QUEST User's Guide

January 26, 2024

Contents

0 Preface

This version of the QUEST User's Guide intends to eventually replace previous renditions of similar user's guides for QUEST. As of this writing, there are three of such renditions:

`https://www.cs.ucdavis.edu/~bai/QUEST/manual.pdf` (dated April 2012)
`manual_old.pdf` (available in this repository, dated November 2013)
`https://code.google.com/archive/p/quest-qmc/` (`wikis` section has similar content, undated)

This document is made from a copy of the original `.tex` file that created `manual_old.pdf`, though it might change to a brand new `.tex` file or website in the future. Keep in mind that some of the information presented in this guide might be incorrect: this being an artefact of the old user's guide. It is intended for these errors to be fixed in this user's guide in the future, and for more content and detailed explanations to be provided.

Comments or suggestions (for this document or anything in the QUEST repository) are welcome to be emailed to `jwwalker@ucdavis.edu` or opened as an issue or discussion on the repository page.

1 Introduction

Quantum **E**lectron **S**imulation **T**oolbox (QUEST) is a Fortran 90/95 package for performing determinant quantum Monte Carlo (DQMC) methods for strongly correlated electron systems. Its development was motived by a FORTRAN 77 DQMC code[1] written and maintained by Richard Scalettar at the University of California, Davis (UCD). The intention for creating QUEST was to modernize the the legacy code by incorporating three principles:

1. Improved simulation performance: QUEST has improved the performance of the legacy code simulations by using new algorithms, like delayed update, and by integrating modern numerical kernels, BLAS/LAPACK. A six to eight times speedup had been observed for medium sized simulations.

2. To integrate existing programs: QUEST has integrated many legacy codes by modularizing their computational components, which makes QUEST not only a single program, but a toolbox. The advantages of modularization also include the ease of maintenance and the convenience of program interfacing.

3. To assist new simulations development: QUEST has several desired properties for developing new simulations, such as the ability of creating new lattice geometries. Several novel simulations had been done by using QUEST.

Currently, QUEST is still under development and debugging. The latest version can be downloaded from the following link:

`https://github.com/Meromorphics/quest`

Additional information about QUEST (including files for older versions of QUEST) can be found at:

`https://code.google.com/archive/p/quest-qmc/`
`https://www.cs.ucdavis.edu/~bai/QUEST/index.html`

[1] Known as the legacy code.

1.1 Problems that QUEST can solve

If no geometry is specified [2], QUEST uses a two-dimensional periodic rectangular lattice as the default geometry in a simulation [3]. For more information about running simulations in QUEST, see section 1.4 for the most essential information about running simulations (enough for a general end-user) and section SECTION for more advanced simulation configuration.

QUEST is stable for interaction strength from $U = 0$ to $U = 16t$ and inverse temperatures $\beta = 0$ to $20/t$. Here U is the onsite repulsion and t is the coefficient of the kinetic energy term, see Section 2.1 for more information about these parameters. However, away from half-filling, $\mu = 0$, error bars are large for large β (sign problem). The current implementation allows hundreds of sites to be simulated on commodity computers within a reasonable time. Appendix A provides performance benchmark for some example problems.

1.2 Organization of files

The top level files and directories of QUEST include

- `README.md`: general information about QUEST.
- `Makefile`: computer instructions for compiling QUEST.
- `/SRC`: directory for source code
- `/EXAMPLE`: directory for executable programs.
- `/doc`: directory for documentation.
- `/makefiles`: unused makefiles
- `/geometries`: example geometry files
- `/OpenBLAS`: copy of OpenBLAS source code
- `/ford_documentation`: directory for documentation generated by FORD [4].
- `ford.md`: instructions for generating FORD documentation

Under the `EXAMPLE` directory, several example programs are contained[5] .

- `test`: A simple test program for 2 dimensional square lattice.
- `verify`: A program verifying the correctness of QUEST by examining its results against theoretical values of two special cases, $U = 0$ and $t = 0$.
- `tdm`: A test program for time dependent measurements.
- `parallel`: A test program for MPI type parallelization.

[2] See sections 2.1 and 3 for more information about specifying geometries

[3] At the time of writing, this default geometry feature is bugged. Input files must be provided (this will be fixed in a future update).

[4] `https://forddocs.readthedocs.io/en/latest/`

[5] Only `verify.F90` and `ggeom.F90` are present from this list. This remainder of this list is kept here as a reminder for potential programs to include in a future update.

- `wrap`: Programs for different input and output formats.
- `sheet`: Programs for multilayer geometry.
- `DCA`: Programs for using DQMC as DCA solver.
- `geom`: Programs for general geometry and their simulations

The details for each program will be illustrated in Appendix A[6].

1.3 Installation

Installation can be a pain for many programs, so this section attempts to be overly complete to try and make the process as easy as possible. For a more advanced user, the following concise instructions should be enough: first clone the QUEST repository (link just below), then enter it and run `make`. QUEST is developed and tested to run on a Linux (Ubuntu) system with Intel's oneAPI HPC toolkit installed in its default location (change `MKLPATH` in the `Makefile` if it is installed elsewhere). Potential support for the `gnu` compiler is included in the `Makefile`, but compilation using this compiler is untested. Edit the root directories `Makefile` as necessary.

