Read the Docs Template Documentation

Release 1.0

Read the Docs

May 14, 2020

General

CHAPTER 1

Project

Tertiary workflows in single-cell RNA-seq analysis are, in the definition of the Human Cell Atlas (HCA), those processes occuring subsequent to quantification, such as clustering and trajectory inference. Example libraries implementing these analyses are Seurat, Scater and Scanpy.

This is a project to increase the re-usability and reproducibility of these workflows, to facilitate comparison and benchmarking. Our strategy is:

1. Provide command-line access to individual library functions through simple wrapper scripts packaged with Bioconda.
2. Enable pipeline inter-operability by adopting and/or improving file format standards (e.g. Loom) which can be read and written by a variety of packages.
3. Facilitate the generation of workflows built on the above components, in environments such as Galaxy and Nextflow, for deployment in platform-agnostic ways.

1.1 Project overview

Single-cell tertiary analyses include step such as:

- expression matrix normalisation
- cell filtering
- computation of dimension reductions
- clustering itself

These steps may be implemented in a variety of ways including stand-alone tools, scripts, or R package functions. To compare equivalent logical steps between workflows, and to 'mix and match' those components for optimal workflows is therefore a challenging excercise without additional infrastructure.

We aim to 'componentise' tools associated with these workflows and more importantly build standards and infrastructure to facilitate this process for others in future.

1.1.1 Objectives

- Write additional scripts where necessary to expose inividual worklow units as scripts with well-defined inputs and outputs.
- Build (programming) language-independent standards for input, output and intermediate data types.
- Make components avialable in as simple a manner as possible, via the Bioconda repository.
- Make containers available for these packages, suitable for cloud deployment.
- Make preliminary assessments of individual workflows, components, and 'mixed' workflows.

1.2 Authors

This project is a collaboration between groups at the European Bioinformatics Institute (EBI) and the Sanger Institute. It is coordinated by Irene Papatheodorou (EBI) and Kerstin Meyer (Sanger). The following individuals are involved in implementation:

- Pablo Moreno, Gene Expression Group, EBI
- Ni Huang, Sanger Institute
- Carlos Talavera-López, Sanger Institute
- Jonathan Manning, Gene Expression Group, EBI
- Suhaib Mohammed, Gene Expression Group, EBI

We are also grateful for the participation of many members of the Galaxy community.

1.3 Components

We are building components in the following categories:

- Wrappers script packages for common single-cell analysis toolkits
- Bioconda packages to facilitate wrapper availability
- Bioconda packages for any previously missing packages
- Containers for Bioconda packages (mostly created automatically via Bioconda)
- Galaxy wrappers for Bioconda-packages wrappers, and associated workflows
- Nextflow workflows built on packaged script components

This page details these components.

Contents

        * *Scanpy*
        * *R/Bioconductor SingleCellExperiment ecosystem*
            - *DropletUtils*
            - *Scater*
            - *SC3*
        * *Seurat*
        * *UCSC Cell Browser*
    - *Low-level workflows*

1.3.1 Wrapper script packages

- workflowscriptscommon - Common functions for re-use over R package wrappers
- bioconductor-singlecellexperiment-scripts - Wrapper scripts around SingleCellExperiment R functions
- dropletutils-scripts - Wrapper scripts for DropletUtils R functions
- bioconductor-scater-scripts - Wrapper scripts around Scater R functions
- bioconductor-sc3-scripts - Wrapper scripts around SC3 R functions
- r-seurat-scripts - Wrapper scripts around Seurat R functions
- scanpy-scripts - Wrapper scripts for Scanpy functions

1.3.2 Bioconda packages

The above packages have been made available via Bioconda in the packages:

- r-workflowscriptscommon
- bioconductor-singlecellexperiment-scripts
- dropletutils-scripts
- bioconductor-scater-scripts
- sc3-scripts
- seurat-scripts
- scanpy-scripts

We also added the previously missing Bioconda package for the DropletUtils Bioconductor package itself:

- bioconductor-dropletutils

1.3.3 Containers

Use of Bioconda packages means we get containers 'for free' as well. See containers for:

- r-workflowscriptscommon
- bioconductor-singlecellexperiment-scripts
- bioconductor-dropletutils

- dropletutils-scripts
- bioconductor-scater-scripts
- sc3-scripts
- seurat-scripts
- scanpy-scripts

We have also developed the following:

- container-galaxy-sc-tertiary - Galaxy container for single cell RNA-Seq tertiary analysis tools

1.3.4 Galaxy components

The following components will shortly be available in a Galaxy 'toolshed'. These are all powered by the same scripts and Conda packages described above, and in this way analyses can be made consistent, whatever workflow construct is used to connect them.

