Scrimer Documentation

Release 1.1

Libor Morkovsky

October 26, 2015

Contents

Scrimer is a GNU/Linux pipeline for designing PCR and genotyping primers from transcriptomic data.

Scrimer has been published in a peer-reviewed journal, please cite this if you use the pipeline or derive your own pipeline from our work:

Scrimer: designing primers from transcriptome data.
Mořkovský L, Pačes J, Rídl J, Reifová R.
Mol Ecol Resour. 2015 Nov;15(6):1415-20. doi: 10.1111/1755-0998.12403.

Note: We present Scrimer as an *end-to-end solution*, from raw reads to usable primers. This is intended to help novice users, so they can better see the whole picture. However, this has an important downside - many steps in the pipeline are quite complex on their own (contig assembly, read mapping, variant calling etc.), and appropriate attention should be paid to checking the intermediate results.

In general, the most common solution for each given step is automatically chosen, using some reasonable default settings, but also giving the user the option to choose another program - using standard formats for input and output. The fine tuning of each step depending on the input data is up to the users.

CHAPTER 1

Installation

Scrimer is a set of Python and Bash scripts that serve as a glue for several external programs. The Python code is in the `scrimer` package, while Bash commands to run the Python scripts and the external programs can be found in this documentation.

You can install Scrimer either to your own GNU/Linux machine, or use a prebuilt VirtualBox image:

1.1 Scrimer installation

You need a default installation of **Python 2.7** [1] with **virtualenv** [2].

```
# create and activate new python virtual environment for scrimer
# in home directory of current user
virtualenv ~/scrimer-env
. ~/scrimer-env/bin/activate

# install cython in advance because of pybedtools
# and distribute because of pyvcf
pip install cython distribute

# now install scrimer from pypi
# with it's additional dependencies (pyvcf, pysam, pybedtools)
pip install scrimer
```

Scrimer depends on several python modules, that should be installed automatically using the above procedrue.

- **pysam** [3] is used to manipulate the indexed fasta and bam files
- **pybedtools** [4] is used to read and write the annotations
- **PyVCF** [5] is used to access variants data

1.1.1 Special cases

If you're in an environment where you're not able to install virtualenv systemwide, we recommend using the technique described at http://eli.thegreenplace.net/2013/04/20/bootstrapping-virtualenv/.

[1] Python http://www.python.org/
[2] virtualenv http://www.virtualenv.org/en/latest/
[3] pysam http://code.google.com/p/pysam/
[4] pybedtools http://pythonhosted.org/pybedtools/
[5] PyVCF https://github.com/jamescasbon/PyVCF

If you're in a grid environment, this can help with paths that differ on different nodes

```
virtualnev --relocatable ~/scrimer-env
```

1.1.2 Non-python dependecies

Apart from the Python modules, the Scrimer pipeline relies on other tools that should be installed in your PATH. Follow the installation instructions in each package.

For reference we recorded the `commands used to install those dependencies` in the scrimer virtual box image. If your system is Debian 7, the commands could work verbatim.

- **bedtools** [6] is a dependency of pybedtools, used for manipulating with gff and bed files
- **samtools** [7] is used for manipulating short read alignments, and for calling variants
- **LASTZ** [8] is used to find the longest isotigs
- **tabix** [9] creates compressed and indexed verisions of annotation files
- **GMAP** [10] produces a spliced mapping of your contigs to the reference genome
- **smalt** [11] maps short reads to consensus contigs to discover variants
- **GNU parallel** [12] is used throughout the pipeline to speed up some lengthy calculations [13]
- **blat** and **isPcr** [14] are used to check the designed primers
- **Primer3** [15] is used to find the most optimal primes sequences
- **cutadapt** [16] is used to remove cDNA synthesis primers.

Additional tools can be installed to provide some more options.

- **FastQC** [17] can be used to check the tag cleaning process
- **agrep** [18] and **tre-agrep** [19] can be used to check the tag cleaning process
- **sort-alt** [20] provides alphanumeric sorting of chromosome names, rename `sort` to `sort-alt` after compiling
- **IGV** [21] is great for visualizing the data when checking the results
- **newbler** [22] is the best option for assembling 454 mRNA data [23] [24]

[6] bedtools https://github.com/arq5x/bedtools2
[7] samtools http://www.htslib.org/, http://sourceforge.net/projects/samtools/files/
[8] lastz http://www.bx.psu.edu/~rsharris/lastz/
[9] tabix http://www.htslib.org/, http://samtools.sourceforge.net/tabix.shtml
[10] gmap http://research-pub.gene.com/gmap/
[11] smalt http://www.sanger.ac.uk/resources/software/smalt/, we used 0.7.0.1, because the latest version (0.7.3) crashes
[12] GNU parallel http://www.gnu.org/software/parallel/
[13] Tange,O. (2011) GNU Parallel - The Command-Line Power Tool. ;login: The USENIX Magazine, 36, 42-47.
[14] http://users.soe.ucsc.edu/~kent/src/, get `blatSrc35.zip` and `isPcr33.zip`, before `make` do `export MACHTYPE` and `export BINDIR=<dir>`
[15] http://primer3.sourceforge.net/
[16] https://code.google.com/p/cutadapt/
[17] FastQC http://www.bioinformatics.babraham.ac.uk/projects/fastqc/
[18] agrep https://github.com/Wikinaut/agrep
[19] tre-agrep http://laurikari.net/tre/
[20] sort-alt https://github.com/lh3/foreign/tree/master/sort
[21] IGV http://www.broadinstitute.org/igv/
[22] newbler http://454.com/products/analysis-software/index.asp
[23] Mundry,M. et al. (2012) Evaluating Characteristics of De Novo Assembly Software on 454 Transcriptome Data: A Simulation Approach. PLoS ONE, 7, e31410. DOI: http://dx.doi.org/10.1371/journal.pone.0031410
[24] Kumar,S. and Blaxter,M.L. (2010) Comparing de novo assemblers for 454 transcriptome data. BMC Genomics, 11, 571. DOI: http://dx.doi.org/10.1186/1471-2164-11-571

- **MIRA** [25] does well with 454 transcriptome assembly as well [32] [33]
- **sim4db** [26] can be used as alternative spliced mapper, part of the kmer suite, apply our patch [27] to get standard conformant output
- **Pipe Viewer** [28] can be used to display the progress of longer operations
- **BioPython** [29] and **NumPy** [30] are required for running `5prime_stats.py`
- **mawk** [31], awk is often used in the pipeline, and mawk is usually an order of magnitude faster
- **vcflib** [32] has a nice interface for working with vcf files (but new bcftools are good as well)

1.1.3 Add installed tools to your PATH

To easily manage locations of the tools that you're using with the Scrimer pipeline, create a text file containing paths to directories, where binaries of your tools are located. The format is one path per line, for example:

```
/opt/bedtools/bin
/opt/samtools-0.1.18
/opt/lastz/bin
/opt/tabix
/opt/gmap/bin
/data/samba/liborm/sw_testbed/smalt-0.7.4
/data/samba/liborm/sw_testbed/FastQC
/data/samba/liborm/sw_testbed/kmer/sim4db
```

Put this file to your virtual environment directory, e.g. `~/scrimer-env/paths`. You can run the following snippet when starting your work session:

```
export PATH=$( cat ~/scrimer-env/paths | tr "\n" ":" ):$PATH
```

1.1.4 References

Python packages

Other software

Optional software

Papers

1.2 Scrimer virtual machine

The virtual machine is the easiest way to test Scrimer. You get a computer preinstalled with Scrimer, many of its dependencies and a test data set. The test data set is a complete scrimer run with all the intermediate files generated from a subset of reads from our nightingale data set, using zebra finch as a reference genome.

[25] MIRA http://www.chevreux.org/projects_mira.html
[26] sim4db http://sourceforge.net/apps/mediawiki/kmer/index.php?title=Main_Page
[27] patch for sim4db gff output http://sourceforge.net/p/kmer/patches/2/
[28] Pipe Viewer http://www.ivarch.com/programs/pv.shtml
[29] BioPython http://biopython.org/
[30] numpy http://www.numpy.org/
[31] mawk http://invisible-island.net/mawk/
[32] vcflib https://github.com/ekg/vcflib

The machine is set up tu use 2 GB of RAM and one CPU. Those settings can be adjusted in the main VirtualBox interface. We decieded to use 32 bit machine, because it is the most compatible setting. The downside is that you cannot use more than 4 GB of RAM. We can provide 64 bit image on request.

