BIOINFORMATICS + DATA VISUALIZATION

Learning Circos

BIOINFORMATICS AND GENOME ANALYSIS

FONDAZIONE EDMUND MACH, SAN MICHELE ALL'ADIGE, ITALY, 20 JUNE 2019

DOWNLOAD COURSE MATERIALS
v1.00 15 June 2019

A 1-day practical course in genomic data visualization with Circos. This material is part of the Bioinformatics and Genome Analysis course held at the Fondazione Edmund Mach in San Michele all'Adige, Italy.

QUICK LINKS

Course website | Circos documentation best practices getting started | Brewer palette swatches | Color resources | Points of View | Points of Significance

sessions / scripts

sessions / scripts / README

SCRIPTS

This page lists all the scripts from all lectures.

sessions / scripts / create.ebola.tracks

```
#!/bin/bash

# extract genes

echo "Creating gene track"
grep -v \# ebola.genes.txt | cut -f 2,5,6,9 | \
    awk '{print "ebola",$2,$3,$4,"name="$1}' > track.ebola.genes.txt

# extract SNPs

echo "Creating SNP track"
grep -v \# ebola.variation.txt | cut -f 2,3,4,10,15 | \
    perl -pe 's/\w+:\w+//' | \
    awk '{print "ebola",$1,$2,$5,"snp="$3",gene="$4}' > track.ebola.variation.txt
```

---

sessions / scripts / create.human.gene.tracks

```
#!/bin/bash

DATA=.
SCRIPTS=../../../scripts

# Change this to the directory where Circos is installed
CIRCOS=/home/martink/work/circos/svn/

# grep treats special characters as normal unless they're escaped
#
#https://stackoverflow.com/questions/6775904/grepping-using-the-alternative-operator

# About 43,000 genes that are on canonical chromosomes
echo "Creating list of 43,000 genes"
cat $DATA/genes.human.refseq.txt | grep -v "#" | grep -v -i "random\|_alt\|none\|chrun" > genes.43000.txt

# Gene entries with unique names
echo "Creating list of 19,300 genes (unique names)"
cat genes.43000.txt | sort -u -k 13,13 > genes.19300.txt

# Prefixes
```

```
echo "Counting genes for each 3-letter prefix"
cat track.genes.txt | cut -d " " -f 5 | sed 's/name=//' | cut -c 1-3 | sort | uniq -c | sort -rn > prefixes.count.txt

# Gene entries with unique first 3 letter substrings in their name
echo "Creating list of 3,500 genes (unique 3-letter prefix)"
cat genes.19300.txt | awk '{print $13,$0}' | uniq -w 3 > genes.3500.txt

# Histogram of gene size
echo "Creating histogram of gene sizes (in kb)"
cat genes.19300.txt | awk '{print ($6-$5+1)/1000}' | $SCRIPTS/histogram -min 0 -max 200 -binsize 10 > histogram.gene.size.txt

# Number of genes on each chromosome
echo "Counting genes on each chromosome"
cat genes.19300.txt | cut -d $'\t' -f 3 | sort | uniq -c | sort -nr > count.bychr.txt

# Create a gene track with fields
# CHR START END NUM_EXONS name=GENENAME
echo "Creating Circos data file of gene positions"
cat genes.19300.txt | awk '{print $3,$5,$6,$9,"name="$13}' | sed 's/chr/hs/' > track.genes.txt

# Count
BINS="1 5 10 20"
for ws in $BINS; do
  echo "Creating Circos data file of gene counts in $ws Mb windows"
  cat track.genes.txt | $CIRCOS/tools/resample/bin/resample -bin ${ws}e6 -count > track.gene.count.${ws}mb.txt
done
```