The main (and currently being updated) repository for QUEST can be found at:

```
https://github.com/Meromorphics/quest
```

As of this writing, QUEST is only configured to run on Linux (specifically, it is developed on a system running Ubuntu 22.04.3 LTS). Support for Mac and Windows is planned for a future release, but users of these systems might be able to get QUEST to work [7]. The easiest method installation is through the use of `git`. Assuming `git` is installed and the command `git` may be used on a terminal, QUEST may be installed by entering the following on the terminal:

```
git clone https://github.com/Meromorphics/quest.git
```

Which should create a directory named `quest` in the current working directory. After `git` installation, the new directory should be entered by entering on the terminal:

```
cd quest
```

Compilation of QUEST requires `gnumake`. To test if `gnumake` is installed, in a directory where no file named `Makefile` is present (eg an empty directory) entering the following in a terminal:

```
make
```

should return as output:

[6] `verify.F90` is explained in section SECTION and `ggeom.F90` in section SECTION. Appendix A is being kept as a reminder for potential programs to include in a future update.

[7] Before a bug with Intel's `ifx` compiler running QUEST was found, QUEST was confirmed to be running on Windows Subsystem for Linux (WSL). If a user wishes to run QUEST on a Mac or Windows system, the issue is most likely found within the `Makefile` of QUEST.

```
make: *** No targets specified and no makefile found.  Stop.
```

or some similar statement. Assuming `gnumake` is installed and the current working directory is `quest`, QUEST may be compiled by entering the command `make` on the terminal (identical statement to testing the `gnumake` installation). For many users, an error is likely to occur at this step. This is primarily due to how QUEST is developed. QUEST is developed on Ubuntu using compilers installed from Intel's oneAPI:

https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html#gs.3kfmxb

In its default configuration, QUEST requires the oneAPI Base and HPC toolkits to be installed (these give the required compilers and LAPACK library, among other things). To change the installation configuration of QUEST, the `Makefile` in the root directory (directory just `cd`'d into) of QUEST must be configured [8]. By default, QUEST assumes the Intel oneAPI `ifort` is being used [9] and this is installed in the default directory. If oneAPI is not installed in its default directory, the line:

```
MKLPATH = $(MKLROOT)/include/intel64/inp64
```

in `Makefile` may have to be changed to the Intel MKL library path. By default, QUEST is designed to run on one CPU thread. To run on multiple CPU threads by use of MPI, change the following line in `Makefile`:

```
PRG_FLAGS = $(FLAG_BSOFI) $(FLAG_ASQRD) $(FLAG_CKB) $(FLAG_CUDA)
```

to:

```
PRG_FLAGS = $(FLAG_BSOFI) $(FLAG_ASQRD) $(FLAG_CKB) $(FLAG_CUDA) -D_QMC_MPI
```

Note that at the time of writing, the MPI implementation of QUEST has a bug, so running on a single thread is required until this is fixed. QUEST has what older documentation says is "experimental GPU support." This GPU support has not been tested at all recently (no idea if it works), but there are remnants in the `Makefile` that may possibly get it to work.

Furthermore, before using the command `make`, the user must have oneAPI's environment variables set in their active terminal. To test if these environment variables are set, the user may enter on the terminal:

```
ifort
```

If the environment variables are not set, the output should have an appearance similar to:

[8] Quest contains many makefiles within its directories. Only the makefile located in the root directory should be ran using `make`, attempting to run `make` on the others will result in an error (but these files are still required for QUEST to compile).

[9] Attempting to use Intel oneAPI's `ifx` compiler results in an error when running `./ggeom`. This bug is intended to be fixed in a later release. For now, using `ifort` will result in compiler warnings but no errors.

```
Command 'ifort' not found, did you mean:
command 'fort' from deb fort-validator (1.5.3-1build1)
command 'isort' from deb isort (5.6.4-1)
Try: sudo apt install <deb name>
```

If environment variables are set correctly, the output should have an appearance similar to:

```
ifort: remark #10448: Intel(R) Fortran Compiler Classic (ifort)
is now deprecated and will be discontinued late 2024.
Intel recommends that customers transition now to using
the LLVM-based Intel(R) Fortran Compiler (ifx) for continued
Windows* and Linux* support, new language support, new language
features, and optimizations. Use '-diag-disable=10448'
to disable this message.
ifort: command line error: no files specified; for help type "ifort -help"
```

On a fresh installation of Linux, it is known that `libomp` must be installed for compilation to work correctly, which may be installed by running in a terminal:

```
sudo apt-get install libomp-dev
```

It is unknown if any other similar dependencies are required to install QUEST. User research may be required for such dependencies may be required [10].

Assuming compilation of QUEST succeeded with no errors (many warnings are expected and will be fixed in a future update), the user should verify that QUEST runs as expected as follows. Enter the directory `EXAMPLE/verify` (eg by entering `cd EXAMPLE/verify` in the root directory of QUEST). Then run the file `verify` by entering on the terminal:

```
./verify
```

Unless output was directed to a file (eg by using `> output_file` in the above command), text should begin to fill the terminal. An example of what output should be running down the terminal screen is:

```
Parameters : t =  0.00, mu = -0.50, U =  0.00, beta =  1.50,
============================================================================
                   Theoretical | Computed (avg +- error) |  |T-C| : error
----------------------------------------------------------------------------
         Density :   0.641643  |  0.641643 +-  0.000000  |    0.00 :  0.00
          Energy :   0.320821  |  0.320821 +-  0.000000  |    0.00 :  0.00
Double occupancy :   0.000000  |  0.000000 +-  0.000000  |    0.00 :  0.00
============================================================================
```

[10] If any other dependencies are found to be required, please create an issue or discussion on the Github repository or email `jwwalker@ucdavis.edu`

The most important part is what is printed at the very end:

```
81.94% within 1 error bar (Expected 63.2%).
95.83% within 2 error bar (Expected 86.5%).
Running time:   6.723399     (second)
```

The two percentages in the left columns should be around or greater than the two percentages in the right column. If this is not the case, QUEST ran incorrectly and further analysis of simulations should not be performed until either QUEST is bugfixed or installed correctly. The reason being is that this code tests out QUESTS predictions for two Hubbard models that have an analytic solution: the Hubbard model with 1 site, and a 4×4 Hubbard model with no Coulomb interaction ($U = 0$). See `EXAMPLE/verify/verify.F90` for more details. These two tests are particularly important if you wish to make your own changes to QUEST (to make sure QUEST runs correctly).

