pbqff tutorial

Brent R. Westbrook

January 5, 2021

Contents

1 Introduction

pbqff is a program for making the generation of quartic force fields (QFFs) as easy as possible. Its name was inspired by the C&E News article "As DFT matures, will it become a push-button technology?" by Sam Lemonick. The consensus among the Tschumper and Fortenberry groups was that "DFT" or computational chemistry more generally could never be fully "push-button" because of the need for expert analysis of the data. Using pbqff is no different; you still need to be that expert to make sure the output is reasonable and to write it up into a paper. However, pbqff should make the experience of actually running the QFF as painless as possible. If you have ever run a QFF "by-hand," you know the complexity of the procedure. These tedious and repetitive steps lend themselves perfectly to being done by the computer, freeing you to think about something more interesting. If you haven't, I will prepare a separate document giving a general QFF tutorial. To understand what pbqff is doing, you should have a good grasp on what *you* would be doing if you had to do its job by hand. It's not strictly necessary to understand pbqff though, and the goal for me was really to make the implementation details irrelevant to the end user. If you really want to understand the entirety of pbqff and all its implementation details, you should just read the source code, which can be found at github.com/ntBre/pbqff. Assuming you want to get to work, I hope this tutorial document will give you all the information you could possibly want for actually using pbqff effectively. I will also prepare a separate man page to use for quicker reference once you have a basic understanding.

2 Mental Framework

2.1 Internal Coordinates

As I alluded to in the introduction, the idea behind pbqff is to follow the exact steps a person would by hand in running a QFF, but to do so quickly and exactly via the computer. As it stands, I think of pbqff as "replacing" an undergraduate researcher because we usually give undergraduates all of the template input files they need. pbqff requires that you give it template intder, spectro, anpass, and Molpro files for a symmetry internal coordinate (SIC) run, which it can then modify as needed. This modification assumes that the template input files correspond to molecules with the same coordinate system as the target molecule, so the automation at that stage is fairly limited. Graduate students (and advanced undergraduates) usually learn how to generate or more substantially modify intder, anpass, and spectro files, so once pbqff can do that, it will have "replaced" them as well. Also as mentioned above, these replacements are not total, as it still requires some thought to evaluate the program output and to come up with new molecules for input. In a section on mental models, think of the replacing as a rough model for how much pbqff can do.

2.2 Cartesian Coordinates

The deal for Cartesian (or XYZ) coordinate QFFs is a little better because intder and anpass are not used. Consequently, you only need template spectro and Molpro files. This is also true for gradient calculations since they are also based on Cartesian coordinates. Even more fortunately, I think spectro input files are the easiest to generate, so the earliest full automation will come to these types of calculations as well. If you are familiar with our traditional SIC QFFs, you may be surprised that intder and anpass are not used. Intder is used for coordinate transformations from internal coordinates to Cartesian coordinates and back again, so working directly in Cartesian coordinates from the start obviates any need for it. Anpass is used for least-squares fitting of the potential energy surface, whence we gather the force constants. For the Cartesian QFF, pbqff numerically computes each force constant directly, so there is no need to do a fitting. These facts should not change your mental model of the whole procedure, however. The big idea is still to take a geometry, displace its atoms, compute single-point energies at each of the displaced points, use those energies to obtain force constants, and then jam those force constants into spectro to get spectroscopic data. In Cartesian coordinates you just get to take shortcuts in the displacement and force constant steps. Like with SICs, these are the exact steps you would take in doing a Cartesian QFF by hand, but there are so many points involved in Cartesian QFFs that we never do them by hand.

2.3 Job Submission

The other slightly tricky part about pbqff if you are more used to our conventional QFF schemes is that it does not submit, or even generate input files for, all of the jobs at once. Doing everything at once is obviously convenient when you are doing it by hand because you can submit all of the computations to the supercomputer and then go do something else. Instead, pbqff watches the running jobs, continuously checking for ones that finish, and writes and submits more calculations as the old ones finish. It also deletes the files associated with the old jobs to save space. Space is not typically a major concern with SIC QFFs since even the largest systems we have worked with have only 10000 or so points. In contrast, even water has half that many points in Cartesian coordinates, and larger molecules that we've actually run have had over 200000. Eventually we would like to extend this to millions of points, so keeping all of the files around is not really feasible. Again, you should be able to picture this in the same way as doing it by hand, but it's easier to convince the computer to sit and constantly refresh qstat than to make a human do it.

