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

September 18, 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.

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 its repository linked above or, 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 and packages SciPy, NumPy and matplotlib
- 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"
```

Python and its packages can be easily installed on a variety of platforms using the Anaconda distribution – check its installation instructions for your system. Then, from Anaconda, you can issue the command

```
conda install -c anaconda  ipython jupyterlab scipy numpy matplotlib
```

Anaconda is recommended due to its convenience, but the system's package management (e.g., apt-get) or Python's own package management, pip3, can be used instead if preferred – see, for instance, SciPy installation instructions.

A Fortran compiler and the libraries needed for ALF can be installed in a Debian-based Linux via

```
sudo apt-get install  gfortran liblapack-dev make
```

In MacOS, gfortran can be found at https://gcc.gnu.org/wiki/GFortranBinaries#MacOS, where detailed instructions are available. You will need to have Xcode as well as the Apple developer tools installed.

For Windows and other Linuxes and Unixes, please check the Tutorials' repository README.

Starting

Jupyter notebooks are run through a Jupyter server[1] 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.
```

[1] Note that pyALF can also be used to start a simulation from the command line, without starting a Jupyter server or using a notebook. For instance: `python3.7 Run.py -R -alfdir /home/debian/ALF-1.2/ -config "Intel" -executable_R Hubbard -mpi True` starts a parallel run of the Hubbard model, using ALF compiled with `ifort`. Notice that `Run.py` requires a 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`

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 be 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).

2. Trotter systematic error - Hubbard on the square lattice

In this example we use the pyALF interface to run ALF with the Mz choice of Hubbard-Stratonovich transformation (i.e., coupled to the z-component of the spin) on a 6×6 site square lattice, at $U/t = 4$ half-band filling, and inverse temperature $\beta t = 5$.

We carry out a systematic $\Delta\tau t$ extrapolation keeping $\Delta\tau t L_{\mathrm{Trotter}} = 2$ constant. Recall that the formulation of the auxiliary field QMC approach is based on the symmetric Trotter decomposition

$$e^{-\Delta\tau\left(\hat{A}+\hat{B}\right)} = e^{-\Delta\tau\hat{A}/2} e^{-\Delta\tau\hat{B}} e^{-\Delta\tau\hat{A}/2} + \mathcal{O}\left(\Delta\tau^3\right)$$

The overall error produced by this approximation is of the order $\Delta\tau^2$.

Bellow we go through the steps for performing this extrapolation: setting the simulation parameters, running it and analysing the data. A reference plot for this analyses is found in ALF documentation, Sec. 2.3.2 (Symmetric Trotter decomposition).

---

1. Import `Simulation` class from the `py_alf` python module, which provides the interface with ALF, as well as mathematics and plotting packages:

[1]:
```
from py_alf import Simulation            # Interface with ALF
#
import numpy as np                       # Numerical library
from scipy.optimize import curve_fit     # Numerical library
import matplotlib.pyplot as plt          # Plotting library
```

2. Create instances of `Simulation`, specifying the necessary parameters, in particular the different $\Delta\tau$ values:

[2]:
```
sims = []                                # Vector of Simulation instances
print('dtau values used:')
for dtau in [0.05, 0.1, 0.15]:           # Values of dtau
    print(dtau)
    sim = Simulation(
        'Hubbard',                       # Hamiltonian
        {                                # Model and simulation parameters for each Simulation instance
        'Model': 'Hubbard',              #    Base model
        'Lattice_type': 'Square',        #    Lattice type
        'L1': 6,                         #    Lattice length in the first unit vector direction
        'L2': 6,                         #    Lattice length in the second unit vector direction
        'Checkerboard': False,           #    Whether checkerboard decomposition is used or not
        'Symm': True,                    #    Whether symmetrization takes place
        'ham_T': 1.0,                    #    Hopping parameter
        'ham_U': 4.0,                    #    Hubbard interaction
        'ham_Tperp': 0.0,                #    For bilayer systems
        'beta': 5.0,                     #    Inverse temperature
        'Ltau': 0,                       #    '1' for time-displaced Green functions; '0' otherwise
        'NSweep': 200,                   #    Number of sweeps
        'NBin': 10,                      #    Number of bins
        'Dtau': dtau,                    #    Only dtau varies between simulations, Ltrot=beta/Dtau
        'Mz': True,                      #    If true, sets the M_z-Hubbard model: Nf=2, N_sum=1,
        },                               #             HS field couples to z-component of magnetization
        alf_dir='~/Programs/ALF',        # Local ALF copy, if present
```

```
    )
    sims.append(sim)
```

```
dtau values used:
0.05
0.1
0.15
```

