Lessons learned at LLNL

Zachary Matheson

June 14, 2017

Nucleon localization function

1 March 2017

Most recently I tried using Erik's modified version of HFBTHO to run for several constraints along Q_{30} (or anywhere, really). What I'd like to do is use HFBTHO to generate densities quickly for ^{176}Pt between $Q_{20} = 241$ and $Q_{20} \approx 300$, and at $Q_{30} = 4, 18$ (something I decided semi-arbitrarily once upon a time). I'm putting that on hold for a bit while I work on this inertia thing. Erik sent me some notes for perhaps getting the code to do what I want it to do, which are in my email. The files are currently in /p/lscratchh/matheson/locali-176Pt/hfbtho (/erik for testing his version of the code). Another thought I had was to try constraining Q_{40} to something reasonable, and then releasing that constraint to find the actual density (hopefully) nearby.

17 May 2017

Okay, I finally have some localizations worth talking about. I ended up settling on ^{176}Pt, going along the axes $Q_{30} = 6$ and 18. Then once I got close to scission (and things started having trouble converging), I turned off the constraint on Q_{30} thinking that would perhaps help with convergence and also round to fragments with the nearest whole number of particles. And it seems to have somewhat worked.

The more mass-asymmetric case ended up shifting from $Q_{30} = 18$ to a little over 20, which is fine, though it should be noted that those calculations failed to converge and I'm not totally sure why. But taking them as fully-converged, correct solutions (in any case, I expect the densities to be pretty close to reality), we produce the fragments ^{93}Nb, with an elongation $Q_{20} \approx 3-5b$, and ^{83}Rb, with an elongation $Q_{20} \approx 10-15b$ (both fragments also show a small mass asymmetry of less than $1b^{\frac{3}{2}}$, no doubt due to the heavy interactions between the two non-yet-fully-independent fragments). The weirdness of those two fragments leads me to wonder if perhaps I released the Q_{30} constraint too late in the development of fragments, such that those fragments were determined (according to Nicolas's fission fragment toolkit in HFODD), *at least* by $Q_{20} = 285b$, and probably well-before that.

In the more mass-symmetric case (but still with $Q_{30} \neq 0$), it actually appears that the system chose to produce two equal-mass fragments (^{88}Y), and then to just deform one of them more than the other. So even though their mass and charge distribution is symmetric, the kinetic energy distribution is not.

Now, what I'm wondering is what is the benefit to all of this? I think the goal was that it would be nice to be able to identify fission fragments well before scission. Well, let's look at the output files I actually saved (why did I not think to save them all?! ⌢). In the symmetric case, at $Q_{20} = 285b$, the calculation actually failed to converge because the number of iterations was reached. But we did get an output, and according to that output the "prefragments" at that stage were ^{96}Tc and ^{80}Br. $Q_{20} = 300b$ also failed to converge, and also

gives two asymmetric fragments (^{91}Nb and ^{85}Rb, but the localization plots *look* symmetric, in case that means anything). When I got around to doing $Q_{20} = 315b$, the fragments are actually *still* not symmetric (according to the fragment toolbox) (I'll just print the output here):

```
*   ------------------------------------------------------------------------   *
*             |   LEFT  FRAGMENT (z < zN)   |   RIGHT FRAGMENT (z > zN)       *
*   ------------------------------------------------------------------------   *
*       <A>   |          88.9232            |          87.0768                *
*       <Z>   |          39.4393            |          38.5607                *
*       Etot  |         -33.0708            |         -46.1156                *
```

And looking at the localization, you get the sense that it's going to be a *fairly* even split, but you can't be totally sure because there are still some subtle differences between the two halves.

Okay, well, let's hold on here. Say we take these, and we have some sense of what the fragments are likely to be so we plot those as well. And then we sort of "overlap" the fragments and see if it matches up with the parent. So the way you could do this is to run calculations for several fragments in the region near where you expect fragments to appear, and give them the deformations you expect them to have at scission, and see if you can distinguish between the various fragments: do they have distinct clustering structure? Or perhaps some kind of unique shell-looking behavior?

18 May 2017

Just by way of update on the mass-symmetric(-ish) case: I brought the quadrupole deformation out to 400 and still no signs of splitting. The properties of the fragments are equilibrating so perhaps it won't scission until it is able to conserve energy by doing so? Which would probably mean redistributing energy amongst the particles for a bit longer and then it'll be ready to break apart.

Impact of basis deformation on E_{HFB}

1 March 2017

We wanted to see how much the observables of the system would change if we used a single HO basis across an entire PES. This is based on a misunderstanding I had of something Jhilam said, where in order to use his inertia code, you need adjacent points to have the same basis deformation and other basis properties (so that you can numerically take derivatives of the densities at those points). It turns out he gets around this problem by changing the basis every 30b along Q20, but all the same we thought it would be good to check the dependence of system observables on the basis, especially since the half-life is so dependent on small deviations in the potential energy (a change of 1 MeV can affect the half-life by orders of magnitude).

To test, I took three points on the PES I had generated for ^{294}Og: (-14, 0), (72, 0), and (148, 28). According to the output file, the basis deformation chosen for each of these (by the automatic basis setting routine in HFODD) was, respectively, AL20 = 0.187, 0.424, and 0.608. I took those record files and used them to restart a new calculation, this time with the basis deformation set uniformly to AL20 = 0.61 (and AL40 = 0.10). The superdeformed asymmetric shape was, understandably, least affected by the change, with the kinetic energy varying by about 3 MeV but the total energy varying by only about 0.14 MeV (-2085.878548 vs -2086.012955). Quasiparticle and canonical single-particle states were nearly identical, and fragment properties were almost the same (except for the interaction energies, which were quite different). The elongated symmetric shape differed by about 0.6 Mev (-2080.815396 vs -2080.202346). The oblate ground state didn't converge in the allotted time, but based on its last iteration it was probably going to finish with an HFB energy around -2078.649984 (compared to -2080.263986 from before), a difference of about 1.6 MeV.

