Genetical genomics with Affymetrix gene expression arrays

Alberts, Rudi

CHAPTER 6

Computational protocols for genetical genomics analyses with Affymetrix arrays

Abstract

We here describe computational protocols for genetical genomics experiments with Affymetrix arrays. In genetical genomics experiments, gene expression profiles of genetically related individuals are combined with molecular markers in the DNA to reveal expression quantitative trait loci (eQTL), that indicate regulatory links between genes. The protocols presented involve 1) pre-processing of Affymetrix array data; 2) QTL analysis and 3) verification of Affymetrix probe sequences against messenger and genomic sequence databases.

Nowadays, gene expression microarrays have become more and more affordable and genetical genomics (Jansen and Nap 2001) experiments are performed more often and on a larger scale. Accompanying this development is a growing need for good analysis tools. Especially the Affymetrix technology, which a popular platform for profiling gene expression, poses many challenges in data analysis since it uses multiple 25-mer probes per gene to measure its mRNA abundance.

Here we describe a workflow from raw Affymetrix data in the form of .CEL files to QTL visualization and the probe elimination procedure of chapter 5 (see Figure 6.1). The figure is divided into a pre-processing part (background correction and normalization) and a processing part (QTL analysis and probe elimination). The R code for each of the computation steps (squared boxes) is given in sections 6.2 through 6.6.

Next, we will describe how a custom track in the UCSC Genome Browser visualizing individual Affymetrix probes can be created and used.

6.1 Installing R and Bioconductor

R is a freely available, open source statistical programming language (`www.r-project.org`). Bioconductor (`www.bioconductor.org`) is an open source and open development software project for the analysis and comprehension of genomic data. For the pre-processing of Affymetrix microarray data we will use the *affy* package from Bioconductor within R. Perform the following steps to setup R and Bioconductor:

- Download R from www.r-project.org and install it.
- To install Bioconductor, start R and type `source("http://www.bioconductor.org/biocLite.R")` followed by `biocLite()`.
- Depending on the array you are using, you may need to install the appropriate Chip Definition File. Go to `http://bioconductor.org/data/cdfenvs/repos/html/` to find the name of the appropriate package and type in R `biocLite('mgu74av2cdf')` where `mgu74av2` is replaced by the name of your chip.

6.2 Background correction and quantile normalization

Collect the .CEL files in one directory and use the following R code for background correction and quantile normalization:

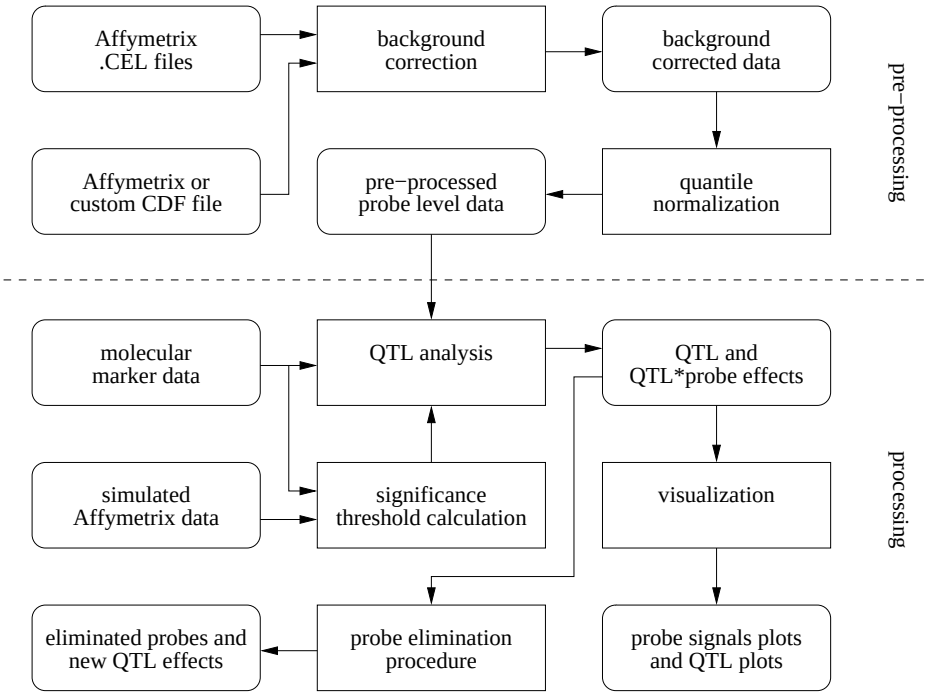

Figure 6.1: Pre-processing and processing of Affymetrix array data

```
rmapreprocessing <- function( celfilepath,outfilecsv )
{
  library(affy)

  A   <-  ReadAffy( celfile.path=celfilepath ) # readin all .CEL files in directory
  A2  <-  bg.correct.rma( A )                  # RMA background correction
  A3  <-  normalize.AffyBatch.quantiles( A2 )  # RMA quantile normalization
  # take log2 and add probe names
  pmperprobe  <- cbind( probeset=probeNames(A3),
                        round( log2(data.frame(pm(A3))),3 ) )
  write.table(pmperprobe,outfile,sep=",")      # write output to file
}
```

