Rotating Bosons Project

Casey E. Berger

April 7th, 2017 – February 22nd, 2018

Contents

1 Algorithm and Pseudocode for Aarts Paper

$\epsilon = 5 * 10^{-5}$
n (Langevin time steps) $= 5 * 10^{6}$
$m = \lambda = 1$
$N_x = N_\tau = 4, 6, 8, 10$
$0 \leq \mu \leq 1.7$
$\beta = N_\tau * \tau$
Steps:

1. Initialize our four fields ($\phi_1^R, \phi_1^I, \phi_2^R$, and ϕ_2^I at each site on an $N_x^3 N_\tau$ lattice

   Randomize the field values between -1 and 1 and then they should adjust accordingly.

2. Evolve the whole lattice (each site independently) for n steps of size ϵ in complex Langevin time

   - Generate real, Gaussian distributed noise.
   - Generate the action from the fields.
   - Use the Langevin evolution equations (discrete)
   - Take de-correlated samples (start by saving every 20 steps but we can play around with these details later)

3. Repeat this over all the different sizes and values for the chemical potentials

2 Code for Aarts Paper

2.1 *v4.2.cpp*

Version 4.2 starts by setting number of parallel threads to 16 and declares the following variables:

- strings str and mu_string

  str is for reading in the inputs and searching for parameters, mu_string is for the line that contains the list of mu values.

- ints dim, Nx, Nt, size, nL, and nTh

  dim is the spatiotemporal dimension, Nx and Nt are number of steps in (each) space and time direction, size is the spatial volume of the lattice ($Nx^{\text{dim}-1}$), nL is number of steps in Complex Langevin time, and nTh is number of thermalization steps.

  Note that at the moment, we are actually doing the thermalization in analysis (just ignoring the first 50k data points), but it was originally planned to happen in this code.

- doubles l, m, and eps

  l is the interaction strength (lambda), m is the mass of the bosons, and eps is the time step size in Langevin time.

- vector of doubles mu_vals

  This will eventually contain the list of values given for the chemical potential, extracted from mu_string.

Next, the code checks for input files given in the command. If it finds none, it exits with an error.

If an input file is given, it checks to see if the input file can be opened. If it cannot, it exits with an error.

If it can open the input file, it opens it and initializes an integer count and an array of nine strings, corresponding to the nine input values:

```
string inputs [9] = {"dim", "lambda","m","Nx", "Nt", "nL", "nTh",
    "eps","mu"};
```

Using a while loop (while "count" is less than 9, the number of inputs we want), the code searches through the text of the input file. When it finds the first entry in our array of inputs, it replaces the name of the parameter with its value in the input file and increments the count, then repeats the search with the next entry in the array, until it has found the last one.

The code next converts these 9 strings into their appropriate types and initializes the pre-defined variables with those values:

```
dim = stoi(inputs[0]);
l = stod(inputs[1]);
m = stod(inputs[2]);
```

```
Nx = stoi(inputs[3]);
size = pow(Nx,(dim-1));
Nt = stoi(inputs[4]);
nL = stoi(inputs[5]);
nTh = stoi(inputs[6]);
eps = stod(inputs[7]);
mu_string = inputs[8];
mu_vals = mu_list(mu_string);
```

Here, mu_list is a function that takes the string of mu values and returns a vector of those values as doubles.

Now, all the parameters we need have been initialized, and we begin our parallel loop over our values of the chemical potential. We declare and initialize n_mu, the total number of mu values we have. We then loop over i from i=0 to i¡n_mu.

**Note: it is important to not include the value n_mu - that will cause the loop to re-do the first mu in our vector. This caused a lot of problems in the first round of data taking.

For each i, we declare a double - mu - and assign it the value of the ith element of our mu vector. The code then generates a logfile name with the format ”logfile_mu_muvalue_N_Nxvalue.log” and the ”append” read/write option. It opens the logfile, prints the spatial dimensions (dim-1), the number of spatial sites and number of temporal sites, and ”starting clock”.

An initial time for the calculation for this value of mu is then taken, stored as t0.

