Trotter-Suzuki-MPI Python Documentation

Release 1.5.1

Peter Wittek, Luca Calderaro

February 15, 2016

CHAPTER

ONE

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:

- Fast execution by parallelization: OpenMP and CUDA are supported.
- Many-body simulations with non-interacting particles.
- Gross-Pitaevskii equation.
- 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.

1.1 Copyright and License

1.2 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.

1.3 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

TWO

DOWNLOAD AND INSTALLATION

The entire package for is available from the Python Package Index, containing the source code and examples. The documentation is hosted on Read the Docs.

The latest development version is available on GitHub.

2.1 Dependencies

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

2.1.1 Installation

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

```
$ sudo pip install trottersuzuki
```

If you want the latest git version, follow the standard procedure for installing Python modules:

```
$ sudo python setup.py install
```

2.1.2 Build on Mac OS X

Before installing using pip, 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

THREE

QUICK START GUIDE

3.1 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.Lattice(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.

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 $\hbar = 1$):

$$i\frac{\partial}{\partial t}\psi(t) = H\psi(t) \tag{3.1}$$

being

$$H = \frac{1}{2m}(P_x^2 + P_y^2) + V(x,y) + g|\psi(x,y)|^2 + \omega L_z \tag{3.2}$$

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

$$H = \begin{bmatrix} H_1 & \frac{\Omega}{2} \\ \frac{\Omega}{2} & H_2 \end{bmatrix} \tag{3.3}$$

where

$$H_1 = \frac{1}{2m_1}(P_x^2 + P_y^2) + V_1(x,y) + g_1|\psi(x,y)_1|^2 + g_{12}|\psi(x,y)_2|^2 + \omega L_z \tag{3.4}$$

$$H_2 = \frac{1}{2m_2}(P_x^2 + P_y^2) + V_2(x,y) + g_2|\psi(x,y)_2|^2 + g_{12}|\psi(x,y)_1|^2 + \omega L_z \tag{3.5}$$

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 Gauss
```

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
```

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
```

3.2 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).

3.2.1 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
```

3.2.2 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
```

3.2.3 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
```

3.2.4 Imprinting

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

$$\psi(x, y) \to \psi'(x, y) = f(x, y)\psi(x, y) \tag{3.6}$$

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
```

3.2.5 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

FOUR

EXAMPLES

4.1 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 = Lattice(256, 15)  # create a 2D lattice

potential = HarmonicPotential(grid, 1, 1)  # define an symmetric harmonic potential with unit freque
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 s

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())
```

4.2 Imaginary time evolution to approximate the ground-state energy

```
import numpy as np
from trottersuzuki import *

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

potential = HarmonicPotential(grid, 1, 1)  # define an symmetric harmonic potential with unit freque
```

```
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 s

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())
```

