pySpawn User Manual

Version 0.0

Copyright © 2017 Benjamin G. Levine

Contents

[2 Introduction 3](#_Toc498369145)

[2.1 How to Use This Manual 3](#_Toc498369146)

[2.2 What Is PySpawn? 3](#_Toc498369147)

[2.3 What Are The Primary Goals of the PySpawn Project? 3](#_Toc498369148)

[2.4 Features 3](#_Toc498369149)

[2.5 Citing pySpawn 3](#_Toc498369150)

[2.6 License 4](#_Toc498369151)

[2.7 Acknowledgements 4](#_Toc498369152)

[3 Obtaining and Installing PySpawn 5](#_Toc498369153)

[3.1 Obtaining PySpawn 5](#_Toc498369154)

[3.2 Requirements and Dependencies 5](#_Toc498369155)

[3.3 Installing PySpawn 5](#_Toc498369156)

[4 Abbreviations Used in this Manual 6](#_Toc498369157)

[5 Guided Tour of Your First PySpawn Simulation 7](#_Toc498369158)

[5.1 Building Hessian File for Initial Condition Sampling 7](#_Toc498369159)

[5.2 AIMS Simulation Script 7](#_Toc498369160)

[5.3 Output 10](#_Toc498369161)

[5.4 Analysing Simulation Data with the Analysis Module 11](#_Toc498369162)

[6 PySpawn Algorithm 14](#_Toc498369163)

# Introduction

## How to Use This Manual

Unlike electronic structure calculations, ab initio multiple spawning calculations produce a larger amount of data than can be read by a human in a short period of time. That data has a much more complex structure than the data created by classical molecular dynamics simulations. The PySpawn algorithm is complex and may be unintuitive to both those who have never run AIMS before and those who are familiar with the older FMS90 code. As such, before starting to run PySpawn, it is recommended to carefully read chapters 2-6 of this manual. These provide a full introduction to the principle one needs to understand to set up and run a PySpawn simulation. Later chapters provide more detailed reference for input, etc.

## What Is PySpawn?

PySpawn is an implementation of ab initio multiple spawning (AIMS) in Python 2.7.

## What Are The Primary Goals of the PySpawn Project?

PySpawn is designed with the following goals in mind:

* The core of pySpawn is designed to be small and thus easy to maintain.
* PySpawn is designed to be extensible. At present, the following functionality can, in principle, be implemented without modification of existing code:
  + Interfaces to additional electronic structure codes/methods
  + Quantum mechanical Hamiltonians (e.g. adiabatic representation vs. diabatics vs. diabatized Gaussians on adiabatic surfaces, interpolated vs. analytical derivative couplings, spin orbit Hamiltonian, explicit light field)
  + Quantum mechanical integrators
  + Classical mechanical integrators
* PySpawn is designed with large, shared computer resources in mind. Specifically, pySpawn is designed to facilitate
  + Subdivision of large jobs into small chunks
  + Restart
  + Parallelization

## Features

* Runs FMS/AIMS jobs in the adiabatic representation
* Derivative couplings computed by norm preserving interpolation (NPI)
* PySpawn interfaces to a development version of TeraChem via a protobuf interface

## Citing pySpawn

If you use the PySpawn software package please cite the following:

“pySpawn is a nonadiabatic molecular dynamics software package written by Benjamin G. Levine”

This citation will eventually be replaced by a true publication.

Note that this is a citation for the PySpawn software package, not any particular method. In addition to citing this package, please be sure to cite papers corresponding to the simulation methods that you used:

AIMS (cite these for all PySpawn simulations):

* M. Ben-Nun, J. Quenneville, and T. J. Martínez, J. Phys. Chem. A, **104**, 5161−5175 (2000).
* M. Ben-Nun and T. J. Martínez, Adv. Chem. Phys., **121**, 439−512 (2002).

Diabatized Gaussians on adiabatic surfaces (DGAS):

* G. A. Meek and B. G. Levine, J. Chem. Phys., **145**, 184103 (2016).

Norm preserving interpolation (NPI; this is the default in both adiabatic and DGAS simulations):

* G. A. Meek and B. G. Levine, J. Phys. Chem. Lett., **5**, 2351 (2014).
* G. A. Meek and B. G. Levine, Chem. Phys., **460**, 117 (2015).

Additional citations are likely necessary for the electronic structure methods and software used.

