A gentle introduction to computational chemistry and density functional theory

Written by Nathan M. Lui (2023)
With contributions from Dr. Ryan A. Woltornist (2021)
and many others; see the full list of contributors in the GitHub repo

This booklet is updated periodically as a offline version of the material; nevertheless the website will always have the most up to date version of the course.
Last updated: 12 Jan 2021

The Short Course

The *Short Course* is designed as a primer for advanced undergraduates and beginning graduate students. It is intended to give the completely inexperienced reader a step-by-step guide to running electronic structure calculations on the AS-CHEM computing cluster at Cornell University, but it is our hope that these instructions are easily generalizable to other computing clusters. The Short Course's main computational engine is `Gaussian16` since it was originally designed for Collum group members, however a *fully open-source* edition (utilizing `Psi4`) is currently being written.

Software
Linux basics
My first script
`SLURM` basics
My first `SLURM` job
The `Gaussian` input file
My first `Gaussian` job
Understanding the `Gaussian` output file
Putting it all together: calculating cyclohexane A-values

The Long Course

The Long Course is a work in progress!

The *Long Course* is a set of more advanced topics in scripting and computational chemistry. It was written for those who have, or would like to, incorporate more advanced calculations/models into their research. The topics start by streamlining computational workflow (with `bash` scripting) and then progress towards the more under-the-hood options of computational engines.

Using `bash` to streamline computational workflow and data processing
So what exactly is an optimization?
Cost efficiency: Selecting basis sets and functionals without going overboard
Using custom basis sets and effective core potentials
What is a transition state?
The art of finding transition structures
The implicit/explicit solvation war
Electronic structure theory

Problems

My least favorite math teacher would always say that the only way to learn calculus is to solve lots of calculus problems. This is a collection of case studies and practice problems that you can use to try your own hand at computational chemistry. Let me know if there's anything else you'd like to see here!

Cyclohexane A-values (from the Short Course)
The Smelly Dimer Problem
E-Z isomers of 3-(4-nitrophenyl)but-2-en-2-yl triflate

Resources

The Code Repo: Exercises and Problems

g16 “cheat codes” (routing line templates)

A collection of papers/resources I've amassed over the years.

Contributions and Corrections

This course is a living, breathing work-in-progress so if you spot any typos or if there are topics you'd like to add (to see added) to the course check out the GitHub repo or email me!

Supplemental Readings

The *International Journal of Quantum Chemistry* has published an excellent series of tutorial reviews for novices and professionals alike. I highly recommend looking through them. Below are a few you may find particularly helpful.

Fourteen Easy Lessons in Density Functional Theory
by John P. Perdew and Adrienn Ruzsinszky
Int. J. Quantum Chem. **2012**, *110* (15), 2801
DFT in a nutshell
by Kieron Burke and Lucas O. Wagner
Int. J. Quantum Chem. **2013**, 113 (2), 96
The devil in the details: A tutorial review on some undervalued aspects of density functional theory calculations by Pierpaolo Morgante and Roberto Peverati
Int. J. Quantum Chem. **2020**, *120* (18), e26332

The Short course

1. Software

The programs engaged in this document were chosen as favorites (or in some cases, relics) of Collum group members. There is a large emphasis on those that are open source and free to use (**listed in bold below**), however there are a plethora of alternatives available across the internet and the following list should be taken as a starting point, not a superlative.

You'll need the following programs for this course

Cornell VPN client (for Cornell students who intend to use the AS-CHEM cluster)
SFTP client (e.g. **FileZilla**, **putty**)
Molecular modeling program (e.g. GaussView, **Avogadro**, SAMSON, etc...)
Command-line terminal (e.g. the built in **terminal** on Linux/Mac OS or **cygwin** if you're using a Windows machine)
Command-line text editor (e.g. **Vim**, **Vi**, **Nano**; helpful, but not required)
A text editor (e.g. **Brackets**, Sublime, **VSCode**, **Notepad++**, etc...)
3D-rendering/ray-tracing software, optional (e.g. **CYLView**, SAMSON, etc)

FileZilla

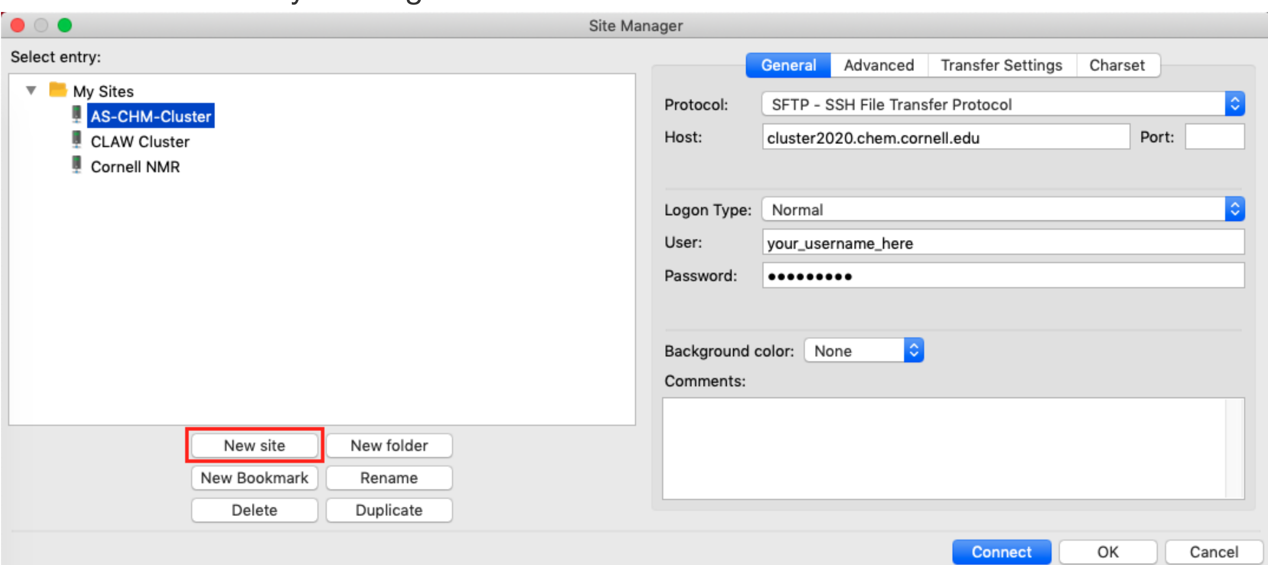
`FileZilla` is an SFTP (**s**ecure **f**ile **t**ransfer **p**rotocol) client that we'll use to move files between our own computers and the cluster; we'll have to do this since we can't just wander over to the Baker server room and stick a USB drive into the stacks every time we want our data.

Setup

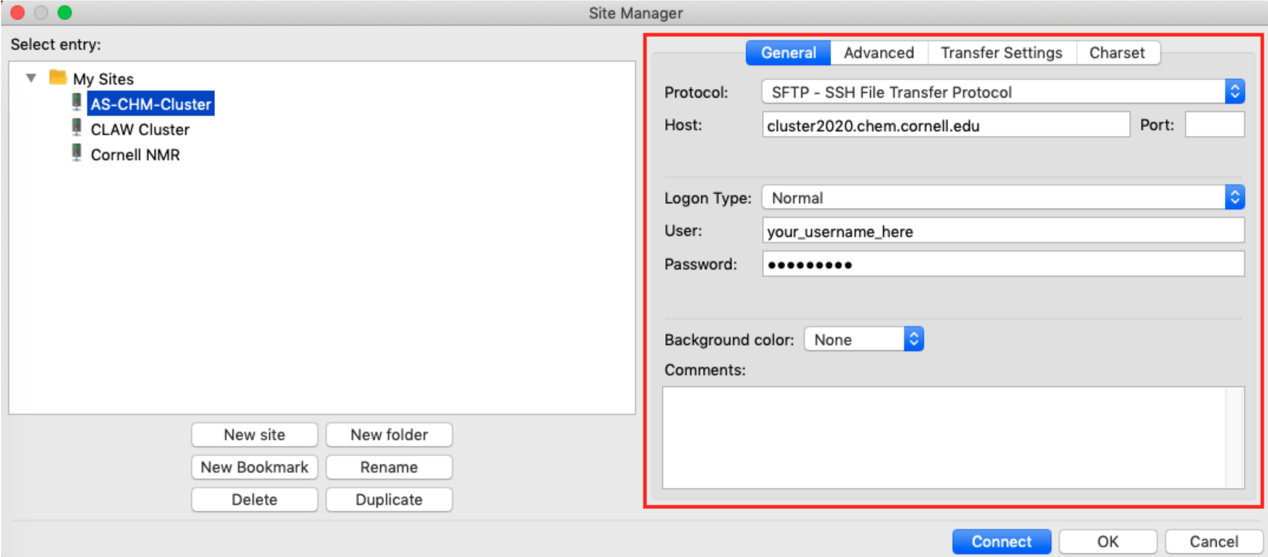
> If you haven't talked to your system admin to set yourself up a cluster account then stop here and get that done first. You'll need a fully set up account (on our cluster or your own) to get the most out of this course.

These setup instructions are for the AS-CHEM cluster only, if you're not trying to connect to this cluster, contact your system administrator for instructions on how to transfer files to/from your own system.

0. Install and connect to the Cornell VPN
1. Download the latest `FileZilla` release. Follow these instructions to install it.
2. Start FileZilla and open the site manager by going to File > Site Manager.
3. Add a new connection by clicking "New Site"



4. Configure the settings for the new site as shown:



- Protocol: `SFTP - SSH File Transfer Protocol` - Host: `cluster2020.chem.cornell.edu` - Logon type: `Normal` or `Ask for password` if you want to be asked every time - User: Your cluster logon (typically your netID) - Password: If you selected “Normal” for logon type then enter your cluster password. If “Ask for password” then you will be prompted for your password every time you connect.

5. Click `Connect` to connect to the cluster and make sure you have everything set up correctly. Your cluster home directory should show up on the right.

To reconnect in the future just go back to the site manager, select the cluster site and click `Connect`. Remember that you need to be connected to the Cornell VPN to access the cluster.

Command-line terminals

If you are working on a Mac or Linux machine feel free to skip this part. Linux distributions and MacOS come preinstalled with a command-line terminal.

Technically, Windows machines come with `PowerShell` (which is its own shell and scripting language (two things you'll learn about in the next lesson)), but for the sake of uniformity and since the cluster runs on CentOS, we'll use a `bash` shell. Windows users should download and install `cygwin`. **By default cygwin does not come with text editors installed** (don't ask me why) so you'll have to go through the setup.exe program to install these packages (start with `vim` and `nano`). This blog post may come in handy.

Molecular modeling programs

GaussView

Due to licensing constraints you'll need to visit ChemIT to get `GaussView` installed on your computer.

Avogardo

`Avogadro` is a free, open-source molecular editor and visualizer that can be used as an alternative to `GaussView`. It comes fully documented and has built-in compatibility with `Gaussian` input adn output. Download and install it from SourceForge.

Text Editors

Brackets

`Brackets` is an open-source, lightweight text editor that was built for web designers and front-end developers, but it has access to a the local filesystem a really helpful feature in keeping our files organized. Check out the `Brackets` Wiki to learn more (always start with the README).

VSCode

`Visual Studio Code` is what's known as an IDE, for **i**ntegrated **d**evelopment **e**nvironment. It is open-source, but not lightweight. `VS Code` is super powerful, with built-in features like line-by-line

debugging, intelligent code completion, built-in terminal, expanded language support, and spell check. If you're just getting started, *this is not the text editor you're looking for*, but as you become more experienced you may find that you want more than a simple code editor. Read the docs to get started.

