



THE CHRONUS QUANTUM (CHRONUSQ)
SOFTWARE PACKAGE
USER'S GUIDE

David Williams
Joshua J. Goings
Patrick J. Lestrangle
Xiaosong Li

Last Revised: November 13, 2016

Contents

1	General Overview	2
2	Obtaining and Compiling ChronusQ	2
2.1	GitHub Repository	2
2.2	Dependencies	2
2.2.1	C++11	3
2.2.2	CMake	3
2.2.3	Libint	3
2.2.4	Eigen3	3
2.2.5	BTAS	3
2.2.6	Boost	4
2.2.7	HDF5	4
2.2.8	Python	4
2.2.8.1	ConfigParser	4
2.2.8.2	h5py	4
2.2.9	LAPACK and BLAS	4
2.3	Configure and Compilation	5
2.3.1	Configure	5
2.3.2	Compilation	6
2.4	Testing Installation	6
3	Input Files	8
3.1	Specifying your molecule: the [Molecule] section	8
3.2	Setting up a gaussian basis set with the [BASIS] section	9
3.3	Defining the job type with the [QM] section	9
3.4	Controlling the SCF optimization: the [SCF] section	10
3.5	Parallelism and other miscellaenous options: the [MISC] section	10
3.6	Real time time-dependent Hartree-Fock: the [RT] section	11

1 General Overview

The purpose of this guide is to provide a step-by-step procedure to compile the Chronus Quantum Chemistry (ChronusQ) software package as well as outline some of the basic functionality currently provided. As such, detailed descriptions of the compilation procedure and the anatomy of the input file will be discussed. This guide is not meant to be a developer’s reference, although a ChronusQ developer’s manual is currently in the works and will be made publically available in the near future.

At its core, ChronusQ provides an open-source platform for the development of explicitly time-dependent electronic structure and cutting edge post self-consistent field (SCF) methods. As ChronusQ is a “free” (both in the financial and the GNU definitions) software package, use and development by the scientific community as a whole is strongly encouraged. Those interested in the development of ChronusQ should send inquiries to Xiaosong Li (xsli@u.washington.edu).

Please see the Table of Contents at the beginning of this document to find the information you desire regarding the compilation or use of ChronusQ.

2 Obtaining and Compiling ChronusQ

2.1 GitHub Repository

Currently, the only method for obtaining ChronusQ is through the Li Research Group GitHub repository located at http://github.com/liresearchgroup/chronusq_public.git. Those interested in the development of ChronusQ may request access to the private development GitHub repository through Xiaosong Li (xsli@u.washington.edu). To obtain a copy of the source code (via command line), the `git` program must be installed and be in your shell’s working path. This can be confirmed via the following command:

```
> which git
/usr/local/bin/git
```

where `/usr/local/bin/git` is the location of the `git` program installed on the (my) system. The location may vary, but as long as you get a location, `git` is installed. Once a `git` installation has been verified, obtaining a copy of the ChronusQ is as simple as:

```
> git clone http://github.com/liresearchgroup/chronusq_public.git
```

which will place a copy of the ChronusQ source into the directory `./chronusq_public`. Any comments, concerns or problems regarding obtaining source code from the `git` repository may be directed towards David Williams (dbwy@u.washington.edu).

2.2 Dependencies

The ChronusQ software package depends on a number of other open source packages to perform some of the underlying tasks that are required by approximate quantum mechanical methods (i.e. (multi-)linear algebra, gaussian integral evaluation, etc). While ChronusQ strives to be a stand alone package, many of the incorporated functionality depend heavily on outside open source software. Any problems regarding the installation of these dependencies should be resolved via the documentation of that software. Any problems regarding the communication of these dependencies and ChronusQ can be directed to David Williams (dbwy@u.washington.edu).

2.2.1 C++11

ChronusQ (and some of its dependencies) rely on the C++11 standard. The GNU Compilers have already incorporated this standard, but not as the default. Unless you are using GCC 5.X+, you will likely have to add `-std=c++0x` or `-std=c++11` to the compile flags to force use of C++11. The configure/compile procedure described in Section 2.3 will try to smartly figure out the C++11 compile flags, but one may have to manually set the compile flags via `CMake` variables (also in Section 2.3)

2.2.2 CMake

ChronusQ utilizes the `CMake` utility to facilitate portability and flexibility of compilation through automatic Makefile generation. `CMake` is readily available through your OS distribution package manager (GNU/Linux or OSX). For example, in Fedora 22, one may obtain (if one has root privileges) `CMake` via

```
> sudo dnf install cmake
```

If for some reason you are unable to obtain a pre-packaged version of `CMake` through a package manager, the source and installation instructions may be obtained from <http://www.cmake.org>.