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

[3]:
```
sims[0].compile()                          # Compilation needs to be performed only once
```

```
Compiling ALF... Done.
```

4. Perform the simulations, as specified in each element of `sim`:

[4]:
```
for i, sim in enumerate(sims):
    sim.run()                              # Perform the actual simulation in ALF
```

```
Prepare directory "/home/stafusa/ALF/pyALF/Hubbard_Square_L1=6_L2=6_Checkerboard
=False_Symm=True_T=1.0_U=4.0_Tperp=0.0_beta=5.0_Dtau=0.05_Mz=True" for Monte
Carlo run.
Resuming previous run.
Run /home/stafusa/Programs/ALF/Prog/Hubbard.out
Prepare directory "/home/stafusa/ALF/pyALF/Hubbard_Square_L1=6_L2=6_Checkerboard
=False_Symm=True_T=1.0_U=4.0_Tperp=0.0_beta=5.0_Dtau=0.1_Mz=True" for Monte
Carlo run.
Resuming previous run.
Run /home/stafusa/Programs/ALF/Prog/Hubbard.out
Prepare directory "/home/stafusa/ALF/pyALF/Hubbard_Square_L1=6_L2=6_Checkerboard
=False_Symm=True_T=1.0_U=4.0_Tperp=0.0_beta=5.0_Dtau=0.15_Mz=True" for Monte
Carlo run.
Resuming previous run.
Run /home/stafusa/Programs/ALF/Prog/Hubbard.out
```

5. Calculate the internal energies:

[5]:
```
ener = np.empty((len(sims), 2))            # Matrix for storing energy values
dtaus = np.empty((len(sims),))             # Matrix for Dtau values, for plotting
for i, sim in enumerate(sims):
    print(sim.sim_dir)                     # Directory containing the simulation output
    sim.analysis()                         # Perform default analysis
    dtaus[i] = sim.sim_dict['Dtau']                                  # Store Dtau value
    ener[i] = sim.get_obs(['Ener_scalJ'])['Ener_scalJ']['obs']  # Store internal energy
```

```
/home/stafusa/ALF/pyALF/Hubbard_Square_L1=6_L2=6_Checkerboard=False_Symm=True_T=
1.0_U=4.0_Tperp=0.0_beta=5.0_Dtau=0.05_Mz=True
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
/home/stafusa/ALF/pyALF/Hubbard_Square_L1=6_L2=6_Checkerboard=False_Symm=True_T=
1.0_U=4.0_Tperp=0.0_beta=5.0_Dtau=0.1_Mz=True
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
/home/stafusa/ALF/pyALF/Hubbard_Square_L1=6_L2=6_Checkerboard=False_Symm=True_T=
1.0_U=4.0_Tperp=0.0_beta=5.0_Dtau=0.15_Mz=True
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
```

[6]:
```
print('For Dtau values', dtaus, 'the measured energies are:\n', ener)
```

```
For Dtau values [0.05 0.1  0.15] the measured energies are:
 [[-29.714802   0.041044]
 [-29.784918   0.043263]
 [-29.818716   0.029992]]
```