6.3 Significance threshold calculation

First, the significance thresholds for QTL and QTL×probe effect will be calculated via simulation. The real molecular marker data is used and the Affymetrix expression data is simulated from a normal distribution with mean 0 and sd 1. The R code is as follows:

```
qtlthreshold <- function( markers,batchnrs,nrind=30,outfilecsv )
{
```

```
  library(nlme) #necessary for lme function

  givemarker <- function (letter) # change marker letters to numbers
  {
    if (letter=="B") return(1)
    if (letter=="N") return(2)
  }

  set.seed(1234)             # initialize random number generator
  nrp       <- 16            # number of probes
  nrvalues  <- nrp*nrind     # number of values necessary per probe set
  rand      <-rnorm( nrvalues*10000+1000 ) # generate one vector with random
                                           # expression data with mean 0 and sd 1.
  anovainalg <- data.frame( batch=rep(batchnrs,each=nrp),
                            probe=1:nrp, mouse=rep(1:nrind,each=nrp) ) # general
                            # part of anova input, only marker will be added

                               # vectors to store simulation results:
  maxFmarker      <- NULL      # F-value QTL
  maxFinteraction <- NULL      # F-value QTL:probe
  maxpmarker      <- NULL      # P-value QTL
  maxpinteraction <- NULL      # P-value QTL:probe

  N <- apply( markers, c(1,2), givemarker ) # change markers to numbers

  for (j in 0:9999)  #simulate 10000 times
  {
    cat( j )
              # get data for one simulation from the rand vector
    exprR <<- round( rand[(j*nrvalues+1):(j*nrvalues+nrvalues)],2 )
    Fmarker       <- NULL # vectors to store results of this simulation
    Finteraction  <- NULL
    pmarker       <- NULL
    pinteraction  <- NULL

    for (mar in 1:ncol(markers))  # loop over all markers
    {
                # add marker and expression data to anovain
    exprR <<- round( rand[(j*nrvalues+1):(j*nrvalues+nrvalues)],2 )
    Fmarker       <- NULL # vectors to store results of this simulation
    Finteraction  <- NULL
    pmarker       <- NULL
    pinteraction  <- NULL

    for (mar in 1:ncol(markers))  # loop over all markers
    {
      # cat(mar)
                # add marker and expression data to anovain
      anovain <- data.frame( anovainalg, marker=rep(N[,mar],each=nrp),
                             value=exprR )
           # fit linear mixed-effects model
      g <- lme( value~factor(batch) + factor(probe) +
                factor(batch):factor(probe) + factor(marker) +
                factor(marker):factor(probe),
                data=anovain,
                random=~1|factor(mouse) )
      ag <- anova(g) # compute anova table
      # extract F and p values from anova table
      Fmarker       <- c( Fmarker,      as.numeric(ag[[3]][4]))
      Finteraction  <- c( Finteraction, as.numeric(ag[[3]][6]))
      pmarker       <- c( pmarker,      as.numeric(ag[[4]][4]))
      pinteraction  <- c( pinteraction, as.numeric(ag[[4]][6]))
    }
    # collect maximum F and p values for all simulations
    maxFmarker      <-c( maxFmarker,      max(Fmarker) )
    maxFinteraction <-c( maxFinteraction, max(Finteraction) )
    maxpmarker      <-c( maxpmarker,      max(pmarker) )
    maxpinteraction <-c( maxpinteraction, max(pinteraction) )
  }
                # collect all results in one data.frame
  thresholds <- data.frame( maxFmarker=maxFmarker, maxFinteraction=maxFinteraction,
                            maxpmarker=maxpmarker, maxpinteraction=maxpinteraction )

  write.table( thresholds,file=outfilecsv ) # store results
}
```

6.4 QTL analysis

QTL analysis is performed as described in Chapter 3 using the following code:

```
mapqtlperprobeset <- function( markers,expressions,batchnrs )
{
  library(nlme)                 # necessary for lme function

  nrp     <- nrow(expressions)  # number of probes
  onevec  <- NULL               # put data in one vector

  for( k in 1:ncol(expressions) )
  {
    onevec <- c(onevec,expressions[,k])
  }

  anovainalg <- data.frame( batch=rep(batchnrs,each=nrp),
                            probe=1:nrp,mouse=rep(1:ncol(expressions),each=nrp),
                            value=onevec ) # general part of anova input,
                            # only marker will be added

                             # vectors to store results:
  Fmarker       <- NULL      # F-value QTL
  Finteraction  <- NULL      # F-value QTL:probe
  pmarker       <- NULL      # P-value QTL
  pinteraction  <- NULL      # P-value QTL:probe

  for (mar in 1:ncol(markers)) #  QTL mapping on all markers
  {
                # add marker data to anovainnow
    anovainnow <- cbind( anovainalg, marker=rep(markers[,mar], each=nrp) )
          # fit linear mixed-effects model
    g   <- lme( value~factor(batch) + factor(probe) +
                factor(batch):factor(probe) + factor(marker) +
                factor(marker):factor(probe),
                data=anovainnow,
                random=~1|factor(mouse) )
    ag  <- anova(g) # compute anova table

                      # get F and p values from anova
    Fmarker       <- c( Fmarker,      as.numeric(ag[[3]][4]))
    Finteraction  <- c( Finteraction, as.numeric(ag[[3]][6]))
    # derive p values using the pf function, to avoid zero p values of which
    # you can not take a log
    pmarker<-c(pmarker,pf(1/as.numeric(ag[4,3]),ag[4,2],ag[4,1]))
    pinteraction<-c(pinteraction,pf(1/as.numeric(ag[6,3]),ag[6,2],ag[6,1]))
  }

  # collect results
  perprobeset <- data.frame( marker=colnames(markers), Fmarker=Fmarker,
                             Finteraction=Finteraction, pmarker=pmarker,
                             pinteraction=pinteraction )
  return( perprobeset ) #return results
}

mapqtlperprobe <- function( markers, expressions, batchnrs )
{
  nrp     <- nrow( expressions )  # number of probes
  onevec  <- NULL                 # put data in one vector

  for( k in 1:ncol(expressions) )
  {
    onevec<-c(onevec,expressions[,k])
  }
  anovainalg <- data.frame( batch=rep(batchnrs,each=nrp), probe=1:nrp,
                            mouse=rep(1:ncol(expressions),each=nrp), value=onevec ) # general
                            # part of anova input, only marker will be added

                            # vectors to store results:
  perprobe        <- NULL   # results

  for( mar in 1:ncol(markers) )  #  QTL mapping on all markers
  {
    Fmarkerperprobe <- NULL   # F-value QTL
    pmarkerperprobe <- NULL   # p-value QTL
                # add marker data to anovainnow
```

```
    anovainnow <- cbind( anovainalg, marker=rep(markers[,mar], each=nrp) )
    for (p in 1:nrp)
    {
      nowin <- anovainnow[anovainnow$probe==p,] # select data for probe p
      g     <- lm( value~factor(batch)+factor(marker),nowin ) # fit linear model
      ag    <- anova(g) # compute anova table
                        # get F and p values from anova
      Fmarkerperprobe <- c( Fmarkerperprobe,as.numeric(ag[[4]][2]))
      pmarkerperprobe <- c( pmarkerperprobe,as.numeric(ag[[5]][2]))
    }
  rownames(perprobe) <- 1:nrow(perprobe)
  return(perprobe) #return results
}
```