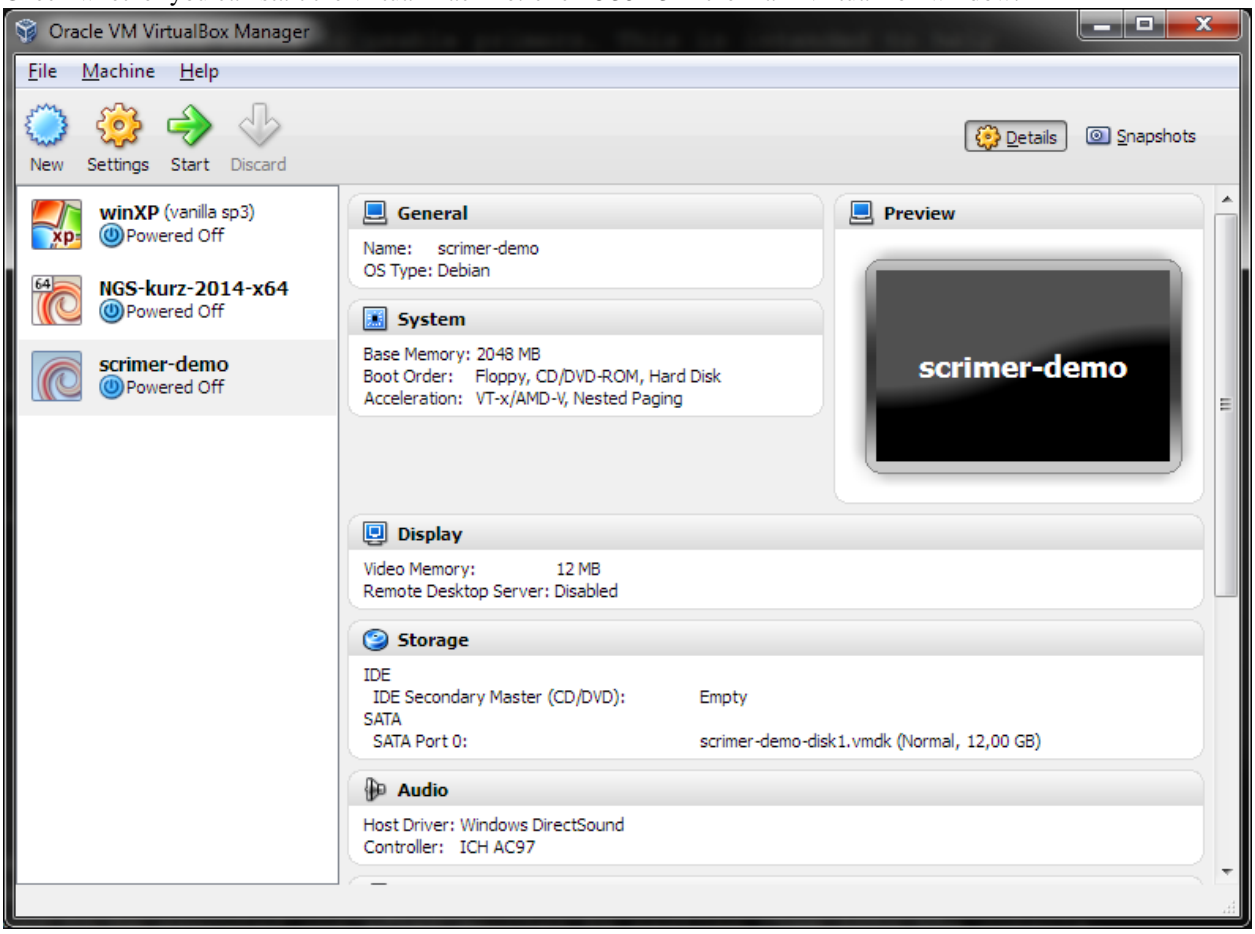
Because of licensing issues, the image does not contain Newbler. Trinity is not included as well, because of high memory requirements of the assembly. We recommend to perform the assembly on a dedicated machine and then transfer the data back to the scrimer project directory.

1.2.1 Virtual machine installation

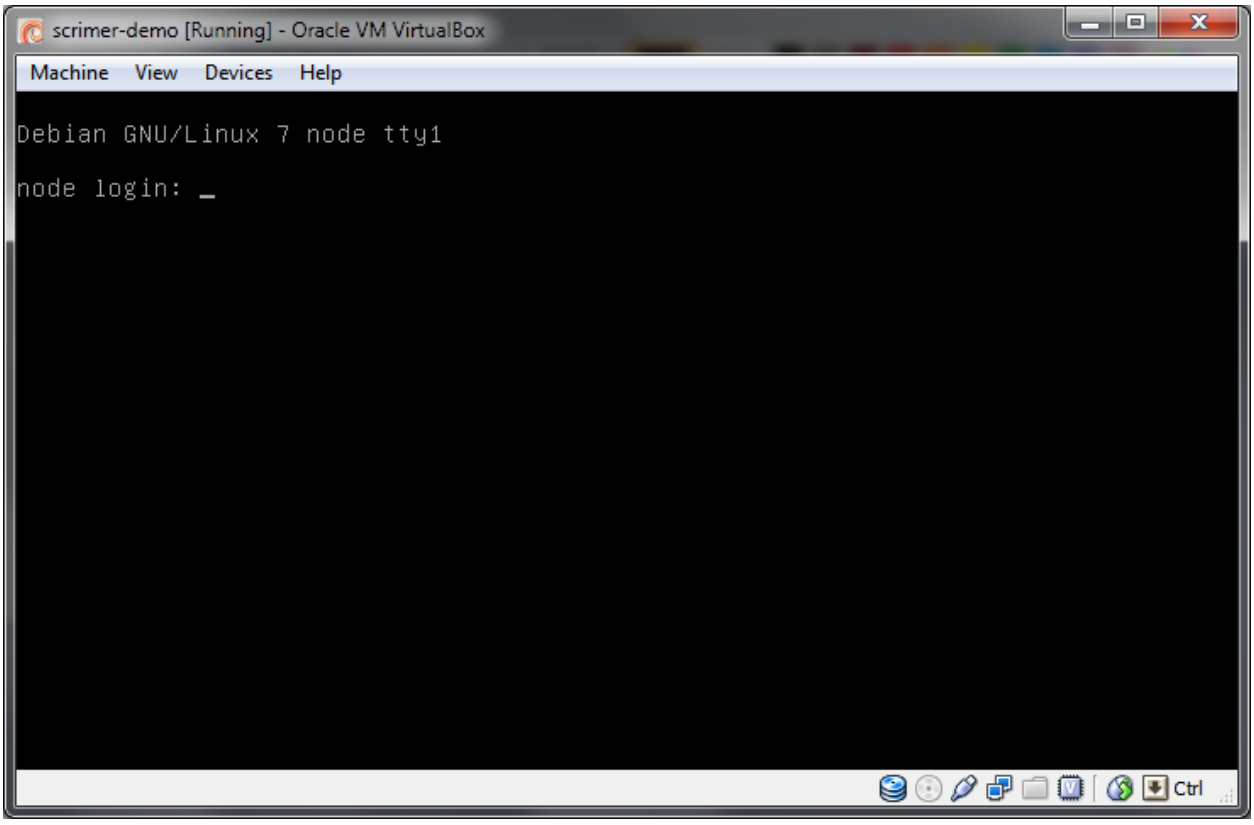
Installation steps (it should take less than 10 minutes):

- Install VirtualBox (https://www.virtualbox.org/wiki/Downloads). It works on Windows, Linux and Mac.
- Download the virtual machine image from http://goo.gl/Xf2cVU. You'll get a single file with `.ova` extension on your hard drive.
- You can either double click the `.ova` file, or run VirtualBox, and choose `File > Import Appliance`. Follow the instructions after the import is started.

After a successful instalation you should see something like this (only the machine list will contain jsut one machine). Check whether you can start the virtual machine: click `Start` in the main VirtualBox window:



After a while you should see something like this:



You don't need to type anything into that window, just checking that it looks like the screenshot is enough.

Machine configuration details:

- Administrative user: *root*, password: *debian*
- Normal user: *user*, password: *user*
- ssh on port 2222

Windows

Install PuTTY and WinSCP. PuTTY will be used to control the virtual computer. WinSCP will be used to transfer files between your computer and the virtual computer.

- PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html - look for putty.exe)
- WinSCP (http://winscp.net/eng/download.php - look for Installation package).

Mac OS X and Linux

Ssh is used to control the virtual computer. It should be installed in your computer.

Files can be transferred with `scp`, `rsync` or `lftp` (recommended) from the command line. *Scp* and *rsync* could be already installed in your system, if you want to use *lftp*, you'll probably have to install it yourself.

Mac users that prefer grapical clients can use something like *CyberDuck*. See http://apple.stackexchange.com/questions/25661/whats-a-good-graphical-sftp-utility-for-os-x .

1.2.2 Connecting to the virtual machine

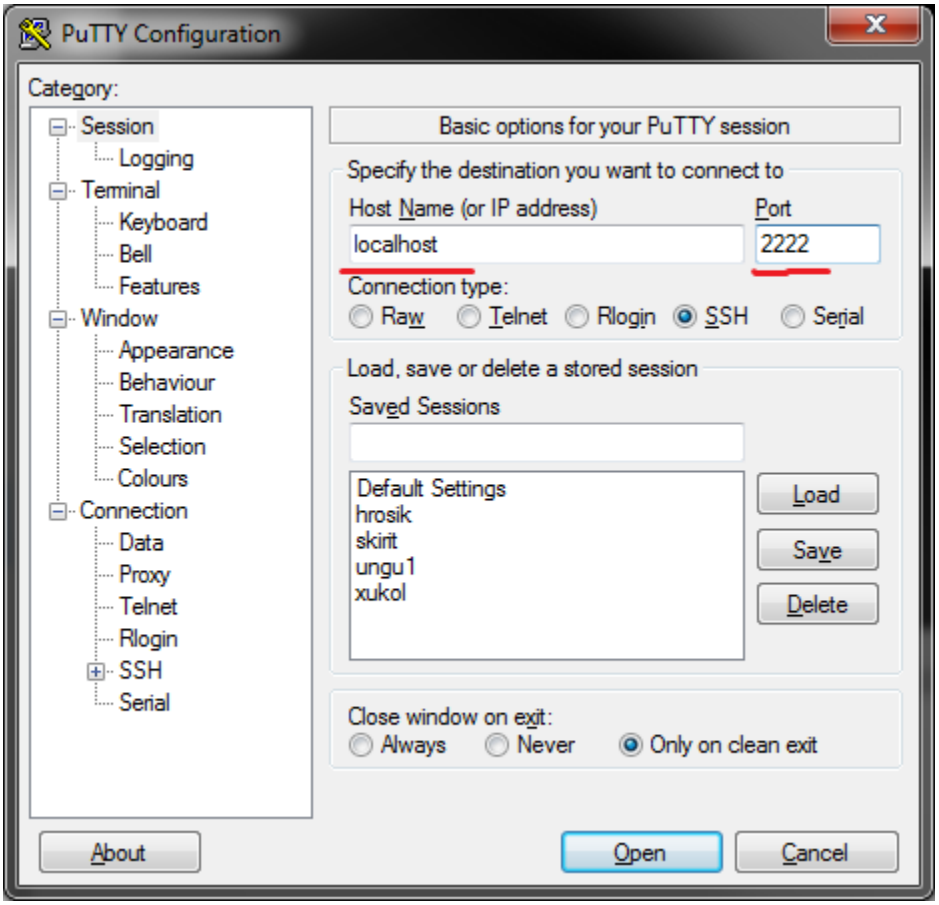
Note: You need to start the virtual machine first!

