# Trotter-Suzuki-MPI Python Documentation

***Release 1.5***

**Peter Wittek, Luca Calderaro**

January 31, 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

# EXAMPLES

## 3.1 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 *

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

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

# define the Hamiltonian:
particle_mass = 1.
hamiltonian = Hamiltonian(grid, potential, particle_mass)

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

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

# 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()

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

## 3.2 Imaginary time evolution to approximate the ground-state energy

```
import numpy as np
from trottersuzuki import *

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

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

# define the Hamiltonian:
particle_mass = 1.
hamiltonian = Hamiltonian(grid, potential, particle_mass)

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

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

# 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()

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

# 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()
```

## 3.3 Dark Soliton Generation in Bose-Einstein Condensate using Phase Imprinting

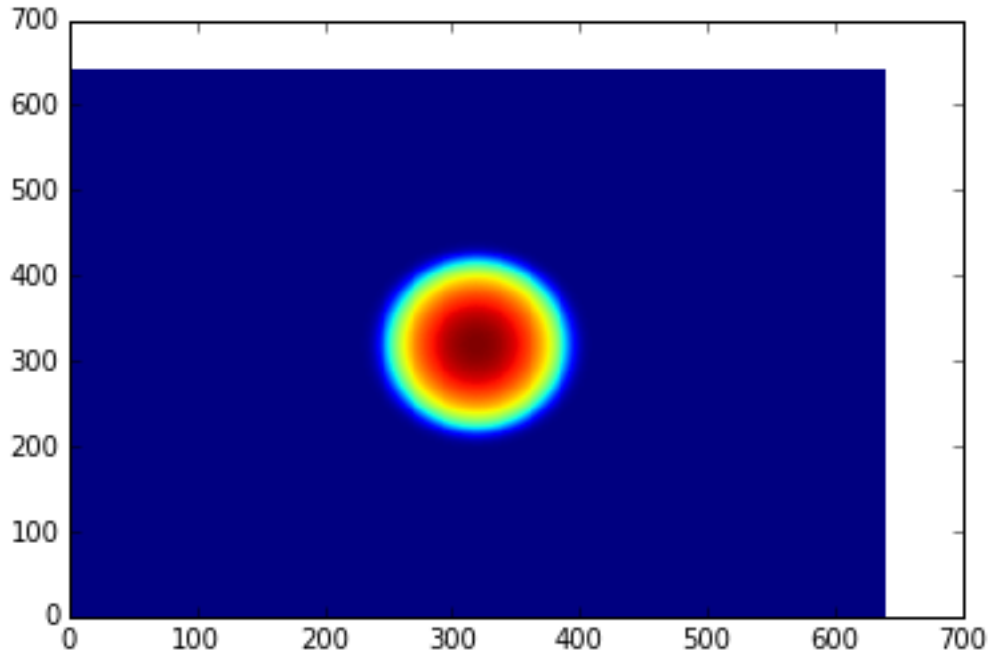
This example simulates the evolution of a dark soliton in a Bose-Einstein Condensate. For a more detailed description, refer to this notebook.

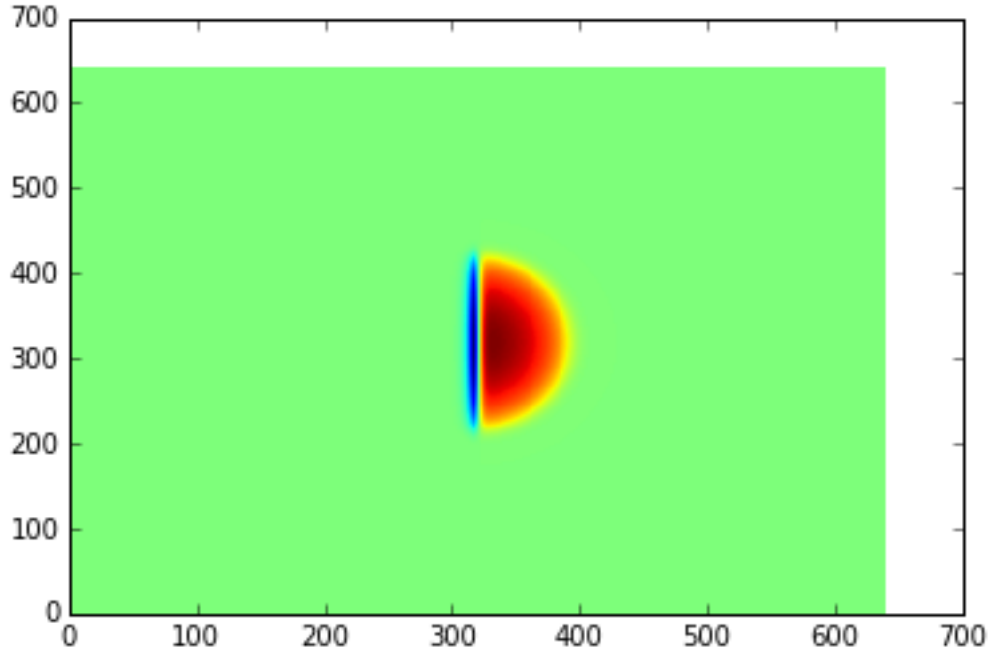
```
from __future__ import print_function
import numpy as np
import trottersuzuki as ts
from matplotlib import pyplot as plt


def get_external_potential(dim):
    """Helper function to define external potential.
    """
    def ext_pot(_x, _y):
        x = (_x - dim*0.5) * delta_x
```

