Trotter-Suzuki-MPI Python Documentation

Release 1.6.2

Peter Wittek, Luca Calderaro

Aug 07, 2017

Contents

CHAPTER 1

Introduction

The module is a massively parallel implementation of the Trotter-Suzuki approximation to simulate the evolution of quantum systems classically. It relies on interfacing with C++ code with OpenMP for multicore execution, and it can be accelerated by CUDA.

Key features of the Python interface:

- Simulation of 1D and 2D quantum systems.
- Cartesian and cylindrical coordinate systems.
- Fast execution by parallelization: OpenMP and CUDA are supported.
- Many-body simulations with non-interacting particles.
- Solving the Gross-Pitaevskii equation (e.g., dark solitons, vortex dynamics in Bose-Einstein Condensates).
- Imaginary time evolution to approximate the ground state.
- Stationary and time-dependent external potential.
- NumPy arrays are supported for efficient data exchange.
- Multi-platform: Linux, OS X, and Windows are supported.

Copyright and License

Acknowledgement

The original high-performance kernels were developed by Carlos Bederián. The distributed extension was carried out while Peter Wittek was visiting the Department of Computer Applications in Science & Engineering at the Barcelona Supercomputing Center, funded by the "Access to BSC Facilities" project of the HPC-Europe2 programme (contract no. 228398). Generalizing the capabilities of kernels was carried out by Luca Calderaro while visiting the Quantum Information Theory Group at ICFO-The Institute of Photonic Sciences, sponsored by the Erasmus+ programme. Further computational resources were granted by the Spanish Supercomputing Network (FY-2015-2-0023 and FI-2016-3-0042) and the Swedish National Infrastructure for Computing (SNIC 2015/1-162 and 2016/1-320), and a hardware grant by Nvidia. Pietro Massignan has contributed to the project with extensive testing and suggestions of new features.

Citations

1. Bederián, C. & Dente, A. (2011). Boosting quantum evolutions using Trotter-Suzuki algorithms on GPUs. *Proceedings of HPCLatAm-11, 4th High-Performance Computing Symposium*. PDF
2. Wittek, P. and Cucchietti, F.M. (2013). A Second-Order Distributed Trotter-Suzuki Solver with a Hybrid CPU-GPU Kernel. *Computer Physics Communications*, 184, pp. 1165-1171. PDF
3. Wittek, P. and Calderaro, L. (2015). Extended computational kernels in a massively parallel implementation of the Trotter-Suzuki approximation. *Computer Physics Communications*, 197, pp. 339-340. PDF

CHAPTER 2

Download and Installation

The package is available from the Python Package Index, containing the source code and examples, and also from the conda-forge channel. The documentation is hosted on Read the Docs.

The latest development version is available on GitHub. Further examples are available in Jupyter notebooks.

Dependencies

The module requires Numpy. The code is compatible with both Python 2 and 3.

If you want to use the GPU kernel, ensure that CUDA is installed and set the *CUDAHOME* environment variable before you start.

Installation via conda

The conda-forge channel provides packages for Linux and macOS that can be installed using conda:

```
$ conda install -c conda-forge cvxopt
```

Installation via pip

The code is available on PyPI, hence it can be installed by

```
$ sudo pip install trottersuzuki
```

Installation from source

If you want the latest git version, first clone the repository and generate the Python version. Note that it requires autotools and swig.

```
$ git clone https://github.com/trotter-suzuki-mpi/trotter-suzuki-mpi
$ cd trotter-suzuki-mpi
$ ./autogen.sh
$ ./configure --without-mpi --without-cuda
$ make python
```

Then follow the standard procedure for installing Python modules from the *src/Python* folder:

:: $ sudo python setup.py install

Build on Mac OS X

Before installing using pip or from source, gcc should be installed first. As of OS X 10.9, gcc is just symlink to clang. To build trottersuzuki and this extension correctly, it is recommended to install gcc using something like:

```
$ brew install gcc48
```

and set environment using:

```
export CC=/usr/local/bin/gcc
export CXX=/usr/local/bin/g++
export CPP=/usr/local/bin/cpp
export LD=/usr/local/bin/gcc
alias c++=/usr/local/bin/c++
alias g++=/usr/local/bin/g++
alias gcc=/usr/local/bin/gcc
alias cpp=/usr/local/bin/cpp
alias ld=/usr/local/bin/gcc
alias cc=/usr/local/bin/gcc
```

Then you can issue

```
$ sudo pip install trottersuzuki
```

CHAPTER 3

Quick Start Guide

Simulation Set up

Start by importing the module:

```
import trottersuzuki as ts
```

To set up the simulation of a quantum system, we need only a few lines of code. First of all, we create the lattice over which the physical system is defined. All information about the discretized space is collected in a single object. Say we want a squared lattice of 300x300 nodes, with a physical area of 20x20, then we have to specify these in the constructor of the `Lattice` class:

```
grid = ts.Lattice2D(300, 20.)
```

The object `grid` defines the geometry of the system and it will be used throughout the simulations. Note that the origin of the lattice is at its centre and the lattice points are scaled to the physical locations.

The physics of the problem is described by the Hamiltonian. A single object is going to store all the information regarding the Hamiltonian. The module is able to deal with two physical models: Gross-Pitaevskii equation of a single or two-component wave function, namely (in units $\hbar = 1$):

$$i\frac{\partial}{\partial t}\psi(t) = H\psi(t)$$

being, for Cartesian coordinates

$$H = \frac{1}{2m}(P_x^2 + P_y^2) + V(x,y) + g|\psi(x,y)|^2 + g_{LHY}|\psi(x,y)|^3 + \omega L_z$$

while for cylindrical coordinates

$$H = -\frac{1}{2m}(1/r\partial_r(r\partial_r - l^2/r^2) + \partial_z^2) + V(r,z) + g|\psi(r,z)|^2 + g_{LHY}|\psi(x,y)|^3$$

and $\psi(t) = \psi_t(x,y)$ for the single component wave function, or

$$H = \begin{bmatrix} H_1 & \frac{\Omega}{2} \\ \frac{\Omega}{2} & H_2 \end{bmatrix}$$

where

$$H_1 = \frac{1}{2m_1}(P_x^2 + P_y^2) + V_1(x,y) + g_1|\psi(x,y)_1|^2 + g_{12}|\psi(x,y)_2|^2 + \omega L_z$$
$$H_2 = \frac{1}{2m_2}(P_x^2 + P_y^2) + V_2(x,y) + g_2|\psi(x,y)_2|^2 + g_{12}|\psi(x,y)_1|^2 + \omega L_z$$

and $\psi(t) = \begin{bmatrix}\psi_1(t)\\ \psi_2(t)\end{bmatrix}$, for the two component wave function.

First we define the object for the external potential $V(x,y)$. A general external potential function can be defined by a Python function, for instance, the harmonic potential can be defined as follows:

```
def harmonic_potential(x,y):
    return 0.5 * (x**2 + y**2)
```

Now we create the external potential object using the `Potential` class and then we initialize it with the function above:

```
potential = ts.Potential(grid)  # Create the potential object
potential.init_potential(harmonic_potential)  # Initialize it using a python function
```

Note that the module provides a quick way to define the harmonic potential, as it is fequently used:

```
omegax = omegay = 1.
harmonicpotential = ts.HarmonicPotential(grid, omegax, omegay)
```

We are ready to create the `Hamiltonian` object. For the sake of simplicity, let us create the Hamiltonian of the harmonic oscillator:

```
particle_mass = 1. # Mass of the particle
hamiltonian = ts.Hamiltonian(grid, potential, particle_mass)  # Create the␣
↪Hamiltonian object
```

The quantum state is created by the `State` class; it resembles the way the potential is defined. Here we create the ground state of the harmonic oscillator:

```
import numpy as np  # Import the module numpy for the exponential and sqrt functions

def state_wave_function(x,y):  # Wave function
    return np.exp(-0.5*(x**2 + y**2)) / np.sqrt(np.pi)

state = ts.State(grid)  # Create the quantum state
state.init_state(state_wave_function)  # Initialize the state
```

The module provides several predefined quantum states as well. In this case, we could have used the `GaussianState` class:

```
omega = 1.
gaussianstate = ts.GaussianState(grid, omega)  # Create a quantum state whose wave␣
↪function is Gaussian-like
```

We are left with the creation of the last object: the `Solver` class gathers all the objects we defined so far and it is used to perform the evolution and analyze the expectation values:

```
delta_t = 1e-3  # Physical time of a single iteration
solver = ts.Solver(grid, state, hamiltonian, delta_t)  # Creating the solver object
```

If you want to solve the problem on the GPU, request the matching kernel:

```
delta_t = 1e-3  # Physical time of a single iteration
solver = ts.Solver(grid, state, hamiltonian, delta_t, kernel_type="gpu")
```

Note that not all functionality is available in the GPU kernel.