As long as `make` ran without issue and `verify` gives good sounding results, QUEST should in theory be installed correctly.

1.4 Basic simulation usage

In essence, QUEST runs user defined "geometry" files and returns results (measurements from a simulation defined by an input geometry). Advanced usage of QUEST is just fine tuning what is returned and how (see section 3 for more details). In this section we detail how to get QUEST to give you results from input geometry files (details on how to make geometry files are in section SECTION). For a typical end-user, running simulations using the instructions provided here is enough.

To run a simulation in the most basic form, three files are required. QUEST provides one of these files:

```
/EXAMPLE/geom/ggeom.F90
```

It is strange and confusing that such an important file is located in a directory of the example directory, but such is life. This file performs simulations (it is the "simulator"). The second required file is an input file. The input file contains the information that the simulator uses to perform a simulation. To run an input file (assuming the terminal has as its working directory `EXAMPLE/geom`), enter:

```
./ggeom <input_file>
```

where `<input_file>` is the input file to `ggeom`. For example, upon installation, QUEST has one input file named `in` ready to run (for examples sake, this file may be deleted without harm):

```
./ggeom in
```

An input file contains parameters defining how the simulation should be run (such as how many warmup steps should be done for Monte Carlo, and the imaginary time step for the Suzuki-Trotter expansion). If some parameters are skipped in the input file that are used in the simulator, these parameters are given "good" default values [11] (see section 3 for more details about default parameter values). Precise details about input files are shown in section SECTION, but here we will detail two lines in an input file that are of particular importance. Any input file should [12] contain the following two lines:

```
ofile = <output>
gfile = <geometry>.geom
```

`ofile` stands for output file, since QUEST will create (or overwrite) three output files with names:

```
<output>.geometry
<output>.out
<output>.tdm.out
```

So `<output>` should be a name (no file extension). For example, the file `in` mentioned earlier has `ofile = test` so the output would be three files names `test.geometry`, `test.out`, and `test.tdm.out`. See section 2.2 for details of the contents of these files (basically measurement results and a restatement of the input geometry). `<geometry>.geom` is the third and final user defined file required to run a simulation in its most basic form. As the name suggests, this file defines the geometry of the simulation. There are a few methods in which a `.geom` file may be formatted, see section 3 for more details. A sample geometry file is in the same directory as `./ggeom`, and more can be found in `/geometries`.

2 Basic usages

In this section we will introduce the basic input parameters and output formats of QUEST. Those parameters and formats are not universal, since QUEST allows each program to define its own input parameters and output results, as introduced in section 3. However, by walking through those basic parameters and formats, one can understand what QUEST needs and what QUEST can do, and the input/output styles of QUEST.

2.1 Input file

An input file of QUEST is consisted of a list of parameter assignments,

```
parameter = value
```

and single line comments, which are of the form (comments can only be declared one line at a time):

```
#<comment>
```

[11] It is a known bug that some parameters are not actually taking default values when running the simulator says that they are. This will be fixed in a future release

[12] Again, with no bugs, QUEST should give "good" default values, but for practical purposes these two parameters should always be defined.

Each parameter is associated with one of the following types: *integer*, *real*, *real array*, and *string*. The basic parameters include

1. `oname` (string): file name for output files.
2. `gfile` (string.geom): geometry file for lattice
3. `n` (integer): total number of sites.
4. `nx` (integer): number of sites in the x-direction.
5. `ny` (integer): number of sites in the y-direction.
6. `nz` (integer): number of sites in the z-direction.
7. `U` (real array): Hubbard parameters.
8. `t` (real array): Hubbard parameters.
9. `mu` (real array): Hubbard parameters.
10. `L` (integer): number of time slice.
11. `dtau` (real): discretization parameter.
12. `HSF` (integer): indicator of how Hubbard Stratonovich field is input.
13. `HSFin` (string): file name of input file of HSF.
14. `HSFout` (string): file name of output file of HSF.
15. `nwarm` (integer): number of warmup sweeps.
16. `npass` (integer): number of measurement (sample) sweeps.
17. `nmeas` (integer): frequency of performing equal time measurements.
18. `tausk` (integer): frequency of performing unequal time measurements.
19. `nbin` (integer): number of bins for measurement results.
20. `seed` (integer): random number seed.
21. `north` (integer): frequency of performing orthogonalization in matrix product.
22. `nwrap` (integer): frequency of performing recomputing in Green's function calculation.
23. `difflim` (real): tolerable difference of the matrices computed from different methods.
24. `errrate` (real): tolerable error rate of recomputing.
25. `ntry` (integer): number of sites to be flipped in the global sweep. UPD: This parameter controls the frequency of a type of Monte Carlo move which is needed for $U \geq 8$.