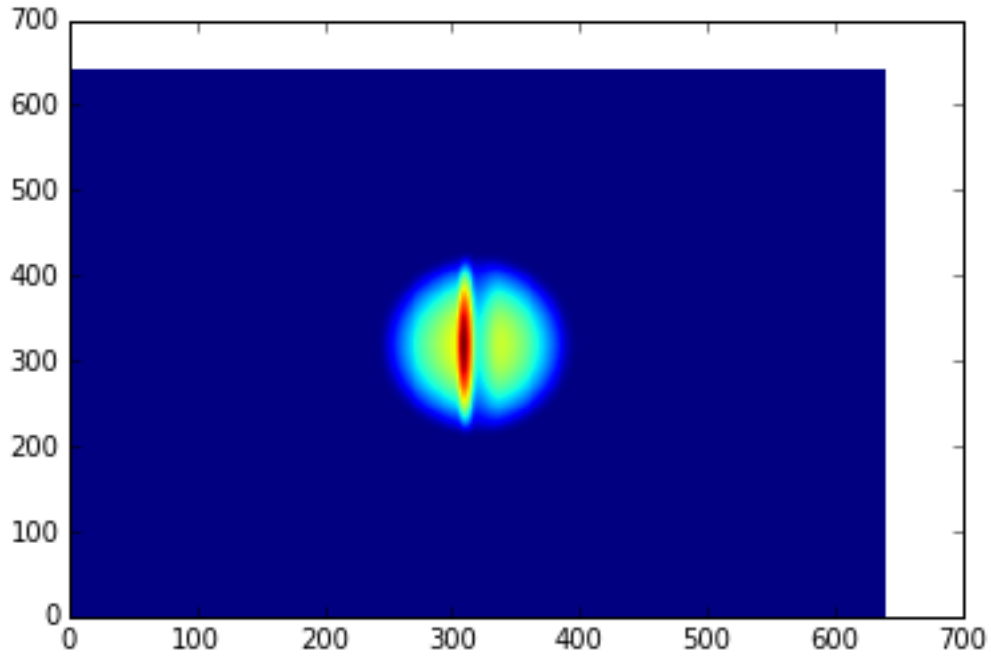
```
        y = (_y - dim*0.5) * delta_y
        w_x = 1
        w_y = 1 / np.sqrt(2)
        return 0.5 * (w_x*w_x * x*x + w_y*w_y * y*y)

    potential = np.zeros((dim, dim))
    for y in range(0, dim):
        for x in range(0, dim):
            potential[y, x] = ext_pot(x, y)
    return potential

# lattice parameters
dim = 640                    # number of grid points at the edge
length = 50.                         # physics length of the lattice
delta_x = length / dim
delta_y = length / dim

# Hamiltonian parameter
particle_mass = 1
scattering_lenght_2D = 5.662739242e-5
num_particles = 1700000
coupling_const = 4. * np.pi * scattering_lenght_2D * num_particles

external_potential = get_external_potential(dim)

###################################
# Ground state approximation
###################################

# initial state
p_real = np.ones((dim, dim))
p_imag = np.zeros((dim, dim))
for y in range(dim):
    for x in range(dim):
        p_real[y, x] = 1./length

Norm2 = ts.calculate_norm2(p_real, p_imag, delta_x, delta_y)
print(Norm2)

# evolution variables
iterations = 18000
delta_t = 1.e-4

# launch evolution
ts.evolve(p_real, p_imag, particle_mass, external_potential, delta_x, delta_y,
          delta_t, iterations, coupling_const=coupling_const, imag_time=True)

Norm2 = ts.calculate_norm2(p_real, p_imag, delta_x, delta_y)
print(Norm2)

heatmap = plt.pcolor(p_real)
plt.show()

###################################
# Phase imprinting
###################################

a = 1.98128
```

```
theta = 1.5 * np.pi

for y in range(dim):
    for x in range(dim):
        tmp_real = np.cos(theta * 0.5 * (1.+np.tanh(-a * (x-dim/2.)*delta_x)))
        tmp_imag = np.sin(theta * 0.5 * (1.+np.tanh(-a * (x-dim/2.)*delta_x)))
        tmp = p_real[y, x]
        p_real[y, x] = tmp_real * tmp - tmp_imag * p_imag[y, x]
        p_imag[y, x] = tmp_real * p_imag[y, x] + tmp_imag * tmp

np.savetxt('InistatePhaseImprinted_real.dat', p_real, delimiter=' ')
np.savetxt('InistatePhaseImprinted_imag.dat', p_imag, delimiter=' ')

heatmap = plt.pcolor(p_real)
plt.show()

###################################
# Real time evolution
###################################

# evolution variables
iterations = 2000
delta_t = 5.e-5
kernel_type = 0

# launch evolution
ts.evolve(p_real, p_imag, particle_mass, external_potential,
          delta_x, delta_y, delta_t, iterations, coupling_const=coupling_const)

# calculate particle density
norm_2 = np.ones((dim, dim))
for y in range(dim):
    for x in range(dim):
        norm_2[y, x] = (p_real[y, x] * p_real[y, x] +
                        p_imag[y, x] * p_imag[y, x]) * num_particles

heatmap = plt.pcolor(norm_2)
plt.show()
```