3 March 2017

I've written a Python script which extracts the basis parameters from a previous run of HFODD, and uses it to initialize a new run for several neighboring points in order to facilitate an inertia calculation down the line. A problem we noticed, though, was that, while setting the α_2 deformation parameter worked fine, the basis was totally different because there was an additional constraint on α_4. It turns out that what had happened is that the code automatically sets default values of α_2 and $alpha_4$, UNLESS you set the code to choose the basis automatically, in which case it only sets a value for α_2. You can get around that by deleting the preset line in HFODD and recompiling, but I'd like for there to be an easier way. Perhaps the basis matrix is initialized to zero? So we could get around it by just setting $\alpha_4 = 0$ in the input file? $\Rightarrow$ Aha, yes. That'll work. So we're benchmarking the time on that now.

Another thing we're testing is comparing versions of the inertia code. Jhilam sent Nicolas an input and an output for ^{240}Pu. I'm running a single point now. Later, I'll run the surrounding points using both my convention and Jhilam's for how widely-spaced the points should be. I'll also compare the inertia computer with the code Jhilam sent me versus the one in Nicolas' repository.

Unfortunately, this run uses Lipkin-Nogami, and for some reason the parser doesn't write the data to the XML file, which ultimately means I'm going to have to set up the subsequent runs by hand. The question is whether to do so using Jhilam's convention (for benchmarking purposes) or the one Nicolas and I talked about (where the points are much closer and you might get a better value for the inertia). Computing the neighboring runs will probably still take a similar amount of time (it took roughly 50-60 iterations for a deviation of about 0.001 units in each direction. It'll probably be more for something farther away but if you already have those computed anyway it might not be such a big deal).

So I went ahead and did that. Once those jobs complete, we can try to analyze the inertia using each of the inertia codes, just to make sure they both

give the same results. I used Jhilam's grid spacing, but we can try it again later with the narrower grid spacing once we see how long it takes for these points (which have a grid spacing of 1 in the multipole constraints, and 0.1 in the pairing constraints) to converge and then decide if using the narrower grid is economical and useful. Actually, this would be a good test case for that; if we see a noticeable improvement in accuracy, then the extra time-to-solution for the narrower grid might be worthwhile.

Of course, to do that we'll need working inertia codes. Right now, I don't undertand why, but for some reason we're getting some kind of runtime error in LAPACK:

`IntelMKLERROR : Parameter8wasincorrectonentrytoZGEMM`

6 March 2017

Today I worked on two computational problems: the large number of iterations, even for a small perturbation from the record file's original point; and benchmarking a working inertia calculation against Jhilam's results.

For problem #1, I noticed that even after setting the basis parameters manually in the input file, I was still seeing that a different basis was used during the run and [consequently?] these were still taking 70-80 iterations to converge. I showed Nicolas and he had me flip a Lagrange multiplier continuation switch in the input file, but I've re-run the code since then and the problem still exists, that the new run uses a slightly different basis than the original. and it still takes 70-80 iterations to converge.

For problem #2, I compiled the code for a 3D benchmarking run to compare with Jhilam's results. The code runs without that weird MKL error I was getting, but the results are incorrect.

I'm getting:

77.00 1.00 2.00 −1801.1466270.000000 0.598205 0.598233 0.000000 0.000000 0.598219 (1)

whereas Jhilam is getting:

77.00 1.00 2.00 −1801.1830.010463 0.033608 0.000842 −0.001305 −0.000412 0.000152 (2)

7 March 2017

Today, after modifying the input file and doing some debugging, I'm getting:

$$0.011378\ 0.034354\ 0.000000\ -0.001988\ 0.000000\ 0.000000 \tag{3}$$

which is within 0.001 across the board. So not bad, but not exact, either. My question now then is if this is something that should be exactly deterministic. I'd think that it should be, yes. The max basis size has changed at compile time but I don't think that affects the final results. ⇒ Ah. I think the problem is that I changed the basis characteristics in the input file to HFODD. I calculated

the surrounding points using the basis from the centerpoint. Which, I suppose is useful to know... In fact, yes, I did that because we need the basis to be the same, no? So in order to reproduce Jhilam's exact results, we'd need to know his exact basis.

Pending that, I think I officially have a working inertia code. There are some nice things I can do to streamline inertia calculations, probably within Python as opposed to Fortran, but for now it's something to start with.

I'm still not sure what's going on with the other thing, with the basis getting changed even after setting it manually. I even tried another run setting INPOME=0, just to see what would happen (even though I expected that to make it worse), and frankly I didn't see a difference (the output files reported the same wrong basis).

When you come in tomorrow, the things to work on should be: 1) spend the morning working on deriving the ATDHFB inertia, then 2) in the afternoon, talk to Nicolas about the inertia code you got working (maybe Jhilam will have sent you a basis to use), and perhaps see if Nicolas has any additional insight on the basis problem you're seeing here.

8 March 2017

Good news: Nicolas figured out what was happening with the basis. Apparently there were rounding truncation errors that popped up when the parameters FCHOM0 and AL20 were written to file, and the line FREQBASIS isn't even read at all. Going back into the source and figuring out how FCHOM0 and AL20 were computed for the centerpoint, and plugging these values into the input file with lots of decimal points seems to have solved the problem. It still takes several iterations to converge, but it will hopefully be faster than before (it's still running so we'll see).

