**MolSSI Day 2 Activity: Putting it all together**

*On Windows*: Make sure you are using the “Anaconda prompt” which will give you access to the git and python commands. However, you will not be able to use normal Linux commands such as ls or rm for working with files (instead you use DOS commands such as dir and del, respectively).

*On Mac*: All commands will work normally in the Terminal.

**Introduction**

In this activity, you will do the following:

1. Fork a GitHub repository (containing a Python script that runs OpenMM) to your personal account.
2. Clone the repository to your local computer.
3. Understand the OpenMM script, run the simulation, and visualize results.
4. Implement a new feature into your OpenMM script, allowing the simulation to do something new.
5. Commit your changes and push them to your personal repository on GitHub.
6. Create a pull request for your features to be added back to the original repository.
7. (Optional): Resolve merge conflicts that result from multiple features being implemented simultaneously.
8. **Fork the original GitHub repository to your personal account.**

Log into your GitHub account, visit the original repository located at <https://github.com/leeping/molssi-synthesis> and click the Fork button on the upper right.

This will make a copy of the original repository under your personal account. It contains all of the code in the original repository; the only difference is that it knows it’s derived from the original one.

Your *forked repository* is located at https://github.com/your\_username/molssi-synthesis .

*Note:* You will be able to make changes to your forked repository, but not the original repository. This is a key aspect of how GitHub manages multiple people contributing features to the same code.

1. **Clone the repository to your computer.**

Make sure you’re in the molssi-day2 folder on your Desktop.

Run the command:

git clone https://github.com/your\_username/molssi-synthesis.git

This will create a folder called molssi-synthesis in the folder where you ran the command.

Enter the folder and you will see several files - trpcage.pdb, run\_openmm.py, README and vmd.rc.

*Note*: These two files are created from some OpenMM tools with a few tweaks added by Lee-Ping. trpcage.pdb is generated using the PDBFixer tool and providing the PDB ID 2JOF (the Trp-cage miniprotein); the coordinates are moved into the simulation box between (0, 0, 0) nm and (3, 3, 3) nm. Otherwise, OpenMM will shift the coordinates when the simulation begins and it will be confusing.

1. **Understand the script, run the simulation, and visualize results.**

trpcage.pdb contains a solvated PDB structure of the Trp-cage miniprotein. It’s a bit small for a protein simulation, but designed so that we can get results fast.

*Note:* Up until this point, we’ve been doing a lot of coding in Jupyter notebooks (called IPython notebooks if you’re old school like me.) They have the advantage that you can write code interactively, but sometimes we want to make “traditional” scripts or programs that run from start to finish. Here you have two options for running the code. From a command prompt inside the folder, run the script using the command:

python run\_openmm.py

Or you may paste the script into the notebook and run it. It’s best to organize things as:

Cell 1: Import statements

Cell 2: The lines pdb = … through platform = … (the simulation setup),

Cell 3: The rest of the code (running the simulation).

*At this point we can talk about the various lines in the script, depending on how much you know from previous workshop activities.*

When the script is running, it will print information to the screen about the simulation time, simulation speed, potential energy, temperature, density, and a progress indicator.

After the script is finished running, you may view the trajectory in VMD. Open VMD from the Start or Applications menu, Go into the “File ➝ New Molecule” menu, and select trpcage.pdb. ***With the molecule selected in VMD Main***, go into the “File ➝ Load Data into Molecule” menu, and select trajectory.dcd.

VMD will display your PDB and trajectory file as a “molecule object” with 101 frames - 1 from the PDB file and 100 from the trajectory file. Note that if you do not load the PDB file first, the trajectory cannot be loaded because it doesn’t contain any topology information (i.e. the list of atoms, residues, and bonds).

*Optional – visualization:* By default, VMD will use the “Lines” representation and show all the atoms, which isn’t that useful to see what’s going on. In the VMD main menu, open the “Graphics ➝ Representations” menu, and change the selection to “protein” to highlight just the protein. You can also change the representation to CPK, Licorice, or VDW to view the atoms in a more “three-dimensional” way.

VMD also has a graphical representation for secondary structure, which highlights alpha helices, beta sheets, turns and random coils. It is very cool but it has a flaw. The secondary structure that is shown when you play the trajectory is always the first frame - in other words, it does not update the secondary structure in your visualization even if the simulation changes it. To fix this, I’ve provided a VMD startup file in the GitHub repository that provides the command ss 0, which updates the secondary structure at each step. To use the startup file, you need to place it into the correct location, which depends on the OS: On Windows, the file is already named properly.

On Windows, move the file to vmd.rc to where VMD is installed

C:\Program Files (x86)\University of Illinois\VMD

and overwrite the vmd.rc file that’s already there,

On Mac, move the file to ~/.vmdrc

(i.e. move to your home folder, and rename it to .vmdrc, which also hides the file.)