A final wrinkle is the use of GNU parallel to reduce the number of individual PBS jobs that need to be submitted. parallel is basically a miniature version of the whole queuing system in that it takes a list of jobs to run and dispatches them as resources are available. While this is yet another departure from how you would likely submit these jobs by hand, you can certainly use parallel when submitting jobs by hand. It should also have no bearing on your interaction with the program, but if you are curious why there are so few jobs in your queue this is the explanation.

3 Program Input

In addition to the template files addressed in the previous and upcoming sections, pbqff takes its own input file. This section will walk through all of the accepted input options and offer some example inputs for the various supported calculation types.

Unlike some programs, the input directives are totally order-agnostic. As long as they are in the input file, they will be recognized. Available keywords are shown in Table 1. The case of the keywords is ignored by the parser, but each keyword must be followed by an equals sign (=). Comments can only start at the beginning of a line, by including "#" as the first character on the line. Geometry input is unique in that it expects to look like `geometry={...}`, beginning with an equals sign and opening curly brace and terminating with a closing curly brace. As shown in the options for the GeomType, the geometry can be input as either a Cartesian or XYZ geometry or a Z-matrix. If the Cartesian coordinates are used, the program expects a fully-formed XYZ geometry, including the number of atoms line and the comment line. These lines are skipped, so it's not important that they be accurate, but they must be present.

The following examples can be found in my home area on hpcwoods under Programs/pbqff/examples. The files embedded in this document should be synced with the ones found there, and they all should be tested to run correctly.

3.1 SIC Example

Below is an example input file for an SIC run. Since much of the information for the SICs is found in the other template input files, the pbqff file is about as minimal as it gets. Based on the defaults given in Table 1, even the `program`, `queue`, and `geomType` lines are technically redundant, but it's nice to include some of these for future reference. Example:

Listing 1: pbqff input for an SIC calculation

```
program=molpro
queueType=maple
geomType=zmat
geometry={
O
H 1 oh
H 1 oh 2 hoh
oh=1.0 ANG
hoh=109.5 DEG
}
intder=/ddn/home6/r2533/programs/intder/Intder2005.x
```

```
anpass=/ddn/home6/r2533/programs/anpass/anpass_cerebro.x
spectro=/ddn/home6/r2533/programs/spec3jm.ifort-O0.static.x
```

One thing to note is that there is no brace between the Z-matrix itself and the values of the parameters, as you might expect if you are used to Molpro. Just remember that this is the format expected by pbqff, and it will convert it to the Molpro format when necessary.

3.2 Cartesian Example

The Cartesian example is a bit more involved since you have to specify some non-default values, and you can't rely on the intder file for the step sizes or derivative level. Again, many of these options are technically optional since they are the same as the defaults, but it's nice to be explicit when possible. Example:

Listing 2: pbqff input for a Cartesian calculation

```
program=gocart
queueType=maple
geomType=xyz
geometry={
 3
 Comment
 H          0.0000000000       0.7574590974       0.5217905143
 O          0.0000000000       0.0000000000      -0.0657441568
 H          0.0000000000      -0.7574590974       0.5217905143
}
delta=0.005
deltas=1:0.075,4:0.075,7:0.075
flags=noopt
deriv=4
sleepint=5
joblimit=8000
chunksize=64
numjobs=8
intder=/ddn/home6/r2533/programs/intder/Intder2005.x
anpass=/ddn/home6/r2533/programs/anpass/anpass_cerebro.x
spectro=/ddn/home6/r2533/programs/spec3jm.ifort-O0.static.x
```

Of note here is a first example of an XYZ geometry. You can see that the number of atoms line and comment are present. Alignment and spacing are not important, so you can freely paste the geometry in however you think looks best. Another important aspect of this example is the demonstration of the `deltas` input. In this example, all of the steps in the x direction will be larger (of size 0.075 Å), while the rest will be 0.005 Å.