3D-rendering and ray-tracing

CYLView

`CYLView` is a free molecular visualizer currently in development by Claud Y. Legault at the University of Sherbrook (Canada). Invaluable for drawing publication-quality chemical structures from computational output. Both versions `CYLView1.0` and `CYLView20` can be downloaded here. Mac users will need to install `XQuartz` in order to use `CYLView1.0`. Its recommended that you install both versions as `CYLView20` is still pretty barebones.

2. Linux Basics

The AS-CHEM cluster (like many other high-performance computing clusters) runs on a Linux-based operating system called CentOS so navigating its filesystem requires some knowledge of the Unix shell. As this guide is written for those without much computer experience, the more experienced reader may feel free to skip this section.

What is the shell?

The *shell* is a program that exposes a computer's operating system to a user or another program.[1] It is not the same as the program with which will interact with it: the *terminal*; however, since the terminal is the main mode of communication with a shell, you commonly see these terms used interchangeably.

There are many different shells; the most common, and the one we will use here, is the bourne again shell or `bash`, but others, like `zsh`, `tcsh`, and `csh`, do and can also be used here with slight modifications.

> `bash` is the command-line *shell*, but it is also the name of the accompanying *scripting language*. So in this course we use the `bash` program to send commands to the operating system using the `bash` language.

Navigating the filesystem

When navigating a computer system via a *command-line* system you exist in a directory (imagine the little streetview guy wandering around a city, or, if you can remember it, Zork) and, unless otherwise specified, commands take action in and on the directory in which you are currently located a.k.a. the current working directory.

When you open terminal you should see something like:

```
user@computer | ~ $
```

This is called the *terminal prompt* or the *command prompt*; it displays your username and the computer that you're logged on to. The `~` is a shorthand for your home directory, but more on that in a bit. Commands are entered after the `$`.

To view the current working directory type `pwd`:

```
NathanLui@local | ~ $ pwd
/Users/NathanLui
```

Moving around

To navigate between directories use the command `cd` followed by the path of the directory you want to enter. For example, to navigate to the `Documents` folder use `cd Documents`. Notice how the `~` changes to display the path to the new directory `~/Documents`.

```
NathanLui@local | ~ $ pwd
/Users/NathanLui
NathanLui@local | ~ $ cd Documents
NathanLui@local | ~/Documents $ pwd
/Users/NathanLui/Documents
```

The terminal will understand two types of paths: relative or absolute. A directory's **absolute** path begins with `/` and describes *the exact location of the directory*. The command `pwd` returns a directory's absolute path. A directory's **relative** path describes the location of a directory *relative to the current working directory*.
In the example above the absolute path of `Documents` is `/Users/NathanLui/Documents`, but relative to our home directory the relative path is just `Documents`.

Looking around

Use the command `ls` to look inside the current working directory.

```
NathanLui@local | ~/Documents $ ls
launchCodes.txt          playGame.sh          Presentation Slides
```

Of course this doesn't tell us much about these files. So we use the flag `-l`

```
NathanLui@local | ~/Documents >>> ls -l
total 9688
-rw-r--r--@  1 NathanLui    45K Jul  6 15:19 launchCodes.txt
-rwxr-xr-x   1 NathanLui   192B Jun 29 11:25 playGame.sh
drwxr-xr-x   9 NathanLui   627B Jun  1  2021 Presentation Slides
```

This tells us more (sometimes more than we want to know) about our files and folders. On the far left of the output is the list of file's permission, or *mode of access*. These modes control what the file is allowed to do and who is allowed to do them. There are 3 access levels for any file: the owner, the group, and everyone. The 10 characters of the permissions section designate **r**ead, **w**rite, and e**x**ecute (run) access for these three groups (plus a general descriptor at the beginning).

For example, the file `launchCodes.txt` has the access descriptor `-rw-r--r--` meaning it is a regular file `-` for which the user (u) can read and write but not execute `rw-`, my group (g) can only read `r--`, and others (o) else can only read `r--`. Whereas, the file `playGame.sh` is a program, or a shell script. A shell script is run (executed) by the user, so it needs the permission code `x` to to function properly. Notice above that `playGame.sh` can be executed by the user, group, and everyone else. Sometimes you'll need to change permissions (usually, you need to give a file executable permission) and this can be done using the `chmod` command (**ch**ange **mod**e) followed by the new set of permissions. A more detailed explanation of file atributes and permissions can be found here.[2]

Making and editing files

To make or edit files in terminal you'll use one of the preinstalled text editors: `vi`, `vim`, or `nano`. My personal favorite is `vim`. If you don't have experience using a command line text editor it will take a bit of getting used to. A cheat sheet for `vim` can be found here.[3] `nano` is the command-line text editor that generally is the easiest to pick up for first-time users. You can find the full documentation[4] and cheat sheet[5] on its website. If editing files in the terminal isn't your style, you can always download your file of interest from the cluster and edit it locally and then upload it back when you're done, but this will get tedious.

To edit or create a file simply type `vim /path/to/file`. If the file already exists `vim` will open it and you can edit to your heart's content (of course, it should go without saying that text editors can only

edit text based files). If the system can't find the file `vim` will open a blank file with the path/name that you specified. `vim` will not save that file until you write it with `:w` .

Asking for help

> Help will always be given in Linux to those who ask for it.
>
> —*Harry Potter's IT teacher, probably...*

These are the essential, but very basic Unix commands to get you started. There can be quite the learning curve when transitioning from a graphical system to the command line so I'll leave you with two of the most important commands to remember when you're stuck: `man` and `apropos` .

Say you want to take a peek at the permissions of a certain file, but you can't remember the flag for the detailed output. The `man ls` command brings up the manual page for `ls` . In it you'll find detailed documentation for the command including its signature, description, options, examples, and related commands. To exit the manual page press `q` . The Linux manual is also available online.[6]

Now, that's nice if you know what command you need for, but say you want to make a new directory (folder) and you're not sure how. This is where `apropos` comes in.

```
NathanLui@local | ~ $ apropos make directory
...
makewhatis(8)             - create whatis database
makewhatis.local(8)       - start makewhatis for local file systems
mkdir(1)                  - make directories
mkfifo(1)                 - make fifos
mklocale(1)               - make LC_CTYPE locale files
...
```

`Apropos` searches the linux manual pages for the query and returns results sorted *alphabetically*.

Other useful commands

```
mkdir </path/to/directory>                        Make a new directory
rmdir </path/to/directory>                        Remove a directory
rm </path/to/file>                                Remove a file
mv </path/to/file> </path/to/directory>           Move file to new directory
cp </path/to/file> </path/to/directory>           Copy file to new directory
cat </path/to/file>                               Display the contents of a file
bash </path/to/executable>                        Run the executable
```

Let's build a program

Go to the next lesson to write your own script!

References

(1) Shell (computing)

(2) File permissions

(3) https://www.radford.edu/~mhtay/CPSC120/VIM_Editor_Commands.htm

(4) `nano` documentation

(5) `nano` cheat sheet

(6) The Linux Manual

3. My First Script

Let's write a script in `bash`! We'll do this using the command line (with `vim`), but feel free to use any text editor.

Navigate to your home directory and open a new file named `hello.sh`.

```
NathanLui@local | ~ $ vim hello.sh
```

In `vim`, type `i` to enter insert mode and type:

```
#!/bin/bash

echo "Hello world!"
```

The first line is called the *shebang* (a portmanteau of ha**sh** (#) and **bang** (!)).[1] It tells the operating system where to find the *interpreter* for the program. In this case we are telling the OS that this script can be read and run by `bash` which is located at `/bin/bash`. Many different interpreters can be used as an alternative to `bash`, for example `#!/bin/python2.7` tells the OS that this script is written in `python` and it should be run using an old version of python (2.7) located at `/bin/python2.7`.

> Recall that `bash` is both the shell and the scripting language, so `bash` commands we give, also known as the syntax, in the script are executed by the interpreting program `/bin/bash`.

Comments

The interpreter doesn't treat this line as a program call since it starts with `#`, the `bash` comment symbol. Any text in a `bash` script that is preceded with a `#` will be ignored by the interpreter. Note that different languages have different comment symbols/types (e.g. `(* OCaml *)`, `% MATLAB`, `// Java`, `<!-- HTML -->`, etc...). Comments within your code serve two purposes:

1. when **other people** read your code they understand your thinking and how you chose to implement the program, and
2. when **you** read your code, months or years later, **you** understand your thinking and how you chose to implement the program

> Comment wide and comment often, but don't comment the obvious!

Now back to our script, press `esc` to back out of insert mode and type `:wq` to **w**rite the file and **q**uit `vim`. If you're doing this in a graphical text editor save the file to the home directory with the name `hello.sh`.

Now let's look for our new file in the home directory:

```
NathanLui@local | ~ $ ls
launchCodes.txt          playGame.sh          Presentation Slides
hello.sh
```

There it is! So lets run it with the command `bash hello.sh`.

```
NathanLui@local | ~ $ bash hello.sh
The command was not found or was not executable
```

That's not good! We're sure the file exists, so it must be our access modes. Let's check:

```
NathanLui@local | ~ $ ls -l
-rw-r--r--   1 NathanLui  Users    33B Dec 21 13:21 hello.sh
```

There's the issue, not a problem since it's one we already know how to fix.

```
NathanLui@local | ~ $ chmod +x hello.sh             # n.b. the +x gives x
NathanLui@local | ~ $ ls -l                         # permission to everyone
-rwxr-xr-x   1 NathanLui  Users    33B Dec 21 13:21 hello.sh
```

Now our program should run without a hitch.

```
NathanLui@local | ~ $ bash hello.sh
Hello world!
```

Lets try something a bit more difficult. Open that script back up with `vim hello.sh` , add a variable called `food` and give it a value (like your favorite food):

```
#!/bin/bash

echo "Hello world!"
food="pizza"
```

Now let's call that variable with:

```
echo "My favorite food is $food."
```

The `$` tells the interpreter that we want the object stored in the variable `food` . Our full script now looks like:

```
1 #!/bin/bash
2
3 echo "Hello world!"
4 food="pizza"
5 echo "My favorite food is $food."
```

hello.sh hosted with ❤ by GitHub view raw

When we run it, the script now prints:

```
NathanLui@local | ~ $ bash hello.sh        # no need to change permissions
Hello world!                               # this time since we did it
My favorite food is pizza.                 # for this file earlier
```

Look at you go! One last thing that we should talk about is an environmental variable. An environmental variable is one whose value is set outside the program. Let's edit our script one more time. Append the line:

```
echo "I am $age years old."
```

So our script is now:

1	#!/bin/bash
2	
3	echo "Hello world!"
4	food="pizza"
5	echo "My favorite food is $food."
6	echo "I am $age years old."

helloEnv.sh hosted with ❤ by GitHub view raw

But we haven't declared the `age` variable yet. In some languages this would throw an error, but if we run our program we see:

```
NathanLui@local | ~ $ bash hello.sh
Hello world!
My favorite food is pizza.
I am  years old.
```

This is because `bash` automatically initializes uninitialized variables to `null` at first use (i.e. it has no value). So how do we get the script to print our age? We can initialize `age` as an environmental variable and export it to our script. We can do this in one step using `export`.

```
NathanLui@local | ~ $ export age=26
NathanLui@local | ~ $ bash hello.sh
Hello world!
My favorite food is pizza.
I am 26 years old.
```

You might be wondering why would ever need to do this. Often times we'll be working with programs and complex algorithms that can't be easily modified, or we'll want to set variables once instead of every single time we run the program. These tasks can be easily accomplished using environmental variables.