Scanpy

(currently at v1.3.2)

- Read10x - Read 10x into hdf5 object handled by scanpy
- FilterCells - Filter cells based on counts and numbers of genes expressed
- FilterGenes - Filter genes based on counts and numbers of cells expressed
- NormaliseData - Normalise data to make all cells having the same total expression
- FindVariableGenes - Find variable genes based on normalised dispersion of expression
- ScaleData - Scale data to make expression variance the same for all genes
- RunPCA - Run PCA for dimensionality reduction
- ComputeGraph - Compute graph to derive kNN graph
- FindClusters - Find clusters based on community detection on KNN graph
- RunUMAP - Run UMAP to visualise cell clusters using UMAP
- RunTSNE - Run tSNE to visualise cell clusters using tSNE
- FindMarkers - FindMarkers to find differentially expressed genes between groups

R/Bioconductor SingleCellExperiment ecosystem

DropletUtils

(currently at v1.0.3)

Read 10x data into a SingleCellExperiment object

Scater

(currently at v1.8.4)

- CalculateCPM - CalculateCPM from raw counts
- Normalise - Normalise expression values by library size in log2 scale
- CalculateQcMetrics - CalculateQcMetrics based on expression values and experiment information
- DetectOutlier - DetectOutlier cells based on expression metrics
- Filter - Filter cells and genes based on pre-calculated stats and QC metrics

SC3

(currently at v1.8.0)

- Prepare - Prepare a sc3 SingleCellExperiment object
- Estimate -Estimate the number of clusters for k-means clustering
- Calculate - Transformations Calculate Transformations of distances using PCA and graph Laplacian
- Calculate - Distances Calculate Distances between cells
- K-Means - K-Means perform k-means clustering
- Calculate - Consensus Calculate Consensus from multiple runs of k-means clustering
- DiffExp - Calculates DE genes, marker genes and cell outliers

Seurat

(currently at v2.3.1)

- Read10x - Loads 10x data into a serialized seurat R object
- CreateSeuratObject - create a Seurat object
- FilterCells - filter cells in a Seurat object
- NormaliseData - normalise data
- FindVariableGenes - identify variable genes
- ScaleData - scale and center genes
- RunPCA - run a PCA dimensionality reduction
- RunTSNE - run t-SNE dimensionality reduction
- Plot dimension reduction - graphs the output of a dimensional reduction technique (PCA by default). Cells are colored by their identity class.
- FindClusters - find clusters of cells
- FindMarkers - find markers (differentially expressed genes)
- Export2CellBrowser - Export2CellBrowser produces files for UCSC CellBrowser import.

UCSC Cell Browser

(currently at 0.4.38)

UCSC Cell Browser displays single-cell clusterized data in an interactive web application.

1.3.5 Low-level workflows

The following Nextflow workflows are available:

- scanpy-workflow - This is a fully pameterised workflow linking the components of scanpy-scripts.

1.4 Running Hinxton Single Cell Interactive Analysis Environment

Hinxton Single Cell is a Galaxy setup with tools for tertiary analysis of single cell data. Galaxy is an open, web-based platform for accessible, reproducible, and transparent computational biomedical research.

Within this project, we have created a Galaxy flavour (a Galaxy setup with defined tools and workflows) for the tertiary analysis of single cell RNA-Seq data.

This guide shows you how to run the Galaxy Single Cell Tertiary analysis setup on your own machine using Minikube. The setup relies on the Kubernetes container orchestrator.

1.4.1 Requirements

- Helm installed: Please follow official instructions for installing Helm.
- Access to a Kubernetes cluster (with shared file system accessible through a Persistent Volume or equivalent). - For development purposes or local tests, the local Minikube environment can be used. Install minikube following official instructions for minikube.
- kubectl cli: The command line argument for connection to a Kubernetes instance (remote cluster or local minikube). If not installed as part of Minikube steps, follow ONLY the installation steps (not the configuration ones) from here.

Minikube

If using minikube, you need to make sure that it is running. If you just installed it, you need to execute `minikube start`. In general you can check the status of minikube through `minikube status`. For a smoother run, you might want to assign more RAM to Minikube.

1.4.2 First time installation

If using helm for the first time, you will need to initialise helm on the cluster and add the helm repo to the local helm directories:

```
helm init --wait
helm repo add galaxy-helm-repo https://pcm32.github.io/galaxy-helm-charts
```

if you have done this once in the past, you might need, from time to time, to update the local repo, by doing:

```
helm repo update
```

1.4.3 Retrieve and customise configuration file

The galaxy-stable Helm chart that we will use allows the configuration of most Galaxy settings. For that, get the sample config file.

Edit or make a copy of this file based on your needs and the galaxy-stable chart documentation. However, if you only need to run on Minikube, the sample config file should do fine. You might want to set a user and password there.

1.4.4 Normal run

Now the only step needed is to initialize Galaxy through the helm chart just added. For this execute:

```
helm install -f hinxton-singlecell-1.0.0_g18.05-minikube.yaml \
  --version 2.0.2 galaxy-helm-repo/galaxy-stable
```

1.4.5 Access Galaxy

To access your local Galaxy setup you need to find out the ip from minikube:

```
minikube ip
```

Usually the ip number will be `192.168.99.100` (but confirm with the above call). Then access on that ip port 30700 with the galaxy prefix `http://192.168.99.100:30700/`.

If you are not using minikube, the Galaxy instance will be on port 30700 of any of the node's IPs of the k8s cluster (unless that you had setup the ingress).