3.3 Gradient Example

The gradient version is virtually identical to the Cartesian version, except that the `program` is specified as `grad`. You may also notice that the comment line says that the reference energy was computed at the DF-CCSD(T)-F12 level rather than regular CCSD(T)-F12. Molpro only has analytic gradients for density-fitted coupled cluster, so if you want to use gradients keep that in mind. Based on some forthcoming research from our group, you probably don't want to use gradients though. Example:

Listing 3: pbqff input for a gradient calculation

```
program=grad
queueType=maple
geomType=xyz
geometry={
     3
 DF-CCSD(T)-F12/CC-PVTZ-F12   ENERGY=-76.36827708
 H              0.0000000000          0.7578204038           0.5219210812
 O              0.0000000000         -0.0000000012          -0.0660052896
 H              0.0000000000         -0.7578204026           0.5219210802
}
flags=noopt
deriv=4
joblimit=8000
chunksize=64
intder=/ddn/home6/r2533/programs/intder/Intder2005.x
anpass=/ddn/home6/r2533/programs/anpass/anpass_cerebro.x
spectro=/ddn/home6/r2533/programs/spec3jm.ifort-O0.static.x
```

4 Template Files

As mentioned previously, pbqff still requires you to give it template input files for Molpro and spectro, in the case of Cartesian QFFs, and for Molpro, spectro, intder, and anpass for SICs. It should probably take template PBS files as well, but the flexibility therein has been less necessary so far. The Molpro file is the most important because it is required to run any computations, so we will start with that one. Next, we will look at the spectro input file since that is common to both Cartesian and SIC QFFs. However, the spectro input file is not technically required for the program to run usefully. The final real products of pbqff are the force constant files that are fed into spectro. The program will print an error if it can't find the spectro input file, but you can easily run spectro yourself (or with my gspectro program) afterward. For SICs both the intder and anpass files are strictly required by the program to even start running.

4.1 Molpro

The basic Molpro template file is very straightforward. The only difference indicating that it is a template is the lack of a closing brace in the geometry section. Other than that, you can include any option accepted by Molpro and it will be transmitted directly into each of the single point energy input files. The `optg` line is optional if you are going to use the `noopt` flag anyway, but if you use that flag the program will automatically remove that line from the input.

Listing 4: Basic Molpro template file example

```
memory,1,g

gthresh,energy=1.d-12,zero=1.d-22,oneint=1.d-22,twoint=1.d-22;
gthresh,optgrad=1.d-8,optstep=1.d-8;
nocompress;

geometry={
basis={
default,cc-pVTZ-f12
}
set,charge=0
set,spin=0
hf,accuracy=16,energy=1.0d-10
{CCSD(T)-F12,thrden=1.0d-8,thrvar=1.0d-10}
{optg,grms=1.d-8,srms=1.d-8}
```

For a gradient calculation, a couple minor changes are required, as shown in the listing below. In this case, the gradient is what needs to be extracted from the output file, not the energy. Molpro prints the energy with sufficient precision by default, but it only prints the gradient to 8 decimal places. As a result, we need to use the `varsav` directive to save the gradients in each direction to variables, and then the `show` directive to print each of them in a 20.15 format. The 20 is actually irrelevant, and 12 decimal places should be sufficient, but you might as well request a little extra precision just to be safe. Additionally, as touched on before, Molpro only has analytic energy gradients for DF-CCSD(T), so make sure to use both DF-HF and DF-CCSD(T) for the energy calculations.

Listing 5: Molpro template file example for a gradient calculation

```
memory,1,g

nocompress;

geometry={
basis={
default,cc-pVTZ-F12
}
set,charge=0
```

```
set,spin=0
df-hf
DF-CCSD(T)-F12
forces,varsav
show[f20.15],gradx
show[f20.15],grady
show[f20.15],gradz
```

No additional convergence criteria are specified in this gradient listing, so make sure you include those if you are doing publication-quality research. The typical criteria used in our papers are demonstrated in the SIC listing.