Scripting is useful for more than telling the world your favorite food and how old you are. Its our primary way of sending instructions to the cluster. When we submit jobs to the CHEM cluster's resource manager `SLURM` we do so in the form of shell scripts. More on that in the next chapter.

If you want to read more about the power of scripting I wholely recommend Al Sweigart's book Automate the Boring Stuff with Python, a fantastic (and free) resource for budding programmers (and busy grad students).[2]

References

(1) Shebang (Unix)
(2) Automate the Boring Stuff with Python

4. `SLURM`

`SLURM`, formerly known as the **S**imple **L**inux **U**tility for **R**esource **M**anagement, is a type of program called a workload manager.[1] On large, multi-user systems it can be advantageous and equitable for a program to control the allocation of computational resources, `SLURM` does just that.

When you want to run a job on the CHEM cluster you have to ask the `SLURM` daemon for resources to allocate to your job. It then takes your script, figures out how much compute power you want, and, if the nodes/memory are available, runs your script on them. If not, it places them in a queue until the requested resources become available to you.

`SLURM` has its own set of commands, and its full documentation can be found here,[2] but here we'll go over only the most important ones: `sinfo` , `pestat` , `squeue` , `sbatch` , and `scancel` .

Gathering information

`sinfo` gives us information about the status of the cluster's computing nodes (a node is a single computer in the cluster).

```
nml64@as-chm-cluster | ~ $ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
chemq        up   infinite      1  alloc chem001
chemq        up   infinite      5   idle chem[002-006]
collumq      up   infinite      4  down* dbc[001-003,005]
collumq      up   infinite      1    mix dbc009
collumq      up   infinite      1  alloc dbc007
collumq      up   infinite      4   idle dbc[004,006,008,010]
widomq       up   infinite      1  drain bw001
widomq       up   infinite      1    mix bw007
widomq       up   infinite      5   idle bw[002-006]
```

By default, `sinfo` lists the nodes by their partition; the highest level of cluster organization (a **partition** is a set of **compute nodes**). On our shared system nodes are partitioned by ownership, but other systems may have partitions based on usage (e.g. large jobs, small jobs, post-processing, data visualization, etc...) allowing the admin to install different programs on different partitions.

By default `SLURM` commands only show us nodes we have access to (more on how to change that below). So, for example, in the above snippet we have 3 partitions `chemq` , `collumq` , and `widdomq` . `collumq` has 10 total nodes, 4 of which (dbc1-3 and dbc5) are currently down, 1 of which (dbc7) is fully allocated to a running job, 1 of which (dob9) is mixed, which means it still has resources available, and 4 of which (dbc4,6,8,10) are idle. `sinfo` doesn't provide the most readable output, so sometimes its easier to use `pestat` .

Notice how the terminal's command prompt has changed from `NathanLui@Local` to `nml64@as-chm-cluster` . This is because I'm now connected to the cluster, instead of working locally on my own computer (more on how to do this in the next section).

`pestat` is quite similar to `sinfo -N` (`-N` provides a node-oriented view of the cluster), but I find the layout much easier to read. `pestat` also gives us data as to the CPU and memory capacities of each node which will be helpful later.

```
nml64@as-chm-cluster | ~ $ pestat
Hostname       Partition     Node Num_CPU  CPUload  Memsize  Freemem  Joblist
                            State Use/Tot              (MB)     (MB)  JobId User ...
   bw001          widomq    drain*  0  12     0.00    48277    37640
   bw002          widomq     idle   0  12     0.00    64382    62151
   bw003          widomq     idle   0  12     0.00    64382    62139
   bw004          widomq     idle   0  12     0.00    64382    62144
   bw005          widomq     idle   0  12     0.00    64382    62135
   bw006          widomq     idle   0  12     0.00    64382    62135
   bw007          widomq      mix   2  12     1.00*   64382    14752  8609 m----
 chem001           chemq    alloc  16  16    15.98    31935    24991  8691 j-----
 chem002           chemq     idle   0  16     0.00    31935    29712
 chem003           chemq     idle   0  16     0.00    31935    29711
 chem004           chemq     idle   0  16     0.00    31935    29721
 chem005           chemq     idle   0  16     0.00    31935    29728
 chem006           chemq     idle   0  16     0.00    31935    29730
  dbc001         collumq    down*   0   8     0.00*   16032        0
  dbc002         collumq    down*   0   8     0.00*    7968        0
  dbc003         collumq    down*   0   8     0.00*   16032        0
  dbc004         collumq     idle   0  16     0.00    24085    21791
  dbc005         collumq    down*   0  16     0.00*   24085        0
  dbc006         collumq     idle   0  16     0.00    24085    21805
  dbc007         collumq    alloc  12  12    11.96    32126    26506  8652 nml64
  dbc008         collumq     idle   0  12     0.00    32126    29959
  dbc009         collumq      mix  12  40    11.82   192049   182642  8679 nml64
  dbc010         collumq     idle   0  40     0.00   192049   189551
```

`squeue` displays the current job queue:

```
nml64@as-chm-cluster | ~ $ squeue
    JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST
     8609     chemq matlab_t    m----  R 5-13:38:51      1 bw007
     8652   collumq trans-Na    nml64  R 3-20:58:56      1 dbc007
     8679   collumq cis-NaTB    nml64  R 1-03:59:54      1 dbc009
     8691     chemq A2HMPA3_   j-----  R      39:33      1 chem001
```

Notice how `squeue` gives us a lot of information about the running jobs; it tells us the job number, who's running the job, the number of node(s), which node(s), their respective partitions, and how long the jobs have been running for.

By default, `squeue` and `sinfo` only gives us data on the nodes we have permission to use, but if we wanted to check on other nodes we can use the `-all` switch.

```
nml64@as-chm-cluster | ~ $ sinfo -all
Tue Dec 21 15:29:47 2021
PARTITION AVAIL  TIMELIMIT   JOB_SIZE ROOT OVERSUBS     GROUPS  NODES       STATE NODELI
chemq        up   infinite 1-infinite   no       NO chemit,col      1   allocated chem00
chemq        up   infinite 1-infinite   no       NO chemit,col      5        idle chem[0
slinq        up   infinite 1-infinite   no       NO slin,chemi      1   allocated sl001
slinq        up   infinite 1-infinite   no       NO slin,chemi      1        idle sl002
wilsonq      up   infinite 1-infinite   no       NO wilson,che      2        idle jjw[00
chenq        up   infinite 1-infinite   no       NO chen,chemi      1       mixed pc002
chenq        up   infinite 1-infinite   no       NO chen,chemi      1        idle pc001
loringq      up   infinite 1-infinite   no       NO loring,che      2       down* rl[00
loringq      up   infinite 1-infinite   no       NO loring,che      1     drained rl004
loringq      up   infinite 1-infinite   no       NO loring,che      1        idle rl002
collumq      up   infinite 1-infinite   no       NO collum,che      4       down* dbc[00
collumq      up   infinite 1-infinite   no       NO collum,che      1       mixed dbc009
collumq      up   infinite 1-infinite   no       NO collum,che      1   allocated dbc007
collumq      up   infinite 1-infinite   no       NO collum,che      4        idle dbc[00
widomq       up   infinite 1-infinite   no       NO chemit,col      1     drained bw001
widomq       up   infinite 1-infinite   no       NO chemit,col      1       mixed bw007
widomq       up   infinite 1-infinite   no       NO chemit,col      5        idle bw[00
lambertq     up   infinite 1-infinite   no       NO lambert,ch      2   allocated tl[00
nml64@as-chm-cluster | ~ $ squeue -all
Tue Dec 21 15:30:50 2021
JOBID PARTITION     NAME     USER    STATE       TIME TIME_LIMI  NODES NODELIST(REASON
 8590     slinq B3LYP-D3    y----  RUNNING 6-01:40:46 UNLIMITED      1 sl001
 8608     chenq matlab_t    m----  RUNNING 5-13:51:21 37-12:00:00      1 pc002
 8609    widomq matlab_t    m----  RUNNING 5-13:48:40 37-12:00:00      1 bw007
 8614     chenq matlab_t    m----  RUNNING 5-01:41:36 37-12:00:00      1 pc002
 8652   collumq trans-Na    nml64  RUNNING 3-21:08:45 17-12:00:00      1 dbc007
 8679   collumq cis-NaTB    nml64  RUNNING 1-04:09:43 17-12:00:00      1 dbc009
 8686     slinq B3LYP-D3    y----  RUNNING    3:18:46 UNLIMITED      1 sl001
 8688     slinq B3LYP-D3    y----  RUNNING      52:00 UNLIMITED      1 sl001
 8691     chemq A2HMPA3_   j-----  RUNNING      49:22 17-12:00:00      1 chem001
 8692  lambertq Dimer-6m     k---  RUNNING      19:07 5-00:00:00       1 tl001
 8693  lambertq Dimer-6m     k---  RUNNING      19:07 5-00:00:00       1 tl002
```

There are many switches you can use to filter the output of `squeue` and `sinfo` by user `--user`, partition `--partition`, node state `--state`, etc.

These are some of the most important commands we'll use in this tutorial. A short cheat sheet can be found here.[3]

Submitting jobs

`sbatch` and `scancel` are mirror commands. `sbatch <script>` submits the job script to the SLURM daemon for resource allocation, and returns a job ID number. `scancel <job ID>` cancels a job after allocation i.e., before or after a job starts running. Any files that have already been written

will be preserved as they are when `scancel` is executed (keep this in mind if you choose to write any large scratch files to your job directory instead of `/scratch`). In the next section, we'll learn about how to format submission scripts and submit our first `SLURM` job.

References

(1) SLURM Workload Manager
(2) SLURM Documentation
(3) SLURM Cheat Sheet

5. My First `SLURM` Job

Now that we know how to gather information about the system, how do we ask it to run a job for us?

`SLURM` needs to know two things to run a job: **what we want to do** and **the resources we need to do it**. We'll use a *shell script* to specify both of these parameters.

Let's make a new script called `submit.sh`. In your text editor copy and paste the following (minimal) submission script:

```
#!/bin/bash

#SBATCH -p chemq            # submit to partition: chemq
#SBATCH -J hello            # job name
#SBATCH -o out.txt          # name output file
#SBATCH -N 1                # run on one node
#SBATCH --mem=0             # allocate all available memory

# ^^^ Above are the resource requests ^^^
#    vvv Below are the job tasks vvv

echo 'Starting job'
bash hello.sh               # run our first script from the
echo 'Resting 30 sec'       #   previous exercise
sleep 30                    # do nothing for 30 sec
echo 'Ending job'
```

submit.sh hosted with ❤ by GitHub　　view raw

Save this file in its own folder with a descriptive name like `myFirstSlurmJob`. Place the script `hello.sh` from the first exercise into this folder too. Now, in order to run this job you need to be on a system that is managed by `SLURM`. So let's log on to the AS-CHEM cluster.