Finally we can perform both real-time and imaginary-time evolution using the method `evolve`:

```
iterations = 100  # Number of iterations to be performed
solver.evolve(iterations, True)  # Perform imaginary-time evolution
solver.evolve(iterations)  # Perform real-time evolution
```

Analysis

The classes we have seen so far implement several members useful to analyze the system (see the function reference section for a complete list).

Expectation values

The solver class provides members for the energy calculations. For instance, the total energy can be calculated using the `get_total_energy` member. We expect it to be 1 ($\hbar = 1$), and indeed we get the right result up to a small error which depends on the lattice approximation:

```
tot_energy = solver.get_total_energy()
print(tot_energy)
```

```
1.00146456951
```

The expected values of the X, Y, P_x, P_y operators are calculated using the members in the `State` class

```
mean_x = state.get_mean_x()  # Get the expected value of X operator
print(mean_x)
```

```
1.39431975344e-14
```

Norm of the state

The squared norm of the state can be calculated by means of both `State` and `Solver` classes

```
snorm = state.get_squared_norm()
print(snorm)
```

```
1.0
```

Particle density and Phase

Very often one is interested in the phase and particle density of the state. Two members of `State` class provide these features

```
density = state.get_particle_density()  # Return a numpy matrix of the particle␣
↪density
phase = state.get_phase()  # Return a numpy matrix of the phase
```

Imprinting

The member `imprint`, in the `State` class, applies the following transformation to the state:

$$\psi(x, y) \to \psi'(x, y) = f(x, y)\psi(x, y)$$

being $f(x, y)$ a general complex-valued function. This comes in handy when we want to imprint, for instance, vortices or solitons:

```
def vortex(x, y):  # Function defining a vortex
    z = x + 1j*y
    angle = np.angle(z)
    return np.exp(1j * angle)

state.imprint(vortex)  # Imprint the vortex on the state
```

File Input and Output

`write_to_files` and `loadtxt` members, in `State` class, provide a simple way to handle file I/O. The former writes the wave function arranged as a complex matrix, in a plain text; the latter loads the wave function from a file to the `state` object. The following code provides an example:

```
state.write_to_file("file_name")  # Write the wave function to a file
state2 = ts.State(grid)  # Create a new state
state2.loadtxt("file_name")  # Load the wave function from the file
```

For a complete list of methods see the function reference.

CHAPTER 4

Examples

Expectation values of the Hamiltonian and kinetic operators

The following code block gives a simple example of initializing a state and calculating the expectation values of the Hamiltonian and kinetic operators and the norm of the state after the evolution.

```
import numpy as np
from trottersuzuki import *

grid = Lattice2D(256, 15)  # create a 2D lattice

potential = HarmonicPotential(grid, 1, 1)  # define an symmetric harmonic potential␣
↪with unit frequecy
particle_mass = 1.
hamiltonian = Hamiltonian(grid, potential, particle_mass)      # define the␣
↪Hamiltonian:

frequency = 1
state = GaussianState(grid, frequency)  # define gaussian wave function state: we␣
↪choose the ground state of the Hamiltonian

time_of_single_iteration = 1.e-4
solver = Solver(grid, state, hamiltonian, time_of_single_iteration)  # define the␣
↪solver

# get some expected values from the initial state
print("norm: ", solver.get_squared_norm())
print("Total energy: ", solver.get_total_energy())
print("Kinetic energy: ", solver.get_kinetic_energy())

number_of_iterations = 1000
solver.evolve(number_of_iterations)  # evolve the state of 1000 iterations

# get some expected values from the evolved state
print("norm: ", solver.get_squared_norm())
```

```
print("Total energy: ", solver.get_total_energy())
print("Kinetic energy: ", solver.get_kinetic_energy())
```

Imaginary time evolution to approximate the ground-state energy

```
import numpy as np
from trottersuzuki import *

grid = Lattice2D(256, 15)  # create a 2D lattice

potential = HarmonicPotential(grid, 1, 1)  # define an symmetric harmonic potential␣
↪with unit frequecy
particle_mass = 1.
hamiltonian = Hamiltonian(grid, potential, particle_mass)  # define the Hamiltonian:

frequency = 3
state = GaussianState(grid, frequency)  # define gaussian wave function state: we␣
↪choose the ground state of the Hamiltonian

time_of_single_iteration = 1.e-4
solver = Solver(grid, state, hamiltonian, time_of_single_iteration)  # define the␣
↪solver

# get some expected values from the initial state
print("norm: ", solver.get_squared_norm())
print("Total energy: ", solver.get_total_energy())
print("Kinetic energy: ", solver.get_kinetic_energy())

number_of_iterations = 40000
imaginary_evolution = true
solver.evolve(number_of_iterations, imaginary_evolution)      # evolve the state of␣
↪40000 iterations

# get some expected values from the evolved state
print("norm: ", solver.get_squared_norm())
print("Total energy: ", solver.get_total_energy())
print("Kinetic energy: ", solver.get_kinetic_energy())
```

Imprinting of a vortex in a Bose-Einstein Condensate

```
import numpy as np
import trottersuzuki as ts

grid = ts.Lattice2D(256, 15)  # create a 2D lattice

potential = HarmonicPotential(grid, 1, 1)  # define an symmetric harmonic potential␣
↪with unit frequecy
particle_mass = 1.
coupling_intra_particle_interaction = 100.
hamiltonian = Hamiltonian(grid, potential, particle_mass, coupling_intra_particle_
↪interaction)  # define the Hamiltonian:
```

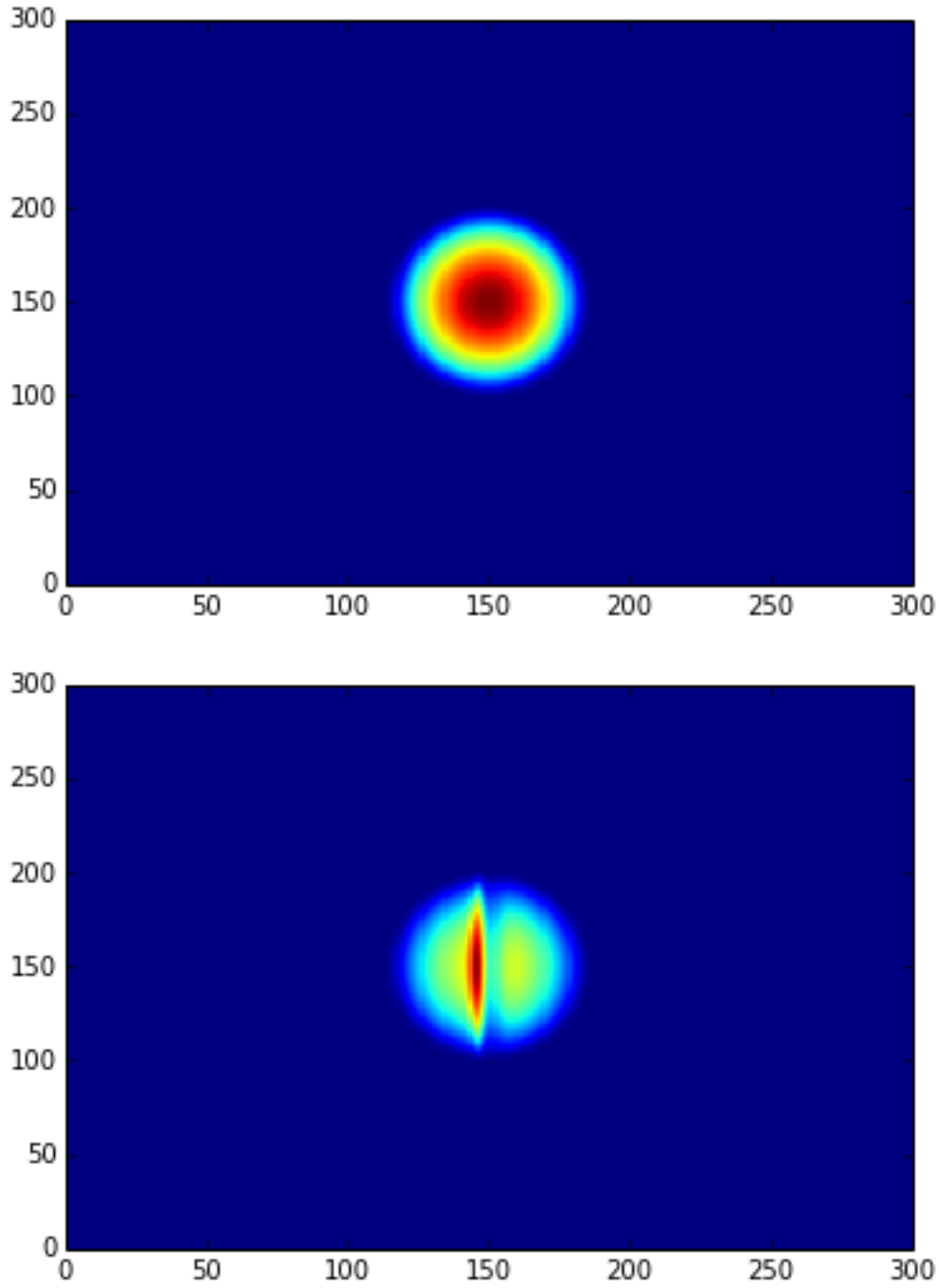
```
frequency = 1
state = GaussianState(grid, frequency)  # define gaussian wave function state: we␣
↪choose the ground state of the Hamiltonian
def vortex(x, y):  # vortex to be imprinted
    z = x + 1j*y
    angle = np.angle(z)
    return np.exp(1j * angle)

state.imprint(vortex)  # imprint the vortex on the condensate

time_of_single_iteration = 1.e-4
solver = Solver(grid, state, hamiltonian, time_of_single_iteration)  # define the␣
↪solver
```

