rPlant

Kurt Michels, University of Arizona, kamichels@math.arizona.edu
Barb Banbury, University of Tennessee, darwinthesun@gmail.com

July 8, 2015

Contents

1 Introduction

The iPlant Collaborative has developed many resources to deal with the emerging computational challenges facing biology. The project was initially designed to support the plant sciences, but thanks to a generic approach, can be equally used by other disciplines. Users have access to many different applications for data analysis, including clustering/network analyses, QTL mapping, sequence alignments, phylogenetic tree building, and comparative methods.

The main interface is its user-friendly Discovery Environment (`http://www.iplantcollaborative.org/discover/discovery-environment`). A command-line interface, the agave API (`http://agaveapi.co/`, (Dooley et al. 2012)), is linked to the Discovery environment. The Agave API is used for computationally intensive applications. The API is a RESTful application programming interface (Fielding 2000) that allows direct interaction with all of iPlant resources. One of the ways to access the API is to use *cURL* statements (Stenberg 1996), an example of a *cURL* statement will be detailed in one of the sections. The API provides access to authentication, data manipulation and storage, and job submittal via HTTPS- and command-line functions. The benefit of using the API is having programmatic access that allows advantages to power users (e.g. submitting jobs via batch files). The *rPlant* package provides a direct link between high performance resources located at the Texas Advanced Computing Center (`http://www.tacc.utexas.edu`) that the API can access and the R environment, by essentially creating wrappers around the *cURL* statements, using the *RCurl* (Lang 2007) package.

2 Getting Started

This vignette assumes you have the current version of R (R Core Team 2015). First, install and load the package. A stable release is available through CRAN (`http://cran.r-project.org/web/packages/rPlant/`) or a working repository can also be used through R-Forge (`https://r-forge.r-project.org/projects/rplant/`).

You can register as an iPlant user on their website (`http://user.iplantcollaborative.org/`) generating a unique username and password combination.

2.1 Gaining Access to the API

```
Validate(username, password, api="agave")
```

The username/password combination will be used in the `Validate` function. The `Validate` function is required for every rPlant session and needs to be the first thing executed or the session will fail. In addition, it has a four-hour expiration. *rPlant* functions will auto renew a session, thereby extending the expiration. If a session sits idle and expires, the user will not have to re-validate a session as the functions will do this automatically. The only time a user will need to use the `Validate` function is at the start of a new session.

```
> require(rPlant)
> username <- "enter your username"
> password <- "enter your secret password"
> Validate(username, password, api="agave", print.curl=TRUE)
[1] "curl -sku 'Rxa8JVUXz84jWj0L2t27xHwWwGsa:GDXZZ0OGa_v3OAflYoAJ6fvvjboa' -X POST -d
'grant_type=client_credentials&scope=PRODUCTION&username=enter your username&
password=enter your secret password' https://agave.iplantc.org/token"
[2] "Error: Bad Request"
```

The function checks if the username and password are valid iPlant credentials. If they aren't the above error is displayed. If the function is successful then nothing is printed. On *rPlant's* backend, a new R environment (rPlant.env) was created that stores all of the validation objects, including the user key and secret, user name and password, and token expiration. These items can be examined by using the `ls(rPlant.env)` and using the $ operator to display individual objects.

Every *rPlant* function has the option `print.curl=TRUE` or `FALSE`. This refers to `cURL` a computer software project providing a way to transfer data using various protocols, for detail on `cURL` see `http://curl.haxx.se/docs/`. These statements (w/o the outside quotes) can be copied and pasted into a terminal in linux or unix. An example of a curl statement for the Agave API is shown above '[1]'. And if `cURL` is installed on the computer then the statements can be executed. You will see that these statements do the exact same thing as the *rPlant* functions. This is one of the big advantages of *rPlant*, it can be used on any computer (including windows) and there is no need for the user to install `cURL` on that computer, because *rPlant* uses the package `RCurl`.

Note: This package abides by the unix rule, "silence is golden". If a function is successful then no output will be displayed. If an error is attained then the error will be printed.

3 Uploading Files

3.1 UploadFile function

`UploadFile(local.file.name, local.file.path="", filetype=NULL, print.curl=FALSE, suppress.Warnings=FALSE)`

The first step is to upload files onto iPlant servers, which will be referred to as the iPlant cloud. To clarify the `UploadFile` takes a file from your computer and uploads it onto the cloud, NOT a file from the R workspace. The local.file.path is the path to the file on your computer. If one has objects in the workspace that need to be uploaded, then save them to the computer first before uploading.

```
> data(DNA.fasta)
> write.fasta(sequences = DNA.fasta, names = names(DNA.fasta), file.out = "DNA.fasta")
> UploadFile(local.file.name="DNA.fasta")
```

```
[1] "Error: object 'DNA.fasta' already exists in '' directory"
```

In the event that the file already exists on the iPlant server, an error will report and the file will not be uploaded. For details on how to check contents of iPlant directories, see Section 4, and for manipulating files (like deleting, moving, sharing) see Section 5.

The file format for the uploaded file can be defined using `filetype`, some programs will only accept certain types of files. This can also be left as NULL, if the iPlant application doesn't require it. For the fasta file the file type is FASTA-0. See section 3 for the various file types that are supported.

3.2 Supported File Types

`SupportFile(print.curl=FALSE)`

There are 53 file types supported by iPlant, use the `SupportFile` function to see all of the available file types (i.e. `PHYLIP` file type is "PHYLIP-0" and `ClustalW` is "ClustalW-1.8").

```
> SupportFile()
```

```
 [1] "2bit-0"           "ASN1-0"           "BAM-0.1.2"        "Barcode-0"
 [5] "BED-0"            "BlastN-2.0"       "Bowtie-0"         "BZIP2-0"
 [9] "CEL-3"            "ClustalW-1.8"     "CSV-0"            "DOT-0"
[13] "EMBL-0"           "EXPR-0"           "FAI-0"            "FASTA-0"
[17] "FASTQ-Illumina-0" "FASTQ-Int-0"      "FASTQ-Solexa-0"   "FASTQ-0"
[21] "Genbank-0"        "GFF-2.0"          "GFF-3.0"          "GFF-3.0"
[25] "GraphML-0"        "GTF-2.2"          "HTML-4"           "HTML-5"
[29] "Newick-0"         "NEXUS-0"          "PAIR-0"           "PDB-3.2"
[33] "Phylip-0"         "PhyloXML-1.10"    "Pileup-0"         "SAI-0.1.2"
[37] "SAM-0.1.2"        "SBML-1.2"         "SBML-2.4.1"       "SBML-3.1"
[41] "Soap-PE-1"        "Soap-SE-1"        "Stockholm-1.0"    "TAB-0"
[45] "TAR-0"            "Text-0"           "VCF-3.3"          "VCF-4.0"
[49] "WIG-0"            "TGZ-0"            "HEAD-0"           "TAIL-0"
[53] "CAT-0"
```

4 Manipulating directories on iPlant cloud

Once a file (in our case "DNA.fasta") has been uploaded onto the iPlant cloud the user would want to make sure the file has been uploaded correctly. By using the `ListDir` function, the person can specify which directory to look at and the contents (both files and directories) of that directory will be shown. The default directory shown is the home directory. There are a few other directory manipulation functions, they are: `CopyDir`, `MakeDir`, `ShareDir`, `PermissionsDir`,`RenameDir`, `MoveDir` and `DeleteDir`.

