Simulating ChIP-seq experiments

Peter Humburg

October 13, 2014

1 Introduction

This document provides a brief introduction to the use of `ChIPsim` in form of a worked example. The main purpose of this package is to provide a framework for the simulation of ChIP-seq experiments. The simulation of nucleosome positioning experiments is implemented as part of the package, see 'Extending ChIPsim' for a discussion of how other types of ChIP-seq experiments can be implemented with the help of this package.

The simulation of ChIP-seq experiments can be a powerful tool to get a better understanding of ChIP-seq data and the challenges of analysing it. A first step into this direction was taken by Zhang *et al.* [1]. They developed a simulation of a transcription factor binding ChIP-seq experiment and used this do derive an appropriate background model. The simulation framework provided by `ChIPsim` goes one step further. It simulates the location of various genomic features, such as nucleosome positions or transcription factor binding sites, and the resulting sequence reads. This can then be used to test entire analysis pipelines, from read mapping to feature identification. We may be interested in testing a specific part of the pipeline, e.g., investigate the performance of different read mapping tools or different peak calling algorithms. While this can be very useful we may gain further insights into the performance of our pipeline by testing different combinations of tools to capture interactions between them. Once an analysis pipeline is established we can use the simulation to help us plan new experiments by providing an estimate of how much sequencing will be required to obtain the desired outcomes.

In the next section we will discuss the general structure of the simulation to get a better idea of how it works and where we might intervene to change its behaviour. This is followed by an example to demonstrate the simulation of a nucleosome positioning experiment on a small genome. This includes examples to show how the parameters of the simulation can be adjusted but does not cover extensions to different experiment types. Such extensions are discussed at the end of this document where a more complex example demonstrates how we could implement the simulation of a transcription factor binding simulation.

2 Overview

The simulation is divided into six stages that are illustrated below for the example of a nucleosome positioning experiment. The general structure is the same, regardless of experiment but the details are adjusted to suit the problem under consideration.

Generate feature sequence A Markov chain is used to generate a sequence of features, e.g. nucleosomes of different stability, to cover the supplied genome. Each state of the Markov chain represents a different feature type and generates a set of associated parameters.

Compute nucleosome density Each feature type is associated with a feature generating function that translates the feature sequence into a feature density, i.e., for each position

in the genome covered by the feature it computes the probability that a nucleosome is centred there.

Compute read density Using information about the length distribution of DNA fragments and the distribution of binding sites within them, the feature density is translated into a read density for each strand. The read density at a given position on a given DNA strand is proportional to the probability that a sequence read starting at that position is produced during the sequencing process.

Sample read start sites Based on the read densities computed in the previous step the location for the desired number of reads is generated.

Create read names A name is assigned to each read.

Obtain read sequence and quality Read positions are translated into the corresponding DNA sequence, qualities are assigned to each read and used to introduce errors into the read sequence.

All six stages of the simulation are carried out by the function `simChIP`. It is possible to provide intermediate results to skip some of these steps. For example, if we already have a feature density we could use that instead of generating a new one. The read sequences and qualities generated by the simulation can be exported in FASTQ format. In the default configuration of `simChIP` this is done automatically if an output file name is provided. Each stage of the simulation is carried out by a different function that is called by `simChIP` with the appropriate arguments. Individual functions can be replaced to make substantial changes to the behaviour of the simulation in cases where simply adjusting some of the function's arguments is not enough to achieve the desired result.

3 Using the nucleosome positioning simulation

With that general structure in mind we will now take a look at how we might use the ChIPsim package for a simple nucleosome positioning simulation. This section focuses on the build in simulation of a nucleosome positioning experiment. While we will see how this can be modified to adapt the simulation to a specific experiment this example mostly relies on the default settings to keep things simple.

3.1 Setup

We start by loading the ChIPsim package. To ensure that the results of the simulation are reproducible the random number generator is initialised with a fixed seed.

```
> library(ChIPsim)
> set.seed(1)
```

If we just want to run the nucleosome positioning simulation with default settings no further preparations are necessary in practice. The remainder of this section is only required because we want to run the example without relying on additional files. It also serves as an example of how the defaults can be adjusted to fit the needs of a particular experiment.

The later stages of the simulation require a reference genome to generate read sequences. For the purpose of these examples we will use a very simple (and relatively short) simulated genome sequence. The variable `chrLen` gives the length (and number) of chromosomes, we could easily generate a different genome size by changing these numbers.

```
> chrLen <- c(2e5, 1e5)
> chromosomes <- sapply(chrLen, function(n) paste(sample(DNA_BASES, n, replace = TRUE), coll
> names(chromosomes) <- paste("CHR", seq_along(chromosomes), sep="")
> genome <- DNAStringSet(chromosomes)
```

To read the genome sequence from a (FASTA formatted) file we would simply use the name of the fasta file instead of the generated sequence above.

Since the output of the simulation is a FASTQ formatted file we will also need read qualities. By default the simulation uses read quality scores from a real sequencing experiment for this purpose. Since that is not very practical for this example we will provide a function to generate read qualities instead. This will also help to demonstrate how aspects of the simulation can be adjusted for specific needs.

```
> randomQuality <- function(read, ...){
+         paste(sample(unlist(strsplit(rawToChar(as.raw(64:104)),"")),
+                                         nchar(read), replace = TRUE), collapse="")
+ }
```

The `randomQuality` function will produce read quality scores in Illumina 1.3 format. Apart from the encoding the quality scores will not be very similar to real read qualities but this is sufficient for the purpose of this example.