4.3 Imprinting of a vortex in a Bose-Einstein Condensate

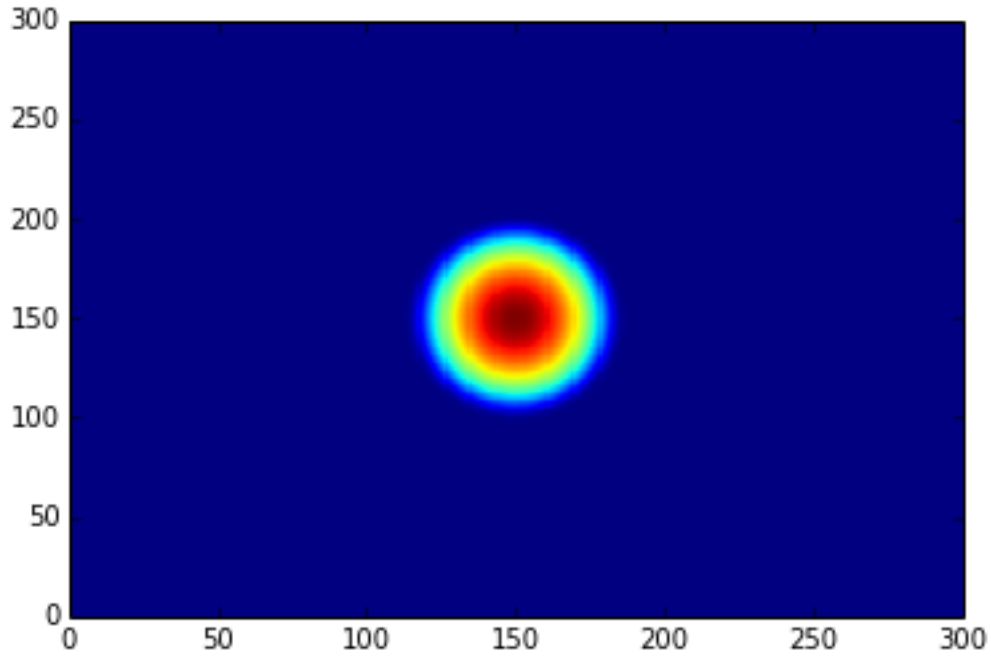
```
import numpy as np
import trottersuzuki as ts

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

potential = HarmonicPotential(grid, 1, 1)  # define an symmetric harmonic potential with unit freque
particle_mass = 1.
coupling_intra_particle_interaction = 100.
hamiltonian = Hamiltonian(grid, potential, particle_mass, coupling_intra_particle_interaction)  # de

frequency = 1
state = GaussianState(grid, frequency)  # define gaussian wave function state: we choose the ground s
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
```

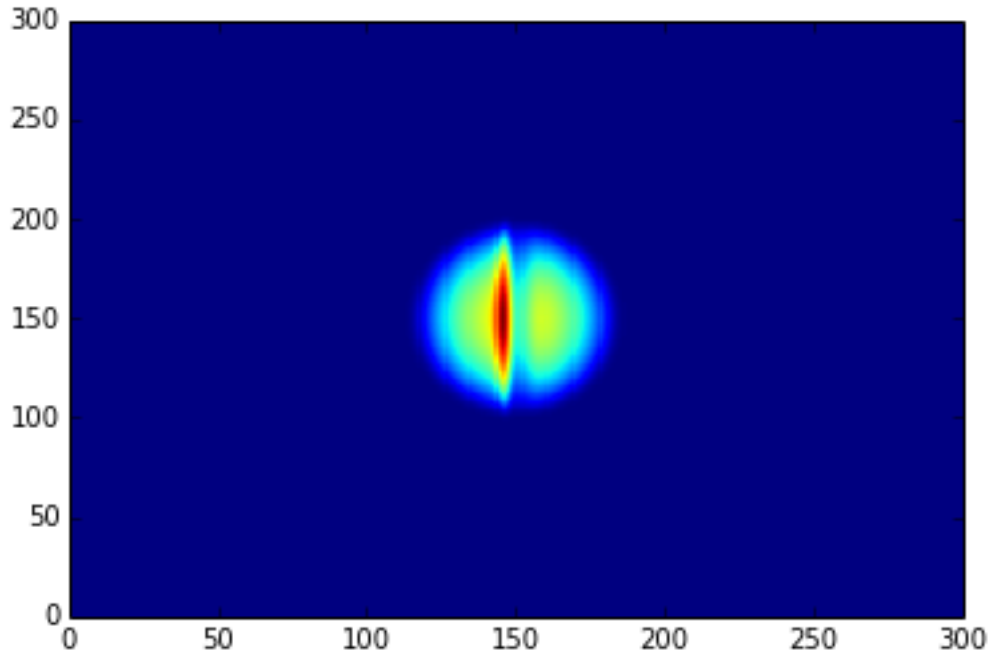
4.4 Dark Soliton Generation in Bose-Einstein Condensate using Phase Imprinting

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.Lattice(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:




300
250
200
150
100
50
0
0
50
100
150
200
250
300

CHAPTER

FIVE

FUNCTION REFERENCE

5.1 Lattice Class

class `trottersuzuki.`**`Lattice`**

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

Constructors

`Lattice`(*dim=100*, *length=20.0*, *periodic_x_axis=False*, *periodic_y_axis=False*)

Construct the Lattice.

