Assembly Demo

In this demo, we'll take a toy dataset from the case study and process it through our *de novo* transcriptome assembly pipeline.

Overview:
Steps 1-3 guide you through pulling all the files and software you need from our GitHub (also covered in Day 1 - Data management and movement).
Steps 4 configures the set up and installs the software for the workflow.
Steps 5-8 focus on setting up the input data and normalizing it in preparation for assembly.
Steps 9-15 are the meat and potatoes of our pipeline, during which you run four assemblers (Velvet, SOAP, Trinity, and TransAbyss).
Step 16 covers how to run our automated QC on all subassemblies and your final assembly.
Step 17 gets you set up for the Annotation Demo.

Step 1) First, we need to get the code. This code is housed on github, which makes grabbing code pretty easy.

In a web browser, go to github.com/ncgas. Here, you will see some options - toward the top, you should see "de-novo-transcriptome-assembly-pipeline". Click that, which will bring you to the repository for that code.

Just like we did with the transfer activity, you should see a big green button that says "Code" with a down arrow. Click this, and copy the location by clicking the clipboard next to the link.

```
#On you vm:
cd ~

#git clone the repository:
git clone https://github.com/NCGAS/de-novo-transcriptome-assembly-pipeline.git

#see what you have!
ls
```

Step 2) Once this is complete, you should see a directory called "de-novo-transcriptome-assembly-pipeline", just like the repository. Head on into the repo, and see what's inside:

```
cd de-novo-transcriptome-assembly-pipeline
ls
```

Step 3) Hm... All we see is a Project_Carbonate_v4 folder and some notes files. This is an older version that isn't portable. We need version 5, which is our new, fancy, container-based version...

Git is not just a place to park code, it's also a convenient way to have version control. What you are looking at now is the production version of the code, but we're using something still in development.

If we look back at the github website, we can see a drop down menu next to the Code, Issues, etc menu - it will default to say “master”. If you click that, you can see there is a v5_dev branch. That looks promising!

```
git checkout v5_dev
```

Now you should see a Project_v5 folder! Now we have a Project_v5 folder - this isn't specifying a specific machine, which means it's the portable version! If we want to see all the available branches that we’ve viewed on the command line, we can see them by using:

```
git branch --list
```

Let's head into that new Project_v5 folder.

```
cd Project_v5
ls
```

NOTE: you will remain on this branch until you change it, even if you sign out and back in!

Step 4) Okay, now things are starting to look familiar! We see the file set up we talked about in the introduction to the pipeline - there's a folder for our input_files, there's folders for each assembler, and a final_assemblies folder. We also see a Setup.ba script, which should be green, meaning it is an executable file. Let's see if it has a handy help option:

```
./Setup.ba -h
```

Whoo! It does! It prints a quick run down of all the options to set up the code. Let's do the following (make sure you understand what this command means, given the help!):

```
./Setup.ba -e your@email.com -s single -r 75 -d -b "/home"
```

-e: your email address, which isn't critical here, but is a required part of the command
-s: our data is single-end
-r: our read length is 75bp
-d: download that software!
-b: bind our /home directory to the containers so it can read and write to where we are working!

While that is running, let's talk about software.

We built this workflow to be as portable as possible. This means you can run it on the cloud, like we are doing here, or you can run it on a cluster, as long as singularity is installed. You

won't be able to install singularity on your own (requires admins), so you will have to load that first. It's usually as easy as:

```
module load singularity
```

Then, the workflow will look for the required software and install the missing pieces, saving you space in your directories. If you want to do that, you have to load the software first, to make sure the workflow can find it. For example, if we were running this on Carbonate, we could do the following first

```
module load singularity
module load trinityrnaseq/2.8.4
module load soapdenovotrans/1.03

./Setup.ba -e your@email.com -s single -r 75 -d -b "/N/projects,/N/slate,/N/home" -q general
```

You can see that -b changes depending on where you are working on the system. Our spaces to work on clusters are often a bit different than the ones in a Ubuntu VM, so make sure to pay attention to that.

