1 Preamble

This is a brief introduction to performing some numerical computations utilising the pieces of Julia code delivered with this guide. This guide is available as a PDF file and an interactive Jupyter notebook.

2 On the Julia programming language

This is not a comprehensive guide to the use of the Julia programming language. If the reader is interested in one, I recommend taking a look at the excellent workshop by Dr. Carsten Bauer (http://carstenbauer.eu/#workshops). Assuming the reader is familiar with Python, this guide should be legible and give a fairly good idea of how to write working Julia code. The most important difference to Python is that Julia automatically compiles functions the first time they are called in a piece of code, allowing performance similar to C++ or Fortran.

Once the reader has managed to write some working code, I'd recommend reading the "Gotchas" part (the first file of Day 2) of the above workshop in order to make their code run smoothly. Once the code runs smoothly, I'd recommend taking a look at the Day 3 of the workshop, wherein parallelisation is discussed.

2.1 Setting up Julia

For this section of the guide I'll assume the reader is using a Debian based Linux distribution (e.g. Ubuntu). If memory serves, installing Julia on Windows is trivial using the binaries.

2.1.1 Step 1: dependencies

These are quite likely already installed, but running the line

```
sudo apt install build-essential libatomic1 python gfortran perl wget m4 cmake
pkg-config curl
```

won't hurt.

2.1.2 Step 2: Installing Julia

Running the lines

```
git clone https://github.com/julialang/julia
```

```
cd julia
```

```
make
```

```
make install
```

will install the newest version of Julia. This will take quite some time.

2.1.3 Step 3: Installing Julia packages

Most of the pieces of code provided with this guide don't require additional packages. However, we'll be computing the ground states of some systems in the examples in this guide, and we'll be

using the KrylovKit package for that.

Julia comes with its own package handling tool, which can be accessed by running the REPL:

`julia`

and typing `]`. Then, KrylovKit can be installed with the command `add KrylovKit`.

Other packages that might be useful are DelimitedFiles, BSON, Combinatorics, Multisets, and Plots.

DelimitedFiles can be used to save data in text form for e.g. plotting using Python and Matplotlib.

BSON can be used to save data in the BSON format, unsurprisingly, which can be useful if the data is used again in Julia code.

Combinatorics and Multisets are necessary for running the Transform.jl code provided with this guide.

The Plots package can be used for plotting in Julia. It tends to be a bit slow, so I prefer using Matplotlib. I will, however, be using Plots at one point in this guide. If the reader wants that part of the notebook to work interactively, Plots needs to be installed(`]add Plots`).

2.1.4 Step 4: Installing Jupyter

This step is only necessary if the reader wants to use the interactive Jupyter notebook version of this guide, or is interested in the materials of the previously mentioned workshop.

On the Julia side we'll need to install the package IJulia (`]add IJulia`). The easiest way to install Jupyter without additional software is via pip:

`pip install notebook`,

after which Jupyter can be accessed by running

`jupyter notebook`.

3 Using the code

This guide comes with five pieces of Julia code: Basis.jl, Operators.jl, Time.jl, Transform.jl, and Density.jl. In the rest of the guide we'll discuss some of the contents of these codes, and provide some examples for performing simple numerical computations using them.

3.1 The basis

When doing numerical linear algebra, it is important to be sure that all vectors and matrices are represented in the same basis and in the same order. The Hilbert space of bosonic lattice model with L sites, and a fixed number of bosons N has the dimension

$$d = \binom{L+N-1}{N}.$$

Fortunately, there is a simple formula for assigning unique indices running from 1 to d for a set of basis vectors, as detailed in A. I. Streltsov et al, Phys. Rev. A 81, 022124 (2010). There is also

a simple algorithm for iterating through the basis in that order, but I've unfortunately lost the reference for that.