6.5 Visualization

The following code is used to create plots of the Affymetrix probe signals.

```
probesignals <- function( pngfilename, probeset, pos, signals, mycolors )
{
  png( pngfilename, height = 550, width = 700 )  # open png file
  pos <- pos - min(pos) + 1                      # let positions start from 1
  lengthprobe <- 25
  par( mai=c(0, 1, 0.2, 0.3) )                      # set margins
  par( fig=c(0,1,0.84,1) )                       # upper part of figure
  # plot empty box
  plot( c( pos,max(pos)+lengthprobe+2 ),
        c( 1:length(pos),length(pos) ),
        xlab=paste("probeset",probeset),
        ylab="probe",type="n",xaxt="n" )

  for( i in 1:length(pos) ) # per probe, plot line and probe number
  {
    lines( c(pos[i], pos[i]+lengthprobe), c(i,i), lwd=1, col="blue" )
    text( pos[i]+13, i+0.2, i )
  }

  text( pos[length(pos)]+25, 2, "Relative probe position (bp)", pos=2)
  par( mai=c(1.2, 1, 0.1, 0.3)) # set margins
  par( fig=c(0,1,0,0.8), new=T, xpd=T )              # lower part of figure
  myxlab <- paste( "probe signals for probeset", probeset )
  # plot the signals (either PM or MM)
  matplot( signals, type="l", main="", xlab=myxlab, xaxt="n",
           ylab="log2 intensity", lty=1:ncol(signals), ylim=range(5:15),
           col=mycolors)
  axis( 1, 1:nrow(signals), 1:nrow(signals) ) # add numbers to X axis
  dev.off()
}
```

The following code is used to create QTL plots on probe set and on probe level.

```
qtlplotperprobeset <- function( pngfilename, probeset, markerpos, marker, interaction,
                                thresmarker, thresinteraction, offsetperchr)
{
  png(file=pngfilename, bg="white", width=1100, height=300)
  par(mai=c(1, 1, 0.5, 0.3)               # set margins
  maxheight <- max( marker, interaction ) # get max for scaling the plot

  # make empty plot of wished size
  plot( markerpos, marker, type="n",
        main=paste("QTL analysis per probe set for gene .. probe set",probeset),
        xlab="marker", ylab="- 10log(p value)", ylim=range(0:maxheight*1.2))

  for( i in 1:20 )
  {
    lines( c( offsetperchr[i],offsetperchr[i] ),
           c( 0,maxheight ),lty=1,lwd=1,col="grey") # add lines and names for chr's
    if( i == 20 ) { lines( c(2598,2598), c(0,maxheight),lty=1,lwd=1,col="grey") }
```

```
    if( i < 20  ) { text( offsetperchr[i], maxheight, i, cex=1.2, pos=4) }
    if( i == 20 ) { text( offsetperchr[i], maxheight, "X", cex=1.2, pos=4) }
  }

  lines( markerpos, interaction, col="green4") # plot -10log(p) values and thresholds
  lines( markerpos, marker, col="blue")
  lines( c(0,2598), c(thresinteraction,thresinteraction),col="green4",lty=2)
  lines( c(0,2598), c(thresmarker,thresmarker),col="blue",lty=2)
  dev.off()
}

qtlplotperprobe <- function( pngfilename, probeset, markerpos, markerperprobe,
                             thres, offsetperchr)
{
  png(file=pngfilename, bg="white", width=1100, height=700)
  numberofprobes<-length(unique(qtlperprobe$probenr)) # get number of probes
  maxheight <- max(markerperprobe$min10logpmarker) # get max for scaling the plot
  stepsize  <- maxheight
  # make empty plot of wished size
  plot( markerpos, markerperprobe[markerperprobe$probenr==1,5], type="n",
        main=paste("QTL analysis per probe for gene .. probe set",probeset),
        xlab="marker", ylab="-10log(p)", ylim=range(0:(maxheight*110*numberofprobes)/100),
        cex=1) # make empty plot of wished size
  totmaxheight <- (maxheight*110*numberofprobes)/100

  for (i in 1:20)
  {
    lines( c(offsetperchr[i], offsetperchr[i]), c(0,totmaxheight),
           lty=1, lwd=1, col="grey" ) # add lines and names for chr's
    if( i==1 ) {
        text(offsetperchr[i],totmaxheight,"chr 1",pos=4)
    }
    else if( i < 20 && i > 1 ) {
        text(offsetperchr[i],totmaxheight,i,pos=4)
    }
    else if( i == 20 ) {
      lines( c(2598,2598), c(0,totmaxheight), lty=1, lwd=1, col="grey" )
      text( offsetperchr[i], totmaxheight, "X", pos=4 )
    }
  }

  mycols <- rep( c("orange1","orange4","lightblue4","blue4"), 6 )
  for( i in 1:numberofprobes )
  {
    # plot -10log(p) values and thresholds per probe
    pp <- data.frame( markerperprobe[markerperprobe$probenr==i,5] )
    lines( markerpos, pp[,1]+(i-1)*stepsize, col=mycols[i] )
    text( 24, (i-1)*stepsize, i, col=mycols[i], pos=2 )
    lines( c(0,2598), c((i-1)*stepsize+thres, (i-1)*stepsize+thres),
           lty=2, col=mycols[i] )
  }
  dev.off()
}
```