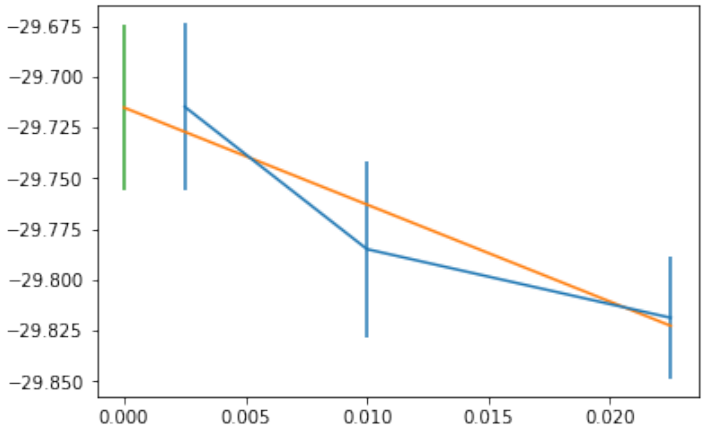
[7]:
```
plt.errorbar(dtaus**2, ener[:, 0], ener[:, 1])

def func(x, y0, a):
    return y0 + a*x**2
popt1, pcov = curve_fit(func, dtaus, ener[:, 0], sigma=ener[:, 1], absolute_sigma=True)
perr1 = np.sqrt(np.diag(pcov))
print(popt1, perr1)
xs = np.linspace(0., dtaus.max())
plt.plot(xs**2, func(xs, *popt1))

plt.errorbar(0, popt1[0], perr1[0])
```

```
[-29.71520822  -4.77614903] [0.04068013 2.44507809]
```

[7]:
```
<ErrorbarContainer object of 3 artists>
```



2.1. Exercises

1. Try out the four different combinations for `Checkerboard` and `Symm` settings in order to observe their effect on the output and run time. Reference: Sec. 2.3.2 - Symmetric Trotter decomposition - of the ALF documentation, especially Fig. 2.

3. Testing against ED - Hubbard on a ring

In this example we use the pyALF interface to run ALF with the Mz choice of Hubbard Stratonovitch transformation on a four site ring, at $U/t = 4$ and inverse temperature $\beta t = 2$. For this set of parameters, the exact internal energy reads:

$$\left\langle -t \sum_{\langle i,j\rangle,\sigma} \hat{c}^{\dagger}_{i,\sigma}\hat{c}_{j,\sigma} + U\sum_{i=1}^{N} \hat{n}_{i,\uparrow}\hat{n}_{j,\downarrow} \right\rangle = -1.47261997t$$

To reproduce this result we will have to carry out a systematic $\Delta\tau t$ extrapolation keeping $\Delta\tau t L_{\mathrm{Trotter}} = 2$ constant.
Recall that the formulation of the auxiliary field QMC approach is based on the Trotter decomposition

$$e^{-\Delta\tau\left(\hat{A}+\hat{B}\right)} = e^{-\Delta\tau\hat{A}/2}e^{-\Delta\tau\hat{B}}e^{-\Delta\tau\hat{A}/2} + \mathcal{O}\left(\Delta\tau^3\right)$$

The overall error produced by this approximation is of the order $\Delta\tau^2$.

Bellow we go through the steps for performing this extrapolation: setting the simulation parameters, running it and analyzing the data.

---

1. Import `Simulation` class from the `py_alf` python module, which provides the interface with ALF, as well as mathematics and plotting packages:

[1]:
```
from py_alf import Simulation            # Interface with ALF
#
import numpy as np                       # Numerical library
from scipy.optimize import curve_fit     # Numerical library
import matplotlib.pyplot as plt          # Plotting library
```

2. Create instances of `Simulation`, specifying the necessary parameters, in particular the different $\Delta\tau$ values:

[2]:
```
sims = []                                # Vector of Simulation instances
print('dtau values used:')
for dtau in [0.05, 0.1, 0.15]:           # Values of dtau
    print(dtau)
    sim = Simulation(
        'Hubbard',                       # Hamiltonian
        {                                # Model and simulation parameters for each Simulation instance
        'Model': 'Hubbard',              #    Base model
        'Lattice_type': 'N_leg_ladder',  #    Lattice type
        'L1': 4,                         #    Lattice length in the first unit vector direction
        'L2': 1,                         #    Lattice length in the second unit vector direction
        'Checkerboard': False,           #    Whether checkerboard decomposition is used or not
        'Symm': True,                    #    Whether symmetrization takes place
        'ham_T': 1.0,                    #    Hopping parameter
        'ham_U': 4.0,                    #    Hubbard interaction
        'ham_Tperp': 0.0,                #    For bilayer systems
        'beta': 2.0,                     #    Inverse temperature
        'Ltau': 0,                       #    '1' for time-displaced Green functions; '0' otherwise
        'NSweep': 400,                   #    Number of sweeps
        'NBin': 100,                     #    Number of bins
        'Dtau': dtau,                    #    Only dtau varies between simulations, Ltrot=beta/Dtau
        'Mz': True,                      #    If true, sets the M_z-Hubbard model: Nf=2, N_sum=1,
        },                               #             HS field couples to z-component of magnetization
        alf_dir='~/Programs/ALF',        # Local ALF copy, if present
    )
    sims.append(sim)
```

```
dtau values used:
0.05
0.1
0.15
```