4.2 spectro

The spectro input file should look like the listing below and must be named spectro.in. The necessary sections are the initial `# SPECTRO ####` section, which includes the input directives; the `# GEOM ####` section, which has the geometry; the `# WEIGHT ####` section, which obviously has the masses; and the `# CURVIL ####` section, which contains the curvilinear coordinate system to be used for the rotational and geometry analysis. No resonance information should be included, and correspondingly all of the input directives that enable resonance accounting should be set to zero. I plan to have the program enforce this in the future, but for now you need to make sure of it. The geometry does not need to have the right geometry or even the same atoms as the input molecule because the program will rewrite the file with the appropriate values. However, the order of the atoms should match the order present in the input geometry. If you are given a template spectro file, I would advise changing the geometry in the pbqff input file to match the spectro order. If instead you generate the spectro file yourself, just make sure it lines up with the input file. The weight section does not need to be correct unless you enable its use in the input directives, as is usually the case with spectro. It may not even have to be present, but that is a spectro question, not a pbqff question.

Listing 6: Example spectro input file for water

```
# SPECTRO ##########################################
    1    1    2    1    0    0    0    4    0    0   00    0    0    0    0
    0    0    0    0    0    1    0    0    0    0    0    0    0    0    0
# GEOM #########################################
   3    1
 1.00       0.0000000000       1.4320729117       0.9862878311
 8.00       0.0000000000      -0.0000000023      -0.1247319111
 1.00       0.0000000000      -1.4320729095       0.9862878292
# WEIGHT ######
    3
    1      1.972070
    2     16.003074
    3      1.972070
# CURVIL ###########################################
    1    2
    2    3
    2    1    3
```

Apologies for the size, but those integer input parameters take up a lot of room! Just like gspectro, when pbqff runs spectro, it actually runs it twice to incorporate the resonances on the second pass. That's why it's safe to leave out the resonance information in the template file. The input directives will also be updated to reflect the necessary resonances, so taking your results from spectro2.out will give the correct values. However, pbqff cannot yet handle degenerate modes, at least in an explicit sense, so if you need to include a `# DEGMODE ####` section, it might work, but you should double-check that part by hand. If you can confirm one way or the other, please let me know.

4.3 intder

The listing below shows the top portion of a template intder file, which must be named intder.in, but it shouldn't be too interesting. pbqff just expects a normal intder file for generating points, but it will take the geometry you leave in there and try to match the optimized geometry to it by exchanging rows and columns. This typically works very well if the intder file is from a molecule with the same symmetry. If you encounter an error saying "transform failed" this coordinate transform is what it is referring to. In that case, jump to the Troubleshooting section for a discussion of how to address that using a couple of command line flags. Otherwise, just feed it a full intder file, either one you obtained from someone else or a previous calculation or one you generated using taylor.py and its ilk.

Listing 7: Partial example intder input file for water

```
# INTDER ##########################
    3    3    3    0    0    3    0    0    0    1    0    0    0    1    1    0
STRE     1    2
STRE     2    3
BEND     1    2    3
    1   1   1.000000000   2    1.000000000
    2   3   1.000000000
    3   1   1.000000000   2   -1.000000000
    0
      0.000000000000      1.431390244079      0.986041163966
      0.000000000000      0.000000000000     -0.124238450265
      0.000000000000     -1.431390244079      0.986041163966
DISP   69
    1        -0.0050000000
    2        -0.0050000000
    3        -0.0100000000
    0
```

4.4 anpass

Like that of intder, the anpass handling is very straightforward. pbqff will parse the anpass file to make sure it prints the correct number format and then replace the relative energies at the ends of the lines in that section. It does not do any validation of the step sizes or of the unknown blocks at the end, so make sure you use a well-formed anpass input file that corresponds to your intder file. Similar to spectro, pbqff will run anpass once to find a new stationary point, and then run again at that stationary point, just like you would do with the whole "long-line" thing by hand. The file must be named anpass.in, which may require a quick rename if you copy over an existing anpass1.in file from another project. See below for an abbreviated listing of the top of the file, just to be consistent with the other sections. Hopefully you know what it's supposed to look like. The title only agrees with the contents of the file because I got this from someone else. Rest assured that it has no effect on the calculation.