Parameters

- ***dim*** [integer,optional (default: 100)] Linear dimension of the squared lattice.
- ***length*** [float,optional (default: 20.)] Physical length of the lattice's side.
- ***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).

Returns

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

Notes

The lattice created is squared.

Example

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

5.2 State Classes

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

Constructors

State(*grid*)
Create a quantum state.

Parameters

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

Returns

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

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice()  # 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.Lattice()  # Define the simulation's geometry
>>> state = ts.GaussianState(grid, 1.)  # Create the system's state with a gaussian wave fur
>>> 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.Lattice()  # 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.Lattice()  # 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.Lattice()  # 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.Lattice()  # 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.**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.

- ***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.Lattice(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.Lattice()  # 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.Lattice()  # 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.Lattice()  # 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.
- ***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.
- ***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 Lattice 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.Lattice(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.Lattice()  # 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.Lattice()  # 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.Lattice()  # 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.
- ***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.Lattice(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.Lattice()  # 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.Lattice()  # 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.Lattice()  # 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
```

5.3 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.Lattice()  # 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 potenti
```

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.Lattice()  # 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 potenti
```

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*** [Lattice 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.Lattice()  # 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.

5.4 Hamiltonian Classes

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

Constructors

Hamiltonian(*grid*, *potential=0*, *mass=1.*, *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).
- ***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:

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

being m the particle mass, $V(x,y)$ the external potential, g the coupling constant of intra-particle interaction, 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.Lattice()  # 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 c
```

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.

$$H = \begin{bmatrix} H_1 & \frac{\Omega}{2} \\ \frac{\Omega}{2} & H_2 \end{bmatrix} \tag{5.1}$$

being

$$H_1 = \frac{1}{2m_1}(P_x^2 + P_y^2) + V_1(x,y) + g_1|\psi_1(x,y)|^2 + g_{12}|\psi_2(x,y)|^2 + \omega L_z \tag{5.2}$$

$$H_2 = \frac{1}{2m_2}(P_x^2 + P_y^2) + V_2(x,y) + g_2|\psi_2(x,y)|^2 + g_{12}|\psi_1(x,y)|^2 + \omega L_z \tag{5.3}$$

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.Lattice()  # Define the simulation's geometry
>>> potential = ts.HarmonicPotential(grid, 1., 1.)  # Create an harmonic external potential
>>> hamiltonian = ts.Hamiltonian2Component(grid, potential, potential)  # Create the Hamiltc
```

5.5 Solver Class

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

This class defines the evolution tasks.

Constructors

`Solver`(*grid*, *state*, *hamiltonian*, *delta_t*, *kernel_type="cpu"*)

Construct the Solver object for a single-component system.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.
- ***state*** [State object] State of the system.
- ***hamiltonian*** [Hamiltonian object] Hamiltonian of the system.
- ***delta_t*** [float] A single evolution iteration, evolves the state for this time.
- ***kernel_type*** [string,optional (default: 'cpu')] Which kernel to use (either cpu or gpu).

Returns

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

Example

```
>>> import trottersuzuki as ts  # import the module
>>> grid = ts.Lattice()  # 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 Hamiltonia
>>> solver = ts.Solver(grid, state, hamiltonian, 1e-2)  # Create the solver
```

`Solver`(*grid*, *state1*, *state2*, *hamiltonian*, *delta_t*, *kernel_type="cpu"*)

Construct the Solver object for a two-component system.

Parameters

- ***grid*** [Lattice object] Define the geometry of the simulation.
- ***state1*** [State object] First component's state of the system.
- ***state2*** [State object] Second 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.
- ***kernel_type*** [string,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.Lattice()  # 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
>>> solver = ts.Solver(grid, state_1, state_2, hamiltonian, 1e-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.Lattice()  # 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 Hamiltonia
>>> 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_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.Lattice()  # 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 Hamiltonia
>>> 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

5.6 Tools

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 Lattice'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.Lattice()  # 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.vortex_position(grid, state)
array([  8.88178420e-16,   8.88178420e-16])
```

D

E

G

H

I

L

P

S

V

W