

Physics 230: Computational Physics Lab 1

Git Tutorial

During the Course Project, you'll be working in a group to try to develop a single coherent set of calculations, codes, results, and a presentation. This might be spread over multiple interdependent *Mathematica* notebooks. *How can multiple people work on a single project and keep things organized? What if they want to work on the project remotely, but still coordinate?*

It is best to start a code with a simple but working example, slowly adding new parts individually and sequentially. *How do you keep track of these different codes (especially when working with others)? What happens when you start working on a new part and don't finish it before wanting to try a new part?*

The time and thought it takes to write a computer program is far more valuable than the memory needed to store it. Imagine working on your project and then finding that your hard drive died... even though the memory cost is trivial, backing up is better. *What's an efficient way to back up your codes?*

The answer to all of the above questions is Git. Git is the combination of workflow management for a group, a "version control system", and an online backup system. Git can be used for any combination of these three needs and it is certainly not the only way of accomplishing any or all of these goals. But Git is currently the *de facto* standard for most scientific computing projects in Physics and within several BYU Physics & Astronomy Department research groups. For example, we'll be using the Department's GitLab server set up for the purpose of enabling Git-based collaboration.

Git is not hard to learn, but there are many small parts that take some practice. This tutorial will introduce you to the main pieces of Git you will need for your Physics 230 Project. After introducing some basic terminology and concepts, you'll practice using Git in the process of setting up the files for your project. This tutorial is meant to be self-contained and used for future reference for this class. There are many Git tutorials online (e.g., <https://git-scm.com/book/en/v2>) that could be used as supplements and extensions as we will not be covering several important concepts.

We will be setting up our project organization following the recommendations in *Good Enough Practices for Scientific Computing (GEPSC)* which are pretty standard. The first recommendation under "Project Organization" is *GEPSC 4a: "Put each project in its own directory, which is named after the project."* Decide now with your partner what the name of your project directory will be. These are usually 1-2 words, ~10 lower case characters, no spaces or special characters, e.g., `kbokboencounters` or `threebodyJ2` or `haumea_sats`. Don't use `230project` (or similar) since that's not descriptive.

Git Terminology

It will be easier to explain Git by first defining some terms. These are the definitions in the context of this tutorial. (If you're very familiar with Git, you could skim this part.)

- Repository or "repo" – a collection of all the project files in a single directory
- The "local repo" is the copy of the repo (the project directory) on your computer
- The "remote repo" is the online copy of the repo on the Department's GitLab.
- The process of creating a new "version" of your repo is called "committing" and the version is called a "commit".

- Here's an analogy to what you're used to. When you hit "save" on a *Mathematica* notebook file, you have created a new version of that file that is accessible in that exact state later on, even if you close the program. When you save a file, the previous saved version is automatically replaced with the current file. Git is different: every time you "commit" (like hitting "save") you do **not** remove the previous commit. All previous versions of the file are kept. This is like hitting "save as" and using a new filename, but Git automatically keeps track of all the different "commits" and gives them their own name (not "project_v1", "project_v2", etc.). Each Git commit is a snapshot of the project, Git's job is to track and manage all these snapshots.
- Only commits can be backed up on the remote repo and/or shared with other users.
- To "modify" is to change a file in your local repo. When you open up a *Mathematica* notebook, make some changes and then save it, the file is considered "modified". This is **not** the same as "committing" the file. Changes are saved on your local files, but aren't committed (and thus not backed up or shared).
- Before you can commit the changes made in a modified file, you must "stage" the file. Staging a file tells Git that you are ready to record the changes
- Thus, each individual file in your repo can be in one of three states:
 - Committed – unchanged since the previous "commit" (e.g., version)
 - Modified – changed since the previous "commit", but you haven't yet told Git that you want to record these changes
 - Staged – changed and you've told Git that you are ready to "commit" the changes.
- With these aspects described, I'll now be more specific: a "repo" is the current version of the files in the current "commit". If you have modified or staged files on your computer, it's best to consider these as not yet part of the repo.
 - Generally, you modify and stage files in your local repo, make a commit, and then backup/share the commit via the remote repo. That is, you don't modify or stage files on the remote repo.

To make sure you understand these, see if you can correctly fill in the blanks of a conversation between two Git-savvy Physics 230 students. Use words from this list: [repo, repoed, can, can't, commit, committed, stage, staged, modify, modified].

Okay, I just shared the fourth _____ on GitLab. I added constants.nb to the _____. The main changes are such-and-such. I have also made changes to orbitsolver.nb, but I didn't _____ these, so these changes weren't _____ so you _____ see them yet.

