Documentation for the General QMC code

Martin Bercx

October 17, 2016

1 Theory part

1.1 Definition of the physical Hamiltonian and its implementation

The physical Hamiltonians that we can simulate have the general form:

$$\mathcal{H} = \sum_{s=1}^{N_{fl}} \sum_{x,y} c_{xs}^{\dagger} M_{xy} c_{ys} - \sum_{k=1}^{M} U_k \left[\sum_{s=1}^{N_{fl}} \sum_{x,y} \left(c_{xs}^{\dagger} T_{xy}^{(k)} c_{ys} - \alpha_k \right) \right]^2 . \tag{1}$$

The indices x, y are multi-indices that label degrees of freedom, containing lattice sites and spin states: $x = (i, \sigma)$, where $i = 1, \cdots N_{sites}$ and $\sigma = 1, \cdots N_{sun}$, so

$$\sum_{x,y} \equiv \sum_{i=1,j=1}^{N_{sites}} \sum_{\sigma=1,\sigma'=1}^{N_{sun}} . \tag{2}$$

Note, that we introduced *two* different labels for the number of spin states (flavours): N_{fl} and N_{sun}. The number of correlated sites which is a subset of all sites, is labelled by M ($M \leq N_{sites}$). Let us further define $N_{dim} = N_{sun} N_{sites}$ such that the matrices $\boldsymbol{M}$ and $\boldsymbol{T}^{(k)}$ are of dimension $N_{dim} \times N_{dim}$ (in the code for the Hubbard model: $N_{dim} = Latt\%N{=}N_{unitcells}$) N_{sites} is the total number of spacial vertices, so it can be the product of orbital sites per unit cell $N_{orbitals}$ and number of unit cells $N_{unitcells}$ of the underlying Bravais lattice. Check the definition of $N_{orbitals}$ in the code.

I suggest to use a more intuitive notation and to label the hopping matrix by T and the interaction matrix by V:

$$\mathcal{H} = \sum_{s=1}^{N_{fl}} \sum_{x,y} c_{xs}^{\dagger} T_{xy} c_{ys} - \sum_{k=1}^{M} U_k \left[\sum_{s=1}^{N_{fl}} \sum_{x,y} \left(c_{xs}^{\dagger} V_{xy}^{(k)} c_{ys} - \alpha_k \right) \right]^2 . \tag{3}$$

This notation is used from now on.

1.2 Structure of the matrices T and $V^{(k)}$ and their implementation

In general, the matrices $\mathbf{T}$ and $\mathbf{V}^{(k)}$ are sparse matrices. This property is used to minimize computational cost and storage. In the following, we discuss the implementation of the single-particle matrix representation $\mathbf{V}^{(k)}$ of the interaction operator. The same applies for the hopping matrix $\mathbf{T}$.

We denote a subset of N_{eff} (in the code, N_{eff} is called just N) degrees of freedom by the set $[z_1, \cdots z_{N_{eff}}]$ and define it to contain only vertices for which an interaction term is defined:

$$V_{xy}^{(k)} \neq 0 \quad \text{only if} \quad x, y \in [z_1^{(k)}, \cdots z_{N_{eff}^{(k)}}^{(k)}] . \tag{4}$$

We define the projection matrices $\mathbf{P}_V^{(k)}$ of dimension $N_{eff}^{(k)} \times N_{dim}$:

$$(P_V^{(k)})_{i,z} = \delta_{z_i^{(k)},z} \,, \tag{5}$$

where $i \in [1, \cdots N_{eff}^{(k)}]$ and $z \in [1, \cdots N_{dim}]$. The matrix operator $\boldsymbol{P}_V^{(k)}$ picks out the non-vanishing entries of $\boldsymbol{V}^{(k)}$, which are contained in the $(N_{eff}^{(k)} \times N_{eff}^{(k)})$ - dimensional matrix $\boldsymbol{O}_V^{(k)}$:

$$\boldsymbol{V}^{(k)} = \boldsymbol{P}_V^{(k)T} \boldsymbol{O}_V^{(k)} \boldsymbol{P}_V^{(k)} \,, \tag{6}$$

and

$$V_{xy}^{(k)} = (P_V^{(k)})_{ix} \left[O_V^{(k)}\right]_{ij} (P_V^{(k)})_{jy} = \sum_{i,j}^{N_{eff}^{(k)}} \delta_{z_i^{(k)},x} \left[O_V^{(k)}\right]_{ij} \delta_{z_j^{(k)},y} \,. \tag{7}$$