Dark Soliton Generation in Bose-Einstein Condensate using Phase Imprinting

This example simulates the evolution of a dark soliton in a Bose-Einstein Condensate. For a more detailed description, refer to this notebook.

```
from __future__ import print_function
import numpy as np
import trottersuzuki as ts
from matplotlib import pyplot as plt

grid = ts.Lattice2D(300, 50.)  # # create a 2D lattice

potential = ts.HarmonicPotential(grid, 1., 1./np.sqrt(2.))  # create an harmonic␣
↪potential
coupling = 1.2097e3
hamiltonian = ts.Hamiltonian(grid, potential, 1., coupling)  # create the Hamiltonian

state = ts.GaussianState(grid, 0.05)  # create the initial state
solver = ts.Solver(grid, state, hamiltonian, 1.e-4)  # initialize the solver
solver.evolve(10000, True)  # evolve the state towards the ground state

density = state.get_particle_density()
plt.pcolor(density)  # plot the particle denisity
plt.show()

def dark_soliton(x,y):  # define phase imprinting that will create the dark soliton
    a = 1.98128
    theta = 1.5*np.pi
    return np.exp(1j* (theta * 0.5 * (1. + np.tanh(-a * x))))

state.imprint(dark_soliton)  # phase imprinting
solver.evolve(1000)  # perform a real time evolution

density = state.get_particle_density()
plt.pcolor(density)  # plot the particle denisity
plt.show()
```

The results are the following plots:



Imaginary time evolution in a 1D lattice using radial coordinate

```
import matplotlib.pyplot as plt
import numpy as np
import trottersuzuki as ts

angular_momentum = 1 # quantum number
radius = 100 # Physical radius
```

```
dim = 50 # Lattice points
time_step = 1e-1
fontsize = 16

# Set up the system
grid = ts.Lattice1D(dim, radius, False, "cylindrical")

def const_state(r):
    return 1./np.sqrt(radius)

state = ts.State(grid, angular_momentum)
state.init_state(const_state)

def pot_func(r,z):
    return 0.

potential = ts.Potential(grid)
potential.init_potential(pot_func)

hamiltonian = ts.Hamiltonian(grid, potential)
solver = ts.Solver(grid, state, hamiltonian, time_step)

# Evolve the system
solver.evolve(30000, True)

# Compare the calculated wave functions with respect to the groundstate function
psi = np.sqrt(state.get_particle_density()[0])
psi = psi / np.linalg.norm(psi)

groundstate = ts.BesselState(grid, angular_momentum)
groundstate_psi = np.sqrt(groundstate.get_particle_density()[0])
groundstate_psi = groundstate_psi / np.linalg.norm(groundstate_psi)

# Plot wave functions
plt.plot(grid.get_x_axis(), psi, 'o')
plt.plot(grid.get_x_axis(), groundstate_psi)
plt.grid()
plt.xlim(0,radius)
plt.xlabel('r', fontsize = fontsize)
plt.ylabel(r'$\psi$', fontsize = fontsize)
plt.show()
```

CHAPTER 5

Mathematical Details

What follows is a brief description of the approximation used to calculate the evolution of the wave function. Formulas of the evolution operator are provided.

Evolution operator

The evolution operator is calculated using the Trotter-Suzuki approximation. Given an Hamiltonian as a sum of hermitian operators, for instance $H = H_1 + H_2 + H_3$, the evolution is approximated as

$$e^{-i\Delta t H} = e^{-i\frac{\Delta t}{2}H_1} e^{-i\frac{\Delta t}{2}H_2} e^{-i\frac{\Delta t}{2}H_3} e^{-i\frac{\Delta t}{2}H_3} e^{-i\frac{\Delta t}{2}H_2} e^{-i\frac{\Delta t}{2}H_1}.$$

Since the wavefunction is discretized in the space coordinate representation, to avoid Fourier transformation, the derivatives are approximated using finite differences.

Kinetic operators

In cartesian coordinates, the kinetic term is $K = -\frac{1}{2m}\left(\partial_x^2 + \partial_y^2\right)$. The discrete form of the second derivative is

$$\partial_x^2 \psi(x) = \frac{\psi(x+\Delta x) - 2\psi(x) + \psi(x-\Delta x)}{\Delta x^2}$$

It is useful to express the above equation in a matrix form. The wave function can be vectorized as it is discrete, hence the partial derivative is the matrix

$$\frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{pmatrix} = \frac{1}{\Delta x^2}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \frac{1}{\Delta x^2}\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} - \frac{2}{\Delta x^2}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

On the right-hand side of the above equation the last matrix is the identity and in the approximation of Trotter-Suzuki it gives only a global shift of the wave function, which can be ignored. The other two matrices are in the diagonal block form and can be easily exponentiated. Indeed, for the real time evolution:

$$\exp\left[i\frac{\Delta t}{4m\Delta x^2}\begin{pmatrix}0&1\\1&0\end{pmatrix}\right]=\begin{pmatrix}\cos\beta&i\sin\beta\\i\sin\beta&\cos\beta\end{pmatrix}.$$

While, for imaginary time evolution:

$$\exp\left[\frac{\Delta t}{4m\Delta x^2}\begin{pmatrix}0&1\\1&0\end{pmatrix}\right]=\begin{pmatrix}\cosh\beta&\sinh\beta\\\sinh\beta&\cosh\beta\end{pmatrix},$$

with $\beta=\frac{\Delta t}{4m\Delta x^2}$.

In cylindrical coordinates, the kinetic operator has an additional term, $K=-\frac{1}{2m}\left(\partial_r^2+\frac{1}{r}\partial_r+\partial_z^2\right)$. The first derivative is discretized as

$$\frac{1}{r}\partial_r\psi(r)=\frac{\psi(r+\Delta r)-\psi(r-\Delta r)}{2r\Delta r},$$

and in matrix form

$$\frac{1}{2\Delta r}\begin{pmatrix}0&\frac{1}{r_0}&0\\-\frac{1}{r_1}&0&\frac{1}{r_1}\\0&-\frac{1}{r_2}&0\end{pmatrix}=\frac{1}{2\Delta r}\begin{pmatrix}0&\frac{1}{r_0}&0\\-\frac{1}{r_1}&0&0\\0&0&0\end{pmatrix}+\frac{1}{2\Delta r}\begin{pmatrix}0&0&0\\0&0&\frac{1}{r_1}\\0&-\frac{1}{r_2}&0\end{pmatrix}.$$

The exponentiation of a block is, for real-time evolution:

$$\exp\left[i\frac{\Delta t}{8m\Delta r}\begin{pmatrix}0&\frac{1}{r_1}\\-\frac{1}{r_2}&0\end{pmatrix}\right]=\begin{pmatrix}\cosh\beta&i\alpha\sinh\beta\\-i\frac{1}{\alpha}\sinh\beta&\cosh\beta\end{pmatrix}.$$

While, for imaginary time evolution:

$$\exp\left[\frac{\Delta t}{8m\Delta r}\begin{pmatrix}0&\frac{1}{r_1}\\-\frac{1}{r_2}&0\end{pmatrix}\right]=\begin{pmatrix}\cos\beta&\alpha\sin\beta\\-\frac{1}{\alpha}\sin\beta&\cos\beta\end{pmatrix}.$$

with $\beta=\frac{\Delta t}{8m\Delta r\sqrt{r_1r_2}}$, $\alpha=\sqrt{\frac{r_2}{r_1}}$ and $r_1,r_2>0$. However, the block matrix that contains $1/r_0$ has a different exponentiation, since :math:‘r_0 < 0 ‘.