Check your answers with a TA. Much of Git is getting a hang of the concepts and jargon, so you shouldn't go forward until this makes some sense and be sure to refer back to these definitions if you are ever unsure. We'll cover the terminology for interacting with the remote repo below.

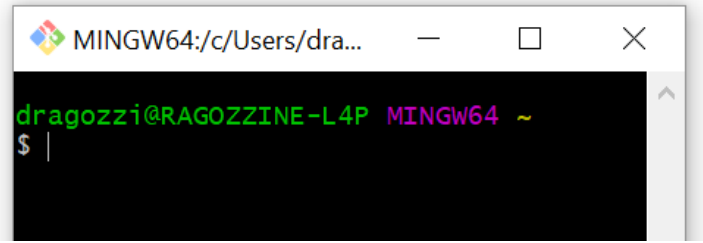
Using Git Bash and the Command Line

The best way to learn Git is to practice on a simple test case. Although the underlying Git and terminology is the same, there are many ways that Git can be implemented in practice. There are "only" ~dozens of different things that Git can do, so there are many "Graphical User Interfaces" (GUIs) that have sprung up to implement it. (It's worth noting now that Git itself is completely free.) Many GUIs are free and work in various environments (Windows, Mac, UNIX) and can be helpful. But the most fundamental way to use Git is to use the "Command Line" (and giving you familiarity with a command line tool is a nice side effect since there are other computational physics programs that use it).

If you are familiar with a Git GUI, you still need to follow this tutorial on using Git at a Command Line... however, when it is time for you start using Git for your project, you can use whatever method you like.

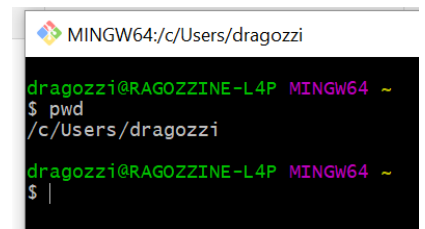
If you want to use your own computer for this tutorial, you could pause here and install the Git Command Line following the instructions online here: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git> . Note that "Terminal" is an application in Mac and UNIX.

(If you're familiar with the Command Line, you can skip this paragraph.) A Command Line tool is a program that you interact using text and text only. The text that you type in is called a "command". As with *Mathematica*, you have to type the command *exactly* as specified, or else it usually won't work. Commands are typically lower case... spaces, dashes, and slashes have very specific meanings. You hit Enter to execute a command. The place where you type the command is called the "prompt". In the example to the right, the \$ is the prompt (the vertical line blinks). Prompts often include helpful information like the username ("dragozzi") and the computer name ("RAGOZZINE-L4P"), but you can ignore these, keeping in mind that they will be different for you.



A very common Command Line environment is called "bash" and is used on many UNIX/Linux machines. On Windows computers (like those in the Lab), we have installed a version of bash that can run Git commands. Fire it up by typing "Git Bash" in the Windows search (magnifying glass). A window like the above screenshot should pop up.

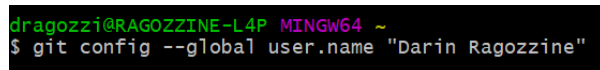
Let's practice using Git Bash. At the prompt, type "pwd" and hit Enter – pwd is short for "present working directory", i.e., where in the filesystem you are. Git Bash will respond by printing the directory where you are located. You can see in the screenshot, I'm told that Git Bash is in the directory /c/Users/dragozzi (yours will be different). After completing the command, a new prompt appears.



To run Git commands, you'll start by typing "git" and then the command. Normally, there's a few Git setup commands that you should do before starting, but we'll just do two. First, tell Git what your name is. The command for this is

```
git config --global user.name "Your Name"
```

where you replace `Your Name` with your full name. Here `git` means that it's a Git command: "config", `--global` is called a "flag" telling `git` how to interpret the command. `user.name` is what you are configuring and the double quotes surround the text that will become the global user name. If you try this and get an error, then double check that you typed everything correctly, including spaces. Use a similar method to set up your email:



```
git config --global user.email "YourEmail@example.com"
```

In our modern world, where you can ask Google questions with your voice, it seems antiquated to interact via text alone. No argument there... the only thing more old-fashioned than this is using punch cards! Even so, command line interfaces are still common and they allow for a level of precision and customization that make even Siri and Alexa jealous. 😊

A Git Example

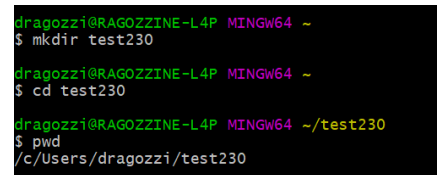
Okay, let's go. We're going to do a simple local repo just to get some practice before going to the more complicated group-shared remote repo.