The parameters can be roughly divided into three groups:

1. Parameters for Hubbard model:

   The parameters `nx` and `ny` specify the `x`-dimension and `y`-dimension of the 2D rectangular lattice to be simulated. Note that parameter n, the number of total sites, is equivalent to `nx`×`ny`. The parameters `t`, `mu`, and `U` are used in the Hubbard Hamiltonian

$$\mathcal{H} = -t \sum_{\langle i,j \rangle, \sigma} \left(c_{i\sigma}^{\dagger} c_{j\sigma} + c_{j\sigma}^{\dagger} c_{i\sigma} \right) - \mu \sum_{i,\sigma} n_{i\sigma} + U \sum_{i} \left(n_{i\uparrow} - \frac{1}{2} \right) \left(n_{i\downarrow} - \frac{1}{2} \right),$$

   for kinetic energy, chemical energy and potential energy respectively. Parameter `L` and `dtau` are for inverse temperature β.

$$\beta = L\Delta\tau.$$

   Parameter `HSF` indicates how the Hubbard-Stratonovich Field (HSF) is initialized.

$$\text{HSF} = \begin{cases} -1, & \text{randomly generating HSF;} \\ 0, & \text{use HSF in the memory;} \\ 1, & \text{read HSF from a file.} \end{cases}$$

   If `HSF=1`, then `HSFin` is the input file name. The generated HSF can be also output to a file by specifying the file name in `HSFout`.

2. Parameters for Monte Carlo simulation and physical measurements: Parameter `nWarm` and `nPass` in the second group decide how many Monte Carlo loops need be executed for warm up and measurement sweep. Parameter `nMeas` and `tausk` specify the frequency of performing physical measurements. Parameter `nBin` determine how the computed data is divided into bins. Parameter `ntry` is used to specify how many global flipping should be performed per sweep for the large U.

3. Parameters regarding numerical concerns.

   Parameter `seed` is used for random number generator. If it is 0, a new seed will be generated from system time. Parameter `nOrth` specifies how often the stabilization algorithm should be performed in the calculation of Green's function. Parameter `nWrap` provides the initial frequency of recomputing Green's function. In QUEST, `nWrap` will be dynamically adjusted according to the errors of updating. Parameters `difflim` and `errrate` are used for the adjusting algorithm, which specify the tolerable matrix difference and the acceptable error rate.

In its current version, QUEST assumes all simulations are done for the Hubbard model. The source directory contains files that can be used to simulate the Holstein model, but these have not been tested (and implementation must manually be done if these were to be tested out, the Holstein model does not have a `ggeom.F90` file).

2.2 Output Results

The output of QUEST varies in different programs. The goal for this section is to introduce what QUEST can output, and their formats. The output of QUEST can be classified into three types

- Input/condifuration parameters.
- Equal time measurements.

- Unequal time measurements.

The input/configuration parameters are as introduced in the previous section. Some of them may be created or changed during the simulation, like `seed` and `nWrap`. Those configuration parameters can help identifying the output results.

In terms of formats, concerned only with measurements, there are three kinds

- Single real number.
- Array of real numbers.
- Array of complex numbers.

If a measurement is a single real number, it will be shown with three terms: name, average, and error. For example,

```
    Density :     1.000000 +-     0.000000
```

The measurement formatted in an array of real numbers (or complex numbers) is a *function*, whose arguments can be anything, like the distances between sites. The output of this type enumerates all its function arguments and values. For example, the equal time Green's function of a 4×4 periodic lattice is output like

```
Equal time Green's function:
          dx =  0, dy =  0        0.500000 +-     0.000000
          dx =  1, dy =  0       -0.119358 +-     0.000754
          dx =  2, dy =  0        0.000000 +-     0.000000
          dx =  0, dy =  1       -0.118936 +-     0.000414
          dx =  1, dy =  1        0.000000 +-     0.000000
          dx =  2, dy =  1        0.020050 +-     0.000307
          dx =  0, dy =  2        0.000000 +-     0.000000
          dx =  1, dy =  2        0.019829 +-     0.000184
          dx =  2, dy =  2        0.000000 +-     0.000000
```

The first line is the name of measurements. Below that, the first column is the arguments of the function. The second and the third columns are the averages and errors.

Complex results will be defined similarly with separated error bars for the real part and imaginary part.

For the detail formula of each measurements, please referee the working notes.

2.2.1 Equal time measurements

There are three groups of equal time measurements. The first group is the measurements that aggregate values from entire lattice. The second group is the autocorrelation functions that average pair of sites within the same distance class. The third group measures the pair susceptibilities.

1. Up spin occupancy
2. Down spin occupancy
3. Potential energy

4. Kinetic energy
5. Total energy
6. Density
7. XX Ferromagnetic structure factor
8. ZZ Ferromagnetic structure factor
9. XX Antiferromagnetic structure factor
10. ZZ Antiferromagnetic structure factor
11. Equal time Green's function
12. Density-density correlation function (up-up)
13. Density-density correlation function (up-dn)
14. XX Spin correlation function
15. ZZ Spin correlation function
16. S-wave pair structure factor
17. SX-wave pair structure factor
18. D-wave pair structure factor
19. SXX-wave pair structure factor
20. DXX-wave pair structure factor
21. PX-wave pair structure factor
22. PY-wave pair structure factor
23. PXY-wave pair structure factor
24. PYX-wave pair structure factor

2.2.2 Unequal time measurements

Every unequal time measurement is a function of imaginary time. The format of array typed measurement with two arguments, space and time, is like

```
G(nx,ny,ti)
dx =  0, dy =  0
          0   0.50000 +-   0.00000
          1   0.36076 +-   0.00311
          2   0.27942 +-   0.00328
      ...
dx =  1, dy =  0
```

```
           0  -0.12042 +-   0.00162
           1  -0.07183 +-   0.00097
           2  -0.04248 +-   0.00133
       ...
 dx =  2, dy =  0
       ...
```

where `dx =  0, dy =  0` are labels of space and followed by a list of time label and corresponding values.