Let us, for example, take a small system with $L = 4, N = 2$, and look at its basis. We'll start by computing the Hilbert space `dimension`:

[1]:
```
include("Basis.jl"); #this imports the functions from Basis.jl
L = 4
N = 2
d = dimension(L, N)
```

[1]:
```
10
```

In our indexing system, the first basis vector is the one with all bosons on the first site. In this example $|2000\rangle$. We can find the index of a specific basis vector using the function `find_index`:

[2]:
```
find_index([2, 0, 0, 0])
```

[2]:
```
1
```

[3]:
```
find_index([0, 1, 1, 0])
```

[3]:
```
6
```

The function `next!` changes a basis vector for the next one in order:

[4]:
```
basis_vector = [2, 0, 0, 0]
println(find_index(basis_vector), " ", basis_vector)
for i in 1:d
    next!(basis_vector)
    println(find_index(basis_vector), " ", basis_vector)
end
```

```
1 [2, 0, 0, 0]
2 [1, 1, 0, 0]
3 [1, 0, 1, 0]
4 [1, 0, 0, 1]
5 [0, 2, 0, 0]
6 [0, 1, 1, 0]
7 [0, 1, 0, 1]
8 [0, 0, 2, 0]
9 [0, 0, 1, 1]
10 [0, 0, 0, 2]
1 [2, 0, 0, 0]
```

Note that `next!` loops back to the first vector after it has reached the dth vector. Here I've used the Julia convention of noting an in-place function with an exclamation point.

[5]:
```
using LinearAlgebra #this line imports the built-in linear algebra package
vector = rand(2)
```

```
normalize!(vector)
println(vector, ", ", norm(vector), "\n")

vector = rand(2)
vector2 = normalize(vector)
println(vector, ", ", norm(vector))
println(vector2, ", ", norm(vector2))
```

```
[0.42101626913115997, 0.9070530861679921], 1.0

[0.3321690230825485, 0.2209388322397834], 0.3989363702234904
[0.8326366004093891, 0.5538197284845451], 1.0
```

3.2 Operators

Now that we have a basis to work with, we can construct some operators and state vectors to operate on. Let us, as an example, write a function that returns the operator $a^\dagger_{\ell+1}a_\ell$:

[6]:
```
using LinearAlgebra, SparseArrays #import some built-in packages
include("Basis.jl")

function example_hop(L, N, l)
    d = dimension(L, N)
    hop = spzeros(d, d) #create a d x d sparse matrix
    basis_vector = zeros(Int64, L) #create a L-component vector of integers;␣
 ↪default is Float64
    basis_vector[1] = N
    for i in 1:d
        if i != 1
            next!(basis_vector)
        end

        if basis_vector[l] > 0
            modified_vector = copy(basis_vector)
            modified_vector[l] -= 1
            modified_vector[l + 1] += 1
            index = find_index(modified_vector)
            hop[index, i] = sqrt(basis_vector[l] * modified_vector[l + 1])
        end
    end

    return hop
end
```

[6]:
```
example_hop (generic function with 1 method)
```

Now let's check that it does what we want it to do. We'll do this by initialising the vector $|0200\rangle$ in the full Hilbert space and operating on it with $O = a^\dagger_3 a_2$:

[7]:
```
L = 4; N = 2; d = dimension(L, N)
state_vector = zeros(d)
state_vector[find_index([0, 2, 0, 0])] = 1.
O = example_hop(L, N, 2)
print_state(L, N, state_vector) #this is a function from Basis.jl

state_vector .= O * state_vector
print_state(L, N, state_vector)

state_vector .= O * state_vector
print_state(L, N, state_vector)

state_vector .= O' * state_vector #here ' denotes the adjoint
print_state(L, N, state_vector)

state_vector .= example_hop(L, N, 1)' * state_vector
print_state(L, N, state_vector)
```

```
1.0 |0200⟩         1.0

1.4142 |0110⟩         2.0

2.0 |0020⟩         4.0

2.8284 |0110⟩         8.0

2.8284 |1010⟩         8.0
```

The file Operators.jl contains a selection of functions for creating operators. Below are some examples, starting with the number operators n_ℓ:

[8]:
```
include("Operators.jl")
L = 4; N = 2; d = dimension(L, N)
state_vector = zeros(d)
state_vector[find_index([0, 2, 0, 0])] = 1.

n2 = number(L, N, 2)
println(state_vector' * n2 * state_vector)

ns = numbers(L, N)
println(state_vector' * ns[2] * state_vector)
println(state_vector' * ns[1] * state_vector)
```

```
2.0
2.0
0.0
```

The terms of the one-dimensional Bose–Hubbard Hamiltonian

$$H = H_U + H_J,$$

where

$$H_U = -\frac{U}{2}\sum_\ell n_\ell(n_\ell - 1),$$

and

$$H_J = J\sum_\ell (a_\ell^\dagger a_{\ell+1} + \text{H.c.}),$$

can be constructed separately, with the functions:

[9]:
```
HU = interaction(L, N);
HJ = hopping(L, N, periodic = false);
```

Note that these don't include the parameters U and J. We get the total Hamiltonian with e.g. $U = 1, J = 0.5$ by taking the sum:

[10]:
```
J = 0.5
H = HU .+ J .* HJ #here the . denote componentwise operations
```

[10]:
```
10×10 SparseMatrixCSC{Float64, Int64} with 28 stored entries:
 -1.0       0.707107          ...
  0.707107           0.5
           0.5            0.5
                    0.5          0.5
           0.707107
                    0.5       ...  0.5    0.707107
                        0.5                    0.5
                                    -1.0       0.707107
                                0.5    0.707107            0.707107
                                                0.707107   -1.0
```

Both terms can be generated with a single function:

[11]:
```
HU, HJ = split_hamiltonian(L, N, periodic = false);
```

There is also a function for generating the Hamiltonian of a rectangular $L_1 \times L_2$ array:

[12]:
```
L1 = 3; L2 = 3;
HU, HJ = split_hamiltonian_2D(L1, L2, N, periodic = 0);
#periodic = 0 for open boundaries
#periodic = 1 for one periodic boundary on the L1 axis, I think
#periodic = 2 for both boundaries periodic
```

A random disorder term

$$H_D = \sum_\ell \omega_\ell n_\ell, \omega_\ell \in [-1, 1]$$

can be generated as follows:

[13]:
```
HD = disorder(L, N)
```

[13]:
```
10×10 SparseMatrixCSC{Float64, Int64} with 10 stored entries:
 0.158973                                  ...
          -0.571929
                    -0.504432
                              -0.243188

                                          ...

                                             -1.16784
                                                      -0.906594
                                                                -0.64535
```

A specific disorder pattern can also be used:

[14]:
```
pattern = ones(L)
for l in 2:L
    pattern[l] = -pattern[l - 1]
end

HD = disorder(L, N, dis = pattern)
```

[14]:
```
10×10 SparseMatrixCSC{Float64, Int64} with 10 stored entries:
 2.0
     0.0
         2.0
             0.0
                 -2.0
                      0.0
                          -2.0
                               2.0
                                   0.0
                                       -2.0
```

3.3 Eigensolvers

The built-in LinearAlgebra package contains eigensolvers for the full spectra of dense matrices:

[15]:
```
using LinearAlgebra, SparseArrays
include.(["Operators.jl", "Basis.jl"]) #adding the . to any function allows it␣
 ↪to be applied to an array of inputs
L = 4; N = 3;
HU, HJ = split_hamiltonian(L, N)
H = Matrix(HU .+ 0.05 .* HJ) #convert the Hamiltonian into a dense matrix
out = eigen(H)
println("ground state energy E = ", out.values[1], "\nground state:")
print_state(L, N, out.vectors[:, 1],  0.9999)
```

```
ground state energy E = -3.0077012489315673
ground state:
-0.7049 |0300⟩          0.4969
0.7049 |0030⟩          0.9938
0.0334 |3000⟩          0.9949
-0.0334 |0003⟩          0.996
0.0321 |0210⟩          0.9971
-0.0321 |0120⟩          0.9981
0.0306 |1200⟩          0.999
-0.0306 |0021⟩          1.0
```

If there is no need to compute the full eigendecomposition, we can use e.g. KrylovKit for specific eigenpairs:

[16]:
```
using LinearAlgebra, SparseArrays, KrylovKit
include.(["Operators.jl", "Basis.jl"])
L = 4; N = 3;
HU, HJ = split_hamiltonian(L, N)
H = HU .+ 0.05 .* HJ #keep the matrix sparse
vals, vecs, info = eigsolve(H, 1, :SR) # 1 to compute a single eigenpair, :SR␣
 ↪for smallest real eigenvalue,
println("ground state energy E = ", vals[1], "\nground state:")
print_state(L, N, vecs[1],  0.9999)
```

```
ground state energy E = -3.0077012489315713
ground state:
-0.7049 |0030⟩          0.4969
0.7049 |0300⟩          0.9938
0.0334 |0003⟩          0.9949
-0.0334 |3000⟩          0.996
0.0321 |0120⟩          0.9971
-0.0321 |0210⟩          0.9981
0.0306 |0021⟩          0.999
-0.0306 |1200⟩          1.0
```

3.4 Time evolution

The uncommented part of the Time.jl code includes some functions for performing time-evolution using a simple Krylov subspace method. Below is a brief demonstration on how it works.

