Methods for Detecting Mutations in Non-model Organisms

by

Adam James Orr

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved November 2020 by the
Graduate Supervisory Committee:

Reed Cartwright, Chair
Melissa Wilson
Kenro Kusumi
Jesse Taylor
Susanne Pfeifer

ARIZONA STATE UNIVERSITY

December 2020

ABSTRACT

Next-generation sequencing is a powerful tool for detecting genetic variation. However, it is also error-prone, with error rates that are much larger than mutation rates. This can make mutation detection difficult; and while increasing sequencing depth can often help, sequence-specific errors and other non-random biases cannot be detected by increased depth. The problem of accurate genotyping is exacerbated when there is not a reference genome or other auxiliary information available.

I explore several methods for sensitively detecting mutations in non-model organisms using an example *Eucalyptus melliodora* individual. I use the structure of the tree to find bounds on its somatic mutation rate and evaluate several algorithms for variant calling. I find that conventional methods are suitable if the genome of a close relative can be adapted to the study organism. However, with structured data, a likelihood framework that is aware of this structure is more accurate. I use the techniques developed here to evaluate a reference-free variant calling algorithm.

I also use this data to evaluate a k-mer based base quality score recalibrator (KBBQ), a tool I developed to recalibrate base quality scores attached to sequencing data. Base quality scores can help detect errors in sequencing reads, but are often inaccurate. The most popular method for correcting this issue requires a known set of variant sites, which is unavailable in most cases. I simulate data and show that errors in this set of variant sites can cause calibration errors. I then show that KBBQ accurately recalibrates base quality scores while requiring no reference or other information and performs as well as other methods.

Finally, I use the *Eucalyptus* data to investigate the impact of quality score calibration on the quality of output variant calls and show that improved base quality score calibration increases the sensitivity and reduces the false positive rate of a variant calling algorithm.

DEDICATION

For my family. They taught me to be curious, tenacious, and persistent; to believe in myself, do my best, and never give up.

ACKNOWLEDGMENTS

Thanks to my advisor, whose input helped substantially improve this work and for giving me the space and support I needed to work and grow. Thanks to my committee, whose input substantially improved this dissertation and the work described herein. Thanks to the members of my lab past and present, who created a cordial, pleasant, and intellectually stimulating environment to work in and provided helpful feedback. Thanks to the Arizona State University School of Life Sciences and College of Liberal Arts and Sciences for the Spring 2020 Completion Fellowship, which greatly improved this dissertation. Thanks to the National Science Foundation, National Institutes of Health, and U.S.-Israel Binational Science Foundation which funded parts of this work.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1 INTRODUCTION 1

2 A PHYLOGENOMIC APPROACH REVEALS A LOW SOMATIC MUTATION RATE IN A LONG-LIVED PLANT 6

2.1 Abstract 6

2.2 Background 7

2.3 Materials and Methods 8

2.3.1 Field Sampling 8

2.3.2 DNA Extraction, Library Preparation, and Sequencing 9

2.3.3 Creation of a Pseudo-reference Genome 10

2.3.4 Variant Calling for Positive Control 11

2.3.5 Positive Control 12

2.3.6 Variant Calling for Estimating the Rate and Spectrum of Somatic Mutations 14

2.3.7 Estimation of the False-negative Rate 15

2.3.8 Estimation of the False-discovery Rate 16

2.4 Results and Discussion 16

2.4.1 Field Sampling and Sequencing 16

2.4.2 Positive Control Analysis 17

2.4.3 Estimation of the Somatic Mutation Rate 19

2.4.4 What Drives Differences in Somatic Mutation Rates Among Species? 22

CHAPTER Page

2.5 Data Accessibility 25
2.6 Authors' Contributions 25
2.7 Competing Interests 25
2.8 Funding 25
2.9 Footnotes 25
3 KBBQ: A REFERENCE-FREE METHOD FOR BASE QUALITY SCORE RECALIBRATION 31
3.1 Introduction 31
3.1.1 Quality Scores 31
3.1.2 Base Quality Score Recalibration 33
3.1.3 Alternate Approaches for BQSR 37
3.2 Methods 38
3.2.1 KBBQ Program Input and Parameters 39
3.2.2 Testing and Validation 41
3.3 Results 46
3.4 Discussion 48
3.5 Conclusion 50
4 EVALUATING THE IMPACT OF QUALITY SCORE CALIBRATION ON VARIANT CALLING 55
4.1 Introduction 55
4.2 Methods 57
4.3 Results 62
4.3.1 Variants Detected in *E. melliodora* 75
4.4 Discussion 77