Another aspect of the simulation we may want to change for this example is how output is produced. By default the simulation writes the read sequences and qualities to a FASTQ file. Instead, we would like to produce a data frame that holds this information. To achieve this we have to provide a function that accepts read positions together with some other parameters and returns the `data.frame` we want.

```
> dfReads <- function(readPos, readNames, sequence, readLen, ...){
+
+         ## create vector to hold read sequences and qualities
+         readSeq <- character(sum(sapply(readPos, sapply, length)))
+         readQual <- character(sum(sapply(readPos, sapply, length)))
+
+         idx <- 1
+         ## process read positions for each chromosome and strand
+         for(k in length(readPos)){ ## chromosome
+                 for(i in 1:2){ ## strand
+                         for(j in 1:length(readPos[[k]][[i]])){
+                                 ## get (true) sequence
+                                 readSeq[idx] <- as.character(readSequence(readPos[[k]][[i]
+                                                 strand=ifelse(i==1, 1, -1), readLen=readLe
+                                 ## get quality
+                                 readQual[idx] <- randomQuality(readSeq[idx])
+                                 ## introduce sequencing errors
+                                 readSeq[idx] <- readError(readSeq[idx], decodeQuality(read
+                                 idx <- idx + 1
+                         }
+                 }
+         }
+         data.frame(name=unlist(readNames), sequence=readSeq, quality=readQual,
+                         stringsAsFactors = FALSE)
+ }
```

3.2 Simulation

Now that we have a reference genome and a way to generate read qualities we can simply call `simChIP` to simulate sequencing data. The first argument of `simChIP` is the number of reads we would like to generate, the second is the reference genome. In this case we will use the genome sequence we generated above, we could simply pass the name of a FASTA formatted file that contains the reference sequence instead. The third argument is the prefix we would like to use for the output files. For this example we will prevent file creation by setting `file = ""`. Usually it is a good idea to write results to a file, which will produce a FASTQ formatted file with the sequences of all simulated reads. Using file output will also generate additional files containing intermediate results like the underlying feature sequence or resulting read densities for later reuse. The `control` argument allows us to change the parameters of the simulation. The entire simulation process is divided into six stages and parameters for each can be set separately by providing named entries in the list passed as `control` argument. As explained above, the six stages of the simulation are:

features Generate feature sequence (for each chromosome);

bindDensity Compute binding site density;

readDensity Compute read density for both strands;

sampleReads sample from read density to determine read start sites;

readNames Create read names;

readSequence Obtain read sequence and quality.

These parameters are then passed to a function that carries out that step of the simulation. Which function is used can be controlled through the `functions` argument.

Here we will mostly use the defaults but we need to change the way read sequences and qualities are generated. We can use `defaultFunctions` to obtain the default list of functions and simply replace the one for the final step of the simulation. Instead of completely replacing a part of the simulation we sometimes just want to adjust the default behaviour slightly. In many cases that is possible by changing a parameter value. We will illustrate the process by changing the mean fragment length to 150bp (the default is 160bp).

```
> myFunctions <- defaultFunctions()
> myFunctions$readSequence <- dfReads
> nReads <- 1000
> simulated <- simChIP(nReads, genome, file = "", functions = myFunctions,
+                 control = defaultControl(readDensity=list(meanLength = 150)))

Generating features...(0.04239893 secs)
Computing binding site density... (6.805141 secs)
Computing read density... (4.02717 secs)
Sampling reads... (0.51542 secs)
Generating read names... (0.00181818 secs)
Determining read sequence and quality... (1.569619 secs)
Total time: 12.96438 secs
```

Note the use of `defaultControl`. This allows us to maintain the default values for all parameters except the ones we list explicitly. The `defaultControl` function uses the same argument names

as `functions`. Each argument is expected to be a named list with the names corresponding to the names of arguments of the target function.

We now have a data frame with 1000 simulated read sequences together with the location in the genome they originated from, the read densities that were used determine read positions as well as the underlying nucleosome densities and feature sequence. The list returned by `simChIP` contains all of this information in different components. Their names should give some idea of what they are:

```
>         names(simulated)

[1] "features"     "bindDensity"  "readDensity"  "readPosition" "readSequence"
[6] "readNames"
```

Of course we can also look at individual components in more detail. Here are the first few reads:

```
> head(simulated$readSequence)

    name                                sequence
1 read_1 CCTGTCAATGTGGCCGTCCGAACGTTCGGGACTCAA
2 read_2 TGGGCGACTCAAATTGAGGCTCTCGTGGCGCTTCGC
3 read_3 TTGCCACTAGACCAGCCATGAATTCTGCCAGACGTC
4 read_4 GCGTCCGACGCGGTCTAGTCGTAACGTTGAGATATT
5 read_5 TTTCCTACGGAGGCTACAAAAGGTACGCAGAGTGCT
6 read_6 GGGATGTACCGAACCAGGCGGCACCGAGAATTGTCA
                                   quality
1    KgRh]XIcBdL[cWHfRBKbBcTMZLTMIYaTA]NA
2 BGSfhN`OBSSK\\fMLQ\\QP`NR^ZLPUT][]HM\\[
3    ^`]W_LFYILZQZd^Lghg_eSfKGY[`U]S_gbOR
4  YdESNThRSRS\\TeJFAYZ^IKgb`[LbhLfg\\TD@
5   hI]SBabUDXFdMbQRNIRP`f_AEF^LTg_hL\\ab
6    IX^EcXcBX`LNcOQXXacIcZQZ@`MDa`aGLDUK
```