The formulae you'll ultimately need to implement in your Python script for preserving the basis are the following:

$$\omega_0 = 0.1 q_{20} e^{-0.02 q_{20}} + 6.5 \tag{4}$$

$$\texttt{FCHOM0} = \frac{\omega_0}{\left(\frac{41}{A^{\frac{1}{3}}}\right)} \tag{5}$$

$$\alpha = 0.05\sqrt{q_{20}} \tag{6}$$

where q_{20} refers to the quadrupole deformation of the centerpoint.

9 March 2017

Aha! Figured out what was causing the ZGEMM error in the inertia file. I was using a max basis size of 1200 when I compiled the inertia code (as defined in hfodd_sizes_....f90), but the particular matrices I was trying to multiply were created using a basis size of like 1600. So that's resolved. One thing, though,

is that, while it gives the same results as the 3D case, they are in a different order. So that's something you should clean up in the code.

Additionally, I've redone the calculations for the inertia benchmark, this time using the correct basis for the points surrounding the center point, and fixing whatever Lipkin-Nogami problem I was having, and this is what I get:

$$0.011719\ 0.034843\ 8.287077\ -0.002180\ -0.054160\ 0.020690 \tag{7}$$

with $E_{HFB} = -1801.146627$ Again, for reference, here are Jhilam's results:

$$0.010463\ 0.033608\ 0.000842\ -0.001305\ -0.000412\ 0.000152 \tag{8}$$

Without Jhilam's basis, we still don't have his result (in fact, it's even further than it was before). But the biggest problem is in that λ variation somehow. Still not sure what's going on with that.

Just judging from the outputs, it looks like everything converged properly and the results make sense (energies were approximately the same, particularly for those which did not explicitly involve pairing; unconstrained multipole moments are approximately the same as far as I can see). Could it be that I put the qp files out of order? I have the magnitudes right for sure (lambda changes by 0.01; the Lagrange coefficient magnitudes for equally-spaced grid points go as $\pm\frac{1}{2\delta q} = \frac{1}{0.02} = 50$). I'm running it now with the signs switched, just in case I had them backwards somehow.

$$0.011719\ 0.034843\ 8.287077\ -0.002180\ 0.054160\ -0.020690 \tag{9}$$

But if that doesn't work (which it didn't), then perhaps it's a problem related to the parameter I switched? Is there a modification to the formula when you change λ instead of a multipole moment? Did I change the wrong λ? Was I only supposed to change λ for protons *or* neutrons but not both?

10 March 2017

Okay, I have information about the basis Jhilam apparently used: for $Q_{20} = 77$, he would have used the basis corresponding to $Q_{20} = 60$ (as far as FCHOM0 and α_{20} are defined). In general, his method is to round down to the nearest multiple of 30. With this I get:

$$0.011663\ 0.034763\ 8.610518\ -0.002182\ -0.054110\ 0.020785 \tag{10}$$

with $E_{HFB} = -1801.173695MeV$

13 March 2017

It seems there's no problem with the λ terms in the inertia output (at least, not one unique to just the pairing terms). The issue there is that Jhilam normalized his coordinates in the end, such that $\delta x = 1$ for every collective coordinate instead of the $d\lambda = 0.01$ I'm using. Since this only affects the denominator in

my derivatives $\frac{\delta\rho}{\delta q}$, this means shifting the decimal point over by 2 slots (or 4 in the $\lambda - \lambda$ case) for the pairing terms.

So it *looks* like the inertia code is working. Everything is correct to within 20% compared to Jhilam's result. But it would be good to make absolutely sure, which means examining Jhilam's output files, if possible.

Inertia Wrapper

17 March 2017

The past couple days you've been working on parallelizing the inertia routine, and figuring out how to quickly setup runs for the "satellite" points on your PES. You've got some scripts that will help, although between writing them a couple days ago and using them now, it seems you've already sort of forgotten what they need to work - or rather, you see that they aren't quite as universal as you were hoping. You need an XML file, which needs output files. If you have a big XML file and need to split it up, you did that by hand today but I'm guessing there's an easier way. But it looks like you WILL need to do some splitting up. Not the way you had originally anticipated, where you submitted a job for every single point on the PES to have its satellites computed. But since the speedup for using nearby points only seems to occur when you use the same basis as the centerpoint, this means that you have to group centerpoints by Q_{20}. That'll reduce the number of jobs on Quartz by 15, but it's still a nuisance.

Anyway, you've done that for values of Q_{20} between 0 and 8, and you're waiting for those to complete (hopefully they'll take less than 3hrs!). Once they do, you can test out your post-run_whatever.py Python script to see if it really *does* group things nicely. You should have it generate the inertia input file, with multipole moments, Lagrange coefficients, and file names correctly and automatically (it's mostly there already; I think you just need to add the multipole constraints). And then you can test out your MPI version of the inertia code. You've tested it and it works when compiled and run serially (at least, it gives the same results as before, which you'll assume are correct until you hear back from Jhilam).

21 March 2017

When you get around to setting up your inertia wrapper, I think the nicest and most user-friendly thing you could do would be to activate the Python setup script.

1. It should take as inputs the XML file and the directories containing record, qp, and maybe output files (and Lipkin files if you're into that kind of thing).

Note Whatever naming scheme you use, it should be independent of the indexing scheme used originally.

