PS1:SF-Seq Quality Assessment and Read Analysis

Adrian Bubie

Goals:

Assess the quality of two of our Splicing Factor Sequencing (SF-Seq) read libraries from our round of RNA-Seq in Bi622. For this assignment, the following libraries are analyzed:

```
19_3F_fox
23_4A_control
```

Once quality has been tracked using `FastQC`, we will perform adapter trimming on these libraries and determine how these trimmed results compare with the fragment analysis data for the libraries.

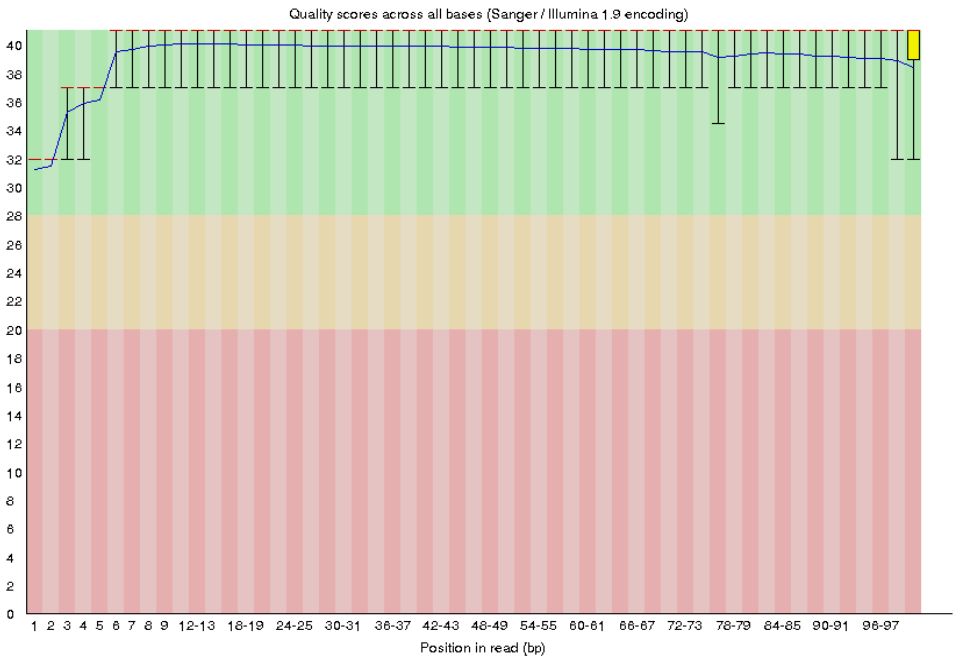
Finally, we will align the trimmed reads against rRNA sequences pulled from rFam database for the mouse (*mus musculus*) genome and report the number of reads that align to the rRNA, using `gsnap`.

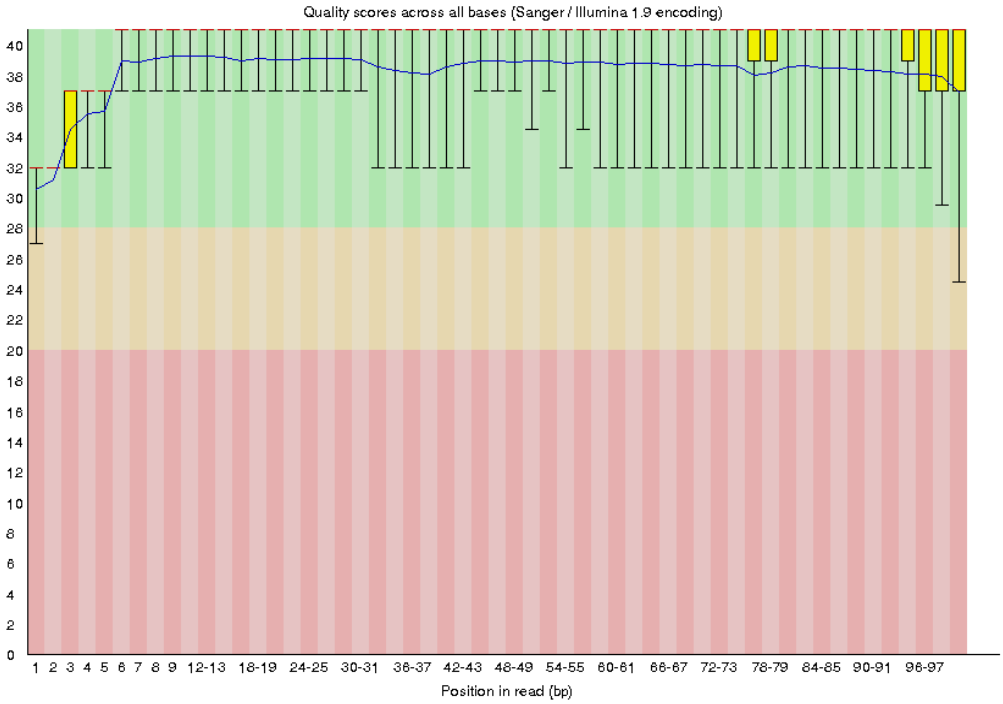
Part 1: SF-Seq read quality score distributions

(Please see the associated "FastQC_srun.sh" script file on github for the full list of commands used to produce the FastQC results for the two libraries assessed.)

A. Assessment of libraries using FastQC

19_3F_fox R1/R2 Quality Measures Generated by FastQC:

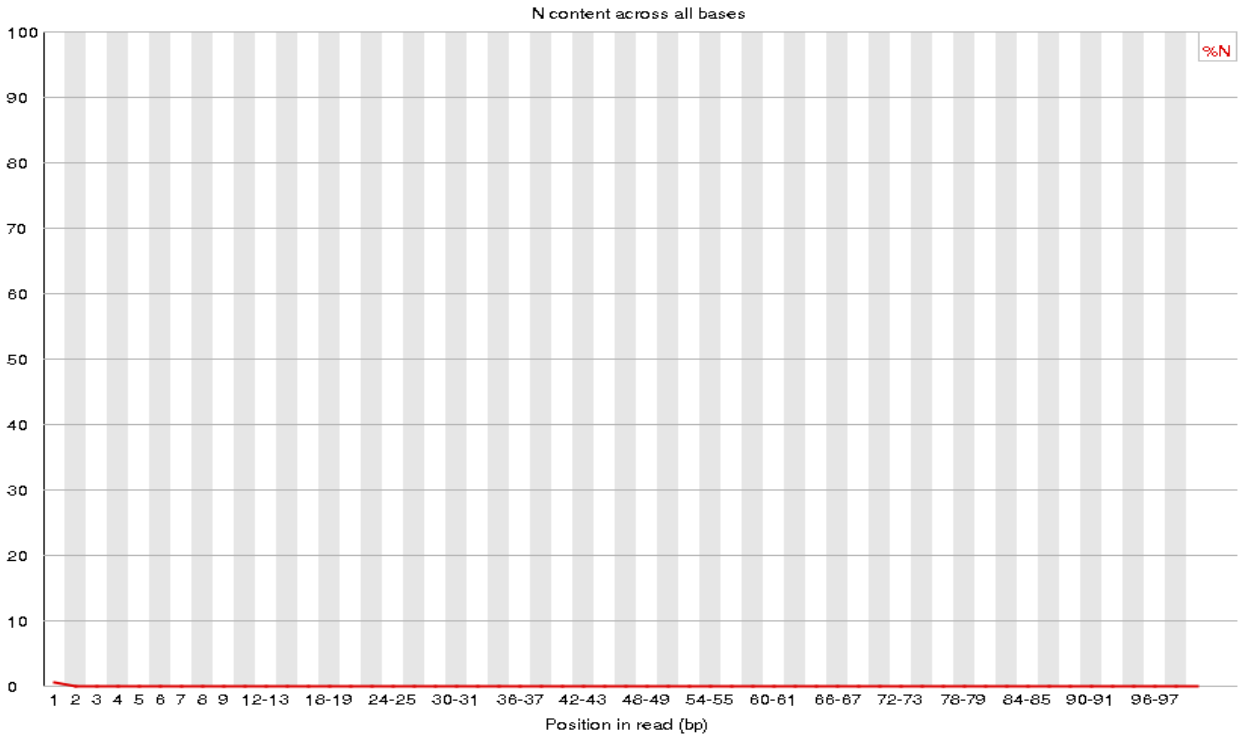




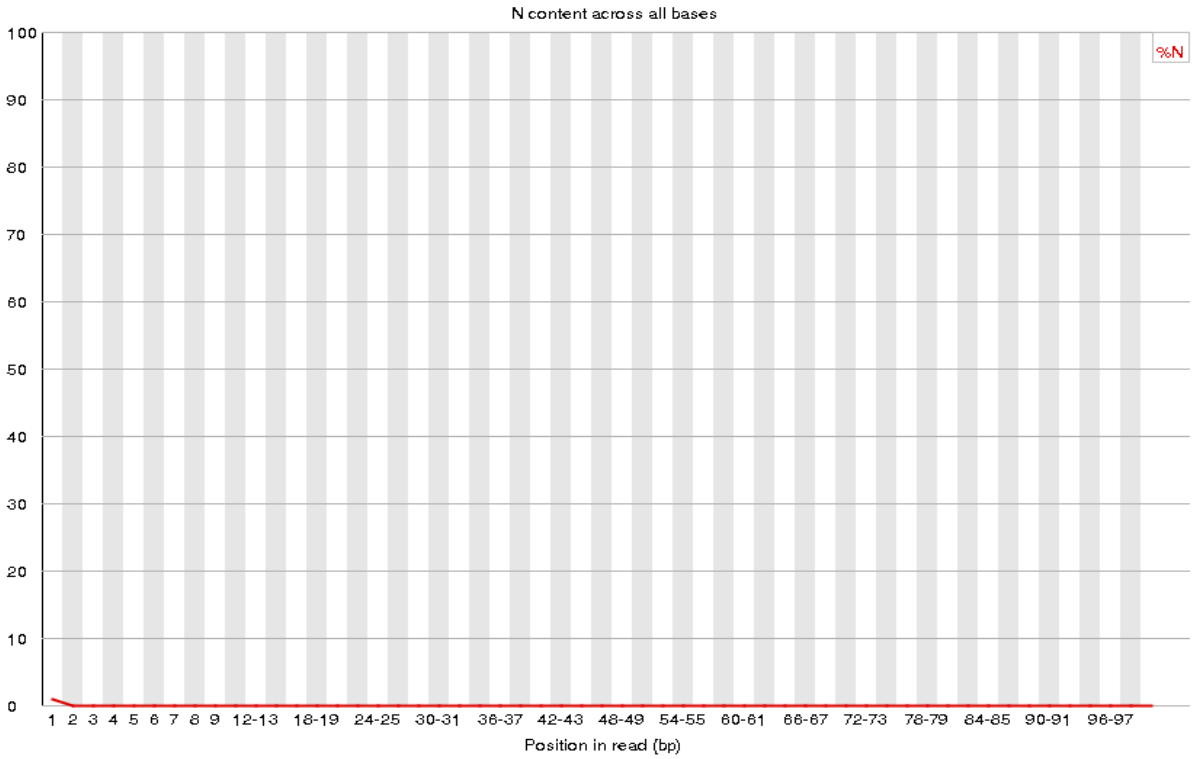
Here, we notice that the average quality is a little lower, and a little less consistent across the reverse reads in this library, though most likely not to a significant degree, and not enough of a drop to signal an issue by the FastQC analysis.

It is clear, however, that both reads see a relative drop in average quality among the first 7-8 bases compared to the rest of the positions in the read. This would indicate that the sequencer had more difficulty resolving these positions for reads across the board, and subsequently, we can expect to see higher counts of per-base N content at these first positions. This is seen to be true when looking at the per-base N content (%) graphs provided by FastQC, where there is a slight uptick (1-2%) of N's in the first 2 positions in the reads, before leveling out to just about 0% for the rest of the positions. N's are normally scored lower by Phred quality measures, so this is consistent with the trend seen in the average quality plots.

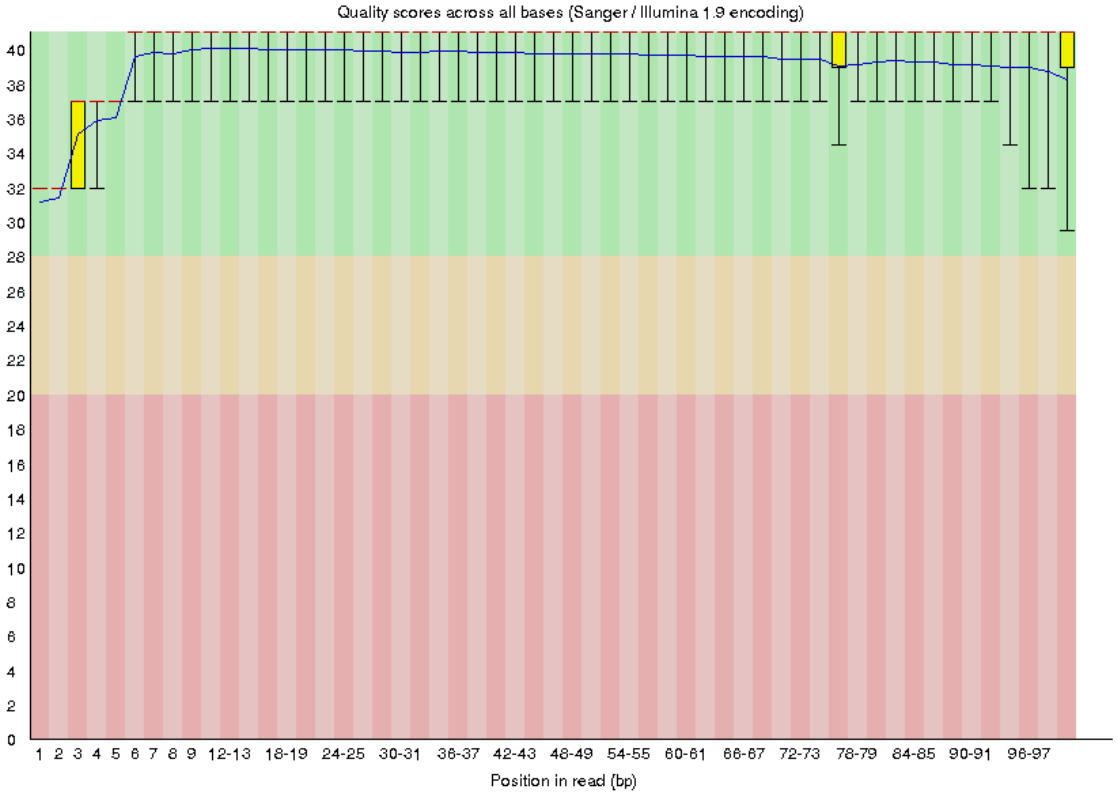
Per Base N-Content for R1:

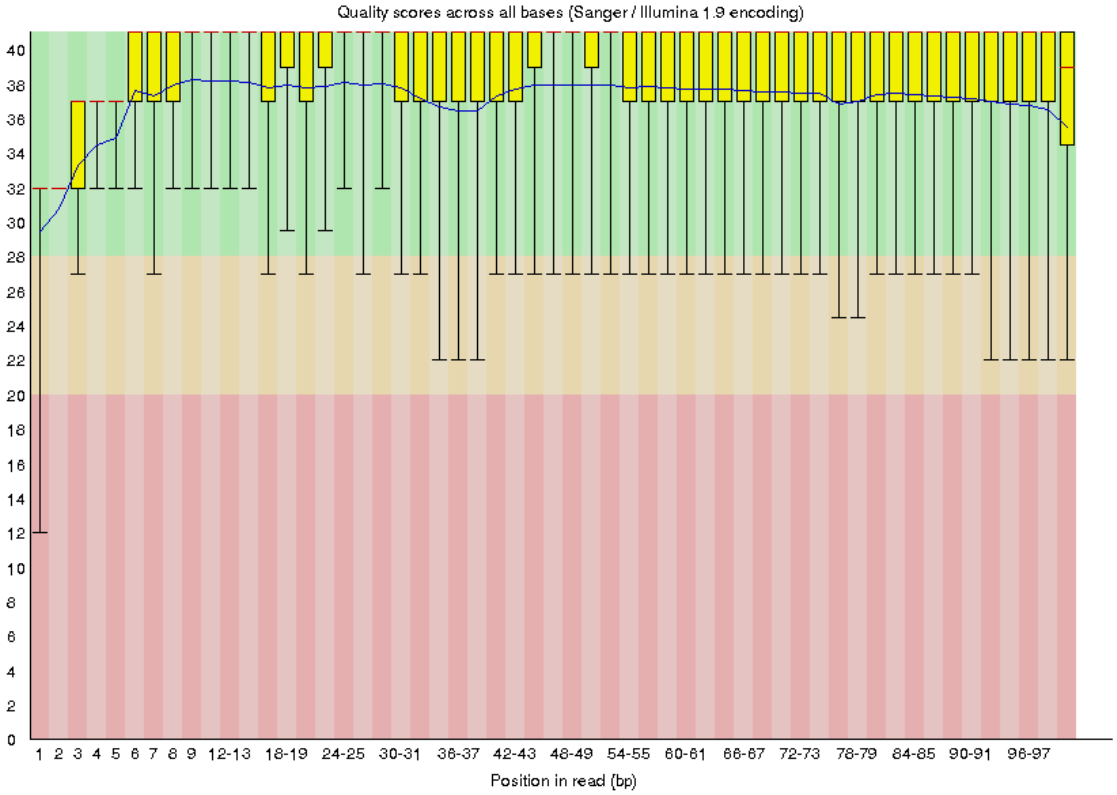


Per Base N-Content for R2:



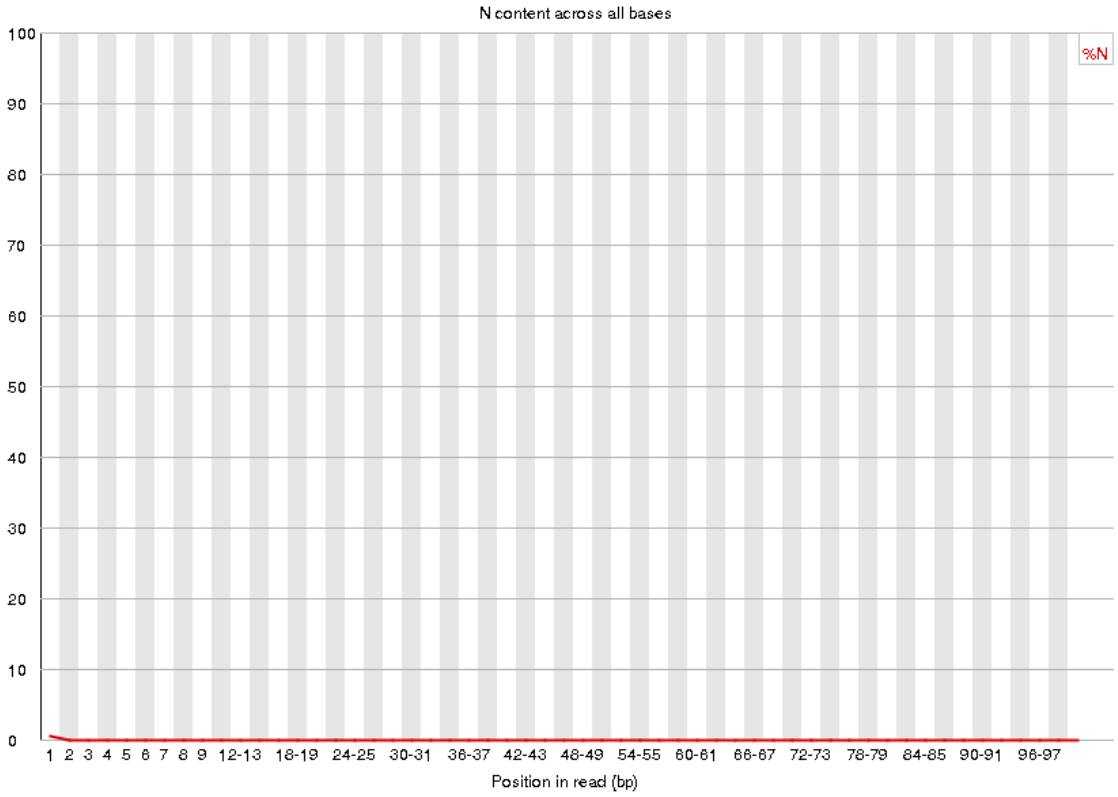
23_4A_control R1/R2 Quality Measures Generated by FastQC:



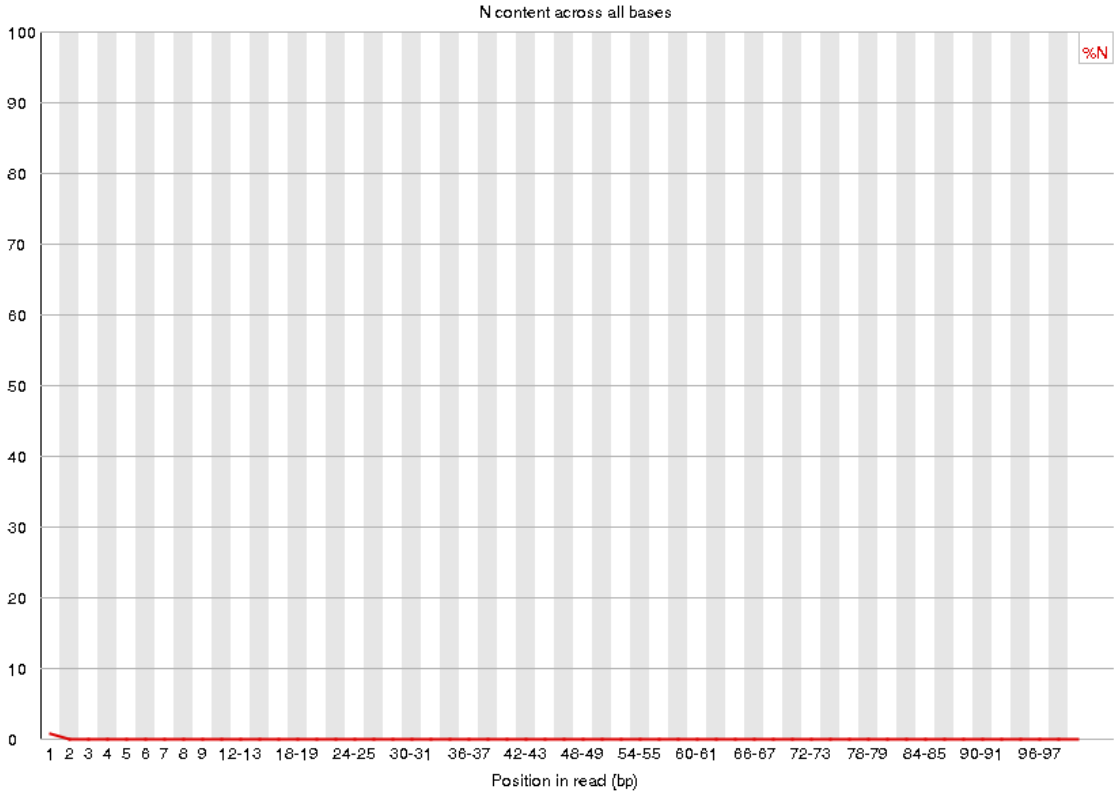


In a similar vein to the 19_3F_fox reads, we see that the average quality per position for the reverse strand reads (R2) is lower, though the difference between the two read sets is much more exaggerated in this library. However, the same general trend is observed with the first 7-8 base positions seeing significantly lower average quality scores across both read sets, meaning we can expect a similar per-base N content trend as the previous library (19_3F_fox). Looking at the plots, we see it is so, though perhaps with slightly lower N-content than we might expect for the first 2 positions, based on the drop in quality for the first bases in the average quality score graphs:

Per Base N-Content for R1:



Per Base N-Content for R2:

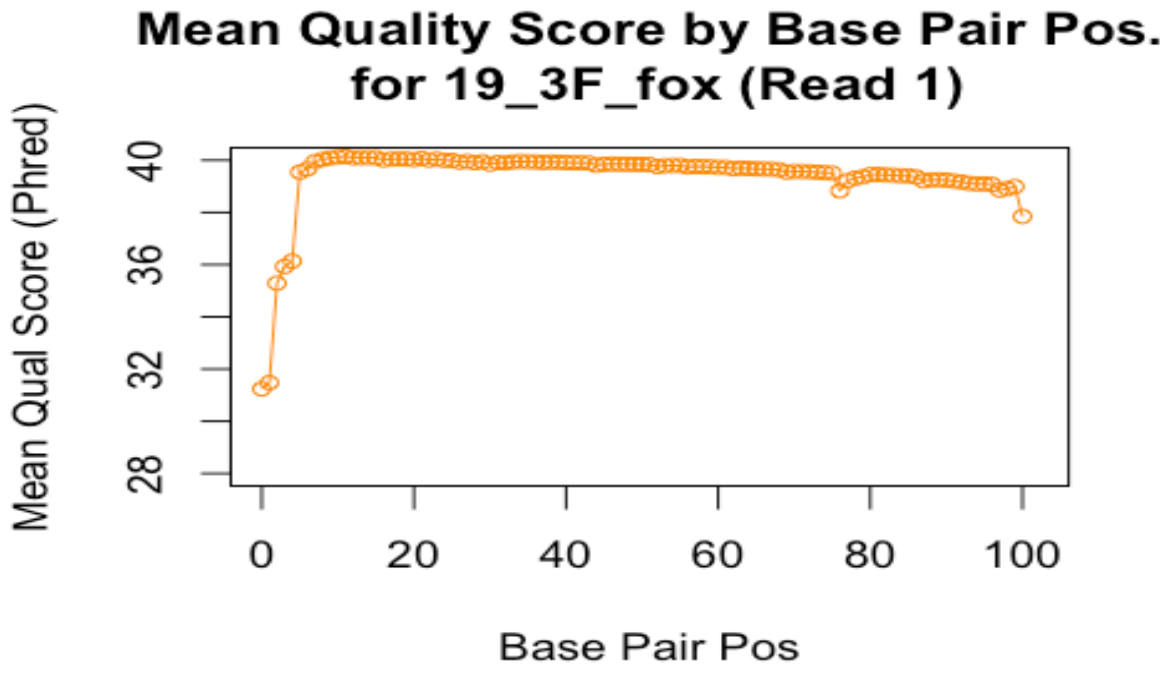


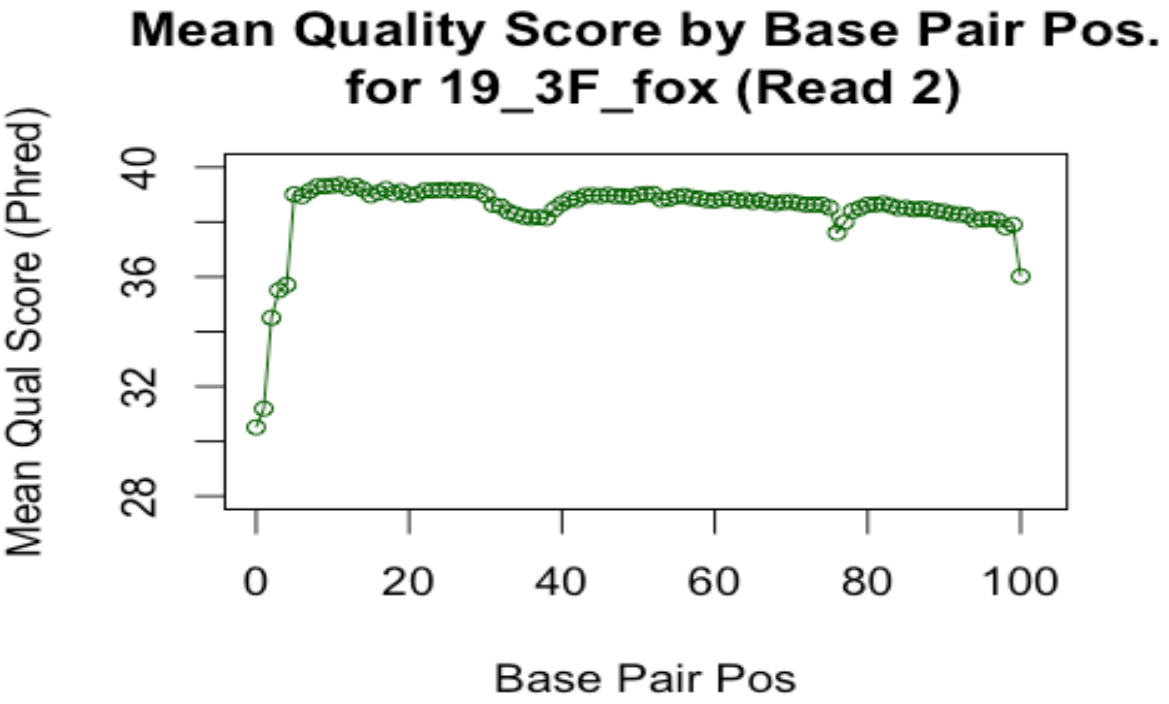
In the case of both libraries, the increase in N-content can only explain the drop in quality among the first few base positions; there is some other reason for the quality drop-off for the reverse strand reads (R2) in both libraries that is *not* related to poor sequencer resolution at the beginning of a read.

B. Assessment of libraries using Python scripts:

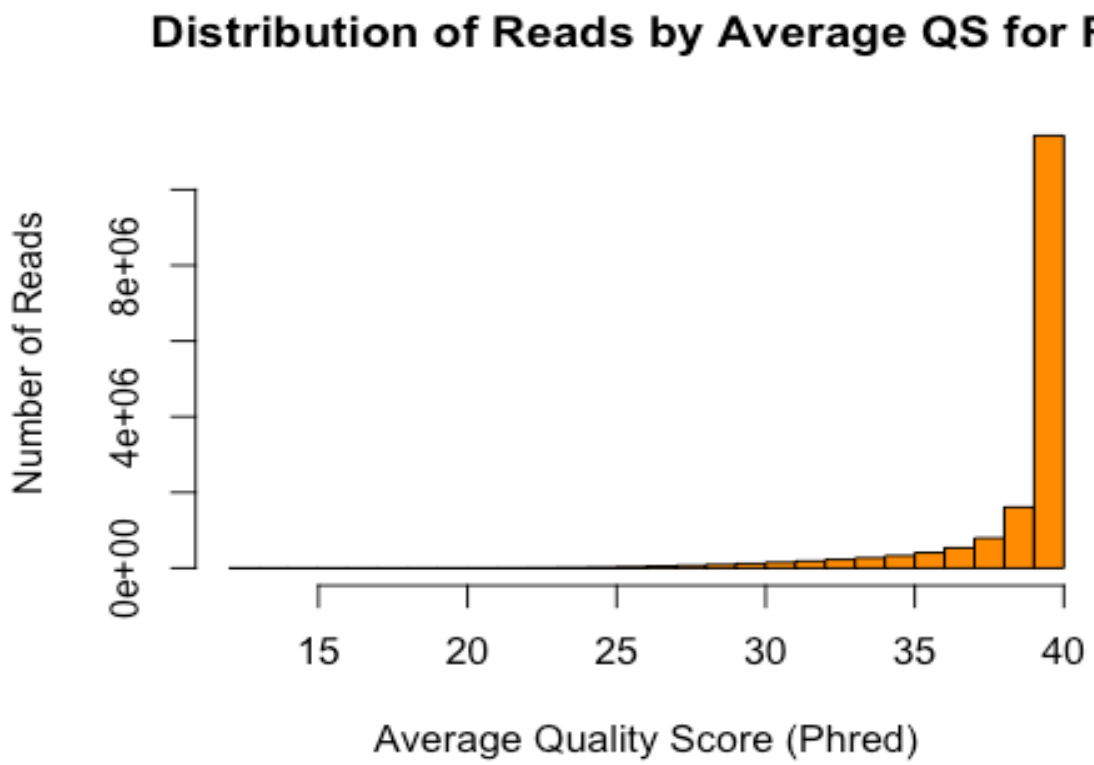
The same Quality analysis code used to assess the non-demultiplexed libraries was run on each library (Python code and BASH scripts can be found with the associated files in github; see the README for details), and Quality distributions and average quality score by position are plotted below, using R:

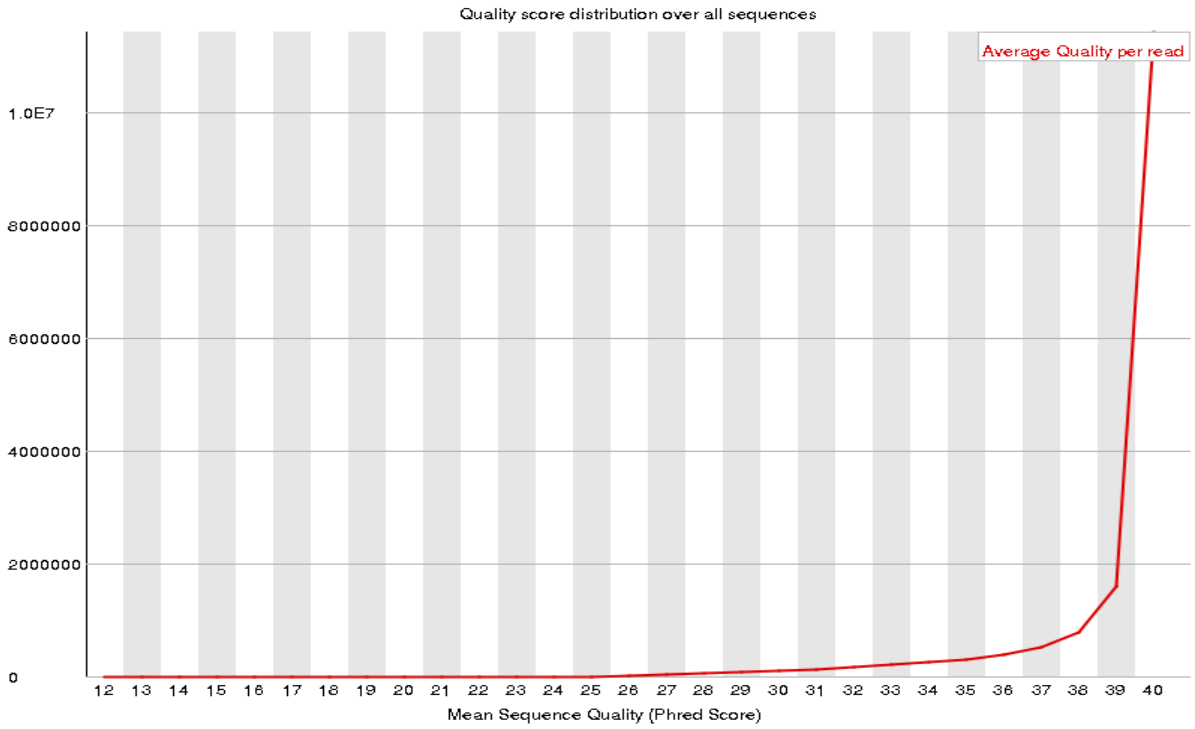
19_3F_fox library:

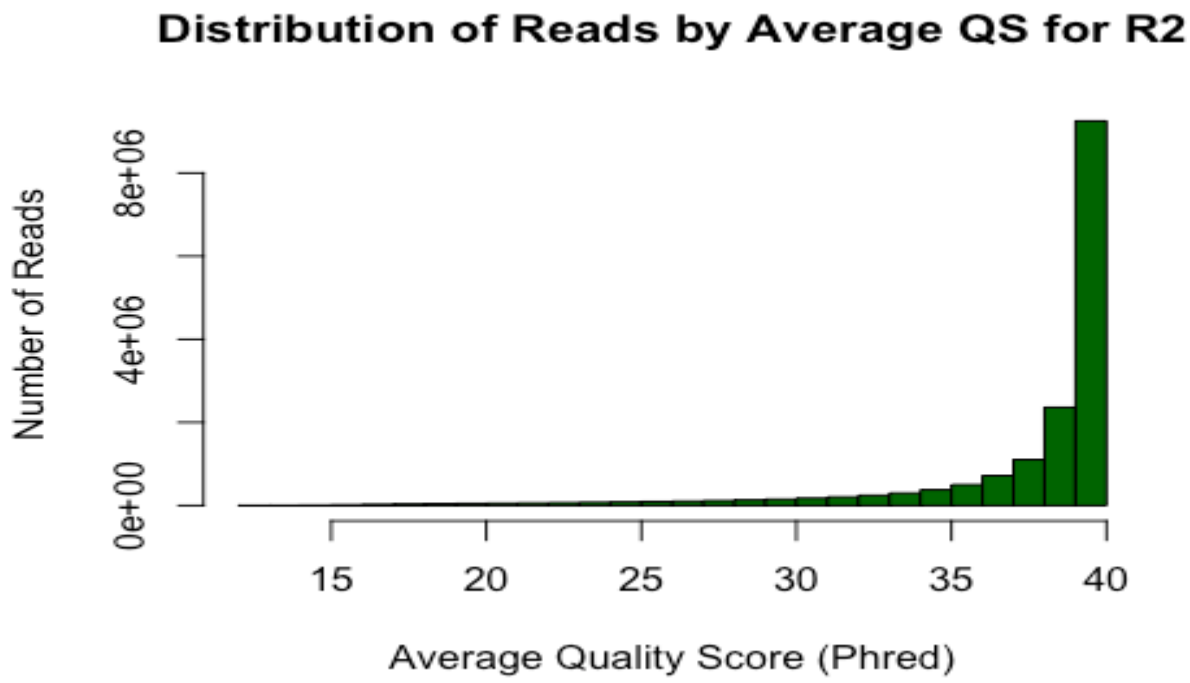


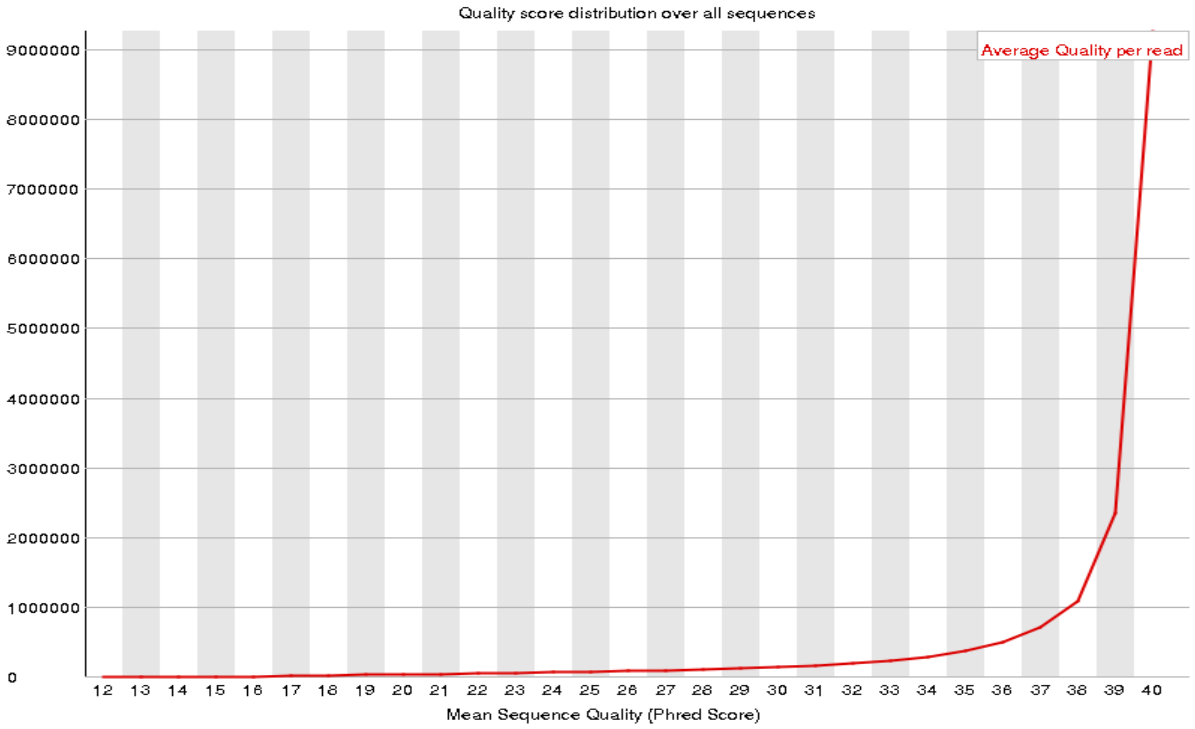


These plots appear to agree with the average quality per base figures produced for the 19_3F_fox library by FastQC. Taking a look at the Quality distributions produced by the python script in comparison to those produced by FastQC:


Distribution of Reads by Average QS for R1
Number of Reads
0e+00
4e+06
8e+06
15
20
25
30
35
40
Average Quality Score (Phred)

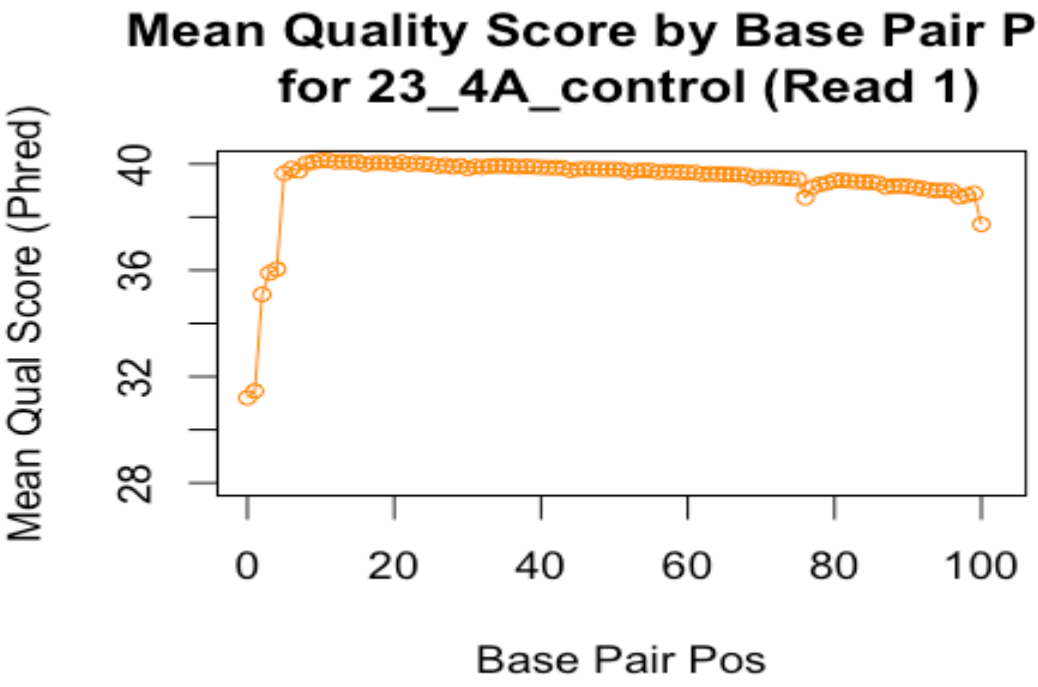

Quality score distribution over all sequences
Average Quality per read
1.0E7
8000000
6000000
4000000
2000000
0
12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
Mean Sequence Quality (Phred Score)

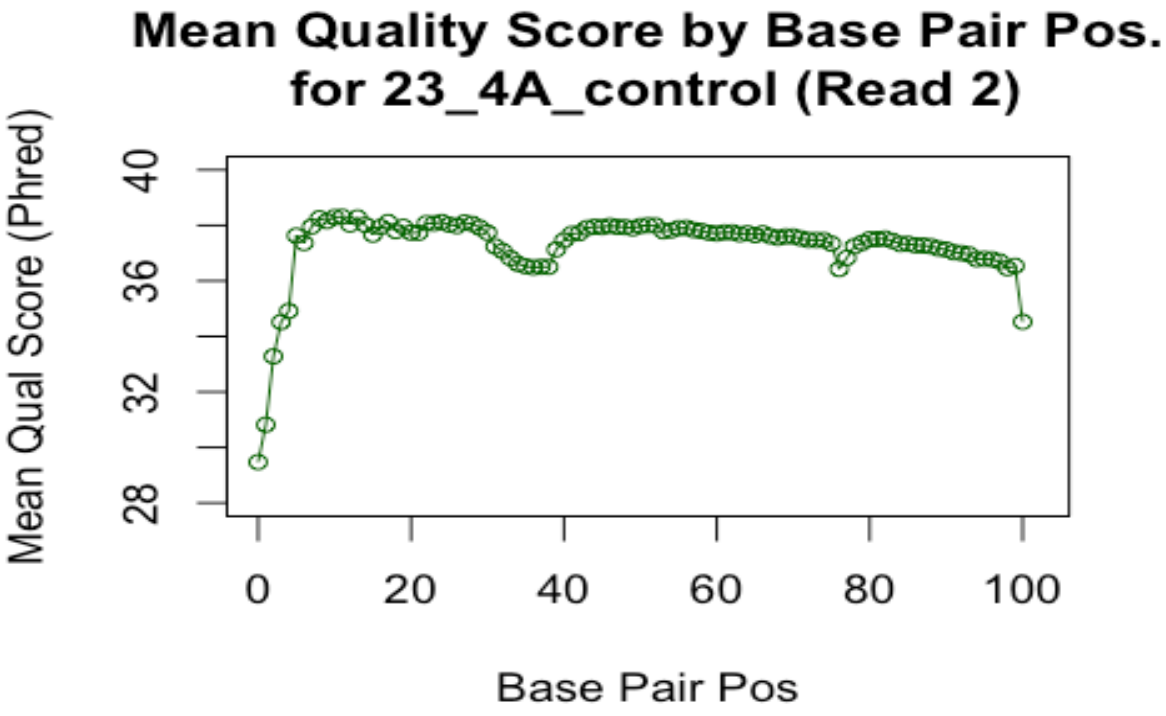




We see that these also very closely match, both in the trend of the quality scores (vast majority at 38 and above for both read sets) and the actual counts of the number of reads at each qual score. This gives us some confidence that our assessment of the data using python is accurate, since it agress with the results from FastQC for both plot types.