Connect to control the machine

To control the machine, you need to connect to the ssh service.

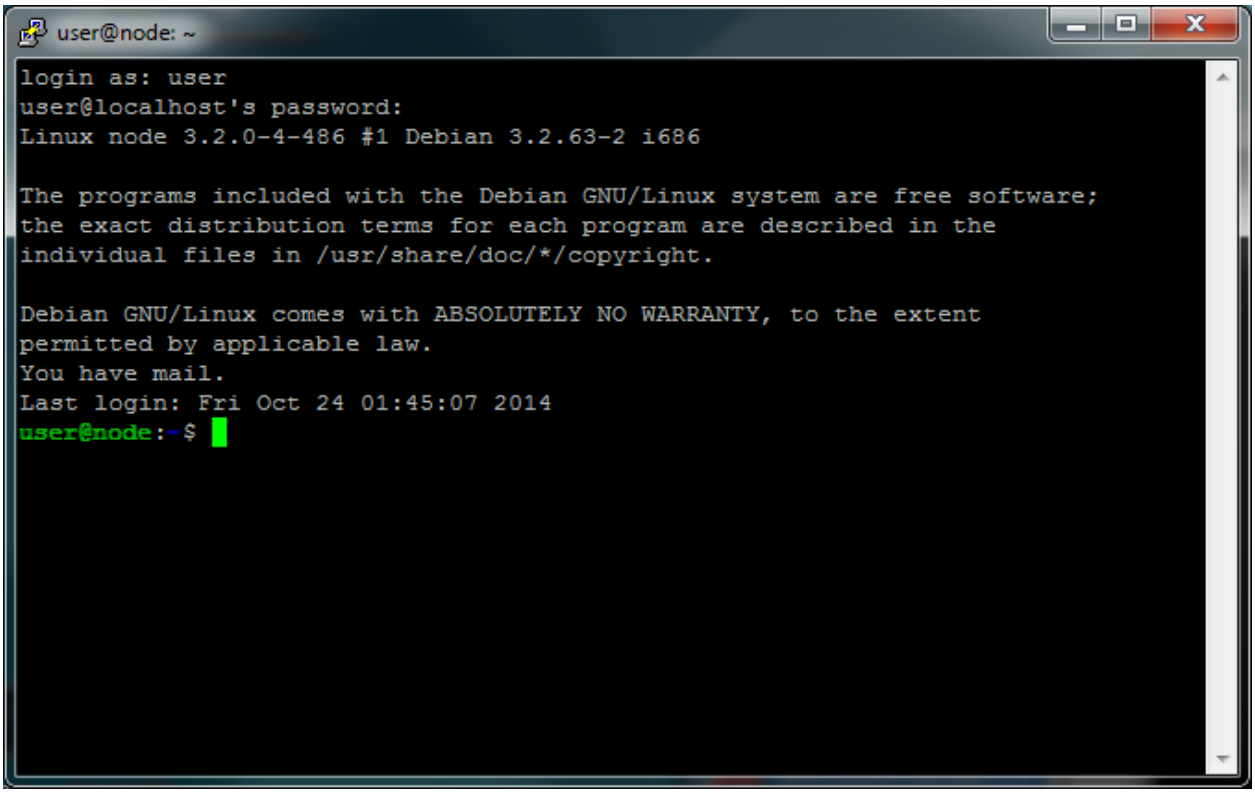
In Windows this is done with PuTTY.

- start PuTTY
- fill Host Name: `localhost`
- fill Port: `2222`
- click Open or press <Enter>



In the black wnidow that appears, type your credentials:

- login as: `user`
- user@localhost's password: `user`

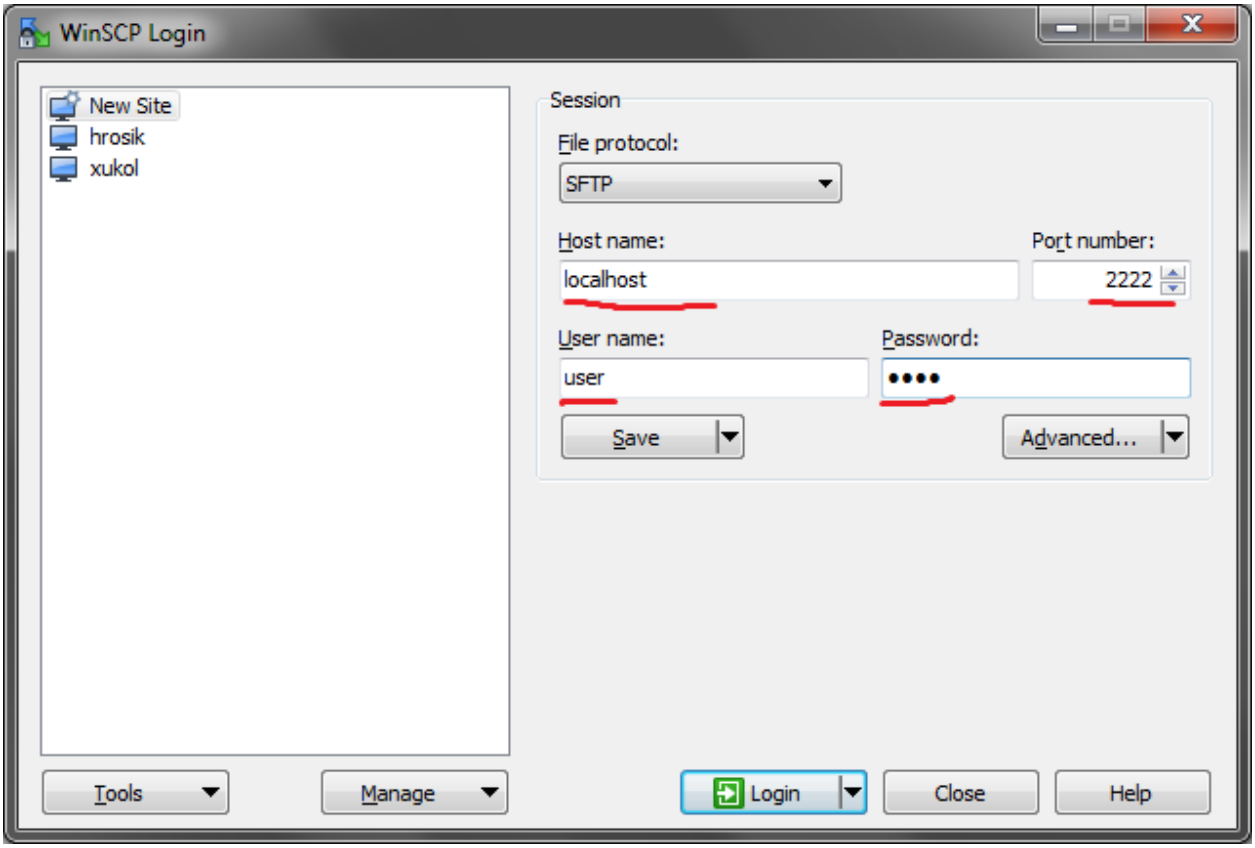


In Mac OS X or Linux, this is done with ssh tool:

```
ssh -p 2222 user@localhost
```

Connect to copy files

In Windows, WinSCP is used to copy files to Linux machines. You use the same information as for PuTTY to log in.



In Mac OS X or Linux, the most simple command to copy a file into a home directory of `user` on a running virtual machine is:

```
scp -P 2222 myfile user@localhost:~
```

Additionally a demo data set is available (it is already included in the VirtualBox image). It contains the whole project tree of one Scrimer run with all the intermediate files and an IGV session file, which can be used to load all the relevant tracks to IGV at once.

CHAPTER 2

Pipeline workflow

The workflow is divided into several steps. The intermediate results of these steps can be visually inspected and checked for validity. There is no reason to continue with the process when a step fails, as further results will not be meaningful. The whole pipeline is a series of pre-made shell commands which are each supposed to be executed one after another by pasting them into the console.

The following pages describe these steps in detail, explain some choices we made and suggest ways how to check validity of the intermediate results.

Under normal circumsances it should take about a day to push your data through the pipeline, if everything goes well.

2.1 Set up project dependent settings

All commands in Scrimer scripts and this manual suppose that you will set some environment variables that define your project and that you add the required tools into your `PATH`.

2.1.1 Directory layout

Here we present the layout that we use to organize all the data needed to run the pipeline. The inputs together with intermediate (and final) results total to hundreds of files. Having those files organized can help prevent mistakes.

Note: The method of organizing your data presented here is just our suggestion. Python scripts doing most of the work are not dependent on any particular directory structure.

Genomes directory

We assume that genome data is shared among different projects and different people on the same machine. Thus we place it in a location that is different from project specific data. This is where the reference genome should be placed.

Project directory

A directory containing files specific for one input dataset. Various steps can be run with various settings in the same project directory. We organize our files in a *waterfall* structure of directories, where each directory name is prefixed with a two digit number. The directory name is some short meaningful description of the step, the first digit in the prefix corresponds to part of the process (read mapping, variant calling etc.), and the second digit distinguishes substeps or runs with different settings.

To start a new project, create a new directory. To use Scrimer you have to convert your data to the `fastq` format. Put your `.fastq` data in a subdirecotry called `00-raw`.