Some measurements are from Fourier transformation, which are complex. Their format will be like

```
  0(   0.05179 +-   0.00270)  +i(   0.00000 +-   0.00000)
  1(   0.05085 +-   0.00248)  +i(  -0.00061 +-   0.00055)
  2(   0.04343 +-   0.00190)  +i(  -0.00067 +-   0.00078)
  3(   0.03755 +-   0.00144)  +i(  -0.00080 +-   0.00051)
  4(   0.03004 +-   0.00099)  +i(  -0.00090 +-   0.00076)
  5(   0.02512 +-   0.00092)  +i(  -0.00100 +-   0.00048)
  6(   0.02152 +-   0.00074)  +i(  -0.00134 +-   0.00031)
  7(   0.01829 +-   0.00072)  +i(  -0.00087 +-   0.00055)
  8(   0.01499 +-   0.00066)  +i(  -0.00051 +-   0.00055)
  9(   0.01332 +-   0.00065)  +i(  -0.00042 +-   0.00035)
```

The numbers inside the first parenthesis are the average and error of the real part and the numbers in the second parenthesis are for imaginary part.

1. Unequal time Green's function: $G(dx, dy, t)$
2. Discrete cosine transformed $G(dx, dy, t)$:$G(qx, qy, t)$
3. Fourier Transformed $G(dx, dy, t)$: $G(dx, dy, w)$
4. Fourier Transformed $G(qx, qy, t)$: $G(qx, qy, w)$
5. chi function: $\chi(dx, dy, t)$
6. Discrete cosine transformed $\chi(dx, dy, t)$: $\chi(qx, qy, t)$
7. Fourier Transformed $\chi(dx, dy, t)$: $\chi(dx, dy, w)$
8. Fourier Transformed $\chi(qx, qy, t)$: $\chi(qx, qy, w)$
9. S-Wave pair structure factor, vertex and nonvertex
10. SX-Wave pair structure factor, vertex and nonvertex
11. D-Wave pair structure factor, vertex and nonvertex
12. SXX-Wave pair structure factor, vertex and nonvertex
13. DXX-Wave pair structure factor, vertex and nonvertex
14. PX-Wave pair structure factor, vertex and nonvertex

15. PY-Wave pair structure factor, vertex and nonvertex
16. PXY-Wave pair structure factor, vertex and nonvertex
17. PYX-Wave pair structure factor, vertex and nonvertex
18. FTed S-Wave pair structure factor, vertex and nonvertex
19. FTed SX-Wave pair structure factor, vertex and nonvertex
20. FTed D-Wave pair structure factor, vertex and nonvertex
21. FTed SXX-Wave pair structure factor, vertex and nonvertex
22. FTed DXX-Wave pair structure factor, vertex and nonvertex
23. FTed PX-Wave pair structure factor, vertex and nonvertex
24. FTed PY-Wave pair structure factor, vertex and nonvertex
25. FTed PXY-Wave pair structure factor, vertex and nonvertex
26. FTed PYX-Wave pair structure factor, vertex and nonvertex

3 Advanced usages

To assist in creating new simulations,[13] QUEST provides several simple mechanisms to

1. Add new configuration parameters,
2. Create new lattice geometries,
3. Add new measurements.

This section will introduce how to apply them.

3.1 Add new configuration parameters

Section 2.1 lists the basic input parameters, which are enough for current programs. However, for new simulations, additional parameters may be required. QUEST allows new parameters to be specified through current configuration system. For example, in the diluted lattice, in which some sites are randomly removed, the percentage of removal sites can be specified in the input file

```
rmv_ratio = 0.1
```

just like the other parameters.

To add new configuration parameters, user/developer need to edit a file called `config.def`. When the subroutine `DQMC_Config_Read` is called, it will first searches `config.def` for the definition of parameters. If the file exists, then the program will use the parameter defined in the file. Otherwise, it uses default parameter set, as described in section2.1.

A parameter in `config.def` is defined by a quintuple:

[13]The new simulations do not include creating new models. Currently, QUEST is pretty much limited in Hubbard's model.

{name, type, isArray, printed, default}

where

name: a string, maximum characters 30, specifying the name of the parameter.

type: an integer specifying the data type of the parameter.

$$\begin{cases} \text{type=1}, & \text{real;} \\ \text{type=2}, & \text{integer;} \\ \text{type=3}, & \text{string.} \end{cases}$$

isArray: a boolean {T,F} specifying whether the parameter is an array. Note, when type is a string, isArray cannot be T.

printed: a boolean {T,F} specifying whether the parameter will be printed in the output.

default: a string, maximum characters 30, specifying the default value of the parameter.[14] QUEST will convert it to a proper data type based on the "type" tuple.

Each tuple is separated by spaces. For example, the parameter `rmv_ratio` can be defined in the `config.def` as

```
rmv_ratio  1 F T 0.1
```

The file `config.def` should be placed in the directory where the executable is.

3.2 Square lattice geometry file with comments.

```
#NDIM
2
```

Dimension of the lattice. 3 for cubic, 2 for square, triangular, etc

```
#PRIM
1.0 0.0
0.0 1.0
```

Primitive lattice vectors. One can give a full 3×3 matrix for #PRIM even if #NDIM ¡ 3. If you do that, QUEST only reads the upper left #NDIM $\times$ #NDIM block.

```
#ORB
s0 0.0 0.0 0.0 #0
```

This is what alllows QUEST to do geometries like the honeycomb lattice which requires a basis. #ORB has one line for each atom in the basis. The first entry is a string which serves as a label for the atom. The next three entries in the line are the position of the atom in the unit cell. QUEST authomatically assignes a number to each atom (ie to each line in the #ORB section of the .geom file) beginning with zero. In the #ORB section QUEST demands three dimensional objects.