Note that you may need to tweak the instructions in this tutorial if you don't follow it exactly (for whatever reason). You'll

You can turn any directory into a Git repository. For our test, let's **make** a new empty **directory** and then **change directories** so that Bash is in that directory. Type the following commands, remembering to hit Enter after each one.

```
mkdir test230
cd test230
pwd
```

This should create a new directory for you to work in, move you to that directory, and then show you that you are in that directory.



```
dragozzi@RAGOZZINE-L4P MINGW64 ~
$ mkdir test230
dragozzi@RAGOZZINE-L4P MINGW64 ~
$ cd test230
dragozzi@RAGOZZINE-L4P MINGW64 ~/test230
$ pwd
/c/Users/dragozzi/test230
```

Now, let's tell Git that this directory is the home to a new repo:

```
git init
```

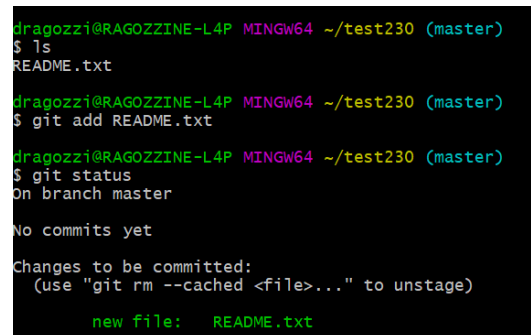
As you can infer from Git's response, it creates a new folder (".git") which is where Git keeps all the internal information it needs (and you can ignore it... the initial dot means it is meant to be a "hidden" directory). There's also a new piece of information in the prompt: a blue (master) which tells you what "branch" you are working in. Branches are an important part of Git, but more complicated than we'll need in Physics 230, so you can ignore this.

Let's modify, stage, and commit a file. Since we have no files here yet, "modify" means to create a new file. Make a new textfile in this directory. You can do this using Notepad or you can even be brave and use a bash text editor ("nano" is the easiest to use, keeping in mind that "^" means "Control"). Name the file README.txt and put in a few words of text (like "This is the README file for the test230 project."). Note that if you use Notepad to create this file, you must save it into the test230 directory, wherever it is (as reported by pwd). To see what files are in a directory, use the Bash command "ls" (short for "list").

With README.txt modified, the command to "stage" the file (preparing it to be "commit"ed) is "git add <filename>" where filename is the name of the file to stage.

```
git add README.txt
```

Git didn't respond with any information... that's normal. (You might get a Warning about "LF" and "CRLF", just ignore it.) To see what state (modified, staged, or committed) each file is, I find myself frequently using git status. Here Git reports that we haven't made any commits yet, but the new README.txt is "staged" to be committed.



```
dragozzi@RAGOZZINE-L4P MINGW64 ~/test230 (master)
$ ls
README.txt
dragozzi@RAGOZZINE-L4P MINGW64 ~/test230 (master)
$ git add README.txt
dragozzi@RAGOZZINE-L4P MINGW64 ~/test230 (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   README.txt
```

Since a "commit" is like saving the file and since Git tracks all the commits, it is really important and helpful to include a short description of what is new in this "commit". That way, you can look through the list of commits and understand what the previous versions are. Including such a "commit message" is required by Git. Let's commit README.txt to our "repo" and write a helpful commit message.

```
git commit -m "Initialize the README file for project test230"
```

Let's talk about the Git output. First, you'll see a 7-8 character code like `da9bc71` associated with your commit. This is called the "abbreviated commit hash" (the full "hash" is 40 characters) and is essentially the "name" of this commit, but you don't need to write it down or memorize it or anything. Git will also report on how many files changed and whether things were added or deleted.

Important information about committing:

- **Commit often!** Whenever you make a self-contained change that (you think) works, go ahead and commit. Remember that committing is different from saving, so you don't have to commit in order to save; wait until you are ready for a new "snapshot" of the project (or want to backup/share the code).
- Keep your commit message to <50 characters (though it is possible to use more)
- Use your message to describe *why* and *what* not how. We won't get into it, but it's not hard to compare different commit snapshots which quickly shows how this commit is different from the last one. Much more valuable is why you made a new commit and what changed.
- The standard style is for commit messages to be written using imperative verbs so that it can finish the following sentence: "If applied, this commit will _____".