2.1.2 project.sh

Create a file called `project.sh` in your project directory. It will consist of `KEY=VALUE` lines that will define your project specific settings, and each time you want to use Scrimer you'll start by:

```
cd my/project/directory
. project.sh
```

Example `project.sh`:

```
# number of cores you want to use for parallel calculations
CPUS=8

# location of genome data in your system
# you need write access to add a new reference genome to that location
GENOMES=/data/genomes

# reference genome used
GENOME=taeGut1
GENOMEDIR=$GENOMES/$GENOME
GENOMEFA=$GENOMEDIR/$GENOME.fa

# genome in blat format
GENOME2BIT=$GENOMEDIR/$GENOME.2bit

# gmap index location
GMAP_IDX_DIR=$GENOMEDIR
GMAP_IDX=gmap_${GENOME}

# smalt index
SMALT_IDX=$GENOMEDIR/smalt/${GENOME}k13s4

# primers used to synthetize cDNA
# (sequences were found in .pdf report from the company that did the normalization)
PRIMER1=AAGCAGTGGTATCAACGCAGAGTTTTTGTTTTTTTCTTTTTTTTTTNN
PRIMER2=AAGCAGTGGTATCAACGCAGAGTACGCGGG
PRIMER3=AAGCAGTGGTATCAACGCAGAGT
```

2.1.3 Adding the tools to your PATH

For each scrimer session, all the executables that are used have to be in one of the directories mentioned in `PATH`. You can set up your `PATH` easily by using the file you created during *installation*:

```
export PATH=$( cat ~/scrimer-env/paths | tr "\n" ":" ):$PATH
```

Such line can be at the end of your `project.sh` file, so everythig is set up at once.

Alternatively you can copy all the tool executables into your virtual environment bin directory (`~/scrimer-env/bin`).

2.2 Prepare the reference genome

2.2.1 Download and prepare the reference genome

- A list of available genomes is at http://hgdownload.cse.ucsc.edu/downloads.html
- We download the full data set, but it's possible to interrupt the download during xenoMrna (not needed, too big).
- `md5sum` is a basic utility that should be present in your system, otherwise check your packages (yum, apt-get, ...)

```
# location of genome data that can be shared among users
mkdir -p $GENOMEDIR
cd $GENOMEDIR

rsync -avzP rsync://hgdownload.cse.ucsc.edu/goldenPath/$GENOME/bigZips/ .

# check downloaded data integrity
md5sum -c md5sum.txt
cat *.md5 | md5sum -c
```

Now unpack the genome. This process differs for different genomes - some are in single .fa, some are split by chromosomes. Some archives are *tarbombs*, so unpack to directory `chromFa` to avoid a possible mess:

```
mkdir chromFa
tar xvzf chromFa.tar.gz -C chromFa
```

Create concatenated genome, use Heng Li's *sort-alt* to get the common ordering of chromosomes:

```
find chromFa -type f | sort-alt -N | xargs cat > $GENOME.fa
```

2.2.2 Download all needed annotations

Annotation data is best obtained in UCSC table browser in BED format and then sorted and indexed by *BEDtools*

For example: http://genome.ucsc.edu/cgi-bin/hgTables?db=taeGut1:

```
# directory where annotations are stored
ANNOT=annot
sortBed -i $ANNOT/ensGene.bed > $ANNOT/ensGene.sorted.bed
bgzip $ANNOT/ensGene.sorted.bed
tabix -p bed $ANNOT/ensGene.sorted.bed.gz
```

FIXME: rozepsat Or using compressed files:

```
zcat -d $ANNOT/refSeqGenes.bed.gz | sortBed | bgzip > $ANNOT/refSeqGenes.sorted.bed.gz
zcat -d $ANNOT/ensGenes.bed.gz | sortBed | bgzip > $ANNOT/ensGenes.sorted.bed.gz
tabix -p bed $ANNOT/ensGenes.sorted.bed.gz
tabix -p bed $ANNOT/refSeqGenes.sorted.bed.gz
```

2.2.3 Build indexes for all programs used in the pipeline

Some programs need a preprocessed form of the genome, to speed up their operation.

```
# index chromosome positions in the genome file for samtools
samtools faidx $GENOMEFA

# build gmap index for zebra finch
# with some newer versions it is necessary to use -B <path/to/bindir>
# beware, this requires quite a lot of memory (gigabytes)
gmap_build -d $GMAP_IDX -D $GMAP_IDX_DIR $GENOMEFA

# smalt index
# needed only for speeding up sim4db
mkdir -p $GENOMEDIR/smalt
smalt index -s 4 $SMALT_IDX $GENOMEFA

# convert to blat format
faToTwoBit $GENOMEFA $GENOME2BIT
```

2.3 Remove cDNA synthesis adaptors

2.3.1 Quality check of the raw data

Results of the quality check of the raw data can be used as a reference point to check the improvements done by this step.

```
IN=00-raw
OUT=10-fastqc
mkdir $OUT
fastqc --outdir=$OUT --noextract --threads=$CPUS $IN/*.fastq
```

2.3.2 Split the files by multiplex tags

If you used multiplexing during library preparation, your reads have either already been split in your sequencing facility, or you have to split the reads according to the 'barcodes' stored in the sequences yourself.

- for 454, this is done by `sfffile`, you have to convert resulting sff files to fastq format
- for Illumina this can be done e.g. by eautils

```
IN=03-sff
OUT=04-sff-split
mkdir -p $OUT

parallel -j 1 sfffile -s RLMIDs -o $OUT/{/.} {} ::: $IN/*.sff
```

2.3.3 Remove cDNA synthesis primers with cutadapt

Another source of noise in the data is the primers that were used for reverse transcription of mRNA and for the following PCR amplification of cDNA. We remove them using `cutadapt`.

If you have more or less than three primers, you have to change the command by adding/removing `--anywhere=` parts.

```
# data from previous steps
IN=10-mid-split
OUT=12-cutadapt
mkdir $OUT

# cut out the evrogen sequences using GNU parallel and cutadapt
# cutadapt supports only 'N' wildcards, no ambiguity codes
parallel -j $CPUS "cutadapt --anywhere=$PRIMER1 --anywhere=$PRIMER2 --anywhere=$PRIMER3 \
  --error-rate=0.2 --overlap=15 --minimum-length=40 \
  --output=$OUT/{/.}.fastq --rest-file=$OUT/{/.}.rest {}" ::: $IN/*.fastq > $OUT/cutadapt.log
```

2.3.4 Check the results

It is necessary to check the results of adaptor cutting.

First you can check how many of the primers were missed by cutadapt. `agrep` uses a different matching algorithm than cutadapt, so some remaining hits are usually found. `/dev/null` is used as the second input to `agrep` so the filenames are output.

```
NERR=5
parallel agrep -c -$NERR "$PRIMER3" {} /dev/null ::: $OUT/*.fastq|grep -v /dev
```

The next thing to check is the logs produced by `cutadapt`. Results for our data - 454 Titanium data from Smart kit synthesized cDNA:

- ~70% trimmed reads
- ~10% trimmed basepairs
- ~10% too short reads

```
grep -A5 Processed $OUT/cutadapt.log | less
```

The mean length of the removed sequences should be close to length of the adapter (31 in this case):

```
less $OUT/cutadapt.log
```

```
# Lengths of removed sequences (5')
# length  count   expected
# 5       350     264.7
# 6       146     66.2
# ...
# 30      6414    0.0
# 31      63398   0.0
# 32      6656    0.0
# ...
```

The size of the `.rest` files is 1/500 of the `.fastq` (should be 1/250 for `.fasta`)

```
ls -l $OUT
```

The `fastqc` checks should be +- ok.

```
fastqc --outdir=13-fastqc --noextract --threads=8 $OUT/*.fastq
```

2.3.5 Visual debugging

If something in the previous checks looks weird, look directly at the data. Substitute filenames below with the names of your files.

Look where the primers are in the sequence. `tre-agrep` is used to color the output of `agrep`, because `agrep` throughput is ~ 42 MB/s while `tre-agrep` throughput is ~ 2 MB/s.

```
FQFILE=$IN/G3UKN3Q01.fasta
NERR=5
agrep -n -$NERR "$PRIMER3" $FQFILE |tre-agrep -$NERR "$PRIMER3" --color|less -S -R
```

To find out how many differences should be allowed in the pattern matching, we try to find a value of `NERR` where the primer sequence starts to match randomly inside the reads, and not only in at the beginning. Notice the `^` in the first command, marking the start of the read.

```
agrep -c -$NERR "^$PRIMER3" $FQFILE && agrep -c -$NERR "$PRIMER3" $FQFILE

# numbers for tag-cleaned G59B..
# 4 errors: 11971 12767
# 5 errors: 16366 17566
# 6 errors: 17146 23858
# 7 errors: 18041 67844
```

In our sample results, numbers start to diverge for `NERR` > 5, so 5 is a good choice.

2.3.6 Read count statistics

For a single file:

```
# read count statistics
# @ can be in the beginning of quality string, so filter the rows in order

# count of sequences
awk '((NR%4)  == 1)' $FQFILE | wc -l
# or more effective
echo $(( $(wc -l $FQFILE) / 4 ))

# count of sequenced bases
awk '((NR%4)  == 2)' $FQFILE | wc -m
```

For all files in `OUT`:

```
# parallel, IO bound task, so run one process a time
OUT=12-cutadapt
echo "read_count base_count filename"
parallel -j 1 'echo $( gawk "((NR%4)  == 1)" {} | wc -l ) $( gawk "((NR%4)  == 2)" {} | wc -m ) {}'
```

2.4 Assemble reads into contigs

2.4.1 Use newbler to assemble the reads

Here **newbler** is used to assemble the contigs. For 3 GB of read data the assembly took 25 CPU hours and 15 GB RAM.

```
IN=12-cutadapt
OUT=22-newbler
CPUS=19

# run full assembly
runAssembly -o $OUT -cdna -cpu $CPUS -m $IN/*.fastq
```

2.4.2 Remove contigs that are similar to each other

The aim is to get one transcript per locus, preferably the longest one. Otherwise the read mapping process would be faced with many ambiguous locations. We achieve this by:

- doing all-to-all comparison within the isotigs
- grouping isotigs that are similar up to given thresholds of coverage and identity, (disjoint sets forest graph algorithm is used)
- and selecting only the longest contig for each group

```
IN=22-newbler
OUT=$IN
SEQFILE=$IN/454Isotigs.fna
MINCOVERAGE=90
MINIDENTITY=95

# taken from lastz human-chimp example, should be report only highly similar hits
# filter self matches with awk
lastz $SEQFILE[multiple] $SEQFILE \
      --step=10 --seed=match12 --notransition --exact=20 --noytrim \
      --match=1,5 --ambiguous=n \
      --coverage=$MINCOVERAGE --identity=$MINIDENTITY \
      --format=general:name1,size1,start1,name2,size2,start2,strand2,identity,coverage \
      | awk '($1 != $4)' > $SEQFILE.lastz-self

# takes 8 minutes, finds 190K pairs

# find the redundant sequences
tail -n +2 $SEQFILE.lastz-self | find_redundant_sequences.py > $SEQFILE.redundant

# add the short sequences to discard list
grep '>' $SEQFILE | awk '{ sub(/length=/,"",$3); sub(/^>/, "", $1); if($3 - 0 < 300) print $1;}' >> $

# get rid of the redundant ones
seq_filter_by_id.py $SEQFILE.redundant 1 $SEQFILE fasta - $SEQFILE.filtered
```

2.4.3 Check the results

```
# check the input contig length distribution
grep '>' $SEQFILE | awk '{ sub(/length=/,"",$3); print $3}' | histogram.py -b 30

# view how many contigs we got after filtering
grep -c '>' $SEQFILE.filtered
```

2.4.4 Assembling Illumina data

Trinity gives fairly good results for transcriptome assembly from Illumina data. Simple Trinity usage looks like:

```
# as mentioned on the homepage
Trinity --seqType fq --JM 50G --left reads_1.fq  --right reads_2.fq --CPU 6
```

Some more tips on assembling 'perfect' transcripts by Don Gilbert.

