Surface Frog - A Surface Hopping Program

User Manual

Kush Patel

April 2020

Contents

Chapter 1

Introduction

This program is a Python (3.7) implementation of Tully's Fewest Switches Surface Hopping algorithm[1]. In addition, it includes experimental algorithms currently under development by Bittner *et al*. Users can easily include their own surfaces. Currently, the code only works for one-dimensional two-level systems. Additionally, the code is not tested in MacOS or Windows operating systems.

Chapter 2

Setup

Surface Frog is designed to run immediately without the need for an installation. However, the following dependencies must be included in the user's Python environment:

- Json
- Numpy
- Scipy
- Multiprocessing

Chapter 3

Main Files

This chapter lists and describes the function of the main files included in the package.

`__main__.py`

This script handles input parameter files as well as the individual trajectories. Parallelization happens here. Output information of all trajectories will be dumped into a corresponding file in `[./Output/]`.

`engine.py`

This script handles all of the physics contained within a single trajectory.

`models/`

This directory contains all surface models users may wish to simulate. Included models reflect those described in Ref 1.

- `SimpleAvoidedCrossing.py`
- `DualAvoidedCrossing.py`
- `ExtendedCouplingWithReflection.py`

Chapter 4

Input Parameters

Surface Frog takes `json` files as input. This section lists the currently changeable parameters.

`collapseQ`

Type: Boolean
Default: false
Indicates whether to collapse the state whenever a surface hop occurs. If set to True, then on hop to state k, the density matrix will be set to $\rho = 0.0, \rho_{k,k} = 1.0$.

`dt`

Type: Float
Default: 20.0
Units: $\hbar/E_h$ (atomic units)
Time step in atomic units.

`force_style`

Type: String
Default: 'state'
Options: 'state','meanfield'
Meanfield option will compute forces as a mix between surface potentials using state populations (density matrix diagonal components) as the weights. State option will always use the surface potential corresponding to the system's current state.

hop_style

Type: String
Default: 'Tully'
Options: 'Tully', 'Conditioned'
Defines which method is used to determine when a hop is to occur. 'Tully' uses the method described in Ref. 1.

logQ

Type: Boolean
Default: false
Set this variable as True to enable trajectory logging. If set to True, detailed trajectory information will be dumped into a corresponding file in `[./LogFiles/]`.

model

Type: String
Default: SimpleAvoidedCrossing
Defines which surface from the directory `[./models/]` will be simulated. See [later chapter] for implementing new surfaces.

nprocs

Type: Integer
Default: 1
Number of CPU processors to include for parallel processing. If only one CPU is requested, parallel processing is skipped and simulations run in serial.

nsims

Type: Integer
Default: 1
Number of trajectories to simulate.

propagator

Type: String
Default: 'standard'
Options: 'standard', 'LvN'
Defines which method is used to propagate the electronic wavefunction. Currently 'LvN' is not working correctly.

p0

Type: Float
Default: 5.0
Units: $\hbar/a_0$ (atomic units)
Initial momentum.

sg_p

Type: Float
Default: 0.0
Units: $\hbar/a_0$ (atomic units)
Normal distribution width (σ) for initial momenta. For parameter sets requesting multiple trajectories, initial momenta can be distributed around `[p0]` and are computed via normal distribution:

$$f(p) = \frac{1}{\sigma\sqrt{2\pi}} Exp\left[-\frac{(p-p0)^2}{2\sigma^2}\right]$$

Default value of `sg_p=0.0` implies no distribution (all trajectories will start with exactly `[p0]` momenta).

sg_x

Type: Float
Default: 0.0
Units: a_0 (atomic units)
Normal distribution width (σ) for initial positions. For parameter sets requesting multiple trajectories, initial positions can be distributed around `[x0]` and are computed via normal distribution:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} Exp\left[-\frac{(x-x0)^2}{2\sigma^2}\right]$$

Default value of `sg_p=0.0` implies no distribution (all trajectories will start at exactly `[x0]` position).

state0

Type: Integer
Default: 0
Indicates which state/surface the trajectory starts on. 0 implies lowest (ground) state.

tag

Type: String
Default: 'sim'
Short keyword to distinguish clusters of simulations from one another. `tag` is include as part of the output file names.