Additionally, there is a new option we are using here: -q. The ./Setup.ba -h output tells us that this is a means to define which queue we want to use, which will also change by system. On Carbonate, general is the name of our - you guessed it - general purpose queue. If you had a queue called long_mem, debug, etc. you can set all of your scripts to run in that queue, as long as you define that with -q!

If you were to run this command on Carbonate, you'd see messages that tell you that Trinity was already found, so it skipped that installation and the same with Soap.

The final command that might be of interest is the -c flag. Let's see what that does:

```
./Setup.ba -c
```

Niiiice, all the citations for the software in one place ^_^.

Once your software has finished installing (takes ~20 min), you can scroll up and see some of the output. We've tried to make this at least reasonably readable - the software looks for the software by name (and tells you so!), then either skips it because it was found, or pulls the container if it can't find it. Then it tells you whether that was successful or not. If you have problems with installation - please feel free to email us with this output!

Step 5) Okay, hopefully your installation has completed. Let's look at what we have now:

```
ls
```

Hm, not much changed. That's because the software all installed in the software directory. You can feel free to look around in there, but for now, we'll just get moving with our assembly. Thinking back to our introduction to the workflow, input seems like a logical place to start!

```
cd input_files
ls
```

Step 6) Look at that - the code ships with demo data ^_^. These are short versions of the reads that Lydia produced in the case study. We shrunk down the size of the dataset to make sure it runs within a reasonable amount of time while still giving you a chance to see what you should expect from running the workflow.

There's a README in almost every folder, so let's start there!

```
less README
```

This tells us how to push all of our reads into one file, called reads.fq (for single-end data) or left.fq and right.fq (for paired-end data). If we wanted to use our own data, we could just replace this demo data with our own data, and combine it in the same way. Please do not attempt that here, as our VMs for the workshop are far too small to run real data!

Let's combine our data into the reads.fq file:

```
cat *.fastq > reads.fq
ls
```

Step 7) Once you have that file, you can run the normalization step. RNA-seq is by design repetitive. This is useful in quantification, but just adds computational burden to the assembly. It is standard-practice to reduce the data depth to 30x coverage. Trinity conveniently ships with this code, so we will use theirs.

You should see a RunTrinity.normalize.sh file, again in green. Let's look at this file, which is good practice before you run any script - trust no one's code ^_^:

```
less RunTrinity.normalize.sh
```

At the top, we see a SLURM header, meaning we could submit this job to a cluster. The email and queue can be set up, you can change the resource requests as you desire as well. Below the #SBATCH block, you can see some code. Most of our scripts will look like this, so here's a quick overview of what some of these lines are doing:

`source ../setup_files/path_set`: This is reading in a file that stores some variables, such as $PWDHERE, which defines our current directory location for ease of use.

`export PATH=$PATH:$PWDHERE/software/Trinity`: This is adding our software folder for Trinity to the PATH variable, which is a kind of treasure map that the system uses to find software. We add the current PATH first, so that if you pre-load existing software, it will

default to using that first, but then as a last resort look in the software folder that shipped with the code.

cd $PWDHERE/input_files: This moves us into the correct folder, the input_files folder, before running the code.

export reads=$PWDHERE/input_files/reads.fq: This simply stores the read file in a variable. This seems a bit needless, but having this here allows for quick changes to target other files if you want to customize or troubleshoot the code later. Basically, it allows you to change which reads you are normalizing without having to dig through the actual commands.

insilico_read_normalization.pl --seqType fq -JM 100G --max_cov 30 --single $reads --PARALLEL_STATS --CPU 16: This is the meat of the script - this is the normalization command that comes with Trinity, reducing the coverage to 30x. We set this to run on 16 CPUs for cluster use, but the VMs are smart enough to throttle this back when we only have, say, 6 CPUs.

ln -s $PWDHERE/input_files/single.norm.fq $PWDHERE/input_files/reads-norm.fq: This command makes a symlink (think shortcut) to the single.norm.fq filename that is output by trinity. Sometimes you will see this when we have to reformat filenames to make them work with scripts - some scripts don't like ".", etc.