Comment that the P matrices have only one non-vanishing entry per column. To set the two-particle interaction part, we therefore have to specify the matrix elements $\left[O_V^{(k)}\right]_{ij}$, the set $[z_1^{(k)}, \cdots z_{N_{eff}^{(k)}}^{(k)}]$, and the values U_k and α_k. Be more specific here what really has to specified in the actual code. The same logic also applies to the implementation of the hopping interaction be more specific.

1.3 Implementation: the `Operator` variable

In the code implementation, we define a structure called `Operator`. This structure variable `Operator` bundles several components that are needed to define and use an operator matrix in the program. In Fortran a structure variable like this is called a derived type. The components it contains are: the projector $\boldsymbol{P}_V$, the matrix $\boldsymbol{O}_V$, the effective dimension N_{eff} and a couple of auxiliary matrices and scalars. In general, we will not only have one structure variable `Operator`, but a whole array of these structures.

Name of variable in the code	Description
`Op_X%N`	effective dimension N_{eff}
`Op_X%O`	matrix $\mathbf{O}_V$
`Op_X%U`	matrix containing the eigenvectors of $\mathbf{O}_V$
`Op_X%E`	eigenvalues of $\mathbf{O}_V$
`Op_X%P`	projection matrix $\mathbf{P}_V$
`Op_X%N_non_zero`	number of non-vanishing eigenvalues of $\mathbf{O}_V$
`Op_X%g`	coupling
`Op_X%alpha`	constant

Table 1: Components of the `Operator` structure variable `Op_V`.

2 Tutorial to set up the $SU(2)$-Hubbard model on a square lattice

The $SU(2)$ symmetric Hubbard model is given by

$$\mathcal{H} = -t \sum_{\langle i,j \rangle,\sigma} \left(c_{i,\sigma}^\dagger c_{j,\sigma} + \text{H.c.} \right) + \frac{U}{2} \sum_i \left[\sum_\sigma \left(c_{i\sigma}^\dagger c_{i\sigma} - 1/2 \right) \right]^2 \,. \tag{8}$$

To bring Eq. (8) in the general form (3), we set

$$\begin{aligned} N_{fl} &= 1 \\ N_{sun} &= 2 \\ T_{xy} &= -t\delta_{\langle i,j\rangle}\delta_{\sigma,\sigma'} \\ M &= N_{sites} \\ U_k &= -U/2 \\ V_{xy}^{(k)} &= \delta_{x,y}\delta_{i,k} = \delta_{i,j}\delta_{\sigma,\sigma'}\delta_{i,k} \\ \alpha_k &= 1/(N_{sites}N_{sun})^2 = 1/(2N_{sites})^2 \ . \end{aligned} \tag{9}$$

In the following, we skip the N_{sun}-spin degree of freedom which is present in the multi-indices x, y of the matrices $\boldsymbol{T}$ and $\boldsymbol{V}^{(k)}$. So $N_{dim} = N_{sites}$. Is the code limited to $SU(N)$ symmetric models with respect to the N_{sun} degree of freedom? So both $\boldsymbol{T}$ and $\boldsymbol{V}^{(k)}$ have dimension N_{sites}. Note that in this example $N_{dim} = Latt\%N$ since there is only one spacial orbital per unit cell of the underlying Bravais lattice

2.1 Hopping interaction

The hopping matrix is implemented as follows. We allocate an array of dimension 1×1, called `Op_T`. It therefore contains only a single `Operator` structure. We set the effective dimension (here, $N_{eff} = N_{dim}$), and allocate and initialize this structure by a single call to the subroutine `Op_make`:

```
call Op_make(Op_T(1,1),Ndim)
```

Since the effective dimension is identical to the total dimension, it follows trivially, that $\boldsymbol{P}_T = \mathbb{1}$ and $\boldsymbol{O}_T = (\boldsymbol{T}_{ij})$ Although a checkerboard decomposition is not yet used for the Hubbard model, in principle it can be implemented.