## License

PySpawn is distributed under the MIT License:

Copyright (c) 2017 Benjamin G. Levine

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## Acknowledgements

* The protobuf interface was developed by Stefan Seritan in the group of Todd J. Martínez at Stanford University.
* Many thanks to B. Scott Fales, Todd J. Martínez, and Stefan Seritan for their contributions and input.
* We gratefully acknowledge funding from the National Science Foundation under grant CHE-1565634.

# Obtaining and Installing PySpawn

## Obtaining PySpawn

PySpawn can be obtained from GitHub: <https://github.com/blevine37/pySpawn17>.

## Requirements and Dependencies

PySpawn is written in Python 2.7 and is not currently compatible with Python 3.

PySpawn requires the following dependencies:

* numpy
* h5py
* tcpb (TeraChem protobuf interface; required to run with TeraChem only)
* google (required by tcpb for TeraChem interface only)
* protobuf (required by tcpb for TeraChem interface only)

These can easily be installed using pip:

pip install numpy h5py google protobuf

## Installing PySpawn

Having obtained pySpawn and installed the necessary dependencies, cd to the top directory and install it by running

python setup.py install

# Abbreviations Used in this Manual

Various abbreviations will be used in this manual:

AIMS ab initio multiple spawning

a.u. atomic units

CAS complete active space

CASCI complete active space configuration interaction

CASSCF complete active space self-consistent field

CISNO configuration interaction singles natural orbital

FMS full multiple spawning

FOMO floating occupation molecular orbital

NPI norm-preserving interpolation

PES potential energy surface

SA state averaged

TBF trajectory basis function

# Guided Tour of Your First PySpawn Simulation

This chapter will explain the example calculation in examples/ethylene\_sacasscf. This job simulates the dynamics of ethene on the SA2-CAS(2/2)/6-31G potential energy surface. A single initial nuclear basis function is initiated, with its initial position and momentum sampled from the ground state vibrational Wigner distribution computed in the harmonic approximation. This section will not detail all the inner workings of the code, but instead will highlight

* the series of steps needed to run a simulation with pySpawn
* the input scripts
* the output structure

**Most of what you need to know to start using pySpawn to run simulations is in this chapter, so it will be in your best interest to read it carefully!** A more detailed description of input parameters, the analysis module, and simulation algorithm will be presented in subsequent chapters.

## Building Hessian File for Initial Condition Sampling

The first step in any AIMS simulation is choosing the initial conditions. One popular choice, used here, is to sample the average position and momentum of the initial TBF from the vibrational Wigner distribution, computed in the harmonic approximation. This requires three pieces of information:

* the ground state minimum structure
* the Hessian (2nd derivative) matrix of PES at the ground state minimum
* the masses of the atoms

Before beginning, one must compute the minimum and Hessian and store them in pySpawn’s preferred format: hessian.hdf5. We will assume that the reader can optimize the ground state minimum energy geometry of their chosen molecule. The script build\_hessian.py takes this structure, drives TeraChem to compute the Hessian matrix, and then stores both the geometry and Hessian in hessian.hdf5.

The script is self-explanatory. There are a few important points to keep in mind:

* **Most importantly, if you wish to recompute the Hessian you must remove any existing** hessian.hdf5 **files!**
* It is your responsibility to make sure that the structure that is inputted is the desired ground state minimum structure.
* The final hessian.hdf5 file contains both the geometry and Hessian matrix. When sampling the initial conditions, both the geometry and Hessian will be drawn from this file.
* If the job should die before the hessian is completed, it can be restarted trivially by running this same script. The restarted job will read the existing hessian.hdf5 file and pick up where it left off.

## AIMS Simulation Script

With hessian.hdf5 in hand, we are ready to run an AIMS simulation of ethene. The Python script that does this is start\_c2h4.py. This first section of this script defines several important parameters of the simulation:

* seed is the random number seed (for Wigner initial conditions).
* clas\_prop defines the classical propagator. vv opts for velocity Verlet integration.
* qm\_prop defines the propagator of the quantum amplitudes (the expansion coefficients in the AIMS wave functions). rk2 opts for adaptive second-order Runge-Kutta.
* qm\_ham defines the Hamiltonian used to propagate these amplitudes. adiabatic opts for AIMS in the adiabatic basis with NPI time-derivative couplings.
* potential defines the means by which the electronic structure and PES will be computed. terachem\_cas opts for on-the-fly calculation at a CAS level of theory using TeraChem. This option will work for SA-CASSCF, FOMO-CASCI, or CISNO-CASCI.
* t0 is the initial time of the simulation (in a.u.).
* ts is the time step of the simulation (in a.u.).
* tfinal is the final time of the simulation (in a.u.).
* numdims is dimensionality of the system (3 \* number of atoms).
* numstates is the number of electronic states in the calculation.

