ALF Tutorial

The *ALF* (*A*lgorithms for *L*attice *F*ermions) project release 2.0 tutorial

Florian Goth, Johannes S. Hofmann, Jonas Schwab,
Jefferson S. E. Portela, Fakher F. Assaad

August 22, 2020

The ALF package provides a general code for auxiliary-field Quantum Monte Carlo simulations and default analysis. In this tutorial we show how users from beginners to specialists can profit from ALF. This document is divided in two parts:

Part I. The first, introductory part of the tutorial is based on ALF's python interface – `pyALF` – which greatly simplifies using the code, making it ideal for: *obtaining benchmark* results for established models; *getting started* with QMC and ALF; or just *quickly running* a simulation.

Part II. The second part is independent of the first and aimed at more advanced users who want to simulate their own systems. It guides the user on how to modify the package's Fortran source code and presents the resources implemented to facilitate this task.

This document is intended to be self-contained, but the interested reader should check ALF's documentation, which contains a thorough, systematic description of the package.

[Update link when ALF 2 is open.]

Part I. Just run it

What follows is a collection of self-explanatory Jupyter notebooks written in Python, each centered on a detailed example followed by a few simple exercises. The notebooks printed bellow can be found, together with the necessary files and an increasing number of additional notebooks exploring ALF's capabilities, in the pyALF repository.

Requirements

You can download pyALF from the its repository linked above, or simply run from the command line:

```
git clone git@git.physik.uni-wuerzburg.de:ALF/pyALF.git
```

To run the notebooks you need the following installed in your machine:

- Python
- Jupyter
- the libraries Lapack and Blas
- a Fortran compiler, such as `gfortran` or `ifort`,

where the last two are required by the main package ALF. Also, add pyALF's path to your environment variable `PYTHONPATH`. In Linux, this can be achieved, e.g., by adding the following line to `.bashrc`:

```
export PYTHONPATH="/local/path/to/pyALF:$PYTHONPATH"
```

Notice that `Run.py` assumes the existence of the configuration file `Sims`, which defines the simulation parameters. An entry of `Sims` might read as:

```
{"Model": "Hubbard", "Lattice_type": "Square", "L1": 4 , "L2": 4, "NBin": 5, "ham_T":
   ↪  0.0, "Nsweep" : 2000, "Beta": 1.0, "ham_chem": -1.0 }
```

Starting

Jupyter notebooks are run through a Jupyter server started, e.g., from the command line:

```
jupyter notebook
```

(or, depending on the installation, `jupyter-notebook`) which opens the "notebook dashboard" in your default browser, where you can navigate through your file structure to the pyALF directory. There you will find the interface's core module, `py_alf.py`, some auxiliary files, and notebooks such as the ones included bellow. Have fun.

Notebooks

1. A minimal ALF run

In this bare-bones example we use the pyALF interface to run the canonical Hubbard model on a default configuration: a 6×6 square grid, with interaction strength $U = 4$ and inverse temperature $\beta = 5$.

Bellow we go through the steps for performing the simulation and outputting observables.

---

1. Import `Simulation` class from the `py_alf` python module, which provides the interface with ALF:

[1]:
```
from py_alf import Simulation            # Interface with ALF
```

2. Create an instance of `Simulation`, setting parameters as desired:

[2]:
```
sim = Simulation(
    "Hubbard",                           # Hamiltonian
    {                                    # Model and simulation parameters for each Simulation instance
    "Model": "Hubbard",                  # Base model
    "Lattice_type": "Square"},           # Lattice type
)
```

3. Compile ALF, downloading it first from the ALF repository if not found locally. This may take a few minutes:

[3]:
```
sim.compile()                            # Compilation needs to be performed only once
```

```
Repository /home/stafusa/ALF/pyALF/Notebooks/ALF does not exist, cloning from
git@git.physik.uni-wuerzburg.de:ALF/ALF.git
Compiling ALF... Done.
```

4. Perform the simulation as specified in `sim`:

[4]:
```
sim.run()                                # Perform the actual simulation in ALF
```

```
Prepare directory "/home/stafusa/ALF/pyALF/Notebooks/Hubbard_Square" for Monte
Carlo run.
Create new directory.
Run /home/stafusa/ALF/pyALF/Notebooks/ALF/Prog/Hubbard.out
```