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

[3]:
```
sims[0].compile()                        # Compilation needs to be performed only once
```

```
Compiling ALF... Done.
```

4. Perform the simulations, as specified in each element of `sim`:

[4]:
```
for i, sim in enumerate(sims):
    sim.run()                           # Perform the actual simulation in ALF
```

```
Prepare directory "/home/stafusa/ALF/pyALF/Hubbard_N_leg_ladder_L1=4_L2=1_Checke
rboard=False_Symm=True_T=1.0_U=4.0_Tperp=0.0_beta=2.0_Dtau=0.05_Mz=True" for
Monte Carlo run.
Resuming previous run.
Run /home/stafusa/Programs/ALF/Prog/Hubbard.out
Prepare directory "/home/stafusa/ALF/pyALF/Hubbard_N_leg_ladder_L1=4_L2=1_Checke
rboard=False_Symm=True_T=1.0_U=4.0_Tperp=0.0_beta=2.0_Dtau=0.1_Mz=True" for
Monte Carlo run.
Resuming previous run.
Run /home/stafusa/Programs/ALF/Prog/Hubbard.out
Prepare directory "/home/stafusa/ALF/pyALF/Hubbard_N_leg_ladder_L1=4_L2=1_Checke
rboard=False_Symm=True_T=1.0_U=4.0_Tperp=0.0_beta=2.0_Dtau=0.15_Mz=True" for
Monte Carlo run.
Resuming previous run.
Run /home/stafusa/Programs/ALF/Prog/Hubbard.out
```

5. Calculate the internal energies:

[5]:
```
ener = np.empty((len(sims), 2))         # Matrix for storing energy values
dtaus = np.empty((len(sims),))          # Matrix for Dtau values, for plotting
for i, sim in enumerate(sims):
    print(sim.sim_dir)                  # Directory containing the simulation output
    sim.analysis()                      # Perform default analysis
    dtaus[i] = sim.sim_dict['Dtau']                             # Store Dtau value
    ener[i] = sim.get_obs(['Ener_scalJ'])['Ener_scalJ']['obs']  # Store internal energy
```

```
/home/stafusa/ALF/pyALF/Hubbard_N_leg_ladder_L1=4_L2=1_Checkerboard=False_Symm=T
rue_T=1.0_U=4.0_Tperp=0.0_beta=2.0_Dtau=0.05_Mz=True
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
/home/stafusa/ALF/pyALF/Hubbard_N_leg_ladder_L1=4_L2=1_Checkerboard=False_Symm=T
rue_T=1.0_U=4.0_Tperp=0.0_beta=2.0_Dtau=0.1_Mz=True
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
/home/stafusa/ALF/pyALF/Hubbard_N_leg_ladder_L1=4_L2=1_Checkerboard=False_Symm=T
rue_T=1.0_U=4.0_Tperp=0.0_beta=2.0_Dtau=0.15_Mz=True
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
```

[6]:
```
print('For Dtau values', dtaus, 'the measured energies are:\n', ener)
```

```
For Dtau values [0.05 0.1  0.15] the measured energies are:
 [[-1.473383  0.002569]
 [-1.478316  0.002525]
 [-1.46798   0.002359]]
```