CHAPTER Page

4.4.1 KBBQ Recalibration Yielded More *E. melliodora* Calls 83
4.5 Conclusion 84
5 CONCLUSION 88
REFERENCES 90
APPENDIX
A PERMISSION TO INCLUDE PREVIOUSLY PUBLISHED WORK 100
B SUPPLEMENTARY MATERIAL FOR CHAPTER 2 102
B.1 Effect of Biological Replicate Filters on Variant Calling 103
B.2 False Negative Rates Estimated for Each Branch of the Tree 104
B.3 Effects of the ExcessHet Filter on Variant Calling 105
B.4 Correlation Between Physical and Genetic Branch Lengths 106
B.5 Effect of Filtering on the Number of Variants 107
C SEARCHING FOR PHYLOGENETIC SIGNAL IN THE OUTPUT OF A REFERENCE-FREE VARIANT CALLER 109
C.1 Introduction 110
C.2 Methods 111
C.3 Results 112
C.4 Discussion 112
C.5 Conclusion 115
D CODE TO REPRODUCE ANALYSES IN APPENDIX C 118
E CODE TO REPRODUCE ANALYSES IN CHAPTER 3 126
E.1 Data Analysis Code 127
E.2 Code to Generate Plots 135
F KBBQ PROGRAM CODE 146

APPENDIX Page

F.1 Headers 147
F.2 Source 177
G CODE TO REPRODUCE ANALYSES IN CHAPTER 4 226
G.1 Code to Call Variants With CHM1-CHM13 Data 227
G.2 Code to Call Variants With Simulated Data 231
G.3 Code to Call Variants With *E. melliodora* Data 234
G.4 ROC Plotting Code 243
G.5 Summary Plotting Code 250

LIST OF TABLES

Table Page

3.1 KBBQ Parameters 42

3.2 Errors of Different Calibration Methods 47

3.3 Errors of Different Calibration Methods on Simulated Data 49

4.1 False Positive Calibration Errors 63

4.2 False Negative Calibration Errors 65

4.3 Combined Calibration Errors 65

4.4 Variant Caller Performance on Simulated Data 77

4.5 Number of Detected Variants After KBBQ Recalibration 78

4.6 Number of Detected Variants Using Uncalibrated Reads 79

B.1 The Effects of the Replicate Filter on the Detection of Variants 103

B.2 False Negative Rate in Each Branch of the Tree 105

B.3 The Effect of Different Filters on the Number of Sites Remaining Under Consideration 108

C.1 Number of Detected Variants After Each Filtering Step 112

C.2 Number of Sites With a Missing Genotype After Each Filtering Step . . 115

LIST OF FIGURES

Figure		Page
2.1	The *Eucalyptus melliodora* Individual Sequenced in This Study	9
2.2	Phylogenetic Trees Reconstructed From Somatic Mutations Resemble the Physical Structure of the Tree More Closely Than Expected by Chance	13
3.1	Base Quality Score Recalibration Example	36
3.2	Comparison of Calibration Methods	47
3.3	Comparison of Calibration Methods on Simulated Data	48
3.4	Quality Score Distribution of Recalibrated Simulated Data	49
4.1	False Positive Only Calibration	63
4.2	False Negative Only Calibration	64
4.3	Combined Calibration	66
4.4	Combined Calibration Heat Map	67
4.5	More Accurate Known Sites Increases Number of Unfiltered Positive Calls	68
4.6	Both False Negative and False Positive Rate Contribute to Increased Number of Unfiltered Positive Calls	69
4.7	Better Calibration Increases Sensitivity and Reduces Precision	69
4.8	F-statistic of Unfiltered Calls	70
4.9	Unfiltered ROC Curves	71
4.10	Filtered Positive Calls	73
4.11	Filtered Call Sensitivity and Precision	73
4.12	Sensitivity and Precision Before and After Filtering	74
4.13	Filtered F-statistic	74
4.14	Sensitivity and Precision of Calls on Simulated Reads	75