Step 8) Now that we understand what this script is doing, let's run it! Since the script is an executable, we can run it just like we did with ./Setup.ba:

./RunTrinity.normalize.sh

This is useful on a VM, as it ignores all the SLURM header stuff at the top because without a job handler, it just looks like comments. If you were to run this on a cluster however, you'd want to submit it to the queue:

sbatch RunTrinity.normalization.sh

Step 9) That should run pretty quick - this is a small dataset. Once it is done, you should see a blue reads-norm.fq file. Feel free to look at it with less to confirm it looks like a real fasta and isn't just blank:

less reads-norm.fq

Now we can use this same file with all of our different assemblers. They all operate the same, so we'll start with the only one that is a little complicated - Velvet.

cd ../Velvet
ls

Here you will see... More scripts and a README. I recommend reading the READMEs, especially the first time you run any scripts. This is pretty short, with the gist being that you

have to run the steps in order, but you can run both 1s at the same time. Finally, we'll need to run the Combine.sh script at the end. We also have a link to the manual and license for software, if you were to want to customize the Velvet scripts - say if you have stranded data, or some other variation on what we have set as defaults.

Step 10) As you should always do, let's look at the scripts - first RunVelvet1.sh

```
less RunVelvet1.sh
```

This will look similar to much of what we saw in the normalization script - there's a SLURM header if we need it, a line to pull in the variables shared throughout, a line to set up PATH, a line to set up the reads variable, and then the main code.

The main code here is the first step of Velvet, running three kmers at the same time. If you wanted to, you can edit the velveth command - just pull up the manual and see what you need to do to make it fit your needs. A common thing you may want to change is the kmer for instance. It's easy enough to figure that out ^_^.

Again, the point of providing you with these scripts, instead of automating this process entirely with a workflow management system like snakemake, is to make it easy to change the commands, to make sure you can follow what it is doing, and also make it flexible - you can change any of these files. We are just streamlining the organization, the job scripts, and the overall plan so that you don't have to start from scratch.

There is one thing you may notice, which you will see over and over again - each line for velveth ends in an &, and then there is a line that just says "wait"...

This is a way to run three lines in parallel - each of the three lines that end in "&" will run at the same time (in background). "wait" just tells the system to wait until all processes are done, then move to the next line. If you forget this line, your job will forget about the things in background (out of sight, out of mind!!), and think it's done!

This script is also designed to auto-submit the next step if you are running it with a job scheduler. This saves you time with larger files, but it will complain a bit when we run it on a VM. We can ignore its complaining though ^_^.

Now that we understand what this is doing, let's run it!

```
./RunVelvet1.sh
```

RunVelvet1b.sh is the same thing, just with different kmers, so let's run that as well. If you were on a cluster, you wouldn't have to wait for the first job to finish before submitting this one as well.

```
./RunVelvet1b.sh
ls
```

Step 11) You should see some folders appear as output. Continue this process of looking at the scripts and running each step:

```
./RunVelvet2.sh
./RunVelvet2b.sh
./RunVelvet3.sh
./RunVelvet3b.sh
```

Step 12) Now we have completed the three steps of velvet/oases and are ready to clean this up and send it to the final_assemblies folder. The Combine.sh script tags each output with the assembler name and kmer, and then combines them all into a file called "Velvet.fa", which it puts in the final_assemblies folder.

```
less Combine.sh
./Combine.sh
```

NOTE: you may see some of these kmers fail and other assembler kmers fail. This is largely because our data is so small. You may also see a failure on memory (SOAP is very memory hungry). In a normal situation, you'd want to boot a larger VM or request more memory from the cluster (remember, recommended starting resource needs are in the job header!). However, in this case, you can skip the SOAP assembly if it's being problematic.