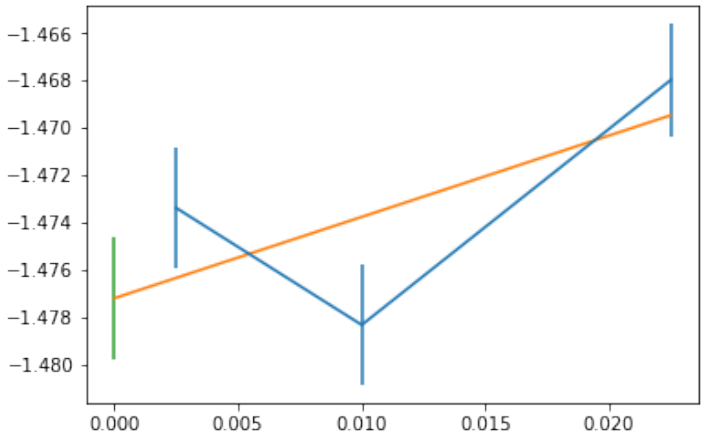
[7]:
```
plt.errorbar(dtaus**2, ener[:, 0], ener[:, 1])

def func(x, y0, a):
    return y0 + a*x**2
popt1, pcov = curve_fit(func, dtaus, ener[:, 0], sigma=ener[:, 1], absolute_sigma=True)
perr1 = np.sqrt(np.diag(pcov))
print(popt1, perr1)
xs = np.linspace(0., dtaus.max())
plt.plot(xs**2, func(xs, *popt1))

plt.errorbar(0, popt1[0], perr1[0])
```

```
[-1.47718877  0.3429858 ] [0.00254357 0.17130454]
```

[7]:
```
<ErrorbarContainer object of 3 artists>
```



3.1. Exercises

1. Redo the extrapolation for different values of βt (e.g., for $\beta t = 1$, the internal energy is -0.62186692t, and for $\beta t = 4$, it is -1.90837196t).
2. Experiment with different settings for `Checkerboard` and `Symm`.

4. Projective algorithm

In this example we use the pyALF interface to run ALF's projective algorithm with the Mz choice of Hubbard Stratonovich transformation on a 4-site ring.

The projective approach is the method of choice if one is interested in ground-state properties. The starting point is a pair of trial wave functions, $|\Psi_{T,L/R}\rangle$, that are not orthogonal to the ground state $|\Psi_0\rangle$:

$$\langle \Psi_{T,L/R}|\Psi_0\rangle \neq 0.$$

The ground-state expectation value of any observable $\hat{O}$ can then be computed by propagation along the imaginary time axis:

$$\frac{\langle \Psi_0|\hat{O}|\Psi_0\rangle}{\langle \Psi_0|\Psi_0\rangle} = \lim_{\theta\to\infty} \frac{\langle \Psi_{T,L}|e^{-\theta\hat{H}}e^{-(\beta-\tau)\hat{H}}\hat{O}e^{-\tau\hat{H}}e^{-\theta\hat{H}}|\Psi_{T,R}\rangle}{\langle \Psi_{T,L}|e^{-(2\theta+\beta)\hat{H}}|\Psi_{T,R}\rangle},$$

where β defines the imaginary time range where observables (time displaced and equal time) are measured and τ varies from 0 to β in the calculation of time-displace observables. For further details, see Sec. 3 of ALF documentation.

1. Import `Simulation` class from the `py_alf` python module, which provides the interface with ALF, as well as numerical and plotting packages:

[1]:
```
from py_alf import Simulation             # Interface with ALF
#
import numpy as np                        # Numerical library
from scipy.optimize import curve_fit      # Numerical library
import matplotlib.pyplot as plt           # Plotting library
```

2. Create instances of `Simulation`, specifying the necessary parameters, in particular the `Projector` to `True`:

[2]:
```
sims = []                                 # Vector of Simulation instances
print('Theta values used:')
for theta in [5, 10, 20]:                 # Values of Theta
    print(theta)
    sim = Simulation(
        'Hubbard',                        # Hamiltonian
        {                                 # Model and simulation parameters for each Simulation instance
        'Model': 'Hubbard',               #    Base model
        'Lattice_type': 'N_leg_ladder',   #    Lattice type
        'L1': 4,                          #    Lattice length in the first unit vector direction
        'L2': 1,                          #    Lattice length in the second unit vector direction
        'Checkerboard': False,            #    Whether checkerboard decomposition is used or not
        'Symm': True,                     #    Whether symmetrization takes place
        'Projector': True,                #    Whether to use the projective algorithm
        'Theta': theta,                   #    Projector parameter
        'ham_T': 1.0,                     #    Hopping parameter
        'ham_U': 4.0,                     #    Hubbard interaction
        'ham_Tperp': 0.0,                 #    For bilayer systems
        'beta': 1.0,                      #    Inverse temperature
        'Ltau': 0,                        #    '1' for time-displaced Green functions; '0' otherwise
        'NSweep': 400,                    #    Number of sweeps
        'NBin': 10,                       #    Number of bins
        'Dtau': 0.05,                     #    Only dtau varies between simulations, Ltrot=beta/Dtau
        'Mz': True,                       #    If true, sets the M_z-Hubbard model: Nf=2, N_sum=1,
        },                                #            HS field couples to z-component of magnetization
        alf_dir='~/Programs/ALF',         # Local ALF copy, if present
    )
    sims.append(sim)
```

```
Theta values used:
5
10
20
```