Figure Page

4.15 F-statistic of Calls on Simulated Reads 76
4.16 ROC of Calls on Simulated Reads 76
B.1 The Relationship Between Physical Branch Length and Genetic Branch Length 106
B.2 Physical Branch Length and Genetic Branch Length Not Forced Through Origin 107
C.1 Trees Inferred at Each Filtering Step 114

Chapter 1

INTRODUCTION

Next-generation sequencing is a powerful technology with numerous applications. It has enabled data collection and hypothesis testing at an unprecedented scale, with an enormous amount of sequencing data generated daily (Kodama et al. 2012). The high-throughput nature of modern sequencing technologies makes it cheap and efficient to genotype one or more samples. Additionally, next-generation sequencing serves as the basis of a variety of methods for interrogating a wide array of molecular functions and epigenetic states including RNA-seq (Nagalakshmi et al. 2008), ChIP-seq (Johnson et al. 2007), DNase-seq (Boyle et al. 2008), MNAse-seq (Schones et al. 2008), ATAC-seq (Buenrostro et al. 2015), bisfulfite sequencing (Frommer et al. 1992), and many more. As a testament to how powerful and useful the technology is for understanding the natural world, it has achieved such widespread use despite being highly error-prone (Fox et al. 2014; Meacham et al. 2011; Nakamura et al. 2011). DNA sequencing technology continues to improve; increasing read length and reduced substitution, insertion, and deletion errors are seemingly inevitable as new methods develop and the technology is refined (Branton et al. 2008; Eid et al. 2009; Fox et al. 2014). Yet even the most sophisticated and accurate sequencing platforms will require analysis with sound statistical methods to turn raw data into interesting and useful insights about the natural world.

In this dissertation, I explore and develop methods for gaining insight from noisy next-generation sequencing data in challenging contexts. In particular, I focus on genotyping and detecting mutations in non-model organisms. While there has been an explosion in the amount of sequencing data generated since the advent of the

technology, most species on the planet have still not been sequenced; even fewer have complete and accurate reference genomes (Lewin et al. 2018). Yet, funding available specifically for the generation and refinement of new reference genomes is difficult to acquire (Richards 2015). Model organisms are immensely useful for generating knowledge and are vital to gaining understanding of the world; however, it is clear that model organisms do not hold the answers to every interesting question. As such, it is important and necessary to develop methods for data analysis that work well even with limited *a priori* knowledge.

In the second chapter, I present collaborative work on sensitively detecting somatic mutations in the non-model eucalypt, *Eucalyptus melliodora*. This work was previously published in *Proceedings of the Royal Society B* and documents my and my coauthors' efforts to estimate the somatic mutation rate of an individual eucalypt. As a control, I used a conventional variant calling algorithm paired with custom filters to investigate whether the physical tree structure of the organism also gives it a tree-like genetic structure, as one may predict based on how trees grow and mature. I also investigate the limits of this approach and find bounds on the estimates of several population genetic parameters we are able to estimate from the data.

The sequencing data and calls made in the second chapter are used throughout this dissertation as a basis of comparison for many of the techniques I use. The third chapter details my investigation of base quality score recalibration, a technique used to fix inaccurate quality measurements attached to sequencing data. I introduce an efficient software program `kbbq` to perform base quality score recalibration on whole genome sequencing reads without a reference genome or other *a priori* knowledge and show it performs just as well as the most common method for base quality score recalibration.

Errors are common relative to mutations in sequencing data, (Fox et al. 2014; Wu

et al. 2017) and since correlated errors are by definition non-independent, such errors cannot be detected by increased sequencing depth alone (Meacham et al. 2011; Taub, Corrada Bravo, and Irizarry 2010). Correcting quality score calibration issues in sequencing reads could in principle improve the ability of a variant caller to distinguish between true variation and noise in a way that increased depth cannot. While there is some evidence that it does help increase sensitivity (Ni and Stoneking 2016), it is an open question whether the benefits of base quality score recalibration outweigh the time, computational cost, and risk of introducing additional error. While the process has long been a part of the GATK Best Practices pipeline, it has recently fallen out of favor (Van der Auwera 2020a). In the fourth chapter, I investigate the effect of base quality score recalibration on variant caller sensitivity and specificity.

All together, this work explores many avenues for detecting variants in non-model organisms. There are many ways to do so, and many more that aren't discussed here. Each has strengths and weaknesses, but understanding what these are is critical to success. While choosing the correct method can be a challenging and daunting task, with careful preparation and rigorous examination of the available methods, any researcher can achieve satisfactory results.

REFERENCES

Boyle, Alan P., Sean Davis, Hennady P. Shulha, Paul Meltzer, Elliott H. Margulies, Zhiping Weng, Terrence S. Furey, and Gregory E. Crawford. 2008. "High-resolution mapping and characterization of open chromatin across the genome." *Cell* 132 (2): 311–322. https://doi.org/10.1016/j.cell.2007.12.014.

Branton, Daniel, David W Deamer, Andre Marziali, Hagan Bayley, Steven A Benner, Thomas Butler, Massimiliano Di Ventra, et al. 2008. "The potential and challenges of nanopore sequencing." *Nature Biotechnology* 26 (10): 1146–1153. https://doi.org/10.1038/nbt.1495.

Buenrostro, Jason, Beijing Wu, Howard Chang, and William Greenleaf. 2015. "ATAC-seq: A Method for Assaying Chromatin Accessibility Genome-Wide." *Current protocols in molecular biology / edited by Frederick M. Ausubel ... [et al.]* 109:21.29.1–21.29.9. https://doi.org/10.1002/0471142727.mb2129s109.

Eid, John, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, et al. 2009. "Real-Time DNA Sequencing from Single Polymerase Molecules." *Science* 323 (5910): 133–138. https://doi.org/10.1126/science.1162986.

Fox, Edward J., Kate S. Reid-Bayliss, Mary J. Emond, and Lawrence A. Loeb. 2014. "Accuracy of Next Generation Sequencing Platforms." *Next generation, sequencing & applications* 1. https://doi.org/10.4172/jngsa.1000106.

Frommer, M., L. E. McDonald, D. S. Millar, C. M. Collis, F. Watt, G. W. Grigg, P. L. Molloy, and C. L. Paul. 1992. "A genomic sequencing protocol that yields a positive display of 5-methylcytosine residues in individual DNA strands." *Proceedings of the National Academy of Sciences* 89 (5): 1827–1831. https://doi.org/10.1073/pnas.89.5.1827.

Johnson, David S., Ali Mortazavi, Richard M. Myers, and Barbara Wold. 2007. "Genome-Wide Mapping of in Vivo Protein-DNA Interactions." *Science* 316 (5830): 1497–1502. https://doi.org/10.1126/science.1141319.

Kodama, Yuichi, on behalf of the International Nucleotide Sequence Database Collaboration, Martin Shumway, on behalf of the International Nucleotide Sequence Database Collaboration, Rasko Leinonen, and on behalf of the International Nucleotide Sequence Database Collaboration. 2012. "The sequence read archive: explosive growth of sequencing data." *Nucleic Acids Research* 40 (D1): D54–D56. https://doi.org/10.1093/nar/gkr854.

Lewin, Harris A., Gene E. Robinson, W. John Kress, William J. Baker, Jonathan Coddington, Keith A. Crandall, Richard Durbin, et al. 2018. "Earth BioGenome Project: Sequencing life for the future of life." *Proceedings of the National Academy of Sciences* 115 (17): 4325–4333. https://doi.org/10.1073/pnas.1720115115.

Meacham, Frazer, Dario Boffelli, Joseph Dhahbi, David IK Martin, Meromit Singer, and Lior Pachter. 2011. "Identification and correction of systematic error in high-throughput sequence data." *BMC Bioinformatics* 12 (1): 451. https://doi.org/10.1186/1471-2105-12-451.

Nagalakshmi, Ugrappa, Zhong Wang, Karl Waern, Chong Shou, Debasish Raha, Mark Gerstein, and Michael Snyder. 2008. "The Transcriptional Landscape of the Yeast Genome Defined by RNA Sequencing." *Science (New York, N.Y.)* 320 (5881): 1344–1349. https://doi.org/10.1126/science.1158441.

Nakamura, Kensuke, Taku Oshima, Takuya Morimoto, Shun Ikeda, Hirofumi Yoshikawa, Yuh Shiwa, Shu Ishikawa, et al. 2011. "Sequence-specific error profile of Illumina sequencers." *Nucleic Acids Research* 39 (13): e90. https://doi.org/10.1093/nar/gkr344.

Ni, Shengyu, and Mark Stoneking. 2016. "Improvement in detection of minor alleles in next generation sequencing by base quality recalibration." *BMC genomics* 17:139. https://doi.org/10.1186/s12864-016-2463-2.

Richards, Stephen. 2015. "It's more than stamp collecting: how genome sequencing can unify biological research." *Trends in Genetics* 31 (7): 411–421. https://doi.org/10.1016/j.tig.2015.04.007.

Schones, Dustin E., Kairong Cui, Suresh Cuddapah, Tae-Young Roh, Artem Barski, Zhibin Wang, Gang Wei, and Keji Zhao. 2008. "Dynamic Regulation of Nucleosome Positioning in the Human Genome." *Cell* 132 (5): 887–898. https://doi.org/10.1016/j.cell.2008.02.022.

Taub, Margaret A, Hector Corrada Bravo, and Rafael A Irizarry. 2010. "Overcoming bias and systematic errors in next generation sequencing data." *Genome Medicine* 2 (12): 87. https://doi.org/10.1186/gm208.

Van der Auwera, Geraldine A. 2020a. "Geraldine Van der Auwera on Twitter." Twitter. https://twitter.com/VdaGeraldine/status/1311000033127550976.

Wu, Steven H, Rachel S Schwartz, David J Winter, Donald F Conrad, and Reed A Cartwright. 2017. "Estimating error models for whole genome sequencing using mixtures of Dirichlet-multinomial distributions." *Bioinformatics* 33 (15): 2322–2329. https://doi.org/10.1093/bioinformatics/btx133.

Chapter 2

A PHYLOGENOMIC APPROACH REVEALS A LOW SOMATIC MUTATION RATE IN A LONG-LIVED PLANT

by Adam J. Orr†, Amanda Padovan†, David Kainer†, Carsten Külheim, Lindell Bromham, Carlos Bustos-Segura, William Foley, Tonya Haff, Ji-Fan Hsieh, Alejandro Morales-Suarez, Reed A. Cartwright, and Robert Lanfear

†*Equal contribution.*

This chapter was originally published as an article in Proceedings of the Royal Society B *with* doi:10.1098/rspb.2019.2364.

2.1 Abstract

Somatic mutations can have important effects on the life history, ecology, and evolution of plants, but the rate at which they accumulate is poorly understood and difficult to measure directly. Here, we develop a method to measure somatic mutations in individual plants and use it to estimate the somatic mutation rate in a large, long-lived, phenotypically mosaic *Eucalyptus melliodora* tree. Despite being 100 times larger than *Arabidopsis*, this tree has a per-generation mutation rate only ten times greater, which suggests that this species may have evolved mechanisms to reduce the mutation rate per unit of growth. This adds to a growing body of evidence that illuminates the correlated evolutionary shifts in mutation rate and life history in plants.

2.2 Background

Trees grow from multiple meristems which contain stem cells that divide to produce the somatic and reproductive tissues. A mutation occurring in a meristematic cell will be passed on to all resulting tissues, potentially causing an entire branch including leaves, stems, flowers, seeds, and pollen to have a genotype different from the rest of the plant (Plomion et al. 2018; Schmid-Siegert et al. 2017). These different genotypes may lead to phenotypic changes, potentially with important consequences for plant ecology and evolution (Ally, Ritland, and Otto 2010; Buss 1983; Klekowski and Godfrey 1989; Sutherland and Watkinson 1986; Walbot 1985; Whitham and Slobodchikoff 1981). For example, somatic mutations could explain how long-lived plants adapt to changing ecological conditions (Gill et al. 1995), and are thought to influence long-term variation in the rates of evolution and speciation among plant lineages (Lanfear et al. 2013). Somatic mutations can degrade genetic stocks used in agriculture and forestry (Khoury, Laliberté, and Guarino 2010; Schoen, David, and Bataillon 1998), confer herbicide resistance to weed species, (Michel et al. 2004) and have been linked to declining plant fitness in polluted areas (Klekowski et al. 1994). However, despite the importance of somatic mutations and recent progress in understanding them (Bobiwash, Schultz, and Schoen 2013; Hanlon, Otto, and Aitken 2019; Plomion et al. 2018; Schmid-Siegert et al. 2017; Schultz and Scofield 2009; Wang et al. 2019), there remain significant analytical challenges in inferring somatic mutation rates from sequencing data in plants.

We present a solution to the challenges of measuring the somatic mutation rate that leverages the phylogeny-like structure of the plant itself to estimate the genome-wide somatic mutation rate of the individual. Our strategy has three key features. First, we sequence the full genome of three biological replicates of eight branch tips.

Using three biological replicates per branch tip significantly reduces the false-positive rate, because many types of error (e.g. sequencing error or mutations induced during DNA extraction or library preparation) are very unlikely to appear at the same position in all three replicates, making it easy to distinguish these errors from biological signal. Second, our strategy includes an inbuilt positive control, because we can ask whether the phylogenetic tree we reconstruct from the set of putative somatic mutations across the eight branch tips reflects the known physical structure of the tree (i.e. whether phylogeny correctly reconstructs ontogeny, as is expected for plant development in most cases, but see below). Third, the approach allows us to estimate the false-negative rate and the false-discovery rate of our inferences directly from the replicate samples (see below).

We applied this approach to a long-lived yellow box (*Eucalyptus melliodora*) tree, notable for its phenotypic mosaicism: a single large branch in this individual is resistant to defoliation by Christmas beetles (*Anoplognathus* spp Coleoptera: Scarabaeidae) due to stable differences in leaf chemistry and gene expression (Edwards et al. 1990; Padovan et al. 2013). We find that the rate of somatic mutation per generation is relatively high, but the rate per metre of growth is surprisingly low in comparison to other species. We suggest potential proximate and ultimate reasons for the wide variation in somatic mutation rates across plants.

2.3 Materials and Methods

2.3.1 Field Sampling

We used a known mosaic *E. melliodora* (yellow box). This tree is found near Yeoval, NSW, Australia (-32.75°, 148.65°). We collected the ends of eight branches in the canopy (figure 2.1). Branches were collected using an elevated platform mounted