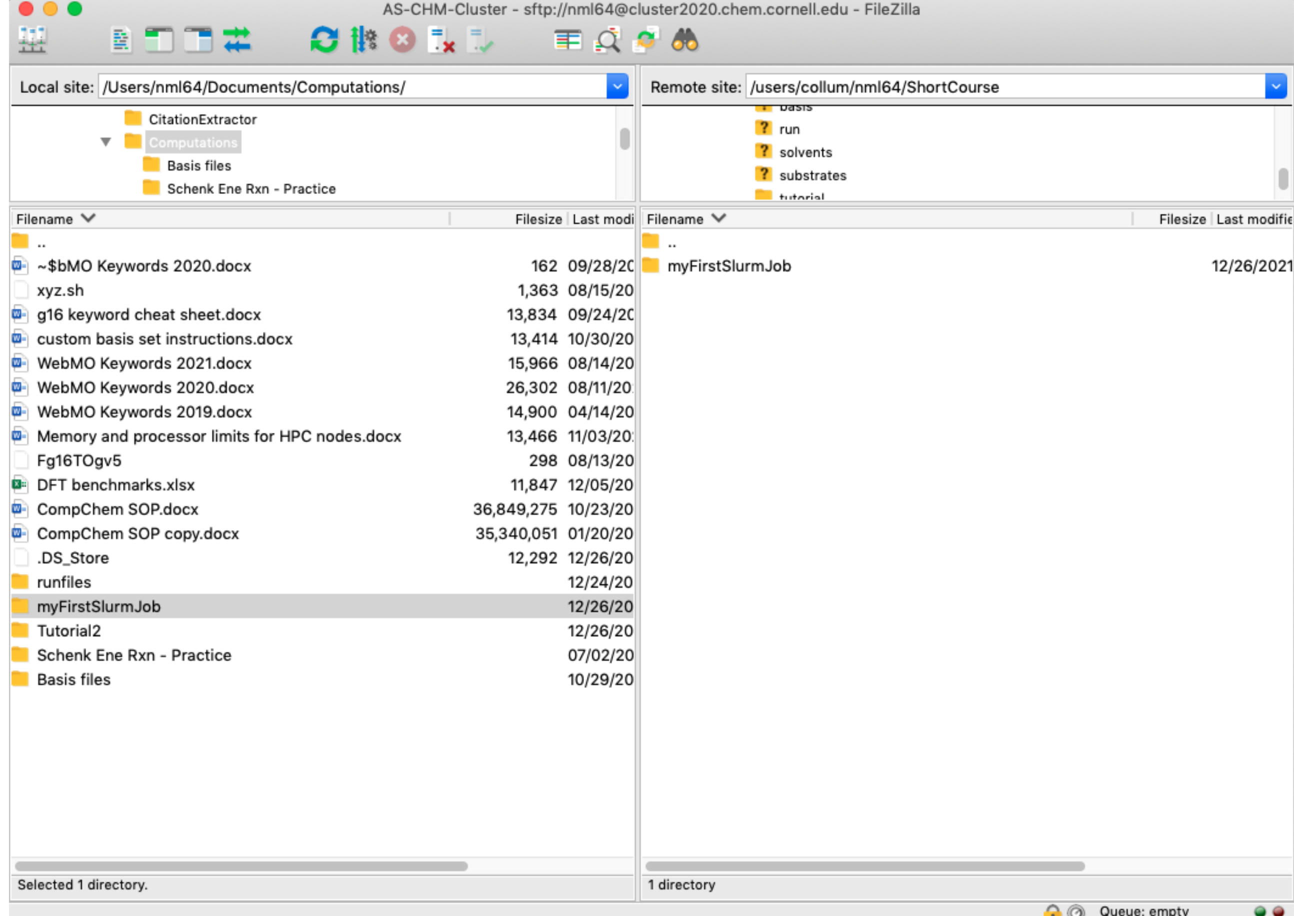
> You'll need to be connected to the Cornell VPN to access the cluster. If you are a Cornell chemistry student and don't have access to the cluster go see ChemIT (or your group IT representative) to set up your cluster account. If you are not a Cornell chemistry student you'll need to follow your institution's cluster login instructions. Depending on how your cluster is set up some of the instructions below may not work, when in doubt contact your system administrator.

Open up the command-line, type `ssh <yourNetID>@cluster2020.chem.cornell.edu` and you'll see a password prompt appear. As you enter your password **nothing will appear**; this is normal. The terminal is recording your keystrokes as usual, but will not display them for security purposes.

```
NathanLui@local | ~ $ ssh nml64@cluster2020.chem.cornell.edu
Password:

Last login: Sun Dec 26 14:04:09 2021 from <IP address>
nml64@as-chm-cluster | ~ $
```

See how the terminal prompt has now changed from `NathanLui@local` to `nml64@as-chm-cluster` to indicate that I'm now working on the cluster. We can navigate the cluster with the same commands we learned earlier. To test our script we'll need to use our SFTP client (FileZilla) to transfer our scripts to the cluster. If you haven't yet, go set up FileZilla using the directions in section 1. Once you've done that, open FileZilla and connect to the `AS-CHEM` cluster. Drag your whole `myFirstSlurmJob` folder into the cluster pane to transfer it.



> Of course, you have the option to create a new folder on the cluster directly using `mkdir` and then drag the individual shell scripts into that file, but as your experience grows as will the number of files you'll have to keep track of. It will be much more manageable if the organization of your local system mirrors that of the cluster. Transferring whole directories ensures that paths will remain the same. For more, see best practices.

Now, lets navigate into that folder and take a look:

```
nml64@as-chm-cluster | ~ $ cd myFirstSlurmJob/
nml64@as-chm-cluster | ~/myFirstSlurmJob $ ls -l
total 8.0K
-rwxr-xr-x 1 nml64 collum  79 Dec 26 15:59 hello.sh
-rwxr-xr-x 1 nml64 collum 338 Dec 26 15:59 submit.sh
```

Now we can submit our job to the `SLURM` workload manager:

```
nml64@as-chm-cluster | ~/myFirstSlurmJob $ sbatch submit.sh
Submitted batch job 8716
```

Checking the job queue and node status shows us the progress of our new job:

```
nml64@as-chm-cluster | ~/myFirstSlurmJob $ squeue
 JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
  8716     chemq    hello    nml64  R       0:16      1 chem006
nml64@as-chm-cluster | ~/myFirstSlurmJob $ pestat
Hostname       Partition     Node Num_CPU  CPUload  Memsize  Freemem  Joblist
                            State Use/Tot              (MB)     (MB)  JobId User ...
...
 chem006           chemq      mix  16  16    16.00    31935    29651  8716 nml64
...
```

But wait a second! Where is our output? We've tasked 16 CPUs with 28 GB of memory to tell the whole world "Hello!", where did it all go? Let's take a look at our directory.

```
nml64@as-chm-cluster | ~/myFirstSlurmJob $ ls -l
total 12K
-rwxr-xr-x 1 nml64  79 Dec 26 15:59 hello.sh
-rw-r--r-- 1 nml64  81 Dec 26 17:35 out.txt
-rwxr-xr-x 1 nml64 338 Dec 26 15:59 submit.sh
nml64@as-chm-cluster | ~/myFirstSlurmJob $ cat out.txt
Starting job
Hello world!
My favorite food is pizza.
I am  years old.
Resting 30 sec
Job complete
```

So that's where its all gone to! `SLURM` redirects all standard output from the terminal to the output file that we specified in the resource requests section.

There's still another issue! The program doesn't know how old we are because the environmental variable we declared in the last tutorial doesn't get transferred with the file (i.e. we're in a different environment). So we have to redeclare `age` in this environment.

```
1   #!/bin/bash
2
3   #SBATCH -p chemq           # submit to partition: chemq
4   #SBATCH -J hello           # job name
5   #SBATCH -o out.txt         # name output file
6   #SBATCH -N 1               # run on one node
7   #SBATCH --mem=0            # allocate all available memory
8
9   # ^^^ Above are the resource requests ^^^
10  #    vvv Below are the job tasks vvv
11
12  echo 'Starting job'
```

```
13  export age=26
14  bash hello.sh            # run our first script from the
15  echo 'Resting 30 sec'    #   previous exercise
16  sleep 30                 # do nothing for 30 sec
17  echo 'Ending job'
```

submitEnv.sh hosted with ❤ by **GitHub** view raw

This fixes our issue and if we run the job again we can see that the script works as it's supposed to!

```
nml64@as-chm-cluster | ~/myFirstSlurmJob $ sbatch submit.sh
Submitted batch job 8719
nml64@as-chm-cluster | ~/myFirstSlurmJob $ ls -l
total 12K
-rwxr-xr-x 1 nml64 collum 107 Dec 26 17:46 hello.sh
-rw-r--r-- 1 nml64 collum 100 Dec 26 17:56 out.txt
-rwxr-xr-x 1 nml64 collum 352 Dec 26 17:55 submit.sh
nml64@as-chm-cluster | ~/myFirstSlurmJob $ cat out.txt
Starting job
Hello world!
My favorite food is pizza.
I am 26 years old.
Resting 30 sec
Job complete
```

`SLURM` will overwrite data files with the same name

One important thing to note is that we ran this job multiple times *in the same directory*. So `SLURM` **wrote over** `out.txt` the second time we ran the job. There is **no way** to get back our first `out.txt` (trivially, you could scroll up in the terminal history looking for our previous `cat out.txt` call, but this isn't really a generalizable solution). This could be problematic since we might not remember how we got to the previous `out.txt` and how to recreate its results. In general, a single folder should represent a single program call so that unintentional overwrites cannot happen. In other words:

> *Every new job should begin in its **own** new folder.*

In the next section, we'll talk about the final part of our recipe: the Gaussian input file.

6. The `Gaussian` Input File

`Gaussian16` (`g16`) input files are plain text files that end in `.com` or `.gjf`. They can be generated using a molecular modeling program like `GaussView` or `Avogadro` or in a simple text editor (provided one has the atomic coordinates already). The general format of a `Gaussian` input file is given below accompanied by a short description of each section.

```
Link 0 commands              ! Specifies memory, CPU, and other job information
Routing information          ! Specifies job parameters
<<<BLANK LINE>>>
Title line                   ! Free-format comment line
<<<BLANK LINE>>>
Charge and Multiplicity      ! Space delimited
Molecule specification(s)    ! Cartesian, Z-matrix, or Redundant Internals
<<<BLANK LINE>>>
Job/option specific input    ! See g16 manual
<<<BLANK LINE>>>
<<<BLANK LINE>>>
```

Formatting and syntax

In general, `Gaussian` input files follow simple and flexible syntax and grammar rules:[1]

- Inputs are case-insensitive i.e., opt, OPT, and oPt request the same optimization job
- Comments are made using the exclamation point (`!`)
- External files can be read into an input using the syntax: `@path/to/file`
- Keywords and keyword options can be specified using the following options:
  - keyword=option
  - keyword(option)
  - keyword=(option1, option2, ...)
  - keyword(option1, option2, ...)

Link 0

Link 0 command lines begin with the percent `%` symbol and set program control options such as resource limits, whether to save and what to name scratch files, where to access old scratch files, etc… The link 0 commands are detailed here.[2] Each link 0 command requires its own line. Every link 0 option can also be called as a command line flag in your shell script or passed to `Gaussian` as an environmental variable. Equivalent commands and the corresponding precedence are detailed under the “Equivalencies” tab in the link above.

> This section (and everything else in the input file) is an instruction to `Gaussian`, not `SLURM`. The memory and CPU allocations specified in the input file should, ideally, be equal to those requested from `SLURM` (so that there is no "wasted" computing power), but must not be greater than those requested from `SLURM`, otherwise `Gaussian` will attempt to allocate more resources than has been made available to it which will lead to job failure.

The `.chk` file

A common link 0 command is `%chk`, used to save the checkpoint file from the calculation (typically deleted with other scratch files when the job is completed). Every iteration of a calculation `Gaussian` saves an image of job in the checkpoint file (like a savepoint) allowing the user to go back and restart a failed job. In addition to various savepoints, the checkpoint file also allows the user to view the molecular orbitals in `GaussView`. Additionally, specifications (e.g. basis set, functional, molecular geometries, etc...) can be read from the checkpoint file for future jobs; however, this means the `.chk` file can get very large (on the order of 100s of mb). Saving `.chk` files indiscriminately is unadvisable unless you have ample storage space.