The results are the following plots:


700
600
500
400
300
200
100
0
0
100
200
300
400
500
600
700


700
600
500
400
300
200
100
0
0
100
200
300
400
500
600
700


700
600
500
400
300
200
100
0
0
100
200
300
400
500
600
700

CHAPTER

FOUR

# FUNCTION REFERENCE

## 4.1 Lattice Class

**class** trottersuzuki.**Lattice**(*dim=100*, *length_x=20.0*, *length_y=20.0*, *periodic_x_axis=False*, *periodic_y_axis=False*, *angular_velocity=0.0*)

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

As to single-process execution, the lattice is a single tile which can be surrounded by a halo, in the case of periodic boundary conditions.

- *Lattice(dim=100, length_x=20.0, length_y=20.0, periodic_x_axis=False, periodic_y_axis=False, angular_velocity=0.0)*

  Construct the Lattice.

  ***–dim* :** Linear dimension of the squared lattice.

  ***–length_x* :** Physical length of the lattice's side along the x axis.

  ***–length_y* :** Physical length of the lattice's side along the y axis.

  ***–periodic_x_axis* :** Boundary condition along the x axis (false=closed, true=periodic).

  ***–periodic_y_axis* :** Boundary condition along the y axis (false=closed, true=periodic).

  ***–angular_velocity* :** Angular velocity of the frame of reference.

C++ includes: trottersuzuki.h

## 4.2 State Classes

**class** trottersuzuki.**State**(**args*)

This class defines the quantum state.

C++ includes: trottersuzuki.h

**get_mean_px**()

Return the expected value of the P_x operator.

**get_mean_pxpx**()

Return the expected value of the P_x^2 operator.

**get_mean_py**()

Return the expected value of the P_y operator.

**get_mean_pypy**()

Return the expected value of the P_y^2 operator.

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

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

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

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

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

**get_phase**()
Return a matrix storing the phase of the wave function.

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

**write_particle_density**(*fileprefix*)
Write to a file the squared norm of the wave function.

**write_phase**(*fileprefix*)
Write to a file the phase of the wave function.

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

**class** trottersuzuki.**ExponentialState**(*_grid*, *_n_x=1*, *_n_y=1*, *_norm=1*, *_phase=0*, *_p_real=None*, *_p_imag=None*)
This class defines a quantum state with exponential like wave function.

This class is a child of State class.

•*ExponentialState(_grid, _n_x=1, _n_y=1, _norm=1, _phase=0, _p_real=None, _p_imag=None)*

Construct the Lattice.

Construct the quantum state with exponential like wave function.

