Lessons learned at LLNL

Zachary Matheson

April 10, 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.

1 March 2017

Impact of basis deformation on EHFB

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 asymettric 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:

$$77.00\ 1.00\ 2.00\ -1801.146627\ 0.000000\ 0.598205\ 0.598233\ 0.000000\ 0.000000\ 0.598219 \tag{1}$$

whereas Jhilam is getting:

$$77.00\ 1.00\ 2.00\ -1801.183\ 0.010463\ 0.033608\ 0.000842\ -0.001305\ -0.000412\ 0.000152 \tag{2}$$

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. $\Rightarrow$ 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 $E_{HFB} = -1801.173695 MeV$

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.

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.

...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
```