Bioinformatics Methods

Forward Epigenetics using a CRISPR-based Screening Method

Anna Köferle

A thesis presented for the degree of
Doctor of Philosophy

University College London
London
September 2016

Contents

Chapter 1

Degenerate gRNA libraries

1.1 Finding all gRNA targets in the repeat-masked human genome

The genomic target sites of gRNAs designed for use with the *S.pyogenes* CRISPR/-Cas system are of the general form $GN_{20}GG$. The presence of the G at the start of the motif is required for initiation of transcription from the U6 promoter, present on many gRNA expression vectors. This is followed by 19 random bases (N) that determine the gRNA target site - these bases comprise the gRNA protospacer. The protospacer sequence is followed by the protospacer-adjacent motif (PAM), which is not part of the gRNA sequence but is present in the genomic target sequence just 3' of the gRNA target site. For the *S.pyogenes* CRISPR/Cas system, the PAM has to be comprised of the bases NGG.

A gRNA library is a pooled collection of gRNAs. A random library has a complexity of 4^{19}, which corresponds to 10^{12} different sequences. The aim here is to reduce this complexity and maximise binding to the genome. To this end, I first identified the number of occurences of the sequences GN20GG in the human genome. The Bioconductor (Bioconductor version 2.14) package BSgenome [9] was used to identify all sequences matching this pattern in the repeat-masked human genome in R (version 3.1.1). An R script named `find_all_gRNAs.R` contains the code and outputs a file containing chromosome coordinates and strand information for all hits, with the last column containing the pattern identifier. The script looks for the $GN_{20}GG$ by default (and outputs a file named `GN20GG_masked_allregions.txt`), unless another pattern is supplied by the user as follows:

```
./find_all_gRNAs.R -p [pattern, e.g. "GTACN"], -n [pattern
   -name, e.g. "my-pattern"], -o [outputfilename, e.g. "
   All_GTACN_hg19_RM.txt"]
```

The regions mapping to chromosomes 1-22 and the sex chromosomes were extracted from the outputfile `GN20GG_masked_allregions.txt` as follows:

```
grep -v 'random' GN20GG_masked_allregions.txt | grep -v '
```

```
   hap' | grep -v
'chrUn' | grep -v chrM > GN20GG_masked_autoXY.txt;

#remove the last column
awk '{print $1 "\t" $2 "\t" $3 "\t" $4}'
   GN20GG_masked_autoXY.txt >
GN20GG_masked_autoXY.bed;
```

1.2 Identifying gRNAs that overlap with known promoters

In order to generate an annotation file containing promoter regions, the annotation file for known transcripts (human GRCh37-70) was downloaded from Ensembl (version70) and parsed through a script supplied by Gareth Wilson. This script can be found here: https://github.com/regmgw1/regmgw1_scripts/blob/master/ensembl_scripts/transcript2promoter.pl). This script extracts coordinates -1000 and +500 bp from the start of the transcript. From this file non-overlapping promoter sequences were derived by strand-specific merging using the Bedtools suite (v2.17.0).

```
#strand-specific merging of promoter annotation file
mergeBed -s -i promoters.gff | awk '{print "chr"$1 "\t" $2
    "\t" $3 "\t"
$4}' > promoters_merged.bed

#use Bedtools to find regions that overlap promoter
   regions with minimum
of 1 bp
intersectBed -a GN20GG_masked_autoXY.bed -b
   yourpath2annotation_files/
human_GRCh37_70/promoters_merged.bed -wa >
```

```
GN20GG_masked_autoXY_promoters_merged;
```

Then, FASTA coordinates were retrieved using twoBitToFa:

```
#Separate according to whether pattern is on plus or minus
    strand
grep '+' GN20GG_masked_autoXY_promoters_merged >
GN20GG_masked_autoXY_promoters_merged_PLUS;
grep '-' GN20GG_masked_autoXY_promoters_merged >
GN20GG_masked_autoXY_promoters_merged_MINUS

#make into a gff file
awk '{print $1 ":" ($2 - 1) "-" $3}'
   GN20GG_masked_autoXY_promoters_PLUS  >
GN20GG_masked_autoXY_promoters_PLUS.gff;

twoBitToFa yourpath2/human/GRCh37/hg19.2bit
GN20GG_masked_autoXY_promoters_merged_PLUS.fa
-seqList=GN20GG_masked_autoXY_promoters_merged_PLUS.gff

#repeat for file containing hits on the minus strand
```

The Python script `reverse_complement_fasta.py` was used to reverse-complement the FASTA sequences on the minus strand [2]. The script was invoked as follows:

```
python ReverseComplementFasta.py
   GN20GG_masked_autoXY_promoters_merged_MINUS.fa >
GN20GG_masked_autoXY_promoters_merged_MINUS_REVERSE_Complement
   .fa
```

The fasta files on the plus and minus strand were combined using the cat command, lowercase letters converted to uppercase using seqret, and sequences collapsed into a unique set with `fastx_collapser`.

```
#combine
cat GN20GG_masked_autoXY_promoters_merged_PLUS.fa
GN20GG_masked_autoXY_promoters_merged_MINUS_REVERSE_Complement
   .fa >
GN20GG_masked_autoXY_promoters_merged_TOTAL.fa

#convert to uppercase
seqret GN20GG_masked_autoXY_promoters_merged_TOTAL.fa
GN20GG_masked_autoXY_promoters_merged_TOTAL_UPPER.fa -
   sformat fasta -supper Y

#get unique fasta sequences
fastx_collapser <
   GN20GG_masked_autoXY_promoters_merged_TOTAL_UPPER.fa >
GN20GG_masked_autoXY_promoters_merged_PlusMinus_UNIQUE.fa
```

This yields a file containig 4,113,530 sequences.

1.3 Identifiying a consensus sequence for gRNAs that fall into promoters

The Bioconductor package Biostrings [8] was used to derive a consensus sequence from the list of FASTA sequences generated above as follows:

```
>library(Biostrings)

>promMINUS<-readDNAStringSet(
"GN20GG_masked_autoXY_promoter_minus_REVERSECOMPLEMENT.fa
   ", format="fasta")
```

```
>fm<-consensusMatrix(promMINUS)
minus<-fm[1:4,]
pwm_minus<-t(t(minus)/rowSums(t(minus)))
```

The following code was used to generate sequence logo plots [4].

```
>library(ggplot2)

>berrylogo<-function(pwm,gc_content=0.5,zero=.0001){
 backFreq<-list(A=(1-gc_content)/2,C=gc_content/2,G=
    gc_content/2,T=
  (1-gc_content)/2)
  pwm[pwm==0]<-zero
bval<-plyr::laply(names(backFreq),function(x){log(pwm[x,])
   -log(
  backFreq[[x]])})
row.names(bval)<-names(backFreq)
p<-ggplot2::ggplot(reshape2::melt(bval,varnames=c("nt","
   pos")),
  ggplot2::aes(x=pos,y=value,label=nt))+
    ggplot2::geom_abline(ggplot2::aes(slope=0), colour = "
       grey",size=2)+
    ggplot2::geom_text(ggplot2::aes(colour=factor(nt)),
       size=8)+
    ggplot2::theme(legend.position="none")+
    ggplot2::scale_x_continuous(name="Position",breaks=1:
       ncol(bval))+
    ggplot2::scale_y_continuous(name="Log relative
       frequency")
  return(p)
}
```

```
#invoke the function with:
berrylogo(pwm_minus, gc_content=0.5, zero=.0001)
```

1.3.1 Reducing complexity by identifying the most significant clusters

I reasoned that it might be possible to reduce the complexity of a random library by clustering. I defined regions of interest as the 4,671,728 genomic hits of the form GN20GG that fall into or next to known promoter sequences in the human genome with a minimum of 1 bp overlap. I used LCS-HIT (Version 0.5.2) [7] to cluster those regions of interest on the basis of sequence similarity. Clusters were ranked in descending order by the number of members and the top 15 clusters extracted.

```
#cluster sequences based on sequence similarity threshold
   0.2 and use the "exact algorithm"
lcs_hit-0.5.21/lcs_hit -i GN20GG_masked_autoXY_promoters
_merged_PlusMinus_UNIQUE.fa -O LCSHIT_OUTPUT20G1 -c 0.2 -g
    1 &
```

LCS-HIT outputs the FASTA-identifiers only, therefore FASTA sequences were extracted using the script `RetrieveFasta.pl`, downloaded from reference [1].

```
#!/usr/bin/perl

#usage = path_to/RetrieveFasta.pl IDFIle FastaLibrary >
   OUTPUT

if ($#ARGV !=1) {
        print "usage: RetrieveFasta.pl IDFile FastaLibrary
            > outputfile\n";
        exit;
}
```

```
use strict;
use Bio::DB::Fasta;

my $database;
my $fasta_library = $ARGV[1]; #opens second user-supplied
   variable as the
#library file
my %records;

open IDFILE, "<$ARGV[0]" or die $!; #opens first user-
   supplied variable as
#the ID file
open OUTPUT, <STDOUT>;

# creates the database of the library, based on the file
$database = Bio::DB::Fasta->new("$fasta_library") or die "
   Failed to create
Fasta DP object on fasta library\n";

# now, it parses the file with the fasta headers you want
   to get
while (<IDFILE>) {

      my ($id) = (/^>*(\S+)/);  # capture the id string (
         without the initial ">")
      my $header = $database->header($id);
      #print "$header\n";
      print ">$header\n", $database->seq( $id ), "\n";
      print OUTPUT  ">$header\n", $database->seq( $id ),
         "\n";
}

#remove the index file
unlink "$fasta_library.index";

#close filehandles
close IDFILE;
close OUTPUT;
exit;
```

This step is exemplified here for the top cluster (Cluster190).

```
./RetrieveFasta.pl Cluster190
   GN20GG_masked_autoXY_promoters_merged
_PlusMinus_UNIQUE.fa > Cluster190.fa;
```

For each of the top 15 clusters consensus sequences were computed using the Bioconductor package Biostrings.

```
library(Biostrings)
Clust190<-readDNAStringSet("Cluster190.fa", format="fasta
   ")
consensusString(Clust190, ambiguityMap=IUPAC_CODE_MAP,
   threshold=0.19,
shift=0L, width=NULL)
```