We can also examine some of the nucleosome and read densities. Figure 1 shows the nucleosome and read densities surrounding the first stable nucleosome in the feature sequence. The stable nucleosome in the centre is flanked by two phased regions.

```
> feat <- simulated$features[[1]]
> stableIdx <- which(sapply(feat, inherits, "StableFeature"))
> start <- feat[[stableIdx[1]]]$start
> plot((start-2000):(start+500), simulated$bindDensity[[1]][(start-2000):(start+500)], xlab=
+                 ylab="Density", type='l')
> lines((start-2000):(start+500), simulated$readDensity[[1]][(start-2000):(start+500),1], co
> lines((start-2000):(start+500), simulated$readDensity[[1]][(start-2000):(start+500),2], co
> legend("topright", legend=c("Nucleosome density", "Read density (+)", "Read density (-)"),
```

Once the simulation is completed we can use the read sequences and qualities as input for an analysis pipeline. Typically the first step is to map the reads back to the genome (preferably using the read qualities in the process), followed by the application of a peak-finding algorithm to identify peaks in the read counts. The performance of both parts of the analysis can be assessed with the help of data generated by this simulation. This may prove useful to compare and improve analysis pipelines for a variety of ChIP-seq experiments. Once a pipeline is established the simulation can also be used to estimate the amount of sequencing, i.e. the coverage, necessary for a given experiment.

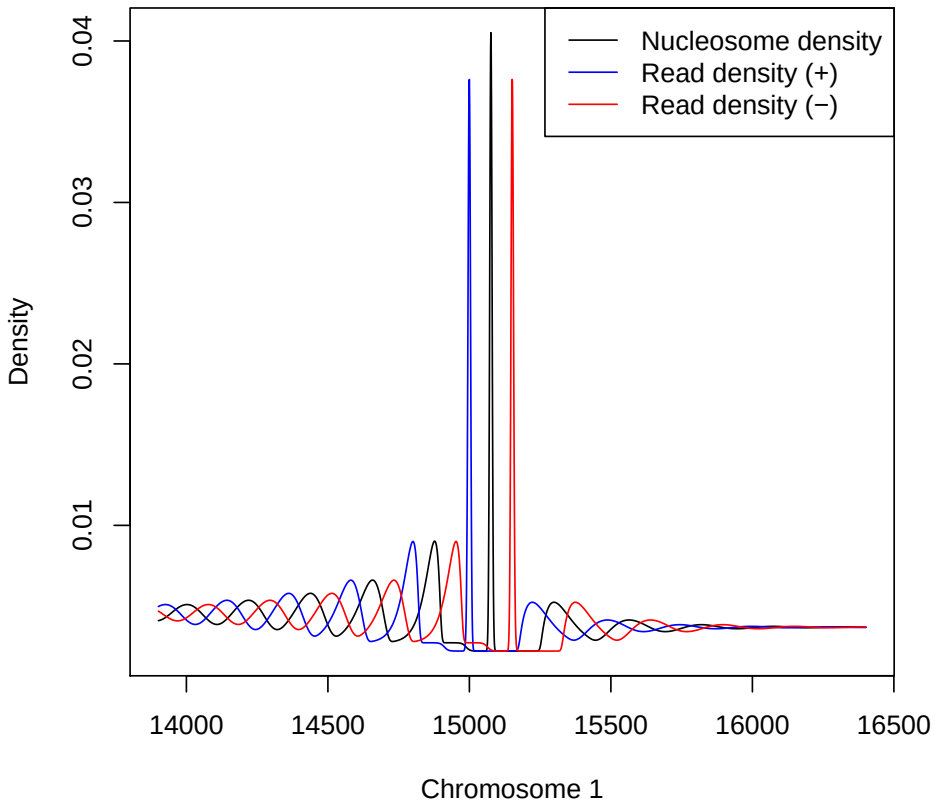

Figure 1: Nucleosome and read densities for a stable feature and flanking regions.

3.3 Using output files

If we had instructed `simChIP` to generate output files by providing a base file name to argument `file` the list would contain the name of the output file for each stage rather than the data. These file names are automatically generate by appending an appropriate suffix to the base name at each stage. This name is also used to determine whether output from a previous run is available for re-use. For example, if we set `file = "test"` in the above call to `simChIP` the simulation will generate several files including 'test_features.rdata', 'test_bindDensity.rdata' and 'test_readDensity.rdata' for the first three stages of the simulation. If any of these files are found in the working directory and we set `load = TRUE` the last file in the sequence is loaded and used as input for the next stage, skipping all previous stages. In the above example the names of files produced by the final three stages would be 'test_1_readPosition.rdata', 'test_1_readNames.rdata' and 'test_1_fastq.txt'. These all relate to the reads that were sampled from the read densities and are not loaded as intermediate results. Instead `simChIP` scans the working directory for files with these names of the form 'test_n_fastq.txt' to determine which index n to use for the current run of the simulation. This enables us to run the same simulation with the same underlying feature sequence, nucleosome and read density repeatedly, each time generating a new set of reads. This essentially simulates repeated sequencing of the same library. Of course it is also possible to introduce some variation into the process. For example if we wanted to study the effect of changes to the DNA fragment size distribution we could use the same nucleosome density but recalculate the read densities with a new set of parameters.

4 Extending ChIPsim

The `ChIPsim` package provides a framework for the simulation of ChIP-seq experiments. While, as we have seen above, the package includes an implementation of a nucleosome positioning simulation the real strength of `ChIPsim` is its ability to accommodate a variety of different experiments. Here we will demonstrate how `ChIPsim` can be extended to other types of experiments. As an example we will create a simulation of transcription factor binding sites based on the one proposed by Zhang *et al.* [1]. This simulation divides the genome into a number of non-overlapping binding sites separated by background regions. The background between binding sites is split into blocks of 1kb, each with its own sampling weight.