---

sessions / scripts / create.human.segdup.tracks

```
#!/bin/bash

DATA=.
SCRIPTS=../../../scripts

# Change this to the directory where Circos is installed
CIRCOS=/home/martink/work/circos/svn/

# Parse the segmental duplications into a Circos link format. Ignore all duplications on
# random or unanchored chromosomes.
echo "Creating Circos segmental duplication links"
cat $DATA/segdup.human.txt | grep -v \# | cut -d $'\t' -f 2,3,4,8,9,10 | grep -v -i 'random\|chrun\|_alt\|none' | sed 's/chr/hs/g' > track.s

# Histogram of sizes
echo "Creating histogram of segmental duplication size (in kb)"
cat track.segdup.all.txt | awk '{print ($3-$2-1)/1000}' | histogram.v2 -min 1 -max 20 -binsize 1 > histogram.segdup.size.txt

# Get a list of unique chromosomes from the link track ...
# ... search for the chromosome at start of line, add link size, sort by size, add i=NR where NR
#     is the awk record number and append to file tmp.txt

echo "Creating Circos segmental duplication links with size rank for each chromosome"

for chr in `cat track.segdup.all.txt | cut -d $'\t' -f 1 | sort -u` ; do
grep -w ^$chr track.segdup.all.txt | awk 'BEGIN { OFS="\t" } {print $3-$2,$0}' | sort -nr | awk 'BEGIN { OFS="\t" } {print $0,"sizerank="NR}
done

# Sort the indexed link by size (first field), remove the field and output to a track file

cat tmp.txt | sort -nr | cut -d $'\t' -f 2- > track.segdup.indexed.txt

\rm tmp.txt
```

---

sessions / scripts / create.yeast.links

---

sessions / scripts / create.yeast.tracks

```
#!/bin/bash

# Run this from day.1/data

DATA=.
SCRIPTS=../../scripts

# Change this to the directory where Circos is installed
CIRCOS=/home/martink/work/circos/svn/

echo "Creating list of 16,000 genes"
cat conservation/coords.txt | awk '{print $2,$3,$4,$4-$3+1,"name="$1}' > genes.txt

BINS="5 10 20 50 100"

for ws in $BINS; do
```

```
  echo "Counting genes in $ws kb bins"
  bin=$((ws * 1000))
  cat genes.txt | ~/work/circos/svn/tools/resample/bin/resample -bin $bin -count > genes.count.${ws}kb.txt
done

for ws in $BINS; do
  bin=$((ws * 1000))
  echo "Counting genes in $ws kb bins"
  cat genes.txt | $CIRCOS/tools/resample/bin/resample -bin $bin -count > genes.count.${ws}kb.txt
  echo "Calculating gene average size in $ws kb bins"
  cat genes.txt | $CIRCOS/tools/resample/bin/resample -bin $bin -avg > genes.avgsize.${ws}kb.txt
done

for type in conservation duplication; do
  echo "Creating $type links"
  cat $type/C* $type/S* $type/Z* | $SCRIPTS/create.yeast.links -coord $type/coords.txt > links.$type.txt
  echo "...extracting largest 10,000 links"
  N=10000
  cat links.$type.txt | awk '{print $3-$2+1,$0}' | sort -nr | head -$N | cut -d " " -f 2- > links.$type.$N.txt

  for ws in $BINS; do
      bin=$((ws * 1000))
      echo "...counting links in $ws kb bins"
      cat links.$type.txt | $CIRCOS/tools/binlinks/bin/binlinks -bin $bin -num > links.$type.count.${ws}kb.txt
  done

done
```