2.2 Two-particle interaction

To implement this interaction, we allocate an array of `Operator` structures. The array is called `Op_V` and has dimensions $N_{dim} \times 1$. We set the effective dimension ($N_{eff} = 1$), and allocate and initialize this array of structures by calling the subroutine `Op_make`:

```
do i = 1, Latt%N
call Op_make(Op_V(i,1),1)
enddo
```

For each lattice site i, the projection matrices $\boldsymbol{P}_V^{(i)}$ are of dimension $1 \times N_{dim}$ and have one non-vanishing entry: $(P_V^{(i)})_{1j} = \delta_{ij}$. The effective matrices are again trivial: $\boldsymbol{O}_V^{(i)} = 1$.

Name of variable in the code	Description
`Ndim`	spacial dimension of the lattice (total number of sites)
`Latt%N`	Number of unit cells of the underlying Bravais lattice
`Op_T`	Array of structure variables that bundles all variables needed to define the hopping operator.
`Op_V`	Array of structure variables that bundles all variables needed to define the two-particle interaction operator.
`Nsun`	Number of spin degrees
`Nfl`	Number of spin flavors

Table 2: Common variables that are set in the Hamiltonian, operator and lattice modules of the code.

To do next:

- dicuss the measurements: what observables exit and how do I add a new one?
- discuss the implementation of the lattice.

3 Using the code

Example simulation, tutorial: where to find and how to start

3.1 Parameter files

describe the input parameters, give sample values for the stabilization parameters

3.2 Analysis files

how the analysis of Monte Carlo data is done

4 List of files

all files that constitute the code, with a brief description

4.1 cgr1.f90 & cgr2.f90

Stable computation of the physical single-particle equal time Green function $G(\tau)$.

4.2 control_mod.f90

Includes a set of auxiliary routines, regarding the flow of the simulation. Examples are initialization of performance variables, precision tests and controlled termination of the code.

4.3 gperp.f90

4.4 Hamiltonian_Hub.f90

Here, the physical simulation parameters (the model parameters) and the lattice parameters are read in. The lattice, the non-interacting and the interacting part of the Hubbard Hamiltonian are set according to the parameters and the chosen Hubbard-Stratonovich decomposition.

4.5 Hop_mod.f90

4.6 inconfc.f90

The auxiliary-field QMC method is based on a Hubbard-Stratonovich decomposition of the interaction term. This decomposition introduces a space-time array of (discrete) configurations of auxiliary fields, i.e. Ising spins. In this routine, an existing configuration is read in, checks on its dimensionality are made and, in case no prior configuration exists, a random configuration of Ising spins is set up.

4.7 main.f90

Top-level part of the program. Here, the program flow which consists of initialization, sweeps through the space-time lattice, and finalizing the program, is coded.

4.8 nranf.f90

Auxiliary routine controlling the evaluation of random numbers.

4.9 Operator.f90

The algorithms is centered around evaluation of single-particle operators, represented as square matrices. In this routine, the abstract type Operator is defined, including information on the coupling strength, the sites that participate in the single-particle hopping process, and the type of Hubbard-Stratonovich transformation. This routine collects all program relevant operations that are applied to the type Operator, like initializations or multiplications.

4.10 outconfc.f90

```
Description in plain text:
At the end of the simulation, the last configuration of Hubbard-Stratonovich variables,
together with the last set of random numbers  is written to the file confout.
Prior to the start of a new simulation of the identical space-time dimesion,
one can (manually) copy the file confout to the file confin and
make the new run use the old configuration.
Doing this saves warmup time compared to a complete random (unphysical) configuration.

Input/output variables
in:
inout:
inout:
out:

Dependencies
include: mpif.h
modules: Hamiltonian
interfaces:
global variables:  ntrot, nsigma
subroutines:  MPI_COMM_RANK, MPI_COMM_SIZE, Get_seed_Len, Ranget

Things to check:
Rename subroutine to confout.f90 for consistency
```

4.11 print_bin_mod.f90

```
Description in plain text:
Here the way to write the measure bins to the respective output files is coded.
A bin is an average over many individual measurements.
The bin defines the unit of Monte Carlo time.

Dependencies
include:
modules:
interfaces: Print_bin

Contains
subroutines: Print_bin_c, Print_bin_r, Print_scal, Print_bin_tau

Things to check:
```