1.5 Provide wrapper scripts

You can help by writing or providing your wrapper scripts to make functionality from libraries more accessible from the command line. Adhering to some standards while doing this makes scripts more useful to the community

1.5.1 Wrapper script recommendations

Writing wrapper scripts for R packages

Recommended resources

optparse

We recommend use of the optparse package for development of user-friendly R scripts.

workflowscriptscommon R package

We have built the workflowscriptscommon package to hold functions common to R script wrapper writing. Using this package (and adding functions where required) is a useful way of reducing repetition between R wrappers.

The package is available in Bioconda and can be installed using conda, like:

```
conda install r-workflowscriptscommon
```

Existing functions

The workflowscriptscommon package defines some simple functions that will be of use in writing wrappers:

- wsc_parse_args() wraps the use of optparse's OptionParser() for convenience, allowing checks for mandatory arguments
- wsc_parse_numeric() parses numeric vectors out of strings
- wsc_split_string() parses character vectors out of strings
- wsc_read_vector() parses a named vector from a two-column file
- wsc_write_vector() writes a named vector to a two-column file

New functions

If you develop further functions that might be of use across other wrapper scripts, you can propose them for addtion to this package via pull request to the develop branch of the source repository at https://github.com/ebi-gene-expression-group/workflowscriptscommon. If you need to use new functions before they are added to a release and made available via the Bioconda package, you can install the R package directly from GitHub:

```
devtools::install_github('https://github.com/ebi-gene-expression-group/
↪workflowscriptscommon', ref='develop')
```

Define inputs and outputs

The first step in writing a wrapper script is to define inputs and outputs and which function(s) are required to run the desired process. Look at those functions' documentation (?function), record their inputs and outputs their types, and decide how those arguments will be processed by the script.

Suggested patterns:

- string arguments can be passed directly from the command line
- short string or numeric vectors can be passed as comma-separated strings
- longer vectors (gene lists etc) can be supplied as files with one entry per line, read via readLines() and written via writeLines()

Writing scripts

Each wrapper script should be named to reflect the r package and the functions called as closely as possible. For example a wrapper for the single calculateCPM() function of scater will be called scater-calculate-cpm.R. For more

complex wrappers involving multiple function calls, make sure your naming is sensible and easily interpretable by the user. We prefer hyphenated script names.

An example wrapper scripts is seurat-read-10x.R from the r-seurat-scripts package. Some useful conventions demonstrated there are:

Use 'env' to locate the Rscript binary in the hash-bang:

```
#!/usr/bin/env Rscript
```

Then load the package 'optparse' (for option parsing) and the common functions package:

```
suppressPackageStartupMessages(require(optparse))
suppressPackageStartupMessages(require(workflowscriptscommon))
```

Note the use of suppressPackageStartupMessages to make sure we don't ouput more junk to stdout than we need to. DO NOT put the command for loading the main R package here (for example Seurat). Loading such packages can take a surprisingly long time, and we need to make sure our command-line arguments are good before committing to it.

Argument parsing

Now write the code that will parse command line options in the finished script. Use make_option() from the optparse package to parse the command line arguments into a list of objects as defined by your examination of inputs and outputs defined above. Here is an example of the 10x data parsing function, taken from a wrapper in the r-seurat-scripts package:

```
option_list = list(
  make_option(
    c("-d", "--data-dir"),
    action = "store",
    default = NA,
    type = 'character',
    help = "Directory containing the matrix.mtx, genes.tsv, and barcodes.tsv files␣
↪provided by 10X. A vector or named vector can be given in order to load several␣
↪data directories. If a named vector is given, the cell barcode names will be␣
↪prefixed with the name."
  ),
  make_option(
    c("-o", "--output-object-file"),
    action = "store",
    default = NA,
    type = 'character',
    help = "File name in which to store serialized R matrix object."
  )
)
```

This takes two character arguments specifying input and output files. We can then use one of the functions mentioned above to parse out the actual argument values whilst checking that no mandatory arguments are missing:

```
opt <- wsc_parse_args(option_list, mandatory = c('input_object_file', 'output_object_
↪file'))
```

You may also want to check the values yourself, for example to see if files specified are actually present:

```
# Check parameter values

if ( ! file.exists(opt$input_object_file)){
  stop((paste('Directory', opt$input_object_file, 'does not exist')))
}
```

Translating files to data structures

When writing wrapper scripts pay careful attention to how the data types required for the wrapped R function relate to how that information is supplied to the wrapper script itself. For example, where input is a vector but it's likely to be very short (e.g. a list of gene biotypes), it might be acceptable to supply this list to the script in a simple comma-separated string, which can be parsed into a vector using wsc_split_string(). Longer lists (e.g. gene names) should be supplied in a single-column text file that can be parsed using readLines(). Where vector names are important, for example specifying set of values for a metadata variable for a list of cells, these should be supplied in two-column (label/value) rows which can then be parsed by wsc_read_vector().

These datatype handling operations will likely need to evolve- please contribute using the PR mechanism on the workflowscripts common package as mentioned above.

Processing and outputs

The above done, feel free to load the package whose functions you're wrapping, and write the processing functionality:

```
suppressPackageStartupMessages(require(Seurat))
```

Once you have added processing code, pay attention to the output formats you use. R objects should be serialised using saveRDS(), and where feasible additional text-based formats should be used. Even complex R objects will eventually need to be output as formats readable by e.g. Python, but this not essential right now.