6.6 Probe elimination procedure

The probe elimination procedure, as described in chapter 5, is performed using this code:

```
probeelimination <- function( signals, batchnrs, marker )
{

  library(nlme)
  # needed to make parameter estimates add up to zero
  options( contrasts=c("contr.sum","contr.poly") )

  nrp    <- nrow(signals)   # nr. of probes
  onevec <- NULL            # put data in one vector
```

```
  for( k in 1:ncol(signals) )
  {
    onevec <- c( onevec, signals[,k] )
  }

  anovain <- data.frame( batch=rep(batchnrs,each=nrp), probe=1:nrp,
                         mouse=rep(1:ncol(signals),each=nrp), value=onevec,
                         marker=rep(marker,each=nrp) )
  g       <- lme( value~factor(batch) + factor(probe) +
                  factor(batch):factor(probe) + factor(marker) +
                  factor(marker):factor(probe), data=anovain,
                  random=~1|factor(mouse) )
  ag      <- anova(g)
  pmarker <- pf( 1/ag[[3]][4], ag[[2]][4], ag[[1]][4] ) # qtl effect before dropping
  int     <- pf( 1/ag[[3]][6], ag[[2]][6], ag[[1]][6] ) # qtl *probe effect before dropping
  cat( probeset, " startint: ", int, "\n", file="dropped.txt", append=T )

  dropped <- NULL
  cat( "", file=paste( "drops", probeset, ".csv", sep=""), append=F )

  ## probe backwards elimination procedure: drop probes one by one,
  ##  see for which probe the interaction effect is reduced most,
  ##  leave this probe out permanently (it is added to 'dropped').
  ##  repeat for the remaining probes. store qtl and interaction effect after each drop.

  # drop until 2 probes left and store all results
  while( length(dropped) < nrp-2 )
  {
    newint <- NULL
    # drop each probe one by one that is not yet permanently dropped
    for( k in 1:nrp )
    {
      if( !k %in% dropped )
      {
        doit    <- dropprobes( c(k,dropped),anovainnow )
        newint  <- rbind( newint,
                          data.frame( probe=k, marker=doit[1], int=doit[2],
                                      b6bigger=doit[3]))
      }
    }
    # determine for which probe the interaction effect became the lowest.
    newint  <- newint[order(newint$int,decreasing=T),]
    # store this probe in 'dropped'
    dropped <- c(dropped,newint[1,1])
    int     <- newint[1,3]

    ## what happened to the dropped probes - here drop the non-dropped
    ##  probes to determine this

    qtldroppedones      <- -1
    intdroppedones      <- -1
    b6biggerdroppedones <- -1
    restant             <- 1:nrp
    restant             <- restant[ !restant %in% dropped ]
    if( length(dropped) > 1 )
    {
      doit                <- dropprobes( restant,anovainnow )
      qtldroppedones      <- doit[1]
      intdroppedones      <- doit[2]
      b6biggerdroppedones <- doit[3]
    }
    # store all results in one file
    cat( "dropped: ", newint[1,1], "marker: ", newint[1,2], "int: ", newint[1,3],
         "markerdropped: ", qtldroppedones,"alarm: ",newint[1,2]>qtldroppedones,
         "b6bigger", newint[1,4], "b6biggerdropped", b6biggerdroppedones,
         "intdropped", intdroppedones, "\n", file="dropped.txt", append=T )
     # store results in a file per probe set
    cat( "dropped: ,", newint[1,1], ",marker: ,", newint[1,2],",int: ,", newint[1,3],
         ",markerdropped,", qtldroppedones, ",alarm: ,", newint[1,2]>qtldroppedones,
         ", b6bigger, ", newint[1,4], ", b6biggerdropped, ", b6biggerdroppedones,
         ", intdropped, ", intdroppedones, "\n",
         file=paste("drops",probeset,".csv",sep=""), append=T )
  }
}

# todrop like c(1,5,7,8,9)
dropprobes <- function( todrop, data )
```