2.5 Map contigs to the reference genome

2.5.1 Mapping options

GMAP

```
INFILE=20-jp-contigs/lu_master500_v2.fna.filtered
OUT=30-tg-gmap
mkdir $OUT

OUTFILE=$OUT/$( basename ${INFILE%%.*} ).gmap.gff3

gmap -D $GMAP_IDX_DIR -d $GMAP_IDX -f gff3_gene -B 3 -x 30 -t $CPUS\
    --cross-species $INFILE  > $OUTFILE
```

sim4db

Sim4db is considerably slow, but it has an option to provide a *mapping script*. We exploit this by creating a mapping script by using fast short read aligner with fragments of contigs we want to map, and then convert hits of those short reads into candidate mapping locations.

Please use the patched version sim4db to obtain correct mapping coordinates in the gff files.

```
INFILE=20-newbler/454Isotigs.fna.filtered
OUT=31-tg-sim4db
mkdir $OUT

# these values are derived, it's not necessary to change them
FRAGS=$OUT/${INFILE##*/}.frags
SMALT_OUT=$FRAGS.cigar
SIM4_SCR=${FRAGS%.*}.sim4scr
OUT0=${FRAGS%.*}.tmp.gff3
OUTFILE=${FRAGS%.*}.gff3
```

Use `smalt` as a fast mapper to find all +-50 kBase windows for predicting exon/gene models with sim4db:

```
# create fragments, using slightly modified fasta_fragments.py from lastz distribution
cat $INFILE | fasta_fragments.py --step=80 > $FRAGS

# map the fragments with smalt (takes few minutes), reporting all hits (-d -1) scoring over 60
smalt_x86_64 map -n 8 -f cigar -o $SMALT_OUT -d -1 -m 60 $SMALT_IDX $FRAGS

# construct the script for sim4db
cat $SMALT_OUT | cigar_to_sim4db_scr.py $GENOMEFA.fai | sort --key=5n,5 > $SIM4_SCR
```

Run sim4db using the script. (It takes several seconds for the whole genome.):

```
sim4db -genomic $GENOMEFA -cdna $INFILE -script $SIM4_SCR -output $OUT0 -gff3 -interspecies -mincover

# fix chromosome names
sed s/^[0-9][0-9]*:chr/chr/ $OUT0 > $OUTFILE
```

2.5.2 Transfer genome annotations to our contigs

Annotate our sequences using data from similar sequences in the reference genome. Annotations are transferred in coordinates relative to each of the mapped contigs. The input annotation data have to be sorted and indexed with tabix. Multiple contig mappings and multiple reference annotations can be used.

```
# use multiple mappings like this:
# COORDS="30-tg-gmap/lu_master300_v2.gmap.gff3 31-tg-sim4db/lu_master500_v2.fna.filtered.gff3"
# use multiple annots like this:
# ANNOTS="$GENOMEDIR/annot/ensGene_s.bed.gz $GENOMEDIR/annot/refSeq_s.bed.gz"
COORDS=30-tg-gmap/454Isotigs.gmap.gff3
ANNOTS=$GENOMEDIR/ensGene.sorted.gz
OUT=32-liftover
mkdir -p $OUT

for C in $COORDS
do
    liftover.py "$C" $ANNOTS > $OUT/${C##*/}-lo.gff3
done
```

2.5.3 Create a 'transcript scaffold' using the annotations

Construct a 'transcript scaffold' (contigs joined in their order of appearance on reference genome chromosomes). This is mainly because of viewing convenience with IGV. 'N' gaps should be larger than the longest read size to avoid mapping of the reads across gaps:

```
# filtered contigs
INFILE=20-newbler/454Isotigs.fna.filtered
# transferred annotations from previous step
ANNOTS=32-liftover/*-lo.gff3
# output directory
OUT=33-scaffold
# name of the output 'genome'
GNAME=sc-demo

mkdir $OUT
OUTGFF=$OUT/$GNAME.gff3

scaffold.py $INFILE $ANNOTS $OUT/$GNAME.fasta $OUTGFF

# index the new genome
samtools faidx $OUT/$GNAME.fasta

# sort, compress and index the merged annotations
# so they can be used further down in the pipeline
OUTFILE=${OUTGFF%.*}.sorted.gff3

sortBed -i $OUTGFF > $OUTFILE
bgzip $OUTFILE
tabix -p gff $OUTFILE.gz
```

The transcript scaffold with the sorted `.sorted.gff3` is the first thing worth loading to IGV.

2.6 Map reads to the scaffold

2.6.1 Map all the reads using smalt

Set up variables:

```
# data from previous steps
SCAFFOLD=33-scaffold/sc-demo.fasta
INFILES=10-cutadapt/*.fastq
OUT=40-map-smalt

SMALT_IDX=${SCAFFOLD%/*}/smalt/${SCAFFOLD##*/}-k13s4
```

Create index for the scaffold and map the reads. Mapping 3 GB of reads (fastq format) takes ~5 hours in 8 threads on Intel Xeon E5620, 0.5 GB memory per each mapping. This step would benefit from parallelization even on multiple machines (not implemented here).

```
# create smalt index
mkdir -p ${SMALT_IDX%/*}
smalt index -s 4 $SMALT_IDX $SCAFFOLD

# map each file, smalt is multithreaded so feed the files sequentially
mkdir -p $OUT
parallel -j 1 "smalt map -n $CPUS -p -f sam -o $OUT/{/.}.sam $SMALT_IDX {}" ::: $INFILES

# Illumina reads can be maped e.g. with bwa
bwa index $SCAFFOLD
parallel -j 1 "bwa mem -t $CPUS $SCAFFOLD {} > $OUT/{/.}.sam" ::: $INFILES
```

Merge mapping output to single file

Create a fasta index for the scaffold:

```
samtools faidx $SCAFFOLD
```

2.6.2 Create readgroups.txt

According to your sample wet lab details, create a `readgroups.txt` file. Because `samtools merge -r` attaches read group to each alignment (line) in the input according to the original filename, the format is ($BASENAME is the fastq file name without suffix, $SAMPLE is your biological sample, ${BASENAME%%.*} is the dna library name, all `tab` separated):

```
@RG ID:$BASENAME     SM:$SAMPLE      LB:${BASENAME%%.*}      PL:LS454 DS:$SPECIES
```

The library name (LB) is important because of `rmdup`, description (DS) is here used to identify the species.

Note: The order of the rows matters for the vcf output, the sample columns order is probably the order of first appearance in the @RG.

The following code generates most of the `readgroups.txt` file, you have to reorder lines and fill the places marked with '??':

```
OUT=40-map-smalt
DIR=10-cutadapt

find $DIR -name '*.fastq' | xargs -n1 basename | sed s/.fastq// | awk '{OFS="\t";lb=$0;sub(/\..*$/,"'

# edit the file (ctrl-o enter ctrl-x saves and exits the editor)
nano $OUT/readgroups.txt

# sort the readgroups according to species
<$OUT/readgroups.txt sort -k6,6 > $OUT/readgroups-s.txt
```

2.6.3 Prepare the sam files

Extract the sequence headers from the first `.sam` file (other files should have identical headers):

```
SAMFILE=$( echo $OUT/*.sam | awk '{print $1;}' )
samtools view -S -t $SCAFFOLD.fai -H $SAMFILE > $OUT/sequences.txt
cat $OUT/sequences.txt $OUT/readgroups-s.txt > $OUT/sam-header.txt
```

`samtools merge` requires sorted alignments, sort them in parallel. This creates `.bam` files in the output directory:

```
parallel -j $CPUS "samtools view -but $SCAFFOLD.fai {} | samtools sort - {.}" ::: $OUT/*.sam
```

2.6.4 Merge

Merge all the alignments. Do not remove duplicates because the duplicate detection algorithm is based on the read properties of genomic DNA ([1], [2]).

```
samtools merge -ru -h $OUT/sam-header.txt - $OUT/*.bam | samtools sort - $OUT/alldup
samtools index $OUT/alldup.bam
```

Check the results

Unmapped read counts.

```
parallel -j $CPUS 'echo $( cut -f2 {}|grep -c "^4$" ) {}' ::: $OUT/*.sam
```

Mapping statistics

```
samtools idxstats $OUT/alldup.bam | awk '{map += $3; unmap += $4;} END {print  unmap/map;}'
```

Coverage sums for IGV

```
igvtools count -z 5 -w 25 -e 250 $OUT/alldup.bam  $OUT/alldup.bam.tdf ${CONTIGS%.*}.genome
```

2.7 Detect and choose variants

2.7.1 Call variants with samtools

Set up input and output for the current step:

[1] http://seqanswers.com/forums/showthread.php?t=6543
[2] http://seqanswers.com/forums/showthread.php?t=5424

```
SCAFFOLD=33-scaffold/sc-demo.fasta
ALIGNS=40-map-smalt/alldup.bam

# outputs
OUT=50-variants
VARIANTS=$OUT/demo-variants
mkdir -p $OUT
```

Run the variant calling in parallel. Takes 3 hours for 15 samples on a single Intel Xeon E5620:

```
vcfutils.pl splitchr $SCAFFOLD.fai | parallel -j $CPUS "samtools mpileup -DSu -L 10000 -f $SCAFFOLD -

# samtools > 0.1.19
vcfutils.pl splitchr $SCAFFOLD.fai | parallel -j $CPUS "samtools mpileup -u -t DP,SP -L 10000 -f $SCA
```

Merge the intermediate results. `vcfutils.pl` is used to generate the correct ordering of parts, as in the `.fai`:

```
vcfutils.pl splitchr $SCAFFOLD.fai | xargs -i echo $OUT/part-{}.bcf | xargs -x bcftools cat > $VARIAN
bcftools index $VARIANTS-raw.bcf

# samtools > 0.1.19
# workaround for https://github.com/samtools/bcftools/issues/134
vcfutils.pl splitchr $SCAFFOLD.fai | parallel "bcftools view -O z $OUT/part-{}.bcf > $OUT/{.}.vcf.gz
bcftools concat -O b $OUT/*.vcf.gz > $VARIANTS-raw.bcf
bcftools index $VARIANTS-raw.bcf
```

Remove the intermediate results, if the merge was ok:

```
vcfutils.pl splitchr $SCAFFOLD.fai | xargs -i echo $OUT/part-{}.bcf | xargs rm
```

2.7.2 Call variants with FreeBayes

This is an alternative to the previous section. FreeBayes uses local realignment around INDELs, so the variant calling for 454 data should be better.

```
SCAFFOLD=33-scaffold/sc-demo.fasta
ALIGNS=40-map-smalt/alldup.bam
OUT=51-var-freebayes

GNAME=$( echo ${SCAFFOLD##*/} | cut -d. -f1 )
mkdir -p $OUT
vcfutils.pl splitchr $SCAFFOLD.fai | parallel -j $CPUS "freebayes -f $SCAFFOLD -r {} $ALIGNS > $OUT/

# join the results
OFILE=$OUT/variants-raw.vcf

# headers
FILE=$( find $OUT -name ${GNAME}-*.vcf | sort | head -1 )
<$FILE egrep '^#' > $OFILE
# the rest in .fai order
vcfutils.pl splitchr $SCAFFOLD.fai | parallel -j 1 "egrep -v '^#' $OUT/${GNAME}-{}.vcf >> $OFILE"

# filter the variants on quality (ignore the warning messages)
<$OFILE bcftools view -i 'QUAL > 20' -O z > $OUT/variants-qual.vcf.gz
tabix -p vcf $OUT/variants-qual.vcf.gz
```

2.7.3 Filter variants

We're interested in two kinds of variant qualities

- all possible variants so they can be avoided in primer design
- high confidence variants that can be used to answer our questions

Filtering strategy:

- use predefined `samtools` filtering
  - indels caused by 454 homopolymer problems generally have low quality scores, so they should be filtered at this stage
- remove uninteresting information (for convenient viewing in IGV)
  - overall low coverage sites (less than 3 reads per sample - averaged, to avoid discarding some otherwise interesting information because of one bad sample)
- select the interesting variants, leave the rest in the file flagged as 'uninteresting'
  - only SNPs
  - at least 3 reads per sample
  - no shared variants between the two species

Samtools filtering

Quite high *strand bias* in RNASeq data can be expected, so don't filter on strand bias (`-1 0`). Use the defaults for other settings of `vcfutils varFilter` command:

- minimum RMS mapping quality for SNPs [10]
- minimum read depth [2]
- maximum read depth [10000000]
- minimum number of alternate bases [2]
- SNP within INT bp around a gap to be filtered [3]
- window size for filtering adjacent gaps [10]
- min P-value for baseQ bias [1e-100]
- min P-value for mapQ bias [0]
- min P-value for end distance bias [0.0001]
- FLOAT min P-value for HWE (plus F<0) [0.0001]

```
bcftools view $VARIANTS-raw.bcf | vcfutils.pl varFilter -1 0 | bgzip > $VARIANTS-filtered.vcf.gz
tabix -p vcf $VARIANTS-filtered.vcf.gz
```

Convenience filtering

The required average depth per sample can be adjusted here. Using `pv` as a progress meter. `pv` can be substituted by `cat`:

```
# filter on average read depth and site quality
VCFINPUT=$VARIANTS-filtered.vcf.gz
VCFOUTPUT=$VARIANTS-filt2.vcf.gz
pv -p $VCFINPUT | bgzip -d | vcf_filter.py --no-filtered - avg-dps --avg-depth-per-sample 5 sq --site

# samtools > 0.1.19 produce conflicting info tags, get rid of it if the above filtering fails
pv -p $VCFINPUT | bgzip -d | sed 's/,Version="3"//' | vcf_filter.py --no-filtered - avg-dps --avg-dep

# freebayes output
zcat $OUT/variants-qual.vcf.gz| vcfutils.pl varFilter -1 0 | vcf_filter.py --no-filtered - avg-dps --
```

Interesting variants

The filtered out variants are kept in the file, only marked as filtered out. This way both the selected and non-selected variants can be checked in IGV. Required minumum depth per sample can be adjusted here:

```
# samtools files
VCFINPUT=$VARIANTS-filt2.vcf.gz
VCFOUTPUT=$VARIANTS-selected.vcf.gz

# freebayes files
VCFINPUT=$OUT/demo-filt1.vcf.gz
VCFOUTPUT=$OUT/demo-selected.vcf.gz

<$VCFINPUT pv -p | zcat | vcf_filter.py - dps --depth-per-sample 3 snp-only contrast-samples --sample
tabix -p vcf $VCFOUTPUT
```

2.7.4 Check the results

Extract calculated variant qualities, so the distribution can be checked (-> common power law distribution, additional peak at 999):

```
zcat $VCFINPUT | grep -v '^#' | cut -f6 > $VCFINPUT.qual
# and visualize externally

# or directly in terminal, using bit.ly data_hacks tools
zcat $VCFINPUT | grep -v '^#' | cut -f6 | histogram.py -b 30
```

Count selected variants:

```
zcat -d $VCFOUTPUT | grep -c PASS
```

Count variants on **chromosome Z**:

```
zcat -d $VCFOUTPUT | grep PASS | grep -c ^chrZ
```

2.8 Design primers

2.8.1 Design primers with Primer3

Set inputs and outputs for this step:

```
VARIANTS=50-variants/demo-variants-selected.vcf.gz
SCAFFOLD=33-scaffold/sc-demo.fasta
ANNOTS=33-scaffold/sc-demo.sorted.gff3.gz

GFFILE=demo-primers.gff3
OUT=60-gff-primers

GFF=$OUT/$GFFILE
PRIMERS=${GFF%.*}.sorted.gff3.gz
mkdir -p $OUT

# for all selected variants design pcr and genotyping primers
# takes about a minute for 1000 selected variants, 5 MB gzipped vcf, 26 MB uncompressed genome, 5 MB
# default values are set for SNaPshot
export PRIMER3_CONFIG=/opt/primer3/primer3_config/
design_primers.py $SCAFFOLD $ANNOTS $VARIANTS > $GFF

# use --primer-pref to set preferred length of genotyping primer
# this is useful for other genotyping methods, like MALDI-TOF
design_primers.py --primer-pref 15 --primer-max 25 $SCAFFOLD $ANNOTS $VARIANTS > $GFF
```

Sort and index the annotation before using it in IGV. For a small set of primers it is not necessary to compress and index the file, IGV can handle raw files as well.

```
sortBed -i $GFF | bgzip > $PRIMERS
tabix -f -p gff $PRIMERS
```

Create a region list for IGV to quickly inspect all the primers.

```
<$GFF awk 'BEGIN{OFS="\t";} /pcr-product/ {match($9, "ID=[^;]+"); print $1, $4, $5, substr($9, RSTART
```

2.8.2 Convert scaffold to the blat format

```
SCAFFOLD2BIT=$OUT/${SCAFFOLD##*/}.2bit
faToTwoBit $SCAFFOLD $SCAFFOLD2BIT
```

2.8.3 Validate primers with blat/isPcr

Recommended parameters for PCR primers in blat [3]: `-tileSize=11`, `-stepSize=5`

Get the primer sequences, in formats for isPcr and blat:

```
PRIMERS_BASE=${PRIMERS%%.*}
extract_primers.py $PRIMERS isPcr > $PRIMERS_BASE.isPcr
extract_primers.py $PRIMERS > $PRIMERS_BASE.fa
```

Check against transcriptome data and the reference genome:

```
# select one of those a time:
TARGET=$SCAFFOLD2BIT
TARGET=$GENOME2BIT

TARGET_TAG=$( basename ${TARGET%%.*} )
isPcr -out=psl $TARGET $PRIMERS_BASE.isPcr $PRIMERS_BASE.isPcr.$TARGET_TAG.psl
blat -minScore=15 -tileSize=11 -stepSize=5 -maxIntron=0 $TARGET $PRIMERS_BASE.fa $PRIMERS_BASE.$TARGE
```

[3] http://genomewiki.ucsc.edu/index.php/Blat-FAQ

Note: TODO: It would be nice to add the annotations found by `isPcr` to the primer gff3 tags (not implemented yet).