The threshold option allows the user to define the percentage threshhold at which a given nucleotide will be incorporated into the consensus sequence at a given position. The threshold was varied between 0.14 and 0.25 so as to yield consensus sequences representing roughly 10^4, 10^5 and 10^6 different sequences respectively. The complexity, or number of sequences represented by a cluster, was computed using the following C script from [3], `AllSequencesFromConsensus.c`.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#define RUN(base) copy[index]=base; recurs(seq,copy,index
   +1,len)

static void recurs(const char *seq,char* copy,int index,
   const int len)
   {
   if(index==len)
      {
      fwrite(copy,sizeof(char),len,stdout);
      fputc(’\n’,stdout);
      }
   else
      {
      switch(toupper(seq[index]))
           {
           case ’A’:case ’T’: case ’G’: case ’C’:  RUN(seq
              [index]); break;
           case ’N’:RUN(’A’);RUN(’T’);RUN(’G’);RUN(’C’);
              break;
           case ’W’:RUN(’A’);RUN(’T’);break;
           case ’S’:RUN(’G’);RUN(’C’);break;
           case ’B’:RUN(’T’);RUN(’G’);RUN(’C’);break;
           case ’D’:RUN(’A’);RUN(’T’);RUN(’G’);break;
           case ’H’:RUN(’A’);RUN(’T’);RUN(’C’);break;
           case ’V’:RUN(’A’);RUN(’G’);RUN(’C’);break;
           case ’K’:RUN(’G’);RUN(’T’);break;
           case ’M’:RUN(’A’);RUN(’C’);break;
           case ’R’:RUN(’A’);RUN(’G’);break;
           case ’Y’:RUN(’C’);RUN(’T’);break;
           default: fprintf(stderr,"Bad base in %s (%c)\n
              ",seq,seq[index]);
           exit(EXIT_FAILURE);break;
           }
      }
   }

int main(int argc,char** argv)
   {
   char* seq;
   int len,i;
   if(argc!=2)
      {
      fprintf(stderr,"Usage : %s <dna>",argv[0]);
      return EXIT_FAILURE;
      }
   seq=argv[1];
   len=strlen(seq);
   char* copy=malloc((len+1)*sizeof(char));
   if(copy==NULL)
      {
      fprintf(stderr,"Out of memory\n");
      exit(EXIT_FAILURE);
      }
   copy[len]=’\0’;
   recurs(seq,copy,0,len);
   free(copy);
   return 0;
   }
```

```
#compile with
gcc -o AllSequencesFromConsensus AllSequencesFromConsensus
   .c

#run the script and pipe the output to line-count  (here,
   shown for the
consensus sequence of Cluster190_T20)
./AllSequencesFromConsensus RRRGRGRRRRRGRRGRRGV | wc -l
```

Next, the ”GenomeSearch” function of the Biostrings package (see Section 1.1) was used to run each of the consensus sequences for each of the top 15 clusters against the masked human genome.

```
#store the sequences that should be run against the genome
    in a
DNAStringSet object dict0, here shown for Cluster190 only
>dict0<-DNAStringSet(c("GRRRGRGRRRRRGRRGRRGVNGG", "
   GRRRRRRRRRRRRRRGRRGVNGG",
"GVRRRRRRRRRRRRRRRRSVNGG", "GVRRRRRRRRRRRRRRRRSVNGG",
"GVVRRRRRRRRRRRRRRVVVNGG", "GVVVRRRRRRRRRRRRVVVVNGG"))

#provide an identifier for each of the consensus sequences
   ,
#here, cluster number and chosen threshold were used,
   again shown for Cluster190 only
names(dict0)<-c("Cluster190_T20", "Cluster190_T19", "
   Cluster190_T18", "Cluster190_T17", "Cluster190_T16", "
   Cluster190_T15")

#invoke as follows
GenomeSearch_masked(dict0, outfile="
   Top15Clusters_masked_allregions.txt")
```

Because CRISPRs guide RNAs are known to tolerate up to 3 mismatches in their target sites [6], I reasoned that counting only exact matches probably underestimates the true number of target sites of any given sequence. Thus, the analysis was repeated allowing for an arbitrary number of up to 3 mismatches (gRNAs are known to tolerate more mismatches, especially towards the 5' end of the protospacer sequence. However, position of the mismatch in the gRNA was not taken into account here).

```
# allowing for a maximum of three mismatches
# change relevant parameter in the "GenomeSearch" function
    of the code

plus_matches <- matchPattern(pattern, subject, max.
   mismatch=3, min.mismatch=0, fixed=c(pattern=FALSE,
   subject=TRUE))

#invoke with
runAnalysis_masked_3mismatch(dict0, outfile="
   Top15Clusters_masked_allregions_3mismatch.txt")
```

UPDATE UPDATE UPDATE UPDATE

Output files from both scripts were processed as follows and the results stored in Table 3.1 on p. 95 of the PhD Thesis.

```
# retrieve hits for autosomes and sex chromosomes only
grep "Cluster190_T20" Top15Clusters_masked_allregions.txt
    | grep -v ’random’ | grep -v ’hap’ | grep -v ’chrUn’ |
    grep -v chrM | awk ’{print $1 "\t" $2 "\t" $3}’ >
   Cluster190_T20_autoXY

#count number of hits that fall into promoter regions
intersectBed -a Cluster190_T20_autoXY -b annotation_files/
   promoters_merged.bed -wa -wb | sortBed | wc -l;
```

```
#count the number of unique promoter regions hit
intersectBed -a  Cluster190_T20_autoXY -b
   annotation_files/promoters_merged.bed -wb | cut -f 5-8
   | sortBed | uniq | wc -l;
```

I defined the targeting efficiency for each of the possible consensus sequences of a given cluster by dividing the number of promoter hits by the total number of unique sequences (complexity) of the consensus. For each cluster the consensus sequences with the highest targeting sequence was chosen. The 15 top clusters were then ranked by targeting efficiency and the top 6 clusters chosen for library preparation.

Chapter 2

Design of theEMT5000 library

2.1 Design of the EMT5000 library

A list of genes known to be involved in EMT was taken from DeCraene et al. [5]. Regions of interest within the promoter of these genes were identified manually by inspection of chromatin marks in the UCSC genome browser. The following genomic regions where chosen as target sites for the design of guide RNAs and saved in the file ”EMT-genepromoter-comprehensive.gff”:

chr1	170626538	170637878	PRRX1
chr2	145272896	145282545	ZEB2
chr2	145310788	145311630	ZEB2
chr6	166578775	166584033	Brachyury (T gene)
chr6	166586466	166588249	Brachyury (T gene)
chr7	19155427	19162115	TWIST1
chr8	49831094	49838789	SLUG
chr10	31549929	31552360	ZEB1
chr10	31603262	31611019	ZEB1
chr14	61113258	61126351	SIX1
chr14	95235188	95236645	GSC (Goosecoid)
chr16	68765472	68768900	Cdh1
chr16	68770501	68779468	Cdh1
chr16	86596822	86601033	FOXC2
chr18	25616470	25616815	Cdh2
chr18	25753143	25759002	Cdh2
chr18	25763319	25764152	Cdh2
chr18	25783828	25784775	Cdh2
chr18	52966479	52970132	TCF4
chr18	52983584	52991765	TCF4
chr18	52994740	52997201	TCF4
chr18	53067559	53071402	TCF4
chr18	53072594	53073776	TCF4
chr18	53087724	53090359	TCF4
chr18	53176467	53178784	TCF4
chr18	53252816	53257791	TCF4
chr18	53259747	53260381	TCF4
chr18	53301547	53303603	TCF4
chr19	1631468	1633670	E47/TCF3
chr19	1646514	1653855	E47/TCF3
chr19	1655335	1656204	E47/TCF3
chr19	1660918	1661677	E47/TCF3
chr20	48592707	48600991	SNAIL1
chrX	56258091	56260688	KLF8

All gRNAs falling into these regions were then found using:

```
intersectBed -a GN20GG_masked_autoXY.gff -b
   EMT_genepromoter_comprehensive.gff
```

```
-f 1 -wa -wb > GN20GG_masked_autoXY_EMT_genepromoter_comprehensive.
   gff
```

Guide RNAs that did not align uniquely to the genome where then removed

```
##for alignment need to first convert gff file into fasta format
##separate + and - strand
cut -f 1,2,3,4 GN20GG_masked_autoXY_EMT_genepromoter_comprehensive.
   gff | grep '+' | awk '{print "chr" $1 ":" ($2-1) "-" $3}' >
   GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_PLUS.2bit;

cut -f 1,2,3,4 GN20GG_masked_autoXY_EMT_genepromoter_comprehensive.
   gff | grep -v '+' | awk '{print "chr" $1 ":" $2 "-" $3}' >
   GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_MINUS.2bit;

##convert to FASTA
twoBitToFa ~/human/GRCh37/hg19.2bit
   GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_PLUS.fa
   seqList=GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_PLUS
   .2bit;

twoBitToFa  ~/human/GRCh37/hg19.2bit
   GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_MINUS.fa
   seqList=GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_MINUS
   .2bit;

##create reverse complement of guide RNAs on the minus strand

python ~/anna_data/gRNA_library_design/ReverseComplementFasta.py
   GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_MINUS.fa >
   GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_MINUS_RC.fa;

##combine
cat GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_PLUS.fa
   GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_MINUS_RC.fa >
    GN20GG_masked_autoXY_EMT_genepromoter_comprehensive.fa;
```

The PAM sequence was removed using trimming.py and alignment to the genome without the PAM (as GN19).

```
bwa aln -n 0 -o 0 -l 10 -N -I ~/path_to/GRCh37/human_GRCh37.tmp
GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_noPAM.fa >
GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_noPAM.sai;

bwa samse -n 10000 ~/path_to/GRCh37/human_GRCh37.tmp
GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_noPAM.sai
GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_noPAM.fa >
GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_noPAM.sam;

samtools view -S -q1
GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_complete_noPAM.
    sam | cut -f  1 | sort | uniq | wc -l

GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_complete_noPAM_unique_strand
    .bed
```

This file contains 5086 sequences, i.e. 5086 gRNA sequences of the form GN19 that align uniquely to the genome and are followed by an NGG PAM were found in the regions of interest.

For cloning into the gRNA vector pgRNA-pLKO.1 (see Methods) by Gibson cloning, vector-derived sequences were attached to either end of the gRNA sequence. To this end the FASTA files for the final set of 5086 unique gRNAs were retrieved and sequences appended as follow:

```
sed ’n; s/$/GTTTTAGAGCTAGAAATAGCAAGTTAAAATAAGGCT/’
    GN20GG_masked_autoXY_EMT_genepromoter_
    comprehensive_complete_noPAM_unique_strand_PAMremoved.fa | sed ’n
    ; s/^/TCTTGTGGAAAGGACGAAACACC/g’ | paste - - | awk ’{print $2 "\t
    " $1}’ > EMT_guides_Custom_Array.txt
```

A pool of sequences of the form 5’-TCTTGTGGAAAGGACGAAACACC-GN19-GTTTTAGAGCTAGAAATAGCAAGT
3’, where GN19 donates the 5086 different guide sequences was then obtained from Custom Array Inc.

Chapter 3

Analysis of screen with the EMT5000 library and stable cell lines

3.1 Bioinformatic analysis of screens with EMT5000 control library and stable cell lines

3.1.1 Read trimming and QC

sequenced on the HiSeq. Reads have the following general structure:

```
NNNNNNNAAGTATTTCGATTTCTTGGCTTTATATATCTTGTGGAAAGGACGAAACACCG-N19-
   GTTTTAGAGCTAGAAATAGCAAGTTAAAATAAGGCTAGTCCGNNNNNNN
```

whereby the first 7 N bases are the 5' barcode, followed by the plasmid 'stuffer', GN19 denotes the gRNA sequence from the EMT5000 library, which is followed by another plasmid sequence and the last 7 N bases are the 3' barcode. The 5' and 3' barcodes serve as unique molecular identifiers (UMIs), allowing counting of original gRNA sequences extracted from lentivirus-infected cells by removing PCR-amplification bias.

Sequences from Lane1 and Lane2 of the flow cell were combined using the unix 'cat' command. Next, the 5' barcode was extracted from the reads using cutadapt (version 1.2.1), requiring a minimum overlap of 35 bp between the plasmid stuffer sequence and the read with a maximum error of 10 % and a minimum barcode length of 7 bp.