We'll set $L = 5, N = 3, J = 2U = 1$ and start with the initial state $|30000\rangle$:

[17]:
```
include.(["Operators.jl", "Time.jl"])
L = 5; N = 3;
HU, HJ = split_hamiltonian(L, N)
H = 0.5 .* HU .+ HJ
```

```
basis_vector = zeros(Int64, L)
basis_vector[1] = N

state = zeros(dimension(L, N))
state[find_index(basis_vector)] = 1.;
```

We'll set the time-step to 0.1 and the Krylov subspace dimension to 10, and evolve for 100 time-steps. At every time-step, we'll save the density profile of the state using the `density_profile` function found in the Density.jl file.

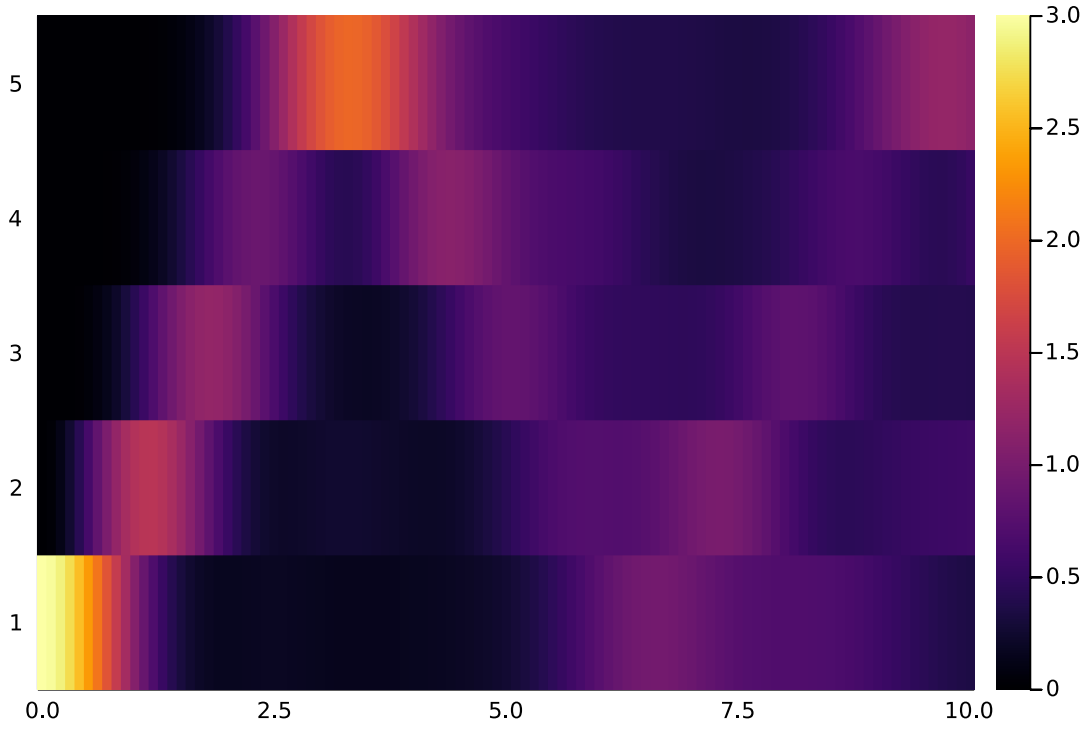
[18]:
```
include("Density.jl")
dt = 0.1; sdim = 10; steps = 100;
out = zeros(L, steps + 1)
out[:, 1] .= density_profile(L, N, state)
for i in 1:steps
    state = propagate(H, state, sdim, dt)
    out[:, i + 1] .= density_profile(L, N, state)
end
```

To see what happened to the density profile during that time, we'll plot the density profile as a function of time using the Plots package.