4.11.1 Print_bin_C

```
Description in plain text:

Input/output variables
in: Latt, type(Lattice)
in: Phase_bin_tmp, complex
in: File_pr, character(Len=64)
in: nobs, integer
inout: Dat_eq,(:,:,:), complex
inout: Dat_eq0(:), complex
out:

Dependencies
include: mpif.h
modules: Lattices_v3
interfaces:
global variables:  TYPE(LATTICE), N, listk, b1_p, b2_p
subroutines:  MPI_COMM_RANK, MPI_COMM_SIZE, MPI_REDUCE, Fourier_R_to_k

Things to check:
STATUS(MPI_STATUS_SIZE), integer,  needed?
```

4.11.2 Print_bin_R

```
Description in plain text:

Input/output variables
in: Latt, type(Lattice)
in: Phase_bin_tmp, complex
in: File_pr, character(Len=64)
in: nobs, integer
inout: Dat_eq,(:,:,:), real
inout: Dat_eq0(:), real
out:

Dependencies
include: mpif.h
```

```
modules: Lattices_v3
interfaces:
global variables:  TYPE(LATTICE), N, listk, b1_p, b2_p
subroutines:  MPI_COMM_RANK, MPI_COMM_SIZE, MPI_REDUCE, Fourier_R_to_k

Things to check:
STATUS(MPI_STATUS_SIZE), integer,  needed?
```

4.11.3 Print_scal

```
Description in plain text:

Input/output variables
in: File_pr, character(Len=64)
in: nobs, integer
inout: Obs,(:), complex
out:

Dependencies
include: mpif.h
interfaces:
global variables:
subroutines:  MPI_COMM_RANK, MPI_COMM_SIZE, MPI_REDUCE

Things to check:
STATUS(MPI_STATUS_SIZE), integer,  needed?
change subroutine name to print_bin_scal for consistency
```

4.11.4 Print_bin_tau

```
Description in plain text:

Input/output variables
in: Latt, type(Lattice)
in: Phase_bin, complex
in: File_pr, character(Len=64)
in: nobs, integer
in: dtau, real
inout: Dat_tau,(:,:,:,:), complex
inout, optional: Dat0_tau(:), complex
out:

Dependencies
include: mpif.h
modules: Lattices_v3
interfaces:
global variables:  TYPE(LATTICE), N, listk, b1_p, b2_p
subroutines:  MPI_COMM_RANK, MPI_COMM_SIZE, MPI_REDUCE, Fourier_R_to_k

Things to check:
STATUS(MPI_STATUS_SIZE), integer,  needed?
```

4.12 tau_m.f90

```
Description in plain text:
module tau_m_mod, with several subroutines

Dependencies
include:
modules: Hamiltonian, Operator_mod, Precdef, Control, Hop_mod

Contains
subroutines: tau_m, propr, proprm1

Things to check:
change file name to tau_m_mod.f90 for consistency
```

4.12.1 TAU_M

```
Description in plain text:

Input/output variables
in: nstm, integer
in: nwrap, integer
in: ust(ndim,ndim; nstm,n_fl), complex
in: vst(ndim,ndim,nstm,n_fl), complex
in: dst(ndim,nstm,n_fl), complex
in: GR(ndim,ndim,n_fl), complex
in: phase, complex
in: stab_nt(0:nstm), integer
inout:
inout:
out:

Dependencies
include:
modules:
interfaces: wrapul, cgr2_1, cgr2_2, cgr2
global variables:  ndim, n_fl, cone, ltrot
subroutines:  obsert, initd,propr, proprm1, wrapur, cgr2_2, Control_Precision

Things to check:
cone?
```

4.12.2 propr

```
Description in plain text:

Input/output variables
in: nt
inout: Ain(ndim,ndim,n_fl)
inout:
out:

Dependencies
include:
modules:
```

```
interfaces:
global variables: ndim, n_fl, op_v, nsigma, Phi, type
subroutines:  Hop_mod_mmthr, Op_mmultR

Things to check:
```