4.1 Listing directories

```
ListDir(dir.name="", dir.path="", print.curl=FALSE, shared.username=NULL,
suppress.Warnings=FALSE, show.hidden=FALSE)
```

The `dir.path` is the path of the directory that `dir.name` is in. The subsection sharing directies explains more, but a user can view a directory that has been shared with them. Sometimes there are hidden files in directories, `show.hidden` will reveal those. Looking in the home directory you can see the "DNA.fasta" file.

```
> ListDir(dir.name="", suppress.Warnings=TRUE)
      name        type
 [1,] "DNA.fasta" "file"
```

On an aside, all functions contain an option, `suppress.Warnings`. If you are absolutely sure that the commands you are entering are correct then to speed up the process have `suppress.Warnings=TRUE`. But a word of caution, the warnings are useful. If warnings are suppressed then problems may arise.

4.2 Creating directories

```
MakeDir(dir.name, dir.path="", print.curl=FALSE, suppress.Warnings=FALSE)
```

The following function will make a directory named "hello" in the home directory.

```
> MakeDir(dir.name="hello")
```

Now the user makes the directory "all" in the "hello" directory.

```
> MakeDir(dir.name="all", dir.path="hello")
```

Here the user makes another directory "robots" in the "all" directory. See how the `dir.path` needs to be constructed (the path to the directory where `dir.name` is located). All of the functions have this same format.

```
> MakeDir(dir.name="robots", dir.path="hello/all")
```

The user can view what is inside the "hello/all/robots" directory and see that there is nothing in there.

```
> ListDir(dir.name="robots", dir.path="hello/all")

     name type
```

4.3 Sharing Directories

`ShareDir(dir.name, dir.path="", shared.username, read=TRUE, execute=TRUE, write=TRUE, print.curl=FALSE, suppress.Warnings=FALSE)`

A really nice feature of *iPlant* is the ability to file share. As was said in the introduction one of iPlant's original goals was to work with very large data sets. And when data sets are too large to share with conventional means, then a sharing feature is absolutely necessary. There are in fact two share functions, one for sharing a single file (`ShareFile`) and the other (`ShareDir`) for sharing an entire directory. When sharing a directory, all files contained within will be shared.

In this example we share the "all" directory with user "henryl".

```
> ShareDir(dir.name="all", dir.path="hello", shared.username="henryl")
```

Also, user "phyllisl" shared a folder with me. I can view those files and directories that have been shared with me.

```
> ListDir(dir.name="data", dir.path="", shared.username="phyllisl")

     name            type
[1,] "muscle3.fasta" "file"
```

Several other functions use the `shared.username` option. For example, the `SubmitJob` function and the wrapper functions. A job can be run on files that are shared (see Section 7).

4.4 Listing Permissions on Directories

`PermissionDir(dir.name, dir.path="", print.curl=FALSE, suppress.Warnings=FALSE)`

Once sharing is done a user can use `PermissionsDir` on any directory to find who the folder is shared with.

```
> PermissionsDir("all", "hello")
      Name  Username         Permissions
 [1,] "all" "admin2"         "R/W"
 [2,] ""    "admin_proxy"    "R/W"
 [3,] ""    "de-irods"       "R/W"
 [4,] ""    "dooley"         "All"
 [5,] ""    "henryl"         "R/W"
 [6,] ""    "ibp-proxy"      "R/W"
 [7,] ""    "ipc_admin"      "R/W"
 [8,] ""    "ipcservices"    "R/W"
 [9,] ""    "kamichels"      "R/W"
[10,] ""    "proxy-de-tools" "R/W"
[11,] ""    "rodsBoot"       "R/W"
```

The folder is confirmed to have been shared with user "henryl".

4.5 Renaming Directories

`RenameDir(dir.name, new.dir.name, dir.path="", print.curl=FALSE, suppress.Warnings=FALSE)`

This function renames a directory.

```
> RenameDir("robots", "tools", "hello/all")
```

And see that it has been changed.

```
> ListDir("all", "hello")

     name    type
[1,] "tools" "dir"
```

When the directory is renamed, permissions on sharing will be removed and need to be redone.

4.6 Copying Directories

```
CopyDir(dir.name, dir.path="", end.path="", print.curl=FALSE, suppress.Warnings=FALSE)
```

This function copies a directory. The following function will copy the directory "tools" from "hello/all" to the home directory.

```
> CopyDir("tools", "hello/all", end.path="")
> ListDir("all", "hello")
      name     type
  [1,] "tools" "dir"
> ListDir("")
      name          type
  [1,] "hello"      "dir"
  [2,] "tools"      "dir"
  [3,] "DNA.fasta"  "file"
```

4.7 Moving Directories

```
MoveDir(dir.name, dir.path="", end.path="", print.curl=FALSE, suppress.Warnings=FALSE)
```

This function moves a directory. The following code will move the directory "tools" from "hello/all" directory to the "hello" directory.

```
> MoveDir("tools", "hello/all", end.path="hello")
> ListDir("all", "hello")
      name  type
> ListDir("hello")
      name       type
  [1,] "tools"   "dir"
```

When the directory is moved, permissions on sharing will removed and have to be redone.

4.8 Deleting Directories

```
DeleteDir(dir.name, dir.path="", print.curl=FALSE, suppress.Warnings=FALSE)
```

This function deletes a directory and all of the subdirectories and any files contained within it.

```
> DeleteDir("tools")
> ListDir("")
      name         type
  [1,] "hello"     "dir"
  [2,] "DNA.fasta" "file"
```

5 Manipulating files on iPlant cloud

The file manipulation tools available in this package are very similar to directory manipulation tools. The file manipulation functions are: `ShareFile`, `PermissionsFile`, `RenameFile`, `CopyFile`, `MoveFile` and `DeleteFile`.

5.1 Sharing Files

```
ShareFile(file.name, file.path="", shared.username, read=TRUE, execute=TRUE,
write=TRUE, print.curl=FALSE, suppress.Warnings=FALSE)
```

As described in the `ShareDir` function, a really nice feature of iPlant is the file sharing feature. This function shares one file at a time.