2.8.4 Check the results

Count and check all places where `primer3` reported problems:

```
<$GFF grep gt-primer | grep -c 'PROBLEMS='
<$GFF grep gt-primer | grep 'PROBLEMS=' | less -S

# count unique variants with available primer set
<$GFF grep gt-primer|grep -v PROBLEM|egrep -o 'ID=[^;]+'|cut -c-13|sort -u|wc -l
```

Use agrep to find similar sequences in the transcript scaffold, to check if the sensitivity settings of blat are OK. Line wrapping in `fasta` can lead to false negatives, but at least some primers should yield hits:

```
# agrep is quite enough for simple checks on assemblies of this size (30 MB)
SEQ=GCACATTTCATGGTCTCCAA
agrep $SEQ $SCAFFOLD|grep $SEQ
```

Import your primers to any spreadsheet program with some selected information on each primer. Use copy and paste, the file format is `tab` separated values. When there is more than one genotyping primer for one PCR product, the information on the PCR product is repeated.

```
extract_primers.py $PRIMERS table > $PRIMERS_BASE.tsv
```

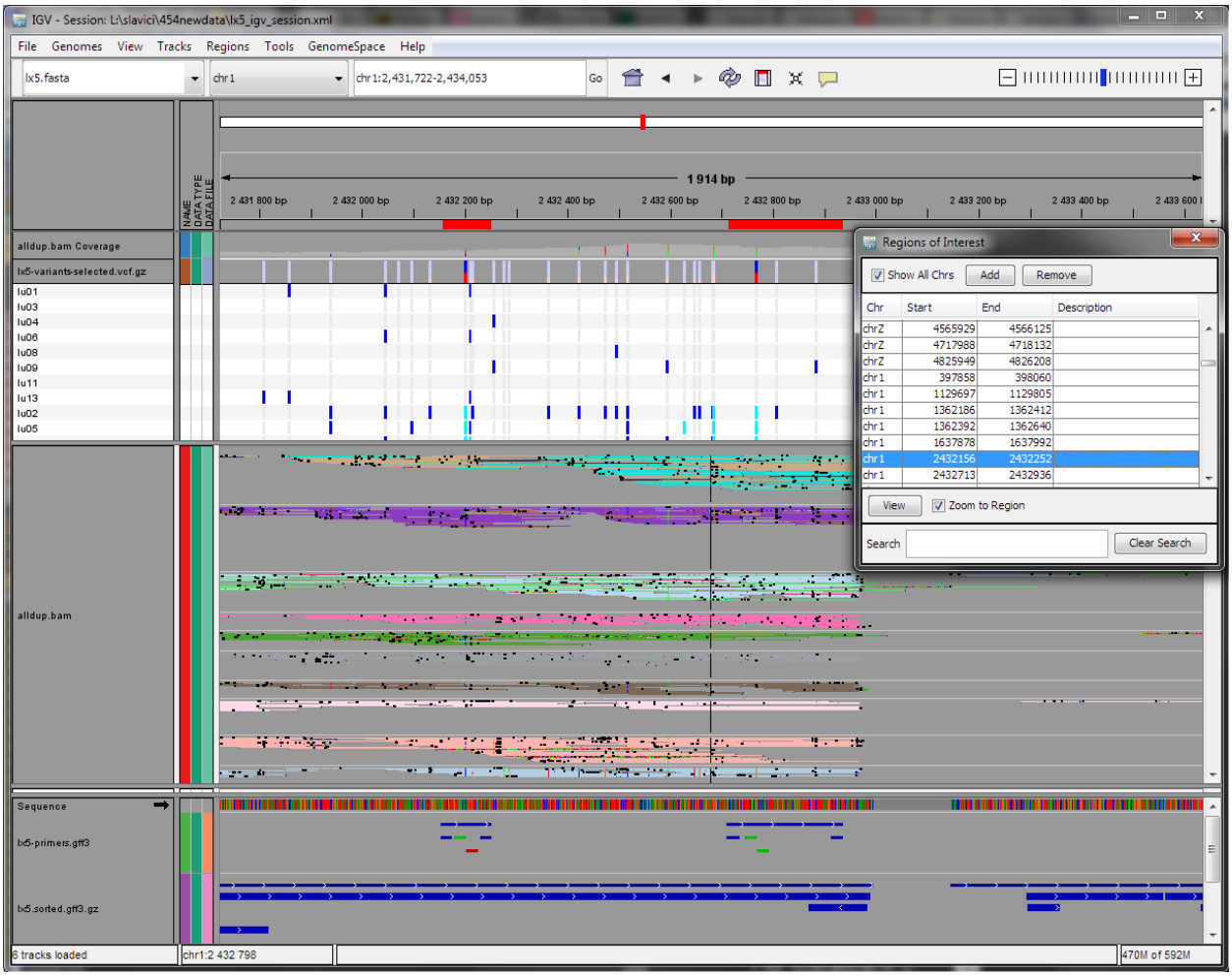
2.9 IGV with all data produced by Scrimer

Tracks, from top to bottom:

- genome navigator, 'genome' produced in Map contigs to the reference genome
- `alldup.bam coverage` - total coverage for the region, produced in Map reads to the scaffold
- `lx5-variants-selected.vcf.gz` - summary of the variants, filtered variants are shown in lighter color, produced in Detect and choose variants
- sample list - provides detailed information on variants in each sample
- `alldup.bam` - details on coverage and SNP (colored) / INDEL (black), in the context menu choose `Group alignments by > sample` and `Color alignments by > read group`, produced in Map reads to the scaffold
- sequence
- `lx5-primers.gff3` - resulting primers, hover with mouse for details on calculated properties, produced in Design primers
- `lx5.sorted.gff3.gz` - annotations for the scaffold - predicted and transferred exons, produced in Map contigs to the reference genome
- floating window with a list of designed primers, produced in Design primers

2.9.1 How to get to this view

- run IGV (version 2.2 is used here)

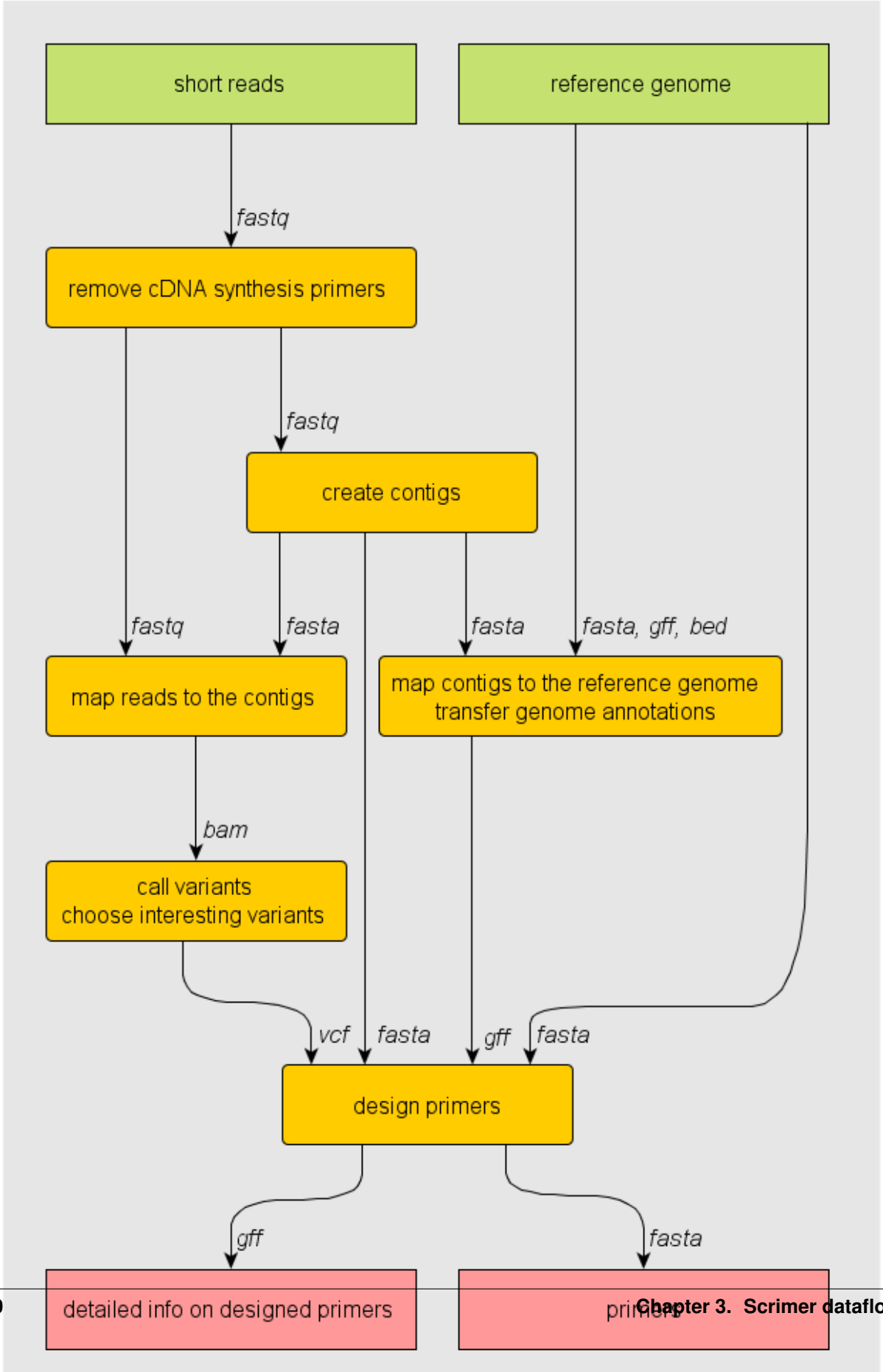

IGV - Session: L:\slavici\454newdata\lx5_igv_session.xml
File Genomes View Tracks Regions Tools GenomeSpace Help
lx5.fasta
chr1
chr1:2,431,722-2,434,053
Go
1 914 bp
2 431 800 bp
2 432 000 bp
2 432 200 bp
2 432 400 bp
2 432 600 bp
2 432 800 bp
2 433 000 bp
2 433 200 bp
2 433 400 bp
NAME
DATA TYPE
DATA FILE
alldup.bam Coverage
lx5-variants-selected.vcf.gz
lu01
lu03
lu04
lu06
lu08
lu09
lu11
lu13
lu02
lu05
alldup.bam
Sequence
lx5-primers.gff3
lx5.sorted.gff3.gz
Regions of Interest
Show All Chrs
Add
Remove
Chr
Start
End
Description
chrZ 4565929 4566125
chrZ 4717988 4718132
chrZ 4825949 4826208
chr1 397858 398060
chr1 1129697 1129805
chr1 1362186 1362412
chr1 1362392 1362640
chr1 1637878 1637992
chr1 2432156 2432252
chr1 2432713 2432936
View
Zoom to Region
Search
Clear Search
6 tracks loaded
chr1:2 432 798
470M of 592M