5. Perform some simple analyses:

[5]:
```
sim.analysis()                           # Perform default analysis; list observables
```

```
Analysing Ener_scal
Analysing Part_scal
Analysing Pot_scal
Analysing Kin_scal
Analysing Den_eq
Analysing SpinZ_eq
Analysing Green_eq
Analysing SpinXY_eq
Analysing SpinT_eq
Analysing SpinXY_tau
Analysing SpinZ_tau
Analysing Den_tau
Analysing Green_tau
Analysing SpinT_tau
```

6. Store computed observables list:

[6]:
```
obs = sim.get_obs()                    # Dictionary for the observables
```

which are available for further analyses. For instance, the internal energy of the system (and its error) is accessed by:

[7]:
```
obs['Ener_scalJ']['obs']
```

[7]:
```
array([[-29.983503,   0.232685]])
```

---

7. Running again: The simulation can be resumed to increase the precision of the results.

[8]:
```
sim.run()
sim.analysis()
obs2 = sim.get_obs()
print(obs2['Ener_scalJ']['obs'])
print("\nRunning again reduced the error from ", obs['Ener_scalJ']['obs'][0][1]," to ",␣
↪obs2['Ener_scalJ']['obs'][0][1], ".")
```

```
Prepare directory "/home/stafusa/ALF/pyALF/Notebooks/Hubbard_Square" for Monte
Carlo run.
Resuming previous run.
Run /home/stafusa/ALF/pyALF/Notebooks/ALF/Prog/Hubbard.out
Analysing Ener_scal
Analysing Part_scal
Analysing Pot_scal
Analysing Kin_scal
Analysing Den_eq
Analysing SpinZ_eq
Analysing Green_eq
Analysing SpinXY_eq
Analysing SpinT_eq
Analysing SpinXY_tau
Analysing SpinZ_tau
Analysing Den_tau
Analysing Green_tau
Analysing SpinT_tau
[[-29.819654   0.135667]]

Running again reduced the error from  0.232685  to  0.135667 .
```

Note: To run a fresh simulation - instead of performing a refinement over previous run(s) - the Monte Carlo run directory should deleted before rerunning.

1.1. Exercises

1. Rerun once again and check the new improvement in precision.
2. Look at a few other observables (`sim.analysis()` outputs the names of those available).

3. Change the lattice size by adding, e.g., `"L1": 4,` and `"L2": 1,` to the simulation parameters definitions of `sim` (step 2).

Part II. Getting your hands dirty - changing the code

Part II consists in a set of guided advanced exercises.

A lot already comes implemented in ALF, but unavoidably, as one proceeds in their own investigations, a new model has to be implemented or a new observable defined – and for that one has to grapple with the package's Fortran source code. However, in ALF this is made easy by means of predefined structures, templates, and the examples below.

Downloading the code and tutorial

To download the code, type `git clone git@git.physik.uni-wuerzburg.de:ALF/ALF_code.git` in a shell.
To download the tutorial including solutions type:
`git clone git@git.physik.uni-wuerzburg.de:ALF/ALF_Tutorial.git` again in a shell.

Exercise 1 – Dimensional crossover

[To be updated.]

1a) Modifying the hopping

Here we will modify the code so as to allow for different hopping matrix elements along the x and y directions of a square lattice. To do so, one merely has to do the following

- Add an extra variable, `Ham_Ty`, in the parameter file in the `VAR_Hubbard` name space
- Declare the variable `Ham_Ty` in the `Hamiltonian_Examples.f90` .
- Read in this variable in `Ham_set` subroutine of the `Hamiltonian_Examples.f90` file.
- Modify the hopping matrix in the subroutine `Ham_Hop` in the `Hamiltonian_Examples.f90` file.

```
 DO I = 1, Latt%N
    I1 = Latt%nnlist(I,1,0)
    I2 = Latt%nnlist(I,0,1)
    Op_T(nc,n)%O(I,I1) = cmplx(-Ham_T,    0.d0, kind(0.D0))
    Op_T(nc,n)%O(I1,I) = cmplx(-Ham_T,    0.d0, kind(0.D0))
!!!!!! Modifications for Exercise 2 (a)
     !Op_T(nc,n)%O(I,I2) = cmplx(-Ham_T,    0.d0, kind(0.D0))
     !Op_T(nc,n)%O(I2,I) = cmplx(-Ham_T,    0.d0, kind(0.D0))
     Op_T(nc,n)%O(I,I2) = cmplx(-Ham_Ty,    0.d0, kind(0.D0))
     Op_T(nc,n)%O(I2,I) = cmplx(-Ham_Ty,    0.d0, kind(0.D0))
!!!!!!
      Op_T(nc,n)%O(I ,I) = cmplx(-Ham_chem, 0.d0, kind(0.D0))
ENDDO
```

In the directory `Solutions/Exercise_2` we have duplicated the ALF and commented the changes that have to be carried out to the file `Hamiltonian_Examples.f90` in the `Prog` directory.