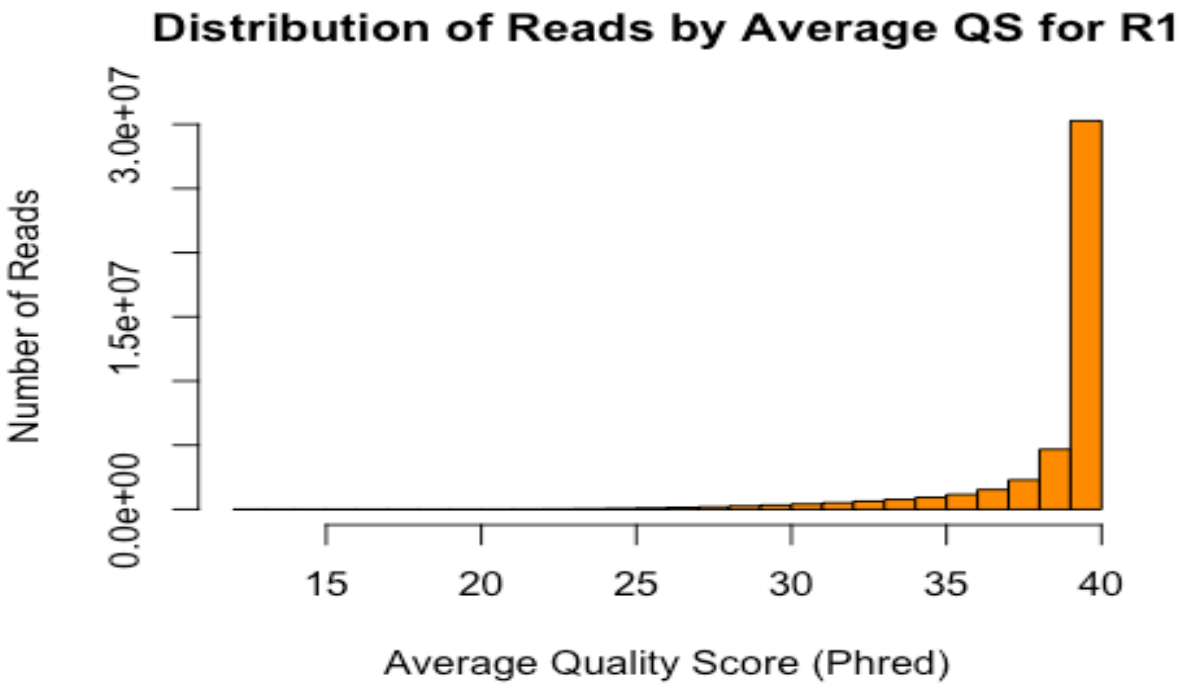
23_4A_control library:

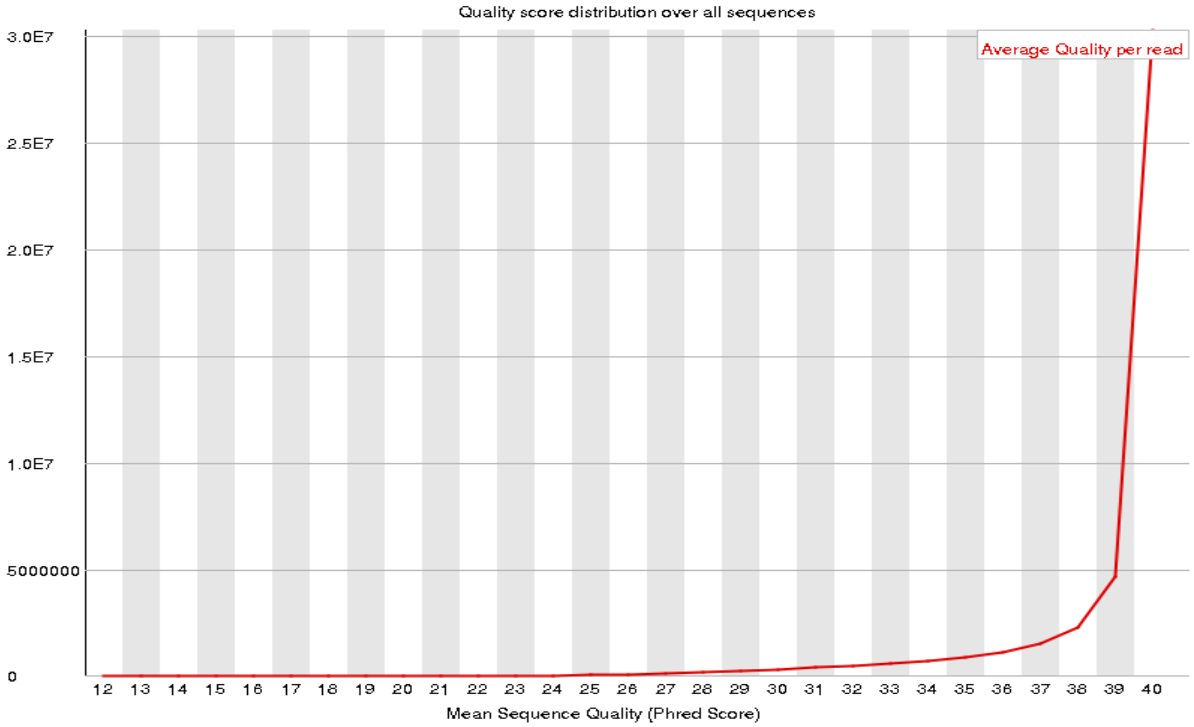


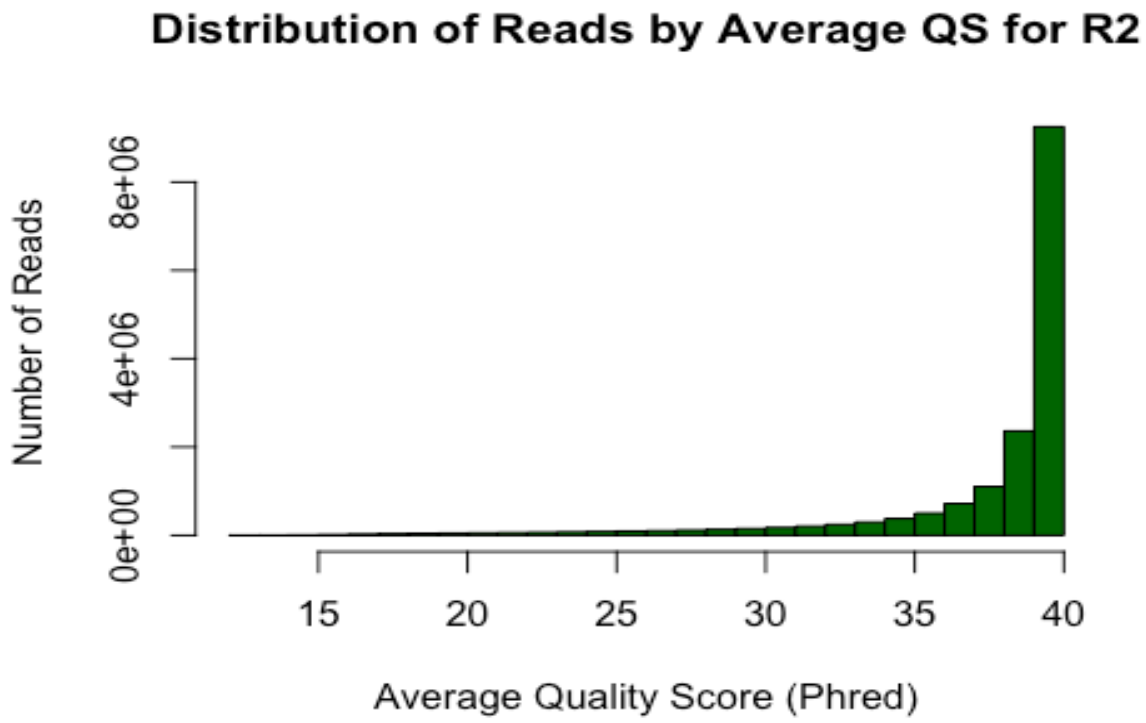


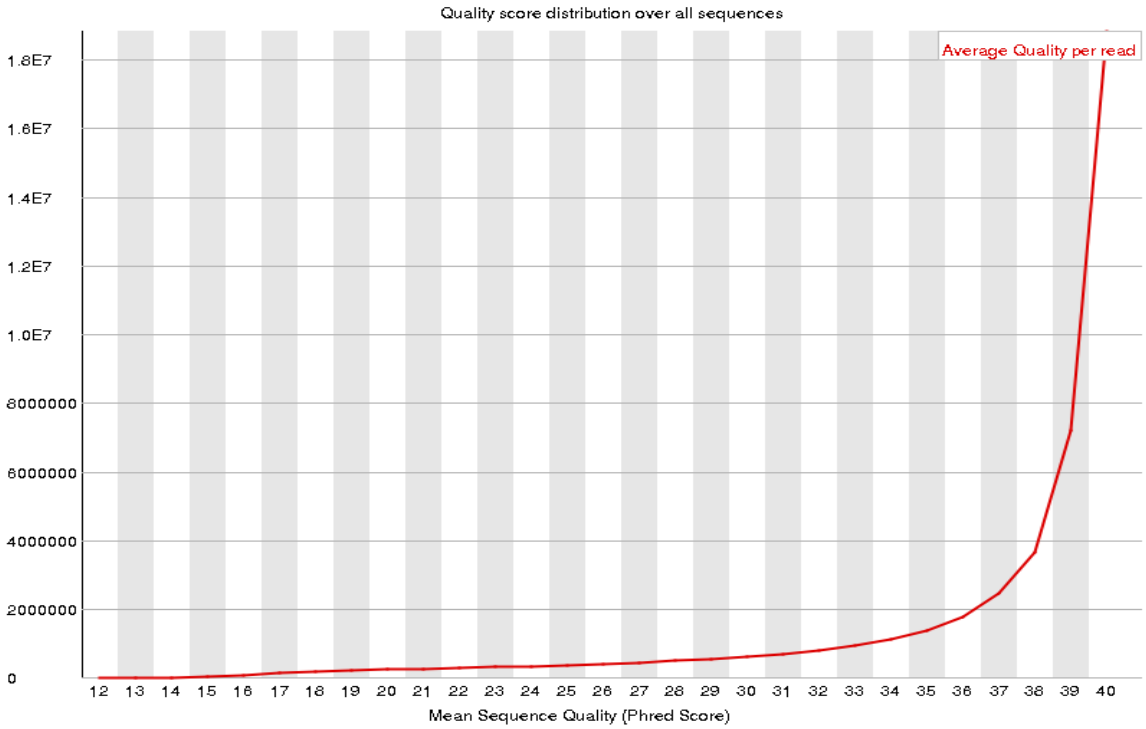
Like the 19_3F read sets, these graphs closely resemble the per-base position quality plots produced by the FastQC software.

In comparing the Quality score distributions for the forward and reverse reads between our python script results and FastQC, again, the plots closely resemble each other for both R1 and R2 sets:









Quality Assessment Script Runtimes:

One observation about the runtime of the python scripts on these two libraries is that they were significantly longer compared to the runtime needed to process the libraries by FastQC. Running a time check on the scripts indicate that the 19_3F_fox library (the smaller of the two

at 16348255 reads) took just 19 minutes, while the 23_4A_control library (the larger set, with 44303262 reads) completed in 53 minutes. The FastQC program completed analysis of both libraries in under 15 minutes, illustrating a much more efficent and perhaps powerful method of calculating these data sets, most likely due to being written in Java (a more efficent language). FastQC may also be subsetting data to function with a smaller data set and compiling the results as the process continues.

Part 2: Adaptor trimming comparison

There are several adapter trimming programs we can use to trim the reads from our libraries. `cutadapt` removes adapter sequences from sequencing reads by allowing the user to specify an adapter string for each read set (since we are using paired end reads); it does not automatically look for the reverse compliment of the adapter for the paired reads. `cutadapt` also allows for multiple adapter sequences to be specified (to allow for errors) and several options for setting min or max cutoff lengths and output formats, but only trims the best fitting strings from each read. This is also appears to be the only program that can handle 'N' characters in searching for adapter strings by default among the trimming options. `cutadapt` also only allows for fastq or FASTA formatting, but the files can be zipped.

`process_shortreads`, which is part of the stacks program, allows for a slightly more striaghtforward process. This program also takes in both sets of paired reads, but can take bam or BUSTARD, file formats as well as fastq formats. Additionally, while `process_shortreads` also requires both the forward and reverse compliment adapter sequences to be specified, it allows for a simple specification of the number of mismatches that are allowed when processing each read to determine if it should be trimmed (`cutadapt` allows a similar specification, '-e', but as a percentage of mismatches to the length of the matching string).

`Trimmomatic` uses similar input to the other programs (paired end fastq files) except when specifying the adapter sequences: instead of feeding the adapter as a string, the option 'ILLUMINACLIP' will automatically clip any illumina specific adapters from the reads. This means that this program is not as flexible as the other programs, and currently appears to not allow custom adapters to be trimmed. Additionally, the documentation indicates that it can only process fastq files (zipped or unzipped).

For this analysis, I've selected to use `cutadapt`, with default settings for trimming paired-end reads.

```
From Doug's adapter set, I have the following primers:
R1: GATCGGAAGAGCACACGTCTGAACTCCAGTCAC
R2: GATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT

# Command used to run cutadapt:
>$ time cutadapt -a GATCGGAAGAGCACACGTCTGAACTCCAGTCAC -A
GATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT -f fastq -o 19_3F_fox_R1.trimmed.fastq -p
19_3F_fox_R2.trimmed.fastq 19_3F_fox_S14_L008_R1_001.fastq
19_3F_fox_S14_L008_R2_001.fastq
>$ time cutadapt -a GATCGGAAGAGCACACGTCTGAACTCCAGTCAC -A
GATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT -f fastq -o 23_4A_control_R1.trimmed.fastq -p
```

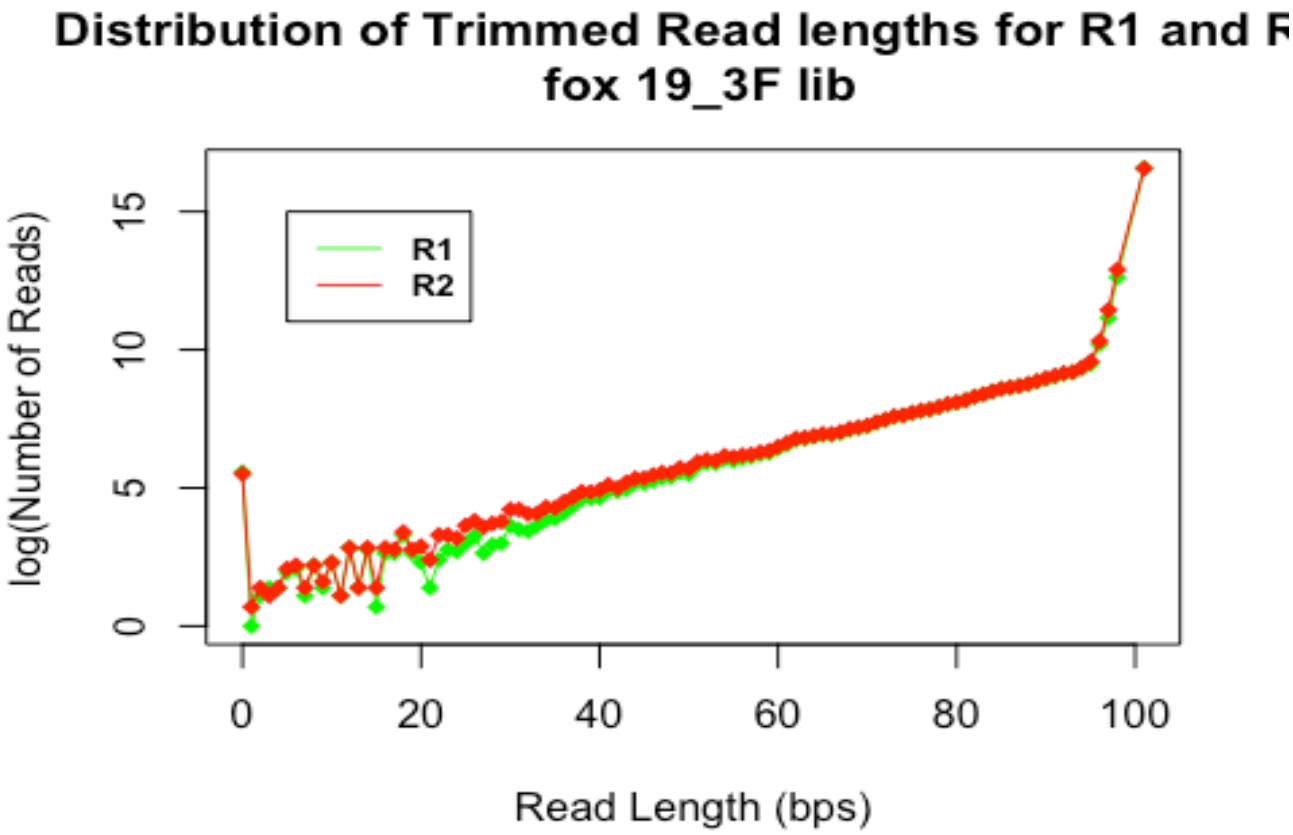
```
23_4A_control_R2.trimmed.fastq 23_4A_control_S17_L008_R1_001.fastq
23_4A_control_S17_L008_R2_001.fastq
```