In the following sections we will implement a variant of this simulation using the `ChIPsim` framework, demonstrating how the different stages of the simulation can be adjusted for different types of experiments. This is followed by an alternative, more complex simulation of the same process.

4.1 A simple model

In this section we will attempt to recreate the simulation of Zhang *et al.* (or at least its core aspects). As we will see later there are some differences in the assumptions underlying the work of Zhang *et al.* and `ChIPsim`, which means that we will not use the full extend of `ChIPsim`'s capabilities. This section is followed by the description of a more complex model of transcription factor ChIP-seq that is better suited for an implementation in `ChIPsim`.

4.1.1 Markov chain

At the core of the simulation is a Markov model that generates a sequence of different features. Each state of the model corresponds to a different feature class and generates a set of param-

eters required to determine the sampling weight for each feature. For the transcription factor simulation we will only need two states, representing binding sites and background respectively.

To specify the Markov model we need to define the transition probabilities between the states and the initial state distribution. For each state we will also need a function to generate the parameters of the corresponding features. We start by specifying the transition probabilities. To do this we need to create a list with components corresponding to the states of the model (we will us "Binding" and "Background") each of which holds an (S3) object of class "StateDistribution", which is simply a numeric vector with the non-zero transition probabilities to other states.

```
> transition <- list(Binding=c(Background=1), Background=c(Binding=0.05, Background=0.95))
> transition <- lapply(transition, "class<-", "StateDistribution")
```

Note that we are allowing several background regions to follow each other while binding regions always have to be followed by a background region. Also note that the transition probability from background to binding regions determines the expected number of binding regions. Unlike Zhang *et al.* we are not determining the number in advance.

We will not allow the sequence to start with a binding site. Therefore, again using the "StateDistribution" class, the initial state distribution is:

```
> init <- c(Binding=0, Background=1)
> class(init) <- "StateDistribution"
```

The next step is to define functions to generate the parameters for binding and background regions. The parameters for a region should always include '`start`' and '`length`', indicating the start position and length of the region respectively. Note that '`start`' will be determined by the function that generates the feature sequence (see below) so that we only have to accept it as a parameter. The only other parameter required for background regions is the sampling weight. Following the model of Zhang *et al.* we use a gamma distribution to model the background sampling weight for each region.

```
> backgroundFeature <- function(start, length=1000, shape=1, scale=20){
+         weight <- rgamma(1, shape=1, scale=20)
+         params <- list(start = start, length = length, weight = weight)
+         class(params) <- c("Background", "SimulatedFeature")
+
+         params
+ }
```

The convention used by `ChIPsim` is to name functions that generate parameters for the different feature classes "*state*Feature", where *state* is the name of the state. These functions always return a list of parameters. This list has classes "*state*" and "SimulatedFeature". It is important that the first class matches the name used for the state in other places.

The binding site model requires some additional parameters. Zhang *et al.* set the average sampling weight $\bar{w}_f$ for binding sites to be the average weight of background regions multiplied by an enrichment coefficient t: $\bar{w}_f = t \times \bar{w}_b$. They then increase the sampling weight of binding sites as sequence reads are assigned to them, resulting in a power-law distribution of sampling weights. Since `ChIPsim` separates the generation of feature densities and the sampling of sequence reads from those densities into two distinct stages it is not easily possible to use the same procedure here. Instead we will use a Pareto distribution with parameter r to determine w_f for each binding site. The minimum of the distribution is chosen such that its mean is $\bar{w}_f$ (Note that this requires $r > 1$).

```
> bindingFeature <- function(start, length=500, shape=1, scale=20, enrichment=5, r=1.5){
+         stopifnot(r > 1)
+
+         avgWeight <- shape * scale * enrichment
+         lowerBound <- ((r - 1) * avgWeight)
+         weight <- actuar::rpareto1(1, r, lowerBound)
+
+         params <- list(start = start, length = length, weight = weight)
+         class(params) <- c("Binding", "SimulatedFeature")
+
+         params
+ }
```

With this in place we can now generate a feature sequence using the `ChIPsim` function `makeFeatures`. This function takes the description of a Markov model together with several other parameters and produces a list of features for a genomic region of given length. This list is of class "SimulatedExperiment" and usually has a more specialized class indicating the specific type of experiment as well. We will see later how this class information is used. To generate a feature sequence we can use something like this:

```
> set.seed(1)
> generator <- list(Binding=bindingFeature, Background=backgroundFeature)
> features <- ChIPsim::makeFeatures(generator, transition, init, start = 0, length = 1e6, g
+                 experimentType="TFExperiment", lastFeat=c(Binding = FALSE, Background = T
```

We have already discussed the `transition` and `init` parameters of `makeFeatures`. The `generator` parameter also relates to the Markov model; it provides a list of the emission distributions for each state. Another interesting argument of `makeFeatures` is `globals`. This allows us to pass a list of parameters that will be passed on to all states of the model. Here we use this mechanism to ensure that the values used by both states for `shape` and `scale` match. There is another parameter called `control` that allows us to pass a list of arguments to individual states. The feature sequence generated above contains 54 binding sites, see Figure 2 for a plot of the resulting weight distributions of binding and background regions. Below is the first feature in the sequence:

```
> features[[1]]

Object of class Background
start: 1
length: 1000
weight: 5.918505
```

Before we turn our attention to the conversion of this sequence into binding site densities it is worth noting that there is another function we can use to generate a feature sequence. Using `placeFeatures` provides some additional functionality. Firstly, `placeFeatures` makes an effort to generate a feature sequence that fills as much of the genomic region as possible. This is especially useful for cases where we are dealing with variable length features that may otherwise lead to a relatively large area without any features at the end of the genomic region. Secondly, it calls `reconcileFeatures`, an S3 method that dispatches on the class of the feature sequence (i.e., the experiment type) and provides a way to post-process the feature sequence. The nucleosome simulation uses this to ensure that some of the parameters of adjacent features