Let's do it again. Complete the following

1) Modify your file briefly (and save it).

1.5) Use `git status` to see what's going on

2) Stage your file using `git add`

2.5) Use `git status` to see what's going on

3) Commit the staged changes using `git commit -m "Good commit message"`

That should all go smoothly and should only take ~1 minute.

Now add a new file to your repo:

1) Add a new file (very short)

1.5) Use `git status` to see what's going on

2) Stage your file using `git add`

2.5) Use `git status` to see what's going on

3) Commit the staged changes using `git commit -m "Good commit message"`

Some important things to remember:

- Modifying a file does **not** stage it; you must run `git add` every time AND for each file.
 - Running `git status` helps remind you what files have changed and might need staging.
 - You can modify multiple files but only stage and commit selected files, that's why there needs to be a `git add` command that tells Git which of the recent changes you actually want to commit.

- Things can get confusing if you stage a file and then modify it... for now, better to only stage a file when you are ready to commit it. That is, run `git add` and then `git commit` immediately thereafter.
- I'm not giving you all the Git tools, just the central ones. For example, if you want to see how your modified files are different from the previous commit, you can use `git diff`. This can be really helpful, but I've decided to focus on the core commands.

This modify, stage, commit process is the core of Git workflow, so make sure you are comfortable doing it before moving on.

You can close Git Bash by typing the command `exit` or just closing the Window. Closing this doesn't affect your repo or files... but since we're done with this repo, you can go and delete the "test230" directory now if you'd like.

I hope you are catching a vision of why Git is so powerful and useful. If you've ever had 8 different versions of a file with names like "project0.nb", "project1.nb", "project1.5.nb", "project1.8_works.nb", "project1.9_doesntwork.nb", "project2_kindaworksbuttooslow.nb", etc., and then gone back 3 months later and try to figure out what to do, then you can see how Git would help keep things more organized. © Indeed, I urge you to consider using Git outside of this project and in whatever circumstances it would be helpful.

Working with a remote repo on BYU's GitLab

In the previous example, we only had a local repo. There was no obvious way to backup/share the files using Git. Let's now work on a remote repo.

All students have automatic access to the Department's GitLab via your usual netid and password. Go to pulsar.byu.edu and log in. Go to "Explore Public Projects" and spend a couple of minutes looking around. (If you are familiar with GitHub or Bitbucket or similar online Git tools, GitLab is like our own personal version of those tools.)

Physics and Astronomy GitLab



 A screenshot of a web login interface. At the top, there are three tabs: "Physics and Astronomy" (selected), "Standard", and "Register". Below the tabs, there are two input fields: "Physics and Astronomy Username" containing the text "dragozzj" and "Password" containing a series of dots. There is a checkbox labeled "Remember me" which is unchecked. Below the fields is a green "Sign in" button. At the bottom of the form, there is a "Sign in with" section.

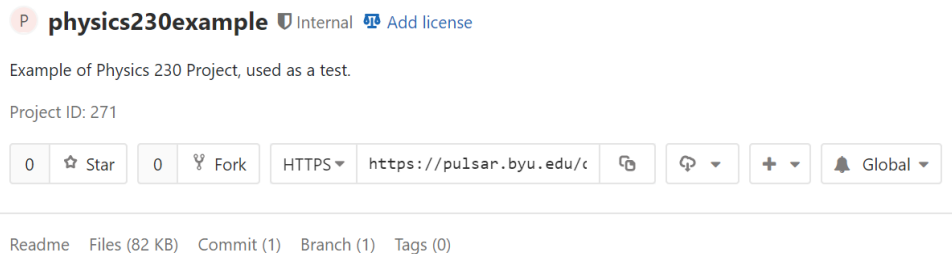
Like Git, GitLab has many different bells and whistles, but we will be focusing on the most basic set of tools necessary for our class project.

Now is the time to meet up with your project partner. If your partner is not here, find someone else whose partner is not here. Ask the TAs or me for help. Choose one group member to be the "host" of project repo. It doesn't matter who... all group members will be able to interact with the repo, but as there will be only one repo for your project, someone has to host it. The host needs to follow the following instructions and the other group members should watch/help. For example, the non-host partner could read these instructions aloud.