[19]:
```
using Plots
time = collect(range(0, 10, length = steps + 1))
sites = collect(range(1, L, length = L))
heatmap!(time, sites, out)
```

```
WARNING: using Plots.current in module Main conflicts with an existing
identifier.
```

[19]:




```
WARNING: using Plots.density in module Main conflicts with an existing
identifier.
```

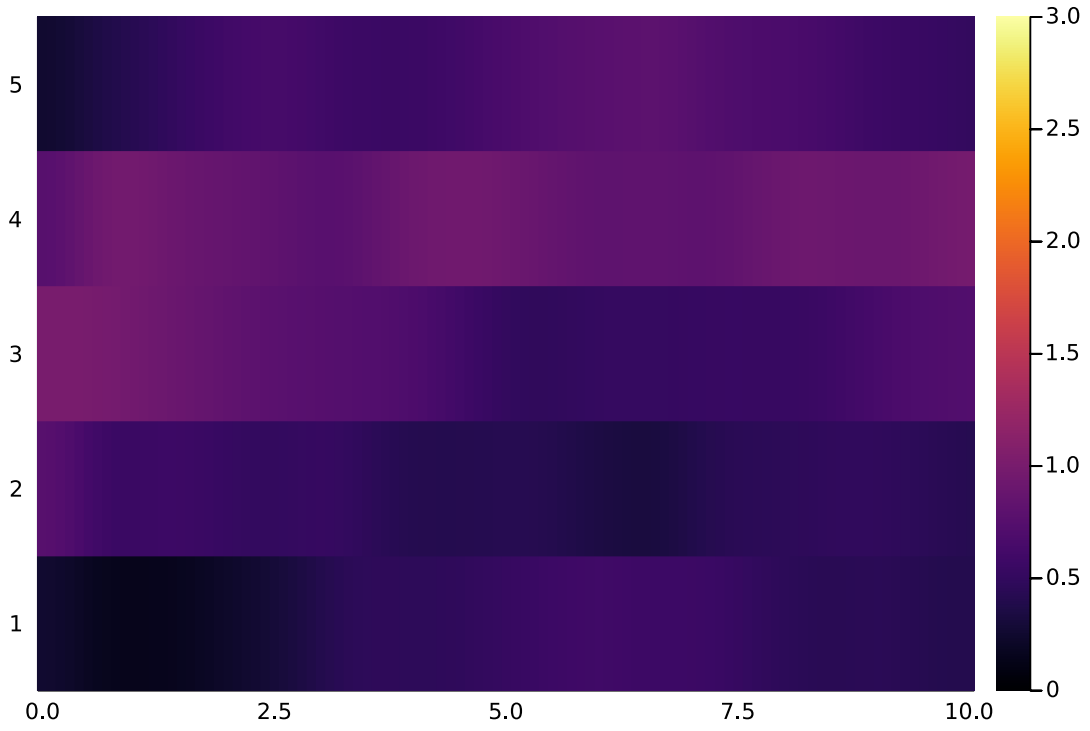
3.5 Reciprocal basis

The Transform.jl code contains functions for the transformations to the reciprocal spaces, i.e. the eigenbases of the hopping terms of the open and periodic Bose–Hubbard models in one dimension. Let's repeat the time-evolution bit from above, but plot the density profile in the reciprocal space this time.

[20]:
```
include.(["Operators.jl", "Time.jl", "Transform.jl", "Density.jl"])
using Plots
L = 5; N = 3;
HU, HJ = split_hamiltonian(L, N, periodic = false)
H = 0.5 .* HU .+ HJ

basis_vector = zeros(Int64, L)
basis_vector[1] = N

state = zeros(dimension(L, N))
state[find_index(basis_vector)] = 1.;

T = transform(L, N, periodic = false)
```

```
dt = 0.1; sdim = 10; steps = 100;
out = zeros(L, steps + 1)
out[:, 1] .= density_profile(L, N, T * state)
for i in 1:steps
    state = propagate(H, state, sdim, dt)
    out[:, i + 1] .= density_profile(L, N, T * state)
end

time = collect(range(0, 10, length = steps + 1))
sites = collect(range(1, L, length = L))
heatmap!(time, sites, out)
```

[20]:



Doesn't look very informative. Anyway, the way the function `transform` is written, it gets really slow with larger N. If there is use for that sort of transformations, I recommend saving the matrices in order to avoid having to compute them multiple times.