As an application of this code, one can consider a ladder system, defined by the parameter file:

```
 !============================================================================
!  Variables for the Hubb program
!----------------------------------------------------------------------------

&VAR_lattice
L1    = 14
L2    = 2
Lattice_type = "Square"
Model = "Hubbard_Mz"  ! Hubbard_SU2, Hubbard_Mz
/

&VAR_Hubbard
ham_T   = 1.0D0
!!!!!! Modifications for Exercise 2
ham_Ty  = 2.0D0
!!!!!!
ham_chem= 0.0D0
ham_U   = 4.0
Beta    = 10.0
dtau    = 0.1
/

&VAR_QMC
Nwrap   = 10
NSweep  = 100
NBin    = 10
Ltau    = 1
LOBS_ST = 1
LOBS_EN = 100
CPU_MAX = 0.1
/

&VAR_errors
n_skip  = 2
N_rebin = 1
N_Cov = 0
/
! slash terminates namelist statement - DO NOT REMOVE
```

When running the code for the above ladder system and analyzing the spin correlation functions, `SpinZ_eqJR` and `SpinXY_eqJR` one will notice that it is hard restore the SU(2) spin symmetry and that very long runs are required to obtain the desired equality

$$\langle S_i^z S_j^z \rangle = \langle S_i^y S_j^y \rangle = \langle S_i^x S_j^x \rangle. \tag{1}$$

The structure of the output files `SpinZ_eqJR` and `SpinXY_eqJR` is described in the doucmentation. For the Mz Hubbard-Stratonovitch transformation it is hence better to consider the improved estimator

$$\langle \vec{S}_i \cdot \vec{S}_j \rangle \tag{2}$$

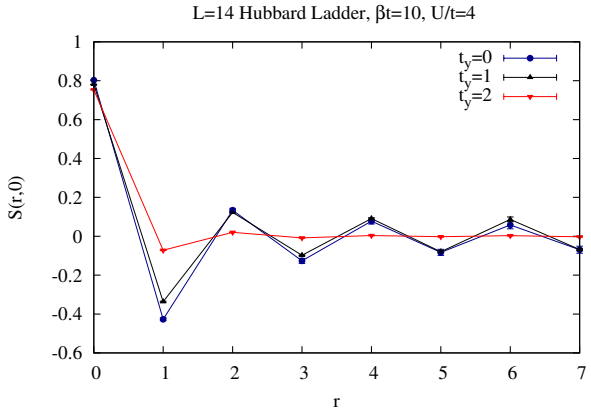
to compute the spin-spin correlations.

1b) Adding a new observable

Here the aim is to include the new observable equal time observable $\langle \vec{S}_i \cdot \vec{S}_j \rangle$ in the `Hubbard_Mz` code. To achieve this, you will have to carry out the following steps.

- In the subroutine `Alloc_obs` in the `Hamiltonian_example.f90` file you will have to add a new equal time observable with a call to `Call Obser_Latt_make(Obs_eq(I),Ns,Nt,No,Filename)` with `Ns = Latt%N; No = Norb; Filename ="SpinT", Nt=1, I=5`
- In the subroutine `Obser` you will have to add the Wick decomposition of this observable.

In the program `Hamiltonian_Examples.f90` to be found in the directory `Solutions/Exercise_2/Prog/` we have commented the changes that have to be carried out to add this observable. The new variable takes the name SpinT and the results you should obtain are summarized in Fig. **??**.



1c) The SU(2) Hubbard-Stratonovich transformation

The SU(2) Hubbard-Stratonovich decomposition, conserves spin rotational symmetry. Run the ladder code with the SU(2) flag in the parameter file switched on (i.e. `Model = Hubbard_SU2`) and compare results.

Exercise 2 – Defining a new model: The one-dimensional t-V model

[To be updated.]

2a) Define new model

In this section, one we will show what modifications have to be carried out for computing the physics of the one dimensional t-V model of spinless fermions.

$$H = -t \sum_i \left(c_i^\dagger c_{i+a} + c_{i+a}^\dagger c_i \right) - \frac{V}{2} \sum_i \left(c_i^\dagger c_{i+a} + c_{i+a}^\dagger c_i \right)^2 \tag{3}$$

The above form is readily included in the ALF since the interaction is written in terms of a perfect square. Expanding the square yields (up to a constant) the desired model:

$$H = -t \sum_i \left(c_i^\dagger c_{i+a} + c_{i+a}^\dagger c_i \right) + V \sum_i \left(n_i - 1/2 \right) \left(n_{i+a} - 1/2 \right) \tag{4}$$

In the directory `Solutions/Exercise_3` we have duplicated the ALF and commented the changes that have to be carried out to the file `Hamiltonian_Examples.f90` in the `Prog` directory so as to include the t_V model. Here are the steps to be carried out.