---

sessions / scripts / histogram

```
# read the values from STDIN and use the -col COLUMN, by default 0
my @values;
while(my $line = <>) {
  chomp $line;
  next if $line =~ /^\#/;
  my @tok = split(" ",$line);
  if(defined $CONF{col} && ! defined $tok[ $CONF{col} ]) {
    die "The line [$line] has no number in column [$CONF{col}]";
  }
  push @values, $tok[ $CONF{col}];
}

# get min and max of values
my ($min,$max) = minmax(@values);

# change min/max range if redefined via -min and/or -max
$min = $CONF{min} if defined $CONF{min};# && $CONF{min} > $min;
$max = $CONF{max} if defined $CONF{max};# && $CONF{max} < $max;

# *very* quick and dirty percentile if -pmin or -pmax used
if($CONF{pmin} || $CONF{pmax}) {
  my @sorted = sort {$a <=> $b} @values;
  $min = $sorted[ $CONF{pmin}/100 * (@sorted-1) ] if $CONF{pmin};
  $max = $sorted[ $CONF{pmax}/100 * (@sorted-1) ] if $CONF{pmax};
}

# determine bin size from either the -binsize command-line parameter
# or from -nbins. At least one of these is defined - we made sure of
# that in validateconfiguration()
my $bin_size;
if($CONF{binsize}) {
  $bin_size = $CONF{binsize};
  $CONF{nbins} = round(($max-$min)/$CONF{binsize});
} else {
  $bin_size = ($max-$min)/$CONF{nbins};
}

# populate bins
my $bins;
for my $i (0..$CONF{nbins}-1) {
  my $start = $min + $i * $bin_size;
  my $end   = $min + ($i+1) * $bin_size;
  my $n     = grep($_ >= $start && ($_ < $end || ( $i == $CONF{nbins}-1 && $_ <= $end)), @values);
  push @$bins, { start=>$start,
                 i=>$i,
                 end=>$end,
                 size=>$bin_size,
                 n=>$n };
}

my $histogram_width = $CONF{height};
my $maxn   = max(map { $_->{n} } @$bins);
my $sumn   = sum(map { $_->{n} } @$bins);
my $cumuln = 0;

# report number and fraction of values smaller than requested minimum
printinfo(sprintf("%10s %10.4f>%6d %6.3f","",
                  $min,
                  int(grep($_ < $min, @values)),
                  int(grep($_ < $min, @values))/@values));
```

```
# report each bin: min, max, count, fractional count, cumulative count
for my $bin (@$bins) {
  $cumuln += $bin->{n};
  my $binwidth = $bin->{n}*$histogram_width/$maxn;
  printinfo(sprintf("%10.4f %10.4f %6d %6.3f %6.3f %-${histogram_width}s",
                    $bin->{start},
                    $bin->{end},
                    $bin->{n},
                    $bin->{n}/$sumn,
                    $cumuln/$sumn,
                    "*"x$binwidth));
}

# report number and fraction of values larger than requested maximum
printinfo(sprintf("%10s %10.4f<%6d %6.3f","",
                  $max,
                  int(grep($_ > $max, @values)),
                  int(grep($_ > $max, @values))/@values));

# aggregate stats for full data set
printinfo(sprintf("%8s %12d","n",int(@values)));
printinfo(sprintf("%8s %12.5f","average",average(@values)));
printinfo(sprintf("%8s %12.5f","sd",stddev(@values)));
printinfo(sprintf("%8s %12.5f","min",min(@values)));
printinfo(sprintf("%8s %12.5f","max",max(@values)));
printinfo(sprintf("%8s %12.5f","sum",sum(@values)));
```

---

sessions / scripts / make.ebola.karyotype

```
#!/bin/bash

echo "parsing UCSC assembly table and making karyotype file"
tail -1 ebola.assembly.txt | cut -f 4 | awk '{print "chr - ebola ebola 0",$1,"black"}' > ebola.karyotype.txt
```

---

sessions / scripts / make.random

```
#!/bin/bash

# Randomize all colors except white and black. Try removing white,black and see what happens! Crazy, eh?

OPT="-param ideogram/show=no -randomcolor white,black"

# Alternatively, remap the colors to the hilarious and wonderfully useless
# watermelon-mint color palette (red-white-green), as requested in
# good fun by students from the EMBO course in Izmir. It's
# particularly fitting here because it matches the colors on the
# Italian flag.
#
# OPT="-param ideogram/show=no -wmmn"

for i in `seq 1 9` ; do
f=`echo "scale=1;(10-$i)/10" | bc`
echo "Drawing for hidden fraction $f to circos.$i.png"
(circos $OPT -outputfile circos.$i.png -param hidefraction=$f > out.$i; echo "Done tile $i fraction $f") &
done

echo "Please wait 30-60 seconds until all processes report finished..."
echo "Running..."
```