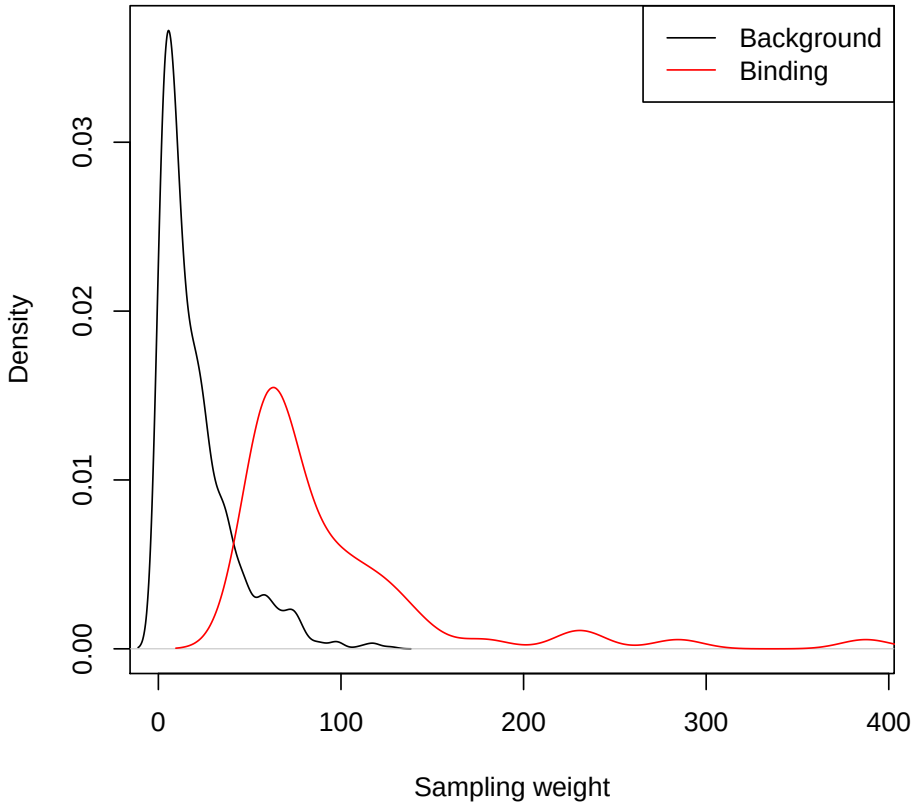

Figure 2: Sampling weight distributions for background and binding regions.

are compatible. This mechanism is not needed for the simulation considered here so that we can use the default. However, if we wanted to use it we could simply define a function `reconcileFeatures.TFExperiment(features, ...)` to perform the desired post-processing.

```
> set.seed(1)
> features <- ChIPsim::placeFeatures(generator, transition, init, start = 0, length = 1e6,
+                  experimentType="TFExperiment", lastFeat=c(Binding = FALSE, Background = T
```

This results in the same feature sequence but a closer look reveals that the default post-processing step has introduced an additional parameter for each feature:

```
> features[[1]]

Object of class Background (reconciled)
start: 1
length: 1000
weight: 5.918505
overlap: 0
```

The overlap between adjacent features defaults to 0, which is what we want here. In some cases it may be desirable to have overlapping features. The nucleosome simulation uses this to ensure smooth transitions between features.

4.1.2 Binding site density

Once the feature sequence is determined it has to be translated into a binding site density. This task is carried out through a call to `featureDensity` which is dispatched on the feature class. This means that we have to provide implementations for `featureDensity.Binding` and `featureDensity.Background`. In this case both functions are simply constant and produce output of a given length.

```
> constRegion <- function(weight, length) rep(weight, length)
> featureDensity.Binding <- function(feature, ...) constRegion(feature$weight, feature$lengt
> featureDensity.Background <- function(feature, ...) constRegion(feature$weight, feature$le
```

A binding site density for the entire genomic region is generated by a call to `feat2dens`.

```
>  dens <- ChIPsim::feat2dens(features)
```

The resulting density for a region containing the first binding site in the sequence is shown in Figure 3. Since the simulation of Zhang *et al.* does not distinguish between reads on the forward and reverse strand we could simply use this binding site density to sample the location of (extended) reads.

```
>  readLoc <- sample(44000:64000, 1e3, prob=dens[44000:64000], replace=TRUE)
```

Here we are assuming that sampled read positions correspond to the fragment centre and extend reads into both directions. Figure 3 shows counts of reads that were extended to 200bp, the average fragment length.

We now have a (simple) simulation of transcription factor ChIP-seq based on the work by Zhang *et al.* This does not incorporate all aspects of their simulation, e.g. the intra-site weight profile is missing from this version, but demonstrates the principle. In the next section we will build on this to create a simulation that takes information about binding site and DNA fragment length into account to generate strand specific read densities.

4.2 Advanced model

This extended version of the transcription factor ChIP-seq simulation builds on the model from the previous section. We will use the same Markov chain as above with the same emission distributions; but unlike before we will not use blocks of 500bp to represent binding sites. Instead the length of a binding site will be defined as the stretch of DNA protected by the transcription factor. What will change is the binding site density for the "Binding" state. We will use a single spike at the centre of the binding site rather than a uniform distribution across the binding site. This will allow us to compute strand specific read densities later. This requires some adjustment to the background density as well.