4.12.3 proprm1

```
Description in plain text:

Input/output variables
in: nt
inout: Ain(ndim,ndim,n_fl)
inout:
out:

Dependencies
include:
modules:
interfaces:
global variables: ndim, n_fl, op_v, nsigma, Phi, type
subroutines:  Hop_mod_mmthl, Op_mmultL

Things to check:
```

4.13 UDV_WRAP.f90

```
Description in plain text:
UDV_Wrap_mod is a module file, containing subroutines on the stabilization of
matrix computations.

Dependencies
modules: MyMats, Files_mod

Contains
subroutines: UDV_Wrap_Pivot, UDV_Wrap

Things to check:
change file name to  UDV_Wrap_mod.f90 for consistency
```

4.13.1 UDV_Wrap_Pivot

```
Description in plain text:

Input/output variables
in: A(:,:), complex
in: ncon, integer
in: n1, integer
in: n2, integer
inout: U(:,:), complex
inout: V(:,:), complex
inout: D(:), complex
out:

Dependencies
include:
modules:
interfaces:
global variables:
subroutines:  UDV_Wrap, MMULT, Compare

Things to check:
```

4.13.2 UDV_Wrap

```
Description in plain text:

Input/output variables
in: A(:,:), complex
in: ncon, integer
inout: U(:,:), complex
inout: V(:,:), complex
inout: D(:), complex
out:

Dependencies
include: mpif.h
modules:
interfaces:
global variables:
```

```
subroutines:  MPI_COMM_SIZE, MPI_COMM_RANK, QR, SVD, MMULT

Things to check:
STATUS(MPI_STATUS_SIZE), integer: not used
```

4.14 upgrade.f90

```
Description in plain text:
The update of the Hubbard-Stratonovich configuration is done sequentially
for each point in the space-time lattice,
i.e. one Hubbard-Stratonovich Ising spin after the other.
In this routine, an update (i.e. a spin flip) is accepted or rejected.
The decision is made using the Metropolis method of importance sampling.

Input/output variables
in: N_op, integer
in: nt, integer
in: OP_dim, integer
inout: GR(ndim,ndim,n_fl), complex
inout: Phase, complex
out:

Dependencies
include:
modules: Hamiltonian, Random_wrap, Control, Precdef
interfaces:
global variables: ndim, n_fl, op_v, nflipl, Phi, n_non_zero, Gaml, P, nsigma, g, alpha, type, E
subroutines:  zgemm, control_upgrade

Things to check:
nranf, integer, external (where is external fct. defined)
log, logical (reserved name)
alpha:  both a local and a global variable. CHECK!!
```

4.15 wrapgrdo.f90

```
Description in plain text:
Single-particle equal-time Green functions are the central object of the code.
The physical single-particle equal-time Green function $G(\tau)$ is updated in wrapgrdo.f90
(down propagation, from $\tau=LTROT$ to $\tau=0$).
The update is sequentially, over all (interacting) lattice sites or lattice bonds.

Input/output variables
in: ntau, integer
inout: gr (ndim,ndim,n_fl), complex
inout: phase, complex
out:

Dependencies
include:
modules: Hamiltonian, MyMats, Hop_mod
interfaces: upgrade
global variables: op_v, phi, nsigma, ndim, n_fl
subroutines: Hop_mod_mmthl, Hop_mod_mmthr_m1, Op_Wrapdo, Upgrade

Things to check:
```

4.16 wrapgrup.f90

```
Description in plain text:
Single-particle equal-time Green functions are the central object of the code.
The physical single-particle equal-time Green function $G(\tau)$ is updated in wrapgrup.f90
(up propagation, from $\tau=0$ to $\tau=LTROT-1$).
The update is sequentially, over all (interacting) lattice sites or lattice bonds.

Input/output variables
in: ntau, integer
inout: gr (ndim,ndim,n_fl), complex
inout: phase, complex
out:

Dependencies
include:
modules: Hamiltonian, Hop_mod
interfaces: upgrade
global variables: op_v, phi, nsigma, ndim, n_fl
subroutines: Hop_mod_mmthr, Hop_mod_mmthl_m1, Op_Wrapup, Upgrade

Things to check:
```