Restart VMD, and load the PDB and DCD file as before. If the file is named properly, the console will print out “LP's VMD startup file successfully loaded” in the console. Then you may type ss 0 into the console to get a secondary structure representation that properly updates when the animation is played. Also you can type align backbone to align all of the structures to the original frame using the protein backbone atoms. This option will only be there if you use my custom startup file. Otherwise you will have to use the RMSD trajectory tool, which is a bit more tedious.

1. **Implement a new feature.**

The cool thing about OpenMM is that your simulation is inside of the Python script, so you have a lot of flexibility to customize it. I encourage you to get creative here and think about different possibilities. Here I provide two examples of interesting things we can do. **You only need to implement 4.A or 4.B below, but not both**. Once again you may use either the Jupyter notebook to help develop your feature, but ultimately we want it to be part of the Python script.

*Note*: When coding in Python, there is a nifty tool that allows you to interact with a running program at any desired location in the code. This is made possible by adding these two lines into the desired location; running the script from the command prompt will pause the script there and give you an interactive prompt.

import IPython

IPython.embed()

*Note*: If you develop in the Jupyter notebook, paste your new codes into the Python script when you’re done. This is because Git works best with codes and script, rather than more complicated file formats such as notebooks.

**4.A. Option A: Implementing an external force that unfolds the protein.**

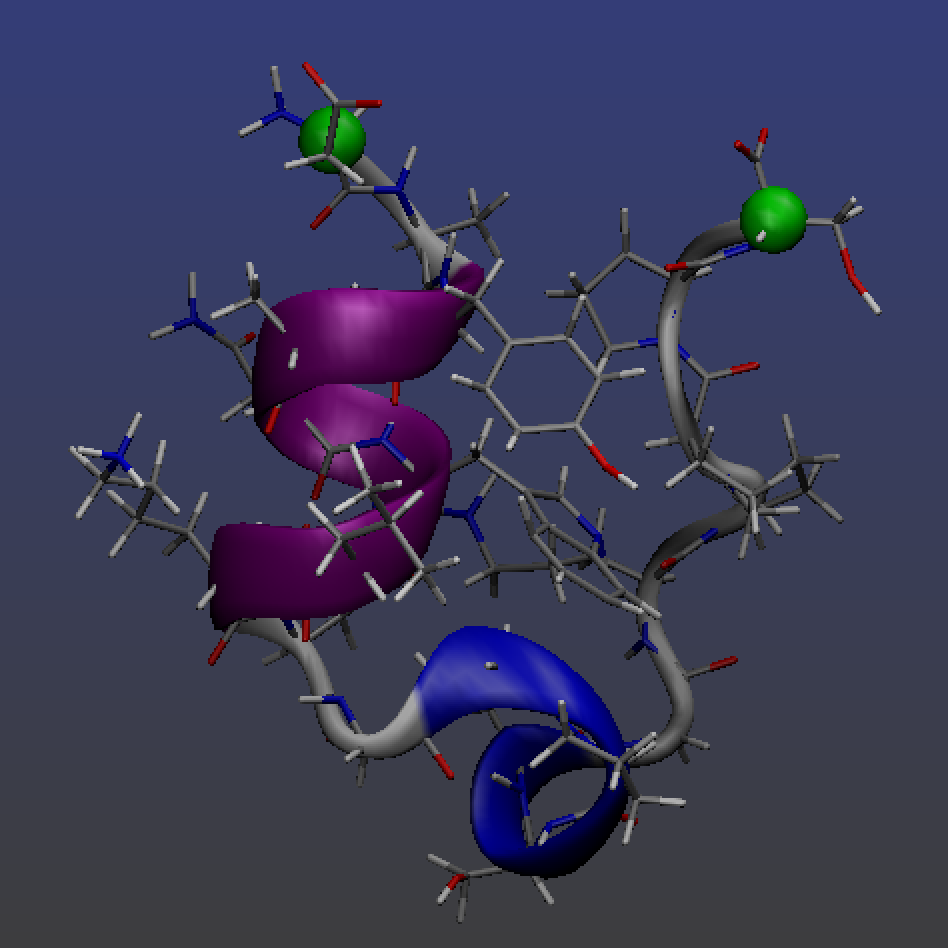
Because Trp-cage is a stable protein in solution, running a molecular dynamics simulation will just cause the protein to tumble and vibrate around - not too interesting if you ask me. But suppose we want to know how large of a force is needed to pull the protein apart?