The following code will share the file "DNA.fasta" with user "phyllisl".

```
> ShareFile(file.name="DNA.fasta", shared.username="phyllisl")
```

5.2 Listing permissions on a file

```
PermissionFile(file.name, file.path="", print.curl=FALSE, suppress.Warnings=FALSE)
```

Once sharing is done a user can use `PermissionsFile` on any file to find who the file is shared with.

```
> PermissionsFile("DNA.fasta")
     Name         Username      Permissions
[1,] "DNA.fasta" "ipcservices" "R/W"
[2,] ""          "kamichels"   "All"
[3,] ""          "phyllisl"    "R/W"
[4,] ""          "rodsadmin"   "R/W"
```

The file is clearly shared with "phyllisl"

5.3 Copying Files

```
CopyFile(file.name, file.path="", end.path="", print.curl=FALSE, suppress.Warnings=FALSE)
```

This function copies the file from one directory to another.
The function copied the file "DNA.fasta" from the home directory into the "hello/all" directory. Below you can see this change:

```
> CopyFile("DNA.fasta", end.path="hello/all")
> ListDir("all", "hello")
     name         type
[1,] "DNA.fasta" "file"
> ListDir("")
     name         type
[1,] "hello"     "dir"
[2,] "DNA.fasta" "file"
```

5.4 Moving Files

```
MoveFile(file.name, file.path="", end.path="", print.curl=FALSE, suppress.Warnings=FALSE)
```

This function moves the file from one directory to another. The move took the file "DNA.fasta" from the home directory into the "hello/all" directory.

```
> MoveFile("DNA.fasta", end.path="hello")
> ListDir("hello")
     name        type
[1,] "tools"     "dir"
[2,] "DNA.fasta" "file"
> ListDir("")
     name        type
[1,] "hello"     "dir"
```

When the file is moved, permissions on sharing will be removed and have to be redone.

5.5 Renaming Files

```
RenameFile(file.name, new.file.name, file.path="", print.curl=FALSE, suppress.Warnings=FALS
```

This function renames a single file.

```
> RenameFile("DNA.fasta", "lp.fasta", "hello/all")
> ListDir("all", "hello")

     name       type
[1,] "lp.fasta" "file"
```

When the file is renamed permissions on sharing will be removed and have to be redone.

5.6 Deleting Files

```
DeleteFile(file.name, file.path="", print.curl=FALSE, suppress.Warnings=FALSE)
```

This function deletes a single file in the specified directory.

```
> DeleteFile("lp.fasta", "hello/all")
> ListDir("all", "hello")

     name type
```

The file "lp.fasta" is no longer in the "hello/all" directory.

6 Applications

The real power in the *rPlant* package is to have the programmatic access to the phylogenetic tools/applications that are available in the API. *rPlant* can be used to interact with any of the API applications.

6.1 Listing Applications

`ListApps(description=FALSE, print.curl=FALSE)`

This function returns a sorted list of the public applications that are available via the Agave API. These applications are ones that can be used with the `SubmitJob` function (see Section 7).

```
> ListApps(description=FALSE)

  [1] "AdjustPvalue-0.0.1u1"
  [2] "AllpathsLG-LONESTAR-44837u1"
  [3] "AllpathsLG_small-genomes-48777u1"
  [4] "AllpathsLG_stampede-48777u1"
  [5] "bayesembler-1.1.1u2"
  [6] "BEAST-2.1.3u1"
  [7] "Bismark-0.13.1u1"
  [8] "Bismark_genome_preparation-0.13.1u1"
  [9] "Bismark_methylation_extractor-0.13.1u1"
 [10] "bowtie2-2.2.4_aligner-2.2.4u1"
   ⋮
```

As said above, the applications shown here are PUBLIC applications to everyone and the users personal ones. Applications in the API are split into two categories, public and private. Private applications are ones that are developed and tested and changed. Only the user who created them or those they share it with can use the private applications. The other category is public applications; after a private application has gone through extensive testing, then the application can be published and it becomes a public application which is available to all iPlant users.

6.2 Application Information

`GetAppInfo(application, return.json=FALSE, print.curl=FALSE)`

The `GetAppInfo` function returns the application with a short description and the input/output filetypes.

```
> GetAppInfo("velveth-stampede-1.2.08u1")

$Description
[1] "Genome assembler for short sequencing reads, first stage."

$Application
[1] "velveth-stampede-1.2.08u1" "Public App"
[3] "Newest Version"

$Information
      kind         id               fileType/value   details
 [1,] "input"      "reads1"         "fasta-0"        "Short Paired Seqs 1"
 [2,] "input"      "reads2"         "fasta-0"        "Short Paired Seqs 2"
 [3,] "input"      "reads3"         "fasta-0"        "Long Paired Seqs"
 [4,] "input"      "reads4"         "fasta-0"        "Short Unpaired Seqs"
 [5,] "input"      "reads5"         "fasta-0"        "Long Unpaired Seqs"
 [6,] "input"      "reads6"         "fasta-0"        "Reference Seqs"
 [7,] "parameters" "kmer"           "string"         "kmer setting"
```

```
 [8,] "parameters" "strandSpecific" "string"         "strand specific"
 [9,] "parameters" "format1"        "string"         "file format, Short Paired Seqs 1"
[10,] "parameters" "format2"        "string"         "file format, Short Paired Seqs 2"
[11,] "parameters" "format3"        "string"         "file format, Long Paired Seqs"
[12,] "parameters" "format4"        "string"         "file format, Short Unpaired Seqs"
[13,] "parameters" "format5"        "string"         "file format, Long Unpaired Seqs"
[14,] "output"     "VelvethOutput"  "TEXT-0"         "Output File"
```

The `GetAppInfo` function returns a list of information about the application that is needed for the `SubmitJob` function. The first element gives a short description of the application. The second element in the list gives information on the application including its use permissions (public vs. private) and whether it is the newest version. The third element in the list is a matrix with four columns of information: kind, id, file type or value, and any details. In the example above, first column ('kind') states there are six inputs for this app, the 'id' column names those inputs as 'reads5', 'reads3', etc. There are also seven parameters for the app, such as 'format2', 'kmer', etc. And one output, which would be the output name. The third column ('fileType/value') returns the type of file the application is expecting if it is input or it returns the type of input necessary for the application parameters, common ones are string, boolean, etc. The last column gives brief details on each input.

7 Submitting Jobs in the rPlant package

7.1 Submitting Job

```
SubmitJob(application, file.path="", file.list=NULL, input.list, args.list=NULL,
job.name, nprocs=1, private.APP=FALSE, suppress.Warnings=FALSE, shared.username=NULL,
print.curl=FALSE, email=TRUE)
```

An important benefit of using *rPlant* is the ability to create batch-scripted files that automate job submittal and retrieval. For example, a user could submit parallel alignment jobs of different gene regions or multiple jobs with the same data and different parameter values. The results could then be automatically downloaded upon completion.

The following function is the main way to submit a job to the iPlant server, and can be used for any iPlant application. You can also submit jobs via the wrapper functions (for example, `Muscle`, see Section 9), which call upon the `SubmitJob` function internally.

```
> UploadFile(local.file.name="DNA.fasta", filetype="FASTA-0")
> myJobM <- SubmitJob(application="muscle-stampede-3.8.32", file.list=list("DNA.fasta"),
+                     input.list=list("stdin"), args.list=list(c("arguments",  private.APP=TRUE,
+                     "-phyiout -center -cluster1 upgma")), job.name="Muscle")
Job submitted.
You can check your job using CheckJobStatus(0001436233736406-e0bd34dffff8de6-0001-007)
```

Several important argument definitions are listed below, but can also be found in the help files: `input.list`: This argument defines what kind of input you are passing the application. You can get application information from the `GetAppInfo` function (`GetAppInfo("muscle-stampede-3.8.32")$Information`). In this example, the 'kind' column states there is one input for this app, and the 'id' column names that input as 'stdin'. Input types change from application to application.