As a final point, make sure all wrapper scripts are executable:

```
chmod +x <script>
```

Writing wrapper scripts for Python packages

Adding scripts to existing wrapper packages

If a wrapper script package exists but does not contain a wrapper for functionality you need, please propose a new wrapper via a pull request. See *Components* for currently written or in-progress components.

To do this write a wrapper following convention and guidelines where possible (see e.g. *Writing wrapper scripts for R packages*), and submit a PR e.g. to r-seurat-scripts. Note that the script won't be available via Bioconda until there's a new release including your script, and the Bioconda recipe has been updated to point to that new release, but this is also a process you can contribute to.

Writing tests

Along with any collection of scripts you write for a package, you should provide testing code which can be run by users after installation of your package (probably via Conda). The test suite per package is composed of at least:

- A bash script which sets up variables, defines inputs and outputs, and exports them, named for the package so that it does not clash with similar scripts when installed by Conda. So the package bioconductor-singlecellexperiment-scripts has a test script called bioconductor-singlecellexperiment-scripts-post-install-tests.sh.
- A set of 'bats' tests (see https://github.com/bats-core/bats-core) to make sure all the tools run.

Test data

Do not package test data with your package. Instead include locations from which data can be downloaded during testing.

Example: bioconductor-singlecellexperiment-scripts

The testing setup for bioconductor-singlecellexperiment-scripts at time of writing was as follows (it's probably evolved a bit now, but the example is still valid).

Bash script

There is a starting section with script usage, where we work out what the script is called and where it is:

```
#!/usr/bin/env bash

script_dir=$(cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )
script_name=$0

# This is a test script designed to test that everything works in the various
# accessory scripts in this package. Parameters used have absolutely NO
# relation to best practice and this should not be taken as a sensible
# parameterisation for a workflow.

function usage {
    echo "usage: $script_name [action] [use_existing_outputs]"
    echo "  - action: what action to take, 'test' or 'clean'"
    echo "  - use_existing_outputs, 'true' or 'false'"
    exit 1
}

action=${1:-'test'}
use_existing_outputs=${2:-'false'}

if [ "$action" != 'test' ] && [ "$action" != 'clean' ]; then
    echo "Invalid action"
    usage
fi

if [ "$use_existing_outputs" != 'true' ] && [ "$use_existing_outputs" != 'false' ];␣
↪then
    echo "Invalid value ($use_existing_outputs) for 'use_existing_outputs'"
    usage
fi
```

The next section defines the remote locations of test data, and local paths to store them in. It also allows for cleanup of test data (provided with the appropriate option):

```
test_matrix_url="https://raw.githubusercontent.com/jdblischak/singleCellSeq/master/
↪data/molecules.txt"
test_annotation_url="https://raw.githubusercontent.com/jdblischak/singleCellSeq/
↪master/data/annotation.txt"

test_working_dir=`pwd`/'post_install_tests'
export test_matrix_file=$test_working_dir/test_data/`basename $test_matrix_url`
export test_annotation_file=$test_working_dir/test_data/`basename $test_annotation_
↪url`

# Clean up if specified

if [ "$action" = 'clean' ]; then
    echo "Cleaning up $test_working_dir ..."
    rm -rf $test_working_dir
    exit 0
elif [ "$action" != 'test' ]; then
    echo "Invalid action '$action' supplied"
    exit 1
fi

# Initialise directories

output_dir=$test_working_dir/outputs
data_dir=$test_working_dir/test_data

mkdir -p $test_working_dir
mkdir -p $output_dir
mkdir -p $data_dir

################################################################################
# Fetch test data
################################################################################

if [ ! -e "$test_matrix_file" ]; then
    wget $test_matrix_url -P $data_dir
    wget $test_annotation_url -P $data_dir
fi
```

Now there is a section where we define inputs and outputs, and where we would also store any parameter values we need the scripts to use during testing:

```
################################################################################
# List tool outputs/ inputs
################################################################################

export raw_singlecellexperiment_object="$output_dir/raw_sce.rds"

## Test parameters- would form config file in real workflow. DO NOT use these
## as default values without being sure what they mean.
```

Note the use of 'export'- only exported variables will be available for testing by bats.

Lastly we call the bats testing script, predicting its name from the name of the current script:

```
################################################################################
# Test individual scripts
################################################################################
```

(continues on next page)

(continued from previous page)

```
# Make the script options available to the tests so we can skip tests e.g.
# where one of a chain has completed successfullly.

export use_existing_outputs

# Derive the tests file name from the script name

tests_file="${script_name%.*}".bats

# Execute the bats tests

$tests_file
```

Note the export of the 'use_existing_outputs' variable. Bats will be able to use this to decide whether or not to bother re-running a test.