4.2.1 Post-processing the feature sequence

Since we plan to change the binding site density such that a transcription factor binding site is represented by a single spike at the centre of the binding site, we have to reduce the background density accordingly. It would be straightforward to incorporate this change into `featureDensity.Background` but for the purpose of this example we will do this as part of the post-processing step that was mentioned but not illustrated before. We will assume that all binding sites have the same length. That should be a reasonable assumption if we only deal with one transcription factor per experiment.

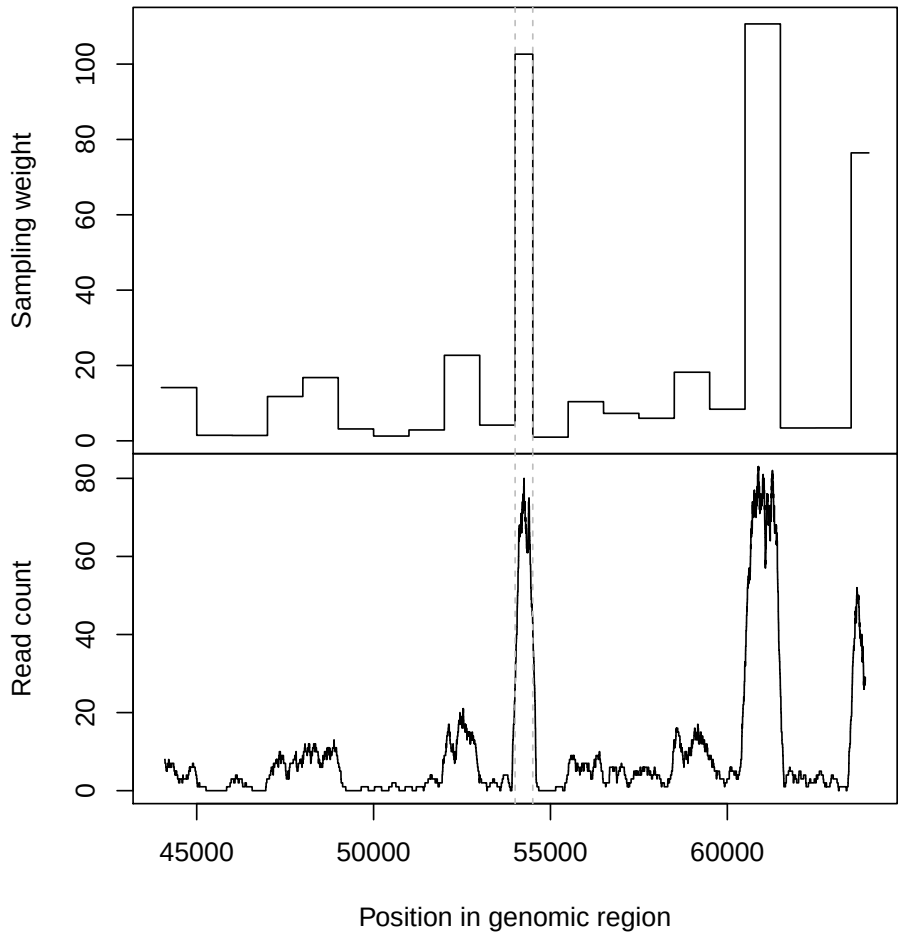

Figure 3: Binding site density for a transcription factor binding site (between the dashed lines) and surrounding background (top). Resulting counts of overlapping extended reads are shown below.

```
>         reconcileFeatures.TFExperiment <- function(features, ...){
+                 bindIdx <- sapply(features, inherits, "Binding")
+                 if(any(bindIdx))
+                         bindLength <- features[[min(which(bindIdx))]]$length
+                 else bindLength <- 1
+                 lapply(features, function(f){
+                                 if(inherits(f, "Background"))
+                                         f$weight <- f$weight/bindLength
+                                 ## The next three lines (or something to this effect)
+                                 ## are required by all 'reconcileFeatures' implementations
+                                 f$overlap <- 0
+                                 currentClass <- class(f)
+                                 class(f) <- c(currentClass[-length(currentClass)],
+                                                 "ReconciledFeature", currentClass[length(c
+                                 f
+                         })
+         }
```

Now we can generate a new feature sequence.

```
> set.seed(1)
> features <- ChIPsim::placeFeatures(generator, transition, init, start = 0, length = 1e6,
+                  experimentType="TFExperiment", lastFeat=c(Binding = FALSE, Background = T
+                  control=list(Binding=list(length=50)))
```

Note the use of the `control` argument to set the length of binding sites. The parameters of the first feature have changed slightly:

```
>         features[[1]]

Object of class Background (reconciled)
start: 1
length: 1000
weight: 0.1183701
overlap: 0
```

4.2.2 Revised binding site density

Here is the modified feature density for binding regions:

```
> featureDensity.Binding <- function(feature, ...){
+          featDens <- numeric(feature$length)
+          featDens[floor(feature$length/2)] <- feature$weight
+          featDens
+ }
```

Now we can generate the new binding site density, using the same call as before.

```
>         dens <- ChIPsim::feat2dens(features, length = 1e6)
```

Figure 5 shows the binding site density in the same genomic region as before.