`file.list`: Similar to `input.list`, the `file.list` argument defines which files are being passed to the application. The named file must be on the DE within the `file.path` and be

formatted to the correct specification (for example, `GetAppInfo("Muscle-3.8.32u4")$Information`). If it the file types don't match then the application will fail.

`args.list`: The `args.list` is where application flagging options can be entered. These typically change default options. Using information from the `GetAppInfo` function, the 'kind' column states there is one parameter for this app, the 'id' column gives the name of that parameter is "arguments", and the "fileType/value" column tells me it is a string. This is where the fourth column 'details' comes in handy; it tells me that the parameter input is "program arguments and options", which means it can accept a string. For example: `args.list=list(c(arguments, "-phyiout -center -cluster1 upgma"))`

The `args.list` is a list that is as long as the number of parameters (so length 7 in the above `velveth` example), that means there as many vectors as there are parameters. All vectors are of length two in all applications. In the first position, is the name of the parameter, "arguments", and in the second position is the value of that parameter, "-phyiout -center -cluster1 upgma". The positions for all applications have the same definitions, just different parameters and vary from application to application.

The function `SubmitJob` will return a list of two objects (in this example, `myJobM`). The first object, is the job number and the second is the job name, both are important information for retrieving results.

If job submits successfully, then the function automatically creates the folder "analyses" within a users cloud (if it did not previously exist). If the job finishes, then a folder is created within the analyses folder that is named the job name.

7.2 Submitting a job with a shared file

Jobs can also be submitted from files stored in other user's cloudspace that are shared. In the below example, a file that had been previously shared (Section 3), a job can be submitted using that file.

```
> myJobS <- SubmitJob(application="muscle-stampede-3.8.32", file.list=list("muscle3.fasta"),
+                     file.path="data", shared.username="phyllisl", private.APP = TRUE,
+                     args.list=list(c("arguments", "-fastaout")),
+                     input.list=list("stdin"), job.name="MuscleShare")

Job submitted.
You can check your job using CheckJobStatus(0001436233741997-e0bd34dffff8de6-0001-007)
```

8 Checking Job Status and Retrieving Job output

Once the job is submitted, it is assigned a job identification number (job.id) that we can use to check the status and download any results files. The job.id is returned with the SubmitJob function, so if you create an object when you submit a job then you can use that object as an identifier as well. Otherwise, you can copy/paste a job ID into any of the functions as a character string. The job number is used in a few rPlant functions including, `CheckJobStatus`, `ListJobOutput`, `RetrieveJob` and `DeleteJob`. If you need to get job IDs from older jobs, you can use the function `GetJobHistory()`.

8.1 Checking job status

`CheckJobStatus(job.id, print.curl=FALSE)`

This function checks the status of a job on the iPlant cloud.

Table 1: Possible Outputs for `CheckJobStatus()`

Stages
PENDING
STAGING_INPUTS
CLEANING_UP
ARCHIVING
STAGING_JOB
FINISHED
KILLED
FAILED
STOPPED
RUNNING
PAUSED
QUEUED
SUBMITTING
STAGED
PROCESSING_INPUTS
ARCHIVING_FINISHED
ARCHIVING_FAILED

```
> CheckJobStatus(myJobS[1])

[1] "PROCESSING_INPUTS"
```

8.2 Listing job status

`ListJobOutput(job.id, print.curl=FALSE, print.total=TRUE)`

This function lists the output files from a finished job. For example, these files are the output from our MUSCLE example above.

```
> ListJobOutput(myJobS[1])
[1] "Error: Job is PROCESSING_INPUTS"
```

Notice that an error was shown because the job output can't be found until the job is finished. So we will wait.

```
> ListJobOutput(myJobS[1])
[1] "There are 1 output files for job '0001436233741997-e0bd34dffff8de6-0001-007'"
[1] "muscle3.fasta"
```

8.3 Looking at Job History

`GetJobHistory(return.json=FALSE, print.curl=FALSE)`

This function displays the entire job history for the user. This is an easy way to grab old job IDs in order to retrieve files or check the status of a set of jobs.

```
> GetJobHistory()
```

```
job.id                job.name          application              status
"0001436233741997..." "MuscleShare..." "muscle-stampede-3.8.32" "FINISHED"
"0001436233736406..." "Muscle..."      "muscle-stampede-3.8.32" "PROCESSING_INPUTS"
```

8.4 Retrieve job files

```
RetrieveJob(job.id, file.vec=NULL, print.curl=FALSE, verbose=FALSE)
```

One very handy thing about the *rPlant* package is the ability to download the files directly from the iPlant cloud to your computer. The following downloads all of the output files.

```
> RetrieveJob(myJobS[1], file.vec=ListJobOutput(myJobS[1], print.total=FALSE))
```

The files have been downloaded into a new directory named after the job name within your working directory. One file at a time can be downloaded as well.

```
> RetrieveJob(myJobS[1], file.vec=c("fasta.aln"))
```

8.5 Stopping job

```
KillJob(job.id, print.curl=FALSE)
```

If a user wants a job stopped, then the `KillJob` function needs to be employed. It will stop the job and delte it from the job history.

```
> KillJob(myJobM[1])
```

8.6 Delete job

```
DeleteJob(job.id, print.curl=FALSE, ALL=FALSE)
```

After the job has been submitted, the results downloaded, and you have no need for the job anymore, you can use the `DeleteJob` function to delete the job. The nice thing about this function is that not only will it delete the job from the job history but it will also delete the job folder and all contents in the analyses folder in the user's cloudspace.

```
> DeleteJob(myJobS[1])
```

You also have the option to erase ALL job history from a user's past. This may be useful after a round of testing.

```
> DeleteJob(ALL=TRUE)
```

9 Submitting Jobs With Wrappers

We have nine dedicated wrapper functions for iPlant applications that will make submitting jobs very easy. These wrappers will use many application defaults and/or change flags into wrapper function arguments. Of course, if a user needs more flexibility in flagging options,

then they can still submit jobs using the `SubmitJob` function.

Writing wrapper functions is not programmatically difficult, but it does require familiarity with the individual programs and their associated data sets. We would like to encourage any users who are using programs without wrappers to submit patches adding wrapper functions or request to be a developer. You can make these feature requests at the R-Forge site: `https://r-forge.r-project.org/tracker/?group_id=1328`.