2. It'll create and populate directories for each Q_{20} value, create path and path_new files for each directory that HFODD can use to calculate the neighbors.

3. Then it'll create the SLURM batch script (with multiple SRUN commands) that HFODD will use to compute the neighboring points.

4. Once HFODD is done, those same MPI ranks will collect the outputs into their proper folders with their proper names. Python will create input files for the inertia code in each subdirectory. Then it will run the inertia code on a subdirectory level.

5. After the inertia code completes on a subdirectory level, the results (and perhaps the output files if you're into that kind of thing) will be collected and compiled into the parent folder, and a master output file will be created.

28 March 2017

I haven't really explained what I've been up to for the past several days (or at least, I haven't committed my thoughts from my notebook to my computer). After discovering that much of the inertia wrapper I was trying to develop had already been done more cleanly by Nicolas, this week I've started developing a new Python class called Point, which corresponds to a single point in the PES. I'm going to give it some neat methods that will sort of streamline the creation of neighboring jobs and such. Right now, I've successfully been able to initialize an instance of Point from an instance of DataFile (just by picking one of them out randomly). The script which was able to do that is the following:

```
from pes import *
from collections import defaultdict,Counter
oldpoint = Point('294Og_PES.xml', 176, 118, var=('q20','q30'))
tree = oldpoint.ReadFile()
glob_dico = oldpoint.GetGlobal(tree)
dico = {}
dummy = oldpoint.GetVariable(tree, val='id')
dico['id'] = dummy['id']
for constraint in oldpoint.var:
dummy = oldpoint.GetVariable(tree, val=constraint)
dico[constraint] = dummy[constraint]

dico_arr = {}
for observable in ['EHFB', 'Z1', 'A1', 'zN', 'qN', 'D']:
dummy = oldpoint.GetVariable(tree, val=observable)
```

```
dico_arr[observable] = dummy[observable]

newpoint = oldpoint.CreateNewPoint(glob_dico, 0, oldpoint.var, dico, dico_arr, 0)
```

One complication that arises, however, is that in order for this to work as currently written, the old point must be initialized from the entire XML file, and that XML file must currently be an element in the Point class (which technically is possible but it sort of violates the goal of the project). It would be nice to have a method which truly takes an actual DataFile and spits out a Point. What you might consider doing is generating a DataFile from a DataFile, and then taking the output DataFile and using it to initialize a Point afterwards. Anyway, something to think about.

29 March 2017

Welp, it turns out I've been working on an outdated branch of the PES module. I got access to the latest one today and spent the afternoon familiarizing myself with what's new. It seems really nice and sleek, so I think it should be pretty easy to work with and adapt what I've been doing. Now I just need to spend some time figuring out what changes.

And I had *just* gotten my instance of `Point` to initialize from a `DataFile`...

31 March 2017

So the good news is I've created a successful Point class in the new `pes_tools` module, and you can specify a set of constraints with their values and it'll spit out an XML tree for that particular point. You can also find neighbors surrounding that point as XML trees, the idea being that you can feed them into input files for use in an inertia calculation.

The thing is it all feels so stupid. It's been done already! Why does it need to be done again? Indeed, this will look a little bit cleaner, hopefully. Then your actual inertia_setup script can probably be actually pretty short. You'll still have to deal with file movement, but a lot of things will be moved behind the scenes into the various pes module methods.

The toughest spot will be the part you've been avoiding the most: figuring out how to divide up a whole large reservation of processors to cover the entire PES efficiently and without overlap. You can fit 3 neighborhoods per node (assuming 4 neighbors per neighborhood and 3 OpenMP ranks per task = 12 tasks per neighborhood, with 36 possible tasks per node). Just FWIW, assuming something like 830 neighborhoods (which is roughly what you've got right now for ^{294}Og), that's going to require $830/3 \approx 277$ nodes. Estimating walltime, I'd give yourself something like an hour and a half to two hours to do the HFODD run for the neighborhood, and another half hour to do the inertia calculation. Three hours *should* be more than enough time to do it all. I say "should" because these SLURM emails sitting in my inbox report inertia calculation times closer to an hour and a half, for the HFODD portion, but they never

actually completed the inertia calculation (I was trying to do them all at once, but it's probably best to just do it by neighborhood, rather than for a whole slice of neighborhoods).

Whatever script you end up creating, you should make it modular and flexible enough that you can pin a path calculation onto the end of it without too much trouble. And it should work in any number of dimensions (which thankfully, so far it does. Your other script would have taken some real tweaking for that to happen, though). The one place where this might struggle is when probing along the pairing parameter. I imagine you could tweak that, too, without it being a huge problem, but that's going to take some thought. Most likely everywhere you have a loop over `collective_variables`, you'll need to stick an `if q == lambda_n or q == lambda_p` statement in there.