The definition of these scalar and string variables is followed by the creation of three dictionary objects:

* tc\_options contains options to be passed to TeraChem. Most of these options are documented in the TeraChem documentation, and thus are not discussed here. Exceptions:
  + atoms is a list of the abbreviations for the atoms in the molecule, in order.
  + cas\_energy\_labels is a list of tuples defining the states whose energies will be passed from TeraChem to pySpawn. The first element of each tuple is the state index (ground state is 0, first excited state is 1…) while the second element is the spin multiplicity (singlet is 1…)
* traj\_params contains parameters of the individual TBFs:
  + time is the initial time of the simulation.
  + timestep defines the time step for classical propagation.
  + maxtime defines the maximum time to which each TBF will be propagated.
  + spawnthresh defines the coupling above which spawning will be triggered.
  + istate is the index of the electronic state to which this TBF belongs (0 is the ground states…)
  + widths is a numpy array containing the widths, α, associated with each degree of freedom (in units of bohr-2).
  + atoms contains the list of atom labels (and should be identical to atoms in tc\_options in all normal cases).
  + masses is a numpy array containing the masses associated with each degree of freedom (in a.u., i.e. electron masses).
  + tc\_options is the dictionary structure defined above which contains options to be passed to TeraChem.
* sim\_params contains parameters of the entire simulation, including the quantum propagation:
  + quantum\_time is the initial time of the simulation (and should be identical to time in the traj\_params object in all normal cases).
  + timestep is the time step for quantum propagation (and should be identical to timestep in the traj\_params object in all normal cases).
  + max\_quantum\_time is the simulation time at which quantum propagation should stop (and should be identical to maxtime in the traj\_params object in all normal cases).
  + qm\_amplitudes is the initial quantum amplitudes.
  + qm\_energy\_shift is an energy shift applied to the diagonal of the Hamiltonian. For numerical convenience, these diagonals should be roughly zero.

Other options for traj\_params and sim\_params are documented in a subsequent chapter.

To this point the script is a simple Python script, defining various variables and data structures that will be passed to pySpawn. Now, we begin by importing methods into the simulation and traj classes that will enable propagation. An instance of the simulation class will contain information pertaining to the entire simulation, including the quantum propagation. An instance of the traj class will describe each TBF. (For those familiar with FMS90, the simulation class is similar in spirit to the “bundle.”)

In order to propagate, two sets of methods must be imported into the simulation class to enable propagation: one for quantum integration and one for the quantum Hamiltonian. We have already specified the flavor of integrator (qm\_prop = “rk2”) and Hamiltonian (qm\_ham = “adiabatic”) that we wish to use above. These two lines simply import the necessary methods:

exec("pyspawn.import\_methods.into\_simulation(pyspawn.qm\_integrator." + qm\_prop + ")")

exec("pyspawn.import\_methods.into\_simulation(pyspawn.qm\_hamiltonian." + qm\_ham + ")")

Similarly, two sets of methods must be imported into the traj class to enable propagation: one to choose the potential and one to choose the classical integrator. Again, we have already specified the flavors we wish above (potential = “terachem\_cas”; clas\_prop = “vv”). These two lines simply import the necessary methods into the traj class:

exec("pyspawn.import\_methods.into\_traj(pyspawn.potential." + potential + ")")

exec("pyspawn.import\_methods.into\_traj(pyspawn.classical\_integrator." + clas\_prop + ")")

Before preceding further, pyspawn.general.check\_files() checks to make sure that there is not any pySpawn hdf5 output in the current directory. The job will fail here if there is. **By default, pySpawn will append to an existing hdf5 output file, not overwrite it, so it is important to delete unrelated output before running a new job. Similarly, every pySpawn job must run in its own directory.**

Now, the following lines instantiate a single traj object and set its various attributes:

traj1 = pyspawn.traj()

traj1.set\_numstates(numstates)

traj1.set\_numdims(numdims)

traj1.set\_parameters(traj\_params)

It’s important that numstates and numdims be set **before** setting the remaining parameters, because the size of various attributes of the traj object depend on these attributes.