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

[3]:
```
sims[0].compile()                         # Compilation needs to be performed only once
```

```
Compiling ALF... Done.
```

4. Perform the simulations, as specified in each element of `sim`:

[4]:
```
for i, sim in enumerate(sims):
    sim.run()                             # Perform the actual simulation in ALF
```

```
Prepare directory "/home/stafusa/ALF/pyALF/Hubbard_N_leg_ladder_L1=4_L2=1_Checke
rboard=False_Symm=True_Projector=True_Theta=5_T=1.0_U=4.0_Tperp=0.0_beta=1.0_Dta
u=0.05_Mz=True" for Monte Carlo run.
Create new directory.
Run /home/stafusa/Programs/ALF/Prog/Hubbard.out
Prepare directory "/home/stafusa/ALF/pyALF/Hubbard_N_leg_ladder_L1=4_L2=1_Checke
rboard=False_Symm=True_Projector=True_Theta=10_T=1.0_U=4.0_Tperp=0.0_beta=1.0_Dt
au=0.05_Mz=True" for Monte Carlo run.
Create new directory.
Run /home/stafusa/Programs/ALF/Prog/Hubbard.out
Prepare directory "/home/stafusa/ALF/pyALF/Hubbard_N_leg_ladder_L1=4_L2=1_Checke
rboard=False_Symm=True_Projector=True_Theta=20_T=1.0_U=4.0_Tperp=0.0_beta=1.0_Dt
au=0.05_Mz=True" for Monte Carlo run.
```

```
Create new directory.
Run /home/stafusa/Programs/ALF/Prog/Hubbard.out
```

5. Calculate the internal energies:

[6]:
```
%%capture
ener = np.empty((len(sims), 2))          # Matrix for storing energy values
thetas = np.empty((len(sims),))          # Matrix for Thetas values, for plotting
for i, sim in enumerate(sims):
    print(sim.sim_dir)                   # Directory containing the simulation output
    sim.analysis()                       # Perform default analysis
    thetas[i] = sim.sim_dict['Theta']                          # Store Theta value
    ener[i] = sim.get_obs(['Ener_scalJ'])['Ener_scalJ']['obs']  # Store internal energy
```

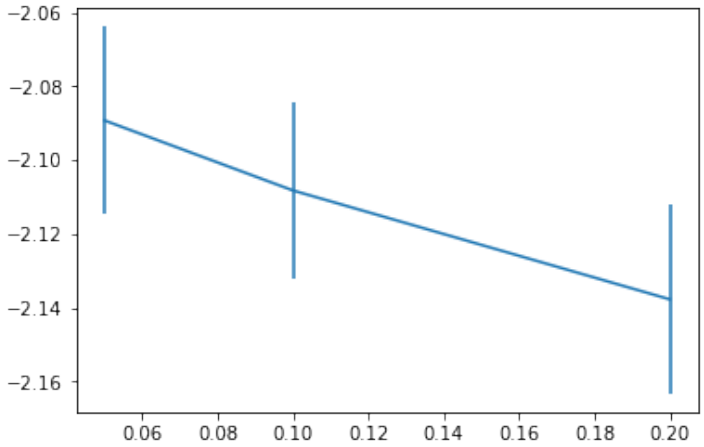
Where the *cell magic* `%%capture` suppresses the output of `sim.analysis()`, which lists the data directories and observables.