4.17 wrapul.f90

```
Description in plain text:
To stabilize the simulation at the time slice $\tau_{2}=i n_{stab}$,
the Green function has to be recomputed regularly,
based on the stable matrices at an earlier stabilization point, $\tau_{1}=(i-1) n_{stab}$.
These stable matrices result from a singular-value-decomposition of the propagation matrix.
They are computed in wrapul.f90  (down propagation).

Input/output variables
in: ntau1, integer
in: ntau, integer
inout: ulup (ndim,ndim,n_fl), complex
inout: dlup (ndim,n_fl), complex
inout: vlup (ndim,ndim,n_fl), complex
out:

Dependencies
include:
modules: Hamiltonian, Hop_mod, UDV_Wrap_mod
interfaces:
global variables: ndim, n_fl, Op_V, Phi, nsigma,
subroutines:  initd, Op_mmultL, Hop_mod_mmthl, mmult, UDV_Wrap

Things to check:
```

4.18 wrapur.f90

```
Description in plain text:
To stabilize the simulation at the time slice $\tau_{2}=i n_{stab}$,
the Green function has to be recomputed regularly,
based on the stable matrices at an earlier stabilization point, $\tau_{1}=(i-1) n_{stab}$.
These stable matrices result from a singular-value-decomposition of the propagation matrix.
They are computed in wrapur.f90  (up propagation).

Input/output variables
in: ntau1, integer
in: ntau, integer
inout: ur (ndim,ndim,n_fl), complex
inout: dr (ndim,n_fl), complex
inout: vr (ndim,ndim,n_fl), complex
out:

Dependencies
include:
modules: Hamiltonian, Hop_mod, UDV_Wrap_mod
interfaces:
global variables: ndim, n_fl, Op_V, Phi, nsigma,
subroutines:  initd, Op_mmultR, Hop_mod_mmthr, mmult, UDV_Wrap

Things to check:
```

```
Description in plain text:

Input/output variables
in:
inout:
inout:
out:

Dependencies
include:
modules:
interfaces:
global variables:
subroutines:

Things to check:
```

5 Module Hamiltonian

Detailed description of the module Hamiltonian since it will be modified by the users
The module contains the following subroutines:

5.1 ham_set

It calls the subroutines

- ham_latt
- ham_hop
- ham_v

It reads in the file

- parameters

It sets the variables `ltrot,n_fl,n_sun`. If compiled as a MPI-program, it broadcasts all variables that define the lattice, the model and the simulation process.

5.2 ham_latt

It sets the lattice, by calling the subroutine

- make_lattice(l1_p, l2_p, a1_p, a2_p, latt)

5.3 ham_hop

Setup of the hopping amplitudes between the vertices of the graph (lattice sites and unit cell orbitals). It calls the subroutines

- op_make(op_t(nc,n),ndim
- op_set(op_t(nc,n))

5.4 ham_v

It calls the subroutines

- op_make(op_v(i,nf),1)
- op_set(op_v(nc,nf))

5.5 s0(n,nt)

It is defined as $s0(n, nt) = 1.d0$. Why? It is superfluous.

5.6 alloc_obs(ltau)

Allocation of equal time and time-resolved quantities.

5.7 init_obs(ltau)

Initializes equal time and time-resolved quantities with zero.

5.8 obser(gr,phase,ntau)

Includes the definition of all equal-time observables (scalars and correlation functions) that are built from the single-particle Green function based on Wick's theorem.

5.9 pr_obs(ltau)

Output (print) of the observables.

5.10 obsert(nt,gt0,g0t,g00,gtt,phase)

Includes the definition of time-resolved observables that are built from the time-resolved single-particle Green function based on Wick's theorem.

6 Tutorial to set up a lattice

7 Installation

7.1 Dependencies

which software and libraries are needed and which version

- libraries: LAPACK, BLAS, EISPACK, NAG *They are included in the package, but NAG is not public-domain (?)*
- tools: cmake
- compiler: gfortran or ifort

7.2 Build the GQMC program from source code

configuration, compile and Installation In the top level directory, where the README file resides, do:

```
mkdir build
cd build
cmake ..
make
```

8 Reference manual

9 License

Use of the GQMC code requires citation of the paper ... The GQMC code is available for academic and non-commercial use under the terms of the license ... For commercial licenses, please contact the GQMC development team.

10 ideas

FAQ, walkthroughs,