Among the wrappers there are three which do alignments: `Muscle`, `Mafft` and `ClustalW`. The alignments will do both protein and nucleotide. Also make sure that the taxon names in the sequence files do not contain tabulators, carriage returns, spaces, ":", ",", ")", "(", ";", "]", "[".

```
> data(PROTEIN.fasta)
> write.fasta(sequences=PROTEIN.fasta, names=names(PROTEIN.fasta), file.out="PROTEIN.fasta")
> UploadFile(local.file.name="PROTEIN.fasta", filetype="FASTA-0")
```

9.1 Muscle

`Muscle(file.name, file.path="", job.name=NULL, args=NULL, aln.filetype="PHYLIP_INT", shared.username=NULL, suppress.Warnings=FALSE, email=TRUE, print.curl=FALSE)`

MUSCLE is a program for creating multiple alignments of amino acid or nucleotide sequences. A range of options is provided that give you the choice of optimizing accuracy, speed, or some compromise between the two. The manual is also available here: `http://www.drive5.com/muscle/muscle_userguide3.8.html`

```
> myJobMuDP <- Muscle("DNA.fasta", aln.filetype="PHYLIP_INT", job.name="muscleDNAphyINT")

Job submitted.
You can check your job using CheckJobStatus(0001436233752831-e0bd34dffff8de6-0001-007)
Result file: phylip_interleaved.aln

> myJobMuDF <- Muscle("DNA.fasta", aln.filetype="FASTA", job.name="muscleDNAfasta")

Job submitted.
You can check your job using CheckJobStatus(0001436233758608-e0bd34dffff8de6-0001-007)
Result file: fasta.aln

> myJobMuDPS <- Muscle("DNA.fasta", aln.filetype="PHYLIP_SEQ", job.name="muscleDNAphySEQ")

Job submitted.
You can check your job using CheckJobStatus(0001436233764337-e0bd34dffff8de6-0001-007)
Result file: phylip_sequential.aln

> myJobMuDC <- Muscle("DNA.fasta", aln.filetype="CLUSTALW", job.name="muscleDNAclustalw")

Job submitted.
You can check your job using CheckJobStatus(0001436233770266-e0bd34dffff8de6-0001-007)
Result file: clustalw2.aln

> myJobMuDM <- Muscle("DNA.fasta", aln.filetype="MSF", job.name="muscleDNAmsf")

Job submitted.
You can check your job using CheckJobStatus(0001436233775814-e0bd34dffff8de6-0001-007)
Result file: msf.aln

> myJobMuPP <- Muscle("PROTEIN.fasta", aln.filetype="PHYLIP_INT", job.name="musclePROTEINphyINT")
```

```
Job submitted.
You can check your job using CheckJobStatus(0001436233780878-e0bd34dffff8de6-0001-007)
Result file: phylip_interleaved.aln
```

```
> myJobMuPF <- Muscle("PROTEIN.fasta", aln.filetype="FASTA", job.name="musclePROTEINfasta")
```

```
Job submitted.
You can check your job using CheckJobStatus(0001436233785816-e0bd34dffff8de6-0001-007)
Result file: fasta.aln
```

```
> myJobMuPPS <- Muscle("PROTEIN.fasta", aln.filetype="PHYLIP_SEQ", job.name="musclePROTEINphySEQ")
```

```
Job submitted.
You can check your job using CheckJobStatus(0001436233790841-e0bd34dffff8de6-0001-007)
Result file: phylip_sequential.aln
```

```
> myJobMuPC <- Muscle("PROTEIN.fasta", aln.filetype="CLUSTALW", job.name="musclePROTEINclustalw")
```

```
Job submitted.
You can check your job using CheckJobStatus(0001436233796047-e0bd34dffff8de6-0001-007)
Result file: clustalw2.aln
```

```
> myJobMuPM <- Muscle("PROTEIN.fasta", aln.filetype="MSF", job.name="muscleDNAmsf")
```

```
Job submitted.
You can check your job using CheckJobStatus(0001436233801067-e0bd34dffff8de6-0001-007)
Result file: msf.aln
```

MUSCLE outputs potentially six different alignments: "fasta.aln", "phylip_sequential.aln", "phylip_interleaved.aln", "clustalw2.aln', "msf.aln" and "html.aln".

9.2 Mafft

```
Mafft(file.name, file.path="", type="DNA", aln.filetype="FASTA", args=NULL, out.name=NULL,
print.curl=FALSE, job.name=NULL, email=TRUE, shared.username=NULL, suppress.Warnings=FALSE)
```

MAFFT is a multiple sequence alignment program for unix-like operating systems. It offers a range of multiple alignment methods, L-INS-i (accurate; for alignment of about 200 sequences), FFT-NS-2 (fast; for alignment of about 10,000 sequences), etc. See `http://mafft.cbrc.jp/alignment/software/`. The manual is also available here: `http://mafft.cbrc.jp/alignment/software/manual/manual.html`.

```
> myJobMaDF <- Mafft("DNA.fasta", job.name="mafftDNAfasta")
```

```
Job submitted.
You can check your job using CheckJobStatus(0001436233806191-e0bd34dffff8de6-0001-007)
Result file: mafft.aln
```

```
> myJobMaDC <- Mafft("DNA.fasta", aln.filetype="CLUSTALW", job.name="mafftDNAclustalw")
```

```
Job submitted.
You can check your job using CheckJobStatus(0001436233811481-e0bd34dffff8de6-0001-007)
Result file: mafft.aln
```

```
> myJobMaPF <- Mafft("PROTEIN.fasta", type="PROTEIN", job.name="mafftPROTEINfasta")
```

```
Job submitted.
You can check your job using CheckJobStatus(0001436233816640-e0bd34dffff8de6-0001-007)
Result file: mafft.aln
```

```
> myJobMaPC <- Mafft("PROTEIN.fasta", type="PROTEIN", aln.filetype="CLUSTALW",
+                    job.name="mafftPROTEINclustalw")
```

```
Job submitted.
You can check your job using CheckJobStatus(0001436233821843-e0bd34dffff8de6-0001-007)
Result file: mafft.aln
```

MAFFT outputs two alignments (both named: mafft.fa): FASTA (http://en.wikipedia.org/wiki/FASTA_format) and CLUSTALW (http://meme.nbcr.net/meme/doc/clustalw-format.html).

9.3 ClustalW

```
ClustalW(file.name, file.path="", type="DNA", aln.filetype="CLUSTALW", args=NULL,
out.name=NULL, job.name=NULL, print.curl=FALSE, shared.username=NULL, suppress.Warnings=FAI
email=TRUE)
```