Bats script

There's only one script in this package, so we have a single Bats test:

```
#!/usr/bin/env bats

@test "SingleCellExperiment creation" {
    if [ "$use_existing_outputs" = 'true' ] && [ -f "$raw_singlecellexperiment_object
↪" ]; then
        skip "$use_existing_outputs $raw_singlecellexperiment_object exists and use_
↪existing_outputs is set to 'true'"
    fi

    run rm -f $raw_singlecellexperiment_object && singlecellexperiment-create-single-
↪cell-experiment.R -a $test_matrix_file -c $test_annotation_file -o $raw_
↪singlecellexperiment_object
    echo "status = ${status}"
    echo "output = ${output}"

    [ "$status" -eq 0 ]
    [ -f  "$raw_singlecellexperiment_object" ]
}
```

Only exported variables are available for use in the test (as well as Bats' own ones, such as '$status').

Things to note:

- We have a conditional use of Bats' 'skip' command. If we have specified that existing outputs should be re-used (via $use_existing_outputs), and if that output file exists, then then test will not run.
- 'run' is a Bats command which executes a command and stores its return code in '$status'.
- The 'echo' elements make sure that we see the full output of any errors- which are otherwise stashed away by Bats.
- The final bracketed elements are lists of assertions. We're saying that we should not have a non-zero return code, and the output file should exist once the commmand has run.

Travis CI integration

You may find it useful to integrate Travis CI to test scripts automatically during development. We have found a .travis.yml file like the following to be useful:

```
before_install:
  - if test -e $HOME/miniconda/bin; then
      echo "miniconda already installed.";
    else
      if [[ "$TRAVIS_PYTHON_VERSION" == "2.7" ]]; then
        wget https://repo.continuum.io/miniconda/Miniconda2-latest-Linux-x86_64.sh -O␣
↪miniconda.sh;
      else
        wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh -O␣
↪miniconda.sh;
      fi

      rm -rf $HOME/miniconda;
      bash miniconda.sh -b -p $HOME/miniconda;
      export PATH="$HOME/miniconda/bin:$PATH";
      hash -r;
      conda config --set always_yes yes --set changeps1 no;
      conda update -q conda;

      conda info -a;

      conda config --add channels defaults;
      conda config --add channels conda-forge;
      conda config --add channels bioconda;

      conda create -q -n test-environment python=$TRAVIS_PYTHON_VERSION r-base r-
↪optparse libpng r-cairo r-workflowscriptscommon bioconductor-singlecellexperiment;
    fi

install:
  - export PATH="$HOME/miniconda/bin:$PATH";
  - source activate test-environment

before_script:
  - export PATH=`pwd`:$PATH

script: ./bioconductor-singlecellexperiment-scripts-post-install-tests.sh

cache:
  directories:
    - $HOME/miniconda
    - post_install_tests

  before_cache:
    - if ! [[ $TRAVIS_TAG ]]; then rm -rf $HOME/miniconda/conda-bld; fi
    - rm -rf post_install_tests/outputs
    - rm -rf $HOME/miniconda/locks $HOME/miniconda/pkgs $HOME/miniconda/var $HOME/
↪miniconda/conda-meta/history
```

This sets up a Conda environment and exectutes the main test script for the package. Note that the conda evironment is cached, as is the directory containing test data, to prevent that data being re-downloaded on every build.

1.6 Create Bioconda packages

Making packages or scripts available via Bioconda makes them easily available to people using the Conda packaging system. A Conda package is created from a definition called a 'recipe', composed primarily of a meta.yaml file to define metadata associated with a package, and a script, build.sh, which installs the software. See https://bioconda.github.io/contributing.html for detailed info.

If a package you're interested in using is not currently available via Bioconda, you can write a recipe very easily and submit to Bioconda as a PR. Once accepted, the package will be available via the usual 'conda install' mechanism. For example we wanted to use the DropletUtils Bioconductor package, and submitted a PR to add it. Likewise, packages of wrapper scripts can be provided through the same mechanism, as described in the following example.

1.6.1 Example: r-seurat-scripts

Create and Clone a fork of bioconda-recipes, and from within that clone create a new branch named for your new package (e.g. r-seurat-scripts):

```
git checkout -b <package name>
```

You will eventually submit a request for this branch to be merged, once your development is complete.

Then create a directory for a new recipe:

```
mkdir -p recipes/<package name>
```

Within that new directory create your meta.yaml and build.sh. For r-seurat-scripts meta.yaml looks like:

```
{% set version = "0.0.3" %}

package:
  name: r-seurat-scripts
  version: {{ version }}

source:
  url: https://github.com/ebi-gene-expression-group/r-seurat-scripts/archive/v{{␣
↪version }}.tar.gz
  sha256: f2dce203cdbec2d1c3e13209925c40318346e9aaa79379901a1017472b841c00

build:
  number: 0
  skip: true  # [win32]
  noarch: generic

requirements:
    host:
        - r-base 3.4.1
    run:
        - r-base 3.4.1
        - r-seurat 2.3.1
        - r-optparse
        - r-workflowscriptscommon

test:
    commands:
        - seurat-read-10x.R --help
        - seurat-create-seurat-object.R --help
```

(continues on next page)