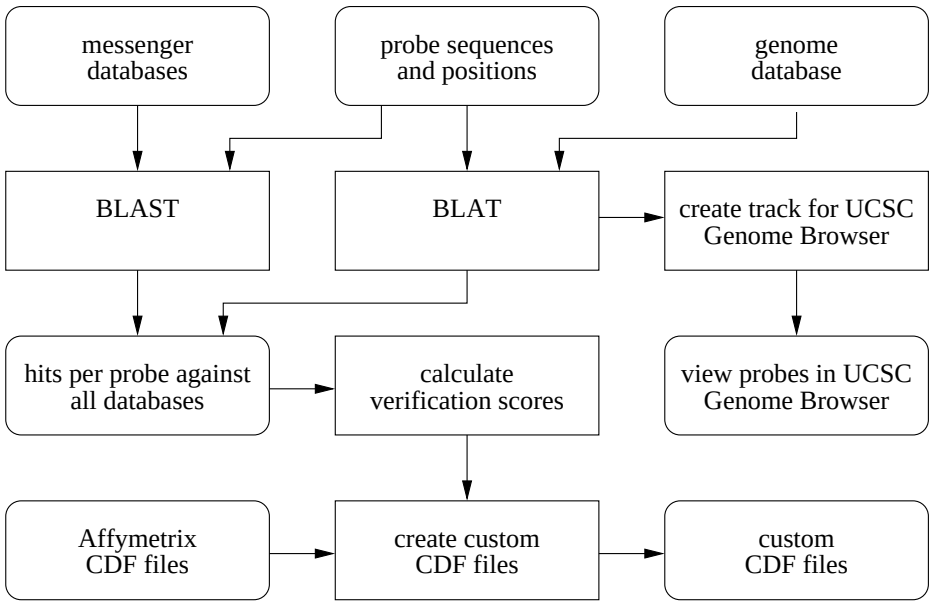

Figure 6.2: Workflow of probe verification protocol

```
{
  temp          <- data[!data$probe %in% todrop,]
                   # fit linear mixed-effects model
  g             <- lme( value~factor(batch) + factor(probe) +
                        factor(batch):factor(probe) +
                        factor(marker) + factor(marker):factor(probe),
                        data=temp, random=~1|factor(mouse) )
  ag            <- anova(g)  # compute anova table
  pmarker       <- pf( 1/ag[[3]][4], ag[[2]][4], ag[[1]][4] )
  pinteraction  <- pf( 1/ag[[3]][6], ag[[2]][6], ag[[1]][6] )
  # check whether B6 has higher (1) or lower (0) expression values than D2
  b6bigger      <- as.numeric(
    mean( temp[temp$marker=="B",]$value ) > mean(temp[temp$marker=="N",]$value) )
  return( c(pmarker, pinteraction, b6bigger) )
}
```

6.7 Probe verification protocol

The workflow for the probe verification protocol is depicted in Figure 6.2. The code is available online at `http://gbic.biol.rug.nl/supplementary/2007/verificationcode`.

6.8 Probes in UCSC Genome Browser

In chapter 5 a visualization of Affymetrix probes in the UCSC Genome Browser (UCSC Genome Browser, `http://genome.ucsc.edu`) is presented (figure 5.1C and D). This custom track can be accessed for human, mouse and rat arrays using URLs in the following format:

```
http://genome.cse.ucsc.edu/cgi-bin/hgTracks?org=ORGANISM
&db=DB&hgt.customText=http://gbic.biol.rug.nl/~ralberts/
tracks/ANCHIP/PROBESET
```

where ORGANISM, DB and CHIP are values from table 6.1 and PROBESET is an Affymetrix probe set name. Example:

```
http://genome.cse.ucsc.edu/cgi-bin/hgTracks?org=human
&db=hg18&hgt.customText=http://gbic.biol.rug.nl/~ralberts/
tracks/ANHGFocus/217869_at
```

ORGANISM	DB	CHIP
human	hg18	HCG110, HGFocus, HGU133A, HGU133A2, HGU133B, HGU133Plus2, HGU95Av2, HGU95B, HGU95C, HGU95D, HGU95E, HuGeneFL
mouse	mm8	MGU74Av2, MGU74Bv2, MGU74Cv2, MOE430A, MOE430B, Mouse4302, Mouse430A2, Mu11KsubA, Mu11KsubB
rat	rn4	RAE230A, RAE230B, Rat2302, RGU34A, RGU34B, RGU34C, RNU34, RTU34

Table 6.1

The custom tracks can be created using the following two perl scripts. The output of the BLAT analyses (see section 6.7) are assumed to be stored in a MySQL table with the following definition. An explanation of the fields that are used later is added.

```
TABLE blatgenome (
matches int(10) unsigned
misMatches int(10) unsigned
repMatches int(10) unsigned
nCount int(10) unsigned
qNumInsert int(10) unsigned
qBaseInsert int(10) unsigned
tNumInsert int(10) unsigned
tBaseInsert int(10) unsigned
strand char(2)              # DNA strand (+ or -) where the hit is
qName varchar(255)          # probe set name
qSize int(10) unsigned      # size of query sequence (probe spanning region)
```

```
qStart int(10) unsigned
qEnd int(10) unsigned
tName varchar(255)           # name of the chromosome
tSize int(10) unsigned
tStart int(10) unsigned
tEnd int(10) unsigned
blockCount int(10) unsigned  # number of matching blocks (often exons)
blockSizes longblob          # sizes of matching blocks
qStarts longblob             # start positions of matching blocks on query sequence
tStarts longblob             # start positions of matching blocks on genomic sequence
id int(7)
)
```