2.2.3 Libint

For the evaluation of molecular integrals over gaussian-type function (GTOs), ChronusQ relies on the `Libint` library of E. Valeev [1]. A preconfigured (uncompiled) library is shipped with ChronusQ (located in the `/deps/src` directory). The configure/compile procedure described in Section 2.3 details the `CMake` options to facilitate compilation of `Libint`. ChronusQ attempts to use the latest version of `Libint`, but we will only support compilation and linking to the locally stored version of `Libint` as the functionality and interfaces may vary between versions.

2.2.4 Eigen3

ChronusQ currently utilizes `Eigen3` [2] as a high-level C++ API for various light-weight linear algebra tasks (i.e. storage, multiplication, etc). `Eigen3` is also made available through most GNU/Linux OS distributions via a standard package manager. One may obtain a pre-packaged `Eigen3` installation (with root access / Fedora 22) via

```
> sudo dnf install eigen3-devel
```

If for some reason you are unable to obtain a pre-packaged version, installation of `Eigen3` is quite simple as it is a header-only library. One need simply download the source tar file from <http://http://eigen.tuxfamily.org/> and place the contents somewhere that ChronusQ can find them. An explanation of the `CMake` variables that need to be set for a non-standard installation of `Eigen3` can be found in Section 2.3.

2.2.5 BTAS

ChronusQ currently utilizes `BTAS` (**B**asic **T**ensor **A**lgebra **S**ubroutines) [3] as a C++11 API for multi-linear algebra. `BTAS` is obtained by ChronusQ automatically

2.2.6 Boost

Various parts of the code depend on the C++ **Boost** libraries [4], namely the **Python** API. Although the installation of **Boost** is relatively simple, we’ve found that the path of least resistance on the end user involves an automatic installation of the needed modules of **Boost** via **CMake**. The needed modules are compiled and linked to by default and we do not currently support linking to a compiled version on the user’s development environment.

2.2.7 HDF5

ChronusQ utilizes **HDF5** [5] for binary file IO for use with checkpointing and scratch file generation. **HDF5** is made available though the standard (GNU/Linux) OS distribution package manager via (Fedora 22 / root access):

```
> sudo dnf install hdf5-devel
```

If for some reason **HDF5** cannot be installed in this manner (i.e. no root access), it may be compiled from source from the tar files on <https://www.hdfgroup.org/>. Instructions on how one may set the **CMake** and shell environment variables to work with ChronusQ can be found in Section 2.3.

2.2.8 Python

ChronusQ utilizes **Python** as a high level API for input file digestion and the actual running of the ChronusQ software. One must have the development versions of **Python** as well as **libxml2** and **libxslt** to use ChronusQ. These may be obtained through the standard package manager of your (GNU/Linux) OS distribution (Fedora 22 / root access) via:

```
> sudo dnf install python-devel libxml2-devel libxslt-devel
```

2.2.8.1 ConfigParser

To parse the input file, ChronusQ relies on the **Python** module **ConfigParser**. One may obtain **ConfigParser** through the **Python** **pip** module via:

```
> pip install configparser
```

2.2.8.2 h5py

To parse the reference value file for the unit tests, ChronusQ utilizes the **Python** wrapper for **HDF5**, **h5py**. This module may be obtained through the **Python** **pip** module via:

```
> pip install h5py
```

2.2.9 LAPACK and BLAS

ChronusQ utilizes **LAPACK** [6] and **BLAS** [7, 8, 9, 10] to perform the most important linear-algebra functionality (i.e. **SVD**, **QR**, diagonalization, etc). **LAPACK** and **BLAS** come standard on most (GNU/Linux) OS distributions, and if not, they are easily obtained via the standard package manager (i.e. Fedora 22 / root access) via:

```
> sudo dnf install lapack-devel blas-devel
```

If for some reason LAPACK and BLAS cannot be installed in this manner (i.e. no root access), we have included an automatic build of these packages through our configuration procedure. Please see Section 2.3. for details.

IMPORTANT: While it is encouraged to attempt to link to optimized LAPACK and BLAS libraries, the developers have experienced many issues when linking to Intel MKL libraries. Please link to the traditional LAPACK and BLAS libraries or ATLAS optimized libraries when configuring ChronusQ.

2.3 Configure and Compilation

This section outlines the configuration and compilation of the ChronusQ software package via CMake. Before this procedure can be carried out, all of the dependencies (unless otherwise stated) from the previous section must be installed. Any problems regarding the configuration or compilation may be directed toward David Williams (dbwy@u.washington.edu).

2.3.1 Configure

The configuration of the machine specific Makefiles to compile ChronusQ are handled by CMake. ChronusQ has adopted an “out-of-source” compilation model to better separate source and compiled code. In this manner, if a compilation or configuration goes wrong, one must simply delete the build directory and start over with no risk of editing the source code. Configuration of ChronusQ takes place via the following general scheme

```
> cd /path/to/chronusq
> mkdir build && cd build
> cmake -D<OPT1>=<V1> -D<OPT2>=<V2> [ETC] ..
```

where <OPT1> and <OPT2> are CMake variables and V1 and V2 are the corresponding values to set these variables. The “..” must be present at the end of the command to let CMake know that there is a file in the previous directory called “CMakeLists.txt”, which contains the CMake configuration instructions. The following CMake variables may be influential to a successful configuration of ChronusQ:

Variable	Purpose	Default Value
CMAKE_CXX_COMPILER	Set the C++ Compiler (full path)	g++
CMAKE_C_COMPILER	Set the C Compiler (full path)	gcc
CMAKE_Fortran_COMPILER	Set the FORTRAN Compiler (full path)	gfortran
CMAKE_CXX_FLAGS	Set the C++ Compile Flags	–
EIGEN3_ROOT	Path that contains the Eigen3 directory	/usr/include/eigen3
BUILD_LA	Build LAPACK and BLAS locally	OFF
BUILD_LIBINT	Build Libint locally	ON

As a working example, on a machine that all of the dependencies had to be installed as a non-root user, the following script generated a successful configuration:

```
#!/bin/sh

echo "Building ChronusQ in "$PWD
export HDF5_ROOT=$(HOME)/HDF5
cmake \
  -DCMAKE_CXX_COMPILER=$(HOME)/gcc/gcc-4.9.2/bin/g++ \
  -DCMAKE_C_COMPILER=$(HOME)/gcc/gcc-4.9.2/bin/gcc \
  -DCMAKE_Fortran_COMPILER=$(HOME)/gcc/gcc-4.9.2/bin/gfortran \
  -DCMAKE_CXX_FLAGS='-w -O2 -std=c++11'\
  -DEIGEN3_ROOT=$(HOME)/eigen/eigen-eigen-c58038c56923 \
  -DBUILD_LA=ON \
  ..
```

2.3.2 Compilation

Once a successful configuration has been achieved, compilation is very simple. From the build directory, simply type

```
> make boost && make btas && make -j <NCORES>
```

and compilation will begin. ChronusQ does not currently have a standard install protocol once the compilation has been successful, so it is not suggested that the user performs a `make install` command, as we are not sure if this will work on all machines. Note that this is being looked into and will be handled before the first non-Beta release.

Once ChronusQ has been successfully compiled, you should find a file named “chronusq.py” in the main build directory. This is the ChronusQ python script and it may be run in two ways. The first is directly through python:

```
> python chronusq.py <input_file>
```

The next is to create an executable out of the script and run it directly

```
> chmod +x chronusq.py
> ./chronusq.py <input_file>
```

With this last method, it is possible to place `chronusq.py` into your `PATH` and run it from outside directories:

```
> export PATH=$PATH":"$PWD
> chronusq.py <input_file>
```

2.4 Testing Installation

After compiling ChronusQ, it is recommended that users test that the code is functioning correctly. We’ve added a set of unit tests that are available in the `tests` folder in the source directory. Inside that directory you will find:

```
chronusq.py -> <build directory>/chronusq.py
```

```
chronus-ref.bin
rununit.py
testXXXX*.inp <many input files>
```

The `rununit.py` utility will run all or some of the unit tests and compare against reference values that are stored in `chronus-ref.bin`. which is an HDF5 file Upon completion, the results of each test will be printed in `summary.txt`.

For typical users, we recommend that you simply run the unit tests with no options, so that the full program can be tested. If you are a developer, there are some options that will give you the flexibility to test only the area of the code that you are currently working on.

You can learn about the options for this utility by viewing it's help page.

```
> python rununit.py -h
```

```
python runtests.py [-o --option=]
```

Options:

```
-h, --help      Print usage instructions
-s, --silent    Disable Print
-k, --kill      Stop testing if a job fails
--type=        Determines types of tests to run. Multiple options
                can be specified by separating with a comma.
                3 classes of tests = [SCF,RT]
                Specify References = [RHF,UHF,GHF,X2C]
                                   [RKS,UKS]
                Reference and Type = [(R|U)HF-SCF]
                                   [(R|U)KS-SCF]
                Dipole Field      = [DField]
--parallel=     Whether to run parallel jobs = [on] or [off]
--complex=      Complex Jobs = [yes] or [no] or [both]
                [both] is the default
--basis=        Only run tests for this basis set
                [STO-3G,6-31G,cc-pVDZ]
```

The `--type` options says to only run jobs that contain the specified string in their designation in `text.index`. For example, if you specify

```
> python rununit.py --type=SCF
```

then you will run all SCF test jobs regardless of the reference, but none of the real-time electronic dynamics (RT) test jobs. The `--type` option is an inclusive option and will run any job that contains one the strings that you specify.

The other options are exclusive and allow the user to eliminate specific types of jobs from their test set. For example, this command will only run the test jobs with the STO-3G basis set where the wave function is constrained to be real.

```
> python rununit.py --complex=no --basis=sto-3g
```

Note that the exclusive options do not take more than one argument.

If you identify an issue please add this to the issues section on the Chronus Quantum Issues page:

https://github.com/liresearchgroup/chronusq_public/issues.

3 Input Files

To use ChronusQ, it is necessary to specify the molecule and job type within an input file. The easiest way to understand the input for ChronusQ is to take a look at an example. Here is a sample input file for water, `h2o.inp`. As written, it performs an RHF/STO-3G calculation on neutral, singlet water using 4 OpenMP threads.

```
#
# Molecule Specification
#
[Molecule]
charge = 0
mult   = 1
geom:
  O    0.000000000 -0.0757918436 0.0
  H    0.866811829 0.6014357793 0.0
  H   -0.866811829 0.6014357793 0.0

#
# Job Specification
#
[QM]
reference = RHF
job       = SCF

#
# Basis Specification
#
[BASIS]
basis     = STO-3G

#
# Misc Settings
#
[Misc]
nsmg      = 4
```

ChronusQ input files are divided into sections that specify the molecular geometry, the type of job, and other miscellaneous options. Lines beginning with `#` are ignored. Inputs are not case-sensitive. Sections are defined by the square bracket, e.g. `[Molecule]` specifies the molecular geometry, charge, etc. When ChronusQ encounters a section, it then searches the following lines for the appropriate commands and keywords. We will look at these sections each in turn.

3.1 Specifying your molecule: the `[Molecule]` section

The `[Molecule]` section specifies the geometry, charge, and multiplicity of the system.

charge

A signed integer that defines the overall electric charge of your molecule.

mult

An integer that defines the spin multiplicity of the molecule. Singlets correspond to 1, doublets to 2, triplets to 3, and so on.

geom

Specifies the geometry of the molecule. The input is always Cartesian, and the default units are in Angstroms. Each line corresponds to one atom. Each line here follows the following format

```
<sp> <atomic symbol> <x-coordinate> <y-coordinate> <z-coordinate>
```

Please note that the `< sp >` is a required space at the begining of the line. This is an artifact of the input file parser, and must be included. That's all there is to the `[Molecule]` specification!

3.2 Setting up a gaussian basis set with the `[BASIS]` section

The `[BASIS]` section contains options to setup the gaussian basis set for the ChronusQ calculation. Only a small subset of basis sets come included with the ChronusQ repository, which can be found in the `basis` subdirectory of the main repository. Any gaussian basis set may be included by placing its definition in the `basis` directory as long as it is specified using the "Gaussian94" format (see the EMSL basis set database for more details). The following sections are recognized for the `[BASIS]` section:

BASIS String. Specifies the gaussian basis set. e.g. `STO-3G`, `cc-pVTZ`, etc.

FORCECART Boolean. "ON" enforces all basis functions with $L > 2$ be described using their cartesian function definitions. This will increase the basis set size and may lead to linear dependencies, especially in the Dunning basis sets. This option should only be realistically used as a debugging option when comparing to codes that cannot utilize solid harmonic gaussians.

3.3 Defining the job type with the `[QM]` section

The `[QM]` section sets up the type of job you want to run, be it a single point Hartree-Fock energy calculation or a real-time propagation. It is also where you specify the single-determinant reference. The following sections are recognized for the `QM` section:

Job

Defines job to be performed. Available options:

SCF Default. Perform self-consistent field energy optimization.

RT Perform real-time propagation. ChronusQ will know to look for section `[RT]` (described later).