```
Command line parameters:
-a AAGTATTTCGATTTCTTGGCTTTATATATCTTGTGGAAAGGACGAAACACC -O 35 -m 7
```

The 3' barcode was retrieved from the read in an analogous way:

```
Command line parameters:
-g GTTTTAGAGCTAGAAATAGCAAGTTAAAATAAGGCTAGTCCG -O 28 -m 7
```

Finally the gRNA sequence was extracted from the read, requiring a minimum length of 2 bp (to discard reads that contain no gRNAs and are derived from primer dimers):

```
Command line parameters:
cutadapt -g AAGTATTTCGATTTCTTGGCTTTATATATCTTGTGGAAAGGACGAAACACC -a
   GTTTTAGAGCTAGAAATAGCAAGTTAAAATAAGGCTAGTCCG -n 2 -m 2 -O 10
```

Next, the barcode and gRNA reads were quality-filtered using $fastq_{q}uality_{f}ilter$ from the fastx-toolbox (version XXXXX). Reads where any base has a Quality score of less than 20 were discarded.

```
Command line parameters: -Q33 -q 20 -p 1
```

Subsequently files were converted from fastq to fasta format using $fastq_to_fasta$.

```
fastq_to_fasta -Q33 -n -i  inputfile.fastq -o outputlfile.fasta
```

gRNA reads were subsequently aligned back onto the indexed EMT5000 reference library using bwa (version: 0.6.2-r126), allowing 2 mismatches and default open gaps (1 indel).

```
Command line options:
bwa aln -n 2 -l 5 -N -I
bwa samse -n 10000
```

The indexed EMT5000 library file used as a reference for alignment was derived from the file $GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_complete_noPAM_unique_strand_PAMremoved.fa$ (see section 2.1) using bwa index:

```
bwa index
    GN20GG_masked_autoXY_EMT_genepromoter_comprehensive_complete_noPAM_unique_strand
    .fa
```

Following alignment using bwa, reads that mapped uniquely to the forward strand were extracted using samtools (version XXXX) as follows:

```
samtools view -F 20 -q 1 -S aligned_gRNA.sam  > gRNA_uniquely_mapped
    .sam
```

To ask how many different gRNAs from the library were sequenced in each sample

```
samtools view -F 20 -S  aligned_gRNA.sam | cut -f 3 | sort | uniq |
    wc -l
```

For subsequent analysis it was necessary to construct a tab-separated file of the following format: want to get a tab-separated 3 column file containing for each read the FASTA identifier (read-ID), gRNA chr:start-end and barcode.

to get the gRNA:

```
samtools view -F 20 -q 1 -S aligned_gRNA.sam | awk '{print$1 ".\t"
    $3}'  > gRNA_uniquelymapped
```

to get the FASTA identifier:

```
samtools view -F 20 -q 1 -S aligned_gRNA.sam | awk '{print$1 "."}' >
     gRNA_uniquelymapped_readID
```

The files containing the quality-filtered 5’ and 3’ barcode (UMI sequence) generated above, were modified as follows (pseudocode shown for 5’ bc only):

```
for i in *_5bc_Q20.fasta
do cat "$i" | paste - - | awk -F ’ ’ ’{print $1 ".\t" $3}’ > "${i%
    _5bc_Q20.fasta}"_5bc_Q20_point.fasta;
done
```

Next, only the barcodes associated with gRNAs that aligned uniquely were retrieved from the barcode file using grep:

```
for i in *_5bc_Q20_point.fasta
do grep -wFf "${i%_5bc_Q20_point.fasta}"gRNA_uniquelymapped_readID "
    $i" > "${i%_5bc_Q20_point.fasta}"
    gRNA_uniquelymapped_readID_with_5bc;
done
```

For each read, identified by its readID, the gRNA sequence and 5’ and 3’ barcodes were combined into a single file. The three files were joined (finding the union) using the JOIN command, which requires the files to be sorted in the following way:

```
for i in *gRNA_uniquelymapped;
do awk -F "\t" ’{print ">" $1 "\t" $2}’ "$i"| sort -k 1b,1 > "${i%}"
    _sorted;
done

for i in *gRNA_uniquelymapped_readID_with_3bc;
do sort -k 1b,1 "$i" > "${i%}"_sorted;
done

for i in *gRNA_uniquelymapped_readID_with_5bc; do sort -k 1b,1 "$i"
    > "${i%}"_sorted;
done
```

Next, the three files were joined as follows:

```
for i in *gRNA_uniquelymapped_sorted;
do join "$i" "${i%gRNA_uniquelymapped_sorted}"
    gRNA_uniquelymapped_readID_with_5bc_sorted | join - "${i%
    gRNA_uniquelymapped_sorted}"
    gRNA_uniquelymapped_readID_with_3bc_sorted > "${i%
    gRNA_uniquelymapped_sorted}"
    gRNA_uniquelymapped_readID_gRNA_5bc_3bc_length14;
done
```

This yields a file containing for each uniquely mapped read its readID, the gRNA it mapped to (chr:start-stop) and the UMI found in the read (of length exactly 14 bp).

3.1.2 Deriving gRNA counts from UMI-barcodes without PCR error correction

The gRNA counts were derived by counting the number of times each gRNA occurs together with a different UMI in the sequencing data.

The script collapse-barcodes.py was run using a maximum edit distance of 0, i.e. not correcting any PCR errors that might have occurred in the barcode.

```
python collapse_barcodes.py Samplefilename 0
```

```
# collapse_barcodes.py v 1.1
# Anna Koeferle Dec 2015, UCL
# adapted from CollapseTCRs.py v1.2 by James Heather and Katharine
    Best

### BACKGROUND ###
# Makes use of random molecular barcode sequences to error-correct
    high-throughput sequencing data.
# The barcodes are random nucleotides (N) added to the library
    amplicon prior to amplification.
# Instead of counting reads/molecules (which is problematic due to
    PCR amplification bias), the co-occurence of different barcodes
    with a sequence of interst is counted (count barcodes instead of
    reads).
# This allows us to correct for PCR amplification bias and PCR error
    /sequencing error.
# The script does the following:
# 1. Import input file of the form (c1) identifier (c2) gRNA chr:
    start-end OR DNA sequence (c3) barcode (varying lengths)
```

```
# 2. The script then sorts according to (c2) gRNA.
# 3. Within each gRNA, the barcodes are grouped according to
   sequence similarity
# If a barcode is within x edit distance from the previous one group
    them together
# 4. barcodes in groups are re-written/corrected to the sequence of
   the most common member of the group
# 5. export a file that has the corrected barcode sequence in c4

### INPUT ###
# takes a tab-delimited file consisting of 3 columns:
# column1: Read identifier
# column2: gRNA from the library read was aligned back to, either as
    DNA sequence or coordinates of the form chr:start-end
# column3: barcode (various lengths N)
# Run: python CollapseBarcodess.py FILENAME.n12

### OUTPUT ###
### Outputs a gRNA frequency file = gRNA and its counts
## can output a .dict file = the collapsed dictionaries used to
   calculate the gRNA frequencies, by default commented out

### USAGE ###
# python CollapseBarcode.py INPUTFILENAME MAX_NUMBER_OF_ERRORS

#### PACKAGES ####

from __future__ import division
import collections as coll
import sys
import Levenshtein as lev
import re
from operator import itemgetter
from time import time, clock
import json
import signal
from Bio.Seq import Seq
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord
from StringIO import StringIO
import numpy as np
import matplotlib.pyplot as plt
import pylab as pylab

##### Setting functions and empty dictionaries for later use
def dodex():
    return coll.Counter()

def breakdown(etc):
  # Used to break a given dcr_etc (i.e. what is stored in a given
     line of the inputfile) into its components
  # Splits on '|', to avoid breaking up within identifiers or fastq
     quality strings

  return re.findall(r"[\w,\<\=\>\-\;\:\?\/\.\@\# ]+", str(etc))
  # breakdown[0] = gRNA / [1] = ID

def guide_frequency(collections_dictionary):
    output=[]
    for i in collections_dictionary:
        clustered = coll.Counter()
        for k,v in collections_dictionary[i].most_common():
            clustered[k] += v
        result = str(i) + ", " + str(len(clustered)) + "\n" # len of
            dictionary returns number of key:values stores output in
            string
        output.append(result)

    return output

def remove_orphan_barcodes(diction):
    #removes barcodes that are seen only once, even after error
       correction
    dict_morethan1 = coll.defaultdict(list)
    for x in diction: #loop over the gRNAs
        counter = coll.Counter() # set an empty counter
        for k,v in diction[x].most_common():  # loop over the
           barcodes and counts for each gRNA
            if v > 1:    # if the count is greater than 1
                counter[k] += v   # add the barcode and counter to
                   the counter dictionary
        dict_morethan1[x] = counter   # append to the gRNAs

    return dict_morethan1


#### Importing data from input###

total_number_of_reads = 0 # count_input_lines

if (len(sys.argv) <> 2):
    print "Missing inputfile! Usage: python
       collapse_barcodes_editdist0.py INPUTFILENAME"
    sys.exit()
else:
    seqfilename = str(sys.argv[1])    # stores name of user-supplied
        file in variable seqfilename
    distance_threshold = int(sys.argv[2]) #stores user-supplied max
       number of edit distances for barcodes to be grouped together
    start_time = time() # set time at start

    print "\nReading", str(seqfilename), "into dictionary..."
    t0 = time() # Begin timer

### TAB- DELIMITED POSITIONS IN THE INPUTFILE ARE: ###

# Read identifier -[0]-
# gRNA -[1]-
# Barcode-[2]-
```

```
    dict_seqfile = coll.defaultdict(list)   #make an empty
       collections default dictionary

    with open(seqfilename) as infile:
        for line in infile:
            total_number_of_reads += 1
            lsplit = line.strip().split(’\t’)
            readID = lsplit[0]
            gRNA = lsplit[1]
            barcode = lsplit[2] #this includes barcodes of varying
               lengths

            values = gRNA + "|" + readID
            dict_seqfile[barcode].append(values) # this now is a
               dictionary with key=barcode and value=a list of gRNA
               and readIDs separated by "|" all with the same key, e
               .g. looks like ’TGTGGGCGACGGGG’: [’chr10
               :31609045-31609068|>D00623:46:H7JVFBCXX
               :1:1101:5035:69776.’, ’chr10:31609045-31609068|>
               D00623:46:H7JVFBCXX:1:1102:10781:28677.’, ’chr10
               :31609045-31609068|>D00623:46:H7JVFBCXX
               :1:1102:9779:13200.’, ’chr10:31609045-31609068|>
               D00623:46:H7JVFBCXX:1:1103:3007:69834.’, ’chr10
               :31609045-31609068|>D00623:46:H7JVFBCXX
               :1:1104:1364:82596.’, ’chr10:31609045-31609068|>
               D00623:46:H7JVFBCXX:1:1104:15350:25118.’]

    timed = time() - t0
    print ’The total number of reads for this sample was:’ +str(
       total_number_of_reads) +’.’