The first perl script reads in the sequences of the chromosomes and compares it with the BLAT hits to count the numbers of n's, the number of mismatches, the number of dashes ('-'), the number of matches and the positions of the mismatches.

```
#!/usr/bin/perl
# compare blat hit with genome to create scores
use strict;

use DBI;
use DBI qw(:sql_types);

my $probes_db = DBI->connect("DBI:mysql:host=localhost;database=$ARGV[0]",
                             "username","password",
                             {PrintError => 0, RaiseError => 1});

# the FASTA files of the chromosomes should be located here:
my $genome_dir = '/var/db/bioinfo/mousearray/ucsc/mm8/chr/';

my $chr_stmt =  $probes_db->prepare("SELECT tName FROM blatgenome GROUP BY tName ORDER BY tName");
my $hits_stmt =  $probes_db->prepare("SELECT id,qName,qSize,strand,blockCount,qStart,qStarts,tStarts,
                 blockSizes FROM blatgenome WHERE tName = ?");
my $target_stmt = $probes_db->prepare("SELECT sequence FROM maskedtarget WHERE probeset_id = ?");

#get all chromosomes
$chr_stmt->execute();

#loop over all chromosomes
while( my @values = $chr_stmt->fetchrow_array()) {

  #select chromosome and load sequence
  my $chr = $values[0];
  my $chr_seq = &loadChr($chr);

  #get all hits for chromosome
  $hits_stmt->bind_param(1,$chr,SQL_VARCHAR);
  $hits_stmt->execute();
  my $count = 0;
  while( my @values = $hits_stmt->fetchrow_array()) {
    my $id       = $values[0]; # id
    my $probeset = $values[1]; # qName
    my $length   = $values[2]; # qSize
    my $strand   = $values[3]; # strand
    my $blocks   = $values[4]; # blockCount
    my $s1pos    = $values[5]; # qStart
    my $spos     = $values[6]; # qStarts
    my $starts   = $values[7]; # tStarts
    my $sizes    = $values[8]; # blockSizes

    #remove trailing comma
    chop($starts);
    chop($sizes);
    chop($spos);

    #split positions into arrays
    my @blockStarts = split(',',$starts);
    my @blockSizes =  split(',',$sizes);
    my @blockPos = split(',',$spos);
    my $chr_pos = $blockStarts[0];

    #recontruct genome sequence based on block info
    my $matched_seq = '';
```

```
my $pos = length($matched_seq);
my @exons;

for (my $i=0;$i < $blocks; $i++) {

  $matched_seq .=  '-' x ($blockPos[$i] -  $pos);

#start eerste exon
  if ($i == 0) {
     push(@exons,$blockPos[0]);
  } else {
    if ($pos < $blockPos[$i]) {
      #end previous exon
      push(@exons,$pos);
      #start new exon
      push(@exons,$blockPos[$i]);
    }
  }
  $matched_seq .= substr($chr_seq,$blockStarts[$i],$blockSizes[$i]);
  $pos = length($matched_seq);
}

$matched_seq .= '-'x ($length - $pos);

#end last exon
push(@exons,$pos);

# take reverse complement if strand = '-'
if ($strand eq '-') {
  $matched_seq = &rc($matched_seq);
  # reverse exon pos
  my @tmp;
  for my $e (@exons) {
    push(@tmp, $length - $e);
  }
  @exons = reverse(@tmp);
}

# get target sequence
$target_stmt->bind_param(1,$probeset,SQL_VARCHAR);
$target_stmt->execute();

while( my @values = $target_stmt->fetchrow_array()) {

   my $target = $values[0];
   my $matched_len = length($matched_seq);
   my $target_len =  length($target);

   my $n_count = 0;    # number of n's
   my $mm_count = 0;   # number of mismatches
   my $nf_count = 0;   # number of -
   my $m_count = 0;    #number of matches;
   my @mismatches;     # mismatch positions

   #for each bp  compare target with matched sequence
   for (my $i = 0;$i < length($target);$i++ ) {
         # get i-th bp from target
     my $q  = substr("\L$target\E",$i,1);
     if ($q eq 'n' ) {
           $n_count++;
     } else {
            # get i-th bp from matched sequence
       my $m= substr("\L$matched_seq\E",$i,1);
       if ($m eq '-') {
         $nf_count++;
       } else {
         if ($m eq $q) {
                   $m_count++;
                 } else {
           $mm_count++;
           push(@mismatches,$i);
         }
       }

     }
  }
  # print results
```

```
      print $id."\t".$chr."\t".$chr_pos."\t".$strand."\t".$count++."\t".$probeset."\t".join(',',@exons)
                ."\t".$n_count."\t".$mm_count."\t".$nf_count."\t".$m_count."\t".join(',',@mismatches)."\t".
                $blocks."\t".$starts."\t".$spos."\t".$sizes."\t".$length."\t".$matched_len.
                "\t".$matched_seq."\n";
    }
  }
}

exit(0);

## loads the sequence of a chromosome from FASTA file
sub loadChr {
   my $chr = shift;
   #load chromosome  sequence
   my $chr_seq= '';
   open (CHR_FILE,$genome_dir.$chr.".fa") || die "can not find file for $chr";
   my $first = 1;
   while (my $line = <CHR_FILE>) {

         #skip first  comment line from fasta
     if ($first) {
           $first = 0;
         } else {
           chomp($line);
           $chr_seq .= $line;
         }
   }
   close CHR_FILE;
   return $chr_seq;
}

## reverse complement
sub rc {
  my $str= shift;
  $str = reverse($str);
  $str =~ tr/ATGCatgc/TACGtacg/;
  return $str;
}
```

The results of this script are assumed to be stored in a MySQL table with the following definition:

```
TABLE genomecheck (
id int(7)
chr varchar(30)
chr_pos  int(12)
strand   varchar(1)
count  int(5)
probeset_id  varchar(12)
exons   text
n_count  int(5)
mm_count  int(5)
nf_count  int(5)
m_count int(5)
mismatches text
blocks int(3)
starts  text
spos text
sizes text
length  int(5)
matched_len int(5)
matched_seq text
)
```

Information about the probes from Affymetrix (Affymetrix - NetAffx Analysis Center, `http://www.affymetrix.com/analysis/index.affx`) should be stored in a table of this format:

```
TABLE probes (
probeset_id varchar(30) # probe set name
pos int(6)              # probe interrogation position
seq varchar(25)         # probe sequence
)
```

The following perl script will generate the custom tracks based on these two tables:

```
#!/usr/bin/perl
# create UCSC custom track showing Affymetrix probes
use strict;
use DBI;
use DBI qw(:sql_types);

my $probes_db = DBI->connect("DBI:mysql:host=localhost;database=$ARGV[0]",
                             "username","password",
                             {PrintError => 0, RaiseError => 1});

# select hits from genomecheck table. results are ordered by score so that the best score comes last:
  the best genome hit appears in the output file
my $gc_stmt =  $probes_db->prepare("SELECT id,probeset_id,strand,blocks,starts,spos,sizes,chr,length,
               matched_seq,(m_count/(m_count + nf_count + mm_count)) as score from genomecheck where
               probeset_id not like 'AFFX%' order by score;");
# select probe positions and sequences
my $pos_stmt = $probes_db->prepare("SELECT pos,seq FROM probes WHERE probeset_id = ? order by pos");
# select position of first probe
my $minpos_stmt = $probes_db->prepare("SELECT min(pos) as minpos FROM probes WHERE probeset_id = ?
                  group by probeset_id;");

$gc_stmt->execute();

while( my @values = $gc_stmt->fetchrow_array()) { # loop over hits

  my $outputfile = $ARGV[0]."/".$values[1];; # output file to store results
  unless(open(FILE, ">$outputfile")) {
    print "Cannot open file \"$outputfile\" to write to! \n\n";
    exit;
  }
  # take values from mysql query
  my $id = $values[0];          # hit id
  my $probeset_id = $values[1]; # probe set name
  my $strand = $values[2];      # strand on DNA (+ or -)
  my $blocks = $values[3];      # number of matching blocks (often exons)
  my $starts = $values[4];      # start coordinates of blocks on chromosome
  my $spos = $values[5];        # positions of matching blocks on matched_seq
  my $sizes = $values[6];       # sizes of matching blocks
  my $chr = $values[7];         # chromosome
  my $length = $values[8];      # length of hit
  my $matched_seq = $values[9]; # genomic sequence of hit

  $pos_stmt->bind_param(1,$probeset_id,SQL_VARCHAR);
  $pos_stmt->execute();
  $minpos_stmt->bind_param(1,$probeset_id,SQL_VARCHAR);
  $minpos_stmt->execute();
  my @minposvalues = $minpos_stmt->fetchrow_array();
  my $minpos = $minposvalues[0];

  #split
  my @blockStarts = split(',',$starts);
  my @blockSizes =  split(',',$sizes);
  my @blockPos = split(',',$spos);

  my $chr_pos = $blockStarts[0];    # position of hit on chromosome
  my $newpos;                       # start coordinate of probe
  my $newpos24;                     # end   coordinate of probe
  my $endpos = $chr_pos+$length+24; # end   coordinate of hit

  # used in case the probe spans 2 exons:
  my $newposB;     # start coordinate second part
  my $newposB24;   # end   coordinate second part
  my $lengthleft;  # overlapping length left exon
  my $lengthright; # overlapping length right exon
  my $difference;  # difference between end and begin

  # let the browser go to the position of the probe set when track is loaded
  print FILE "browser position ".$chr.":".$chr_pos."-".$endpos."\n";
  print FILE "track name=probes description=\"affy probes\" visibility=3 useScore=1 color=0,60,120,\n";

  my $probenumber = 0;

  while( my @values = $pos_stmt->fetchrow_array()) { # loop over probes
    $probenumber = $probenumber+1;
    # the probe interrogation positions obtained from affymetrix are adjusted
```

```
# so that the position of the first probe is zero (that's why minpos is
# subtracted)
# since probe spanning regions are used for blat, we then know that the
# interrogation positions of the probes correspond with the coordinates of
# the matched_seq, ie. using the interrogation positions the probes can be
# found back on the genomic sequence.

my $pos       = $values[0]-$minpos; # interrogation position of probe
my $seq = $values[1];               # probe sequence
# part of genomic sequence where probe matches
my $matched_probe = substr($matched_seq,$pos,25);

# sets score to 400 (probe will be lightblue)
my $score=400;
# sets score to 1000 when probe perfectly matches (dark blue)
if (uc($seq) eq uc($matched_probe)) {
    $score=1000;
}
$newpos=0;
$newpos24=0;
$newposB=0;
$newposB24=0;
$lengthleft=0;
$lengthright=0;
$difference=0;

if ($strand eq '+') {
  for (my $i=0;$i < $blocks; $i++) {# determine in which block the probe starts
    if ($pos>= $blockPos[$i] & $pos<$blockPos[$i]+$blockSizes[$i]) {
      $newpos = $blockStarts[$i]+$pos-$blockPos[$i]; # start and end positions
      $newpos24 = $newpos+25;                        # of probe on chromosome
      # check probe on exon border?
      if ($pos+24 > $blockPos[$i]+$blockSizes[$i]) { # probe spans 2 exons
        $lengthleft = $blockPos[$i]+$blockSizes[$i]-$pos; # left size
        $lengthright = 25 - $lengthleft;             # right size
        $newposB=$blockStarts[$i+1];                 # start and end positions
        $newposB24=$newposB+$lengthright;            # of right part
        $difference=$blockStarts[$i+1]-$newpos;      # diff of end and begin
      }
    }
  }
  if ($newpos>0) {
    if ($newposB==0) { # print the probe info, coordinates and score in output file
      print FILE $chr." ".$newpos." ".$newpos24." ".$probeset_id."-".$probenumber.">>".$seq." "
                .$score." + 0 0 0 1 25 0\n";
    } else {
       print FILE $chr." ".$newpos." ".$newposB24." ".$probeset_id."-".$probenumber.">>".$seq." "
                .$score." + 0 0 0 2 ".$lengthleft.",".$lengthright." 0,".$difference."\n";
    }
  }
} else {     # strand is '-'
  #adjust pos
  $pos = $length-$pos-25; # reverse positions since probes are displayed at '+' strand
  for (my $i=0;$i < $blocks; $i++) { # similar to previous part
    if ($pos>= $blockPos[$i] & $pos<$blockPos[$i]+$blockSizes[$i]) {
      $newpos = $blockStarts[$i]+$pos-$blockPos[$i];
      $newpos24 = $newpos+25;
      # check probe on exon border?
      if ($pos+24 > $blockPos[$i]+$blockSizes[$i]) {
        $lengthleft = $blockPos[$i]+$blockSizes[$i]-$pos;
        $lengthright = 25 - $lengthleft;
        $newposB=$blockStarts[$i+1];
        $newposB24=$newposB+$lengthright;
        $difference=$blockStarts[$i+1]-$newpos;
      }
    }
  }
  # take reverse complement of probe sequence to display it
  # along the '+' strand (which is shown by default in the genome browser)
  $seq = &rc($seq);
  if ($newpos>0) {
    if ($newposB==0) {
      print FILE $chr." ".$newpos." ".$newpos24." ".$probeset_id."-".$probenumber."<<".$seq." "
                .$score." - 0 0 0 1 25 0\n";
    } else {
    print FILE $chr." ".$newpos." ".$newposB24." ".$probeset_id."-".$probenumber."<<".$seq." "
              .$score." - 0 0 0 2 ".$lengthleft.",".$lengthright." 0,".$difference."\n";
```

```
      }
     }
   }
  }
close OUTFILE;
}
exit(0);

## reverse complement
sub rc {
  my $str= shift;
  $str = reverse($str);
  $str =~ tr/ATGCatgc/TACGtacg/;
  return $str;
}
```