4.2.3 Strand specific read densities

The function `bindDens2readDens` provides us with an easy way to convert binding site densities into read densities for both strands. All we need to do to use this function for our transcription factor simulation is to specify a function that returns the length distribution of DNA fragments used in the experiment. This function should accept a numeric first argument and return the proportion of DNA fragments of that length in the sample. Further arguments the function should accept (and may use) are the minimum and maximum fragment length as well as the length of the binding site.

```
>         fragLength <- function(x, minLength, maxLength, meanLength, ...){
+                 sd <- (maxLength - minLength)/4
+                 prob <- dnorm(minLength:maxLength, mean = meanLength, sd = sd)
+                 prob <- prob/sum(prob)
+                 prob[x - minLength + 1]
+         }
```

Here we do not use the length of the binding site but our function either needs a `bind` argument or use '...' to absorb any additional arguments.

```
>         readDens <- ChIPsim::bindDens2readDens(dens, fragLength, bind = 50, minLength = 15
+                         meanLength = 200)
```

This produces a two column matrix with the read density for the forward strand in the first column and the reverse strand density in the second. Figure 4 gives an impression of how the read densities relate to the binding site density.

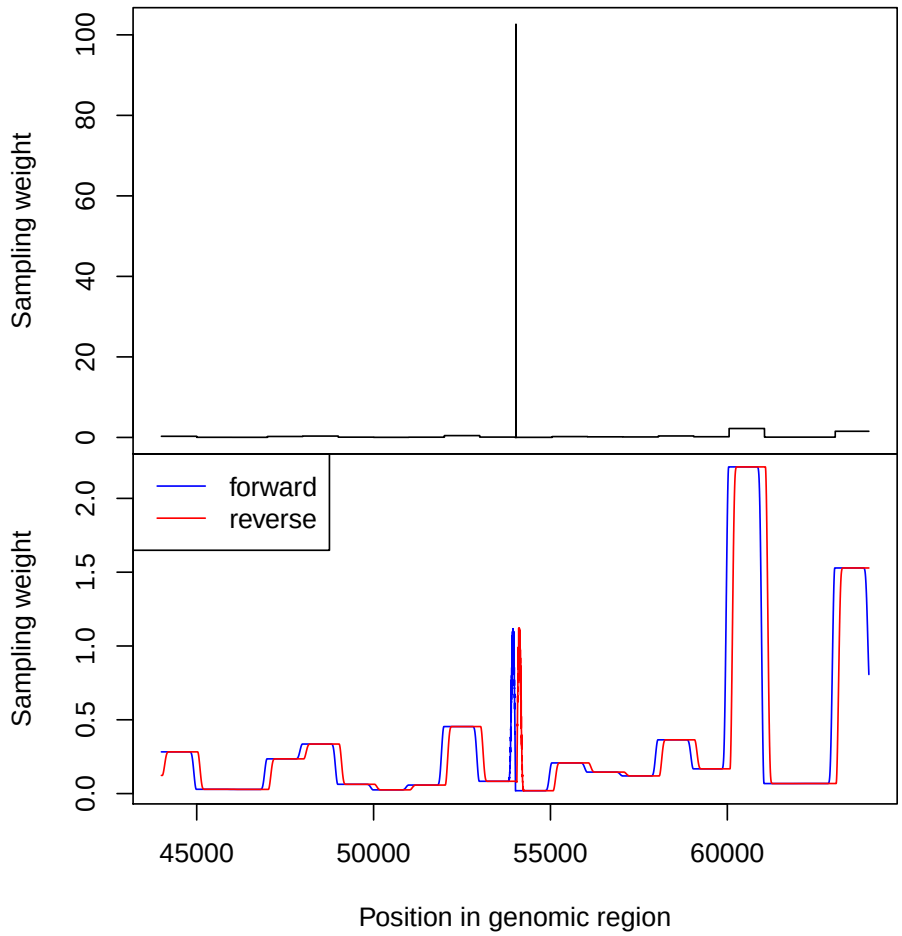

Figure 4: Binding site density for a transcription factor binding site and surrounding background for the modified model (top) and resulting strand specific read densities (bottom).

4.2.4 Sampling reads

Once the read densities are computed it is straightforward to sample read positions. The `sampleReads` function is provided for this purpose. It allows us to specify the number of reads required as well as weights for each strand. Here we sample 100,000 reads from the read densities with equally weighted strands, which is the default.

```
>         readLoc <- ChIPsim::sampleReads(readDens, 1e5)
```

This produces a list with components `fwd` and `rev` giving the start positions of reads on the respective strand. Figure 5 shows the transcription factor binding site with read densities and read positions. Following the procedure in Zhang *et al* we can extend each read to the average