---

sessions / scripts / make.tiles

```
#!/bin/bash

for i in `seq 1 9`; do
convert circos.$i.png -gravity Center -crop 850x850+0+0 circos.$i.crop.png
done

montage -mode Concatenate -geometry 850x850 circos.*.crop.png circos.tiles.png
```

---

sessions / scripts / refresher

```
$\ = "\n"; # default end of line delimiter
$, = " ";  # default field delimiter

my $x = 1;
my @x = (1,2,3);
my %x = (a=>1,b=>2,c=>3);

# Note that $x @x and %x are all DIFFERENT variables, even
# though they have the same name. There is the scalar x, list x and hash x.
# For now, consider them to be independent and unrelated.
#
# Avoid doing things like
#
# @xlist = (1,2,3)
#
# since @xlist is obviously a list.
#
# It's generally bad practise to add the variable type to its name. However,
# I do this in this script for clarity. For example, the variable $xlistref
# would probably just be called $things or $fruits.
#
# Treat the words "list" and "array" as synonymous. Technically, an array
# is the data type and the list is the thing that it contains, but this
# is a minor distinction. For compound lists (e.g. lists of lists) you would
# say "two-dimensional array" and not "two-dimensional list".

print ("if we print a list we get",@x);
print ("if we print a hash we get",%x);

print ("list via Data::Dumper");
print Dumper(\@x);
print ("hash via Data::Dumper");
print Dumper(\%x);

my $xlistref = \@x;
my $xhashref = \%x;

print ("xlistref",$xlistref,"is reference to",ref $xlistref);
print ("xlistref",$xhashref," is reference to",ref $xhashref);

# Dereference the references back to their original types

my @y = @$xlistref;
my %y = %$xhashref;

# looping over a list

for my $item (@x) {
  print ("looping over list",$item);
}

# when you get comfortable with the idea that $x and @x
# are different variables, you can do this

for my $x (@x) {
  print ("looping again over list",$x);
}

for my $i (0..@x-1) {
  print ("looping over list index",$i,"item",$x[$i]);
}

# looping over a reference of the list - requires that
# we dereference the list by prepending the list sigil @
# to the variable.
#
# to access a list item via reference, you use ->

for my $i (0..@$xlistref-1) {
  print ("looping over list ref index",$i,"item",$xlistref->[$i]);
}

# looping over keys in a hash
for my $key (keys %x) {
  print ("key",$key,"value",$x{$key});
}

# looping over keys in a hash reference
#
# to access a hash item via reference you use ->
for my $key (keys %$xhashref) {
  print ("key",$key,"value",$xhashref->{$key});
}

# variables can be evaluated in a context to force
# an "interpretation" of the variable

# when a list is evaluated in scalar context (i.e. it is
# being assigned to a variable that is explicitly a scalar)
# you get the number of items in the list.

my $scalar = @x;
print("list as scalar",$scalar);

# Note that many contexts are list context such as
#
# print @x
```

```
# for (@x)
#

# Since perl 0-indexes lists, the last object is at index

my $lastidx = @x-1;
print("last index",$lastidx);

# The loop below cycles over the indexes in @x

for my $i (0..@x-1) {
  print("index",$i,"value",$x[$i]);
}

# You can also access the last index via the special variable $#x
# so you can write the loop like this

for my $i (0..$#x) {}

# Personally, I do not like this notation and always write @x-1. This
# special variable is very very Perlish (Perl has lots of internal
# variables like this and it's up to you how many of them you want to use).
#
# http://perldoc.perl.org/perlvar.html

# For example, the current iterator in a for loop is accessed by $_

for (@x) {
  print("current item",$_);
}

# Evaluating a scalar in list context

$x = "hello";
my @list = $x;

# simply gives you a single element list ("hello").

# To quickly define lists, use the word quotation operator qw()

@list = qw(some words in a list);

# This is much faster than

@list = ("some","words","in","a","short","list");