(continued from previous page)

```
        - seurat-normalise-data.R --help
        - seurat-filter-cells.R --help
        - seurat-find-variable-genes.R --help
        - seurat-run-pca.R --help
        - seurat-scale-data.R --help
        - seurat-dim-plot.R --help
        - seurat-find-clusters.R --help
        - seurat-run-tsne.R --help
        - seurat-find-markers.R --help
        - which seurat-get-random-genes.R
        - which r-seurat-scripts-post-install-tests.sh

about:
  home: https://github.com/ebi-gene-expression-group/r-seurat-scripts
  dev_url: https://github.com/ebi-gene-expression-group/r-seurat-scripts
  license: GPL-3
  summary: A set of wrappers for individual components of the Seurat package.
      Functions R packages are hard to call when building workflows outside of R,
      so this package adds a set of simple wrappers with robust argument parsing.
      Intermediate steps are currently mainly serialized R objects, but the
      ultimate objective is to have language-agnostic intermediate formats allowing
      composite workflows using a variety of software packages.
  license_family: GPL
extra:
  recipe-maintainers:
    - MathiasHaudgaard
    - FrodePedersen
    - ArneKr
    - johanneskoester
    - bgruening
    - daler
    - jdblischak
```

Note that:

- The download package is specified under 'source'. It links to a static version with a checksum (see the bioconda recipe guidelines). There will need to be a tagged release on the source repository to make this possible.
- Requirements define the dependencies you need. You'll need to specify the package you're wrapping here, as well as dependencies such as r-workflowscriptscommon and r-optparse if you're writing R wrappers.
- The build section as used here (with 'generic') identifies the code as non-platform-specific.
- There are dependencies 'pinned' to specific R versions. This is often a sensible approach to make sure your package continues to work after the default versions of your dependencies are updated.
- The test sections runs commands to check the installation has worked. Using –help makes sure that e.g. optparse has loaded correctly, as well as the scripts being installed. DO NOT run any further testing here- Bioconda is only interested in whether the install works, downstream testing will be done by the user post-install.

r-seurat-script's build.sh script is very simple:

```
#!/usr/bin/env bash

mkdir -p $PREFIX/bin
cp *.R $PREFIX/bin
cp *.sh $PREFIX/bin
cp *.bats $PREFIX/bin
```

This simply copies the scripts to conda's build directory.

With these two files in place you can do a test local install of your Bioconda recipe. Make sure you're in the directory for your recipe and then:

```
conda build .
conda install --force --use-local r-seurat-scripts
```

If you've done things correctly this will clone your package repository and install the scripts.

1.6.2 Submitting to Bioconda

Before submitting to Bioconda you will need to test the recipe in as close a manner as possible to how Bioconda does, in order to prevent wasting their continuous integration resources with buggy recipes. To do so, follow the bioconda contribution guidelines. Ideally, the CircleCI or mulled-build methods should be used, and will use containers to run the tests. This isn't always easy to get working, however, so at a minimum use the non-docker Conda method cited in the documentation:

```
./bootstrap.py --no-docker /tmp/miniconda
source ~/.config/bioconda/activate
bioconda-utils build recipes config.yml --git-range master
```

Assuming the tests complete successfully, you can follow the instructions to submit a pull request, request review etc. With that process complete, your recipe will become a pakage availabe for installation via Conda.

1.7 Provide high-level workflow components

Once a set of functions is available via a Bioconda package, users can easily download and use a tool or script using Conda in whatever workflow environment they choose. We can then add some further helpful layers to facilitate inclusion of the new functionality in workflows.

1.7.1 Workflow component types

Galaxy wrappers

Galaxy is a popular tool for writing and running workflows. Tools are given Galaxy wrappers which then provide a web-based user interface for that tool with outputs of defined types which can be passed to other tools.

Writing Galaxy wrappers

Galaxy wrappers are just XML files, but writing them well can be tricky. The tool we recommend to facilitate Galaxy wrapper generation is Planemo, and we introduce basic instructions for its operation here. For more information on the following refer to https://planemo.readthedocs.io/en/latest/readme.html.

Prerequisites

pip and virtualenv

Before installing Planemo, install the latest version of pip 7.0 or greater and virtualenv-1.9 or greater. To install virtualenv globally with pip (if you have pip 1.3 or greater installed globally):

```
[sudo] pip install virtualenv
```

Refer to user documents for further information https://virtualenv.pypa.io/en/stable/installation/

Planemo

Setting up virtual environment Installation of Planemo

```
virtualenv .venv
. .venv/bin/activate
pip install "pip>=7" # Upgrade pip if needed.
pip install planemo
```

Install problems (probably temporary)

We are currently experiencing blank white screens with Galaxy installs, due to the client parts of Galaxy not being installed correctly. This occurs with the Galaxy installed by Planemo. To address the problem you need to install Galaxy separately and make a couple of tweaks:

- Download the latest Galaxy in a separate directory and do the the 'run.sh' step.
- You may see an error like "Cannot uninstall 'certifi'. It is a distutils installed project and thus we cannot accurately determine which files belong to it which would lead to only a partial uninstall." In this case update the certifi entries to certifi==2018.10.15 in lib/galaxy/dependencies/dev-requirements.txt and requirements.txt and try again.
- You may well see install problems with nodejs. In this case you need to activate the conda environment used by Galaxy and downgrade nodejs to a 9* version (the latest and default version installed is 10*)

Provided the Galaxy install gets most of the way through such that the client is in place, you can then point at this Galaxy install from Planemo with commands like the following (see below):

```
planemo serve --galaxy_root ../galaxy
```

Installing bioconda packages

Developing galaxy tools wrappers for a particular bioconda recipe requires installation of the package in the same environment.