#### now take the inputfile and derive something that looks like
   dcr_collapsed dictionary of Jamie Heather:

print "Counting barcodes associated with each gRNA..."

dict_collapsed = coll.defaultdict(dodex)
```

```
t0 = time() # Begin timer


for key in dict_seqfile:  #loop throught the barcodes, i.e. the key.
    like saying 'for every barcode in the dictionary':
    if len(key) >= 14:   # exclude barcodes that are less than 14
       bases in length
        for value in dict_seqfile[key]:          # loop through the
           values (gRNA | identifier)
            gRNA_seq = breakdown(value)[0] # prints only the value
               correspondign to gRNA
            dict_collapsed[gRNA_seq][key] += 1 # appends to
               dict_collapsed the gRNA sequence, the barcode = key
               and its count. In fact, does the counting

timed = time() - t0

#print'InputData collapsed into dictionary: '
#print dict_collapsed

print '\t Finished! Took', round(timed,3), 'seconds'

###### dict_collapsed looks like this with test datasset: {gRNA:
   Counter({'barcode key':count, 'barcode':count}),
#defaultdict(<function dodex at 0x103000de8>, {'chr10
   :31609167-31609190': Counter({'TTA': 1888, 'TTATTT': 985, 'AAGTAA
   ': 752, 'CCCCCG': 8, 'CTCTTC': 4, 'CTTATG': 4,

print "Removing orphan barcodes..."

t0 = time() #begin timer

try:
    dict_no_orphans = remove_orphan_barcodes(dict_collapsed)
except Exception, msg:
    print str(msg)
```

```
timed = time() - t0
print ’\t Finished! Took’, round(timed,3), ’seconds’

###### calculate the frequencies of gRNAs based on length of the dictionaries
print "Computing gRNA frequencies..."

try:
    frequency_clustered = guide_frequency(dict_collapsed)
except Exception, msg:
    print str(msg)

with open(seqfilename.split(".")[0]+ "_frequency_raw", ’w’) as outfile:
    for item in frequency_clustered:
        outfile.write(item)

########### gRNA counts for dict_threshold

try:
    frequency_no_orphans = guide_frequency(dict_no_orphans)  # run the function guide_frequency on the dictionaries generated above and store output in a string
except Exception, msg:
    print str(msg)


with open(seqfilename.split(".")[0] + "_frequency_no_orphans", ’w’) as outfile: #open a csv file to write to
    for item in frequency_no_orphans: #loop through the elements of the list, and print elements of list to file
        outfile.write(item)

print "\tDONE!"

#print "Outputting dictionaries to file...."
```

```
#json.dump(dict_clustered, open(seqfilename.split(".")[0]+str("
    _collapsed_dict_readcountnorm"), 'w'))
#json.dump(dict_no_orphans, open(seqfilename.split(".")[0]+str("
    _collapsed_no_orphans_dict_readcountnorm"), 'w'))
#json.dump(dict_threshold, open(seqfilename.split(".")[0]+str("
    _collapsed_threshold_dict_readcountnorm"), 'w'))
```

3.1.3 Assessing PCR error by plotting the number of reads per gRNA against number of different gRNA sequences

To assess whether PCR error drives barcode diversity, we treated the gRNA part of the read like a barcode and plotted the correlation between number of reads and counts (see 3.1.6). This assumes that the likelihood of introducing an error into the sequence is the same for the UMI barcodes and gRNA portions of the amplicon.

The barcode sequence was replaced with the gRNA sequence for each read to generate a tab-separated file with columns [0] readID [1] gRNA chr:start-stop [2] 'pseudo-barcode'(gRNA sequence) as follows:

```
#get gRNA sequence
cat Sample_gRNA_Q20.fasta | paste - - | awk -F ' ' '{print $1 ".\t"
    $3}' | sort > Sample_gRNA_Q20_point

#get the read ID
awk -F '\t' '{print $1}' Sample_gRNA_5bc_3bc_length14 | sort >
    Sample_gRNA_5bc_3bc_length14_read_ID

#get gRNA sequence for each readID
grep -wFf Sample_gRNA_5bc_3bc_length14_read_ID Sample_gRNA_Q20_point
     > Sample_gRNA_Q20_point_uniquely_mapped

#sort the previously generate file containing [0] readID, [1] gRNA
    chr:start-stop, [2] UMI of 5' and 3' barcode
sort -k 1b,1 Sample_gRNA_uniquelymapped_readID_gRNA_5bc_3bc_length14
     > Sample_gRNA_uniquelymapped_readID_gRNA_5bc_3bc_length14_sorted
```

```
#sort the gRNA sequence file
sort -k 1b,1 Sample_gRNA_Q20_point_uniquely_mapped >
   Sample_gRNA_Q20_point_uniquely_mapped_sorted

#join the two files on readID
join
   Sample_gRNA_Q20_aligned_2mismatches1gap_uniquelymapped_readID_gRNA_5bc_3bc_lengt
    Sample_gRNA_Q20_point_uniquely_mapped_sorted | awk -F ' ' '{
   print $1 "\t" $2 "\t" $4}' >
   Sample_gRNA_Q20_aligned_2mismatches1gap_uniquelymapped_readID_gRNA_instead_of_ba
```

3.1.4 Deriving gRNA counts from UMI-barcodes with naive PCR error correction

The script collapse-barcodes.py was run using a maximum edit distance of 4.

```
python collapse_barcodes.py Samplefilename 4
```

This means that before counting how many different barcodes are associated with each gRNA, the barcodes are collapsed into groups. Barcodes are first ranked in decreasing order based on the number of reads harbouring its sequence. The barcode with the most reads forms the first group. If the second-ranked barcode is within 4 edit-distances of this barcode it will be assumed to have originated by PCR error and will be added to the group. If the barcode differs from the group by greater then 4 edits, it will form its own group and so on. The number of groups per gRNA is the count (number of original gRNA-barcode combinations) after error correction.

3.1.5 Bayesian PCR error correction of barcoded sequencing data

A Bayesian error correction script was written by James E. Barrett to infer gRNA counts from the UMI data. The model takes into account the fact that the 14 bp UMI consists of a 5' and 3' barcode that was attached to the gRNA amplicon during 2 intial cycles of PCR during the sequencing library prep. The model infers the most likely number of initial gRNA-barcode data given the barcode sequences observed in the sequencing sample.

This Bayesian model takes as input the number of reads associated with each gRNA-UMI combination (without PCR error correction). I calculated these using the script make-csv-4Bayes.py.

```
#make_csv_4Bayes.py v 1.0
#written by Anna Koeferle, 2015
#adapted in large parts from  the script CollapseTCRs.py by Jamie Heather (can be found at https://github.com/JamieHeather/tcr-analysis/blob/master/CollapseTCRs.py)

#This script takes in a tab-separated file containing  [0] read ID [1] gRNA chr:start-end [2] barcode 5primer and 3 prime fused as input
#This script ouptuts a csv file to input into the bayesian PCR error correction script written by James E. Barrett
#The output is a csv file with columns:  [0] gRNA chr:start-end, [1] barcode (14 bp 5' and 3' barcode combined, [2] read count for this barcode gRNA combination (no error correction)

from __future__ import division
import collections as coll
import sys
import Levenshtein as lev
import re
from operator import itemgetter
from time import time, clock
import json
import signal
from Bio.Seq import Seq
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord
from StringIO import StringIO
import numpy as np
import matplotlib.pyplot as plt
import pylab as pylab


def dodex():
    return coll.Counter()
```

```
def breakdown(etc):
  # Splits on '|', to avoid breaking up within identifiers or fastq
     quality strings

  return re.findall(r"[\w,\<\=\>\-\;\:\?\/\.\@\# ]+", str(etc))
  # breakdown[0] = gRNA / [1] = ID

def read_data(seqfilename):
    dict_seqfile = coll.defaultdict(list)   #make an empty
       collections default dictionary

    with open(seqfilename) as infile:
        for line in infile:
            lsplit = line.strip().split('\t')
            readID = lsplit[0]
            gRNA = lsplit[1]
            barcode = lsplit[2]
            values = gRNA + "|" + readID
            dict_seqfile[barcode].append(values)

    return dict_seqfile

def count_barcodes(dict_seqfile):

    dict_collapsed = coll.defaultdict(dodex)

    for key in dict_seqfile:  #loop throught the barcodes, i.e. the
       key. like saying 'for every barcode in the dictionary':
        if len(key) >= 14:   # exclude barcodes that are less than
           14 bases in length
            for value in dict_seqfile[key]:          # loop through
               the values (gRNA | identifier)
                gRNA_seq = breakdown(value)[0] # prints only the
                   value correspondign to gRNA
                dict_collapsed[gRNA_seq][key] += 1 # appends to
                   dict_collapsed the gRNA sequence, the barcode =
                   key and its count. In fact, does the counting

    return dict_collapsed

#write the dictionary into a csv file

def write_dict_to_csv(dict_clustered_name, outfilename):
    outfile = open( outfilename.split(".")[0]+str("
       _no_error_correction.csv"), ’w’ )

    for key, value in dict_clustered_name.items():  # iterate
       through the coll.dictionary
        for item in value.iteritems(): #iterate through the counter
            outfile.write( str(key) + "," + ’,’.join(map(str, item))
                + ’\n’)

if (len(sys.argv) <> 2):
    print "Missing inputfile! Usage: python make_csv_james_model.py
       INPUTFILENAME"
    sys.exit()
else:
    seqfilename = str(sys.argv[1])    # stores name of user-supplied
        file in variable seqfilename
    start_time = time() # set time at start

    print "\nReading", str(seqfilename), "into dictionary..."


dict_seqfile = read_data(seqfilename)
dict_collapsed = count_barcodes(dict_seqfile)

print ’\t Finished! Writing to csv’

write_dict_to_csv(dict_collapsed, seqfilename)

print ’\t DONE!’
```

The script was run over all samples as follows:

```
for i in *length14; do python ../make_csv_4Bayes.py "$i"; done
```

The output of this script was then fed into a Bayesian error correction script.

Bayesian PCR error correction of barcoded sequencing count data script by James E. Barrett

This script was kindly contributed by James E. Barrett.

The Bayesian model infers the number of unique original barcoded gRNA molecules from noise-corrupted count data. The model estimates a corrected read count, which may be interpreted as a proxy for the original noise-free number of unique barcodes associated with a particular gRNA.

Model definition

For each gRNA we observe N barcode pairs denoted by $(\mathbf{y}_i^1, \mathbf{y}_i^2)$ where the superscript denotes the first and second barcodes and $i = 1, \ldots, N$. Elements of the d-dimensional vector $\mathbf{y}_i^\eta \in \{\mathtt{T}, \mathtt{C}, \mathtt{G}, \mathtt{A}\}^d$ where $\eta = [1, 2]$. The number of corresponding sequencing reads is denoted by $\sigma_i \in \mathbb{Z}_+$.

The model assumes that there exist Q *latent barcodes* $\mathbf{x}_1^\eta, \ldots, \mathbf{x}_Q^\eta$ from which the observed barcodes are generated in a noise corrupting stochastic process (PCR amplification errors and random barcode switching). The model further assumes that for each pair $(\mathbf{y}_i^1, \mathbf{y}_i^2)$ only one of the observed barcodes is written in terms of the latent barcode via

$$\mathbf{y}_i^\eta = \sum_{q=1}^{Q} w_{iq}^\eta \theta(\mathbf{x}_q^\eta) \quad \text{subject to} \quad w_{iq}^\eta \in [0, 1] \quad \text{and} \quad \sum_{q,\eta} w_{iq}^\eta = 1. \tag{3.1}$$

There is therefore only one non-zero value of $[\mathbf{w}_i^1, \mathbf{w}_i^2]$ that indicates which latent barcode the observed pair is associated with. The function θ represents a noise corrupting stochastic process where the status of each nucleotide site may be changed randomly with probability $\beta \in [0, 1/2]$. We can therefore write

$$p(y_{i\mu}^\eta | x_{q\mu}^\eta, \beta) = \begin{cases} (1-\beta)\delta_{y_{i\mu}^\eta x_{q\mu}^\eta} + \beta(1 - \delta_{y_{i\mu}^\eta x_{q\mu}^\eta}) & \text{if } w_{iq}^\eta = 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

for $\mu = 1, \ldots, d$. We denote the collections of $\mathbf{x}_q^\eta$, $\mathbf{y}_i^\eta$ and $\mathbf{w}_i^\eta$ by $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{W}$ respectively. The posterior is

$$p(\mathbf{X}, \mathbf{W}|\mathbf{Y}, \boldsymbol{\sigma}, \beta) \propto p(\mathbf{Y}|\mathbf{X}, \mathbf{W}, \boldsymbol{\sigma}, \beta) p(\mathbf{X}) p(\mathbf{W}) \tag{3.3}$$

with

$$p(\mathbf{Y}|\mathbf{X}, \mathbf{W}, \boldsymbol{\sigma}, \beta) = \prod_i \left[\sum_{q,\eta} w_{iq}^{\eta} p(\mathbf{y}_i^{\eta}|\mathbf{x}_q^{\eta}, \beta) \right]^{\sigma_i}. \tag{3.4}$$

Maximum entropy priors for $\mathbf{X}$ and $\mathbf{W}$ are uniform distributions so $p(\mathbf{X})$ and $p(\mathbf{W})$ are constant.

Inference of model parameters

The Maximum A Posteriori (MAP) solution of $\mathbf{W}$ is denoted by $\mathbf{W}^*$. Since only one element of $[\mathbf{w}_i^1, \mathbf{w}_i^2]$ is non-zero the expression (3.4) is maximised by selecting $\text{argmax}_{q,\eta} p(\mathbf{y}_i^{\eta}|\mathbf{x}_q^{\eta}, \beta)$ as the non-zero element.

To find the MAP solution for nucleotide μ of the latent barcode indexed by (q, η) we consider all observed barcodes that generated from it (as defined by $\mathbf{W}$). If we let n_1 and n_0 denote the total number of matches and mismatches respectively between that latent barcode and the associated observed barcodes, then the corresponding data likelihood is $(1-\beta)^{n_{q\mu}^1} \beta^{n_{q\mu}^0}$. This will be maximised if the number of matches is maximised. This is achieved selecting the most common observed nucleotide as the value for the latent nucleotide (while taking into account multiple counts).

If we let N_1 and N_0 denote the total number of matches and mismatches respectively across all of the latent barcodes and observed data then we can write

$$\log p(\mathbf{Y}|\mathbf{X}, \mathbf{W}, \beta) = N_1 \log(1-\beta) + N_0 \log \beta. \tag{3.5}$$

It is straightforward to show that the MAP estimate for beta is

$$\beta = \frac{N_0}{N_0 + N_1}. \tag{3.6}$$

The optimisation subroutine is initialised as follows:

Cluster into Q groups based on the *Hamming distance* between two barcodes (the Hamming distance is equivalent to the *edit distance*):

$$h(\mathbf{y}_i, \mathbf{y}_j) = \frac{1}{d} \sum_{\mu=1}^{d} \delta_{(1-y_{i\mu}) y_{j\mu}}. \tag{3.7}$$

The corrected read counts are inferred as follows:

For a given value of Q we denote the value of the likelihood (3.4) at the MAP parameter estimate by

$$L(Q) = p(\mathbf{Y}|\mathbf{X}^*, \mathbf{W}^*, \beta^*). \tag{3.8}$$

The *Bayes information criterion* (BIC) score is defined by

$$\mathrm{BIC}(Q) = -2 \log L(Q) + 2dQ \log N \tag{3.9}$$

where $2dQ$ is the number of free parameters in the model. The *corrected read count* is defined by

$$Q^* = \mathrm{argmin}_Q \mathrm{BIC}(Q). \tag{3.10}$$

Bayesian error correction script: The Code

This analysis was performed in R:

```
library(reshape2)

### Load and prepare a data file

# Length of barcode
D <- 7
# Load up one of the data files (needs to be in the current
   directory)
data <- read.csv("Samplename_length14_no_error_correction.csv",
   header=FALSE)
# vector of all the unique gRNA names
gRNA <- unique(data$V1)
# Total number of unique gRNAs
G <- length(gRNA)

### Generate datasets of barcodes

# Preallocate a list structure to hold the barcode datasets
Y <- vector('list',G)
```

```
# This loop goes through each gRNA, pulls out all the associated
   barcodes and puts them in a character matrix
for(mu in 1:G){
   ind <- which(data[[1]]==gRNA[mu])
   N <- length(ind)
   # Converts into character matrix (not the most elegant way...)
   Y[[mu]] <- matrix(as.vector(melt(lapply(as.character(data[[2]][
      ind]),strsplit,split=""))$value),nrow=N,ncol=2*D,byrow=TRUE)
}

### Fit model for each gRNA

# Preallocate a list of model resuls
res <- vector('list',G)

# Loop through gRNAs, for each one fit a model and get the corrected
    read count
# This can be parallelised for speed
for(mu in 1:G){
   # Begin tryCatch (catches any errors instead of stopping the loop
      )
   tryCatch({
      # Indices for barcodes matched that the current gRNA
      ind <- which(data[[1]]==gRNA[mu])
      # Vector of read counts
      counts <- data[[3]][ind]
      # Fit the model
      res[[mu]] <- fit_model(Y[[mu]], counts)
   }, error=function(e) NULL) #End tryCatch
} # End loop over gRNAs
```

This analysis calls on the following functions, named fit_model, *LL* and *hamming*, to be defined:

```
fit_model <- function(Y, counts){

   # Wrapper code for fitting a model to barcode count data. Returns
       a fitted model
   # (see LL.R) and the corrected read count.
```

```
#
# Inputs:
#    Y:       N by 2*D character matrix where each row is a
   barcode pair and
#             each element must equal T,C,G,A. D=7 is the barcode
    length.
#    counts:  numeric vector of length N containing the read
   count for each barcode pair
#
#
# Outputs:
#    model:  list where element q is the output from the LL
   function
#    rho:    corrected read count
#
#
# Copyright James Barrett 2015
# Version: 1.0.0
# Date: 9 Nov 2015
# Contact: regmjeb@ucl.ac.uk


# Total number of observed barcode pairs
N <- length(counts)
# Preallocate a list for the output
results <- list(model=vector('list',N), BIC=rep(NA,N))

# ---------- Begin loop over N -------- #
for (q in 1:N){

   # BREAK if there's only one barcode pair observed
   if (N==1){
      results$rho <- 1
      break
   }

   # Fit model using LL function
```

```
      results$model[[q]] <- LL(Y, counts, q)
      results$BIC[q] <- results$model[[q]]$BIC
      results$rho <- which.min(results$BIC) # Current best estimate
         for corrected read count

      # BREAK if there are no mismatches and q=1. This means the
         model has achieved a
      # perfect fit so we can stop.
      if ((results$model[[q]]$mismatches==0) & (q==1)){
         results$rho <- 1
         break
      }

      # BREAK if there are no mismatches.
      if (results$model[[q]]$mismatches==0){
         results$rho <- which.min(results$BIC[1:(q-1)])
         break
      }

      # Search at least until q=10, then check to see if we've hit
         the minimum BIC score.
      # The BIC is non monotonic so don't search in the last five
         values of q.
      if ((q>=10) & (which.min(results$BIC[1:q])<(q-5))) break
   }
   # ---------- End loop over N -------- #

   return(results)
}


LL <- function(Y, counts, Q){

   # Fits a model to observed count data. Returns parameter
      estimates and BIC score.
   #
   # Inputs:
```

```
#    Y:       N by 2*D character matrix where each row is a
   barcode pair and
#             each element must equal T,C,G,A. D=7 is the barcode
    length.
#    counts:  numeric vector of length N containing the read
   count for each barcode pair
#    Q:       scalar, number of latent components
#
#
# Outputs:
#    W1:     numeric vector of length N where component i
   indicates which latent component
#            the first barcode i was generated from. A value of
   zero means the second
#            barcode i was associated with a latent barcode
   instead of the first.
#    W2:     as above but for the second barcodes
#    X1:     Q by D character matrix of latent first barcodes.
   Each row is a barcode.
#    X2:     as above but for the second barcodes
#    matches: total number of matches between barcodes and
   associated observed barcodes
#    mismatches: total number of mismatches
#    beta:   Noise hyperparameter
#    BIC:    Bayes information criterion (used for model
   selection)
#
# Example:
# Y <-  matrix(c("T","C","C","C","C","C","C","G","T","G","A","T
   ","C","T",
# "T","G","G","T","G","G","T","G","A","A","A","G","C","A",
# "T","G","T","T","G","G","T","C","T","C","C","C","G","T"), 3,
   14, byrow=T)
# counts <- c(1,3,1)
# Q <- 1
# result <- LL(Y, counts, Q)
#
```

```
#
# Copyright James Barrett 2015
# Version: 1.0.0
# Date: 9 Nov 2015
# Contact: regmjeb@ucl.ac.uk

D <- 7          # Length of barcode
N <- nrow(Y)    # Total number of pairs

# Split matrix Y into two
Y1 <- Y[,1:7]     # First 7bp barcodes
Y2 <- Y[,8:14]    # Second barcodes


#----------------------------------------------------------------#
# Initial clustering (used to define initial guesses for the
   latent barcodes)
#----------------------------------------------------------------#

# Generate N by N matrix of pariwise Hamming distances for Y1
H <- hamming(Y1)
# Cluster hierarchically (use plot(fit) to visualise)
fit <- hclust(as.dist(H), method = ’ward.D’)
# Use the clustering to split into Q clusters
W1.new <- cutree(fit, k=Q)

H <- hamming(Y2)
fit <- hclust(as.dist(H), method = ’ward.D’)
W2.new <- cutree(fit, k=Q)