Routing information

The routing line of the `Gaussian` job file begins with the pound/hash `#` symbol. The letter following the `#` determines the level of output. The options are `#P`, `#N`, & `#T` which provide verbose, normal, and terse levels of description in the job's output file. The default option is `#N` (which can shortened to `#`). The options that follow are referred to by `Gaussian` as "keywords" and are responsible for setting up the requested calculation.[3]

The routing line continues with a method and basis set declaration in single-slash notation. If no options are specified the default method is a Hartree-Fock calculation (HF) using the minimal Slater-type orbital basis functions (STO-3G), given as HF/STO-3G. g16's built in DFT methods and basis sets are available in the documentation. The choice of functional and basis set are discussed further in a later section.[4,5]

> With the advancement of computational chemistry and the availability of computing resources there is no reason to perform the default HF/STO-3G computation. The most commonly used theory/basis combination in computational organic chemistry is the infamous Becke, 3-parameter, Lee-Yang-Parr (B3LYP) hybrid-exchange correlated functional used in combination with the 6-31g* basis set.

The final required keyword in the routing line is the type of calculation. Typical calculations include geometry optimizations (Opt), vibrational frequency analysis (Freq), single-point molecular energy

calculations (SP), NMR shift calculations (NMR), etc… A full list of `g16`'s computational capabilities can be found here.[6] These job keywords can be specified on their own or with various options.

A calculation minimally requires a functional, basis set, and the type of calculation; however, it is usually necessary to specify customizable options such as implicit solvation, temperature (for frequency analysis), initial guesses, counterpoise correction, empirical dispersion correction, extra basis functions, effective core potentials (i.e., pseudopotentials), etc… These options are specified from the routing line using their respective keyword options.[3] The full list of `Gaussian` keywords and their availbe options can be found in the g16 documentation.[7]

Keywords can be specified in any order. The routing section is fully reproduced in the output file and terminated by a blank line.

Integration grid size

All DFT methods implemented in `Gaussian` involve a grid-based numerical integration of the functional (or its derivatives). Consequently, the accuracy of a DFT calculation is dependent on the resolution of the integration grid. In `Gaussian16` the default integration for all DFT functionals is calculated over a pruned grid with 99 radial shells and 590 angular points, denoted (99,590), and specified by the keyword `Integral(grid=ultrafine)`. This is sufficient for the vast majority of all calculations. Using a smaller grid, while faster, is not recommended for most DFT calculations. Lastly, energy comparisons must be done using the **same grid size.**[4]

Job title

The title/comment line is plaintext that is reproduced in the `Gaussian` output file and terminated by a blank line.

Charge and multiplicity

The charge and multiplicity of the system is given before the molecule specification in standard convention separated by a space. For example, `-1 1` describes an anionic singlet state.

Molecule specification

The molecule specification can be given in either standard cartesian (xyz) or internal (Z-matrix) coordinates. Note that specification in one coordinate system or another will not dictate what coordinates will be used to perform the optimization itself which defaults to `Gaussian`'s redundant internal coordinates, but can be changed by specifying an option in the Opt keyword.

This section may be omitted altogether if reading the start geometry from a pre-existing file (such as a checkpoint (`.chk`) file) using the `geom=checkpoint` option.

The molecule specification is terminated by a blank line.

Job/keyword options

Some keyword options require additional input (such as specifying ECPs or additional basis functions). This additional information is placed after the molecule specification and each individual keyword's specifications are terminated by blank lines.

The input file is terminated by two blank lines.

A simple `g16` input file

Below is a fully functional `Gaussian` input file for a single point molecular energy calculation. Copy + paste it into a text editor and save it as `coolMolecule.com` and see if you can figure out what we're trying to do in the next section. (Hint: open it up in your favorite molecular editor/viewer)

```
%NPROCSHARED=16                        ! run on 16 parallel cores
%MEM=28GB                              ! run with 28 gb memory
%Chk=coolMolecule.chk                  ! save a checkpoint file
#N M062X/def2tzvp SP                   ! calculate single-point energy

My cool molecule

0 1                                    ! neutral singlet
C                                      ! starting geometry for
C 1 B1                                 ! a molecule specified in
C 2 B2 1 A1                            ! gaussian internal coordinates
C 3 B3 2 A2 1 D1 0
C 4 B4 3 A3 2 D2 0                     ! to save space bond lengths,
C 1 B5 2 A4 3 D3 0                     ! angles, and dihedrals are all
H 3 B6 2 A5 1 D4 0                     ! specified as variables
H 2 B7 1 A6 6 D5 0
H 1 B8 6 A7 5 D6 0
H 1 B9 6 A8 5 D7 0
H 4 B10 3 A9 2 D8 0
H 4 B11 3 A10 2 D9 0
H 5 B12 4 A11 3 D10 0                  ! g16 doesn't support comments
H 5 B13 4 A12 3 D11 0                  ! in the molecule specification
H 6 B14 1 A13 2 D12 0                  ! so get rid of these comments
H 6 B15 1 A14 2 D13 0                  ! before you try to run this job
H 3 B16 2 A15 1 D14 0
C 2 B17 1 A16 6 D15 0
H 18 B18 2 A17 1 D16 0
H 18 B19 2 A18 1 D17 0
H 18 B20 2 A19 1 D18 0

   B1              1.51510600
   B2              1.51517951
   B3              1.51543497
   B4              1.51512522
   B5              1.51498514
   B6              1.12087116
   B7              1.12176819
   B8              1.12177478
   B9              1.12093060
   B10             1.12176062
   B11             1.12091101
   B12             1.12095758
   B13             1.12181557
   B14             1.12176019
   B15             1.12097986
   B16             1.12168148
   B17             1.54000000
   B18             1.07000000
   B19             1.07000000
   B20             1.07000000
```

```
A1          111.36252447
A2          111.24127560
A3          111.26574203
A4          111.30936835
A5          109.59001104
A6          109.39681635
A7          109.40740597
A8          109.57492433
A9          109.41106707
A10         109.58676541
A11         109.55885178
A12         109.38713435
A13         109.41082787
A14         109.56859847
A15         109.42519797
A16         109.55953690
A17         109.47120255
A18         109.47120255
A19         109.47123134
D1           55.25714299
D2          -55.23663791
D3          -55.19282652
D4          176.59526965
D5           65.84971766
D6          -65.94975705
D7          176.44368868
D8           65.79362957
D9         -176.57422515
D10         176.61579480
D11         -65.80639557
D12         -65.96332262
D13         176.42532586
D14         -65.75731388
D15        -176.56184324
D16        -178.76723016
D17         -58.76721537
D18          61.23277724
```

References

(1) Gaussian input file syntax

(2) Link 0 commands

(3) Gaussian keyword list

(4) DFT methods

(5) Basis sets

(6) Gaussian capabilities
(7) Gaussian16 documentation

7. My First `Gaussian` Job

If you've been following along then you *almost* have all the pieces to run your first Gaussian job on our cluster.

There's just a little bit of work we need to do tie everything together.

Our `g16` input file

We should tweak the `.com` file that we were working on last section. First, we'll get rid of the link 0 commands since we can easily (and more reliably) pass them to `Gaussian` as environment variables. Next, we'll run a geometry optimization and frequency analysis instead of just calculating the energy.

> A frequency analysis should always accompany geometry optimizations to verify that the final geometry is a global minimum on the potential energy surface (i.e., a true ground state), instead of a saddle point.

To do this we'll specify the following keywords:

```
#N M062X/def2svp          ! use the m06-2x functional with def2svp basis set
OPT                       ! conduct a ground state geometry optimization
FREQ=NoRaman              ! conduct a frequency analysis of the optimized geometry
                ! do not calculate Raman stretches (reduces computation time by ~30%)
temperature=273.15        ! standard temperature
Integral(Grid=UltraFine)  ! use an ultrafine integration grid (99,590)
```

So our full input should look like:

```
1  #N M062X/def2svp OPT FREQ=NoRaman temperature=273.15 Integral(Grid=UltraFine)
2
3  eqMe-cyclohexane
4
5  0 1
6  C
7  C 1 B1
8  C 2 B2 1 A1
   C 3 B3 2 A2 1 D1 0
```

```
9
10  C 4 B4 3 A3 2 D2 0
11  C 1 B5 2 A4 3 D3 0
12  H 3 B6 2 A5 1 D4 0
13  H 2 B7 1 A6 6 D5 0
14  H 1 B8 6 A7 5 D6 0
15  H 1 B9 6 A8 5 D7 0
16  H 4 B10 3 A9 2 D8 0
17  H 4 B11 3 A10 2 D9 0
18  H 5 B12 4 A11 3 D10 0
19  H 5 B13 4 A12 3 D11 0
20  H 6 B14 1 A13 2 D12 0
21  H 6 B15 1 A14 2 D13 0
22  H 3 B16 2 A15 1 D14 0
23  C 2 B17 1 A16 6 D15 0
24  H 18 B18 2 A17 1 D16 0
25  H 18 B19 2 A18 1 D17 0
26  H 18 B20 2 A19 1 D18 0
27
28     B1              1.51510600
29     B2              1.51517951
30     B3              1.51543497
31     B4              1.51512522
32     B5              1.51498514
33     B6              1.12087116
34     B7              1.12176819
35     B8              1.12177478
36     B9              1.12093060
37     B10             1.12176062
38     B11             1.12091101
39     B12             1.12095758
40     B13             1.12181557
41     B14             1.12176019
42     B15             1.12097986
43     B16             1.12168148
44     B17             1.54000000
45     B18             1.07000000
46     B19             1.07000000
47     B20             1.07000000
48     A1            111.36252447
49     A2            111.24127560
50     A3            111.26574203
```

```
A4          111.30936835
A5          109.59001104
A6          109.39681635
A7          109.40740597
A8          109.57492433
A9          109.41106707
A10         109.58676541
A11         109.55885178
A12         109.38713435
A13         109.41082787
A14         109.56859847
A15         109.42519797
A16         109.55953690
A17         109.47120255
A18         109.47120255
A19         109.47123134
D1           55.25714299
D2          -55.23663791
D3          -55.19282652
D4          176.59526965
D5           65.84971766
D6          -65.94975705
D7          176.44368868
D8           65.79362957
D9         -176.57422515
D10         176.61579480
D11         -65.80639557
D12         -65.96332262
D13         176.42532586
D14         -65.75731388
D15        -176.56184324
D16        -178.76723016
D17         -58.76721537
D18          61.23277724


```

eqMeCyhex.com hosted with ❤️ by GitHub   view raw

Let's save it into a folder called `eqMeCyclohexane` and start working on our `SLURM` script.

`SLURM` job script

`Gaussian16` is slightly more resource intensive than our `hello.sh` script, so we'll need to be a bit better at requesting resources from `SLURM` than the minimal requests we made in the last exercise. In your `myFristG16Job` make a new submission script with the following resource allocation request:

```
#SBATCH -p chemq                          # submit to partition: chemq
#SBATCH -J eqMeCyhex                      # job name
#SBATCH -o %x_oe                          # name of stdout/stderr file
#SBATCH -N 1                              # total number of nodes requested
#SBATCH --ntasks-per-node=16              # total number of tasks requested
#SBATCH --mincpus=16                      # assign at least 16 CPUs from each node
#SBATCH --mem=0                           # allocate all of the node's available memory
#SBATCH -t 240:00:00                      # max run time (hhh:mm:ss)
#SBATCH --mail-type=ALL                   # send emails at job START/END/FAIL
#SBATCH --mail-user=<yourNetID>@cornell.edu
```

> n.b. The `%x` in the `-o` option line is a `SLURM` variable for the job name.