Step 13) We're done with Velvet, so let's quickly check that the Combine script did it's job:

```
cd ../final_assemblies
less Velvet.fa
```

Yup, that looks like a fasta file! Everything is labeled too. Looks good!

Step 14) Now you can do the same thing with each of the other three assemblers - Trinity, Soap, and TransAbyss. If you wanted to, you could also run a different assembler (like RNAspades) simply by replicating the steps in another folder. Again, this is a flexible framework!

Complete the other assemblers - see READMEs as you go!

Step 15) Once you have all the assemblers done, you should see a .fa file for each assembler in the final_assemblies folder. Let's go there and look:

```
cd final_assemblies
less README
```

You'll see that the order here is a bit reversed to the assembler folders - we'll want to run that Combine.sh script first, to combine all of our assembler outputs into one file, before running Evigene. The README also details the output you should expect.

NOTE: If you wanted to, you can add more assemblies to this folder, with the *.fa extension, and they too will be included with the Evigene run. This has been done to include Iso-seq data, previous assemblies, etc!

Let's combine and run Evigene:
./Combine.sh
less RunEvigene.sh
./RunEvigene.sh

It's useful to know what Evigene is doing to combine these assemblers - it's the crux of our CDTA assembly. We wrote an entire blog on the methods, and how to interpret some of the output: https://blogs.iu.edu/ncgas/2018/12/17/how-evigene-works/

Evigene will output several files (8, to be exact) and four directories: dropset, inputset, okayset, and tmpfiles. The main files you want to look for are in the okayset/ folder. In there there are *.okay* and *.okalt.* files. The okay files are the longest ("best") single version of a transcript. Transcripts that are similar but not identical are stored in the okalt (alternative forms). You can find the link between the two - see the documentation for evigenes at http://arthropods.eugenes.org/genes2/about/EvidentialGene_trassembly_pipe.html

Step 16) So... we have output... now what? Well, you shouldn't trust anything at face value - so we should get some quality metrics. Conveniently, we include a QC folder for just this reason!

cd QC
less README

The README tells you about what the QC is doing, which is essentially running quast and busco on all of the subassemblies and your final combination. If your combined assembly isn't the best for your purposes, you can pick from any of the subassemblies that you did!

less RunQC.sh

This script is a bit complex compared to what we've been doing, simply because it deals with so many assemblies, multiple tables, and joining files. However, we have commented what each section is doing, in hopes that you can follow it a bit better. If you ever have questions, feel free to email us at help@ncgas.org to get explanations or help with this workflow!

Originally when I designed this demo, this took minutes to run. Now it takes forever! I recommend running it as the last thing for the day, and let it run overnight. Oops! If you want to run it:

#THIS WILL TAKE 1-2 HOURS TO RUN! DON'T DO IT NOW!
./RunQC.sh
less -S QC.table
#-S will turn off line wrapping, which makes table viewing easier in bash

Not surprisingly, this will tell you that the assemblies are kind of terrible as far as quality. We ran very little data, so you cannot expect much. However this does give you an idea of what to expect when you do run this with a full dataset (which often takes ~2-3 weeks to run through a queue).

Step 17) We'll cover annotation of the assembly next - but you will need to do one thing first in preparation - pull the pre-run folder from Canvas. Annotation requires a lot of large files and it doesn't work well with these smaller, workshop sized VMs. More on that later, but what you will need to do is:

a) download the annotation.tar.gz file from Canvas (in Files and in the module for this demo)
b) transfer it to your VM
c) delete the /final_assemblies/annotation folder that came from the GitHub
d) move annotation.tar.gz to /final_assemblies
e) tar xfv annotation.tar.gz to unpack the pre-run version