***–grid*** **:** Lattice object.

***–n_x*** **:** First quantum number.

***–n_y*** **:** Second quantum number.

***–norm*** **:** Squared norm of the quantum state.

***–phase*** **:** Relative phase of the wave function.

***–p_real*** **:** Pointer to the real part of the wave function.

***–p_imag*** **:** Pointer to the imaginary part of the wave function.

C++ includes: trottersuzuki.h

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

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

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

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

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

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

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

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

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

**get_phase**()
Return a matrix storing the phase of the wave function.

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

**write_particle_density**(*fileprefix*)
Write to a file the squared norm of the wave function.

**write_phase**(*fileprefix*)
Write to a file the phase of the wave function.

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

**class** trottersuzuki.**GaussianState**(*_grid*, *_omega*, *_mean_x=0*, *_mean_y=0*, *_norm=1*, *_phase=0*, *_p_real=None*, *_p_imag=None*)
This class defines a quantum state with gaussian like wave function.

This class is a child of State class.

- *GaussianState(_grid, _omega, _mean_x=0, _mean_y=0, _norm=1, _phase=0, _p_real=None, _p_imag=None)*

  Construct the quantum state with gaussian like wave function.

  ***–grid*** **:** Lattice object.

  ***–omega*** **:** Gaussian coefficient.

  ***–mean_x*** **:** X coordinate of the gaussian function's center.

  ***–mean_y*** **:** Y coordinate of the gaussian function's center.

  ***–norm*** **:** Squared norm of the state.

  ***–phase*** **:** Relative phase of the wave function.

  ***–p_real*** **:** Pointer to the real part of the wave function.

  ***–p_imag*** **:** Pointer to the imaginary part of the wave function.

C++ includes: trottersuzuki.h

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

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

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

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

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

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

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

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

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

**get_phase**()
Return a matrix storing the phase of the wave function.

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

**write_particle_density**(*fileprefix*)
Write to a file the squared norm of the wave function.

**write_phase**(*fileprefix*)
Write to a file the phase of the wave function.

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

**class** trottersuzuki.**SinusoidState**(*_grid*, *_n_x=1*, *_n_y=1*, *_norm=1*, *_phase=0*, *_p_real=None*, *_p_imag=None*)
This class defines a quantum state with sinusoidal like wave function.

This class is a child of State class.

C++ includes: trottersuzuki.h

•*SinusoidState(_grid, _n_x=1, _n_y=1, _norm=1, _phase=0, _p_real=None, _p_imag=None)*

Construct the quantum state with sinusoidal like wave function.

***–grid*** **:** Lattice object.

***–n_x*** **:** First quantum number.

***–n_y*** **:** Second quantum number.

***–norm*** **:** Squared norm of the quantum state.

***–phase*** **:** Relative phase of the wave function.

***–p_real*** **:** Pointer to the real part of the wave function.

***–p_imag*** **:** Pointer to the imaginary part of the wave function.

C++ includes: trottersuzuki.h

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

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

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

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

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

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

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

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

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

**get_phase**()
Return a matrix storing the phase of the wave function.

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

**write_particle_density**(*fileprefix*)
Write to a file the squared norm of the wave function.

**write_phase**(*fileprefix*)
Write to a file the phase of the wave function.

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

## 4.3 Potential Classes

**class** trottersuzuki.**Potential**(**args*)
This class defines the external potential that is used for Hamiltonian class.

C++ includes: trottersuzuki.h

- *Potential(*args)*

  Construct the external potential.

  ***–grid*** **:** Lattice object.

  ***–filename*** **:** Name of the file that stores the external potential matrix.

C++ includes: trottersuzuki.h

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

**class** trottersuzuki.**HarmonicPotential**(*_grid*, *_omegax*, *_omegay*, *_mass=1.0*, *_mean_x=0.0*, *_mean_y=0.0*)
*HarmonicPotential(grid, omegax, omegay, mass=1., mean_x=0., mean_y=0.)*

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

This class is a child of Potential class.

•*HarmonicPotential(grid, omegax, omegay, mass=1., mean_x=0., mean_y=0.)*

Construct the harmonic external potential.

Parameters: * *grid* :

Lattice object.

– ***omegax*** **:** Frequency along x axis.

– ***omegay*** **:** Frequency along y axis.

– ***mass*** **:** Mass of the particle.

– ***mean_x*** **:** Minimum of the potential along x axis.

– ***mean_y*** **:** Minimum of the potential along y axis.

