A rough road map for the *MOSE* project

Chan, Daniel, Pietro

May 20, 2015

Contents

Chapter 1

K wall networks

1.1 Multithreading - done

At least the generation of each K-wall network should be done in a single thread, and multithreading should be implemented such that during `phase_scan()` we can run a multi-threaded program.

Saving to and loading from files will take a long time, and running it on a separate thread will be beneficial, although its implementation will not be straightforward.

It is worthwhile to consider how to implement a finer threading, for example a single thread for each K-wall, but this may be too fine and the overhead of threading management will be overwhelming. But on the other hand a thread per network seems to be too crude.

1.2 Intersection algorithm - done

Warning: current algorithm assumes that there is a single intersection between two segments in the same bin.

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.

1.3 KSWCF

1.3.1 Basics - done (maybe can improve a bit more)

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.

1.3.2 Enhancement

At intersections, make KSWCF really handled by phases of central charges (right now it's auto-correcting to get positive pairing). This can be done once signs of priary k-walls are fixed by weierstrass.

1.4 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.

1.5 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.

1.6 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

1.7 Branch cuts - done

Note: done in a different way than suggested here. The main difference is that I got rid of branch-cuts, and only used branch-points, taking all cuts to extend vertically to infinity.

In the function `prepare_branch_locus` in file structure.py (*warning: the whole structure of the code has undergone a major change, need to look for this function*), 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. Cuts should be straight lines going to infinity, at a certain angle `THETA_CUTS` to be specified in the file config.py. A tentative implementation of this kind of cuts is already provided 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` (*Warning: now this class has been replaced by the* `KWall` *class and its daughter classes*), 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.
- Update the intersection-searching procedure: when two K-walls intersect, the intersection pairing should be evaluated based on the charge of the K-wall at the intersection, this of course can only happen after K-walls have evolved *and* intersections with cuts have been taken into account.

1.8 Branch-cut crossing (possible) enhancement

When computing branch-cuts crossing of K-walls, should add the intersection points to the trajectory? Right now it's just taking the nearest point to handle the intersection.

1.9 Charges of branch points (Weierstrass module) done

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). It should also specify the DSZ matrix. All this information should replace `DSZ_MATRIX` and `FIXED_CHARGES` which are now given by hand in the config.py file.

1.9.1 interfacing of main code with weierstrass module

- mainly the weierstrass module will replace the elliptic_fibration module
- Branch-points have a new attribute called 'monodromy_matrix' which should be determined by Weierstrass

- around line 30 of k_wall_network.py, the sign = [-1,+1] or [+1,-1] should also be determined by Weierstrass

1.10 Storing Data - done – see instructions

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
- ✓ Bonus: write a module that (using the external data) will create a separate folder, containing snapshots of the k-walls on the moduli space for all scanned phases

Instructions

To save data, either a single k-wall network, or an entire phase-scan and the related MS walls, use the option `-w` in the command line. For example:

```
$ python2.7 -m mose -l info -s 0 -w -g -c fibration_invented.ini
```

(e.g. in bash, form the folder containing `mose/`)

or

```
run __main__ -l info -s 0 -w -g -c fibration_invented.ini
```

(in ipython)

This will produce a file, whose name will be of the type:

`label_YYYY-MM-DD-hh.mm.mose`

with `label` being either

`single_network` or `phase_scan` .

To read the data, from the python shell import the function `f_recover` from the file `save_to_file.py`, i.e. use:

```
>>> from save_to_file.py import f_recover
```

Then use this function to load the data from a `*.mose` file:

```
>>> f_recover('some_file_of_type.mose')
```

Now there are two possibilities:

- If the `label` is `single_network` then the above command return an instance of the class `KWallNetwork`.
- If the `label` is `phase_scan` then the above command will return a list of the form

$$[[\mathrm{KWN}_1,\ \mathrm{KWN}_2,\ \ldots,\ \mathrm{KWN}_m],\ [\mathrm{MS}_1,\ \ldots \mathrm{MS}_\ell]]$$

with `KWN, MS` being instances respectively of the classes `KWallNetwork, MarginalStabilityWall`

In either case, you can the go on and load plotting functions from our libraries and analyze the data.

1.11 Refined numerical issues with evolution - done

Trajectories evolve according to the PF equations, these are governed by a matrix which becomes singular at the singular locus of the moduli space (since they contain the discriminant). The `odeint` algorithm will get into trouble integrating PF in the neighborhood of these points. In particular, the problems seem to consist in trajectories bending abruptly when they run close to singularities.

Possible solutions:

- Once a trajectory was evolved, search backwards to see if it passed near a singularity. If it did, cut it in a neighborhood of the singular point (where it enters a disk of radius ϵ centered there), then invoke a new evolution algorithm (such as that of primay k-walls), then revert back to PF.
- Once a trajectory was evolved, search backwards to see if it passed near a singularity. If it did, erase the trajectory before next iteration, so it cannot create bogus offsprings due to bad numerics (e.g. the bending will make it intersect k-walls it wasn't supposed to meet).

The first possibility seems better, however the evolution method of primary walls will likely run into trouble too, recall the behavior of that method near a MS wall (which certainly would be met by the evolving kwall, when crossing the disc centered around the singular point).

The second possibility might be numerically heavy, but it could be greatly improved by tracking e.g. the determinant of the PF matrix, and using that as a red flag, instead of computing distances from singularities. It would clearly leave an unexplored area nearby a singularity, however this can be fixed by the idea of section 1.12

Note: solved with the second option. Introduced a parameter called `TRAJECTORY_SINGULARITY_THRESHOLD` that controls the value of the **determinant of the picard-fuchs matrix** along the trajectory's evolution, above which a trajectory is deemed too close to a singularity. To start with, this threshold is set to be 10^6.

1.12 Enhancing MS walls

1.12.1 Adding endpoints - done

A very simple thing to do: from the genealogy data of an MS wall it should be possible to tell whether the wall starts/ends at certain singularities. Then, make a module that creates new intersection-like objects from data of those singularities, and add these to the MS wall.

1.12.2 Interpolating function

Replace numerical data points with anapproximate interpolating function to have a nice smooth MS-wall. A simpler alternative is just to draw a line matching all the dots, although it won't be as nice.

1.13 Computation of central charges

Write an algorithm that computes the central charges along K-walls. At every point of a kwall, it should compute Z_γ, then it should provide a new attribute `kwall.central_charge` consisting of a corresponding list of complex numbers.

1.14 Checks at MS walls

In file misc.py, fill in the function. Given two intersecting kwalls, if the phases of their central charges at the intersection aren't reasonably similar, raise some sort of warning.

1.15 Automatic sorting of the K-factors in the spectrum generator

In file misc.py, fill in the function `sort_parent_kwalls`. It should check the phase-ordering of central charges of the kwalls just before they intersect, then it should return an array `[kwall1, kwall2]` where the phase of `kwall2` is the *smaller* one. This would correspond to having spectrum generator $\mathbb{S} = K_{\gamma_1} K_{\gamma_2}$.

1.16 Fibration-handling module

1.16.1 Storing and calling fibrations - done

We have a code that handles a class of fibrations. The separate data specifying each fibration should be stored in a tidy way, and called by the core part of the program with a simple function. A possibility is to have a different text file containing the data of each fibration, formatted in a standard way, and to load such data by giving the name of the file (e.g. `SU2_Nf=0.xml`).

1.16.2 Creating arbitrary fibrations

A nice addition would be a function which allows the user to add a custom-made fibration. For example, one could be prompted to specify the number of UV parameters $m, \Lambda, \ldots$, then be prompted for $g_2(u), g_3(u)$. Then the code would store this data in one of the fibration files described above, permanently, with a user-defined name. Then this fibration would be available to the user for future study.

1.17 Tilting the u-plane

Make sure that branch cuts do not overlap. If two branch-points align vertically, the code should automatically rotate the u-plane by a phase since the beginning. A suitable phase can be determined automatically to be e.g. a half of the smallest phase between any two discriminant loci.

1.18 Visualization module

As we will be analyzing a big array of data, and storing it for later analysis, it is desirable to have a separate suite that handles visualization of this formatted data. This will have a close interplay

with the GUI. The GUI will pass to this module certain data such as KWallNetwork or MSWall type of objects.
So this module should provide (probably) *two* functions: one for single K-wall-network frames, and the other for MS walls and multiple K-wall-networks. It should take some *options* to determine what kind of data to print

- charges of branch points / Kwalls at various points,
- degeneracies of Kwalls,
- MS wall data such as charges and pairing
- ...

1.19 GUI

Write a graphical user interface. Basic functionalities:

- Analyzing both single networks and phase-scans with MS walls, storing the data in an external file
- Loading the data stored in `*.mose` files and analyzing it, e.g. by plotting it, or by printing certain relevant data, such as charges, degeenracies, etc etc.
- Bonus: write a module that uses the screenshots produced by the function scanning multiple networks, to provide a window with a slider tool and lets you browse the k-wall evolution. This is quite useful for debugging and checking the numerics (esp. the undesired bending of k-wall at critical phases)

1.20 MOSE distribution packages

Figure out how to create installation packages for major OS's (are tablets fair game as well?).

Chapter 2

Extracting Refined Topological Invariants

For the distant future: figure out how to handle "finite networks", and automatically recover topological information on the fibration.

Chapter 3

Hyperkahler metrics

For the distant future: figure alorithms for solving TBA equations and getting the explicit metrics.