For example, installing bioconda scater scripts package can be done like:

```
conda install bioconductor-scater-scripts
```

Writing wrapper XML

For full instructions on this refer to Planemo's documentation itself at https://planemo.readthedocs.io/en/latest/writing_appliance.html. Here we will summarise usage for our use case.

Galaxy wrappers are essentially XML files providing a set of instructions for input and output files and their format, which can then be passed to a command for execution. The Planemo command `tool_init` takes various arguments and generates the skeletal structure of these XML files. Although XML can be written in a text editor Planemo commands makes the process quicker.

For example, we generated a wrapper for scater-read-10x-results.R (this script is now obsolte due to changes in Scater, but the example still serves). This script reads 10X data, creates a SingleCellExperiemnt object from 10X format data, by calling `read10xResults()` from Scater, and saves it in a serialized R object.

Before executing the planamo tools, input test data used in the function must be located within the same environment. Create a folder to store the test data and execute planmo tools within that folder.

Test data:

- barcodes.tsv
- genes.tsv
- matrix.mtx

```
planemo tool_init   --force \
    --macros \
    --id 'scater-read-10x-results' \
    --description 'Loads 10x data into a serialized scater R object' \
    --name 'Scater read 10x data' \
    --requirement bioconductor-scater-scripts@0.0.3 \
    --example_command 'scater-read-10x-results.R -d DATA-DIR -o OUTPUT-OBJECT-FILE' \
    --example_input matrix.txt \
    --example_input genes.tsv \
    --example_input barcodes.tsv \
    --example_output R_scater_serialized.rds \
    --test_case \
    --cite_url 'https://github.com/ebi-gene-expression-group/bioconductor-scater-
↪scripts' \
    --help_from_command 'scater-read-10x-results.R -h'
```

The optional flags are discussed in depth in https://planemo.readthedocs.io/en/latest/writing_appliance.html. But the two most basic ones are `--id` and `--name` which indicate the identifier used by galaxy and a short description of the tool, respectively. Executing this Planemo command will generate `scater-read-10x-results.xml`, `macros.xml` and folder `test-data` and copy of tests data within that folder.

```
<tool id="scater-read-10x-results" name="Scater read 10x data" version="@TOOL_
↪VERSION@+galaxy0">
<description>Loads 10x data into a serialized scater R object</description>
<macros>
    <import>scater_macros.xml</import>
</macros>
<expand macro="requirements" />
<command detect_errors="exit_code"><![CDATA[
    ln -s '$matrix' matrix.mtx &&
    ln -s '$genes' genes.tsv &&
    ln -s '$barcodes' barcodes.tsv &&

    scater-read-10x-results.R -d ./ -o '$R_scater_serialized'
]]></command>
<inputs>
    <param type="data" name="matrix" format="txt" label="Expression quantification␣
↪matrix in sparse matrix format (.mtx)"/>
    <param type="data" name="genes" format="tabular" label="Gene table"/>
    <param type="data" name="barcodes" format="tabular" label="Barcode/Cell table"/>
</inputs>
<outputs>
    <data name="R_scater_serialized" format="rdata" label="${tool.name} on ${on_
↪string}: ${output_format}"/>
```

(continues on next page)

(continued from previous page)

```
</outputs>
<tests>
    <test>
        <param name="matrix" value="matrix.mtx"/>
        <param name="genes" value="genes.tsv"/>
        <param name="barcodes" value="barcodes.tsv"/>
        <output name="R_scater_serialized" file="R_scater_serialized.rds" ftype="rdata
↪" compare="sim_size"/>
    </test>
</tests>
<help><![CDATA[

scater-read-10x-results.R

This is a galaxy interface to scater function read10XResults()

For more information check https://www.bioconductor.org/packages/release/bioc/html/
↪scater.html

    ]]></help>
    <expand macro="citations" />
</tool>
```

Note:

- `Scater-read-10x-results.xml` will have have generic input and input variable names, renamed here for clarity
- The format of rds was renamed to `rdata` as it widely accepted within galaxy community
- Symlinks were created to point input variable names
- It's important to use the appropriate version string in the `tool id` version section. - Running the above command will by default use the version 0.1.0, this needs to be amended to reflect the actual version of the underlying software. - In the Scater script example above (and other components we have built), the wrapper is a thin syntax layer around the tool itself (e.g. Scater), and so the version should reflect that of the tool itself, not that of the wrapper. In this case we use the version of the Scater bioconductor package. - The preferred version format is `"wrapped.software.version+galaxy.wrapper.version"`, for example `"0.0.3+galaxy0"`. When multiple xml file wrap around the same software and therefore share the same software version, it can be replaced by a token that is defined in macros.xml, for example `"@TOOL_VERSION@"` and the version of each wrapper looks like `"@TOOL_VERSION@+galaxy0"`. The optional help section in `"CDATA[...]"` describing the options flag function that is associated with input data needs to be moved to `"<input> <param .../> </input>"` section for clarity in galaxy optional usage.

Macros

macros.xml will help reduce the redundant information in the galaxy wrappers which are repeated. For instance, the version of R used or bioconductor packages and a reference to citation and url to github repository.