Now, specify the job details:

```
g16root=/software                       # the g16.profile file defines the g16
. $g16root/g16/bsd/g16.profile          # defaults; ask admin for its location
export GAUSS_SCRDIR=/scratch
export GAUSS_CDEF='0-15'                # see https://gaussian.com/link0/ for
export GAUSS_MDEF='28GB'                # environment variable definitions
export GAUSS_YDEF='eqMeCyhex.chk'
```

The last four lines define environment variables for `g16`; these are equivalent to setting link 0 commands directly in the `.com` file, but any link 0 commands specified in the `.com` file will override those passed to `Gaussian` as environmental variables or command-line options.[1]

> Recall that the memory and CPU assignments requested by `g16` must be at most those requested from the `SLURM` resource daemon, otherwise the job will fail due it insufficient resources.

The last thing left to do is to call the program. To run `g16` call the executable with `/path/to/g16 <inputFile> <outputFileName>` (by convention, `Gaussian` output files have the ending `.log`):

```
$g16root/g16/g16 eqMeCyhex.com eqMeCyhex.log
# on AS-CHEM the g16 program is located at /software/g16/g16 on other clusters
# it may not be in the same place. Ask your system admin for its location!
```

At this point we have everything we need. The rest of our job details will be read into g16 from the input file. Our full job script should look like:

```
#!/bin/bash

#SBATCH -p chemq                        # submit to partition: chemq
#SBATCH -J eqMeCyhex                    # job name
#SBATCH -o %x_oe                        # name of stdout/stderr file
#SBATCH -N 1                            # total number of nodes requested
#SBATCH --ntasks-per-node=16            # total number of tasks requested
#SBATCH --mincpus=16                    # assign at least 16 CPUs from each node
#SBATCH --mem=0                         # allocate all of the node's available memory
#SBATCH -t 240:00:00                    # run time (hhh:mm:ss)
#SBATCH --mail-type=ALL                 # send emails at job START/END/FAIL
#SBATCH --mail-user=<yourNetID>@cornell.edu

g16root=/software                       # the g16.profile file defines the g16
. $g16root/g16/bsd/g16.profile          # defaults; ask admin for its location
export GAUSS_SCRDIR=/scratch
export GAUSS_CDEF='0-15'                # see https://gaussian.com/link0/ for
export GAUSS_MDEF='28GB'                # environment variable definitions
export GAUSS_YDEF='eqMeCyhex.chk'

$g16root/g16/g16 eqMeCyhex.com eqMeCyhex.log
```

g16run.sh hosted with ❤️ by **GitHub** view raw

Save it into the same folder as `eqMeCyhex.com` and now let's go submit it!

Submitting our first `Gaussian` job

In FileZilla connect to the CHEM cluster and drag your whole `eqMeCyclohexane` folder from your computer to the cluster's home directory. `cd` into that folder and make sure everything's where it should be.

```
nml64@as-chm-cluster | ~ $ cd eqMeCyclohexane/
nml64@as-chm-cluster | ~/eqMeCyclohexane $ ls
eqMeCyclohex.com  g16run.sh
```

Submit the job with `sbatch`. Its usually a good idea to make sure that the job has started properly. When `Gaussian` jobs fail they typically fail in the first 20 seconds (usually due to a FileIO issue, syntax errors, or insufficient resources) so checking that the job is running smoothly prevents you

from coming back later just to realize that your job was killed after 7 seconds because you spelled the input filename incorrectly. If we look at the files in our directory while the job is running, we see three new files our checkpoint file `eqMeCyhex.chk`, our `Gaussian` output file `eqMeCyhex.log`, and our `SLURM` output file `eqMeCyhex_oe`.

```
nml64@as-chm-cluster | ~/eqMeCyclohexane $ sbatch g16run.sh
Submitted batch job 8721
nml64@as-chm-cluster | ~/eqMeCyclohexane $ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
              8721     chemq eqMeCyhe    nml64  R       0:36      1 chem005
nml64@as-chm-cluster | ~/eqMeCyclohexane $ ls
eqMeCyhex.chk  eqMeCyhex.com  eqMeCyhex.log  eqMeCyhex_oe  g16run.sh
nml64@as-chm-cluster | ~/eqMeCyclohexane $ ls
eqMeCyhex.chk  eqMeCyhex.com  eqMeCyhex.log  eqMeCyhex_oe  fort.7  g16run.sh
```

`fort.7`

After the job is completed we see another file `fort.7`. After a job finishes `Gaussian` "punches out" (the parlance comes from a time when computers used punched cards as fileIO)[2] a separate output for the results of that calculation. The output can be specified using the `Punch` keyword, however `Gaussian` punches a new `fort.7` file at each termination. In our combined optimization/frequency calculation the program terminates twice: once following the optimization and again after the vibrational frequency analysis. So, in a combined job you get only the punch out for the final calculation.

Checking for successful termination

We can do a quick check to make sure our job finished correctly using `tail`.

```
nml64@as-chm-cluster | ~/eqMeCyclohexane $ tail eqMeCyhex.log


  ... THE UNIVERSE IS NOT ONLY QUEERER THAN WE SUPPOSE,
  BUT QUEERER THAN WE CAN SUPPOSE ...

                                  -- J. B. S. HALDANE
 Job cpu time:       0 days  0 hours 27 minutes 52.9 seconds.
 Elapsed time:       0 days  0 hours  1 minutes 46.1 seconds.
 File lengths (MBytes):  RWF=     66 Int=      0 D2E=      0 Chk=      5 Scr=      1
 Normal termination of Gaussian 16 at Mon Dec 27 13:06:58 2021.
```

`tail -n` prints the last `n` lines (by default `n` is 10) of the file to the console. The first two lines tell us how long the job took, the third tells us the size of the job's scratch files, and the final line informs

us that `Gaussian` terminated without error. Additionally, if your job terminates successfully `Gaussian` will print you a quote. If the job had failed the output tail would look something like the code bloc below. Different reasons for failure will produce different looking outputs but all failed jobs will produce an `Error termination` message.

```
nml64@as-chm-cluster | ~/.../NaTMIPS/A3_eee_tol $ tail eqMeCyhex.log
 Error on total polarization charges = ********
 SCF Done:  E(RM062X) =  -8076.68013868     A.U. after  129 cycles
            NFock=128  Conv=0.45D-02     -V/T= 7.5222
 SMD-CDS (non-electrostatic) energy       (kcal/mol) =      -2.87
 (included in total energy above)
 Convergence failure -- run terminated.
 Error termination via Lnk1e in /software/g16/l502.exe at Sat Dec 18 17:05:50 2021.
 Job cpu time:       14 days  5 hours  4 minutes 13.4 seconds.
 Elapsed time:        1 days  5 hours 16 minutes 45.2 seconds.
 File lengths (MBytes):  RWF=    958 Int=      0 D2E=      0 Chk=     45 Scr=      1
```

🎉🎉 Great job! You just ran your first `Gaussian` optimization! 👏👏

In the next section, we'll go over how to examine the output file and parse it for the important thermochemical data.

References

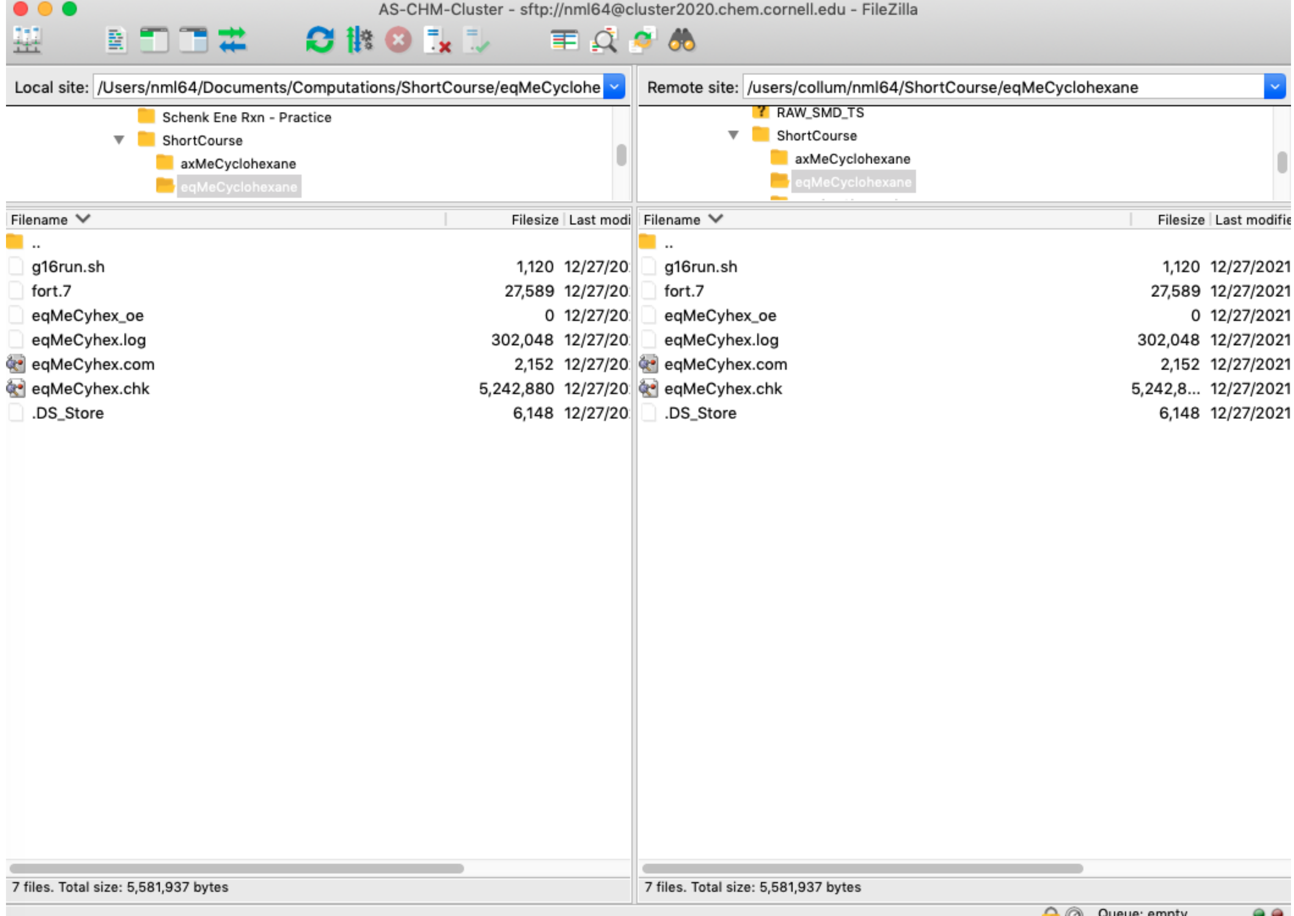
(1) `Gaussian` link 0

(2) Punched Card

(3) The `Gaussian` `Punch` keyword

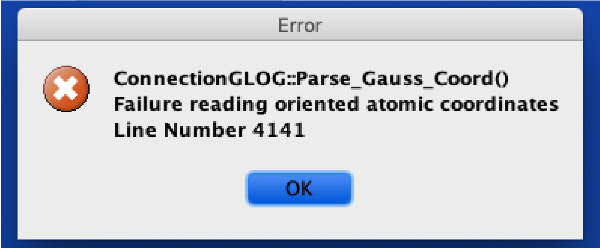
8. Understanding the `Gaussian` output file

For easier analysis, let's drag all of our files back onto our personal laptop using FileZilla. (If you want you can try to read the `Gaussian` `.log` file in the terminal, but you'll soon see why that's not going to scale well.)