- Add the $t-V$ name space in the parameter file so as to read in the appropriate variables.
- Declare new variables in the `Hamiltonian_Examples.f90` file.
- In the `Ham_set` subroutine of the file `Hamiltonian_Examples.f90` set and read in the parameters for the new model, `t_V`. For this model `NF=1` and `N_SUN=1` since we are working with spinless fermions
- In the `Ham_V` subroutine you will have to add the new interaction. For a given bond at a given time-slice, we need to decouple the interaction:

$$e^{\Delta\tau \frac{V}{2} \left(c_i^\dagger c_{i+a} + c_{i+a}^\dagger c_i \right)^2} = \sum_{l=\pm 1, \pm 2} \gamma_l e^{\sqrt{\Delta\tau \frac{V}{2}} \eta_l \left(c_i^\dagger c_{i+a} + c_{i+a}^\dagger c_i \right)} = \sum_{l=\pm 1, \pm 2} \gamma_l e^{g \eta_l \left(c_i^\dagger, c_{i+a}^\dagger \right) O \left(c_i, c_{i+a} \right)^T} \tag{5}$$

  Here is how this translates in the code.

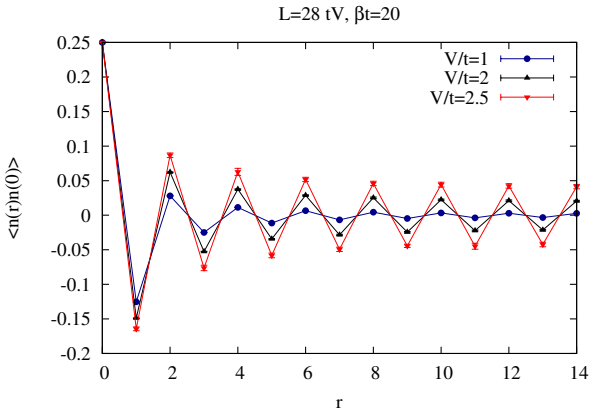
```
Allocate(Op_V(Latt%N,N_FL))
do nf = 1,N_FL
   do i  =  1, N_coord*Ndim
      call Op_make(Op_V(i,nf),2)
   enddo
enddo
Do nc = 1, Latt%N  ! Runs over bonds = # of lattice sites in one-dimension.
   I1 = nc
   I2 = Latt%nnlist(I1,1,0)
   Op_V(nc,1)%P(1) = I1
   Op_V(nc,1)%P(2) = I2
   Op_V(nc,1)%O(1,2) = cmplx(1.d0 ,0.d0, kind(0.D0))
   Op_V(nc,1)%O(2,1) = cmplx(1.d0 ,0.d0, kind(0.D0))
   Op_V(nc,1)%g      = SQRT(CMPLX( DTAU*Ham_Vint/2.d0, 0.D0, kind(0.D0)))
   Op_V(nc,1)%alpha  = cmplx(0d0,0.d0, kind(0.D0))
   Op_V(nc,1)%type =2
   Call Op_set( Op_V(nc,1) )
enddo
```

- Finally you will have to update the `Obser` and `ObserT` routines for the calculation of the equal and time displaced correlations. For the `t_V` model you can essentially use the same observables as for the `Hubbard_SU(2)` model.

You can now run the code for various values of V/t. A Jordan-Wigner transformation will map the `t_V` model onto the XXZ chain:

$$H = J_{xx} \sum_i S_i^x S_{i+a}^x + S_i^y S_{i+a}^y + J_{zz} \sum_i S_i^z S_{i+a}^z \tag{6}$$

with $J_{zz} = V$ and $J_{xx} = 2t$. Hence when $V/t = 2$ we reproduce the Heisenberg model. For $V/t > 2$ the model is in the Ising regime with long-range charge density wave order and is an insulator. In the regime $-2 < V/t < 2$ the model is metallic and corresponds to a Luttinger liquid. Finally, at $V/t < -2$ phase separation between hole rich and electron rich phases occur. Fig. **??** shows typical results.



2b) Challenge

How would you use the code to carry out simulations at $V/t < 0$?