In particular $r_0=-r_1$ and for the real-time evolution, the block is of the form

$$\exp\left[i\frac{\Delta t}{8mr_1\Delta r}\begin{pmatrix}0&-1\\-1&0\end{pmatrix}\right]=\begin{pmatrix}\cos\beta&i\sin\beta\\i\sin\beta&\cos\beta\end{pmatrix}$$

for imaginary-time evolution

$$\exp\left[\frac{\Delta t}{8mr_1\Delta r}\begin{pmatrix}0&-1\\-1&0\end{pmatrix}\right]=\begin{pmatrix}\cosh\beta&\sinh\beta\\\sinh\beta&\cosh\beta\end{pmatrix}$$

with $\beta=-\frac{\Delta t}{8mr_1\Delta r}$.

External potential

An external potential dependent on the coordinate space is trivial to calculate. For the discretization that we use, such external potential is approximated by a diagonal matrix. For real time evolution

$$\begin{aligned}\exp[-i\Delta tV]&=\exp\left[-i\Delta t\begin{pmatrix}V(x_0,y_0)&0&0\\0&V(x_1,y_0)&0\\0&0&V(x_2,y_0)\end{pmatrix}\right]\\&=\begin{pmatrix}e^{-i\Delta tV(x_0,y_0)}&0&0\\0&e^{-i\Delta tV(x_1,y_0)}&0\\0&0&e^{-i\Delta tV(x_2,y_0)}\end{pmatrix}\end{aligned}$$

and for imaginary time evolution

$$\exp[-\Delta t V] = \exp\left[-\Delta t \begin{pmatrix} V(x_0,y_0) & 0 & 0 \\ 0 & V(x_1,y_0) & 0 \\ 0 & 0 & V(x_2,y_0) \end{pmatrix}\right] = \begin{pmatrix} e^{-\Delta t V(x_0,y_0)} & 0 & 0 \\ 0 & e^{-\Delta t V(x_1,y_0)} & 0 \\ 0 & 0 & e^{-\Delta t V(x_2,y_0)} \end{pmatrix}$$

Self interaction term

The self interaction term of the wave function, $g|\psi(x,y)|^2$, depends on the coordinate space, hence its discrete form is a diagonal matrix, as in the case of the external potential. In addition, the Lee-Huang-Yang term, $g_{LHY}|\psi(x,y)|^3$, is implemented in the same way.

Angular momentum

For cartesian coordinates the Hamiltonian containing the angular momentum operator is

$$-i\omega\left(x\partial_y - y\partial_x\right).$$

For the trotter-suzuki approximation, the exponentiation is done separately for the two terms:

- First term, real-time evolution, $\beta = \frac{\Delta t\omega x}{2\Delta y}$

$$\exp[-\Delta t\omega x\partial_y] = \exp\left[-\beta\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\right] = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix}$$

- First term, imaginary-time evolution, $\beta = \frac{\Delta t\omega x}{2\Delta y}$

$$\exp[i\Delta t\omega x\partial_y] = \exp\left[-\beta\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\right] = \begin{pmatrix} \cosh\beta & i\sinh\beta \\ -i\sinh\beta & \cosh\beta \end{pmatrix}$$

- Second term, real-time evolution, $\beta = \frac{\Delta t\omega y}{2\Delta x}$

$$\exp[\Delta t\omega y\partial_x] = \exp\left[-\beta\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\right] = \begin{pmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{pmatrix}$$

- Second term, imaginary-time evolution, $\beta = \frac{\Delta t\omega y}{2\Delta x}$

$$\exp[-i\Delta t\omega y\partial_x] = \exp\left[-\beta\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\right] = \begin{pmatrix} \cosh\beta & -i\sinh\beta \\ i\sinh\beta & \cosh\beta \end{pmatrix}$$

CHAPTER 6

Function Reference

Lattice1D Class

class `trottersuzuki.`**`Lattice1D`**

This class defines the lattice structure over which the state and potential matrices are defined.

Constructors

`Lattice1D`(*dim_x*, *length_x*, *periodic_x_axis=False*, *coordinate_system="cartesian"*)

Construct a one-dimensional lattice.

Parameters

- ***dim_x*** [integer] Linear dimension of the squared lattice in the x direction.
- ***length_x*** [float] Physical length of the lattice's side in the x direction.
- ***periodic_x_axis*** [bool,optional (default: False)] Boundary condition along the x axis (false=closed, true=periodic).
- ***coordinate_system*** [string,optional (default: "cartesian")] Coordinates of the physical space ("cartesian" or "cylindrical").

Returns

- ***Lattice1D*** [Lattice1D object] Define the geometry of the simulation.

Notes

For cylindrical coordinates the radial coordinate is used.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> # Generate a 200-point Lattice1D with physical dimensions of 30
>>> # and closed boundary conditions.
>>> grid = ts.Lattice1D(200, 30.)
```

Members

get_x_axis()
Get the x-axis of the lattice.

Returns

- ***x_axis*** [numpy array] X-axis of the lattice

Attributes

length_x
Physical length of the lattice along the X-axis.

dim_x
Number of dots of the lattice along the X-axis.

delta_x
Resolution of the lattice along the X-axis: ratio between *lenth_x* and *dim_x*.

Lattice2D Class

class trottersuzuki.**Lattice2D**
This class defines the lattice structure over which the state and potential matrices are defined.

Constructors

Lattice2D(*dim_x*, *length_x*, *dim_y=None*, *length_y=None*, *periodic_x_axis=False*, *periodic_y_axis=False*, *coordinate_system="cartesian"*)
Construct the Lattice2D.

Parameters

- ***dim_x*** [integer] Linear dimension of the squared lattice in the x direction.
- ***length_x*** [float] Physical length of the lattice's side in the x direction.
- ***dim_y*** [integer,optional (default: equal to dim_x)] Linear dimension of the squared lattice in the y direction.
- ***length_y*** [float,optional (default: equal to length_x)] Physical length of the lattice's side in the y direction.
- ***periodic_x_axis*** [bool,optional (default: False)] Boundary condition along the x axis (false=closed, true=periodic).
- ***periodic_y_axis*** [bool,optional (default: False)] Boundary condition along the y axis (false=closed, true=periodic).
- ***angular_velocity*** [float, optional (default: 0.)] Angular velocity of the rotating reference frame (only for Cartesian coordinates).
- ***coordinate_system*** [string,optional (default: "cartesian")] Coordinates of the physical space ("cartesian" or "cylindrical").

Returns

- ***Lattice2D*** [Lattice2D object] Define the geometry of the simulation.

Notes

For cylindrical coordinates the radial coordinate is in place of the x-axis and the axial one is in place of the y-axis.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> # Generate a 200x200 Lattice2D with physical dimensions of 30x30
>>> # and closed boundary conditions.
>>> grid = ts.Lattice2D(200, 30.)
```

Members

get_x_axis()
Get the x-axis of the lattice.

Returns

- ***x_axis*** [numpy array] X-axis of the lattice

get_y_axis()
Get the y-axis of the lattice.

Returns

- ***y_axis*** [numpy array] Y-axis of the lattice

Attributes

length_x
Physical length of the lattice along the X-axis.

length_y
Physical length of the lattice along the Y-axis.

dim_x
Number of dots of the lattice along the X-axis.

dim_y
Number of dots of the lattice along the Y-axis.

delta_x
Resolution of the lattice along the X-axis: ratio between *lenth_x* and *dim_x*.

delta_y
Resolution of the lattice along the y-axis: ratio between *lenth_y* and *dim_y*.

State Classes

class trottersuzuki.**State**
This class defines the quantum state.

Constructors

State(*grid*, *angular_momentum*)
Create a quantum state.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.
- ***angular_momentum*** [integer, optional (default: 0)] Angular momentum for the cylindrical coordinates.