Listing 8: Partial example anpass input file for water

```
!INPUT
TITLE
 H2O 2A1 F12-TZ
INDEPENDENT VARIABLES
   3
DATA POINTS
  69    -2
(3F12.8,f20.12)
 -0.00500000 -0.00500000 -0.01000000         0.000128387078
 -0.00500000 -0.00500000  0.00000000         0.000027809414
 -0.00500000 -0.00500000  0.01000000         0.000128387078
 -0.00500000 -0.01000000  0.00000000         0.000035977201
```

5 Running the Program

Now that you have all of the requisite input files for any type of calculation, you are ready to run the program! In this section I will show the basic input command, along with an explanation of the parts that aren't as basic as you would expect, and also describe each of the flags you can use to modify the program's behavior. The most basic form of the command is

```
$ pbqff infile.in
```

assuming that you have added the executable to your path under the name pbqff. Since most of the interesting jobs you will be running will take a while, you likely want to run

```
$ pbqff infile.in &
```

instead, to put the job in the background. However, as you may or may not know, logging out of your terminal session, i.e. exiting your ssh connection, sends SIGHUP to all of the programs you had running. This means in effect that any background task you started is killed when you log out. To combat this, you need to tell the shell not to send SIGHUP using the `disown` command with its `-h` flag. With this in mind, the command I use to run pbqff is

```
$ pbqff infile.in & disown -h
```

Assuming all goes well, this should immediately create infile.out and infile.err. Currently, the .out file does not have too much information in it, but the .err file reports the progress of your calculation once the points start running. To follow these files, I typically use the command

```
$ tail -F infile.out infile.err
```

where the `-F` flag is like the lowercase version, but it will pick up a file if it is newly created. This is not that important for the out and err files, but if you also want to follow either the optimization file or the reference energy file in the case of a Cartesian run, it can come in handy. These two files are `opt/opt.out` and `pts/inp/ref.out`, respectively.

5.1 Directory Structure

These filenames give a good lead-in to the directory structure created by the program. In the case of SIC QFFs, I used my conventional directory structure that looks like

```
.
├── freq
├── freqs
├── opt
└── pts
    └── inp
```

where opt is used for the geometry optimization, freq is used for the Molpro harmonic frequency calculation, pts is used to run intder, while pts/inp holds the actual input files, and freqs is where I run intder, anpass, and spectro to get the anharmonic data. Cartesian QFFs are more streamlined, so everything there is either run in the base directory where pbqff is run or in pts/inp. In both cases, you generally should not need to descend into the created directories unless you are doing some heavy troubleshooting. The only real exception is the SIC freqs directory, which will contain your final spectro2.out file. Another less important exception could be the SIC freq directory if you need to compare the harmonic frequencies to those from intder and spectro in the freqs directory.

5.2 Flags

You can already access the main functions of pbqff and control its behavior to some extent using the input file. However, there are also some additional command line flags that you can use to further modify the behavior. The most important flag is the `-h` or `-help` flag because it will list the help for all of the other flags. I also plan to replicate much of the fundamentals of this section in the man page, but here I hope to give some examples and explanations of the flags in addition to their basic usages. Like the keywords, the flags can be found in Table 2.

A couple important things to mention are the artifacts of the Go flag package. Unlike most (if not all) Unix utilities, the Go flag package does not support joining multiple flags together as in

```
$ ls -lt
```

which is equivalent to

```
$ ls -l -t
```

This means that when you want to resume a calculation from a checkpoint file, which requires that you assent to overwriting portions of the inp directory, you have to pass both the `-c` and `-o` flags as in

```
$ pbqff -o -c infile.in & disown -h
```

What this does afford, however, is the ability to use only a single dash for long options or equivalently to use flags longer than single letters. That's how and why many of the flags are more descriptive than they might have been otherwise. If you are already in the habit of using double dashes for long options, it will also handle that. Similarly, you can join arguments to flags using an equals sign as in

```
$ pbqff -irdy="H O H"
```

or without one as in

```
$ pbqff -irdy "H O H"
```

but you cannot join the argument directly to the flag since it could be parsed as a single-dash long option. For the currently-implemented flags, this isn't really a problem, but I figured it doesn't hurt to point out.