An approach for performing multiple alignments of large numbers of amino acid or nucleotide sequences is described. The method is based on first deriving a phylogenetic tree from a matrix of all pairwise sequence similarity scores, obtained using a fast pairwise alignment algorithm. See details on http://www.clustal.org/clustal2/.

```
> myJobCWDP <- ClustalW("DNA.fasta", job.name="clustalwDNAphylip")

Job submitted.
You can check your job using CheckJobStatus(0001436233827008-e0bd34dffff8de6-0001-007)
Result file: clustalw2.aln

> myJobCWDC <- ClustalW("DNA.fasta", aln.filetype="CLUSTALW", job.name="clustalwDNAclustalw")

Job submitted.
You can check your job using CheckJobStatus(0001436233832129-e0bd34dffff8de6-0001-007)
Result file: clustalw2.aln

> myJobCWDN <- ClustalW("DNA.fasta", aln.filetype="NEXUS", job.name="clustalwDNAnexus")

Job submitted.
You can check your job using CheckJobStatus(0001436233837296-e0bd34dffff8de6-0001-007)
Result file: nexus.aln

> myJobCWDGCG <- ClustalW("DNA.fasta", aln.filetype="GCG", job.name="clustalwDNAgcg")

Job submitted.
You can check your job using CheckJobStatus(0001436233842922-e0bd34dffff8de6-0001-007)
Result file: gcg.aln

> myJobCWDGDE <- ClustalW("DNA.fasta", aln.filetype="GDE", job.name="clustalwDNAgde")

Job submitted.
You can check your job using CheckJobStatus(0001436233848597-e0bd34dffff8de6-0001-007)
Result file: gde.aln

> myJobCWDPIR <- ClustalW("DNA.fasta", aln.filetype="PIR", job.name="clustalwDNApir")

Job submitted.
You can check your job using CheckJobStatus(0001436233854153-e0bd34dffff8de6-0001-007)
Result file: pir.aln

> myJobCWPP <- ClustalW("PROTEIN.fasta", type="PROTEIN", job.name="clustalwPROTEINphylip")

Job submitted.
You can check your job using CheckJobStatus(0001436233859610-e0bd34dffff8de6-0001-007)
Result file: clustalw2.aln
```

```
> myJobCWPC <- ClustalW("PROTEIN.fasta", type="PROTEIN", aln.filetype="CLUSTALW",
+                       job.name="clustalwPROTEINclustalw")

Job submitted.
You can check your job using CheckJobStatus(0001436233865404-e0bd34dffff8de6-0001-007)
Result file: clustalw2.aln

> myJobCWPN <- ClustalW("PROTEIN.fasta", type="PROTEIN", aln.filetype="NEXUS",
+                       job.name="clustalwPROTEINnexus")

Job submitted.
You can check your job using CheckJobStatus(0001436233870724-e0bd34dffff8de6-0001-007)
Result file: nexus.aln

> myJobCWPGCG <- ClustalW("PROTEIN.fasta", type="PROTEIN", aln.filetype="GCG",
+                       job.name="clustalwPROTEINgcg")

Job submitted.
You can check your job using CheckJobStatus(0001436233876135-e0bd34dffff8de6-0001-007)
Result file: gcg.aln

> myJobCWPGDE <- ClustalW("PROTEIN.fasta", type="PROTEIN", aln.filetype="GDE",
+                       job.name="clustalwPROTEINgde")

Job submitted.
You can check your job using CheckJobStatus(0001436233881378-e0bd34dffff8de6-0001-007)
Result file: gde.aln

> myJobCWPPIR <- ClustalW("PROTEIN.fasta", type="PROTEIN", aln.filetype="PIR",
+                       job.name="clustalwPROTEINpir")

Job submitted.
You can check your job using CheckJobStatus(0001436233886809-e0bd34dffff8de6-0001-007)
Result file: pir.aln
```

`ClustalW` outputs six alignments (all named: clustalw.fa): CLUSTALW `http://meme.nbcr.net/meme/doc/clustalw-format.html`, PHYLIP_INT `http://www.bioperl.org/wiki/PHYLIP_multiple_alignment_format`, NEXUS `http://en.wikipedia.org/wiki/Nexus_file`, GCG `http://www.genomatix.de/online_help/help/sequence_formats.html#GCG`, GDE `http://www.cse.unsw.edu.au/~binftools/birch/GDE/overview/GDE.file_formats.html`, and PIR `http://www.bioinformatics.nl/tools/crab_pir.html`.
After this vignette has run out all of these analyses, we can kill any running processes and then delete all the jobs from our user history. If these were real analyses, you would need to finish the runs and save any output files.

9.4 Fasttree

`Fasttree(file.name, file.path="", job.name=NULL, out.name=NULL, args=NULL, type="DNA", model=NULL, gamma=FALSE, stat=FALSE, print.curl=FALSE, shared.username=NULL, email=TRUE, suppress.Warnings=FALSE)`

```
> data(fasta_aa.aln)
> write.table(fasta_aa.aln, "fasta_aa.aln", quote=FALSE, row.names=FALSE, col.names=FALSE)
> UploadFile("fasta_aa.aln")
> data(fasta_dna.aln)
> write.table(fasta_dna.aln, "fasta_dna.aln", quote=FALSE, row.names=FALSE, col.names=FALSE)
> UploadFile("fasta_dna.aln")
> data(phylip_interleaved_aa.aln)
> write.table(phylip_interleaved_aa.aln, "phylip_interleaved_aa.aln", quote=FALSE, row.names=FALSE, col.na
```

```
> UploadFile("phylip_interleaved_aa.aln")
> data(phylip_interleaved_dna.aln)
> write.table(phylip_interleaved_dna.aln, "phylip_interleaved_dna.aln", quote=FALSE, row.names=FALSE, col.
> UploadFile("phylip_interleaved_dna.aln")
```

FastTree infers approximately-maximum-likelihood phylogenetic trees from alignments of nucleotide or protein sequences. See http://meta.microbesonline.org/fasttree/

```
> myJobFaDMuP <- Fasttree("phylip_interleaved_dna.aln", job.name="fasttreeDNAphy")

Job submitted.
You can check your job using CheckJobStatus(0001436233895439-e0bd34dffff8de6-0001-007)

> myJobFaDCWP <- Fasttree("fasta_dna.aln", job.name="fasttreeDNAfasta")

Job submitted.
You can check your job using CheckJobStatus(0001436233896594-e0bd34dffff8de6-0001-007)

> myJobFaDMuF <- Fasttree("phylip_interleaved_aa.aln", job.name="fasttreeAAphy")

Job submitted.
You can check your job using CheckJobStatus(0001436233901564-e0bd34dffff8de6-0001-007)

> myJobFaDCWF <- Fasttree("fasta_aa.aln", job.name="fasttreeAAfasta")

Job submitted.
You can check your job using CheckJobStatus(0001436233907845-e0bd34dffff8de6-0001-007)
```

Fasttree outputs trees in Newick format http://en.wikipedia.org/wiki/Newick_format. The placement of the root is not biologically meaningful. The local support values are given as names for the internal nodes, and range from 0 to 1, not from 0 to 100 or 0 to 1,000. If all sequences are unique, then the tree will be fully resolved (the root will have three children and other internal nodes will have two children). If there are multiple sequences that are identical to each other, then there will be a multifurcation. Also, there are no support values for the parent nodes of redundant sequences.