The header column is written to the logfile, and then the file is closed:

```
mufile << setw(5) << left << "#mu ";
mufile << setw(5) << left << "step ";
mufile << setw(15) << left << "Re[phi_1] ";
mufile << setw(15) << left << "Im[phi_1] ";
mufile << setw(15) << left << "Re[phi_2] ";
mufile << setw(15) << left << "Im[phi_2] ";
mufile << setw(20) << left << "Re[phi^{*}phi] ";
mufile << setw(20) << left << "Im[phi^{*}phi] ";
mufile << setw(20) << left << "Re[S] ";
mufile << setw(20) << left << "Im[S] ";
mufile << setw(20) << left << "Re[<n>] ";
mufile << setw(20) << left << "Im[<n>] ";
mufile << setw(12) << left << "dt (sec) " << endl;
mufile.close();
```

The code then dynamically allocates memory for the Lattice - 8 doubles for the field at each spatiotemporal lattice site, for a total number of $N_x^{dim-1} * N_t * 8$ double precision entries. At each site, 8 field values will be stored - one for each of the four fields at Langevin time step n, and one for each of the four fields at Langevin time step $n+1$.

Next, the fields are initialized everywhere on the lattice, using the function `lattice_init`, which is defined in the file ”lattice_init.cpp,” described here in section 2.2.

The code then loops over our nL steps in Complex Langevin time, performing the Langevin evolution at each step, k, using the function `Langevin_evolution`, which is defined in the file "Langevin_evolution.cpp" and described here in section 2.4. The time interval for each step is also computed here, to help determine if there are hang ups or timing issues that will need to be resolved. If we have gone enough steps in Langevin time to thermalize (k $\geq$ nTh), we compute the observables at that step in Complex Langevin time, using the function `compute_observables`. This function is defined in the file "Observables.cpp" and described here in section 2.5.

The option exists here, if it is not commented out, to save lattice configurations every 10000 steps in Langevin time. This is done with the function `lattice_save`, which is defined in the file "lattice_save.cpp" and described in section 2.3.

Once we have completed nL steps in Langevin time, the code deallocates the memory for the Lattice, using the `delete []` function. Then the final time is taken and the total time for the evolution is computed. The logfile is opened one final time to print the total running time for that value of mu and N, and then it is closed and the main function ends.

2.2 *lattice_init.cpp*

This file defines the function `lattice_init`, which takes the $N_x^{\text{dim}-1} \times N_t$ Lattice and generates 4 field variables at each site using a random number generator with a uniform distribution between -1 and 1.

```
void lattice_init(double *** Lattice, int size, int time_size){
   //generate 4 field variables using a random number generator
   //uniformly distributed between -1 and 1
   //assign these values to the lattice at each site
   std::random_device rd;
   std::mt19937 mt(rd());
   //std::default_random_engine generator;
   std::uniform_real_distribution<double> distribution(-1.0,1.0);
   //assign each field a random value, uniformly distributed between -1
       and 1
   for (int i = 0; i<size;i++){
      for (int j = 0; j<time_size;j++){
         for (int k = 0; k<4;k++){
            double r = distribution(mt);
            Lattice[i][j][k] = r;
            Lattice[i][j][k+4] = r;
         }
      }
   }
}
```

This means that at every spatiotemporal site on the lattice, there will be 8 fields. The fields are initialized so that

$$\begin{aligned}
Lattice[x][t][0] = \quad & Lattice[x][t][4] = & \phi_1^R(t_L = 0) \\
Lattice[x][t][1] = \quad & Lattice[x][t][5] = & \phi_1^I(t_L = 0) \\
Lattice[x][t][2] = \quad & Lattice[x][t][6] = & \phi_2^R(t_L = 0) \\
Lattice[x][t][3] = \quad & Lattice[x][t][7] = & \phi_2^I(t_L = 0).
\end{aligned}$$

In the future, the first set of fields stored in the lattice ($Lattice[x][t][0-3]$) will hold the Langevin time-evolved fields, while the second set will hold the value of the fields at the previous Langevin time step. This will be important in the Langevin evolution, since the entire lattice must be evolved simultaneously.

2.3 *lattice_save.cpp*

This file defines the function `lattice_save`, which saves the lattice configuration at a given time step to a text file.

The code first generates a filename for the saved field, with the format ″v4_mu_muvalue_N_Nxvalue_field_config.log″ and the ″append″ read/write option. It then opens the file (exiting with an error if the file cannot be opened) and writes the header:

```
fout << std::setw(10)<<"coords"; //x,y,z,t for 3d
fout << std::setw(12) << "phi_1^{R}" << std::setw(12) << "phi_1^{I}";
fout << std::setw(12) << "phi_2^{R}" << std::setw(12) << "phi_2^{I}" <<
    std::endl;
```

The function then loops over every spatiotemporal site and prints the coordinates and the four field values at that point in spacetime (fields 0-3, the evolved fields).