7 April 2017

Holy crap, it's been another week! Welp, in any case, I think I'm going to give up trying to get MPI Spawn from within a Python script to launch HFODD runs. I've had some level of success. To pull this off, you needed to be able to launch several concurrent MPI tasks simultaneously, each with several OpenMP threads:

```
Task␣1␣BB/__/__/__/__/__/__/__/__/__/__/__/__/__/__
Task␣2␣__/__/__/__/__/BB/__/__/__/__/__/__/__/__/__
Task␣3␣__/__/__/__/__/__/__/__/__/__/BB/__/__/__/__
```

That's all okay. We got that. Then you needed to spawn onto these processors. You were able to achieve something like that by using the option `--oversubscribe` (although it's not clear that the specially-designated processors were used properly). But the real issue is launching HFODD after that. To do it, you'd need to somehow launch another `srun` with a different number of `OMP_NUM_THREADS`. But to use MPI Spawn properly, you need the spawned process to establish, and later, disconnect from an MPI Communicator (otherwise the process will lag indefinitely). You could get around this by combining the two and spawning a process which calls a Python script (with the proper MPI communicators enabled and whatever) that launches an HFODD subprocess via `srun` (be sure to change the number of threads!). And you know, this might work... but at this point it's just seeming less and less reasonable. And I'm not sure you'd be spawning onto the correct, pre-reserved processors *still*. I think you're better off in the long run just reserving a bunch of resources, and then launching a bunch of simultaneous `srun`s and letting the scheduler divide up the jobs itself. I feel like that's a lot simpler to implement, and probably a lot more portable in the end. To do *this*, you'll need to write a Python script to handle the file management, and then refer to it in the batch script (you can do it in parallel still, I'm sure), followed by HFODD `srun`s, followed by either shell or Python file movement, followed by inertia `srun`s, followed by more file movement. So you might have 3 or 4 Python scripts: pre-HFODD,

post-HFODD/pre-inertia, post-inertia, and a wrapper to write the batch script. That just feels so messy but it feels like the simplest way if you don't have the MPI Spawn (which was supposed to be the "One script to rule them all!"). I mean, I guess maybe the wrapper could write the other ones on the fly?

Just to reiterate and make it clear, though: **Spawning a multithreaded executable**... actually might work, IF you had a specially-made executable just for that purpose. The question is whether it's worth modifying HFODD. So my question is, for a spawned process, do calls to `MPI_COMM_WORLD` refer to the communicator created by the spawn (i.e. `MPI_GET_PARENT`) or everything from before? I'm inclined to believe that it should just be the stuff you care about. So you could probably get it to work by just adding a few lines `MPI_Comm_get_parent( parentcomm )` right after MPI_INIT and `MPI_Comm_free( parentcomm )` right before MPI_finalize (tucked inside some preprocessor option, of course). That might work... Okay, that's my goal for the end of the day. Compile a version of HFODD that is meant to be a worker in an MPI_spawn, and see if you can get it to run.

UPDATE

...Okay, I am officially declaring **MPI Spawn NOT the solution** that I'm looking for. When I set up a test run, there was some error. I didn't have time to diagnose it yet when I talked to Kyle and he pointed out that mixing MPI implementations will lead to all kinds of crashes. That means that since my Fortran executable was built using the default system MVAPICH while my Python launch script was running from a side installation of mpi4py built over OpenMPI, they were going to have issues talking to each other (which is probably what I saw). So no, that's not gonna happen.

10 April 2017

I decided I'm going to try doing things the easy way: just making one large resource allocation request, and in the script submitting several srun commands in the background and letting the scheduler figure out how to spread them out. I think that'll be the easiest and most portable in the long run. Right now the issue I'm having is that I'd like to define and launch those srun commands within a Python script; however, the resource allocation closes when the tasks in the script complete, and any jobs launched in the background by Python are disconnected from the batch script which launched Python, with the result that the scheduler *thinks* that the job has finished even though there are tasks running in the background. I could get around that by having one thread just sleep for a really long time, but that would waste a thread and also it'd waste a lot of walltime if there really was a problem in the executable somewhere that caused it to close early.

I ended up getting by it for the time being by just probing `ps` every 60 seconds, and if there is still an srun process running, the process sleeps for 60 seconds and tries again. I guess this isn't much better than having the one thread sleep for a long time since it still ties up a thread somewhere (which may or may not lead to a performance hit somewhere else? It's not resource intensive

so it shouldn't be a big deal to multithread this on a processor with some other task...), but it does get you to within a minute of the actual execution time, so worst case scenario you waste about 59 seconds of walltime.

And with that, I have a launch mechanism! Thank heavens I can move on now! Now to build the rest of that Python launch script (not to mention finishing up those point-based utilities)...

And the nice thing is that this is fairly portable. FOr a system that uses mpirun/mpiexec instead of srun, you can chain jobs together like this:

```
shell$ mpirun -np 2 a.out : -np 2 b.out
```

And in fact, you might even be able to do something fancier with srun by creating a MPMD config file:
`https://computing.llnl.gov/tutorials/linux_clusters/multi-prog.html`

15 May 2017

This isn't part of the wrapper, per se, but it is relevant to the development of the ATDHFB inertia code, and specifically the temperature-dependent implementation of it. The $T > 0$ version of the code ends up getting *HUGE* values for the inertia, even for small temperatures like $T = 0.05$ which should be almost unchanged from the $T = 0$ case. It's not a matter of accidentally dividing by zero when $f_a = f_b$ because the code explicitly filters those out. It's about the relative sizes of $\dot{\mathcal{R}}$ and f_a, or actually $\frac{\partial \mathcal{R}}{\partial q}$ and f_a. The code calculates both of these based on the HFB matrices and the quasiparticle energies stored in the .QP files. For the test case I looked at (which was a sample case of ^{240}Pu), $\frac{\partial \mathcal{R}}{\partial q} \sim 10^{-15}$ while f_a can range from $f_a \sim 1 - 10^{-309}$ or so. So I doubt it's a matter of numerical instability in the f_a's - the E_a's from which the f_a's derive are of the order $10^0 - 10^2$, after all - well within the working range of any double-precision arithmetic solver. Could it be a problem in the calculation of the derivatives and their densities that is making them accidentally several orders of magnitude larger than they should be? And my guess is no. Anything that produces an error or an instability in $\frac{\partial \mathcal{R}^{11}}{\partial q}$ would also produce a similar instability in $\frac{\partial \mathcal{R}^{12}}{\partial q}$ since they are built from the same things. The magnitudes of both are fairly similar ($\frac{\partial \mathcal{R}^{11}}{\partial q}$ is perhaps one or two orders of magnitude smaller than $\frac{\partial \mathcal{R}^{12}}{\partial q}$, which in this scheme is pretty insignificant), which doesn't mean they're correct but it *does* lend plausibility to the idea that they might be. And presumably the off-diagonal terms are correct, since supposedly they were back when Jhilam first worked with the code.