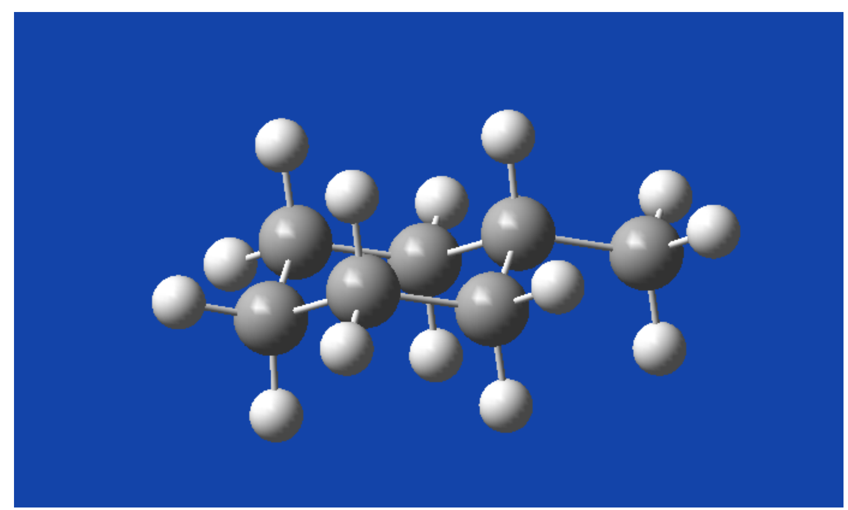
We won't use the `fort.7` file and `eqMeCyhex_oe` has a total size of 0 bytes (i.e., there's nothing written in it) so for the sake of cleanliness we can delete those. First, let's look at our optimized structure to make sure the final geometry makes chemical sense. Open `eqMeCyhex.log` in `GaussView` (download it from the code repo if you're not following along).

If you're using `GaussView5` with `Gaussian16` you'll most likely run into this error:



Don't panic, this issue occurs because `g16` writes some extra information in the output file that `GaussView5` doesn't know how to handle. Use this script provided by Dr. Davor Šakić from the University of Zagreb to generate a output file that `GaussView5` can read.

`GaussView` shows us a perfectly normal equatorial methylcyclohexane.



You should always check your structures to make sure they are generally expected since not all *mathematical* solutions are *physical* ones. Sometimes our jobs will give us chemically nonsensical solutions simply because the algorithm found a particular energy well that it couldn't get out of.

> *The computers are here to do your math, not your thinking.*

Fantastic, let's grab some numbers. One of the first things we noticed is that `eqMeCyhex.log` is just a *really* long text file that `GaussView5` is able to generate a picture from. So, open `eqMeCyhex.log` in a text editor (or follow along in another window using the link above) and we search the output for energies and vibrational/thermochemical data.

First we'll want to check for imaginary (negative) vibrational frequencies which indicate saddle point structures. Search `eqMeCyhex.log` for `Harmonic frequencies` :

```
...
Harmonic frequencies (cm**-1), IR intensities (KM/Mole), Raman scattering
 activities (A**4/AMU), depolarization ratios for plane and unpolarized
 incident light, reduced masses (AMU), force constants (mDyne/A),
 and normal coordinates:
                      1                      2                      3
                      A                      A                      A
 Frequencies --    159.1459               229.6806               248.3997
 Red. masses --      2.2917                 1.2669                 1.3457
 Frc consts  --      0.0342                 0.0394                 0.0489
 IR Inten    --      0.0025                 0.0069                 0.0006
...
```

`Gaussian` prints all vibrational frequencies in the output in *ascending order* so we only need to check the first entry to ensure that all our vibrational frequencies are real.

Next, we'll calculate the energy for our optimized structure. At this point, I **highly recommend** that you read this technical document from Dr. Joseph Ochterski about thermochemistry in `Gaussian`.[1]

It describes how `Gaussian` calculates various thermochemical values and their proper usage in computing ΔG_{rxn}. Searching `eqMeCyhex.log` for `correction` produces:

```
...
 Zero-point correction=                           0.198783 (Hartree/Particle)
 Thermal correction to Energy=                    0.204789
 Thermal correction to Enthalpy=                  0.205654
 Thermal correction to Gibbs Free Energy=         0.171472
 Sum of electronic and zero-point Energies=           -274.642438
 Sum of electronic and thermal Energies=              -274.636431
 Sum of electronic and thermal Enthalpies=            -274.635566
 Sum of electronic and thermal Free Energies=         -274.669748
...
```

> With most *ab initio* methods absolute energies of molecular systems are calculated relative to free electrons and nuclei which is why they are large and negative.

By default, `Gaussian` reports energies in Hartree atomic units (E_h or A.U.):

$$ 1~E_h = \frac{\hbar^2}{m_e~a_0^2} \approx 627.5 ~ \textrm{kcal mol}^{-1} $$

The values we're interested in are:

```
 Thermal correction to Gibbs Free Energy=         0.171472
 Sum of electronic and thermal Free Energies=         -274.669748
```

The `Thermal correction to Gibbs Free Energy` is calculated by:

$$ G_{corr} = E_{thermal} + \mathscr{k}_BT - TS_{total} $$

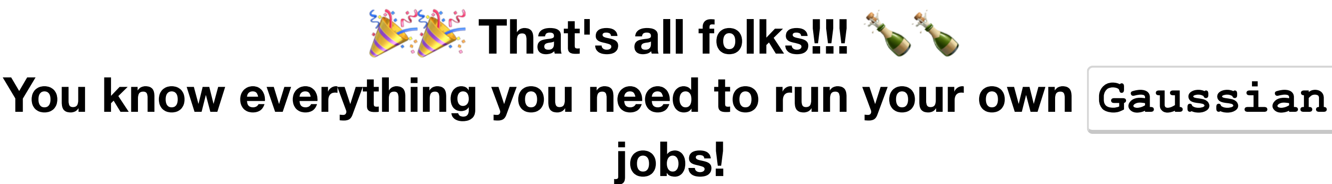
The `Sum of electronic and thermal Free Energies` is the sum of the above `Thermal correction` and the `electronic energy` (also known as the `single point energy` since its the energy at a single point on the potential energy surface).[2] This `thermally-corrected single point energy` is the value that should be used to calculate free energies of reaction (ΔG_{rxn}).

`Gaussian` calculates the `single point energy` of each intermediate geometry it generates during optimization as well as at the start of a vibrational frequency analysis. We can exploit this fact to save us from having to set up another calculation. To find the `single point energy` search `eqMeCyhex.log` for **the last occurrence** of `SCF Done` :

```
 SCF Done:  E(RM062X) =  -274.841184603     A.U. after    9 cycles
```

With this value the relationship between these three quantities becomes clear.

```
SCF energy: E(RM062X)                            =     -274.841184
Thermal correction to Gibbs Free Energy          =        0.171472
--------------------------------------------------------------------------
E(RM062X) + Thermal correction                   =     -274.669748
Sum of electronic and thermal Free Energies      =     -274.669748
```



In our final lesson we'll see how we can use `Gaussian` to calculate relative conformational energies.

A note on split basis calculations

It is common in large systems to use a smaller set of basis functions to find the optimized geometry (this is part of the Long Course) and then use a larger basis set to recalculate the single point energy. In this case the calculated `Sum of electronic and thermal Free Energies` and the `thermally-corrected single point energy` derived from the larger basis set **will not** be the same.

> You must manually correct single point energies when running split-basis calculations.

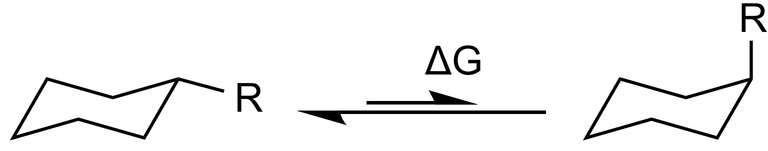
References

(1) Thermochemistry in `Gaussian`

(2) Potential energy surface

9. Calculating cyclohexane A-values

The A-value of a substituent is the energy of the axial cyclohexane conformer relative to the equatorial conformer (i.e., the *isomerization energy*). In organic conformational analysis the A-value is used as the archetypal steric parameter.



In our last exercise we calculated the energy of equatorial methylcyclohexane:

```
eqMeCyhex:
SCF energy: E(RM062X)                          =     -274.841184 Eh
Thermal correction to Gibbs Free Energy        =        0.171472 Eh
--------------------------------------------------------------------------
E(RM062X) + Thermal correction                 =     -274.669748 Eh
Sum of electronic and thermal Free Energies    =     -274.669748 Eh
```

Let's see if you can do the same with the axial conformer. Take a quick break and see if you can set up and execute this calculation on your own.
If you're just following along or get stuck feel free to grab the files from the code repo.

```
axMeCyhex:
SCF energy: E(RM062X)                          =     -274.838661 Eh
Thermal correction to Gibbs Free Energy        =        0.171734 Eh
--------------------------------------------------------------------------
E(RM062X) + Thermal correction                 =     -274.666928 Eh
Sum of electronic and thermal Free Energies    =     -274.666928 Eh
```

Now, you're probably not a physical chemists if you're on this page, so let's convert these numbers to a more common unit and calculate our A-value (remember $1\ E_h \approx 627.5\ \mathrm{kcal\ mol^{-1}}$):

```
eqMeCyhex:
E(RM062X) + Thermal correction                 =     -172355.27 kcal/mol
axMeCyhex:
E(RM062X) + Thermal correction                 =     -172353.50 kcal/mol
--------------------------------------------------------------------------
A-value = deltaG = E(axMeCyhex) - E(eqMeCyHex)  =     1.77 kcal/mol
```

So we get a relative energy of 1.77 kcal mol^{-1}, which is in excellent agreement with the literature values for the A-value of a methyl group.[2]

> These are the kinds of comparisons that underscore much of computational organic chemistry. Even computations of complex mechanistic pathways are reducible to calculations of relative energies.

For more practice, try calculating other A-values and checking them with their experimental values. Then, when you feel like you're ready, give this problem a shot.

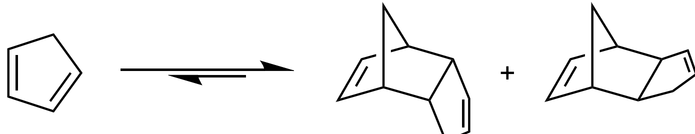
References

(1) IPUAC Gold Book: A-value
(2) The Reich Collection: A-values

Problems

The smelly dimer problem

Cyclopentadiene is a fairly common reagent in both organic and organometallic synthesis, however its use is complicated by its facile dimerization into dicyclopentadiene *via* a Diels-Alder cycloaddition.



Your PI Prof. Batman and his lab manager Dr. Robin ask if you could use your newfound computational skills to study this reaction. In particular, they'd like to know:

- The relative energies of the *exo*- and *endo-*dicyclopentadiene isomers,
- The free energy of dimerization (ΔG_{rxn}), and
- The kinetic product[1] of the reaction (unless you've done completed the Long Course you won't be able to do this one just yet)

See if you can do this one yourself!

Check your work with the solution in the code repo or literature values.[2,3,4]

References

(1) Thermodynamic versus kinetic reaction control

(2) Dicyclopentadiene

(3) *J. Loss Prev. Process Ind.* **2016**, *44*, 433–439

(4) *Ind. Eng. Chem. Res.* **2019**, *58* (50), 22516–22525

The E-Z isomer problem

Your advisor wants you to compute ground state energies for the two isomers of 3-(4-nitrophenyl)but-2-en-2-yl triflate. First, run a standard optimization/frequency calculation for the isomers at the `M062X/def2svp` level of theory. As always, you can check your work at the code repo.