The function then closes the field configuration file.

Issues

***One problem with this: it does not save the information about what Langevin time step we are at! If we decide to use this later, we will need to add a column in the save file for the Langevin timestep or create individual files for each timestep.

2.4 *Langevin_evolution.cpp*

This file defines many functions which all contribute to its main objective: evolve the lattice in Langevin time.

First, description of the functions called by Langevin_evolution:

- Return field $\phi_a(\vec{x}, t)$

  ```
  std::vector<double> phi_a(double *** Lattice, int i, int t, int a, bool
  new_lat)
  ```

This function initializes a vector, which will return the real and imaginary part of a field, determined by $a = 1, 2$ and whether or not it wants the new fields (the first 4 entries) or the old fields (the last 4 entries).

- Positive step function

  `std::vector<int> positive_step(int dim, int dir, int i, int t, int Nx, int Nt)`

  This function determines the coordinates (i', t') that point to the site on the lattice that is one step forward in the specified direction from given coordinates (i, t) with periodic boundary conditions. It returns this information as a vector where the first element is i' and the second is t'.

- Negative step function

  `std::vector<int> negative_step(int dim, int dir, int i, int t, int Nx, int Nt)`

  This function determines the coordinates (i', t') that point to the site on the lattice that is one step backward in the specified direction from given coordinates (i, t) with periodic boundary conditions. It returns this information as a vector where the first element is i' and the second is t'.

- Anti-symmetric tensor

  ```
  double epsilon(int a, int b){
     if (a == b){return 0;}
     else if (a == 1 and b ==2){return 1.;}
     else if (a ==2 and b == 1){return -1.;}
     else {return 0.;}
  }
  ```

  This returns 0 if $\epsilon_{ab} = \epsilon_{11}$ or ϵ_{22}, $+1$ if $\epsilon_{ab} = \epsilon_{12}$, and -1 if $\epsilon_{ab} = \epsilon_{21}$.

- Real drift function

  `double K_a_Re(double m, double l, int a, double *** Lattice, int dim, int Nx, int Nt, int i, int t, double mu)`

  This function computes the real drift function at a point on the lattice, using the old field values. This is because the drift function is meant to evaluate the field at some time t_n in Langevin time and then update each site simultaneously to the new values at time t_{n+1} in Langevin time.

  The value of the drift function, K, is initialized to zero, and the new lattice flag is set to false. The field ϕ_a is retrieved using the function described above.

  Fixed a sign issue in here in the term multiplying sinh (the signs of the fields were flipped)

- Imaginary drift function

`double K_a_Im(double m, double l, int a, double *** Lattice, int dim, int Nx, int Nt, int i, int t, double mu)`

This function computes the imaginary drift function at a point on the lattice, using the old field values, just as in the real drift function.

Fixed a sign issue in here in the term multiplying sinh (the signs of the fields were flipped) AND fixed that the fields were imaginary here when they should have been real.

Now, the main function defined in this file: the Langevin evolution. `void Langevin_evolution(double m, double l, double *** Lattice, int size, int dim, int Nx, int Nt, double mu, double eps)`

The first thing this function does is define the Gaussian noise for the evolution of the real parts of the fields. ϕ_1 and ϕ_2 both get a random distribution defined in a Gaussian distribution with mean = 0 and standard deviation $= \sqrt{2}$ to be the real noise.

Then, the code walks through the entire spatiotemporal lattice and updates the fields:

```
double eta_1 = noise1(mt);
double eta_2 = noise2(mt);
Lattice[i][t][0] =
    Lattice[i][t][4]+eps*K_a_Re(m,l,1,Lattice,dim,Nx,Nt,i,t,mu) +
    sqrt(eps)*eta_1;
Lattice[i][t][1] =
    Lattice[i][t][5]+eps*K_a_Im(m,l,1,Lattice,dim,Nx,Nt,i,t,mu);
Lattice[i][t][2] =
    Lattice[i][t][6]+eps*K_a_Re(m,l,2,Lattice,dim,Nx,Nt,i,t,mu) +
    sqrt(eps)*eta_2;
Lattice[i][t][3] =
    Lattice[i][t][7]+eps*K_a_Im(m,l,2,Lattice,dim,Nx,Nt,i,t,mu);
```

This enters into the new value for the four fields the old field plus the drift function and noise (multiplied by the appropriate power of the Langevin step size).