Next we set the initial positions and momenta by sampling from the Wigner distributions:

traj1.initial\_wigner(seed)

This routine will read the geometry and Hessian matrix from hessian.hdf5 (created above). The masses were included in traj\_params, and thus have been set. The masses are required in order to construct the Wigner distribution, so it is important that they be set the masses before calling initial\_wigner.

Now we instantiate the simulation object, add the traj object that we just created to it, and set the various simulation parameters:

sim = pyspawn.simulation()

sim.add\_traj(traj1)

sim.set\_parameters(sim\_params)

Finally, it is time to propagate the AIMS wave function:

sim.propagate()

To run the script at the linux command prompt, use something like:

python start\_c2h4.py >pyspawn.log 2>&1 &

And we are off to the races!

## Output

PySpawn creates three types of output during a dynamics run: text output (standard output), HDF5 output, and json output, as described in this section.

The text output is produced in the standard output. (If you run the command in the previous section, it will be recorded in pyspawn.log.) The purpose of the text output is to track the progress of the simulation. Each significant step in the calculation results in a line being printed to this output. It is useful for tracking the progress of a job and for debugging. The text output does not contain the quantitative output of the simulation.

The quantitative output of the simulation is recorded in the HDF5 output. Several HDF5 output files are produced. working.hdf5 is the working copy which is being modified on the fly by the simulation. sim.hdf5 is a copy that is synchronized with the most recent json output (described below) for restart purposes. sim.hdf5 should be viewed as the quantitative output of the simulation. It contains all of the time-dependent information needed to reconstruct the time-dependent AIMS wave function, e.g. classical positions and momenta of each TBF at each time step, quantum amplitudes as a function of time, electronic structure information (e.g. CI vectors and orbitals) as a function of time, derivative couplings as a function of time, etc.

HDF5 is a binary format and thus is not directly human readable. One can easily glimpse the contents using h5dump or any HDF5 viewer. However, the pySpawn analysis module (described below) is designed to process sim.hdf5 into human readable format, so that direct viewing of the HDF5 files should not be necessary in most cases. If desired functionality is missing, it should be relatively easy to implement (and contribute) it within the framework of the analysis module.

The final output file is sim.json, which contains a complete snap shot of the current simulation object. It primarily exists for restart purposes, but can be useful to troubleshooting/debugging as well. Unlike sim.hdf5, sim.json does not contain any data prior to the current state of the system, and thus cannot be used for analysis of time-dependent AIMS wave function.

sim.json and sim.hdf5 are synchronized, and are both required for restart. Each time a new pair of files is stored, the previous pair is renamed (e.g. sim.1.json and sim.1.hdf5, etc.) so that at any given time several synchronized pairs are available for restart, in the event that a file should be corrupted or accidently deleted. working.hdf5 is not synchronized with a json file, and thus cannot be used for restart.

## Analysing Simulation Data with the Analysis Module

The sim.hdf5 output file is not human readable. It can be read using standard hdf5 tools, but it is likely more convenient to work with the analysis module built into PySpawn. It includes a set of Python routines to read and process the sim.hdf5 file. It will create numpy arrays containing the data (“datasets”), facilitates various common processing tasks (e.g. computing bond lengths or electronic state populations), and enables this data to be output in columnar form. Used in conjunction with matplotlib, figures can be created directly from sim.hdf5 in a simple python script.

The analysis.py script analyses this simple example simulation, illustrating the most commonly used features of the analysis module. The line

an = pyspawn.fafile("sim.hdf5")

creates a fafile object (short for FMS analysis file object) associated with the sim.hdf5 file. This object contains a dictionary, an.datasets, which contains all of the datasets. Each dataset is a two-dimensional numpy array, where the leading dimension indexes time steps. Upon creation, several datasets are initiated from sim.hdf5. Most notable are datasets containing the time. The quantum\_times dataset contains the times associated with the quantum propagation. A XX\_time dataset (where XX is replaced by a trajectory label, e.g. 00b1) is created for each trajectory as well.

Many methods of the fafile object are titled fill\_something. These objects fill a new dataset with data. For example, the next line

an.fill\_electronic\_state\_populations(column\_filename = "N.dat")

fills a new dataset titled electronic\_state\_populations. The first *Nstates*columns will contain the populations of each electronic state as a function of time. The final column will contain the total population as a function of time. The optional input parameter column\_filename instructs the routine to create a columnar data file containing time (specifically, the quantum\_time dataset) in the first column, and the electronic\_state\_populations dataset in subsequent columns. This is identical to the N.dat file created by the old FMS90 code.

The next several lines demonstrate how to directly create a figure containing the data from this dataset. The next two lines

times = an.datasets["quantum\_times"]

N = an.datasets["electronic\_state\_populations"]

copy the datasets containing the simulation time and the state populations as numpy arrays. The next four lines

plt.plot(times,N[:,0],"ro",times,N[:,1],"bs",markeredgewidth=0.0)

plt.xlabel('Time')

plt.ylabel('Population')

plt.savefig('N.png')

create N.png, a figure showing the ground and first excited state populations as a function of time, using matplotlib. Please consult the matplotlib documentation for details. The commented line

#plt.show()

would display the plot on the screen if uncommented.

The next line

an.write\_xyzs()

creates multi-structure xyz files showing the time evolution of each trajectory, which can be viewed with Molden or VMD, for example.

Methods named fill\_trajectory\_XX will create one dataset per trajectory, containing a given quantity as a function of time. For example, the next line of the analysis script

an.fill\_trajectory\_energies(column\_file\_prefix="E")

create datasets storing the potential energies on each electronic state, classical kinetic energy, and total classical energy for each trajectory. For example, if two trajectories exist in the sim.hdf5 file, 00 and 00b0, six datasets will be created: 00\_poten, 00\_kinen, 00\_toten, 00b0\_poten, 00b0\_kinen, and 00b0\_toten. (Trajectory labels are described in the next chapter.) The optional argument column\_file\_prefix instructs the method to create columnar data files. For example, column\_file\_prefix=”E” will result in the creation of E\_00.dat and E\_00b0.dat. The columns will contain the time (from 00\_time and 00b0\_time, in this case), the potential energies of each electronic state, the classical kinetic energy, and the total classical energy, in that order.

The next line

an.fill\_trajectory\_tdcs(column\_file\_prefix="tdc")

will similarly create datasets XX\_tdc (where XX is a trajectory label) containing the time derivative couplings between the state each trajectory is on and all other electronic states. It will then print them out to columnar data files named tdc\_XX.dat. Note that NPI time derivative couplings are computed at the midpoint of each time step, so the times will be different than for the energies (which are computed on the time steps). The times associate with half time steps are stored on dataset 00\_time\_half\_step.

The next lines compute, store as datasets, and output to columnar data files various geometric parameters as a function of time for each trajectory. For example

bonds = np.array([[0,1],

[1,2],

[1,3],

[0,4],

[0,5]])

an.fill\_trajectory\_bonds(bonds,column\_file\_prefix="bonds")

will create datasets XX\_bonds, where again XX is a trajectory label. Each column of this dataset corresponds to a different interatomic distance, as described in the bonds array. The first column will contain the distance between atom 0 and atom 1, the second between atom 1 and atom 2, the third between atom 1 and atom 3, and so on. These will be outputted to columnar data files just as above.

Similar applies to the remaining lines, which compute angles between triplets of atom, dihedral angles, pyramidalization angles, and twist angles. (Pyramidalization and twist angles are related to dihedral angles, and are defined precisely in the subsequent input reference.)

The final line

an.list\_datasets()

lists all datasets currently in the fafile object.

A useful feature of the analysis module is that it can run on a simulation that is currently running.

# PySpawn Algorithm

The sequence of calculations in a pySpawn simulation is different from that of simulations run with previous AIMS implementations. Past implementations have run simulations as a series of bundle-time-steps, with all classical and quantum variables synchronized at all times. PySpawn divides the simulation into finer-grained units, *tasks*, that may be run asynchronously. This allows finer-grained restart capability and will enable parallelization in future implementations. A single task corresponds roughly to a single electronic structure call. For example, propagating the classical variables of one TBF by one time step is a task. The central object in the pySpawn algorithm is a prioritized queue of tasks to be completed.

Below we describe the serial AIMS algorithm currently implemented in pySpawn in detail. Before describing the algorithm itself, we introduce a few key concepts. This description will assume some knowledge of AIMS/FMS. The reader is referred to the literature to learn about AIMS/FMS if they are unfamiliar with the basics of the AIMS ansatz.