16 May 2017

The good news is that after talking to Nicolas, I've managed to reduce the order of magnitude of the resulting inertia from around 10^{∞} to about 10^3 by instituting a cutoff on all of the terms that get used to build up `amass`. Numerical

errors can start to creep in once you get past about 15 or so digits in a double-precision floating point number. In practice, that means that something with a power $a = x.xx \cdot 10^{-15}$ might have just been a rounding error that should have been zero - or, in general, the rounding error is of the same order of magnitude as the number itself, and therefore we should probably just discard the whole thing as unreliable.

So with a cutoff of 10^{-14} on everything (matrix elements, differences between f_a's, etc.), we got `-8.308560E+0003` when the T=0 calculation, by contrast, gave `1.618428E-0002`. So not perfect still, definitely. But it's better than it was! And with that cutoff set to 10^{-13}, the inertia drops even further to `-3.917522E+0002`. So again, this cutoff is probably one important component to add, but it looks like there's probably something else going on. We can't just keep cutting off more and more data until we get what we want! At this level of detail we should *definitely* be getting something reasonable.

Worth noting, by the way, is that even though I don't have these results saved anywhere, I did a trial cutoff of 10^{-15}. The result was very close to the real result: slightly smaller in magnitude, because certain terms were never added in thanks to the cutoff. But I didn't keep that result because it didn't include a single contribution from the F^{11}, F^{22} terms (as they are referred to in the code, or $\frac{\partial \mathcal{R}_0^{11}}{\partial q_\nu}, \frac{\partial \mathcal{R}_0^{22}}{\partial q_\mu}$ as they are in my notes). So it was essentially cutting out any of the actual temperature dependence.

For $T = 1$ and cutoff= 10^{-13}, the inertia I got was `-2.075285E-0001`, which as far as I can tell is not unreasonable (except why is the sign negative?!).

19 May 2017

Okay, here's sort of a summary table. δq refers to the separation between points used to calculate the derivatives $\frac{\delta \mathcal{R}}{\delta q}$. The numerator cutoff refers to whether or not I truncated the numerator (the densities) to the same level that I truncated the $f_a - f_b$. On the one hand, you could claim they should be truncated because there's a lot of complicated expressions leading into them and you can't really trust the results beyond a certain point; on the other hand, these numbers aren't the ones that seem to be causing the problem - it's the f_a's!

Temp	cutoff	Numerator cutoff?	δq	$\mathcal{M}_{20,20}$
0	none	no	0.5	1.618428E-0002
0	10^{-13}	yes	0.01	1.549012E-0002
0	10^{-15}	yes	0.00001*	0.000000E+0000
0.005	10^{-13}	yes	0.5	-3.891427E+0002
0.005	10^{-13}	yes	0.01	-1.417471E+0001
0.005	10^{-14}	no	0.5	-8.471734E+0003
0.005	10^{-15}	no	0.5	-1.452580E+0005
0.005	10^{-15}	yes	0.5	-1.452580E+0005
0.005	10^{-15}	yes	0.00001*	1.450188E+0004
1	10^{-13}	no	0.5	-2.075285E-0001
1	10^{-15}	no	0.5	-2.003790E+0001

Sooo, another thing: it's probably significant that the $T = 0$ inertia is positive, while all the rest of them are negative. Like, something's definitely up with that. [Repaired 30 May 2017]

* See the note for 9 June 2017, below

30 May 2017

I'd also just like to add (and I'm not sure how best to do it in the table so I'm listing it here separately) that I ran the code for entry 2 in the table with those same .QP files (10^{-13}, $\delta q = 0.01$, numerator cutoff used), everything completely identical except that I artificially claimed that $T = 0.05$. The resulting inertia was -1.413845E+0001.

Then I fixed a sign error and changed conj(F11(mu,nu)) to F11(nu,mu), and the result I got was 1.416943E+0001.

1 June 2017

Nicolas suggested a couple of things. One is that perhaps there's more to calculating $\delta\mathcal{R}(q, f(q))$ than you've currently got going on. Right now you basically ignore any variations in $\mathcal{R}$ with respect to f. That's a thing you could look into if you get stuck again, but since that means adding another term it seems somewhat unlikely that you would add a term that exactly cancels out to give you what you need.

The other is that maybe the problem is really just numerical. When he computes the perturbative expression, there appears a term that ends up canceling the f's in your denominator that are giving you all the trouble, but it's not clear where that would appear in the non-perturbative expression. Presumably that cancellation might exist somewhere, but it could be buried deep inside the computed densities, masked well within the numerical noise of those entities. He compared it to computing the limit $\lim_{x\to 0} \frac{sin(x)}{x}$: if you use the Taylor series expansion of $sin(x)$, then you should get nice convergence. But if you evaluate $sin(x)$ numerically ahead of time and then just divide, you might get something messy as a result.