- `Genomes > Load genome from file`, choose your scaffold
- load all the tracks by `File > Load from File`
- rearrange tracks according to your preference by dragging the label with the mouse
- choose `Regions > Import Regions`, pick the `.bed` file created in Design primers, choose `Regions > Region Navigator`

CHAPTER 3

Scrimer dataflow

Dataflow diagram of the pipeline. Inputs are in green, processing steps in yellow and results in red. Arrows connecting steps are labelled with data format. This image was created using *yEd*.


short reads
reference genome
fastq
remove cDNA synthesis primers
fastq
create contigs
fastq
fasta
fasta
fasta, gff, bed
map reads to the contigs
map contigs to the reference genome
transfer genome annotations
bam
call variants
choose interesting variants
vcf
fasta
gff
fasta
design primers
gff
fasta
detailed info on designed primers
primers

CHAPTER 4

Scrimer components

4.1 Components of the Scrimer pipeline

4.1.1 Components bound to the pipeline

Transfer annotation from the reference genome to the contigs

File: `scripts/liftover.py` Input

- gff file produced by the exon predictor software
- set of bed files containing the features to be transfered to the cDNA

Output

- 'inverted' gff file annotating exons in the transcripts and containing also all the features from the other input files, if they were overlapping with the predicted exons

Author: Libor Morkovsky, 2012

Create a contig scaffold according to contig hits in the reference genome

File: `scripts/scaffold.py` Input

- fasta file with the original sequences
- set of gff files with exon features having the Target attribute (product of `liftover.py`)
- other gff/bed files to remap to the genome

Output

- fasta with the input sequences pasted with 100 Ns
  - unambiguos sequences assigned to 'chromosomes' in the order of the template genome (that was used to generate the Target exon mappings)
  - ambiguos (having more than one possible chromosome) assigned to **chrAmb**
  - unmapped sequences assigned to **chrUnmapped** (to distinguish from chrUn in target genome)
- gff file with the locations of the input sequences (gene) and remapped contents of the input gff files

Algorithm

- go through the input gff files, construct a dictionary {read_name -> {chr -> location}}

- add the lowest coordinate found on a given chromosome (overwriting previous values)
- sort the list with single candidate locations with 'chromocompare'

Author: Libor Morkovsky, 2012

Design primers using sequences, annotations and selected variants

File: `scripts/design_primers.py` Input

- reference sequence (fasta with samtools fai index)
- annotations (gff3, has to contain exon entries)
- filtered variants (vcf, primers are designed for variants with PASS)

Output

- PCR and genotyping primers selected using primer3 (gff3)

Algorithm

- there is only a few selected variants, so the least amount of work will be to do the work only for variants
- for each of the selected variants
  - request exons
  - apply the technical constraints (minimal primer length of 20 from the edge of an exon)
  - patch exon sequence to mark positions of known variants
  - find suitable genotyping primers
  - design PCR primers to flank the (usable) genotyping primers

Author: Libor Morkovsky, 2012, 2014

Export primers from gff3 to the FASTA, tabular or isPcr format

File: `scripts/extract_primers.py` Input

- gff file containing primers
- optional output format

Output

- fa/isPcr file with the primer sequences

Algorithm

- for each line in gff
- **fa output**: if current line has a `SEQUENCE` attribute, output it
- **ispcr output**: only pcr primers are of interest, and in equivalent pairs on encountering gt-xx put it to dict keyed by `Parent` if there are both entries output and remove from dict

Author: Libor Morkovsky, 2012

4.1.2 Tools for FASTA/FASTQ manipulation

Given pairs of matching sequences, create clusters and find the longest representative

File: `scripts/find_redundant_sequences.py` Given pairs of 'almost identical' sequences create clusters of sequences. From each cluster select the longest sequence. Output names of all other sequences (for example to remove them from the data afterwards).

Uses disjoint sets forest to store the clusters so it should scale to millions of sequences.

Input

- custom formated (`--format=general:name1,size1,start1,name2,size2,start2,strand2,identity,co` output from `lastz` on standard input

Output

- names of sequences that were selected as reduntant

Filter sequences in a FASTQ file based on their position in the file

File: `scripts/fastq_kill_lines.py` Input

- FASTQ file
- list of indices (0 based)

Output

- FASTQ file without the given sequences

Author: Libor Morkovsky, 2012

Filter sequences in FASTQ, FASTA based on their identifier

File: `scripts/seq_filter_by_id.py`

Taken from *BioPython*. FASTA and FASTQ readers are pasted in the file, so the program is standalone. Filter a FASTA, FASTQ or SSF file with IDs from a tabular file.

Takes six command line options, tabular filename, ID column numbers (comma separated list using one based counting), input filename, input type (e.g. FASTA or SFF) and two output filenames (for records with and without the given IDs, same format as input sequence file).

If either output filename is just a minus sign, that file is not created. This is intended to allow output for just the matched (or just the non-matched) records.

When filtering an SFF file, any Roche XML manifest in the input file is preserved in both output files.

Note in the default NCBI BLAST+ tabular output, the query sequence ID is in column one, and the ID of the match from the database is in column two. Here sensible values for the column numbers would therefore be "1" or "2".

This tool is a short Python script which requires Biopython 1.54 or later for SFF file support. If you use this tool in scientific work leading to a publication, please cite the Biopython application note:

Cock et al 2009. Biopython: freely available Python tools for computational molecular biology and bioinformatics. Bioinformatics 25(11) 1422-3. http://dx.doi.org/10.1093/bioinformatics/btp163 pmid:19304878.

This script is copyright 2010 by Peter Cock, SCRI, UK. All rights reserved. See accompanying text file for licence details (MIT/BSD style).

This is version 0.0.1 of the script.

Break sequences in FASTA file into fragments

File: `scripts/fasta_fragments.py`

Taken from *lastz*. Break sequences in a fasta file into fragments, so some kind of short read aligner can be used for further processing. Break a fasta file into fragments.

$$$ todo: spread out the fragment starts so that the last fragment ends at the $$$ .. end of a sequence, if possible

$$$ todo: find runs of N and reset the fragment start position to skip past $$$ .. such runs

$$$ liborm(2012): fixed to read only the seq name, not the full line

4.1.3 General purpose tools

Primer3 wrapper

File: `scrimer/primer3_connector.py` A Python interface to the primer3_core executable.

TODO: it is not possible to keep a persistent primer3 process using subprocess module - communicate() terminates the input stream and waits for the process to finish

Author: Libor Morkovsky 2012

class `primer3_connector.`**`BoulderIO`**
Provides Python interface for `BoulderIO` format used by Primer3.

classmethod `deparse`(*records*)
Accepts a dict or a list of dicts, produces a BoulderIO string `(KEY=VAL\n)` with records separated by `'=\n'`.

classmethod `parse`(*string*)
Parse a BoulderIO string `(KEY=VAL\n)` return a list of records, where each record is a dictionary end of the string implies a single `'=\n'` (record separator).

class `primer3_connector.`**`Primer3`**(*p3path='primer3_core'*, ***kwargs*)
Wraps Primer3 executable. *kwargs* are converted to strings and used as default parameters for each call of primer3 binary.

`call`(*records*)
Merge each of the records with *default_params*, the record taking precedence, call the `primer3` binary, parse the output and return a list of dictionaries, `{RIGHT:[], LEFT:[], PAIR:[], INTERNAL:[]}` for each input record uppercase keys (in the result) are the original names from BoulderIO format, lowercase keys have no direct equivalent in primer3 output (`position`, `other-keys`)

Convert CIGAR matches to sim4db 'script'

File: `scripts/cigar_to_sim4db_scr.py` Script that parses `CIGAR` file produced by aligning fragments of contigs to a genome (tested with `smalt` output) and outputs a 'script' for limiting the exon model regions of `sim4db`.

Input

- output of some read mapper in `CIGAR` format
- all the fragments must be reported by the aligner
- the fragment names have to be in the same order as the master sequences
- the fragments must be named like readname_number (like `fasta_fragments.py` does)
- all the hits from one read must follow each other

Output

- sim4db 'script'

Algorithm

- load chromosome definition file
- parse the hits: - extract read name - check if hit is in known chromosome (report error otherwise) - for each hit create +-50 KB region clipped to chromosome ends - when a read name different from the previous one is encountered, merge all the regions - output each contiguos region as a script line

Author: Libor Morkovsky, 2012

Count different bases in 5' end of reads in FASTQ

File: `scripts/5prime_stats.py` Find the most common letter in first n bases of reads in FASTQ file. Useful for finding and recognizing primer sequences in the reads.

Variant filters for PyVCF vcf_filter.py

File: `scrimer/pyvcf_filters.py` Implementation of vcf filters for *pyvcf vcf_filter.py*.

Author: Libor Morkovsky 2012

class `pyvcf_filters.`**`DistinguishingVariants`**(*args*)

Given a group of samples, choose variants that are not shared with the rest of the samples

classmethod `customize_parser`(*parser*)

`filter_name`()

`name` = 'contrast-samples'

CHAPTER 5

Source code and reporting bugs

The source code and a bugtracker can be found at https://github.com/libor-m/scrimer.

CHAPTER 6

License

Scrimer is licensed under the GNU Affero General Public License. Contact the author if you're interested in other licensing terms.

CHAPTER 7

Author

Libor Morkovsky
Department of Zoology
Charles University in Prague
Czech Republic
morkovsk[at]natur.cuni.cz

5

c

d

e

f

l

p

s

Symbols

B

C

D

E

F

L

N

P

S