(Z)-3-(4-nitrophenyl)but-2-en-2-yl triflate (E)-3-(4-nitrophenyl)but-2-en-2-yl triflate

> Do not wait for these calculations to finish. Even running on the cluster, they will take a while. The best thing to do is to submit them and then go set up a reaction or something. If you're running on the `chem` nodes you can expect these jobs to take around 1 to 2 hours.

1) Specifying different built-in basis sets for different atoms

You take your results back to your advisor who doesn't seem very satisfied. They tell you to make sure the substituents are right by beefing up the basis set on the heteroatoms.

Modify your calculations to use the triple ζ basis set `def2tzvp` on all heteroatoms and `def2svp` on C and H. Give it a shot before checking your input files against those at the code repo.

2) Assigning built-in basis sets to individual atoms

Still unhappy with the results, your advisor tells you to re-run the computations, but this time placing diffuse basis functions on **just the nitro group atoms**. The basis set `def2tzvpd` which has the diffuse functions added to `def2tzvp` isn't built in to `Gaussian` so you figure that another similarly large basis set with diffuse functions `aug-cc-pvtz` that is built-in would work just as well.

Run your computations again, this time use `aug-cc-pvtz` to describe the N and *only the two O of the nitro group*. Keep everything else the same i.e.,

- C H: `def2svp`
- S F O(trifyl): `def2tzvp`
- N O(nitro): `aug-cc-pvtz`

3) Incorporating external basis sets into `Gaussian` calculations

After a few second guesses, you're unsure of whether or not `aug-cc-pvtz` is really a suitable substitute for `def2tzvpd` ; you also realize that your computations are taking quite a while, and the other group members are starting to get upset that you're hogging the new compute nodes. You decide to try using `def2tzvpd` instead. Re-run your calculations, this time use `def2tzvpd` instead of `aug-cc-pvtz` to describe the N and *only the two O of the nitro group*. Keep everything else the same i.e.,

- C H: `def2svp`
- S F O(trifyl): `def2tzvp`
- N O(nitro): `def2tzvpd`

Remember that `def2tzvpd` is not built into `g16` so you'll have to get the basis set from the Basis Set Exchange.[1] There are two ways to accomplish this task; see if you can figure them both out before going to the code repo!

Hint: These are the diffuse functions from `def2tzvpd`

```
N 0
S    1   1.00
      0.68441605847D-01      1.0000000
D    1   1.00
      0.12829642058          1.0000000
****
O 0
S    1   1.00
      0.70288026270D-01      1.0000000
P    1   1.00
      0.51112745706D-01      1.0000000
D    1   1.00
      0.14696477366          1.0000000
```

Key takeaways

Once your jobs have finished, extract the corrected energies from your results. I've placed mine in the table below if you're just following along (the input/output files are available in the code repo).

Using these energies can you justify the product distribution observed in the triflation of 3-(*p*-nitro)phenyl-2-butanone (products 5g/6g) in this paper?[2]

Basis Set	**Isomer**	**Energy / kcal mol^{-1}**	**ΔG(E<->Z) / kcal mol^{-1}**	**Total Computation Time / min**
def2svp(all)	*E* *Z*	-973863.597 -973862.450	1.147	37 47
def2tzvp (SNOF) def2svp (CH)	*E* *Z*	-974540.402 -974539.491	0.911	87 92

Basis Set	Isomer	Energy / kcal mol^{-1}	ΔG(E<->Z) / kcal mol^{-1}	Total Computation Time / min
aug-cc-pvtz (nitro) def2tzvp (trifyl) def2svp (CH)	*E* *Z*	-974540.579 -974539.693	0.886	120 134
def2tzvpd (nitro) def2tzvp (trifyl) def2svp (CH)	*E* *Z*	-974542.096 -974541.205	0.891	99 110

There are some key takeaway from the data above:

1. The answer never formally changes. In all cases the *E* isomer, as we expect, is more stable than the *Z* isomer.
2. The caveat is that depending on our choice of basis set, we **do** see changes in the relative energies of the two species; namely, the relative energies tend converge with increasing basis set size.
3. Nevertheless, its important to not lose sight of the forrest in the trees. Look again in the predicted relative energies. In the "worst" case we there is a 1.15 kcal mol^{-1} difference between the isomers; in the "best" case, only 0.89 kcal mol^{-1}. The difference in these two predictions is a mere 0.25 kcal mol^{-1}; it is simple to use this as justification for more computationally intensive calculations, however consider for a second the *experimental* implications of this value.
   A reaction under control of a 1.15 kcal mol^{-1} ΔΔG would predict 11% minor product formation, while one with a 0.89 kcal mol^{-1} ΔΔG would predict 16% of the minor product; barely something to split hairs over.
4. Computational time, while relatively cheap, is not free. The difference in relative energy that comes from using the diffuse augmented basis sets is a whopping 0.025 kcal mol^{-1} (or *25 thousandths of a kcal*). If this number seems small to you now, consider it in the context of the computational time.
   Augmenting *just three atoms* in our molecule with the diffuse functions of `aug-cc-pvtz` increased our total computational time from 87 min to 120 min, a 40% increase in resources. All for 0.025 kcal mol^{-1}. Realizing that a ΔΔG of 0.025 kcal mol^{-1} erodes a selectivity by less than 1%, it seems a little silly. Note that DFT scales in cubic time ($\mathcal{O}(n_e^3)$) with respect to the number of electrons in your system so as the size of your molecule increases this issue will only get much worse.[3] Take a look at this paper for a discussion on the necessity of diffuse functions.[4]

> Computational chemistry is all about choosing which assumptions to make because all models must make assumptions, i.e., **there is no free lunch**. In the most rigorous sense we can, we are always searching for the *good enough* method that balances *chemical accuracy* with *computational cost*.

Resources

(1) Basis Set Exchange
(2) Vinyl cations. 12. Mechanism of reaction of cis- and trans-3-phenyl-2-buten-2-yl triflates. Evidence for vinylidene phenonium ions by Peter J. Stang and Thomas E. Dueber *J. Am. Chem. Soc.* **1977**, *99* (8), 2602
(3) Max Hutchinson on CompSci Stack Exchange
(4) Is the Use of Diffuse Functions Essential for the Properly Description of Noncovalent Interactions Involving Anions? by Antonio Bauzá, David Quiñonero, Pere M. Deyà, and Antonio Frontera *J. Phys. Chem. A* **2013**, *117* (12), 2651

Resources

`g16` routing line templates

This section provides general templates for the most common Gaussian jobs. Minimal explanation is provided and **it is strongly advised** that you read the respective sections for these calculations.

> To run any of the optimizations below using Gaussian's generalized internal coordinates give OPT the `GIC` keyword.

Split-basis calculations

In all cases a split basis set has been utilized to reduce computational costs as our group typically works on relatively large systems. In the following “Basis Set (HL/LL)” refer to the high- and low-level basis sets, respectively. If your systems are small enough or computational resources are considerable enough to treat the entirety of the system with a single basis set then that approach is preferable. As we've covered already, a final single point energy calculation is redundant in this case.

> N.b. SCF energies computed using two different basis sets are incomparable.

Ground State Geometry Optimizations/Energy Calculations

Step 1: Optimize a ground state geometry

```
#N Level of Theory/Basis Set (LL) OPT FREQ=NoRaman
temperature=Temperature Integral(Grid=UltraFine)
```

Step 2: Calculate ground state single point energy

```
#N Level of Theory/Basis Set (HL) SP Integral(Grid=UltraFine)
```

Transition State Optimizations/Energy Calculations

Step 1: Optimization around the active atoms

```
#N Level of Theory/Basis Set (LL) OPT=(TS,CalcFC,ModRedundant,NoEigenTest)
```

Then, at the end of the input file, add: `B [Atom 1 number] [Atom 2 number] F`
Where atoms 1 and 2 will be frozen in the geometry optimization
e.g. `B 74 94 F`

> N.b. there are spaces between each parameter and the next.

If optimizing using Generalized Internal Coordinates (GIC)

```
#N Level of Theory/Basis Set (LL) OPT=(TS,CalcFC,AddGIC,NoEigenTest)
```

At the end of the input file, add: `CoordinateName(freeze)=R(Atom 1 number,atom 2 number)`
Where atoms 1 and 2 will be frozen in the geometry optimization
e.g. `BrC(freeze)=R(54,46)`

> N.b. all coordinates must have a unique name

Step 2: Geometry optimization of the active atoms

```
#N Level of Theory/Basis Set (LL) OPT=(TS,CalcFC,NoEigentest)
freq=NoRaman temperature=Temperature Integral(Grid=UltraFine)
```

Step 3: Calculation of single point transition state energies

```
#N Level of Theory/Basis Set (HL) SP Integral(Grid=UltraFine)
```

QST Transition State Optimizations/Energy Calculations

Step 1: Optimize ground state geometries for reactant ensemble and product ensemble

If using `QST3` , also optimize the best guess for the transition structure.

```
#N Level of Theory/Basis Set (LL) OPT FREQ=NoRaman
temperature=Temperature Integral(Grid=UltraFine)
```

Step 2: Quasi-Newton Transition Structure Search

It is strongly advised to save a checkpoint file for these calculations as you'll need it for the intrinsic reaction coordinate calculation to verify the optimized structure.

```
#N Level of Theory/Basis Set (LL) OPT=(QST2/QST3) FREQ=NoRaman
temperature=Temperature Integral(Grid=UltraFine)
```

Step 3: Calculation of single point transition state energies

```
#N Level of Theory/Basis Set (HL) SP Integral(Grid=UltraFine)
```

Intrinsic Reaction Coordinate (IRC) calculation for verification of transition structures

An IRC requires initial force constants to proceed. The easiest way to do this is to use the ones in the checkpoint file from the previous frequency calculation using option `rcfc`, but if you didn't save the checkpoint file from the TS optimization then pass the option `calcfc` to calculate force constants at the beginning of the calculation.

```
#N Level of Theory/Basis Set (LL) IRC=(rcFC/calcFC)
temperature=Temperature Integral(Grid=UltraFine)
```

A collection of papers/webpages/blogs/lecture slides/etc... that I've amassed over the years (in no particular order)

The Basis Set Exchange: A public library of basis sets maintained by MolSSI (Virginia Tech) and Environmental Molecular Sciences Laboratory (PNNL)
Read more about it!

J.C. Corchado and D.G. Truhlar on Dual-Level Methods for Electronic Structure Calculations

Frank Jensen talking about the Pople basis sets on Stack Exchange

Seminar slides from Mikael Johansson of the University of Helsinki on wave function methods and DFT

Thirty years of density functional theory in computational chemistry: an overview and extensive assessment of 200 density functionals by Narbe Mardirossian and Martin Head-Gordon *Molecular Physics*, **2017**, *115* (19), 2315
This review provides a comprehensive benchmarking of over 200 DFT functionals over several datasets representing different computational datatypes.

Is the Use of Diffuse Functions Essential for the Properly Description of Noncovalent Interactions Involving Anions? by Antonio Bauzá, David Quiñonero, Pere M. Deyà, and Antonio Frontera *J. Phys. Chem. A* **2013**, *117* (12), 2651
Discussion on the necessity of diffuse basis functions in anionic calcualtions

Common Gaussian error messages

The blog of Dr. Joaquin Barroso-Flores (Instituto de Química, UNAM, Mexico City, MX)

Incredibly helpful for general troubleshooting

Group meeting slides from Steven McKerrall (Baran Lab, TSRI, CA, USA)

Computational Chemistry 2 by Prof. Hendrik Zipse (LMU, Munich, DE)

Advanced topics in computational chemistry