Imagine applying a spring force between the alpha carbon atoms of the first and last amino acid. Experimentalists do similar things in the lab using *optical tweezers*, but they cannot apply precise forces to pairs of atoms; instead they chemically modify the target amino acids, chemically link polystyrene beads to them, and use lasers to apply the forces to the beads. In our simulations we can apply the forces on the atoms themselves. :)

The alpha carbon is the carbon atom on each amino acid that the side chain is bonded to. Open trpcage.pdb in a text editor. Look at the third column containing atom names - the line with CA in the third column is the alpha carbon. The sixth column is the *residue ID* which labels the amino acid or water molecule number. Residue IDs start from 1.

In the graphical representations menu, create a new representation using VDW where the selection is name CA and resid A B, where A B stand for the first and last residue IDs of the protein. (How many amino acids does this protein have? In the text editor, scroll down until the residue ID becomes HOH, which is the residue name of water. The last residue that is not labeled HOH is the last amino acid in the protein.)

Locate the atom indices of these two atoms. In the VMD main menu, go to “Mouse ➝ Label ➝ Atoms” and click on the alpha carbons. Information will be printed to the console including the index of the atom (the sequential atom number that starts from zero). It should also show other information such as the atom name and residue ID that you can use to double-check that you clicked on the right atom. If clicking on atoms is difficult, you may turn off the other graphical representations by double-clicking on it in the Graphical Representations window.



**Figure 1:** Three graphical representations of atoms in Trp-cage - NewCartoon with secondary structure coloring, licorice with atom coloring, and alpha carbons of the first and last amino acid highlighted in green.

There’s a variety of documentation you can use to implement the spring force between these particles. The class you need is called **CustomBondForce.** This class allows you to implement any force between selected pairs of atoms, it doesn’t have to represent a covalent bond.

To learn about how to use this class, visit the Python API (application programing interface) documentation located at <http://docs.openmm.org/7.1.0/api-python/index.html> , click on Forces, and then CustomBondForce.

(*Note*: The User Guide also teaches you how to include CustomBondForce in the force field XML file; not needed for this tutorial. Here we’re creating the CustomBondForce in the Python script directly, which gives us a higher degree of control.)

Write the following code after the system object is created, but before the simulation object.

1. Create the CustomBondForce with a line like this:   
   custom\_bond = mm.CustomBondForce(“...”) where … is the energy expression that you want to use for the particles. Here we’ll use a harmonic spring force where the energy expression is 0.5\*k\*(r-r0)^2 ; k is the spring constant and r0 the equilibrium distance. Custom forces are really cool because the user can specify any functional form they want, and OpenMM will actually generate the code for the simulation at runtime.
2. The variables k and r0 are called global parameters. You need to set the force constant using a line such as custom\_bond.addGlobalParameter(“k”, 100) where 100 is the value of the parameter in OpenMM’s unit system (kJ mol–1 nm–2 for bond force constants). Add a line that sets the r0 parameter to 2 nanometers (OpenMM uses nanometers as the default unit).
3. Call the custom\_bond.addBond(a1, a2) function to add a bond between the two particles of interest that are indexed a1 and a2. Use the atom indices that you looked up earlier in this step.
4. Add the force to the System before the Simulation object is created, using a line such as system.addForce(custom\_bond) - same as the syntax that was used for adding the Monte Carlo Barostat.
5. In your Python script, incorporate the above code into a function AddSpringForce(a1, a2, k), so that you may add a spring force between any two atoms with any provided spring constant k. Write some comments to document your function. Add a line to your script that calls the function.

(Note: As you become more experienced, you can always define the function first, and then start add code to it - rather than writing the code first and then putting it into a function.)

Now let’s run the script to see if your feature works.

1. Copy your existing trajectory file to a backup so you can compare the differences in the results.
2. Run the simulation using force constants of 10, 100, and 1000. If your computer is fast enough, you can run for 100,000 or even more steps. If you are using the notebook to run simulations, make sure to set up a new system whenever you modify the spring constant - otherwise you risk adding multiple custom forces to the same simulation, which will be confusing.
3. Use VMD to load the new trajectory and measure the distance between atoms using Mouse ➝ Label ➝ Bonds. How far apart do the atoms get in each simulation? Optional: You may compare results from pulling on different atoms within the protein, such as alpha carbons on residue IDs 5 and 15.
4. Superimpose the final structure of your “stretched” simulation with the original PDB file. This can be done by first running the align backbone and ss 0 commands in the VMD console. Next in your Graphical Representations, highlight the NewCartoon representation and click the Trajectory tab. Under “Draw Multiple Frames”, type in 0:100:100 where you can replace 100 with the total number of frames minus 1. Now the first and last frames will be displayed simultaneously.

**4.B. Option B (advanced): Zeroing out the charges on the protein**

*This feature requires a bit more programming knowledge to implement compared to 4.A. It’s also described in less detail, so try at your own risk!*

We might be interested in knowing the role that the partial charges play in keeping the protein stable. One might investigate this problem by zeroing out the charges on the protein. There are two ways to do this - you could edit the force field XML file and set all the charge parameters to zero, or you could set the charges to zero in the NonbondedForce (part of the System object). Here we will take the second route.