5.3 Checkpoints

Since most interesting calculations are going to take a long time to run, the program comes equipped with a checkpointing system to save the progress of a run as you go. The JSON files created while the program is running serve as these checkpoints, at least for Cartesian QFFs. Checkpoints are not currently implemented for SICs, so if you are particularly worried about an SIC calculation, I would suggest using the `nodel` flag since it's more straightforward to resume an SIC as long as all of the output files are present. When resuming from a checkpoint, the program behaves mostly as if it were starting from scratch, but instead of recalculating all of the energies, it will load as many as possible from the JSON files. I already showed an example of resuming from a checkpoint above, but just to keep it explicit here it is again:

```
$ pbqff -o -c infile.in & disown -h
```

The `-o` flag is required since you are going to have to overwrite some of the previous files, and the `-c` flag triggers the checkpoint loading. You should run this command in the directory where you initially ran pbqff. This should be fairly obvious since that's where your input file is.

When might you need a checkpoint? The most likely scenario on Maple is if your job runs out of CPU time. If you run the command

```
$ ulimit -aH
```

you will get a list of your hard resource limits on Maple. At the time of this writing, the CPU time limit is 72000 seconds or 20 hours. Of course, CPU seconds are not equivalent to wall seconds, so this does not mean you can only run jobs that finish within 20 hours. What it does mean though is that if pbqff exceeds that usage it will be killed automatically. The easiest way to identify that this has happened is by checking the modification time of the err file since it should be updated every second. You can also run the command

```
$ ps aux | grep pbqff
```

to check for a running process with the name pbqff. Alternatively you can grep for your username or your username and pbqff to further narrow the results. pbqff will print its CPU usage at every checkpoint interval, but if you want a more regular way to check, you can use the command

```
$ ps axo pid,user,comm,time | grep pbqff
```

The important part of this command is the `time` portion, but the other fields requested are useful in case you want to kill the program (pid), and for sorting based on username and process name (user and comm). If you check this and see your pbqff instance approaching 20 hours, expect to have to restart it soon. Since pbqff doesn't know when it will be killed, it will likely have started many jobs already that will only be in the way of your new instance. As a result, you should kill all of the jobs in your queue using a command like

```
$ qselect -u $USER | xargs qdel
```

The `qselect` command is used to select jobs in the queue based on the arguments. The `-u` flag selects by user, and the environment variable `USER` is your username. You can also select by name using the `-N` flag. This can be more useful if you have jobs other than those associated with pbqff in the queue and you don't want to kill them. The name of the single point energy jobs submitted by pbqff is "pts," giving the command

```
$ qselect -u $USER -N pts | xargs qdel
```

if you want to be a bit more selective.

Another common issue that will create a huge error dump in the err file is using too many operating system threads. The number of threads available is much more limited than even the CPU time, so if you try running more than one instance of pbqff at a time, both will likely crash with this error. The same can be true of any intensive action. As a result, while pbqff is running, you should be careful to limit your resource usage on Maple. Obviously any interactions with the actual queue need to be done on Maple, but if you need to do some kind of file scripting or even just tailing the pbqff output and error files, you can do that on Woods instead.

6 Troubleshooting

The most common problems have already been addressed in the previous section since they will require a restart from the checkpoints. In this section, I will describe other potential problems you may encounter and offer advice on how to solve them. If you run into any problems not covered here, please reach out to me and I will add it to this section.