Returns

- ***state*** [State object] Quantum state.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> def wave_function(x,y):  # Define a flat wave function
>>>     return 1.
>>> state = ts.State(grid)  # Create the system's state
>>> state.ini_state(wave_function)  # Initialize the wave function of the␣
↪state
```

State (*state*)

Copy a quantum state.

Parameters

- ***state*** [State object] Quantum state to be copied

Returns

- ***state*** [State object] Quantum state.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state with a␣
↪gaussian wave function
>>> state2 = ts.State(state)  # Copy state into state2
```

Members

State.init_state(state_function):

Initialize the wave function of the state using a function.

Parameters

- ***state_function*** [python function] Python function defining the wave function of the state ψ.

Notes

The input arguments of the python function must be (x,y).

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> def wave_function(x,y):  # Define a flat wave function
>>>     return 1.
>>> state = ts.State(grid)  # Create the system's state
>>> state.ini_state(wave_function)  # Initialize the wave function of the␣
↪state
```

imprint (*function*)

Multiply the wave function of the state by the function provided.

Parameters

- ***function*** [python function] Function to be printed on the state.

Notes

Useful, for instance, to imprint solitons and vortices on a condensate. Generally, it performs a transformation of the state whose wave function becomes:

$$\psi(x,y)' = f(x,y)\psi(x,y)$$

being $f(x,y)$ the input function and $\psi(x,y)$ the initial wave function.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> def vortex(x,y):  # Vortex function
>>>     z = x + 1j*y
>>>     angle = np.angle(z)
>>>     return np.exp(1j * angle)
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.imprint(vortex)  # Imprint a vortex on the state
```

get_mean_px()
Return the expected value of the P_x operator.

Returns

- ***mean_px*** [float] Expected value of the P_x operator.

get_mean_pxpx()
Return the expected value of the P_x^2 operator.

Returns

- ***mean_pxpx*** [float] Expected value of the P_x^2 operator.

get_mean_py()
Return the expected value of the P_y operator.

Returns

- ***mean_py*** [float] Expected value of the P_y operator.

get_mean_pypy()
Return the expected value of the P_y^2 operator.

Returns

- ***mean_pypy*** [float] Expected value of the P_y^2 operator.

get_mean_x()
Return the expected value of the X operator.

Returns

- ***mean_x*** [float] Expected value of the X operator.

get_mean_xx()
Return the expected value of the X^2 operator.

Returns

- ***mean_xx*** [float] Expected value of the X^2 operator.

get_mean_y()
Return the expected value of the Y operator.

Returns

- ***mean_y*** [float] Expected value of the Y operator.

get_mean_yy()
Return the expected value of the Y^2 operator.

Returns

•***mean_yy*** [float] Expected value of the Y^2 operator.

get_particle_density()
Return a matrix storing the squared norm of the wave function.

Returns

•***particle_density*** [numpy matrix] Particle density of the state $|\psi(x, y)|^2$

get_phase()
Return a matrix of the wave function's phase.

Returns

•***get_phase*** [numpy matrix] Matrix of the wave function's phase $\phi(x, y) = \log(\psi(x, y))$

get_squared_norm()
Return the squared norm of the quantum state.

Returns

•***squared_norm*** [float] Squared norm of the quantum state.

loadtxt(*file_name*)
Load the wave function from a file.

Parameters

•***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

write_particle_density(*file_name*)
Write to a file the particle density matrix of the wave function.

Parameters

•***file_name*** [string] Name of the file.

write_phase(*file_name*)
Write to a file the wave function.

Parameters

•***file_name*** [string] Name of the file to be written.

write_to_file(*file_name*)
Write to a file the wave function.

Parameters

•***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

class trottersuzuki.**BesselState**

This class defines a quantum state with sinusoidal like wave function.

This class is a child of State class.

Constructors

BesselState(*grid*, *angular_momentum=0*, *zeros=1*, *n_y=0*, *norm=1*, *phase=0*)

Construct the quantum state with wave function given by a first kind of Bessel functions.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.
- ***angular_momentum*** [integer, optional (default: 0)] Angular momentum for the cylindrical coordinates.
- ***zeros*** [integer, optional (default: 1)] Number of zeros points along the radial axis.
- ***n_y*** [integer, optional (default: 1)] Quantum number (available if *grid* is a Lattice2D object).
- ***norm*** [float, optional (default: 1)] Squared norm of the quantum state.
- ***phase*** [float, optional (default: 1)] Relative phase of the wave function.

Returns

- ***BesselState*** [State object.] Quantum state with wave function given by a first kind of Bessel functions. The wave function is given by:

$$\psi(r, z, \phi) = f(r, z)e^{il\phi}$$

with

$$f(r, z) = \sqrt{N}/\tilde{N} J_l(r r_i / L_r) \cos(n_y \pi r/(2L_z)) \mathrm{e}^{(i\phi_0)}$$

being N the norm of the state, $\tilde{N}$ a normalization factor for J_l, J_l the Bessel function of the first kind with angulat momentum l, r_i the radial coordinate of the i-th zero of J_l L_r the length of the lattice along the radial axis, L_z the length of the lattice along the z axis, n_y the quantum number and ϕ_0 the relative phase.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D(300, 30., True, True, 0., "cylindrical")  # Define␣
↪the simulation's geometry
>>> state = ts.BesselState(grid, 2, 1, 1)  # Create the system's state
```

Members

imprint(*function*)

Multiply the wave function of the state by the function provided.

Parameters

- ***function*** [python function] Function to be printed on the state.

Notes

Useful, for instance, to imprint solitons and vortices on a condensate. Generally, it performs a transformation of the state whose wave function becomes:

$$\psi(x,y)' = f(x,y)\psi(x,y)$$

being $f(x,y)$ the input function and $\psi(x,y)$ the initial wave function.

get_mean_px()
Return the expected value of the P_x operator.

Returns

- ***mean_px*** [float] Expected value of the P_x operator.

get_mean_pxpx()
Return the expected value of the P_x^2 operator.

Returns

- ***mean_pxpx*** [float] Expected value of the P_x^2 operator.

get_mean_py()
Return the expected value of the P_y operator.

Returns

- ***mean_py*** [float] Expected value of the P_y operator.

get_mean_pypy()
Return the expected value of the P_y^2 operator.

Returns

- ***mean_pypy*** [float] Expected value of the P_y^2 operator.

get_mean_x()
Return the expected value of the X operator.

Returns

- ***mean_x*** [float] Expected value of the X operator.

get_mean_xx()
Return the expected value of the X^2 operator.

Returns

- ***mean_xx*** [float] Expected value of the X^2 operator.

get_mean_y()
Return the expected value of the Y operator.

Returns

- ***mean_y*** [float] Expected value of the Y operator.

get_mean_yy()
Return the expected value of the Y^2 operator.

Returns

- ***mean_yy*** [float] Expected value of the Y^2 operator.

get_particle_density()
Return a matrix storing the squared norm of the wave function.

Returns

- ***particle_density*** [numpy matrix] Particle density of the state $|\psi(x, y)|^2$

get_phase()
Return a matrix of the wave function's phase.

Returns

- ***get_phase*** [numpy matrix] Matrix of the wave function's phase $\phi(x, y) = \log(\psi(x, y))$

get_squared_norm()
Return the squared norm of the quantum state.

Returns

- ***squared_norm*** [float] Squared norm of the quantum state.

loadtxt(*file_name*)
Load the wave function from a file.

Parameters

- ***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D(300, 30., True, True, 0., "cylindrical")  # Define␣
↪the simulation's geometry
>>> state = ts.BesselState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

write_particle_density(*file_name*)
Write to a file the particle density matrix of the wave function.

Parameters

- ***file_name*** [string] Name of the file.

write_phase(*file_name*)
Write to a file the wave function.

Parameters

- ***file_name*** [string] Name of the file to be written.

write_to_file(*file_name*)
Write to a file the wave function.

Parameters

- ***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D(300, 30., True, True, 0., "cylindrical")  # Define␣
↪the simulation's geometry
>>> state = ts.BesselState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

class `trottersuzuki.`**ExponentialState**

This class defines a quantum state with exponential like wave function.

This class is a child of State class.

Constructors

ExponentialState (*grid*, *n_x=1*, *n_y=1*, *norm=1*, *phase=0*)

Construct the quantum state with exponential like wave function.

Parameters

- ***grid*** [Lattice object] Defines the geometry of the simulation.
- ***n_x*** [integer,optional (default: 1)] First quantum number.
- ***n_y*** [integer,optional (default: 1)] Second quantum number (available if *grid* is a Lattice2D object).
- ***norm*** [float,optional (default: 1)] Squared norm of the quantum state.
- ***phase*** [float,optional (default: 0)] Relative phase of the wave function.

Returns

- ***ExponentialState*** [State object.] Quantum state with exponential like wave function. The wave function is give by:n

$$\psi(x,y) = \sqrt{N}/L \mathrm{e}^{i2\pi(n_x x + n_y y)/L} \mathrm{e}^{i\phi}$$

being N the norm of the state, L the length of the lattice edge, n_x and n_y the quantum numbers and ϕ the relative phase.

Notes

The geometry of the simulation has to have periodic boundary condition to use Exponential state as initial state of a real time evolution. Indeed, the wave function is not null at the edges of the space.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D(300, 30., True, True)  # Define the simulation's␣
↪geometry
>>> state = ts.ExponentialState(grid, 2, 1)  # Create the system's state
```

Member

imprint (*function*)

Multiply the wave function of the state by the function provided.