So we're just going to check it as nicely as we can, under the tightest conditions. We'll use a very small step size to compute the derivatives (10^{-5}), we'll let the iterations converge to a very tight convergence level (10^{-9}), and we'll do it at both $T = 0$ and $T = 0.005$. Then we'll see.

9 June 2017

I'm not 100% I should trust the entries in the table for $\delta q = 0.00001$ because they didn't converge completely. I set a convergence parameter of 10^{-9} but HFODD quit and claimed "chaotic divergence" somewhere around the level of 10^{-7} or so.

12 June 2017

As we continue to explore the orders-of-magnitude discrepancy in the temperature-dependent inertia tensor (which, according the the current code, jumps by something like ~ 5 orders of magnitude when you go from $T = 0$ to $T = 0.05$), Nicolas suggested first checking the convergence of the inertia as a function of δq. Then once we have a handle on that, we can start breaking other pieces down one-by-one. Right now it looks like going from $\delta q = 10^{-1}$ to $\delta q = 10^{-5}$, the magnitude of the inertia drops from about 1.646e-2 to 1.493e-5, which is roughly a 10% drop. If we hold everything else constant, a 10% error in the inertia leads to $\sim \sqrt{1.1} \approx 1.0488$ or nearly a 5% error in the action. That may not sound like much, but then you take the exponential to get the half-life. All in all, I carried it through just back-of-the-envelope style using a value for the action of 19.1 taken from Jhilam's paper as a "typical" value of the action, and a 5% error in the action leads to an uncertainty factor of about 50 in the half-life. In other words, there's a range of not quite two orders of magnitude that are within your "acceptable" range for this level of precision in the inertia. I'm sure there's a good way to make this more rigorous, but there you go.

14 June 2017

I have a plot where I show the convergence of $\mathcal{M}$ as a function of δq saved somewhere in my dudeman4 Google Drive, which I should probably pull out at some point. But another important thing to notice is that the solution diverged noticeably at $\delta q = 10^{-6}$. That may be because the HFODD density calculation terminated at the level 10^{-7}. I'm going to test for that by pushing to a greater HFODD convergence level 10^{-8} and a tighter $\delta q = 10^{-7}$.

Another thing to check is how much the density changes between the last and second-to-last iterations. Because you can figure those two iterations are both basically "converged", but it may be that some matrix elements in those densities change by a "large" amount (say, 10^{-5}) between iterations. Essentially this means that there is effective numerical noise of order 10^{-5} (or whatever it ends up being) in the final resulting density, which means the precision we get when subtracting nearby densities to compute derivatives may be limited.

My preliminary analysis suggests that this won't be an issue. I compared proton densities for the last and second-to-last iterations of some point that I picked out. I don't remember which one (I'm going to guess $\delta q = 10^{-7}$ but I could easily be wrong), but it looks like it doesn't matter anyway because the order of magnitude of the differences appears to be one or two below δq at least. I checked a couple of different norms, but neither one seemed particularly threatening:

```
l2rho= 2.138288179952028E-008
l2aka= 5.496113165431270E-009
maxrho= 1.393350374494798E-009
maxaka= 3.834180094000197E-010
```

^{294}Og

10 April 2017

Moving on back to working with ^{294}Og, it would be nice to do calculations with multiple EDFs as a sort of gauge on uncertainties. For Gogny and UNEDF functionals, pairing is included automatically because it was explicitly included in the parameter fit. This is not the case in general, however. So if we want to use SkM*, for instance, we should figure out ahead of time what the pairing interaction will look like. Nicolas and Witek suggested a couple of ways of doing this. Nicolas suggests using experimental mass evaluations to compute binding energies and then the pairing gap, and I assume we can use a formula like this BCS formula to compute the pairing strength from there:

$$\Delta_{BCS} = (E_{max} - E_{min})e^{-\frac{1}{V_0\rho}} \tag{11}$$

where $(E_{max} - E_{min})$ is the spread of single-particle energies for orbitals participating in the pairing (probably taken to be about 60-100 MeV), $\rho = \frac{dn_s}{dE} \approx \frac{A}{100}MeV^{-1}$ is the energy density of states, and V_0 is the pairing strength. This should be done for both protons and neutrons. Here is the pairing gap for neutrons (there is not enough data for protons):

```
*********************************************************************************
*                                                                               *
*  EXTRACTED EXPERIMENTAL PAIRING GAPS, BCS-EQUIVALENT GAPS  AND S.P. LEVELS    *
*                                                                               *
*********************************************************************************
*                                                                               *
*  Z =118 ******************************************************************** *
*                                                                               *
* NEUT DELTA3  DEMINU  DEMIDD  DEPLUS  ESINGP  SEPAR1  SEPAR2  BIND.-EN  ERROR *
*                                                                               *
*?176  0.4410  0.0090  0.4455  0.8820  0.0000  -7.665   0.000 -2078.045 0.5880 *
*********************************************************************************
```

When all is said and done, you should get something in the ballpark of -250 MeV or so, plus or minus probably like 50 MeV. I think one of those approximations must be wrong somewhere because the numbers I'm getting are nowhere close. Anyway, I'll put that on hold for a moment because we don't have the proton data anyway.

Witek suggests a better way would be to use the inertia-like quantity in equation 4 of `https://journals.aps.org/prl/pdf/10.1103/PhysRevLett.116.152502`. Again, we'd run into the problem of not having enough datapoints nearby, plus I'm also not sure how we'd use this to extract a pairing strength. The paper gives an expression of pairing energy in equation 2, and another way of computing it in this presentation from Nobuo:
`https://public.ornl.gov/conferences/ns2016/3_Wednesday/Session_1/NS2016_W1_3_Nobuo_Hinohara.`

but still no way to relate to the pairing strength, unless you make assumptions about the pairing interaction (like making it only a surface pairing interaction).

18 April 2017

After talking to Nicolas about it the other day, I guess the best way to find the pairing strength given the pairing gap is actually to just try several different pairing strengths in your input file, and then run them until you get the pairing gap you were hoping for (same goes for whatever quantity it was Witek wanted us to compute). This might involve several HFODD runs for several different nuclei, but it's algorithmic enough that you could make that happen automatically by automating it somehow. Anyway, we'll just leave that up to Jhilam since he's doing the SkM* stuff and I'll take care of UNEDF1-HFB.

3 May 2017

Here are some observations about the ongoing PES's that I have up to this point:

1. Symmetric fission has an initial barrier blocking it, but if the nucleus tunnels through (or even wraps around) the barrier, it seems to go to lower energies faster. However, it still seems to scission asymetrically at lower elongations than symmetrically.

2. I'll have to look at the actual outputs or something, but it seems like perhaps triaxiality is reinstated at the first sign of scission around $Q_{20} \approx 200$. At this point, it seems likely that whatever scission takes place would probably have a large mass-asymmetry (that's not clear to me from the Q20-Q22-Q30 plot, however).

3. When I started my Q20-Q22 run, I used restart files corresponding (I think) to Q30=4. The result is that in the region between roughly Q20=75 and Q20=125, there is a strong tendency for the system to move to greater mass asymmetry (it'll follow the asymmetric fission valley in your Q20-Q30 plane). After the ridge dividing symmetric and asymmetric paths instersects with the line Q30=4 (which happens at around Q20=125), it will tend back towards symmetric configurations.

   (a) The first thing I'd like to do is investigate mass asymmetry for larger quadrupole deformations. I notice there continues to be some octupole activity beyond Q20=125 and I don't know if that is an artifact of which points I used to restart HFODD or if that's just what the surface looks like. Just from checking the output file of Q20=152, Q22=12, it *looks* like it started from the Q20=152, Q30=4 file - and NOT Q20=120, Q22=12 - but I'd like to just be sure, I guess.

   (b) I'm torn on whether or not to fix the octupole moment to zero, or just to initialize from rec files with Q30=0 and let it evolve from

there (it probably won't change at all, in practice, but that's not my decision!). I think I WON'T fix it...

5 May 2017

Re: Changing Q30 and seeing how that affects triaxility in Q22 - I restarted an HFODD calculation using mass-symmetric restart files. For the most part, the outputs all preserved Q30=0 (or almost zero), which is kind of what I expected. The consequential PES shows mostly the same structure as before, with the exception of that region between Q20=75 and Q20=100, where the system must decide whether it will tunnel through the barrier to achieve symmetric fission, or whether to follow the valley towards asymmetric fission. It seems to be energetically favorable in this region for the system to choose mass asymmetry (I just picked the point Q20=100, Q22=2 at "random" and the EHFB difference was something like 2 MeV). I'm thinking the right thing to do now, actually, might be to do another calculation, where this time restart files are chosen from the fission valley. The way I did it the first time was to start everything from the entire line Q30=4, thinking that the system would be free to explore all values of Q30. I was partly right, especially for the values below Q20=75. But once it got over that barrier, Q30=4 is actually *inside* the barrier, leading toward symmetric fission. Those higher Q20 deformations never got the chance to explore higher Q30 values because there was a barrier in the way.

In other news, I discovered a bug in my Python PES plotting script. Whenever I would plot a multipole moment on the z-axis, I would have it subtract the minimum value from all of the other values, in essence shifting the origin from 0 to `Z0_min`. That can be helpful for getting a sense of the relationships between points, but not if you want to compare against an absolute scale. Case in point: for detecting scission, you need to know the location of points for which $Q_N \approx 0$. But up to (and including) today, my PES didn't contain any points with $Q_N < 8$. So when I thought I was approaching scission, I was actually approaching $Q_N = 8$. I've still got quite a way to go before I reach $Q_N = 0$!

8 May 2017

It looks to me (without a full set of converged results) that triaxiality plays a more important role in the mass-symmetric fission case, whereas in the mass-asymmetric case the system favors axiality. So you need one of those symmetries to be broken, but not both? Interesting...

19 May 2017

I was just re-reading Nicolas' first fission review article (PRC 90.054305.2014), trying to find some significance to the fact that ^{294}Og reaches its outer turning point so quickly and yet takes a *reeeally* long time to fission (in terms of multipole deformations, that is, and compared to ^{240}Pu; its half-life is actually probably pretty short because it reaches that outer turning point really quickly

and it doesn't have a second minimum fission isomer like ^{240}Pu). And I noticed a comment about cluster radioactivity in actinides, which upon further investigation is sort of an intermediate between fission and alpha decay, and which tends to leave behind ^{208}Pb or something close to it. For ^{294}Og, that would leave the additional daughter fragment ^{86}Kr, which is semi-magic with 50 neutrons. And HOLY CRAP that's exactly what we get! The only region of my PES that seems to have scissioned so far is exactly that! It's a Christmas miracle!

9 June 2017

I still won't get to this for a while yet, but I wonder about the formula for the half-life. I know that's one of the most sensitive fission observables. Particular I wonder about the number of assaults on the fission barrier. That's typically taken to correspond to the vibrational energy 1MeV, which is probably a decent estimate but still something that could change depending on your nucleus and configuration. So I would suggest - at the very least - playing around with this number, just to check the robustness of your results before sending them in.