Tmax

Type: Float
Default: 20000.0
Units: $\hbar/E_h$ (atomic units)
Maximum simulation time. Note, if the code find the particle in an asymptotic position ($abs(x) > 15$) with momentum towards the asymptotic direction the simulation will terminate early.

x0

Type: Float
Default: -5.0
Units: a_0 (atomic units)
Initial position.

Chapter 5

Additional Models

To include additional models, simply create a corresponding python script file and add it into the `[./models/]` directory. A few basic requirements must be followed. Ensure the script's file name coincides with the python class name. Additionally define both the surface potential `V(x)` and the derivative `dV(x)`. Ensure the script imports any necessary packages, especially `numpy`.

Below is an example model with the file named as `MyModel.py`.

```
import numpy as np
from scipy.special import erf

class MyModel:
        def __init__(self, a = 0.25, b = 0.5, mass = 2000.0):
                # Initialize model's internal parameters
                self.a = a
                self.b = b
                self.mass = mass
                self.name = 'ExampleModel'

        # Diabatic Surfaces and Couplings V(x)
        def V(self, x):
                V11 = erf( self.a*x )
                V22 = -V11
                V12 = np.cosh(self.b*x) ** -2
                return np.array([ [V11,V12],[V12,V22] ])

        # Derivatives, dV(x)/dx
        def dV(self,x):
                dV11 = 2*self.a / np.sqrt(np.pi) * np.exp( -(self.a*x)**2 )
```

```
        dV22 = -dV11
        dV12 = -2*self.b*np.tanh(self.b*x)/np.cosh(self.b*x)**2
        return np.array([ [dV11,dV12],[dV12,dV22] ])
```

Chapter 6

My First Trajectory

Parameter File

Here is an example JSON file that illustrates a valid parameter file.

```
{
    "nsims":    64,
    "nprocs":   8,
    "x0":       -5.0,
    "p0":       9.5,
    "state0":   0,
    "dt":       20.0,
    "Tmax":     20000.0,
    "logQ":     true,
    "model":    "SimpleAvoidedCrossing",
    "tag":      "SAC64x8"
}
```

With this parameter set, which in this example we shall save as `[example_params.json]`, we perform 64 individual trajectories distributed over 8 CPU cores. Trajectories begin at $x = -5.0$ with momentum $p = 9.5$ in the ground state. Time step and max time are set to 20.0 and 2000.0 respectively. We have enabled logging to get more detailed output of each trajectory. The model is the Simple Avoided Crossing found in Ref. 1.

Executing a Job

To begin the job, simply run the python script feeding the parameter file as a command line argument.

```
$ python3 path/to/__main.__py path/to/example_params.json
```

Output

Executing the above will generate a file named `[SimpleAvoidedCrossing.SAC64x8]` in the `[Output/]` directory. Below is a sample output summarizing the results. (Your exact results my differ; it is, after all, a stochastic algorithm.)

```
 64 simulations over 8 processes finished in 1.3200798034667969.
State 0 - Reflection: 0
State 0 - Transmission: 63
State 1 - Reflection: 0
State 1 - Transmission: 1
```

With the given parameters, 0 trajectories reflected, 63 trajectories transmitted in the ground state, and 1 trajectory transmitted in the first excited state.

Finally, peeking into the `[LogFiles/]` there are now 64 new files name with the formatting `[model.tag.proc_id]`. Each of these will provide the time series data of their corresponding trajectory.

Bibliography

[1] John C. Tully. Molecular Dynamics with Electronic Transitions. *The Journal of Chemical Physics*, 93(2):1061–1071, 1990.