Parameters

- ***function*** [python function] Function to be printed on the state.

Notes

Useful, for instance, to imprint solitons and vortices on a condensate. Generally, it performs a transformation of the state whose wave function becomes:

$$\psi(x,y)' = f(x,y)\psi(x,y)$$

being $f(x,y)$ the input function and $\psi(x,y)$ the initial wave function.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> def vortex(x,y):  # Vortex function
>>>     z = x + 1j*y
>>>     angle = np.angle(z)
>>>     return np.exp(1j * angle)
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.imprint(vortex)  # Imprint a vortex on the state
```

get_mean_px()
Return the expected value of the P_x operator.

Returns

- ***mean_px*** [float] Expected value of the P_x operator.

get_mean_pxpx()
Return the expected value of the P_x^2 operator.

Returns

- ***mean_pxpx*** [float] Expected value of the P_x^2 operator.

get_mean_py()
Return the expected value of the P_y operator.

Returns

- ***mean_py*** [float] Expected value of the P_y operator.

get_mean_pypy()
Return the expected value of the P_y^2 operator.

Returns

- ***mean_pypy*** [float] Expected value of the P_y^2 operator.

get_mean_x()
Return the expected value of the X operator.

Returns

- ***mean_x*** [float] Expected value of the X operator.

get_mean_xx()
Return the expected value of the X^2 operator.

Returns

- ***mean_xx*** [float] Expected value of the X^2 operator.

get_mean_y()
Return the expected value of the Y operator.

Returns

- ***mean_y*** [float] Expected value of the Y operator.

get_mean_yy()
Return the expected value of the Y^2 operator.

Returns

- ***mean_yy*** [float] Expected value of the Y^2 operator.

get_particle_density()
Return a matrix storing the squared norm of the wave function.

Returns

- ***particle_density*** [numpy matrix] Particle density of the state $|\psi(x, y)|^2$

get_phase()
Return a matrix of the wave function's phase.

Returns

- ***get_phase*** [numpy matrix] Matrix of the wave function's phase $\phi(x, y) = \log(\psi(x, y))$

get_squared_norm()
Return the squared norm of the quantum state.

Returns

- ***squared_norm*** [float] Squared norm of the quantum state.

loadtxt(*file_name*)
Load the wave function from a file.

Parameters

- ***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

write_particle_density(*file_name*)
Write to a file the particle density matrix of the wave function.

Parameters

- ***file_name*** [string] Name of the file.

write_phase(*file_name*)
Write to a file the wave function.

Parameters

- ***file_name*** [string] Name of the file to be written.

write_to_file(*file_name*)
Write to a file the wave function.

Parameters

- ***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

class trottersuzuki.**GaussianState**

This class defines a quantum state with gaussian like wave function.

This class is a child of State class.

Constructors

GaussianState(*grid*, *omega_x*, *omega_y=omega_x*, *mean_x=0*, *mean_y=0*, *norm=1*, *phase=0*)

Construct the quantum state with gaussian like wave function.

Parameters

- ***grid*** [Lattice object] Defines the geometry of the simulation.
- ***omega_x*** [float] Inverse of the variance along x-axis.
- ***omega_y*** [float, optional (default: omega_x)] Inverse of the variance along y-axis (available if *grid* is a Lattice2D object).
- ***mean_x*** [float, optional (default: 0)] X coordinate of the gaussian function's peak.
- ***mean_y*** [float, optional (default: 0)] Y coordinate of the gaussian function's peak (available if *grid* is a Lattice2D object).
- ***norm*** [float, optional (default: 1)] Squared norm of the state.
- ***phase*** [float, optional (default: 0)] Relative phase of the wave function.

Returns

- ***GaussianState*** [State object.] Quantum state with gaussian like wave function. The wave function is given by:n

$$\psi(x,y) = (N/\pi)^{1/2}(\omega_x\omega_y)^{1/4}\mathrm{e}^{-(\omega_x(x-\mu_x)^2+\omega_y(y-\mu_y)^2)/2}\mathrm{e}^{i\phi}$$

being N the norm of the state, ω_x and ω_y the inverse of the variances, μ_x and μ_y the coordinates of the function's peak and ϕ the relative phase.

Notes

The physical dimensions of the Lattice2D have to be enough to ensure that the wave function is almost zero at the edges.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D(300, 30.)  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 2.)  # Create the system's state
```

Members

imprint(*function*)

Multiply the wave function of the state by the function provided.

Parameters

- ***function*** [python function] Function to be printed on the state.

Notes

Useful, for instance, to imprint solitons and vortices on a condensate. Generally, it performs a transformation of the state whose wave function becomes:

$$\psi(x,y)' = f(x,y)\psi(x,y)$$

being $f(x,y)$ the input function and $\psi(x,y)$ the initial wave function.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> def vortex(x,y):  # Vortex function
>>>     z = x + 1j*y
>>>     angle = np.angle(z)
>>>     return np.exp(1j * angle)
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.imprint(vortex)  # Imprint a vortex on the state
```

get_mean_px()
Return the expected value of the P_x operator.

Returns

- ***mean_px*** [float] Expected value of the P_x operator.

get_mean_pxpx()
Return the expected value of the P_x^2 operator.

Returns

- ***mean_pxpx*** [float] Expected value of the P_x^2 operator.

get_mean_py()
Return the expected value of the P_y operator.

Returns

- ***mean_py*** [float] Expected value of the P_y operator.

get_mean_pypy()
Return the expected value of the P_y^2 operator.

Returns

- ***mean_pypy*** [float] Expected value of the P_y^2 operator.

get_mean_x()
Return the expected value of the X operator.

Returns

- ***mean_x*** [float] Expected value of the X operator.

get_mean_xx()
Return the expected value of the X^2 operator.

Returns

- ***mean_xx*** [float] Expected value of the X^2 operator.

get_mean_y()
Return the expected value of the Y operator.

Returns

- ***mean_y*** [float] Expected value of the Y operator.

get_mean_yy()
Return the expected value of the Y^2 operator.

Returns

- ***mean_yy*** [float] Expected value of the Y^2 operator.

get_particle_density()
Return a matrix storing the squared norm of the wave function.

Returns

- ***particle_density*** [numpy matrix] Particle density of the state $|\psi(x, y)|^2$

get_phase()
Return a matrix of the wave function's phase.

Returns

- ***get_phase*** [numpy matrix] Matrix of the wave function's phase $\phi(x, y) = \log(\psi(x, y))$

get_squared_norm()
Return the squared norm of the quantum state.

Returns

- ***squared_norm*** [float] Squared norm of the quantum state.

loadtxt(*file_name*)
Load the wave function from a file.

Parameters

- ***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

write_particle_density(*file_name*)
Write to a file the particle density matrix of the wave function.

Parameters

- ***file_name*** [string] Name of the file.

write_phase(*file_name*)
Write to a file the wave function.

Parameters

- ***file_name*** [string] Name of the file to be written.

write_to_file(*file_name*)
Write to a file the wave function.

Parameters

- ***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

class trottersuzuki.**SinusoidState**
This class defines a quantum state with sinusoidal like wave function.

This class is a child of State class.

Constructors

SinusoidState(*grid*, *n_x=1*, *n_y=1*, *norm=1*, *phase=0*)
Construct the quantum state with sinusoidal like wave function.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.
- ***n_x*** [integer, optional (default: 1)] First quantum number.
- ***n_y*** [integer, optional (default: 1)] Second quantum number (available if *grid* is a Lattice2D object).
- ***norm*** [float, optional (default: 1)] Squared norm of the quantum state.
- ***phase*** [float, optional (default: 1)] Relative phase of the wave function.

Returns

- ***SinusoidState*** [State object.] Quantum state with sinusoidal like wave function. The wave function is given by:

$$\psi(x,y) = 2\sqrt{N}/L \sin(2\pi n_x x/L) \sin(2\pi n_y y/L) \mathrm{e}^{(i\phi)}$$

being N the norm of the state, L the length of the lattice edge, n_x and n_y the quantum numbers and ϕ the relative phase.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D(300, 30., True, True)  # Define the simulation's␣
↪geometry
>>> state = ts.SinusoidState(grid, 2, 0)  # Create the system's state
```

Members

imprint(*function*)

Multiply the wave function of the state by the function provided.