The host should go back to the Welcome page by clicking on the GitLab fox icon in the upper left corner. Let's start your Physics 230 Project! Click on "Create a project". You earlier decided on a project name, put this in the "Project name" field and it will automatically go into the "Project slug" field which determines the URL for your project. Put a couple of lines into the Project description, including a 1-2 sentence description of your project (as planned) and a reference that this is your Fall 2018 Physics 230 Project. Change the visibility level to "Internal", which means that all BYU logins can find it and see everything in it (including group members, the TAs, and me). If that's an issue, let me know.

As we read in *GEPSC 3a*, your project should have a README that describes the project. So check the box that says "Initialize repository with a README". Then click "Create Project". Voila! Your project is created and even has its first commit already done! The first commit was to initialize the project and the Project description you wrote is now in the README.md file. (The .md means that the README is in "Markdown" format, which you can just think of as a fancy text file for our purposes.)

There are a lot of things going on here, but thankfully we won't need most of them. It is convenient at this point to go to the bell that says "Global" and change it to "Watch"... that means



The screenshot shows a GitLab project page for 'physics230example'. At the top, it indicates the project is 'Internal' and has an 'Add license' button. Below this, it says 'Example of Physics 230 Project, used as a test.' and 'Project ID: 271'. There are buttons for 'Star' (0), 'Fork' (0), and a dropdown menu for 'HTTPS' with the URL 'https://pulsar.byu.edu/c'. To the right, there is a 'Global' notification bell icon. At the bottom, it shows 'Readme', 'Files (82 KB)', 'Commit (1)', 'Branch (1)', and 'Tags (0)'.

you'll get an email whenever any changes are made, a helpful way to keep track of what's going on. (The other group members will also want to do this at some point.) Scrolling down, hopefully some things will be familiar. Can you find the 7-8 character abbreviated commit hash? Can you find the commit message (the system automatically used "Initial commit")?

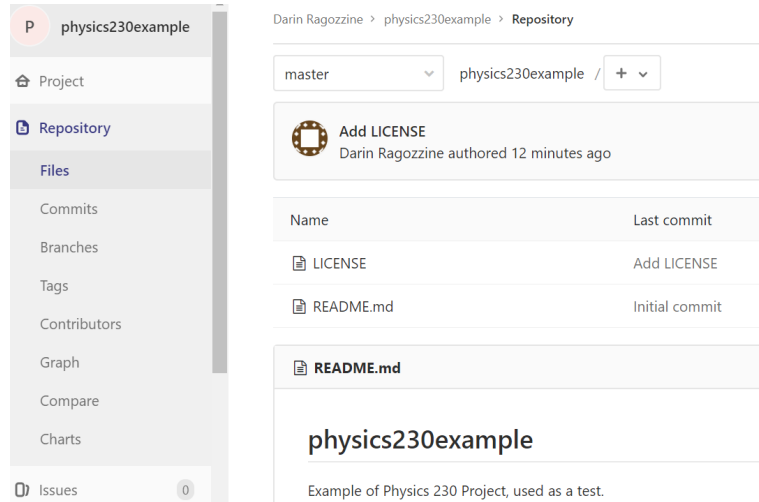
Scroll back up to the project name and you'll see a link for where you can "Add license". As we learned in *GEPSC 3c* having an explicit license is important. Click on this and choose a license template... if you don't know what to do, the MIT License works well and basically says anyone can use whatever you do for any reason.

This is the end of the host portion of the tutorial... everyone should now go back to their own machines.

There are two similar pages that you can find through various links: the GitLab Project page and the GitLab Repository page. For what you need, the Repository page is more important. This shows the files and directories that are in the remote repo, the commit messages, the README file, the abbreviated hash, etc.

Note that the URLs to these pages can be shared with anyone at BYU and it will bring them to the same place. For example, you can see my example repo if you go to

<https://pulsar.byu.edu/dragozzi/physics230example/tree/master>, though you'll have to log in first. It will look kinda like this screenshot. All group members should be able to find your project repo using the link shared by the host similar to the link above.

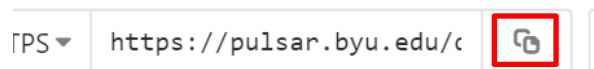


In order for you to work as a group on the project, the host needs to add the other group members. To do so, the host needs to go down to Settings in the menu on the left on GitLab. Under Settings, click on "Members". In the field "Select members to invite", type in their netid. (This will only work if they have logged into GitLab before.) Once their name comes up change the "Role Position" from "Guest" to "Maintainer", which gives them the ability to modify this repo. (Don't fill in the Expiration Date.) Hit "Add to Project" to complete the addition.