[14]Note, even a parameter is an array, its default value can be only set by one number.

#HAMILT block of the code defines the hoppings, on-site energies, and interaction strenghts. The convebtion for the lines in #HAMILT is following: Each line begins with two entries which are the (automatically assigned) atom numbers from #ORB. For problems without a basis these will just be '0.0'. To include a hopping, the next three entries in the line should be the direction to the neighboring site. QUEST authomatically makes $\hat{H}$ Hermitian, so each pair of connected sites requires only one line. The next two entries are the hopping values for the up and down electrons, which are allowed to be different in QUEST. The final entry is U, which should be set to zero for lines defining hopping. To include a U value, use '0.0 0.0 0.0' for the neighboring site inputs and insert a value in the last (eighth) entry.

If you have a geometry with a basis (more than one line in #ORB) the different atoms can be assigned different site energies and interactions.

Here is example for the rectangular geometry:

```
#HAMILT
0 0 1.0 0.0 0.0 1.0 1.0 0.0
0 0 0.0 1.0 0.0 2.2 2.2 0.0
0.0 0.0 0.0 0.0 0.5 0.5 4.0
```

The first two lines are the hoppings in the x and y directions. The third line is an on-site energu 0.5, and is the same for th eup and down species. (the sign convebtion for the site energies is $+\epsilon n_i$, so that 0,5 in #HAMILT corresponds to -0.5 in the Chemical potential. The third line also sets $U = 4$. (If you want, you can separate this into two distinct lines)). We again note that in #HAMILT it is required to supply a three component vector as the pointer to the site, even if #NDIM is two.

Note: The chemical potential defined in input is a **global** chemical potential which applies to all sites in the lattice. The on-site energies in #HAMILT allow for different chemical potentials on different atoms. Thus there is a slight redundancy in the code. Rather than having a global chemical potential one could shift all the site energies. If you do not specify chemical potentials in input, they are set, by default, equal to zero.

There is some redundancy in the way QUEST knows the geometry is rectangular. If the defined supercell in #SUPER has different diagonal entries, then QUEST knows not to assume x and y directions are equivalent. Likewise, a different entry for t_x and t_y will automatically be flagged by QUEST (Finally, the strange string label in #ORB about which we have been so silent can also be used to distinguish different types of atoms and can break symmetries. If you had a three band Cu-O model of the cuprates you could, for example, make the oxygen atoms along the x and y bonds have distinct labels and hence different entries in #HAMILT)

But more fundamentally, QUEST knows about the symmetries of the lattice from the #SYMM block in the .geom file, which is the last one we shall discuss. In square.geom this looks like,

```
#SYMM
d  0.0 0.0 0.0 1.0 0.0 0.0
d  0.0 0.0 0.0 0.0 1.0 0.0
c4 0.0 0.0 0.0 0.0 0.0 1.0
```

QUEST supports 3 types of symmetry definitions. 'cn', where 'n' is an integer, tells QUEST the lattice is symmetric under rotations by $2\pi/n$. The six numbers following by 'cn' specify the three cartesian coordinates of a point belonging to the axis, following by the axis direction in the cartesian coordinates. In this case 'c4' is $\pi/2$ and indicates the x and y directions are equivalent. the point belonging to the axis is the origin '0.0 0.0 0.0' and the axis is the z direction '0.0 0.0 1.0'.

The symmetry 'd' is a mirror plane symmetry. It too is followed by six numbers. The first three are the cartesian coordinates of a point in the plane, and the final three are the cartesian cordinates of the normal to the plane.

Finally, 'I' is used for inversion symmetry. It is followed by three numbers, the cartesian coordinates of the inversion point.

In specifying #SYMM you must list all three components of teh vectors even if the lattice is two dimensional.

3.3 Create new lattice geometries

In QUEST, geometry dependent variables are placed together in a derived data type `Struct`. To create a new lattice geometry, one needs to fill out the data fields in `Struct`. There are several ways to achieve this; each has its advantages and drawbacks.

1. Write a program to fill out the data fields: most efficient way in execution, but need to write programs for different geometries.
2. Use primitive cell definition as input: [15] Most compact representations of general geometry, but needs to understand the definitions of primitive cell.
3. Input data for each data field from files: Most easy way to create a new lattice geometry, but less flexibilities and less efficient.

In this section, we will illustrate the data fields in `Struct` and the file formats for method 3. Note not all the data fields are essential for simulations. Some of them are just for a particular measurements. When lacking one or some of the fields, QUEST will skip the corresponding measurements.

3.3.1 `Struct`

Derived data type `Struct` is defined as

```
type Struct
   integer  :: nSite                  ! number of sites
   character(gname_len):: name        ! name of the structure
   integer, pointer    :: dim(:)      ! dim of the geometry

   integer  :: n_t                    ! number of hopping types
   type(CCS):: T                      ! hopping matrix

   integer  :: n_b                    ! number of neighbors types
   type(CCS):: N                      ! neighborhood matrix

   integer  :: nClass                 ! number of distance classes.
   integer, pointer  :: D(:,:)        ! distance classes
   integer, pointer  :: F(:)          ! counts for each dist class.
   character(label_len), pointer :: &
                          label(:) ! label for distant class.
```

[15]This feature is still underdevelopment.

```
   integer  :: nGroup              ! number of diff singletons.
   integer, pointer  :: map(:)     ! site classification

   real(wp), pointer :: W(:,:)     ! wave functions
   integer  :: nWave
   character(label_len), pointer :: &
                       wlabel(:)! label for wave functions.
   real(wp), pointer :: P(:)       ! phase assignment

   real(wp), pointer :: FT(:,:)    ! Fourier Transform matrix

   logical::checklist(N_CHECKLIST) ! flags
end type Struct
```

Here are the detail comments for each field.