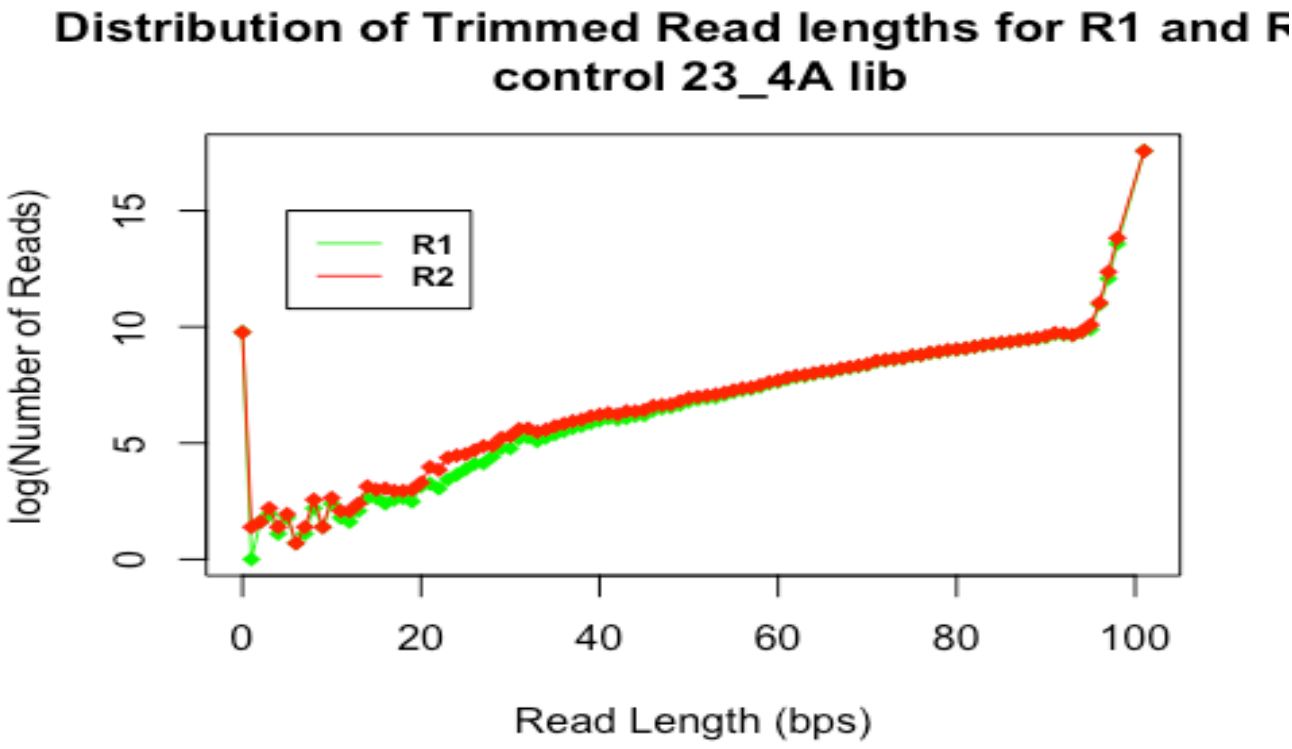
For the 19_3F_fox library, cutadapt trims 2.6% of the forward reads (R1) and 2.8% of the reserve reads (R2); approximately 426,800 and 463,200 reads of the 16 million in the library, respectively.

For the 23_4A_control library, cutadapt trims 2.4% of the forward reads (R1) and 2.6% of the reverse reads (R2); 1,077,000 and 1,163,000 million reads from each of the files, respectively, amongst the 44 million total reads.

Now that both libraries have been adapter trimmed, we can plot the distribution of resulting reads to see how they align against the average insert size of the libraries, based on the previous fragment analysis done prior to sequencing:

```
>$ grep -A 1 --no-group-separator "^@K00" 19_3F_fox_R1.trimmed.fastq | grep -v
"^@" | awk '{ print length($0); }' | sort | uniq -c | sort -g > fox_R1_dist.txt
>$ grep -A 1 --no-group-separator "^@K00" 19_3F_fox_R2.trimmed.fastq | grep -v
"^@" | awk '{ print length($0); }' | sort | uniq -c | sort -g > fox_R2_dist.txt
```





Are our adaptor trimming results are consistent with the insert size distributions?

Since both libraries have a large average size (between 380-420 bp, as determined by the fragment analysis done by Maggie) compared to the read size (101bp), we expect to see very few reads from out libraries that are trimmed because very few reads will actually contain any adapter sequences. This is consistent with our `cutadapt` results, as only 2-3% of the reads are trimmed in each library, illustrating that most of our reads contain actual RNA sequences instead of adapters. Furthermore, we would expect that as we have a distribution of read lengths that are even larger in the initial library, we would expect to see even fewer reads trimmed since even fewer reads would run through the adapter sequences, and the opposite occur if the average fragment size were to approach the size of the adapter.

Part 3: rRNA reads and strand-specificity

The rRNA mouse sequence data was pulled from the rFam database. Using the query search to filter to just rRNA sequences for Mus musculus, I pulled the appropriate fasta files and combined just the rRNA sequences into one file, which was then used to build my gmap database, before aligning both libraries against it using gsnap (Please see associated GSnap_sc.srun script for GMap and Gsnap bash commands run).

To determine the proportion of reads that likely originated from rRNA, we can count the number of reads that did NOT map to any rRNA sequences, and subtract that from the total number of reads to determine the portion of reads that did map to rRNA.

```
>$ grep -v "^@" 19_3f_fox_out.nomapping | wc -l
28246792
```

```
>$ grep -v "^@" 23_4A_control_out.nomapping | wc -l
69721054
```

The 19_3F_fox trimmed library has a total of 32696510 (16348255 x2) reads, which means about ~14.7% of reads mapped to rRNA sequences. The 23_4A_control library has a total of 88606524 (44303262 x2) reads, of which ~22.4% of all reads likely originated from rRNA based on the gsnap alignment.

This is about average based on previous publications into rRNA depeltion or selection-against methods (doi:10.1186/gb-2012-13-3-r23, doi:10.1038/srep41114), which indicate as little as 50% of rRNA can be filtered out (rRNA makes up ~80-85% of all RNA in the cell) of a library, to as much as 99%. These results for our fox and control libraries fit about right in the middle of that range.

How do we know our library is strand-specific?

Possibly the easiest way we can prove our libraries are strand specific is by querying them for Poly-A tail homoploymer regions. Because we know we used a poly-A tail selection method to select for mRNA sequences with these tails, we can expect some number of the reads in one read set (R1 or R2) to contain these repeated A's; but the complementary set of reads to have far fewer, and instead contain reverse complementary Poly-T sequences..

```
# For the 19_3F_fox libary:
>$ grep -A 1 "^@K00" 19_3F_fox_R2.trimmed.fastq | grep
"AAAAAAAAAAAAAAAAAAAAAA" | wc -l
4861
>$ grep -A 1 "^@K00" 19_3F_fox_R1.trimmed.fastq | grep
"AAAAAAAAAAAAAAAAAAAAAA" | wc -l
3492
>$ grep -A 1 "^@K00" 19_3F_fox_R2.trimmed.fastq | grep
"TTTTTTTTTTTTTTTTTTTTTT" | wc -l
11888
>$ grep -A 1 "^@K00" 19_3F_fox_R1.trimmed.fastq | grep
"TTTTTTTTTTTTTTTTTTTTTT" | wc -l
15425
```

So we see that the strands from R2 have a larger number of Poly-A sequences, while the R1 reads have a much larger number of poly-T sequences.

```
# For the 23_4A_control library:
>$ grep -A 1 "^@K00" 23_4A_control_R1.trimmed.fastq | grep
"AAAAAAAAAAAAAAAAAAAAAA" | wc -l
12187
>$ grep -A 1 "^@K00" 23_4A_control_R2.trimmed.fastq | grep
"AAAAAAAAAAAAAAAAAAAAAA" | wc -l
32776
>$ grep -A 1 "^@K00" 23_4A_control_R1.trimmed.fastq | grep
"TTTTTTTTTTTTTTTTTTTTTT" | wc -l
135271
>$ grep -A 1 "^@K00" 23_4A_control_R2.trimmed.fastq | grep
"TTTTTTTTTTTTTTTTTTTTTT" | wc -l
30172
```

We see a similar trend in the 23_4A_control library, which has a far greater number of poly-A sequences in R2 than in R1, and a much larger count of Poly-T sequences in R1 than R2.

Graphically, this relationship could be plotted like this: