epsilon_act_docc[i] + epsilon_act_docc[j] -
276                     epsilon_act_virt[a] - epsilon_act_virt[b]);
277
278                     tval = ((i==j) ? 0.5 : 1.0) * (iajb * iajb + ibja * ibja) * denom;
279                     os_mp2 += tval;
280                     ss_mp2 += tval - ((i==j) ? 1.0 : 2.0)*(iajb*ibja)*denom;
281                 }
282             }
283         } // end loop over j<=i
284     } // end loop over i
285     emp2 = os_mp2 + ss_mp2;
286
287 #ifdef TIME_DF_MP2
288     timer_off("Compute EMP2");
289 #endif
290
291     free(I);
292     free_block(B_ia_p);
293
294     double escf = chkpt->rd_escf();
295     fprintf(outfile,"\tRI-MP2 correlation energy         = %20.15f\n",emp2);
296     fprintf(outfile,"      * RI-MP2 total energy               = %20.15f\n\n",
297     escf + emp2);
298     fprintf(outfile,"\tOpposite-Spin correlation energy  = %20.15f\n",os_mp2);
299     fprintf(outfile,"\tSame-Spin correlation energy      = %20.15f\n\n",ss_mp2);
300     fprintf(outfile,"      * SCS-RI-MP2 total energy           = %20.15f\n\n",
301     escf + SCSScaleOS * os_mp2 + SCSScaleSS * ss_mp2);
302     if(doSCSN){
303         fprintf(outfile,"      * SCSN-RI-MP2 total energy          = %20.15f\n\n",
304             escf + 1.76*ss_mp2);
305     }
306
307     return Success;
308 }
309
310 }} // end namespaces
```

The formation of the metric matrix is performed in a function defined in a seperate file. Breaking large chunks of code into smaller, more manageable, segments is usually a good strategy; if fact, the `main.cc` should probably be further subdivided. To do this, we need a declaration of the external function to be defined (lines 12 and 13 of `df-mp2.cc`) before defining the function itself in a seperate file, in this case we create the following file:

sample-codes/df-mp2/df-mp2-formJ.cc

```
1  #include "psi4-dec.h"
2  #include <libmints/mints.h>
3  #include <libciomr/libciomr.h>
4  #include <libqt/qt.h>
5
6  namespace psi{ namespace dfmp2{
7
8  void formInvSqrtJ(double **&J_mhalf, shared_ptr<BasisSet> basis,
9                    shared_ptr<BasisSet> ribasis, shared_ptr<BasisSet> zero)
10 {
11     // Create integral factories for the RI basis
12     shared_ptr<IntegralFactory>
13             rifactory_J(new IntegralFactory(ribasis, zero, ribasis, zero));
14     shared_ptr<TwoBodyAOInt> Jint(rifactory_J->eri());
15
16     double **J = block_matrix(ribasis->nbf(), ribasis->nbf());
17     J_mhalf = block_matrix(ribasis->nbf(), ribasis->nbf());
18     const double *Jbuffer = Jint->buffer();
19
20 #ifdef TIME_DF_MP2
21     timer_on("Form J");
22 #endif
23
24     int index = 0;
25     for (int MU=0; MU < ribasis->nshell(); ++MU) {
26         int nummu = ribasis->shell(MU)->nfunction();
27         for (int NU=0; NU < ribasis->nshell(); ++NU) {
28             int numnu = ribasis->shell(NU)->nfunction();
29             Jint->compute_shell(MU, 0, NU, 0);
30             index = 0;
31             for (int mu=0; mu < nummu; ++mu) {
32                 int omu = ribasis->shell(MU)->function_index() + mu;
33                 for (int nu=0; nu < numnu; ++nu, ++index) {
34                     int onu = ribasis->shell(NU)->function_index() + nu;
35                     J[omu][onu] = Jbuffer[index];
36                 }
37             }
38         }
39     }
40
41     // First, diagonalize J
42     // the C_DSYEV call replaces the original matrix J with its eigenvectors
43     int lwork = ribasis->nbf() * 3;
44     double* eigval = init_array(ribasis->nbf());
45     double* work = init_array(lwork);
46     int status = C_DSYEV('v', 'u', ribasis->nbf(), J[0],
47                          ribasis->nbf(), eigval, work, lwork);
48     if(status){
49         throw PsiException("Diagonalization of J failed", __FILE__, __LINE__);
50     }
51     free(work);
52
53     // Now J contains the eigenvectors of the original J
54     // Copy J to J_copy
55     double **J_copy = block_matrix(ribasis->nbf(), ribasis->nbf());
56     C_DCOPY(ribasis->nbf()*ribasis->nbf(), J[0], 1, J_copy[0], 1);
57
```

```
58      // Now form J^{-1/2} = U(T)*j^{-1/2}*U,
59      // where j^{-1/2} is the diagonal matrix of the inverse square roots
60      // of the eigenvalues, and U is the matrix of eigenvectors of J
61      for(int i=0; i<ribasis->nbf(); ++i){
62          eigval[i] = (eigval[i] < 1.0E-10) ? 0.0 : 1.0 / sqrt(eigval[i]);
63          // scale one set of eigenvectors by the diagonal elements j^{-1/2}
64          C_DSCAL(ribasis->nbf(), eigval[i], J[i], 1);
65      }
66      free(eigval);
67
68      // J_mhalf = J_copy(T) * J
69      C_DGEMM('t','n',ribasis->nbf(),ribasis->nbf(),ribasis->nbf(),1.0,
70      J_copy[0],ribasis->nbf(),J[0],ribasis->nbf(),0.0,J_mhalf[0],ribasis->nbf());
71      free_block(J);
72      free_block(J_copy);
73
74  #ifdef TIME_DF_MP2
75      timer_off("Form J");
76  #endif
77  }
78
79  }} // Namespaces
```

We also have to remember to add this new source file to the list of sources in the `Makefile`, but it's as easy as that. A sample input file that will drive this code is shown below.

sample-codes/df-mp2/input.dat

```
1   psi:(
2      label      = "blah"
3      wfn = scf
4      reference =  rhf
5      do_tei     = true
6      basis      = cc-pvdz
7      ri_basis   = "cc-pVDZ-RI"
8      zmat       = (
9         0
10        H 1 0.9
11        H 1 0.9 2 105.0
12     )
13  )
14
15  df-mp2:(
16     print  = 2
17  )
```

Notice how the `print` keyword is in a special `DF-MP2` section of the input file - this is to prevent the `input` module from parsing the print keyword and producing excessive output when the print level is high. This code requires `input`, `cints` and `cscf` to be run before it can function.