Trajectory basis function (TBF) – The AIMS wave function is represented as a linear combination of TBFs. These TBFs are time dependent, with an average positions and momentum that are defined by the classical equations of motion. In pySpawn, TBFs all have some initial time (0.0 for the initial TBFs, or some later time for spawned TBFs). From its initial time, each TBF will be propagated forward to some maximum time (typically the end of the simulation, though not necessarily). In addition, spawned TBFs will be propagated backwards in time to some minimum time (described below). The classical variable of a TBF depend only on the TBF itself, and thus are propagated independently of the quantum mechanical amplitude of each TBF, which requires knowledge of all TBFs at all prior times. The current forward propagation time of a give TBF will be called its *current time*, while the current backwards propagation time will be called its *backprop time*. Each TBF is represented by a traj object, described above.

Centroids – In order to compute the off-diagonal Hamiltonian matrix elements coupling TBFs, one needs to perform electronic structure calculations at the *centroids* between those TBFs—the maxima of the overlap density between two TBFs. Whenever a new TBF is spawned, new centroids are created with it. These centroids share the initial time of the spawned TBF, and like the spawned TBF they propagate both backwards and forwards in time towards some minimum/maximum time. Like TBFs, centroids have a current time and backprop time. Because the position of a centroid depends on the position of its two associated TBFs at a given point in time, a centroid cannot be propagated to a given time (forward or backward) until the position of both TBFs are known at that time.

With these ideas in mind, we present the pySpawn algorithm. Each iteration of the pySpawn algorithm consists of the following steps, in order:

1. **Update the centroids** – At the beginning of each iteration, each centroid is considered. If both of its associated TBFs have been propagated beyond its current time, it’s position will be updated and it will be marked for subsequent electronic structure calculations. Similarly, if both of its associated TBFs have been propagated at its backprop time, it’s backprop position will be updated and it will be marked for subsequent electronic structure calculations.
2. **Generate the task queue** – A list of tasks that can currently be performed is created. This includes all TBFs that may be back propagated, all TBFs that may be forward propagated, all centroids that may be back propagated, and all centroids that may be forward propagated. This queue is prioritized by the simulation time associated with each task. (e.g. back-propagating a TBF to simulation time 100.0 a.u. takes priority over forward-propagating a TBF to 200.0 a.u.)
3. **Checking whether simulation is complete** – If all trajectories and quantum mechanical amplitudes have been propagated to their maximum and minimum times, the simulation is done and the code will exit.
4. **Check if maximum wall time is reached** – It is possible to set a maximum wall time for a simulation to run. If this time has been surpassed, the simulation will exit gracefully.
5. **Compute a task** – In the current serial implementation, the highest-priority task in the queue will be completed at this point.
6. **Spawn if necessary** – This step actually involves two different processes:
   1. If the derivative coupling between the current state of a TBF and any other state(s) rises above the spawning threshold (spawnthresh) that TBF will be marked as spawning to that state. A spawning TBF continues to propagate as normal, but will spawn a new trajectory when the derivative coupling reaches a maximum.
   2. If a spawning trajectory has passed a maximum in the derivative coupling, a new trajectory will be spawned. This new trajectory is created, its momentum is adjusted to conserve energy, and all centroid structures are created, as well.
7. **Propagate quantum amplitudes** – The status of all TBFs and centroids is checked. If all structures have been propagated beyond the current time of the quantum amplitudes, the quantum amplitudes will be propagated.
8. **Generate restart output** – The sim.json files is created with the current state of the entire simulation structure. working.hdf5 is copied to sim.hdf5, so that sim.hdf5 and sim.json are synchronized. Old sim.\* files are copied so that several hdf5/json pairs are maintained in case of corruption or accidental deletion.

These 8 steps are repeated until of the end conditions is met (steps 3 and 4).

Trajectories and centroids each have labels, which contain information about the parentage of the trajectory. For jobs with a single initial trajectory, the intitial trajectory will be labeled 00. Trajectories spawned by a give trajectory will have labels constructed:

*PARENT\_LABEL*b*BIRTH\_ORDER*

Here, *PARENT\_LABEL­* is the label of the parent trajectory and *BIRTH\_ORDER* is the index of the child (the first child is 0, the second child is 1, and so on. For example, the first child of 00 will be labeled 00b0. The second child of 00 will be 00b1. The first child of 00b1 will be 00b1b0.

The labels of centroids are created by conjoining the labels of the two associated TBFs with the delimiter \_a\_. For example, the centroid between 00 and 00b1b0 would be labeled 00\_a\_00b1b0.