In any group project, there is a level of respect that is expected. Furthermore, any changes you make to a project are tracked and attributed to you! So, better not to try any Git shenanigans! 😊

It is possible to modify the repository and commit changes directly on the GitLab website through the browser. That can work okay for some things, but won't be sufficient for our project. So, instead, you are going to create a local repo that is a "clone" (or copy) of the remote repo. On your machine, open up a new instance of Git Bash. You can then clone the remote repo to your local machine using `git clone https://pulsar.byu.edu/dragozzi/physics230example.git`

where you replace the URL with the one for your project. The URL can be copied to the clipboard by hitting the copy



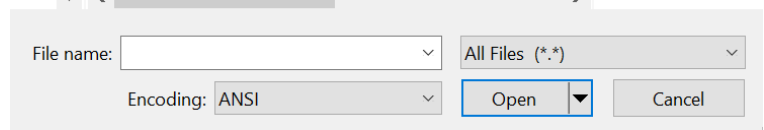
icon (highlighted in red to the right) to the right of the HTTPS link on the Project page. You will have to use your netid to login and tell GitLab who you are. A successful cloning looks like this screenshot. The

```
dragozzi@RAGOZZINE-L4P MINGW64 ~
$ git clone https://pulsar.byu.edu/dragozzi/physics230example.git
Cloning into 'physics230example'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
```

process of cloning creates a new directory: "physics230example" in this case. Go into that directory using `cd` and then see what's there using `ls`:

```
cd physics230example
ls
```


You should see the README.md and LICENSE files. You should be able to open them using Notepad. It's worth noting here that bash inherits from UNIX a minimal interest in file extensions (the information after the . in the filename), which can confuse Windows. When trying to open these files, because they don't end in ".txt", you might need to change the search parameters to "All Files" like shown to the right.



Now you (and all your group members) have a local repo that is connected to a single remote repo. This is perfect for group collaboration, which we'll practice now.

Interacting with a group online repo

Let's start with the simple case of group members working on separate files. Following *GEPSC* let's add three important parts to our repo:

- A) a `TODO.txt` file that gives a list of planned items to work on
- B) a `doc` directory that will hold one of the Project Proposals
- C) a `src` directory that will hold an example *Mathematica* notebook

Choose among yourselves who will do what without overlap (perhaps the host does A and the partner(s) do B and/or C). Use the modify, stage, commit technique from above to do the following. I'll skip the explanatory steps and give examples of commit messages that you should modify to be more relevant to you.

A) TODO

- 1) Make a new `TODO.txt` file and put in a couple of obvious TODOs from your project
- 1.5) `git status`
- 2) `git add TODO.txt`
- 2.5) `git status`
- 3) `git commit -m "Add initial version of TODO.txt"`

B) Project Proposal

- 1) Make a `doc` directory by typing `mkdir doc`
- 2) Find your project proposal (or a group members or a dummy file) and put it into PDF format.
- 3) Place the proposal PDF file into the `doc` directory (using Windows Explorer, for example).
- 3.5) `git status`
- 4) `git add` the file (either using `git add doc/file.PDF` or, if you are already in the `doc` directory (using `cd doc`) then you can just do `git add file.PDF`)
- 4.5) `git status`
- 5) `git commit -m "Add project proposal file doc/file.PDF"`

Note that Git tracks files and keeps them in the correct directories. Also note that you can go up a directory by using "`cd ..`" which changes the directories by going up one folder.

C) Example Mathematica Notebook

- 1) Make a `src` directory by typing `mkdir src`

- 2) Open a new *Mathematica* notebook and enter in one useful line (e.g., a fundamental constant that you plan on using or a filename that you plan on important or an initial comment about the file or whatever).
- 3) Save the notebook to the `src` directory
- 3.5) `git status`
- 4) `git add` the file (see B.4 above about how to do this)
- 4.5 `git status`
- 5) `git commit -m "Add initial src/file.nb containing fundamental constants"`

After committing one of the changes to your local repo, type `git status` again. Before, when there was just a local repo, Git would say "nothing to commit, working tree clean" which meant that it was all up to date. Now the status is different... Git says that you are "ahead" of "origin/master" by 1 commit. For our purposes in this tutorial a branch is effectively the same as a repo and you should replace "origin/master" with "the GitLab online repo". So, translating, this says "your (local) repo is ahead of the GitLab online repo by 1 commit". Here "ahead" means that your local repo has 1 commit beyond what is online, which makes sense because you made some changes.

```
dragozzi@RAGOZZINE-L4P MINGW64 ~/physics230example (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)

nothing to commit, working tree clean
```