[7]:
```
print('For Theta values', thetas, 'the measured energies are:\n', ener)
```

```
For Theta values [ 5. 10. 20.] the measured energies are:
 [[-2.137697  0.025552]
 [-2.108235  0.023964]
 [-2.089294  0.025426]]
```

[8]:
```
plt.errorbar(1/thetas, ener[:, 0], ener[:, 1])
```

[8]:
```
<ErrorbarContainer object of 3 artists>
```



4.1. Exercises

1. [**TO BE FLESHED OUT**] Kondo phase transition. References:
   https://journals.aps.org/prb/abstract/10.1103/PhysRevB.63.155114
   https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.83.796

Part II. Getting your hands dirty - writing new code

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 must grapple with the package's Fortran source code. This second part of the tutorial consists in a set of guided exercises that exemplify how to make basic additions to the code, taking as starting point the template-like, relatively self-contained module `Hamiltonian_Hubbard_Plain_Vanilla_mod.F90`, which is also a good display of ALF's internal workings.

These worked-out exercises, together with ALF's modularity provided by its Predefined Structures should make getting your hands dirty less daunting than it may sound.

Downloading the code and tutorial

One can use the ALF package downloaded automatically by the Python script in the first part of this tutorial, or manually, by typing

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

in a shell. The necessary environment variables and the directives for compiling the code are set by the script `configure.sh`: `source configure.sh GNU`, followed by the command `make`. Details and further options are described in the package's documentation found in its repository.

Similarly, to download the tutorial, including solutions, enter:

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

Exercise 1 – Dimensional crossover

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 we start from the module `Hamiltonian_Hubbard_Plain_Vanilla_mod.F90` found in `<ALF_DIR>/Prog/Hamiltonians/`, which we will simply call "`Vanilla`", and then:

- Add `Ham_Ty` to the `VAR_Hubbard_Plain_Vanilla` name space in the parameter file `parameters`.
- Declare a new variable, `Ham_Ty`, in the module's specification (just search for the declaration of `Ham_T` in `Vanilla`).
- Add `Ham_Ty` to the `VAR_Hubbard_Plain_Vanilla` name space (`namelist`) declaration at the `Ham_Set` subroutine of `Vanilla`.
- Modify the hopping matrix in the subroutine `Ham_Hop` in `Vanilla`:

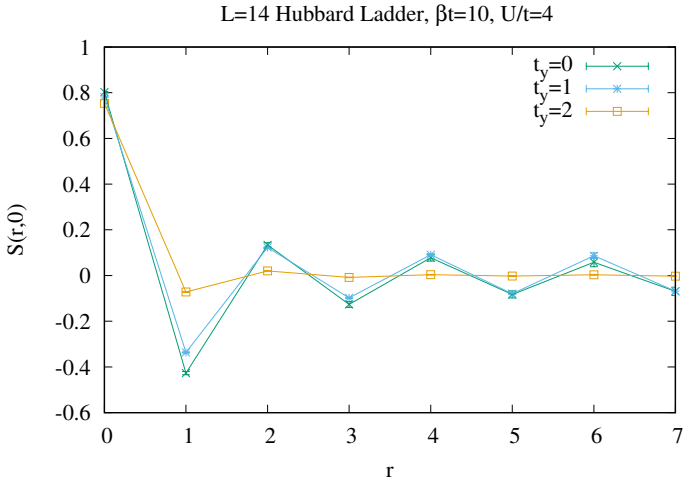
```
 Do I = 1,Latt%N
   Ix = Latt%nnlist(I,1,0)
   Op_T(1,nf)%O(I,  Ix) = cmplx(-Ham_T,    0.d0, kind(0.D0))
   Op_T(1,nf)%O(Ix, I ) = cmplx(-Ham_T,    0.d0, kind(0.D0))
   If ( L2 > 1 ) then
      Iy = Latt%nnlist(I,0,1)
!!!!!!! Modifications for Exercise 1a
      !Op_T(1,nf)%O(I,  Iy) = cmplx(-Ham_T,    0.d0, kind(0.D0))
      !Op_T(1,nf)%O(Iy, I ) = cmplx(-Ham_T,    0.d0, kind(0.D0))
      Op_T(1,nf)%O(I,  Iy) = cmplx(-Ham_Ty,    0.d0, kind(0.D0))
      Op_T(1,nf)%O(Iy, I ) = cmplx(-Ham_Ty,    0.d0, kind(0.D0))
!!!!!!!
   endif
```

```
   Op_T(1,nf)%O(I,  I ) = cmplx(-Ham_chem, 0.d0, kind(0.D0))
   Op_T(1,nf)%P(i) = i
Enddo
```

Note: If you'd like to run the simulation using MPI, you should also add the broadcasting call for `Ham_Ty` to `Ham_Set`. It is a good idea as well to include the new variable to the simulation parameters written into the file `info`, also in `Ham_Set`.