# For speed (only marginal gain)
count.matrix1 <- vector(’list’,D)
count.matrix2 <- vector(’list’,D)
r <- c("T","C","G","A")
for(b in 1:D){
   count.matrix1[[b]] <- matrix(0,N,4)
   ind <- Y1[,b]=="T"
   count.matrix1[[b]][ind,1] <- counts[ind]
   ind <- Y1[,b]=="C"
   count.matrix1[[b]][ind,2] <- counts[ind]
   ind <- Y1[,b]=="G"
   count.matrix1[[b]][ind,3] <- counts[ind]
   ind <- Y1[,b]=="A"
   count.matrix1[[b]][ind,4] <- counts[ind]

   count.matrix2[[b]] <- matrix(0,N,4)
   ind <- Y2[,b]=="T"
   count.matrix2[[b]][ind,1] <- counts[ind]
   ind <- Y2[,b]=="C"
   count.matrix2[[b]][ind,2] <- counts[ind]
   ind <- Y2[,b]=="G"
   count.matrix2[[b]][ind,3] <- counts[ind]
   ind <- Y2[,b]=="A"
   count.matrix2[[b]][ind,4] <- counts[ind]
}


#---------------------------------------------------------------#
# MAP barcode and W solutions
#---------------------------------------------------------------#

MAX <- 10       # Maximum number of iterations to find X and W
counter <- 0    # Count how many iterations have been done so far

# This while loop will find X, then W and interatively update
   their values until they
# converge to stable values.

while (counter < MAX){

   # Allocate Q by D matrix of latent barcodes for first barcode
   X1 <- matrix(0,Q,D)
   for (b in 1:Q){
      for(d in 1:D){
```

```
      # This line computes the most common nucleotide in all
         barcodes that were
      # generated from latent barcode b (as specified by the
         value of W1. Read
      # counts are taken into account.
      X1[b,d] <- r[which.max(colSums(count.matrix1[[d]][W1.new
         ==b,,drop=FALSE]))]
   }
}

X2 <- matrix(0,Q,D)
for (b in 1:Q){
   for(d in 1:D){
      X2[b,d] <- r[which.max(colSums(count.matrix2[[d]][W2.new
         ==b,,drop=FALSE]))]
   }
}

# Next update the values of W
W1.old <- W1.new   # Keep track of the old values so we can
   test if a stable solution has been reached
W2.old <- W2.new
for (i in 1:N){

   # H1 and H2 are vectors of edit distances between barcode i
       and the latent barcodes
   # Equivalent to Hamming distance in this case.
   H1 <- colSums(Y1[i,]!=t(X1))
   H2 <- colSums(Y2[i,]!=t(X2))

   # Test whether the first or second barcode has a smaller
      minimum edit distance for observation i
   if(min(H1)<=min(H2)){
      # If the first barcode is closer then W1.new[i] tells us
          which latent barcode i comes from
      W1.new[i] <- which.min(H1)
      # The zero in W2.new[i] means i belongs to the one of
```

```
            the first barcodes
         W2.new[i] <- 0
      } else {
         W2.new[i] <- which.min(H2)
         W1.new[i] <- 0
      }
   }

   # Stop if the solutions aren't changing anymore
   if (identical(W1.old,W1.new) & identical(W2.old,W2.new)) break
   counter <- counter + 1
}
W1 <- W1.new
W2 <- W2.new


#----------------------------------------------------------------#
# MAP beta and BIC score
#----------------------------------------------------------------#

# Here we count the total number of nucleotide matches and
   mismatches

N.match <- numeric(N)
for (i in seq(1,N)){

   # Only cound matches between the observed and latent barcodes
      that have been associated with each other
   if(min(H1)<=min(H2)){
      N.match[i] <- sum(Y1[i,]==X1[W1[i],])
   } else {
      N.match[i] <- sum(Y2[i,]==X2[W2[i],])
   }
}

matches <- sum(N.match*counts)
mismatches <- D*sum(counts)-matches
```

```
   beta <- mismatches/(matches+mismatches) # Noise hyperparameter

   # Log likelihood
   LL <- matches*log(1-beta) + mismatches*log(beta)
   # Bayes information criterion score
   BIC <- -2*LL + (2*D)*Q*log(N)

   # Return a list with any relevant quantities
   return(list(W1=W1,W2=W2,X1=X1,X2=X2,matches=matches,mismatches=
      mismatches,beta=beta,BIC=BIC))
}

hamming <- function(Y){

   # Computes the pairwise Hamming distance between a matrix of
      genomic DNA sequences
   #
   # Inputs:
   #    Y:       N by D character matrix where each row is a barcode
       pair and
   #             each element must equal T,C,G,A.
   #
   # Outpouts:
   #   H:       An N by N matrix of pairwise Hamming distances
   #
   #
   # Copyright James Barrett 2015
   # Version: 1.0.0
   # Date: 9 Nov 2015
   # Contact: regmjeb@ucl.ac.uk

   D <- 7
   N <- nrow(Y)

   YT <- Y=="T"
   HT <- YT %*% t(YT)
```

```
  YC <- Y=="C"
  HC <- YC %*% t(YC)

  YG <- Y=="G"
  HG <- YG %*% t(YG)

  YA <- Y=="A"
  HA <- YA %*% t(YA)

  H <- 1- (HT + HC + HG + HA)/D

  return(H)
}
```

3.1.6 Diagnostic plot: Number of UMI-corrected counts versus number of reads per gRNA

Calculating the number of reads per gRNA

To calculate the number of reads per gRNA for each sample, the following code was used:

```
for i in *length14; do python ../Reads_per_gRNA.py "$i"; done
```

```
# Reads_per_gRNA.py v 1.0
# Anna Koeferle Nov 2015, UCL
# Reads_per_gRNA.py counts the number of reads associated with each
  gRNA.
# This script takes the a tab-separated file as input:
# the Inputfile has the following columns: [0] read ID [1] gRNA chr:
  start-stop [2] 14 nt barcode
# the output is a csv file: [0] gRNA chr:start-stop , [1] number of
  reads

from __future__ import division
import collections as coll
import sys
import re
import numpy as np
import matplotlib.pyplot as plt
import pylab as pylab

####### Functions ######

def read_data(seqfilename):
    dict_seqfile = coll.defaultdict(list)   #make an empty
       collections default dictionary

    with open(seqfilename) as infile:
        for line in infile:
            lsplit = line.strip().split(’\t’)
            readID = lsplit[0]
            gRNA = lsplit[1]
            barcode = lsplit[2]
            dict_seqfile[gRNA].append(readID)
    return dict_seqfile


def write_dict_to_csv(dict_name, outfilename):
    with open(outfilename, ’w’) as outfile:
        for key in dict_name:
            outfile.write(str(key) + ’, ’ + str(len(dict_name[key]))
                + ’\n’)


def sanity_check(dict_seqfile):
    sanity = {}  ### calculate total number of reads in sample and
       crosscheck
    for key in dict_seqfile:
        sanity[key] = len(dict_seqfile[key])

    sumcheck = 0
    for key in sanity:
        sumcheck += sanity[key]
    return sumcheck

######Script#####

if (len(sys.argv) <> 2):
    print "Missing inputfile! Usage: python Reads_per_gRNA.py
       INPUTFILENAME"
    sys.exit()
else:
    seqfilename = str(sys.argv[1])
    outfilename = seqfilename.split(".")[0]+str("_read_count")

    print "\nReading", str(seqfilename), "into dictionary...and
       printing to csv..."

    dict_seqfile = read_data(seqfilename)

    write_dict_to_csv(dict_seqfile, outfilename)

    total_number_of_reads = sanity_check(dict_seqfile)

    print "Done! The total number of reads for this sample were: " +
        str(total_number_of_reads)
```

Wrapping gRNA counts of all samples into a table

The output of the collapse-barcodes.py script was formatted into a table as follows (The Bayesian model outputs a csv file of counts per gRNA for each sample):

```
# MakeTable_from_counts.py v 1.0
# Anna Koeferle OCt 2015, UCL
# This script takes the output of collapse_barcodes.py (either raw
   or no_orphans table) and formats the output into a dataframe for
   use with PCA and mageck scripts

import numpy as np
import pandas as pd
import fileinput
import sys
from time import time, clock
import pybedtools
from pybedtools import BedTool

####Global variables#####

### load library file into pybedtools BedTool object
### if using a library other than EMT5000 library, need to include
   path to library file here. Expects tab separated bed file of
   genomic target regions with associated target gene name
### e.g. chr "\t" start "\t" stop "\t" target gene name

EMT_lib = pybedtools.BedTool('/Users/anna_koeferle/Documents/UCL/
   HiSeqRun_Sept2015/gRNA_counts/EMT5000_library_regions.bed')

####Functions####

def load_file_to_dict(samplename):
    my_dict = {}
    with open(samplename, 'r') as infile:
        name = samplename
        for line in infile:
            lsplit = line.strip().split(', ')
            gRNA = lsplit[0]
            count = lsplit[1]
            my_dict[gRNA] = count
    return (my_dict, name) # tuple

# load_file_to_dict takes a filename as input and loads the
   corresponding file into a dictionary with keys: gRNA and values :
    gRNA count
# the inputfile is a comma-separated file of the form gRNA (chr:
   start-stop), count. This file is output by collapse_barcodes.py
```

```
def make_data_frame_from_dict(tuple_dict_name): #takes tuple
    my_dict = tuple_dict_name[0]
    samplename = tuple_dict_name[1]
    my_df = pd.DataFrame.from_dict(my_dict, orient=’index’)
    my_df.columns = [samplename.split("/")[-1].split("_")[0]]
    return my_df

# make_data_frame_from_dict takes a dictionary as input and makes it
    into a pandas dataframe using the keys as row indices

def Find_target_gene(my_dataframe, gRNA_library_with_genes):
    ###get index out of dataframe and into bed format
    gRNA_list =list(my_dataframe.index.values)

    gRNA_list_formatted = []

    for gRNA in gRNA_list:
        lsplit = gRNA.split (’:’)
        chrom = lsplit[0]
        startstop = lsplit[1].split(’-’)
        start = startstop[0]
        stop = startstop[1]
        gRNA_list_formatted.append( chrom + " " + start + " " + stop
            + "\n")


    gRNA_string = ’ ’.join(gRNA_list_formatted)
    gRNA_bed = BedTool(gRNA_string, from_string=True)
    gRNA_with_gene = gRNA_bed.intersect(EMT_lib, f=1, wa=True, wb=
       True)

    target_gene_name = [(f[6]) for f in gRNA_with_gene]
    my_dataframe.insert(0, "gene", target_gene_name)
    return my_dataframe


def catenate_dataframes():
    result = pd.DataFrame()  # make an empty pandas data frame

    for sample in sys.argv[1:]:   # loop over each file in the list
       supplied by the user
        new_df = make_data_frame_from_dict(load_file_to_dict(sample)
           )  # get the dataframe returned when running the
           functions load_file_to_dict and make_data_frame_from_dict
        result = pd.concat([result, new_df], axis=1)  # update the
           empty data frame with the additional sample as a separate
            column

    result = Find_target_gene(result, EMT_lib) # inserts a column of
        target gene names for the gRNAs in the index column
    result.index.name = 'sgRNA'
    return result

###Script####

if (len(sys.argv) < 2):
    print "Missing inputfiles! Usage: python MakeTable_from_counts.
       py INPUTFILE1 INPUTFILE2 ... INPUTFILEn"
    sys.exit()

else:
    print "\nReading inputfile(s) into dictionary..."
    t0 = time() # Begin timer

    dataframe_output = catenate_dataframes()

    print "\nGeneratig outputfile..."
    dataframe_output.to_csv('Dataframe.txt', sep='\t', na_rep= '0')
       # na_rep tells what to output instead of NAs need to check
       what mageck can deal with

    timed = time() - t0
    print '\t Finished! Took', round(timed,3), 'seconds'
```

Plotting counts versus number of reads

The number of reads per gRNA were plotted against the counts per gRNA, derived either without error correction, with naive PCR error correction or Bayesian PCR error correction as described above, using the following code:

```
#plot_counts_vs_number_of_reads.py v 1.0.0
# Anna Koeferle, UCL, November 2015
#takes as input a dataframe holding number of reads per gRNA, a
   dataframe holding number of counts per gRNA and a samplesheet,
   listing the samples to be processed.