As suggested, we can send these changes up to the GitLab server using `git push`. For Git, to "push" means to take changes from a local repo and force them into the remote repo. Have the host go ahead and run `git push`

When the host runs `git status` after a successful push shows, they'll see that the local repo is "up to date" with the GitLab online repo. Now the non-host partner can run `git status` and also see that they are 1 commit ahead (from the changes they made). But then trying `git push` leads to a serious error message:

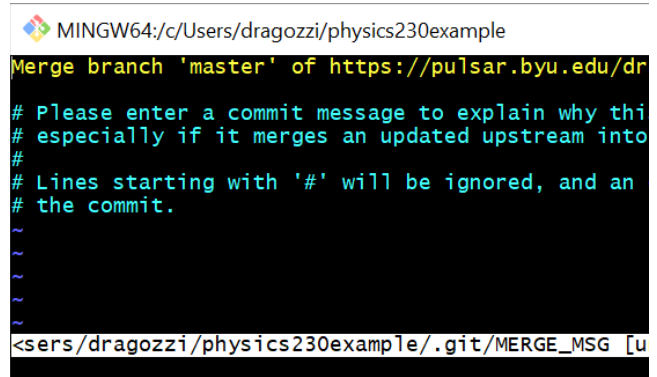
```
dragozzi@RAGOZZINE-L4P MINGW64 ~/physics230example (master)
$ git push
To https://pulsar.byu.edu/dragozzi/physics230example.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://pulsar.byu.edu/dragozzi/physics230example.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Since the host has changed the online repo, the local repo's commits are out of sync with the current version of what's going on. For reasons I won't explain, `git status` doesn't (yet) "know" that it is out of date. In actuality, this local repo is now 1 commit ahead and 1 commit behind the online repo.

The way to resolve this is to first sync up with the online repo. To get the online repo for the first time we “cloned” it, but to get the updated version of the repo as changes are made, you “pull” these changes down from the server using `git pull`

```
git pull
```

Technically, pull does 2 things: 1) finds out what has changed in the online repo and 2) “merges” the changes into the current local repo. A “merge” means that Git tries to figure out whether it can make a single version that has all the changes from two different commits. When this is easy to do (e.g., different files were modified), it usually does a quite good job of figuring out what to do automatically. It will then pop up a text editor (vi) to allow you to enter in the commit message associated with this merge... just type `:q` and hit Enter to get out of vi. Note that the process of merging creates a new commit and vi is there to get your commit message (the yellow is the default message which is saved when you exit out of vi and is fine for now).

A screenshot of a terminal window with a black background and white text. The terminal title bar shows the path 'MINGW64:/c/Users/dragozzi/physics230example'. The main content of the terminal is a yellow prompt: 'Merge branch 'master' of https://pulsar.byu.edu/dr'. Below this, there are several lines of text in a light blue font: '# Please enter a commit message to explain why thi', '# especially if it merges an updated upstream into', '#', '# Lines starting with '#' will be ignored, and an', '# the commit.', followed by several tilde characters '~'. At the bottom, the prompt '<sers/dragozzi/physics230example/.git/MERGE_MSG [u' is visible.

Once you’ve pulled the changes in the online repo and merged them with your changes, now your local repo has all the information and `git status` says that you are now ahead of the GitLab repo. Now that everything is merged properly, you can `git push` and all the changes will go up to GitLab (assuming no one has updated it since your last `git pull`).

Keeping all the repos and commits straight is tricky and sometimes merges are much messier. As a result, when working on a group project, it makes sense to execute `git pull` before you make any modifications... that way you know you are working on the most recent version of the code. It’s worth mentioning here that GitLab also serves as a backup for your code... if your computer dies or you switch Computer Lab computers it’s no problem: `git clone` the online repo and you’re immediately back up to speed.

Merge Conflicts

What if Git can’t figure out how to merge two different sets of changes? When you and your partner are working on different files, the merge is obvious: take the newest version of each file. Even when you are both working on a single file, but in different places in the file, Git can often still figure out the automatic merge. But if you both modify the same line of the same file but in different ways, Git will force you to decide which one is correct. Let’s test this out.

Go to a file (TODO.txt) and pick a line (say, line 2) and both of you edit in your local repo that line. Stage and commit the change. The host should then push this commit up to the online repo. The non-host can then try `git pull` and should get the error message that the Automatic merge failed. `git status` agrees: “you have unmerged paths”. You can’t do anything until you tell Git what to do.

To resolve the merge conflict (in this easy case), simply open the offending file. Here's my TODO.txt in a text editor where you can see that something funny is happening around line 2. Git is showing you the difference between the two conflicting files: what comes after <<<<<<< HEAD and before ===== is what is in your local repo. What comes after ===== and before >>>>>>> [commit hash] is from the online