# If you evaluate a list in a hash context, Perl will expect an
# even number of elements and assign them as key/value pairs. Notice
# that Perl will interpolate variables inside a string defined
# with double quotes.

my %wordhash = @list;
for my $key (keys %wordhash) {
  print "$key -> $wordhash{$key}";
}

# One of the reasons why Perl was very popular in its day was its
# support for regular expressions
#
# https://perldoc.perl.org/perlre.html

my $str = "abccddfffhiiijj";
if($str =~ /c/) {
  print($str,"matches");
}

# For example to match any pair of identical characters you would
# use (.)\1 where . is the character class for any character, the
# (.) is a capture bracket which stores the match in an internal
# variable accessible immediately by \1.
#
# Then during replacement this internal variable is available as $1. The
# notation is different (e.g. vs \1) because we're no longer in the middle
# of doing the match. The match is done and we're now replacing strings.

$str =~ s/(.)\1/[2 x $1]/g;
print("replaced",$str);

# Regular expressions are a big topic, but you only need to know a few
# important things to get started. You can test-drive your regular expressions here
#
# https://regex101.com/

# You'll often be splitting strings into fields.

$str = "10 20    30 55   102 28";

# Using " " tells split to treat any amount of whitespace as a delimiter

my @tok = split(" ",$str);
print("split line into",@tok);
print("second field is",$tok[1]); # remember lists are 0-indexed

# You can now select items that you want

for my $tok (@tok) {
  if($tok eq 10) { # string comparison
    # ...
```

```
  }
  if($tok =~ /0/) {
    print("item",$tok,"has a 0");
  }
}

# Alternatively, a Perlish thing to do is to apply grep() to the list.
# Trep takes a condition as the first argument, which can be the
# regular expression

for my $tok (grep(/0/, @tok)) {
  print("item",$tok,"has a 0 via grep");
}

# Strictly speaking grep(EXPRESSION,LIST) passes each value from LIST
# to the EXPRESSION via $_ and the expression is evaluated. If it
# evaluates to true then grep returns the list item, otherwise it does not.

for my $tok (grep($_ =~ /0/, @tok)) {}

# Grep is a *great* way to filter a list.

# To apply code to each list item, use map()

my @newtok = map { 2*$_ } @tok;
print("remapped list",@newtok);

# The code in { } can be anything and the new list is composed of the
# values returned by the code, as evaluated on each list element via $_

# You can combine map { } and grep(). For example, here we
# multiply each element that contains a 0 by 2. The list is not
# modified - both grep and map return copies of the elements.

print("map{} and grep()",map { 2*$_ } grep(/0/, @tok));

# To pull out all numbers made out of digits from a string

my $input = "aa1bb22ccc333dd444";
while( $input =~ /(\d+)/g ) {
  print "Found $1 in $input";
}

# There is a lot going on here. The regular expression operator =~ applies
# the regular expression (\d+) to $input. Because the regular expression has capture
# brackets and we're using /g and this is evaluated in a while loop, the operator
# will continue to return as many matches as it can find. The substring matched will be stored in the
# special variable $1, which you can think of as the result of the first capture bracket.

# Below I add a second set of capture brackets to the regular expression, which grabs
# the letters immediately following the digit. Here the + means "one or more".

while( $input =~ /(\d+)([a-z]+)/g ) {
  print "Found digit/letter combo $1/$2 in $input";
}

# A final useful regular expression character are the string anchors ^, which
# matches the start of a string, and $, which matches the end. Thus

if($input =~ /\d$/) { }

# will match if $input ends in a digit and

if($input =~ /^\d/) { }

# will match if it starts with a digit. To check that the string is made up exclusively
# of digits you would do this

if($input =~ /^\d$/) { }
```