9.5 RAxML (Randomized Accelerated Maximum Likelihood)

```
RAxML(file.name, file.path="", type="DNA", out.name=NULL, model=NULL, bootstrap=NULL,
algorithm="d", rseed=NULL, multipleModelFileName=NULL, args=NULL, numcat=25, nprocs=12,
job.name=NULL, print.curl=FALSE, shared.username=NULL, substitution_matrix=NULL,
empirical.frequencies=FALSE, suppress.Warnings=FALSE, email=TRUE)
```

RAxML is a program for sequential and parallel Maximum Likelihood based inference of large phylogenetic tress. It has originall been derived from from fastDNAml which in turn was derived from Joe Felsentein's dnaml which is part of the PHYLIP package. See http://bioinformatics.oxfordjournals.org/content/suppl/2014/01/18/btu033.DC1/NewManual.pdf for details.

```
> myJobRDMuP <- RAxML("phylip_interleaved_dna.aln", job.name="raxmlDNAphy")

Job submitted.
You can check your job using CheckJobStatus(0001436233911546-e0bd34dffff8de6-0001-007)

> myJobRDCWP <- RAxML("fasta_dna.aln", job.name="raxmlDNAfasta")

Job submitted.
You can check your job using CheckJobStatus(0001436233915548-e0bd34dffff8de6-0001-007)
```

```
> myJobRPMuP <- RAxML("phylip_interleaved_aa.aln", type="PROTEIN", job.name="raxmlAAphy")

Job submitted.
You can check your job using CheckJobStatus(0001436233927894-e0bd34dffff8de6-0001-007)

> myJobRPCWP <- RAxML("fasta_aa.aln", type="PROTEIN", job.name="raxmlAAfasta")

Job submitted.
You can check your job using CheckJobStatus(0001436233923451-e0bd34dffff8de6-0001-007)
```

For this application there are numerous output files. See pg 16-17 of the manual for complete details. RAxML outputs trees in Newick format `http://en.wikipedia.org/wiki/Newick_format`.

9.6 Genome Wide Association Study models

```
> UploadFile("simulation1.map")
> UploadFile("simulation1.ped")
> UploadFile("geno_test.tfam")
> UploadFile("geno_test.tped")
```

9.7 PLINK

`PLINK(file.list="", file.path="", job.name=NULL, out.basename=NULL, association.method="-assoc", no.sex=TRUE, args=NULL, print.curl=FALSE, multi.adjust=TRUE, email=TRUE, shared.username=NULL, suppress.Warnings=FALSE)`

PLINK is an open-source whole genome association analysis toolset, designed to perform a range of basic, large-scale analyses in a computationally efficient manner, check `http://pngu.mgh.harvard.edu/~purcell/plink/` for details.

```
> myJobPLINKT <- PLINK(file.list=list("geno_test.tfam","geno_test.tped"), job.name="PLINKT")

Job submitted.
You can check your job using CheckJobStatus(0001436306803947-e0bd34dffff8de6-0001-007)

> myJobPLINKR <- PLINK(file.list=list("simulation1.map","simulation1.ped"), job.name="PLINKR")

Job submitted.
You can check your job using CheckJobStatus(0001436306809230-e0bd34dffff8de6-0001-007)
```

There are many output files possible, `http://pngu.mgh.harvard.edu/~purcell/plink/reference.shtml#output`

9.8 PLINK Conversion

`PLINKConversion(file.list="", file.path="", output.type="-recode", job.name=NULL, shared.username=NULL, print.curl=FALSE, suppress.Warnings=FALSE, out.basename=NULL, email=TRUE)`

This function converts the standard PLINK file formats (Regular (ped/map), Transposed (tped/tfam), and Binary (bed/bim/fam)) to various other PLINK file formats.

```
> myJobPLINKCT <- PLINKConversion(file.list=list("geno_test.tfam","geno_test.tped"), job.name="PCT",
+                                 out.basename="plinkout")

Job submitted.
You can check your job using CheckJobStatus(0001436306814749-e0bd34dffff8de6-0001-007)
```

```
> myJobPLINKCR <- PLINKConversion(file.list=list("simulation1.map","simulation1.ped"),
+                                 output.type="--recode --transpose", job.name="PCR")

Job submitted.
You can check your job using CheckJobStatus(0001436306819992-e0bd34dffff8de6-0001-007)
```

There are many output files possible, http://pngu.mgh.harvard.edu/~purcell/plink/reference.shtml#output

9.9 FaST-LMM (Factored Spectrally Transformed Linear Mixed Models)

```
FaST_LMM(input.file.list="", ALL.file.path="", print.curl=FALSE, sim.file.list=NULL,
pheno.file.name=NULL, mpheno=1, args=NULL, covar.file.name=NULL, job.name=NULL,
shared.username=NULL, suppress.Warnings=FALSE, out.basename=NULL, email=TRUE)
```

FaST-LMM (Factored Spectrally Transformed Linear Mixed Models) is a program for performing genome-wide association studies (GWAS) on large data sets. FaST-LMM is described more fully at http://www.nature.com/nmeth/journal/v8/n10/abs/nmeth.1681.html, and also at http://fastlmm.codeplex.com/

```
> myJobFaST_LMMT <- FaST_LMM(input.file.list=list("geno_test.tfam","geno_test.tped"),
+                            job.name="FaST_LMMT")

Job submitted.
You can check your job using CheckJobStatus(0001436306825117-e0bd34dffff8de6-0001-007)

> myJobFaST_LMMR <- FaST_LMM(input.file.list=list("simulation1.map","simulation1.ped"),
+                            job.name="FaST_LMMR")

Job submitted.
You can check your job using CheckJobStatus(0001436307409789-e0bd34dffff8de6-0001-007)
```

Not all information on the FaST-LMM model is here, see the FaST-LMM website http://fastlmm.codeplex.com/, or the FaST-LMM manual for more information.

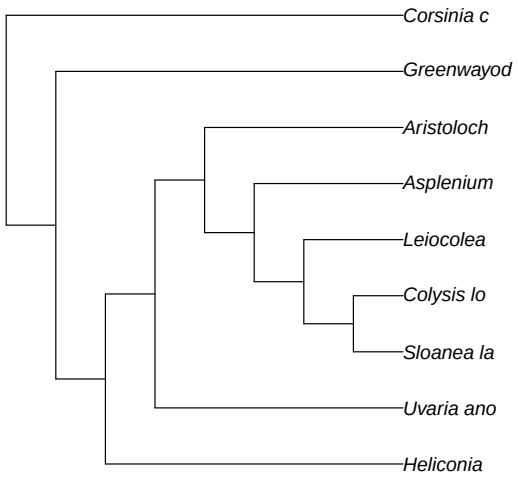
10 Creating workflows

Finally putting everything together, each of these steps can be combined to generate multi-step analyses. This has the benefit of reducing errors that can occur when manually running each application and, more importantly, ensures that results are reproducible. In the following example, a user starts with unaligned sequences on their local computer and ends with aligned sequences and a phylogenetic tree with all applications running on the iPlant cloud. One must be careful though, when running on iPlant cloud the jobs can take days to finish, and when one employs a workflow it could lock up an R session for that long. It would also be just as good to not use the Wait function and you personally keep checking until the job is finished to complete the workflow.

10.1 Workflow One

This first workflow takes an amino acid fasta file, uses `MUSCLE` to get a `FASTA` alignment type. Then `RAxML` model produces a tree which is downloaded and plotted in **R**.