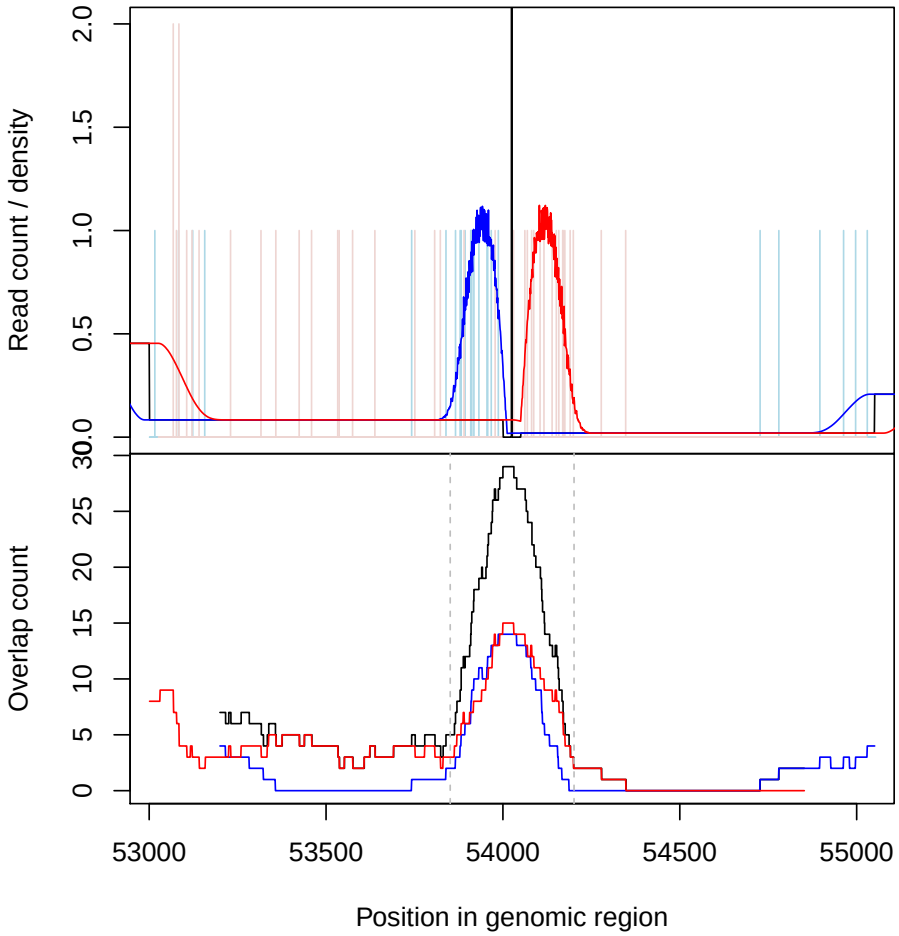

Figure 5: Read counts and densities for a transcription factor binding site. In the top panel vertical light blue and red bars indicate the number of reads starting at each position on the forward and reverse strand respectively. The underlying read densities are shown as blue and red lines and the binding site density is shown in black. The number of overlapping extended reads on the forward (blue) and reverse (red) strand are shown in the bottom panel.

fragment length (200bp) and count the number of overlapping reads at each position. This results in a peak surrounding the binding site that is approximately 350bp wide.

4.2.5 Putting it all together

After generating the desired number of read positions the resulting read sequences can be written to a file using the infrastructure provided by `ChIPsim` and will not be covered here in detail. A final point worth noting is that although we have carried out the various steps of the simulation for demonstration purposes, it is not necessary to do this in practice. The `ChIPsim` package provides the `simChIP` function that allows us to run the entire simulation with a single function call. Using the output method described in "Introduction to ChIPsim" to produce read sequences we first set up the functions to carry out the different stages of the simulation. As we have seen above we can rely on the build-in functions for this, only the last stage needs to be changed to accommodate the fact that we do not want to create output files.

```
>         myFunctions <- ChIPsim::defaultFunctions()
>         myFunctions$readSequence <- dfReads
```

Now we have to adjust the arguments that will be passed to these functions to enable them to work with our transcription factor simulation.

```
>         featureArgs <- list(generator=generator, transition=transition, init=init, start =
+                         length = 1e6, globals=list(shape=1, scale=20), experimentType="TFE
+                         lastFeat=c(Binding = FALSE, Background = TRUE), control=list(Bindi
>         readDensArgs <- list(fragment=fragLength, bind = 50, minLength = 150, maxLength =
+                         meanLength = 200)
```

After generating a random reference sequence we are all set.

```
>         genome <- Biostrings::DNAStringSet(c(CHR=paste(sample(Biostrings::DNA_BASES, 1e6,
>         set.seed(1)
>         simulated <- ChIPsim::simChIP(1e4, genome, file = "", functions = myFunctions,
+                         control = ChIPsim::defaultControl(features=featureArgs, readDensit

Generating features...(0.1484458 secs)
Computing binding site density... (0.2811308 secs)
Computing read density... (5.351334 secs)
Sampling reads... (0.592073 secs)
Generating read names... (0.01217318 secs)
Determining read sequence and quality... (44.17476 secs)
Total time: 50.56235 secs
```

Note that we reduced the number of reads in this simulation to avoid generating a large number of read sequences, which can be time consuming. Apart from this change the result is the same as before:

```
>         all.equal(readDens, simulated$readDensity[[1]])

[1] TRUE
```

5 Session info

```
> sessionInfo()

R version 3.1.1 Patched (2014-09-25 r66681)
Platform: x86_64-apple-darwin10.8.0 (64-bit)

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats4    parallel  stats     graphics  grDevices utils     datasets
[8] methods   base

other attached packages:
[1] ChIPsim_1.20.0      Biostrings_2.34.0   XVector_0.6.0
```

```
[4] IRanges_2.0.0       S4Vectors_0.4.0     BiocGenerics_0.12.0

loaded via a namespace (and not attached):
 [1] actuar_1.1-6            base64enc_0.1-2         BatchJobs_1.4
 [4] BBmisc_1.7              Biobase_2.26.0          BiocParallel_1.0.0
 [7] bitops_1.0-6            brew_1.0-6              checkmate_1.4
[10] codetools_0.2-9         DBI_0.3.1               digest_0.6.4
[13] fail_1.2                foreach_1.4.2           GenomeInfoDb_1.2.0
[16] GenomicAlignments_1.2.0 GenomicRanges_1.18.0    grid_3.1.1
[19] hwriter_1.3.2           iterators_1.0.7         lattice_0.20-29
[22] latticeExtra_0.6-26     RColorBrewer_1.0-5      Rsamtools_1.18.0
[25] RSQLite_0.11.4          sendmailR_1.2-1         ShortRead_1.24.0
[28] stringr_0.6.2           tools_3.1.1             zlibbioc_1.12.0
[31] zoo_1.7-11
```

References

[1] Zhengdong D. Zhang, Joel Rozowsky, Michael Snyder, Joseph Chang, and Mark Gerstein. Modeling chip sequencing in silico with applications. *PLoS Computational Biology*, 4(8), August 2008.