The optional flag `--macros` to Planemo will produce two xml files in current directory. Although it will be named macros.xml by default it's renamed here to scater_macros.xml and uses the revised name pointing to the same name in Scater-read-10x-results.xml.

Here is the xml block in scater-read-10x-results.xml

```
<macros>
    <import>scater_macros.xml</import>
</macros>
```

Here is scater_macros.xml

```
<macros>
    <token name="@TOOL_VERSION@">1.6.0</token>
    <xml name="requirements">
        <requirements>
            <requirement type="package" version="0.0.3">bioconductor-scater-scripts</
↪requirement>
            <yield/>
        </requirements>
    </xml>
    <xml name="version">
        <version_command><![CDATA[
            echo $(R --version | grep version | grep -v GNU)", scater version" $(R --
↪vanilla --slave -e "library(scater); cat(sessionInfo()\$otherPkgs\$scater\$Version)
↪" 2> /dev/null | grep -v -i "WARNING: ")
            ]]></version_command>
    </xml>
    <xml name="citations">
        <citations>
            <citation type="bibtex">
                @misc{githubbioconductor-scater-scripts,
                    author = {LastTODO, FirstTODO},
                    year = {TODO},
                    title = {bioconductor-scater-scripts},
                    publisher = {GitHub},
                    journal = {GitHub repository},
                    url = {https://github.com/ebi-gene-expression-group/bioconductor-
↪scater-scripts},
            }</citation>
         <yield />
         </citations>
    </xml>
</macros>
```

More information on galaxy wrapper xml schema can be found at https://docs.galaxyproject.org/en/latest/dev/schema.html and best practices for development can be found at https://galaxy-iuc-standards.readthedocs.io/en/latest/best_practices.html.

Linting

In order to validate or sanity check the generated XML, Planemo provides the `lint` command to review the tool. The output will look something like this:

```
planemo l
Linting tool /galaxy_wrapper/scater/read-10x/scater-read-10x-results.xml
Applying linter tests... CHECK
.. CHECK: 1 test(s) found.
Applying linter output... CHECK
.. INFO: 1 outputs found.
Applying linter inputs... CHECK
.. INFO: Found 3 input parameters.
```

(continues on next page)

(continued from previous page)

```
Applying linter help... CHECK
.. CHECK: Tool contains help section.
.. CHECK: Help contains valid reStructuredText.
Applying linter general... CHECK
.. CHECK: Tool defines a version [0.1.0].
.. CHECK: Tool defines a name [Scater read 10x data].
.. CHECK: Tool defines an id [scater-read-10x-results].
.. CHECK: Tool targets 16.01 Galaxy profile.
Applying linter command... CHECK
.. INFO: Tool contains a command.
Applying linter citations... CHECK
.. CHECK: Found 1 likely valid citations.
```

We can also test for execution of the R wrapper using the command:

This will create symlinks and use any provided test data to execute the tool in Galaxy.

Nextflow workflows

Nextflow is a tool for writing reproducible workflows with minimal syntax and setup, and many of the components we have built are suitable for integration into workflows in this way.

Setup

Nextflow is available on Bioconda, so can be installed simply, like:

```
conda install Nextflow
```

Currently available workflows

The only Nextflow workflow we have made available so far is scanpy-workflow connecting elements of scanpy-scripts.

See the above link for documentation, but briefly this workflow can be executed with a command like:

```
nextflow run -config <your nextflow.config> \
    ebi-gene-expression-group/scanpy-workflow \
    --matrix <mtx zip> --gtf <gtf> \
    --resultsRoot <final results dir>
```

Nextflow will download the workflow automatically before running it. You can of course clone the repository yourselve and make any desired customisations.

Writing Nextflow workflows

You can of course build workflows using any of the script components we have made available (see *Wrapper script packages*), using Nextflow in any manner you choose. However folowing some standards will make your work reusable.

Using basic pipeline-sharing conventions

Following a few basic conventions will make your pipeline installable and re-usable using commands like the above- see the Nextflow documentation to see how this is done. Some key points:

- Put any executables you need to package with the workflow under bin/
- Name the workflow file 'main.nf'
- Include as much default configuration as is necessary to make the workflow run

We strongly suggest using the Conda functionality provided by Nextflow, it makes installing software dependencies (including our components) much simpler. Using static environment files referred to in each process, rather than just listing the software each time, will make it easier to manage versions etc in the long run (see scanpy-workflow).

nf-core

The nf-core repository is emerging as a standard for building and distributing Nexflow workflows. It's requirements are quite stringent in the cause of producing good-quality workflows, but if you're starting from scratch you may find it beneficial to follow those conventions from the start.

1.8 Writing documentation

Keeping documentation up-to-date on a large project such as this is challenging. Where you spot issues or out-of-date content we would love for you to request access to the documentation repository and make changes to documentation via pull requests.

Documentation should be written in reStructuredText for full compatibility with 'Read the Docs' and we encourage you to add and update documentation as you contribute code elements. Please follow the style guide at https://documentation-style-guide-sphinx.readthedocs.io/en/latest/style-guide.html, for example in the syntax used for different heading levels.