In the directory `Solutions/Exercise_1` we have duplicated ALF's code and commented the changes that have to be carried out to the file `Hamiltonian_Hubbard_Plain_Vanilla_mod.F90` in the `Prog` directory. The solution directory also includes reference data and the necessary `Start` directory (remember to copy its contents to every new `Run` directory, and to have a different `Run` directory for each simulationi).

As an application of this code, we can consider a 2-leg ladder system defined, e.g., with `L1=14`, `L2=2`, `Lattice_type="Square"`, `Model="Hubbard_Plain_Vanilla"` and different values of `Ham_Ty`. The results you should obtain are summarized in Fig. 1.



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

The SU(2) Hubbard-Stratonovich decomposition conserves spin rotational symmetry. Introduce into `Hamiltonian_Hubbard_mod.F90` the same changes done to the `Vanilla` module, described in the previous item, and compare results.

Exercise 2 – Adding a new observable

[Stub]

Here the task if to define a new observable, the kinetic energy correlation, given by

$$\left\langle \hat{O}_{i,\delta}\hat{O}_{j,\delta'}\right\rangle - \left\langle \hat{O}_{i,\delta}\right\rangle \left\langle \hat{O}_{j,\delta'}\right\rangle = S_O\left(i-j,\delta,\delta'\right) \tag{1}$$

where

$$\hat{O}_{i,x} = \sum_{\sigma}\left(\hat{c}^{\dagger}_{i,\sigma}\hat{c}_{i+ax,\sigma} + H.c.\right). \tag{2}$$

[...]

In the 1-D Hubbard we have emergent $SO(4)$ symmetry:

$$\left\langle \bar{S}(r)S(0)\right\rangle \sim \frac{(-1)^r}{r}\ln^d(r) \tag{3}$$

$$\left\langle \hat{O}_{r,x}\hat{O}_{0,x}\right\rangle - \left\langle \hat{O}_{r,x}\right\rangle \left\langle \hat{O}_{0,x}\right\rangle \sim \frac{(-1)^r}{r}\ln^\beta(r) \tag{4}$$

where d =?? and β =?? [**?**].

[It should be added to the Predefined Structures.]

Exercise 3 – Defining a new model: The one-dimensional t-V model

3a) 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.

$$\hat{H} = -t\sum_i \left(\hat{c}_i^\dagger \hat{c}_{i+a} + \hat{c}_{i+a}^\dagger \hat{c}_i\right) - \frac{V}{2}\sum_i \left(\hat{c}_i^\dagger \hat{c}_{i+a} + \hat{c}_{i+a}^\dagger \hat{c}_i\right)^2 \tag{5}$$

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:

$$\hat{H} = -t\sum_i \left(\hat{c}_i^\dagger \hat{c}_{i+a} + \hat{c}_{i+a}^\dagger \hat{c}_i\right) + V\sum_i \left(\hat{n}_i - 1/2\right)\left(\hat{n}_{i+a} - 1/2\right) \tag{6}$$

[Mention tV Jupyter notebook.]

[To be updated:]

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:

$$e^{\Delta\tau\frac{V}{2}\left(\hat{c}_i^\dagger \hat{c}_{i+a} + \hat{c}_{i+a}^\dagger \hat{c}_i\right)^2} = \sum_{l=\pm1,\pm2} \gamma_l e^{\sqrt{\Delta\tau\frac{V}{2}}\eta_l\left(\hat{c}_i^\dagger \hat{c}_{i+a} + \hat{c}_{i+a}^\dagger \hat{c}_i\right)} = \sum_{l=\pm1,\pm2} \gamma_l e^{g\eta_l\left(\hat{c}_i^\dagger, \hat{c}_{i+a}^\dagger\right)O\left(\hat{c}_i, \hat{c}_{i+a}\right)^T} \tag{7}$$

  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:

$$\hat{H} = J_{xx} \sum_i \hat{S}_i^x \hat{S}_{i+a}^x + \hat{S}_i^y \hat{S}_{i+a}^y + J_{zz} \sum_i \hat{S}_i^z \hat{S}_{i+a}^z \tag{8}$$

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. 2 shows typical results.