Reference

Defines your reference wave function. Available options:

`[REAL/COMPLEX] RHF` - Restricted Hartree-Fock reference

[REAL/COMPLEX] UHF - Unrestricted Hartree–Fock reference
 [COMPLEX] GHF - Generalized (Two–Component) Hartree–Fock reference
 [COMPLEX] X2C - Exact–Two Component (Relativistic) Hartree–Fock reference
 [REAL/COMPLEX] R<FUNCTIONAL> - Restricted Kohn–Sham reference
 [REAL/COMPLEX] U<FUNCTIONAL> - Unrestricted Kohn–Sham reference
 Implemented functionals: SLATER,B88,LSDA,SVWN5,BLYP,B3LYP,BHandH

3.4 Controlling the SCF optimization: the [SCF] section

ChronusQ allows you to take finer control over the self-consistent field optimization through the [SCF] section. You can add an external electric field to the SCF here, as well as turn on and off DIIS acceleration and fiddle with the convergence tolerances. Here are the available options:

DENTOL Floating point number that specifies the desired convergence of the density.
 Default = 1e-10

ENETOL Floating point number that specifies the desired convergence of the energy.
 Default = 1e-12

MAXITER Integer that specifies the maximum number of SCF iterations.
 Default = 256

DIIS Boolean that specifies whether to do DIIS acceleration of SCF.
 Default = true
 Note the DIIS algorithm is Pulay’s Commutator-based DIIS.

FIELD Three floats that specify the external static electric field to be applied (in A.U.).
 Default is zero field, equivalent to: FIELD = 0.0 0.0 0.0

GUESS Type of guess for the wave function.
 Available options:

SAD Superposition of Atomic Densities. Default.

RANDOM Generate a random guess, at times very successful for two–component SCF calculations.

CORE Diagonalize the core–hamiltonian as the guess.

READ Read density from restart (.bin) file.

3.5 Parallelism and other miscellaneous options: the [MISC] section

If you compiled ChronusQ to work with SMP parallelism, you can change the number processors to be used in this section. All the keywords in [MISC] are totally optional. The default behavior of parallelism in ChronusQ is to use just one processor. Available options:

NSMP An integer number of processors to use.
 Default = 1

PRINT An integer [$1 \leq \text{print} \leq 4$] that toggles how much information is printed to the output file.

3.6 Real time time-dependent Hartree-Fock: the [RT] section

If in the [QM] section you have set Job = RT, ChronusQ will search for additional commands and options specified in the [RT] section. Here we can define the type of perturbing field (currently based on the electric dipole only), as well as the type of orthonormalization, and how long we want our time-evolution to last. Below are the possible flags:

MAXSTEP

An integer that defines how many time steps you want to take.
Default = 10.

TIMESTEP

A floating point number that defines how large your time step is (in au).
Default = 0.05 au.

EDFIELD

Three floats that indicate the magnitude of the x, y, and z dipole components of the electric field (in au)
Default is zero field, equivalent to: EDFIELD = 0.0 0.0 0.0

TIME_ON

A floating point time, t_{on} , (in au) we want the external field turned on.
Default = 0.0 au.

TIME_OFF

A floating point time, t_{off} , (in au) we want the external field turned off.
Default = 1000.0 au.

ENVELOPE

Envelope function that describes the shape of the external field. Possible options:

STEP Step function,

$$E(t) = \begin{cases} E & t_{on} \leq t \leq t_{off} \\ 0 & else \end{cases}$$

DELTA Delta function,

$$E(t) = E\delta(t - t_{on})$$

References

- [1] E. F. Valeev. Libint: machine-generated library for efficient evaluation of molecular integrals over gaussians, version 2.1.0. <https://github.com/evaleev/libint>, 2015.
- [2] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2010.
- [3] BTAS. <https://github.com/BTAS/BTAS>.
- [4] Boris Schäling. *The Boost C++ Libraries*. XML Press, Second edition, 2014.
- [5] The HDF Group. Hierarchical Data Format, version 5. <http://www.hdfgroup.org/HDF5/>, 1997-2015.

- [6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, Third edition, 1999.
- [7] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software (TOMS)*, 28(2):135–151, 2002.
- [8] J. Dongarra. Basic Linear Algebra Subprograms Technical Forum Standard. *International Journal of High Performance Applications and Supercomputing*, 16(1):1–111, 2002.
- [9] J. Dongarra. Basic Linear Algebra Subprograms Technical Forum Standard. *International Journal of High Performance Applications and Supercomputing*, 16(2):115–199, 2002.
- [10] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.