Parameters

- ***function*** [python function] Function to be printed on the state.

Notes

Useful, for instance, to imprint solitons and vortices on a condensate. Generally, it performs a transformation of the state whose wave function becomes:

$$\psi(x,y)' = f(x,y)\psi(x,y)$$

being $f(x,y)$ the input function and $\psi(x,y)$ the initial wave function.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> def vortex(x,y):  # Vortex function
>>>     z = x + 1j*y
>>>     angle = np.angle(z)
>>>     return np.exp(1j * angle)
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.imprint(vortex)  # Imprint a vortex on the state
```

get_mean_px()
Return the expected value of the P_x operator.

Returns

- ***mean_px*** [float] Expected value of the P_x operator.

get_mean_pxpx()
Return the expected value of the P_x^2 operator.

Returns

- ***mean_pxpx*** [float] Expected value of the P_x^2 operator.

get_mean_py()
Return the expected value of the P_y operator.

Returns

- ***mean_py*** [float] Expected value of the P_y operator.

get_mean_pypy()
Return the expected value of the P_y^2 operator.

Returns

- ***mean_pypy*** [float] Expected value of the P_y^2 operator.

get_mean_x()
Return the expected value of the X operator.

Returns

- ***mean_x*** [float] Expected value of the X operator.

get_mean_xx()
Return the expected value of the X^2 operator.

Returns

- ***mean_xx*** [float] Expected value of the X^2 operator.

get_mean_y()
Return the expected value of the Y operator.

Returns

- ***mean_y*** [float] Expected value of the Y operator.

get_mean_yy()
Return the expected value of the Y^2 operator.

Returns

- ***mean_yy*** [float] Expected value of the Y^2 operator.

get_particle_density()
Return a matrix storing the squared norm of the wave function.

Returns

- ***particle_density*** [numpy matrix] Particle density of the state $|\psi(x, y)|^2$

get_phase()
Return a matrix of the wave function's phase.

Returns

- ***get_phase*** [numpy matrix] Matrix of the wave function's phase $\phi(x, y) = \log(\psi(x, y))$

get_squared_norm()
Return the squared norm of the quantum state.

Returns

- ***squared_norm*** [float] Squared norm of the quantum state.

loadtxt(*file_name*)
Load the wave function from a file.

Parameters

- ***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

write_particle_density(*file_name*)
Write to a file the particle density matrix of the wave function.

Parameters

- ***file_name*** [string] Name of the file.

write_phase(*file_name*)
Write to a file the wave function.

Parameters

- ***file_name*** [string] Name of the file to be written.

write_to_file(*file_name*)
Write to a file the wave function.

Parameters

- ***file_name*** [string] Name of the file to be written.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> state.write_to_file('wave_function.txt')  # Write to a file the wave␣
↪function
>>> state2 = ts.State(grid)  # Create a quantum state
>>> state2.loadtxt('wave_function.txt')  # Load the wave function
```

Potential Classes

class `trottersuzuki.`**`Potential`**

This class defines the external potential that is used for Hamiltonian class.

Constructors

`Potential`(*grid*)

Construct the external potential.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.

Returns

- ***Potential*** [Potential object] Create external potential.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> # Define a constant external potential
>>> def external_potential_function(x,y):
>>>     return 1.
>>> potential = ts.Potential(grid)  # Create the external potential
>>> potential.init_potential(external_potential_function)  # Initialize the␣
↪external potential
```

Members

`init_potential`(*potential_function*)

Initialize the external potential.

Parameters

- ***potential_function*** [python function] Define the external potential function.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> # Define a constant external potential
>>> def external_potential_function(x,y):
>>>     return 1.
>>> potential = ts.Potential(grid)  # Create the external potential
>>> potential.init_potential(external_potential_function)  # Initialize the␣
↪external potential
```

get_value(*x*, *y*)
Get the value at the lattice's coordinate (x,y).

Returns

- ***value*** [float] Value of the external potential.

class trottersuzuki.**HarmonicPotential**
This class defines the external potential, that is used for Hamiltonian class.

This class is a child of Potential class.

Constructors

HarmonicPotential(grid, omegax, omegay, mass=1., mean_x=0., mean_y=0.)`
Construct the harmonic external potential.

Parameters

- ***grid*** [Lattice2D object] Define the geometry of the simulation.
- ***omegax*** [float] Frequency along x-axis.
- ***omegay*** [float] Frequency along y-axis.
- ***mass*** [float,optional (default: 1.)] Mass of the particle.
- ***mean_x*** [float,optional (default: 0.)] Minimum of the potential along x axis.
- ***mean_y*** [float,optional (default: 0.)] Minimum of the potential along y axis.

Returns

- ***HarmonicPotential*** [Potential object] Harmonic external potential.

Notes

External potential function:n

$$V(x, y) = 1/2m(\omega_x^2 x^2 + \omega_y^2 y^2)$$

being m the particle mass, ω_x and ω_y the potential frequencies.

Example

```
>>> import trottersuzuki as ts  # Import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> potential = ts.HarmonicPotential(grid, 2., 1.)  # Create an harmonic␣
↪external potential
```

Members

get_value(*x*, *y*)
Get the value at the lattice's coordinate (x,y).

Returns

- ***value*** [float] Value of the external potential.

Hamiltonian Classes

class trottersuzuki.**Hamiltonian**
This class defines the Hamiltonian of a single component system.

Constructors

Hamiltonian(*grid*, *potential=0*, *mass=1.*, *coupling=0.*, *LeeHuangYang_coupling=0.*, *angular_velocity=0.*, *rot_coord_x=0*, *rot_coord_y=0*)

Construct the Hamiltonian of a single component system.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.
- ***potential*** [Potential object] Define the external potential of the Hamiltonian (V).
- ***mass*** [float,optional (default: 1.)] Mass of the particle (m).
- ***coupling*** [float,optional (default: 0.)] Coupling constant of intra-particle interaction (g).
- ***LeeHuangYang_coupling*** [float,optional (default: 0.)] Coupling constant of the Lee-Huang-Yang term (g_{LHY}).
- ***angular_velocity*** [float,optional (default: 0.)] The frame of reference rotates with this angular velocity (ω).
- ***rot_coord_x*** [float,optional (default: 0.)] X coordinate of the center of rotation.
- ***rot_coord_y*** [float,optional (default: 0.)] Y coordinate of the center of rotation.

Returns

- ***Hamiltonian*** [Hamiltonian object] Hamiltonian of the system to be simulated:

$$H(x,y) = \frac{1}{2m}(P_x^2 + P_y^2) + V(x,y) + g|\psi(x,y)|^2 + g_{LHY}|\psi(x,y)|^3 + \omega L_z$$