C++ includes: trottersuzuki.h

**get_value** (*x*, *y*)
Return the value of the external potential at coordinate (x,y)

## 4.4 Hamiltonian Classes

**class** trottersuzuki.**Hamiltonian**(*_grid*, *_potential=None*, *_mass=1.0*, *_coupling_a=0.0*, *_angular_velocity=0.0*, *_rot_coord_x=0*, *_rot_coord_y=0*)

‘**Hamiltonian(grid, potential=0, mass=1., coupling_a=0., angular_velocity=0.,** rot_coord_x=0, rot_coord_y=0)‘

This class defines the Hamiltonian of a single component system.

•‘**Hamiltonian(grid, potential=0, mass=1., coupling_a=0., angular_velocity=0.,** rot_coord_x=0, rot_coord_y=0)‘

Construct the Hamiltonian of a single component system.

Parameters: * *grid* :

Lattice object.

– ***potential*** **:** Potential object.

– ***mass*** **:** Mass of the particle.

– ***coupling_a*** **:** Coupling constant of intra-particle interaction.

– ***angular_velocity*** **:** The frame of reference rotates with this angular velocity.

– ***rot_coord_x*** **:** X coordinate of the center of rotation.

– ***rot_coord_y*** **:** Y coordinate of the center of rotation.

C++ includes: trottersuzuki.h

**class** trottersuzuki.**Hamiltonian2Component**(*_grid*, *_potential=None*, *_potential_b=None*, *_mass=1.0*, *_mass_b=1.0*, *_coupling_a=0.0*, *_coupling_ab=0.0*, *_coupling_b=0.0*, *_omega_r=0*, *_omega_i=0*, *_angular_velocity=0.0*, *_rot_coord_x=0*, *_rot_coord_y=0*)

‘**Hamiltonian2Component(grid, potential=0, potential_b=0, mass=1., mass_b=1.,** coupling_a=0., coupling_ab=0., coupling_b=0., omega_r=0, omega_i=0, angular_velocity=0., rot_coord_x=0, rot_coord_y=0)‘

This class defines the Hamiltonian of a two component system.

- ‘**Hamiltonian2Component(grid, potential=0, potential_b=0, mass=1., mass_b=1.,** coupling_a=0., coupling_ab=0., coupling_b=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.

  - ***potential*** **:** Potential of the first component.
  - ***potential_b*** **:** Potential of the second component.
  - ***mass*** **:** Mass of the first-component’s particles.
  - ***mass_b*** **:** Mass of the second-component’s particles.
  - ***coupling_a*** **:** Coupling constant of intra-particle interaction for the first component.
  - ***coupling_ab*** **:** Coupling constant of inter-particle interaction between the two components.
  - ***coupling_b*** **:** Coupling constant of intra-particle interaction for the second component.
  - ***omega_r*** **:** Real part of the Rabi coupling.
  - ***omega_i*** **:** Imaginary part of the Rabi coupling.
  - ***angular_velocity*** **:** The frame of reference rotates with this angular velocity.
  - ***rot_coord_x*** **:** X coordinate of the center of rotation.
  - ***rot_coord_y*** **:** Y coordinate of the center of rotation.

C++ includes: trottersuzuki.h

## 4.5 Solver Class

**class** trottersuzuki.**Solver**(**args*)

*Solver(grid, state, hamiltonian, delta_t, kernel_type=”cpu”) Solver(grid, state1, state2, hamiltonian, delta_t, kernel_type=”cpu”)*

This class defines the evolution tasks.

- *Solver(grid, state, hamiltonian, delta_t, kernel_type=”cpu”)*

  Construct the Solver object for a single-component system.

  Parameters: * *grid* :

  Lattice object.

***–state*** **:** State of the system.

***–hamiltonian*** **:** Hamiltonian of the system.

***–delta_t*** **:** A single evolution iteration, evolves the state for this time.

***–kernel_type*** **:** Which kernel to use (either cpu or gpu).

Massively Parallel Trotter-Suzuki Solver

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

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

Construct the Solver object for a two-component system.

Parameters: * *grid* :

Lattice object.

***–state1*** **:** First component's state of the system.

***–state2*** **:** Second component's state of the system.

***–hamiltonian*** **:** Hamiltonian of the two-component system.

***–delta_t*** **:** A single evolution iteration, evolves the state for this time.

***–kernel_type*** **:** Which kernel to use (either cpu or gpu).

C++ includes: trottersuzuki.h

**evolve**(*iterations*, *imag_time=False*)
Evolve the state of the system.

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

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

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

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

**get_rabi_energy**()
Get the Rabi energy of the system.

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

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

**get_total_energy**()
Get the total energy of the system.

## E



## G



## H