#####USAGE######

# python plot_counts_vs_number_of_reads.py [Dataframe-Counts.csv] [
   Dataframe-NumberOfReads.csv] [Path/to/list_of_Samplenames]


from __future__ import division
import numpy as np
import pandas as pd
import sys
import re
import matplotlib.pyplot as plt
import pylab as pylab


####Functions####

def find_delimiter(filename, delimiters):
    #checks if file is csv or tsv and loads file into pandas
       dataframe
    for delim in delimiters:
        df_counts = pd.DataFrame.from_csv(filename, header=0, sep=
           delim, index_col=0)
        if len(df_counts.columns) > 0:
            print ’Delimiter found!’
            break
        else:
            print 'Have not found the correct delimiter yet. Still searching....'
            df_counts = None
    return df_counts

def remove_underscore_column_names(dataset):
    dataset_renamed = dataset.rename(columns=lambda x: re.sub(r"_.*", "",x))
    return dataset_renamed

def remove_duplicates_from_df(dfname):
    new_df = dfname.T.groupby(level=0).first().T
    return new_df

def plot_scatterplot(sample):
    new_df = pd.concat([df_reads[sample], df_counts[sample]], axis=1, keys=['reads', 'counts'])
    fig = plt.figure()
    plt.scatter(new_df['reads'], new_df['counts'], c='steelblue')
    plt.xlabel('number of reads', fontsize=16)
    plt.ylabel('number of counts (unique molecules)', fontsize=16)
    fig.suptitle(str(sample), fontsize=24)
    plt.savefig(str(sample) + '.png', format='png', dpi=300)
    plt.close()

#####Checking user input#####

if (len(sys.argv) <> 4):
    print "Missing inputfile(s)! Usage: python plot_counts_vs_number_of_reads.py [Dataframe-Counts] [Dataframe-NumberOfReads] [Samplesheet]"
    sys.exit()

else:
    dfname_counts = str(sys.argv[1])
    dfname_reads = str(sys.argv[2])
```

```
name_samplefile = str(sys.argv[3])

delimiters = ['\t',', ', ',']

####load the count dataframe:
df_counts = find_delimiter(dfname_counts, delimiters)
if str(type(df_counts)) == "<type 'NoneType'>":
    print 'Delimiter not found in' + str(dfname_counts) + '
       Please try again. Acceptable inputdataframes are tab-
       separated or comma-separated files.'
    sys.exit()

df_counts = remove_duplicates_from_df(
   remove_underscore_column_names(df_counts))

###load the read dataframe:
df_reads = find_delimiter(dfname_reads, delimiters)
if str(type(df_reads)) == "<type 'NoneType'>":
    print 'Delimiter not found in' + str(dfname_reads) + 'Please
        try again. Acceptable inputdataframes are tab-separated
       or comma-separated files.'
    sys.exit()

df_reads = remove_duplicates_from_df(
   remove_underscore_column_names(df_reads))

####load the samplesheet into a list:
samplefile =[]
with open(name_samplefile) as infile:
    for line in infile: #remove everything after the first
       underscoe and remove newline
        samplefile.append(re.sub(r"_.*", "", str(line.rstrip("\n
           "))))

for sample in samplefile:
    try:
        plot_scatterplot(sample)
    except Exception, msg:
        print str(msg)
```

3.1.7 Diagnostic plot: compare gRNA counts before and after PCR error correction

The below python script takes two dataframes of gRNA counts (e.g. with vs without PCR error correction) and plots one against the other.

```
#plot_counts_vs_counts.py v 1.0.0
# Anna Koeferle, UCL, November 2015
#takes as input two dataframes (column: samplename, rows: gRNA
   counts, index: gRNA as chr:start-stop) holding number of counts
   per gRNA, and a samplesheet, listing the samples to be processed


from __future__ import division
import numpy as np
import pandas as pd
import sys
import re
import matplotlib.pyplot as plt
import pylab as pylab

#####USAGE######

# python plot_counts_vs_counts_totalcountnorm.py [Dataframe-Counts-
   xaxis.csv] [Dataframe-Counts-yaxis.csv] [Path/to/
   list_of_Samplenames]

####Functions####

def find_delimiter(filename, delimiters):
    #checks if file is csv or tsv and loads file into pandas
       dataframe
    for delim in delimiters:
        df_counts = pd.DataFrame.from_csv(filename, header=0, sep=
           delim, index_col=0) #read in file using first delimiter
           in list of possible deliminters
        if len(df_counts.columns) > 0: #check if using this
           delimiter has split the file correctly into columns
            print 'Delimiter found!'
            break
        else:
            print 'Have not found the correct delimiter yet. Still
               searching....'
            df_counts = None #set the delimiter to None if it was
               incorrect
    return df_counts

def remove_underscore_column_names(dataset):
    dataset_renamed = dataset.rename(columns=lambda x: re.sub(r"_
       .*", "",x)) #removes everything in sample name after first
       occurence of an underscore
    return dataset_renamed

def remove_duplicates_from_df(dfname):
    new_df = dfname.T.groupby(level=0).first().T #removes duplicate
       colunns from df
    return new_df

def plot_scatterplot(sample):
    new_df = pd.concat([df_counts1[sample], df_counts2[sample]],
       axis =1, keys=['counts1', 'counts2'])
    fig,ax = plt.subplots() # use this to get ax object
    ax.scatter(new_df['counts1'], new_df['counts2'], c='midnightblue
       ', label = None) # plots the scatterplot
    if dfname_counts1.split('/')[-1] == 'bayesian_corrected.csv':
        ax.set_xlabel('counts (Bayesian error correction)', fontsize
           =16) # label for x axis
    elif dfname_counts1.split('/')[-1] == 'Dataframe_allsamples.txt
       ':
```

```
    ax.set_xlabel('counts (PCR error correction max 4mm)',
       fontsize=16) # label for x axis
elif dfname_counts1.split('/')[-1] == '
   Dataframe_allsamples_no_errors.txt':
    ax.set_xlabel('counts (no error correction)', fontsize=16) #
        label for x axis
else:
    ax.set_xlabel('Unknown', fontsize=16) # label for x axis

if dfname_counts2.split('/')[-1] == 'bayesian_corrected.csv':
    ax.set_ylabel('counts (Bayesian error correction)', fontsize
       =16) # label for y axis
elif dfname_counts2.split('/')[-1] == 'Dataframe_allsamples.txt
   ':
    ax.set_ylabel('counts (PCR error correction max 4mm)',
       fontsize=16) # label for y axis
elif dfname_counts2.split('/')[-1] == '
   Dataframe_allsamples_no_errors.txt':
    ax.set_ylabel('counts (no error correction)', fontsize=16) #
        label for y axis
else:
    ax.set_ylabel('Unknown', fontsize=16) # label for y axis

fig.suptitle(str(sample), fontsize=24) #figure title
xdim = ax.get_xlim() # from the x object get the range of the
   data, returns a tuple
xmin = xdim[0] # subset the tuple
xmax = xdim[1]
ydim = ax.get_ylim()
ymin = ydim[0]
ymax = ydim[1]
identity_line = np.linspace(max(xmin, ymin),min(xmax, ymax)) #
   derive a  numpy array and get min and max comparing the
   values
ax.plot(identity_line, identity_line, color="darkorange",
   linewidth=2.0, label='x = y') # plot the identity line x=y
```

```
    ax.axis([xmin,xmax,ymin,ymax]) #rescale the axes so the identity
        line goes to edge of box
    ax.legend(loc =4, frameon =False)
    plt.savefig(str(sample) + '.png', dpi=300)
    plt.close()

#####Checking user input#####

if (len(sys.argv) <> 4):
    print "Missing inputfile(s)! Usage: python
       plot_counts_vs_counts_totalcountnorm.py [Dataframe1-Counts] [
       Dataframe2-Counts] [Samplesheet]"
    sys.exit()

else:
    dfname_counts1 = str(sys.argv[1])
    dfname_counts2 = str(sys.argv[2])
    name_samplefile = str(sys.argv[3])

    delimiters = ['\t',', ', ',']

    ####load the count dataframe:
    df_counts1 = find_delimiter(dfname_counts1, delimiters)
    if str(type(df_counts1)) == "<type 'NoneType'>":
        print 'Delimiter not found in' + str(dfname_counts1) + '
           Please try again. Acceptable inputdataframes are tab-
           separated or comma-separated files.'
        sys.exit()

    df_counts1 = remove_duplicates_from_df(
       remove_underscore_column_names(df_counts1))

    ###load the read dataframe:
    df_counts2 = find_delimiter(dfname_counts2, delimiters)
    if str(type(df_counts2)) == "<type 'NoneType'>":
        print 'Delimiter not found in' + str(dfname_counts2) + '
           Please try again. Acceptable inputdataframes are tab-
           separated or comma-separated files.’
        sys.exit()

    df_counts2 = remove_duplicates_from_df(
       remove_underscore_column_names(df_counts2))

    ####load the samplesheet into a list:
    samplefile =[]
    with open(name_samplefile) as infile:
        for line in infile: #remove everything after the first
           underscoe and remove newline
            samplefile.append(re.sub(r"_.*", "", str(line.rstrip("\n
               "))))

    for sample in samplefile:
        try:
            plot_scatterplot(sample)
        except Exception, msg:
            print str(msg)
```

3.1.8 Diagnostic plot: Counts versus number of sorted cells

This script accepts three user-arguments, a dataframe of gRNA counts (c), a dataframe of numbers of sorted cells per sample (n), and a samplesheet (s) and returns a scatterplot of sorted cells versus counts with one data point per sample in the samplesheet.