```

1) research how to convert orbital elements to positions
<<<<<<< HEAD
2) find existing KBO orbital data online
=====
2) find existing TNO data online
>>>>>>> f1859467360bb7900cd15807ace34867b4d06a23
3) just thought of something else
4) yet another idea

```

```

1) research how to convert orbital elements to positions
2) find existing KBO/TNO orbital data online
3) just thought of something else
4) yet another idea

```

repo. Edit the file by removing the <<<, ==, and >>> lines and choosing how to combine these two, an example being shown to

the left. Then save the file, use `git add` to stage the file, and `git commit` (mentioning in your commit message how you resolved a merge conflict). Then you can `git push` the changes and the local and online repos will be synchronized again.

Review

Unless you want to get fancy, this is all the Git you'll need to know for your project. Pretty much everything so far has been generic information that would work on any Git project. Let's review:

- Git Bash is a Command Line interface where you enter text commands
- `git status` shows you the status of all your files and whether your repo is ahead of the GitLab repo
- `git pull` before starting to work to make sure you have the most up-to-date version
- Use a text editor or *Mathematica* to modify files, saving as you go. Once the files have received a significant and completed change, consider making a Git snapshot.
- Use `git add` to "stage" a modified file, telling Git that you'd like to include it in your next commit
- Use `git commit -m "Informative commit message"` to commit your changes, making a new snapshot of your project directory (without removing the old one)
- Use `git push` to send your new updates up to the GitLab repo
- If needed, using `git pull` to merge changes and using a text editor to handle simple merge conflicts.

Things we did not talk about how to do:

- Going back to previous commits (e.g., "undo"ing commits that didn't work out)
- Keeping track of multiple versions of the project via branches (and forking)
- Keeping track of multiple versions of various files using stashing
- More organized group collaboration (e.g., pull requests instead of git push)
- And many more

There are many tutorials online that can help with these (and Google / StackExchange can be very helpful too). The jargon and experience gained in this tutorial prepares you for most of these, including the excellent Pro Git book (<https://git-scm.com/book/en/v2>), the full manual for everything you need to know to become a Git expert.

Using Git for your Physics 230 Project

I'll now mention some important details for how Git will work for your Physics 230 Project in particular. Although I've laid out the nominal path for using Git in the project above, everyone can adapt Git to their particular needs. Git was made for you and not you for Git... if it's really distracting from your personal and/or group workflow, then don't let it get in the way, though you are required to learn it. In particular, 10% of your Project grade is based on learning Git which is demonstrated by 1) completing this tutorial and 2) each person having a minimum of 10 substantive commits to GitLab. (Substantive means more than one line of code in one file, for example.) If you want to keep this simple by, for example, talking with your partner to avoid merge conflict resolution, you are welcome to do so. The beauty of Git is that can be adapted to many different collaboration styles.

Git was designed for plaintext files and though it can understand *Mathematica* notebooks, it's not very good at handling them. In particular, **it will be very challenging to resolve merge conflicts from two people making changes to the same *Mathematica* notebook file**. (This does mean that, in practice, some of the benefits of Git -- the ability to work independently on the same files -- are reduced for this class project.) If you see any merge conflicts from a *Mathematica* notebook, I recommend handling these by opening the two files in *Mathematica* and comparing them by hand. One way to avoid such issues is to plan on using multiple different notebooks. When multiple notebooks are open at once, they can share variables and Modules, so you can work on different aspects of the project, but then combine efforts without too much trouble from Git.

While Git can be used for basically anything, I don't recommend using it to work on the presentation slides (Google Slides works well). Although, with some coordination, Git would be better than, say, emailing the file back and forth.

I have found that actually using Git (and really trying to learn what's going on) is the best way to learn it. This tutorial has started you on this path and initialized your Physics 230 project. Note that learning Git is a goal in and of itself, independent of the Physics 230 Project. It might be the most useful thing you do in this class!

[Answer to question above: Okay, I just shared the fourth commit on GitLab. I added constants.nb to the (remote) repo. The main changes are such-and-such. I have also made changes to orbitsolver.nb, but I didn't stage these, so these changes weren't committed so you can't see them yet.]