6.1 Transform Failed

If you get this error message in an SIC QFF, it means that the program was unable to match the optimized geometry of your molecule to the pattern it identified in the intder template file. Assuming you have an actually usable intder template, this can be caused by slight symmetry differences in the two geometries that are not ignored by the fairly simple pattern matching algorithm. You may be able to recognize this and modify the template slightly to allow the program to match the pattern automatically, but this is likely difficult to do without knowing the implementation details of the matcher. An easier option is to manually take the optimized geometry and place it in the intder template file as if you were going to run the QFF by hand. From there you can tell the program that the intder file is ready by using the `-irdy` flag. The string argument to this flag is a space-delimited list of the atoms in the geometry. Usually the program gets the order of the atoms from the optimized geometry and keeps track of their order through the pattern matching. Without that, it needs you to tell it which atom is which. As shown above, the full command to restart from this point is

```
$ pbqff -o -irdy "H O H" & disown -h
```

where the overwrite flag is necessary if you have already run the program in the same directory. You will also probably want to add the line `flags=noopt` to your input file since you will have obviously already optimized the geometry to put it in the intder file. That's not necessary for the program to function, but it will save you the time of waiting for it to optimize again.

Table 1: Keywords

Keyword	Type: Available values	Default	Description
QueueType	String: sequoia, maple	maple	Specify which queue to target. Basically choose the internal PBS template to use for submitting jobs.
Program	String: cccr, cart, gocart, grad, molpro	molpro	Specify the subprogram to use. cccr is for CcCR SICs; cart or gocart is for Cartesians; grad is for gradients; and molpro is for normal SICs
Queue	String: workq, r410	Both	Select the queue name to use. Default behavior is to use whatever is available.
Delta	Float: any	0.005	Specify the step size to use for the geometry displacements of a Cartesian QFF.
Deltas	Int:Float pairs: any	0.005	Specify step sizes for individual coordinates in a Cartesian QFF. Format is index:value, where index starts from 1. Pairs are separated by commas. Any indices not specified take their default value from Delta.
GeomType	String: xyz, zmat	zmat	Specify the type of geometry. Currently this is only used to compute the number of coordinates.
Flags	String: noopt	None	Specify command line flags in the input file. Only noopt is currently supported.
Deriv	Int: 2, 3, 4	4	Specify the derivative level to be computed for a Cartesian QFF.
JobLimit	Int: any	1000	Specify the maximum number of jobs to have submitted at once.
ChunkSize	Int: any	64	Specify the number of jobs to submit in a GNU parallel "chunk." This also determines how often files are deleted.
CheckInt	String/Int: no/any	100	Specify the checkpoint interval. An input of "no" disables checkpointing, while an integer value sets the interval.
SleepInt	Int: any	1	Specify the interval at which to poll running jobs in seconds.
NumJobs	Int: any	8	Specify the number of jobs to run per GNU parallel job. Each parallel job requests 64gb of memory, so NumJobs should evenly divide 64.
IntderCmd	String: any	None	Specify the path to the intder executable for SIC QFFs.
AnpassCmd	String: any	None	Specify the path to the anpass executable for SIC QFFs.
SpectroCmd	String: any	None	Specify the path to the spectro executable.
Geometry	String Block: any	None	Specify the geometry. See the examples for details.

Table 2: Command line flags

Flag	Type	Default	Description
c	Bool	False	Resume from checkpoint; requires the o flag to overwrite existing directory.
count	Bool	False	Read the input file for a Cartesian QFF, print the number of calculations needed, and exit.
cpuprofile	String	None	Write a CPU profile to the supplied filename.
debug	Bool	False	Print additional information for debugging purposes.
fmt	Bool	False	Parse existing pts output files and print them in anpass format
freqs	Bool	False	Start an SIC QFF from running anpass on the pts output. This requires that all of the points have been preserved.
irdy	String	None	Ignore the geometry in the input file and use the intder.in file as is. This implies that you should use noopt. The string is a space-delimited list of atomic symbols to pair with the geometry.
nodel	Bool	False	Preserve output files instead of deleting them after use.
o	Bool	False	Allow existing directories created by the program to be overwritten.
pts	Bool	False	Resume an SIC QFF by generating the points from the optimized geometry in the opt directory.
r	Bool	False	Read the reference energy for a Cartesian QFF from an existing pts/inp/ref.out file.