```
### Cellnumber_vs_counts.py v 1.0
### Anna Koeferle Nov 2015

###USAGE###
# python Cellnumber_vs_counts.py

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import re
import matplotlib.colors as mplcolors
import matplotlib as mpl
import argparse
import sys

###Functions####

def find_delimiter(filename, delimiters):
    #checks if file is csv or tsv and loads file into pandas
       dataframe
    for delim in delimiters:
        df_counts = pd.DataFrame.from_csv(filename, header=0, sep=
           delim, index_col=0)
        if len(df_counts.columns) > 0:
            print ’Delimiter found!’
            break
        else:
            print ’Have not found the correct delimiter yet. Still
               searching....’
            df_counts = None
    return df_counts

def remove_underscore_column_names(dataset):
    dataset_renamed = dataset.rename(columns=lambda x: re.sub(r"_
       .*", "",x))
    return dataset_renamed

def remove_duplicates_from_df(dfname):
    new_df = dfname.T.groupby(level=0).first().T
    return new_df

def replace_underscore_with_dash(dataset):
    dataset_renamed = dataset.rename(index=lambda x: re.sub(r"_",
       "-",x))
    return dataset_renamed

def guess_error_correction_method():
    if args.counts.split('/')[-1] == 'bayesian_corrected.csv' or
       args.counts.split('/')[-1] == 'bayesian_corrected_new.csv':
        method = 'Bayesian error correction'
    elif args.counts.split('/')[-1] == 'Dataframe_allsamples.txt':
        method = 'Barcodes grouped if < 4MM'
    elif args.counts.split('/')[-1] == '
       Dataframe_allsamples_no_errors.txt':
        method = 'no error correction'
    else:
        method = 'unkown'
    return method

def restrict_to_samplesheet(mydataframe):
    samplefile = []
    new_df = pd.DataFrame()
    with open(args.samplesheet) as infile:
        for line in infile: #remove everything after the first
           underscroe and remove newline
            samplefile.append(re.sub(r"_.*", "", str(line.rstrip("\n
               "))))

    for sample in samplefile:
        if sample in mydataframe.index:
            print str(sample) + ' is in the dataset.'
            new_df = new_df.append(pd.Series(mydataframe.loc[sample,
                :]))  #append this index and all columns to
               dataframe
        else:
            print str(sample) + ' is NOT in the dataset.'

    return new_df


def plot_scatter(x_count, y_cellNumber, colorvar):
    fig,ax = plt.subplots() # use this to get ax object
    colors = ['midnightblue', 'darkorange']
    levels = [2, 3]
```

```
cmap, norm = mplcolors.from_levels_and_colors(levels=levels,
   colors=colors, extend=’max’)

myscatterplot = ax.scatter(x_count, y_cellNumber, c=colorvar,
   cmap=cmap, norm = norm, edgecolors=’none’, s=40) # plots the
   scatterplot

colormap=myscatterplot.get_cmap()  #get the colormap used for
   the plot
#use this to make proxy artists for the legend
circles_artist=[mpl.lines.Line2D(range(1), range(1), color=’w’,
   marker=’o’, markersize=6, markeredgewidth=0, markerfacecolor=
   item) for item in colormap((np.array([2,3])-2)/1)]

# mpl.lines.Line2D draws the circular object
# using the colors from the list comprehension
# colormap((np.array([2,3])-2)/1) # gets the colors out the
   colormap used for the plot. the array bit normalises the
   values to a range between 0-1 used for mapping

ax.set_xlabel(’Sum of gRNA counts’, fontsize=16) # label for x
   axis
ax.set_ylabel(’Number of sorted cells’, fontsize=16) # label for
    y axis

method = guess_error_correction_method()
title = fig.suptitle(’All libraries - ’ + str(method), fontsize
   =22) #figure title

xdim = ax.get_xlim() # from the x object get the range of the
   data, returns a tuple
xmin = xdim[0] # subset the tuple
xmax = xdim[1]
ydim = ax.get_ylim()
ymin = ydim[0]
ymax = ydim[1]
```

```
    identity_line = np.linspace(max(xmin, ymin),min(xmax, ymax)) #
       derive a  numpy array and get min and max comparing the
       values
    identity_plot = ax.plot(identity_line, identity_line, color="
       dimgrey", linewidth=2.0, label='x = y') # plot the identity
       line x=y

    line_artist = mpl.lines.Line2D([],[], color='dimgrey', linewidth
       =2.0, label='x = y') # draws the grey line in the legend
    ax.axis([xmin,xmax,ymin,ymax]) #rescale the axes so the identity
        line goes to edge of box

    line_legend = ax.legend(handles =[line_artist], loc = 4, frameon
       =False) #first legend is drawn inside the plot reads x= y
    # Add the legend for the line manually to the current Axes. This
        is the trick allowing us to plot two different legends!
    ax = plt.gca().add_artist(line_legend)
    # Create another legend for coloured points
    #lt.legend(handles=[line2], loc=4)

    circle_legend = plt.legend(circles_artist, ['2 cycles', '3
       cycles'], loc = "center left", bbox_to_anchor = (1, 0.5),
       frameon =False, numpoints =1)
    plt.savefig('Sorted_cells_vs_gRNA_counts_' + str(method) + '.png
       ', bbox_extra_artists=(circle_legend, title), bbox_inches='
       tight', format='png', dpi=300) ## bbox extra artist and bbox
       inches makes sure legend is not cut off figure
    plt.close()


###User input###

parser = argparse.ArgumentParser()
parser.add_argument("-c", "--counts", help="dataframe containing
   counts per gRNA")
parser.add_argument("-n", "--cellnumber", help="dataframe containing
    counts of sorted cells")
```

```
parser.add_argument("-s", "--samplesheet", help="samplesheet
   containing one sample name per line, exactly as in dataframes")
args = parser.parse_args()

if not args.counts or not args.cellnumber:
   sys.exit("Missing inputfile!")

print "Reading data...."

delimiters = ['\t',', ', ', ',']

df_counts = find_delimiter(args.counts, delimiters)
if str(type(df_counts)) == "<type 'NoneType'>":
    print 'Delimiter not found in' + str(dfname_counts) + 'Please
       try again. Acceptable inputdataframes are tab-separated or
       comma-separated files.'
    sys.exit()

df_counts = remove_duplicates_from_df(remove_underscore_column_names
   (df_counts))
df_total_counts = pd.DataFrame(df_counts.sum(axis=0), columns = ['
   TotalCount'])
df_total_counts.index.name = None

number_of_cells = find_delimiter(args.cellnumber, delimiters)
if str(type(df_counts)) == "<type 'NoneType'>":
    print 'Delimiter not found in' + str(dfname_counts) + 'Please
       try again. Acceptable inputdataframes are tab-separated or
       comma-separated files.'
    sys.exit()

cell_count = remove_duplicates_from_df(
   remove_underscore_column_names(number_of_cells))
cell_count.drop(cell_count.columns[2:], axis=1, inplace=True)
cell_count = replace_underscore_with_dash(cell_count)
cell_count.index.name = None
```

```
frames = [df_total_counts , cell_count]
df_counts_cellnumber = pd.concat(frames , axis =1)
df_counts_cellnumber = df_counts_cellnumber.dropna(axis = ’index’,
   how = ’any’) #remove any rows that contain an NA
df_counts_cellnumber = restrict_to_samplesheet(df_counts_cellnumber)
    #makes sure only samples that are in the samplesheet are plotted
    in next step

plot_scatter(df_counts_cellnumber[’TotalCount’],
   df_counts_cellnumber[’CellCount’], df_counts_cellnumber[’Cycles
   ’])

print "Done!"
```

3.1.9 Enrichment analysis: Naive approach

3.1.10 Enrichment analysis using DESeq2

Preparing Bayesian error correction output for DESeq2

DeSeq requires for each experiment a list of counts per gRNA. The data for each experiment were extracted from the Bayesian analysis output. gRNAs where 3/4 of the counts in a given experiment are 0 (or NA) are removed before enrichment analysis as follows:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import collections as coll
import re
import pylab

###Functions###

def remove_underscore_column_names(dataset):
    dataset_renamed = dataset.rename(columns=lambda x: re.sub(r"_
       .*", "",x))
    return dataset_renamed

def generate_samplefile(samplefilename):
    samplefile =[]
    with open(samplefilename) as infile:
        for line in infile: #remove everything after the first
           underscoe and remove newline
            samplefile.append(re.sub(r"_.*", "", str(line.rstrip("\n
               "))))
    return samplefile

def remove_3quarters_NAs(dataframe):
    df_with_NAs = dataframe.replace('0', np.nan)
    thresh = int(len(df_with_NAs.columns)/4*3)
    NA_removed =df_with_NAs.dropna(axis='index', thresh=thresh)
    return NA_removed

 def save_df_for_samplefile(samplefile, samplefilename): #this
    function calls all other functions
    df_samplefile = df_bayes[samplefile]
    df_samplefile = remove_3quarters_NAs(df_samplefile)
    df_samplefile = df_samplefile.fillna(0)
    df_samplefile.to_csv(str(samplefilename)+'_counts.csv')

###Script###

df_bayes = pd.DataFrame.from_csv('path_2/bayesian_corrected_counts.
   csv', header=0, sep=',', index_col=0)

samplefilename = 'path_2/samplefile.txt'
samplefile =generate_samplefile(samplefilename)

save_df_for_samplefile(samplefile, samplefilename)
```

3.1.11 Correlation of Log2Fold enrichment scores from DESeq2 between experiments

```
library ("DESeq2")

experiment_info <- read.table("path2/ThisExperiment_DeSeq2_ColData.
   csv", sep=",", header=TRUE, blank.lines.skip = TRUE, row.names =
   "Sample.name")
counts<-read.table("path2/ThisExperiment_counts_bayes_new.csv", sep
   =",", header=TRUE, na.strings="NaN", check.names=FALSE, row.names
   =1)


counts_Batch5[is.na(counts_Batch5)] <- 0  #replace NAs with 0
dds_Batch5<- DESeqDataSetFromMatrix(countData=counts_Batch5, colData
    = experiment_info_Batch5, design = ~treatment)
#converting counts to integer mode
dds_Batch5<-DESeq(dds_Batch5)

res<-results(dds_Batch5)
resOrdered <- res[order(res$padj),]
resOrdered
```

Visualising enrichment

Bibliography

[1] http://www.biostars.org/p/1195/.

[2] http://www.biostars.org/p/14614/.

[3] http://www.biostars.org/p/6219/.

[4] Charles Berry, Sridhar Hannenhalli, Jeremy Leipzig, and Frederic D Bushman. Selection of Target Sites for Mobile DNA Integration in the Human Genome. *PLoS Comput Biol*, 2(11):e157, 2006.

[5] Bram De Craene and Geert Berx. Regulatory networks definingEMT during cancer initiation andprogression. *Nat Rev Cancer*, 13(2), 2013.

[6] Prashant Mali, John Aach, P Benjamin Stranges, Kevin M Esvelt, Mark Moosburner, Sriram Kosuri, Luhan Yang, and George M Church. CAS9 transcriptional activators for target specificity screening and paired nickases for cooperative genome engineering. *Nat Biotechnol*, 31(9):833–838, August 2013.

[7] Youhei Namiki, Takashi Ishida, and Yutaka Akiyama. Acceleration of sequence clustering using longest common subsequence filtering. *BMC Bioinformatics*, 14(Suppl 8):S7, May 2013.

[8] H. Pages, P. Aboyoun, R. Gentleman, and S. DebRoy. *Biostrings: String objects representing biological sequences, and matching algorithms.* R package version 2.30.0.

[9] Herve Pages. *BSgenome: Infrastructure for Biostrings-based genome data packages*. R package version 1.28.0.