1. Create a function called ZeroProteinCharges(), so that you zero out the protein charges by calling this function. Make sure this function is defined after the system object has been created, but before the Simulation is created. Write the rest of the code into this function.
2. The System contains a number of Force objects of different kinds (HarmonicBondForce, HarmonicAngleForce, NonbondedForce etc.) We want to change some parameters in the NonbondedForce. To do this, *we will iterate over the list of Forces in the System* and see which object is an instance of the NonbondedForce class.

The method that iterates over the Forces is described in the API documentation.

Start from the API documentation: <http://docs.openmm.org/7.1.0/api-python/index.html> ,

click on “Core Objects”, go to System, and look through the methods for the right one.

After identifying the correct method, call it in your function to obtain the list of forces, and then write a Python loop to iterate over the items in this list. Inside the loop, check if the Force object is an instance of the NonbondedForce class using the syntax:

if isinstance(force, mm.NonbondedForce):

1. Inside the if statement, you are now ready to iterate over the particles and set the charge parameters to zero. First determine the number of atoms in the protein (by reading the PDB file), and write a for loop from zero up to the number of atoms in the protein.
2. Using the API documentation, look up the method in NonbondedForce for getting and setting the nonbonded parameters for particles.
3. We wish to set the charge parameters for the atoms in the protein to zero. Inside your for loop over protein atoms, first get the charge, sigma, and epsilon parameters for the atom (make sure to set three return values), and then set the parameters for that atom to zero, sigma, and epsilon respectively.
4. Follow steps f-i in Option A.
5. **Now that you have developed your feature, commit your changes on the local machine, and push them to your repository fork on GitHub.**

The commands are:

git commit -m “Type any commit message you want”

(Type in a commit message)

git push origin master

Here, *origin* stands for the remote repository, i.e. your repository fork on GitHub. You can request to display the remote repository names and their URLs using the command:

git remote -v

Your changes are now uploaded to your repository fork on GitHub, but they are not part

of the master branch yet.

1. **Create a pull request on GitHub for your feature to be added to the original repository.**

At this point, your code has a new feature that isn’t part of the original repository; you may request the owner to incorporate your code changes using the pull request button. Click the pull request button to initiate a conversation thread; the owner may accept the pull request right away (updating the original repository with your codes), or ask you to make more changes.

(Demo: Show how additional commits to your forked repository also show up in the PR.)

1. **(Optional) Dealing with merge conflicts.**

Supposing you spent a month developing your new feature, and during this time the original repo went through some substantial changes (for example, the owner of the original repo may have updated it themselves, or accepted a PR from someone else).

If the other changes are *in a different place* from your changes (i.e. in different files in the project, or in different parts of the same file), Git is smart enough to merge the changes together into a combined version, which will then be committed; this is called a *merge commit*.

However, if any changes are occur in the same place in the codes, then a *merge conflict* is said to exist. Because you are the person wanting to add your codes to the original repo, you are now responsible for merging the codes manually.

1. Add the original repo as a second remote repository using the command:

git remote add upstream https://github.com/leeping/molssi-synthesis.git

Here “upstream” is just a common name that is used to denote the original repository.

1. *Pull* the changes from the original repo. This will result in a merge conflict that you will need to resolve.

git pull upstream master

1. Git will tell you which files are in a conflict, and you will need to open a text editor and edit the codes until the conflicts are resolved. Looking at the codes in the text editor, you will now see markings like this:

<<<<<<< HEAD

[Codes for your new feature are here]

=======

[Other changes to the master branch that conflict with your codes are here]

>>>>>>> master

The markings ”<<<<<<< HEAD”, “=======”, “>>>>>>> master” are not Python code, but inserted by Git to highlight what parts need to be fixed. Now it’s your job to arrange your changes and the other changes to make sure everything makes sense, and after you’re finished you should delete the markings as well (otherwise Python will see it as a syntax error).

If you messed up the merge (this happens very often), you can reset everything back to the beginning using the command git reset --hard HEAD, and then run   
git pull upstream master again.

After you’re happy with the merged codes, commit your changes with git commit -a. You won’t need to enter a commit message because Git already writes one for you (Merge branch “master” of github:leeping/molssi-synthesis.git). Just save the commit message.

Now push the changes to your forked repository with git push origin master. *For those curious*: You cannot push changes directly to git push upstream master because you don’t have write permissions to the original repo. The whole point of the pull request system is to give you a way to contribute your changes through communication with the original repo’s owner, rather than allowing you to write to the original repo directly.

Your changes should now be reflected in the pull request, stating that everything is ready to merge. The owner can now accept the pull request and you are done!1