Things to do

Chan, Daniel, Pietro

September 22, 2014

Contents

1 Intersection algorithm

This should be a function called `find_intersections(trajectory_1,trajectory_2)` returning the intersection(s) of the two given trajectories. It shold be placed in the numerics.py file
If the trajectories don't intersect, it should return `[]` otherwise it should return a list `[[point, index_1, index_2], ...]` where

- `point` is the exact point of intersection, given as a complex number
- `index_1` is the index of the element of `trajectory_1.coordinates` that corresponds to the exact intersection point. (Same for `index_2`). This is important because it will be used to determine the pertinent

charge to use in computing progenie by means of the KSWCF. **We are taking the shortcut of *not* adding the actual point to the trajectory, so we introduce a tiny approximation. Keep in mind if trouble arises.**

- The exact point of intersection should be added to the lists `trajectory_1.coordinates` and `trajectory_2.coordinates`, at the appropriate point **warning:** coordinates attributes in the Trajectory class are given as a 2-vector, not as a complex numer.
- Moreover, the corresponding value of the period of η for that trajectory should also be added to `trajectory_1.periods` (and same for trajectory_2), this goes as a complex number.

A very primitive intersection algorithm is already present in numerics.py.

2 KSWCF - done (can improve a bit more maybe)

This should be a function called `progeny_2(data)` placed in the file kswcf.py, that takes `data = [ [` γ_1`,` $\Omega(\gamma_1)$ `] ,[`γ_2`,` $\Omega(\gamma_2)$`] ]` and returns a list of new charges and corresponding degeneracies, *excluding the parents*: `[ ... [` $n\gamma_1 + m\gamma_2$ `,` $\Omega(n\gamma_1 + m\gamma_2)$ `] ... ]`. A primitive example is provided in the file kswcf.py.

3 Rotation of the phase - done

A function called `phase_scan(theta_range)` in structure.py, that repeats the whole computation of primary k-walls, their intersections, generation of descendants, and iterations to get new intersections, new descendats etc.. for different phases.
The argument `theta_range` is of the form `theta_range = [theta_in, theta_fin, steps]`.

At each phase, it *stores the information about intersection points and their genealogy.* This information will then be used to construct full MS walls.

As of now, it keeps track of all intersections and all k-walls at every phase, and returns a big array containing them: this has the shape `[ all_ints , all_trajs ]` where `all_ints = [ ... [ int1, int2, ... ] ... ]` contains arrays of intersections of given phase and similarly `all_trajs = [ ... [ traj1, traj2, ... ] ... ]` contains arrays of trajectories of given phase.

4 Genealogy - done

Not sure how to structure this yet. But it should be a function called `build_genealogy_tree(intersection_point)` that takes as argument an object of the class IntersectionPoint. It should use the attribute `intersection_point.parents` to recursively determine the trajectories from whom the point descends. It should eventually return a structure (type of it is to be determined) that contains the genealogy of such point, meaning which branch-points are the farthest ancestors of the intersection, and in which associative order, eg: `[[BP1,BP2],BP3]`. Not sure if keeping track of branch-point ancestors will be enough, maybe not. Maybe should keep track of trajectories in the middle as well.

The purpose of genealogy is to identify which intersections (occurring at different phases) belong to the same wall of marginal stability.

5 Construction of MS walls - done

Once the phase-rotation algorithm is working and the genealogy is ready, we should be able to group together intersection points coming from different phase-snapshots, and build analytic MS walls out of them. This will need some thinking

6 Branch cuts

In the function `prepare_branch_locus` in file structure.py, should create an array of branch cuts called `bcts` that is to be returned (see the function there). Each branch point will generate a branch cut, if we decide to pick straight lines, at a certain angle, then these cuts are already created correctly by the `init` method of the `BranchCut` class. It remains to

- tune the parameter `branch_cut_cutoff` in file parameters.py (length of the branch cut)
- write the method `check_cuts` in the class `Trajectory`, which is supposed to detect intersection of trajectories with cuts, and determine the `Trajectory.splittings` (find exact point, add it to `Trajectory.coordinates`, then the splitting contains the indices of these exact points. The corresponding kind of addition must be done to `Trajectory.periods`) and determine the values of `Trajectory.local_charge` on segments

of the trajectory between the various `splittings`. See the dummy method in the Trajectory class.

7 Charges of branch points

This should be an algorithm that, at the very beginning —while creating the branch locus— determines automatically which charges to assign to each branch point (hence to each branch cut).

8 Storing Data

When running a long computation, involving probing several (such as O(100)) phases, it would be good to store all data on a separate file. In particular, we should

- Create an external file(s) whose name contains date and time
- Store data in such a way that it is retrievable: just need to store the output of the function `phase_scan`
- Write a module for retrieving (reloading in the RAM) the above data ready for analysis