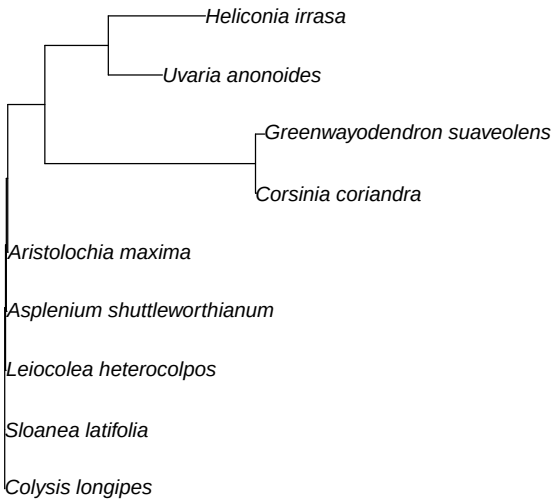
```
> myJobW1MP <- Muscle("PROTEIN.fasta", aln.filetype="FASTA", job.name="muscleWORKFLOW1protein")
> Wait(myJobW1MP[[1]], minWait, maxWait)
> myJobW1RFP <- RAxML("fasta.aln", file.path=paste("analyses/",myJobW1MP[[2]],
+                     sep=""), type="PROTEIN",job.name="raxmlWORKFLOW1protein")
> Wait(myJobW1RFP[[1]], minWait, maxWait)
> RetrieveJob(myJobW1RFP[[1]], c("outtreFe.nwk"))
> read.tree(paste(getwd(), myJobW1RFP[[2]], "RAxML_bestTree.nwk", sep="/")) -> Tree
> plot(Tree)
```



10.2 Workflow Two

The second workflow takes the same amino acid fasta file and this time it uses `Mafft` to get a `FASTA` alignment type. `MUSCLE` also can output a fasta alignment. The `FastTree` model is then used to make the tree.

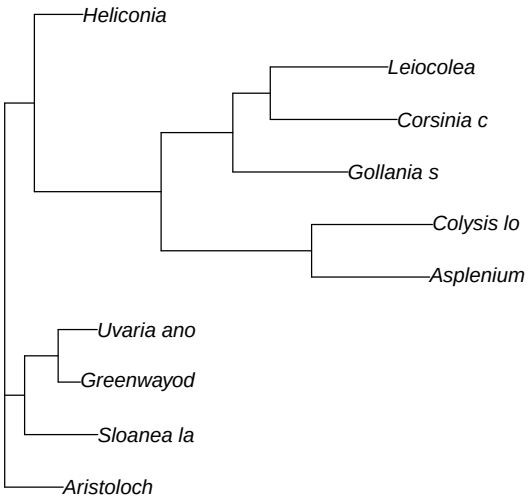
```
> myJobW2CWD <- Mafft("PROTEIN.fasta", type="PROTEIN", job.name="mafftPROTEINfasta")
> Wait(myJobW2CWD[[1]], minWait, maxWait)
> myJobW2FaD <- Fasttree("mafft.fa", type="PROTEIN", file.path=paste("analyses/",myJobW2CWD[[2]],
+                        sep=""), job.name="fasttreeCLUSTALWfasta")
> Wait(myJobW2FaD[[1]], minWait, maxWait)
> RetrieveJob(myJobW2FaD[[1]], c("fasttree.nwk"))
> read.tree(paste(getwd(), myJobW2FaD[[2]], "fasttree.nwk", sep="/")) -> Tree
> plot(Tree)
```



10.3 Workflow Three

The third workflow is again dealing with `FastTree`. `FastTree` can take either a FASTA alignment or a phylip interleaved alignment as inputs. Now `ClustalW` takes a nucleotide fasta file to get a `PHYLIP INTERLEAVED` alignment type. `MUSCLE` also can output that alignment. The `FastTree` model is then used to make the tree.

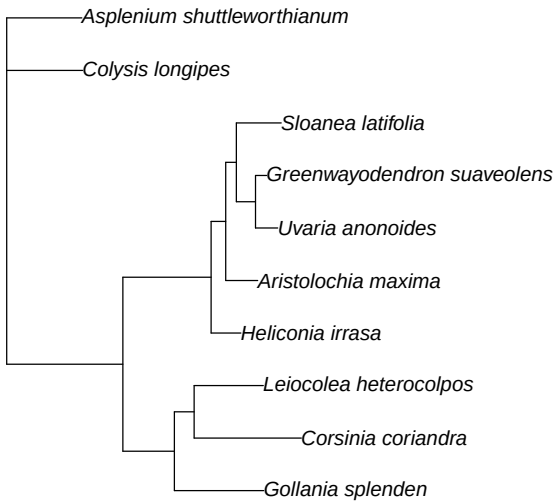
```
> myJobW3MuP <- ClustalW("DNA.fasta", job.name="clustalwDNAfasta")
> Wait(myJobW3MuP[[1]], minWait, maxWait)
> myJobW3FaP <- Fasttree("clustalw2.fa", file.path=paste("analyses/",myJobW3MuP[[2]], sep=""),
+                        job.name="fasttreeMUSCLEdna")
> Wait(myJobW3FaP[[1]], minWait, maxWait)
> RetrieveJob(myJobW3FaP[[1]], c("fasttree.nwk"))
> read.tree(paste(getwd(), myJobW3FaP[[2]], "fasttree.nwk", sep="/")) -> Tree
> plot(Tree)
```



10.4 Workflow Four

The fourth workflow is using `RAxML`. `MUSCLE` takes a nucleotide fasta file to get a `PHYLIP INTERLEAVED` alignment type. `ClustalW` also can output that alignment. The `RAxML` model is then used to make the tree.

```
> myJobW4MuP <- Muscle("DNA.fasta", aln.filetype="PHYLIP_INT", job.name="muscleWORKFLOW4dna")
> Wait(myJobW4MuP[[1]], minWait, maxWait)
> myJobW4RP <- RAxML("phylip_interleaved.aln", file.path=paste("analyses/",myJobW4MuP[[2]], sep=""),
+                    job.name="raxmlWORKFLOW4dna")
> Wait(myJobW4RP[[1]], minWait, maxWait)
> RetrieveJob(myJobW4RP[[1]], c("RAxML_bestTree.nwk"))
> read.tree(paste(getwd(), myJobW4RP[[2]], "RAxML_bestTree.nwk", sep="/")) -> Tree
> plot(Tree)
```



References

Dooley, R., Vaughn, M., Stanzione, D., Terry, S., and Skidmore, E. "Software-as-a-Service: The iPlant Foundation API." In *5th IEEE Workshop on Many-Task Computing on Grids and Supercomputers* (2012).
URL `http://datasys.cs.iit.edu/events/MTAGS12/p07.pdf`

Fielding, R. T. "Architectural styles and the design of network-based software architectures." Ph.D. thesis, University of California (2000).

Lang, D. T. "R as a Web Client–the RCurl package." *Journal of Statistical Software, http://www. jstatsoft. org* (2007).

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria (2015).
URL `http://www.R-project.org/`

Stenberg, D. "1.47. 1 Available under license." *Open Source Used In Cisco Unified Communications Manager 8.6. 2a*, 5:395 (1996).