Once the evolution is done everywhere, the old fields (the second 4) are assigned the values of the new fields (the first 4), since the evolution is complete and we are now at the new Langevin time.

Issues

positive_step and negative_step have been tested in 3 dimensions for lattices of Nx = 4 and 6 and work correctly. More testing should be done in lower dimension if there is any question about whether the lattice is evolving properly. There is a code written to check this in 3d and can be updated for smaller dimensions.

In the drift functions (see handwritten notes, page 3), do the fields marked by b include the cases where $b = a$?

There may be a factor of 2 on the parts multiplying the lambda terms in the drift functions. Check this... I got my lambdas multiplied by 0.5, but Aarts didn't. It's possible that it's a typo in his equations, or that I did something wrong. But looking at the equation for the action, the m^2 term should have twice the weight as the λ term since the λ multiplies the fields to the 4th power in the action and our first complexification introduces a factor of $\frac{1}{\sqrt{2}}$ for each field.

2.5 *Observables.cpp*

This file defines a few mathematical functions and then the functions that determine the observables. Then the compute_observables function computes all the observables and saves them to the logfile for the appropriate mu and N.

First, description of the functions called by compute_observables:

- Dirac delta functions

  ```
      double delta(int a, int b){
      if (a == b){return 1.;}
      else {return 0.;}
      }
  ```

  This simply returns 0 if $a \neq b$ and 1 if $a = b$.

  Fixed this in v4.2! It was returning the opposite!

- Anti-symmetric tensor

  ```
    double epsilon_ab(int a, int b){
      if (a == b){return 0;}
      else if (a == 1 and b ==2){return 1.;}
      else if (a ==2 and b == 1){return -1.;}
      else {return 0.;}
  ```

  This is exactly the same as the function defined in the Langevin_evolution file.

- The Action

  `std::vector<double> Action_S(double m, double l, double *** Lattice, int size, int dim, int Nx, int Nt, double mu)`

  This uses the new lattice values (although it should not matter at this point - they should be the same) to compute the action.

- The Density

  `std::vector<double> Density(double *** Lattice, int size, int dim, int Nx, int Nt, double mu)`

  This function declares a vector of doubles, which will contain the real and imaginary part of the density. It defines the lattice to be used as the new one (although, again, it should not matter). And it initializes the volume to be the spacetime volume of $Nx^{\dim-1} * Nt$.

It then does a walk over the entire spatiotemporal lattice and gets the fields ϕ_1 and ϕ_2 at each point, plus the field at one step forward in the time direction. It then adds the fields as given in the equation for the density and normalizes the real and imaginary parts at the end with the spacetime volume.

The function returns the density as a vector, d, with the first element being the real part and the second element being the imaginary part.

- The Field Modulus Squared

  ```
  std::vector<double> Field_Modulus_Squared(double *** Lattice, int size,
  int Nx, int Nt)
  ```

  This function declares a vector of doubles, which will contain the real and imaginary part of the field modulus squared. It again uses the new values on the lattice and walks over the entire lattice, adding $\sum_a \frac{1}{2}(\phi_a^R \phi_a^R - \phi_a^I \phi_a^I)$ to the real part and $\sum_a (\phi_a^R \phi_a^I)$ to the imaginary part.

  This does NOT normalize the field modulus squared by the lattice volume, although this would be an easy fix. Right now, the analysis code or the plotting code does that.

  The field modulus squared is returned as a vector. The first element is the real part and the second element is the imaginary part.

The function compute_observables opens the logfile that was previously created with append read/write privileges. It then defines doubles for each of the four fields and sums the values at each point on the lattice to get the field value on the entire lattice. Next, it computes the density, action, and field modulus squared using the previously described functions.

Finally, this function outputs the values it has computed, plus the chemical potential, number of sites, and time elapsed to the logfile and closes the file.

Issues

Check the action. It may or may not be right, but you haven't actually needed it for anything, so you haven't checked it.

Maybe normalize the field modulus squared by the volume?