- Sites must be numbered from 1, with continuous numbering.
- String `name` is of length 80 characters, used in display.
- Vector `dim` is used to hold dimension parameters. For example, for a two dimensional square lattice, `dim=(nx,ny)`, where `nx` and `ny` are the number of sites in x direction and in y direction. This field can be of arbitrary length. QUEST does not use this field directly.
- Hopping matrix `T` stores the indices of hopping parameter `t` for adjacent sites.[16] It is normally a sparse matrix; and therefore is represented in the Compressed Column Storage (CCS) format.[17] Field `n_t` is the number of different hopping parameters.
- Neighboring matrix `N` is similar to `T`, and is also represented in the CCS format. It is used in pair measurements, for which the link indices should be consistent with the wave function `W`. Field `n_b` is the number of different links.
- The distance classification is represented by `D`, `F` and `label`. Two pairs of sites are in the same class if they are translation/rotation invariant. The number of classes is specified by `nClass`. The class index is also started from 1. The number of pairs in each class is stored in `F`. String array `label` gives a label for each class, used in output. Matrix `D` records the classification for each pair of site. Number in `D(i,j)` denotes the class index for site i and j.
- The vector `map` classifies sites. Site i and site j are in the same class if they have the same physical parameters, like U and μ. The number `nGroup` denotes the number of site classes.
- Matrix `W` defines wave functions for the pair measurements, which are related to various spherical harmonic functions. The number of functions is specified by `nWave`. Each function has `n_b` elements, and is stored in a column of `W` matrix.
- The phase assignment vector `P` gives each site $\{+1, -1\}$ so that adjacent sites have opposite phases. This is used in spin correlation measurements, and in Green's function calculation when $\mu = 0$.

[16] Two sites i, j are called adjacent if electrons can hop from site i to site j.
[17] The detail of CCS format can be found in `http://www.netlib.org`.

- Matrix `FT` is the Fourier transformation matrix for distance classes, which is used in time dependent measurements.

- Vector `checklist` is a set of flags that indicate which data fields are assigned. The flags include

```
STRUCT_INIT   = 1
STRUCT_DIM    = 2
STRUCT_ADJ    = 3
STRUCT_CLASS  = 4
STRUCT_WAVE   = 5
STRUCT_NEIG   = 6
STRUCT_PHASE  = 7
STRUCT_FT     = 8
```

3.3.2 Files for geometry definition

The file name of geometry definition is specified in the configuration file with parameter name `gfile`. For example,

```
gfile = strip.def
```

tells QUEST to use the file `strip.def` as geometry definition.

The format of the geometry definition is similar to the configuration file.

1. The symbol `#` is used to start a comment.

2. The data fields are set by the assignment

   ```
   data_field_name = data_field_value
   ```

3. Some fields depend on the other. Depended fields should be declared first.

$$\begin{aligned} \texttt{T} &\rightarrow \texttt{nSite} \\ \texttt{B} &\rightarrow \texttt{nSite} \\ \texttt{D} &\rightarrow \texttt{nSite} \\ \texttt{FT} &\rightarrow \texttt{nClass} \\ \texttt{W} &\rightarrow \texttt{n\_b} \text{ and } \texttt{nWave} \end{aligned}$$

4. Not every field in `Struct` need be defined in the file. For example, vector `F` and `map` will be derived from matrix `D`.

5. Array type assignment, including matrix or vector, is different from that in a configuration file. The right-hand-side of assignment should be the number of how many lines to read. For example

   ```
   D = 64
   ```

means there are 64 immediate lines to read for matrix `D`. Those lines are called *content lines*. Note NO empty lines are allowed between content lines.

6. The format for a content line of a matrix is

   ```
   i j value
   ```

   where i is the row index, j is the column index, and `value` is the (i, j) element of the matrix.

7. Format for a content line of a vector is

   ```
   value
   ```

   Those values should be ordered sequentially, since indices are assumed to be implicitly embedded.

An example of how to create new geometry using method 3 can be found in the example program in `EXAMPLE\gemo`.

3.4 Add new measurements

New measurements need be made through programming in QUEST. Several subroutines can be used to create new measurements. The standard procedure to add a new measurement includes three steps.

1. Measuring.
2. Binning.
3. Postprocessing. (Statistics, Fourier transformation, output.)

3.4.1 Measuring

Several components may be needed to create a new measurements.

1. **Equal time Green's function**: Equal time Green's function is defined in the module `dqmc_gfun`, which will be initialized automatically when the subroutine `DQMC_Hub_Init` is called. Suppose `Hub` is typed `Hubbard`, the major data type of entire simulation. The Green's function matrices, spin-up and spin-down, can be obtained from

   ```
   Hub%G_up%G
   Hub%G_dn%G
   ```

   And the signs of their determinants are recorded in

   ```
   Hub%G_up%sgn
   Hub%G_dn%sgn
   ```

2. **Unequal time Green's function**: Unequal time Green's function, denoted G^{τ}_{ρ}, $\rho \in \{\uparrow, \downarrow\}$, is defined in the module `dqmc_gtau`. Unlike equal time Green's functions, G^{τ} are not essential in the simulation. Therefore, user needs to initialize it by calling `DQMC_Gtau_Init` explicitly. The construction of G^{τ} can be made in two ways. The first way is to call `DQMC_Gtau_Big`, which returns entire G^{τ}_{ρ}. The second method is to invoke `DQMC_MakeGtau`, which returns block submatrices of G^{τ}. Since the signs of unequal time Green's functions are the same as the equal times, one can obtain the signs from `Hub%G_up%sgn, Hub%G_dn%sgn`.

3. **Parameters of Hamiltonian**: Parameters, such as t, μ and U, are stored in the data type `Hubbard`. The access is straightforward.

4. **Geometry related information**: Geometry information, such as hopping matrix, can be obtained from the data type `struct`, which is introduced in section 3.2.

3.4.2 Binning

In order to reduce correlation and bias, measurements in QUEST need be grouped into bins. Currently, equal divided binning strategy is used, which means the measurements are evenly divided by the total number of bins. Measurements in the same bin are averaged. The total number of bins are stored in the variable `nbin`.

3.4.3 Statistics

There are two special properties for the physical measurements produced by DQMC method.

1. The distribution is not normal.

2. Measurements are weighted with signs of the determinants of Green's functions, for which the average needs be normalized by the average of signs.

QUEST uses *Jackknife* resampling technique in error estimation. Two subroutines are provided to perform the statistics: `DQMC_JackKnife` and `DQMC_SignJackKnife`. The former is for error estimation of signs; the latter is used for other measurements. Those subroutines are defined in module `dqmc_util`.

3.4.4 Fourier transformation

For unequal time measurements, there are two possible Fourier transformations to be applied: transformation on the real space and transformation on the time domain. For the transformation on real space, since it is geometry dependent, a transformation matrix `FT`, defined in `Struct`, is required.[18] Once the matrix is available, the transformation is just a matrix-matrix multiplication. In the module `dqmc_tdm`, QUEST provides a subroutine `DQMC_DCT_Space` for the space transformation.

The Fourier transformation on the time domain is an integration. The numerical procedure is

1. Refine the time grid.

2. Interpolate the refined data points.

3. Integrate on the interpolated data with Fourier coefficients.

[18] see section 3.2 for more details.

In step 1, QUEST evenly subdivides the time domain by the given parameter `nitvl`. In step 2, QUEST uses spline interpolation, which is supported by the subroutine `DQMC_Spline`. Step 3 requires an additional Fourier matrix, which can be generated from the subroutine `DQMC_Make_FTM`. The entire procedure is coded in the subroutine `DQMC_FT_Time`.

3.4.5 Printing

QUEST has two subroutines that prints out arrays of numbers. Subroutine `DQMC_Print_RealArry` prints out an array of real numbers; `DQMC_Print_ComplexArray` prints out an array of complex numbers. The title of the measurements and the labels of each array items are required for those two functions.

A Example programs

Eight example programs are included in the `EXAMPLE` directory. Here is an short introduction for each of them.

A.1 `test`

Program `test` demonstrates the simplest usage of the QUEST. Besides the timing commands, there is only one line in the program, which runs DQMC simulation on a two dimensional periodic rectangular lattice. In spite of its simplicity, this program can be configured dynamically for different lattice size, Hubbard parameters and execution iterations. Four sample configurations are accompanied within this program, as showing below.

Configuration	Lattice size	Time slice	Running time
`small.in`	4×4	12	2 second
`median.in`	8×8	48	198 second
`large.in`	16×16	96	15,629 second
`extra_large.in`	32×32	192	unknown

Their execution time also presented. This result is obtained by using checkerboard method, 1000 warmup steps and 5000 measurement steps, with Intel MKL BLAS/LAPACK library, on Intel Core 2 Duo 2.4G processors (but only run in one core).

A.2 `verify`

The `verify` program examines the correctness of the execution results of two special cases, single site ($t = 0$) and no Coulomb interaction ($U = 0$). Each test case runs through 9 configurations.

The correctness of those results are checked against the theoretical results. Statistically, 63.2 percent of the computed results are expected to have errors smaller than one standard error, and 86.5 percent of results should be within two standard errors. The verification program runs 1000 warm-up sweep and 5000 measurement sweep for $t = 0$ cases, which gives error bars of 0.33% on the energy and less than 0.22% on the spin correlation `SpinXX`.

A.3 `tdm`

This example program demonstrates how unequal time measurements can be performed. This program also shows the flexibility of using the subroutines in QUEST. Four sample configurations, the same those in `test`, are available for this program. The time dependent measurements are made every 10 measurement sweeps. Their execution time are summarized in Table A.3.

Configuration	Lattice size	Time slice	Running time
`small.in`	4×4	12	3 second
`median.in`	8×8	48	266 second
`large.in`	16×16	96	19446.18 second
`extra_large.in`	32×32	192	unknown

A.4 Others

- `parallel`: This program uses MPI to parallelize the measurement steps. The compilation of this program requires `mpif90`. The base compiler of `mpif90` should be the same as the one used in compiling library.

- `wrap`: This directory contains a module `dqmc_wrapper` and a program `tw`. The module wraps all computation components of QUEST into several simpler functions. The program `tw`, test wrapper, can perform equal time and unequal time measurements.

- `sheet`: This directory contains a module and programs for the multilayer lattice. Module `dqmc_sheet` defines the multilayer geometry. Program `meas1` and `meas2` provides new measurements for the multilayer structure.

- `geom`: The program inside this directory tests the general geometry method 3 mentioned in section 3.2.

- `DCA`: This is a project that interfacing QUEST with other programs. Basically, the entire project uses DQMC as a DCA solver. This demonstrates how to use QUEST as a library in other programs.