6.9 Using R on a Linux cluster

Using a template directory structure, which is available at `http://gbic.biol.rug.nl/~ralberts/template.zip`, R scripts can be run on a Linux cluster. It is assumed that the work can be divided into independent parts, which will run on different nodes of the cluster. There will be one big for loop, that is split up into pieces to be executed on the nodes.

First, copy the R source code, e.g. from here `http://cran.cict.fr/src/base/R-2/R-2.4.1.tar.gz` to your account. Unzip it, change to the R-2.4.1 directory and type .configure followed by make. Now R can be run by typing /R-2.4.1/bin/R.

The template directory contains four subdirectories: 1) input, containing input file; 2) output, where output will appear; 3) scripts, containing the R script(s); and 4) logs, where log files will appear.

We start in the root directory, where makeit.R will create small scripts for submitting jobs to the queuing system. Set the expected time per job to 24:00:00 or 240:00:00, the amount of jobs needed and the proper path to R in this script. Run it by typing 'R CMD BATCH makeit.R'. Files job001 etc. as well as the file submitall are created. This is makeit.R:

```
cat("",file="submitall",append=F)
for (i in 1:3) {
 scriptname<-paste("job",formatC(i,width=3,flag="0"),sep="")
 cat("",file=scriptname,append=F)
 cat("#!/bin/sh \n #PBS -l walltime=23:59:00 \n #PBS -l nodes=1 \n cd $PBS_O_WORKDIR \n export
      USERVARIABLE=\"",i,"\" \n cd scripts \n /home/rugwn140/alberts/R-1.8.1/bin/R CMD BATCH script.R"
      ,file=scriptname,append=T,sep="")
 cat("qsub job",formatC(i,width=3,flag="0"),"\n",sep="",file="submitall",append=T)
}
```

Example output file job001 looks like this:

```
#!/bin/sh
 #PBS -l walltime=23:59:00
 #PBS -l nodes=1
 cd $PBS_O_WORKDIR
 export USERVARIABLE="1"
 cd scripts
 /home/rugwn140/alberts/R-1.8.1/bin/R CMD BATCH script.R
```

Put the input data in the input directory. This is example inputdata in the file /input/data.csv:

```
val1,val2,val3,val4,val5,val6,val7,val8,val9,val10
gene01,1,4,2,3,2,3,1,2,3,2
gene02,101,102,104,102,103,102,104,103,102,103
gene03,1003,1002,1004,1002,1004,1003,1002,1004,1002,1004
gene04,1002,1008,1006,1006,1003,1002,1008,1006,1006,1003
gene05,1002,1005,1002,1007,1004,1002,1004,1003,1002,1004
gene06,4,2,4,6,7,4,5,6,8,7
gene07,2,1,2,3,5,4,3,4,3,4
gene08,3,4,3,2,3,6,5,2,3,4
gene09,6,5,4,5,6,1,4,3,2,3
gene10,3,4,5,6,9,87,6,5,6,4
```

Put the script in the scripts directory. This is an example script that calculates the mean of all rows:

```
# one big for loops is split into parts and the environment variable
# USERVARIABLE determines which part is run

# get the value of the environment variable USERVARIABLE
jobnumber<-as.numeric(Sys.getenv("USERVARIABLE"))

# prepare log file
filename<-paste("../logs/log",formatC(jobnumber,width=3,flag="0"),".txt",sep="")
cat("",file=filename,append=F)
sink(filename,append=T)

# readin data
data<-read.csv("../input/data.csv")

# for collecting the results
jobresult<-NULL

# the amount of steps per job
repeatsperjob<-3

# part of the big for loop is run here
# for example, if jobnumber=1 and repeatsperjob=10, it runs from 1 till 10
#              and if jobnumber=2 and repeatsperjob=10, it runs from 11 till 20
for (j in (((jobnumber-1)*repeatsperjob+1):(jobnumber*repeatsperjob))) {

  # do the calculation, in this example, taking the mean of a row of values
  temp<-data.frame(rownames(data)[j],mean(as.numeric(data[j,])))
  # add the results to jobresult
  jobresult<-rbind(jobresult,temp)

}

# write the results to a file in the output folder
write.table(jobresult,file=paste("../output/out",jobnumber,".csv",sep=""),sep=",",row.names=FALSE,
                                col.names=FALSE)
```

Go back to the root and type 'source submitall' to submit all jobs. Type 'qstat -a' to check the status of your jobs, 'qdel' followed by the jobnumber to delete jobs, and 'qsub' to submit individual jobs. The output will appear in the output folder. Check *.e* and *.o* in the root and the files in logs to see whether everything went well.