being m the particle mass, $V(x,y)$ the external potential, g the coupling constant of intra-particle interaction, g_{LHY} the coupling constant of the Lee-Huang-Yang term, ω the angular velocity of the frame of reference and L_z the angular momentum operator along the z-axis.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> potential = ts.HarmonicPotential(grid, 1., 1.)  # Create an harmonic␣
↪external potential
>>> hamiltonian = ts.Hamiltonian(grid, potential)  # Create the Hamiltonian␣
↪of an harmonic oscillator
```

class trottersuzuki.**Hamiltonian2Component**

This class defines the Hamiltonian of a two component system.

Constructors

Hamiltonian2Component(*grid*, *potential_1=0*, *potential_2=0*, *mass_1=1.*, *mass_2=1.*, *coupling_1=0.*, *coupling_12=0.*, *coupling_2=0.*, *omega_r=0*, *omega_i=0*, *angular_velocity=0.*, *rot_coord_x=0*, *rot_coord_y=0*)

Construct the Hamiltonian of a two component system.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.
- ***potential_1*** [Potential object] External potential to which the first state is subjected (V_1).
- ***potential_2*** [Potential object] External potential to which the second state is subjected (V_2).
- ***mass_1*** [float,optional (default: 1.)] Mass of the first-component's particles (m_1).
- ***mass_2*** [float,optional (default: 1.)] Mass of the second-component's particles (m_2).

- ***coupling_1*** [float,optional (default: 0.)] Coupling constant of intra-particle interaction for the first component (g_1).
- ***coupling_12*** [float,optional (default: 0.)] Coupling constant of inter-particle interaction between the two components (g_{12}).
- ***coupling_2*** [float,optional (default: 0.)] Coupling constant of intra-particle interaction for the second component (g_2).
- ***omega_r*** [float,optional (default: 0.)] Real part of the Rabi coupling ($\mathrm{Re}(\Omega)$).
- ***omega_i*** [float,optional (default: 0.)] Imaginary part of the Rabi coupling ($\mathrm{Im}(\Omega)$).
- ***angular_velocity*** [float,optional (default: 0.)] The frame of reference rotates with this angular velocity (ω).
- ***rot_coord_x*** [float,optional (default: 0.)] X coordinate of the center of rotation.
- ***rot_coord_y*** [float,optional (default: 0.)] Y coordinate of the center of rotation.

Returns

- ***Hamiltonian2Component*** [Hamiltonian2Component object] Hamiltonian of the two-component system to be simulated.

$$H = \begin{bmatrix} H_1 & \frac{\Omega}{2} \\ \frac{\Omega}{2} & H_2 \end{bmatrix}$$

being

$$H_1 = \frac{1}{2m_1}(P_x^2 + P_y^2) + V_1(x,y) + g_1|\psi_1(x,y)|^2 + g_{12}|\psi_2(x,y)|^2 + \omega L_z$$
$$H_2 = \frac{1}{2m_2}(P_x^2 + P_y^2) + V_2(x,y) + g_2|\psi_2(x,y)|^2 + g_{12}|\psi_1(x,y)|^2 + \omega L_z$$

and, for the i-th component, m_i the particle mass, $V_i(x,y)$ the external potential, g_i the coupling constant of intra-particle interaction; g_{12} the coupling constant of inter-particle interaction ω the angular velocity of the frame of reference, L_z the angular momentum operator along the z-axis and Ω the Rabi coupling.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> potential = ts.HarmonicPotential(grid, 1., 1.)  # Create an harmonic␣
↪external potential
>>> hamiltonian = ts.Hamiltonian2Component(grid, potential, potential)  #␣
↪Create the Hamiltonian of an harmonic oscillator for a two-component system
```

Solver Class

class `trottersuzuki.`**`Solver`**

This class defines the evolution tasks.

Constructors

`Solver(grid, state1, hamiltonian, delta_t, Potential=None, State2=None, Potential2=None, kernel_type="cpu")`

Construct the Solver object. Potential is only to be passed if it is time-evolving.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.
- ***state1*** [State object] First component's state of the system.
- ***hamiltonian*** [Hamiltonian object] Hamiltonian of the two-component system.
- ***delta_t*** [float] A single evolution iteration, evolves the state for this time.
- ***Potential*****: Potential object, optional.** Time-evolving potential in component one.
- ***state2*** [State object, optional.] Second component's state of the system.
- ***Potential2*****: Potential object, optional.** Time-evolving potential in component two.
- ***kernel_type*** [str, optional (default: 'cpu')] Which kernel to use (either cpu or gpu).

Returns

- ***Solver*** [Solver object] Solver object for the simulation of a two-component system.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state_1 = ts.GaussianState(grid, 1.)  # Create first-component system's␣
↪state
>>> state_2 = ts.GaussianState(grid, 1.)  # Create second-component system's␣
↪state
>>> potential = ts.HarmonicPotential(grid, 1., 1.)  # Create harmonic␣
↪potential
>>> hamiltonian = ts.Hamiltonian2Component(grid, potential, potential)  #␣
↪Create an harmonic oscillator Hamiltonian
>>> solver = ts.Solver(grid, state_1, hamiltonian, 1e-2, State2=state_2)  #␣
↪Create the solver
```

Members

evolve(*iterations*, *imag_time=False*)

Evolve the state of the system.

Parameters

- ***iterations*** [integer] Number of iterations.
- ***imag_time*** [bool,optional (default: False)] Whether to perform imaginary time evolution (True) or real time evolution (False).

Notes

The norm of the state is preserved both in real-time and in imaginary-time evolution.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> potential = ts.HarmonicPotential(grid, 1., 1.)  # Create harmonic␣
↪potential
>>> hamiltonian = ts.Hamiltonian(grid, potential)  # Create a harmonic␣
↪oscillator Hamiltonian
>>> solver = ts.Solver(grid, state, hamiltonian, 1e-2)  # Create the solver
>>> solver.evolve(1000)  # perform 1000 iteration in real time evolution
```

get_inter_species_energy()
Get the inter-particles interaction energy of the system.

Returns

- ***get_inter_species_energy*** [float] Inter-particles interaction energy of the system.

get_intra_species_energy(*which=3*)
Get the intra-particles interaction energy of the system.

Parameters

- ***which*** [integer,optional (default: 3)] Which intra-particles interaction energy to return: total system (default, which=3), first component (which=1), second component (which=2).

get_kinetic_energy(*which=3*)
Get the kinetic energy of the system.

Parameters

- ***which*** [integer,optional (default: 3)] Which kinetic energy to return: total system (default, which=3), first component (which=1), second component (which=2).

get_potential_energy(*which=3*)
Get the potential energy of the system.

Parameters

- ***which*** [integer,optional (default: 3)] Which potential energy to return: total system (default, which=3), first component (which=1), second component (which=2).

get_rabi_energy()
Get the Rabi energy of the system.

Returns

- ***get_rabi_energy*** [float] Rabi energy of the system.

get_rotational_energy(*which=3*)
Get the rotational energy of the system.

Parameters

- ***which*** [integer,optional (default: 3)] Which rotational energy to return: total system (default, which=3), first component (which=1), second component (which=2).

get_squared_norm(*which=3*)
Get the squared norm of the state (default: total wave-function).

Parameters

- ***which*** [integer,optional (default: 3)] Which squared state norm to return: total system (default, which=3), first component (which=1), second component (which=2).

get_LeeHuangYang_energy()
Get the Lee-Huang-Yang energy.

Returns

- ***LeeHuangYang_energy*** [float] Lee-Huang-Yang energy of the system.

get_total_energy()
Get the total energy of the system.

Returns

- ***get_total_energy*** [float] Total energy of the system.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state
>>> potential = ts.HarmonicPotential(grid, 1., 1.)  # Create harmonic␣
↪potential
>>> hamiltonian = ts.Hamiltonian(grid, potential)  # Create a harmonic␣
↪oscillator Hamiltonian
>>> solver = ts.Solver(grid, state, hamiltonian, 1e-2)  # Create the solver
>>> solver.get_total_energy()  # Get the total energy
1
```

Solver::update_parameters()
Notify the solver if any parameter changed in the Hamiltonian

Tools

map_lattice_to_coordinate_space(*grid*, *x*, *y=None*)
Map the lattice coordinate to the coordinate space depending on the coordinate system.

Parameters

- ***grid*** [Lattice object] Defines the topology.
- ***x*** [int.] Grid point.
- ***y*** [int, optional.] Grid point, 2D case.

Returns

- ***x_p*, *y_p*** [tuple.] Coordinate of the physical space.

get_vortex_position(*grid*, *state*, *approx_cloud_radius=0.*)
Get the position of a single vortex in the quantum state.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.
- ***state*** [State object] System's state.
- ***approx_cloud_radius*** [float, optional] Radius of the circle, centered at the Lattice2D's origin, where the vortex core is expected to be. Need for a better accuracy.

Returns

- ***coords*** [numpy array] Coordinates of the vortex core's position (coords[0]: x coordinate; coords[1]: y coordinate).

Notes

Only one vortex must be present in the state.

Example

```
>>> import trottersuzuki as ts  # import the module
>>> import numpy as np
>>> grid = ts.Lattice2D()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create a state with gaussian wave␣
↪function
>>> def vortex_a(x, y):  # Define the vortex to be imprinted
```

```
>>>     z = x + 1j*y
>>>     angle = np.angle(z)
>>>     return np.exp(1j * angle)
>>> state.imprint(vortex)  # Imprint the vortex on the state
>>> ts.get_vortex_position(grid, state)
array([  8.88178420e-16,   8.88178420e-16])
```

Index

H

I

L

M

P

S

W