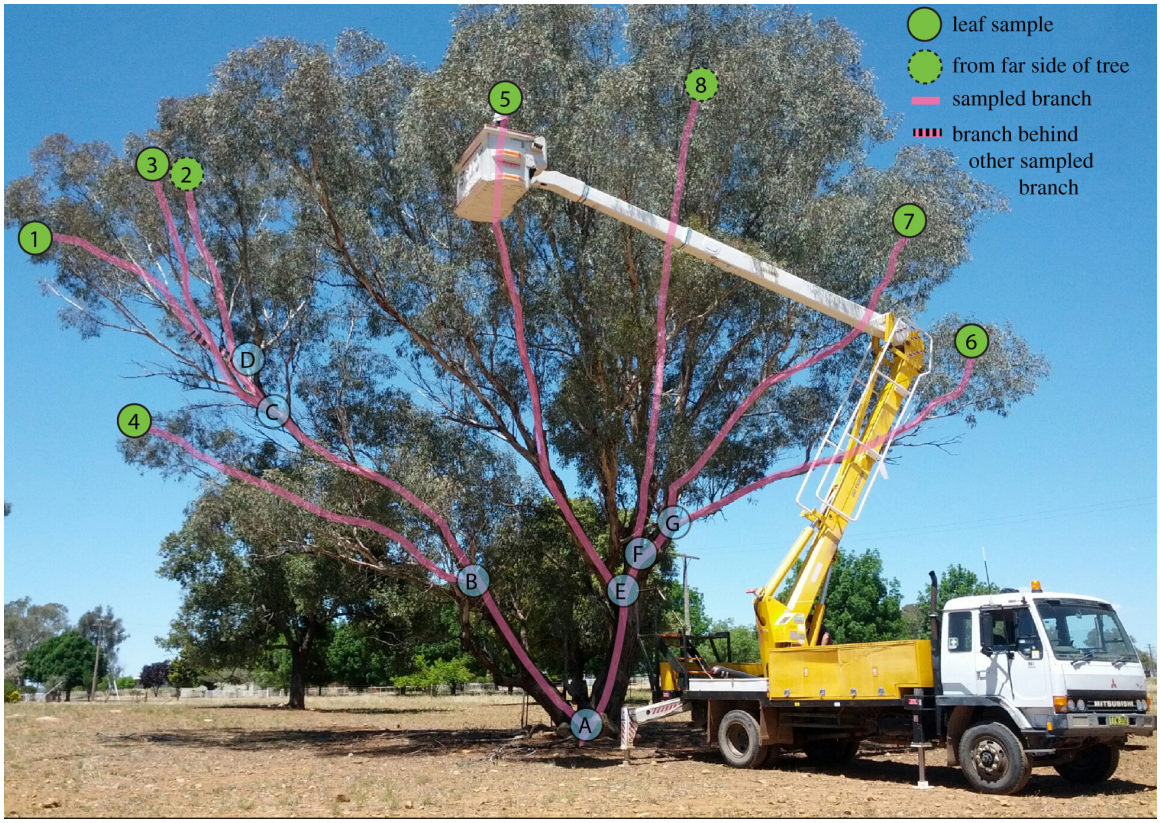


Figure 2.1: The *Eucalyptus melliodora* Individual Sequenced in This Study | The eight branch tips sampled are shown by numbered green circles with internal nodes of the tree shown as letters in blue circles. Circles with dashed outlines are from the far side of the tree. Pink lines trace the physical branches that connect the sampled tips. The herbivore-resistant branch comprises samples 1–3.

on a truck and were placed into labelled and sealed polyethylene bags which were immediately buried in dry ice in the field. Within the 24 h of collection, the samples were transferred to -80°C until DNA extraction. Simultaneously, we used a thin rope to trace each branch from the tip to the main stem. These rope lengths were measured to determine the lengths of the physical branches of the tree.

2.3.2 DNA Extraction, Library Preparation, and Sequencing

The branches were maintained below -80°C on dry ice and in liquid nitrogen while sub-sampled in the laboratory. From each branch, we selected a branch tip

which had at least three consecutive leaves still attached to the stem. From this branch tip, we independently sub-sampled roughly 100 mg of leaf from the 'tip-side' of the mid-vein on three consecutive leaves using a single hole punch into a labelled microcentrifuge tube containing two 3.5 mm tungsten carbide beads. The sealed tube was submerged in liquid nitrogen before the leaf material was ground in a Qiagen TissueLyser (Qiagen, Venlo, Netherlands) at 30 Hz in 30 s intervals before being submerged in liquid nitrogen again. This was repeated until the leaf tissue was a consistent powder, up to a total of 3.5 min grinding time.

DNA was extracted from this leaf powder using the Qiagen DNeasy Plant Mini Kit (Qiagen, Venlo, The Netherlands), following the manufacturer's instructions. DNA was eluted in 100 µl of elution buffer. DNA quality was assessed by gel electrophoresis (1% agarose in 1 × TAE containing ethidium bromide), and quantity was determined by Qubit Fluorometry (Invitrogen, California, USA) following manufacturer's instructions.

We used a Bioruptor (Diagnode, Seraing (Ougrée), Belgium) to fragment 1 g of DNA to an average size of 300 bp (35 s on 'High', 30 s off for 35 cycles at 4°C). The fragmented DNA was purified using 1.6 × SeraMag Magnetic Beads (GE Life-Sciences, Illinois, USA) following the manufacturer's instructions. We used Illumina TruSeq DNA Sample Preparation kit (Illumina Inc., California, USA) following the manufacturer's instructions to generate paired-end libraries for sequencing. These libraries were sequenced on an Illumina HiSeq 2500 (Illumina Inc., California, USA) at the Biomolecular Resource Facility at the Australian National University, Canberra.

2.3.3 Creation of a Pseudo-reference Genome

Since there is no available reference genome for E. melliodora, we created a pseudo-reference genome by iterative mapping and consensus calling. To do this, we first

mapped all of our reads to version 2.1 of the E. grandis reference genome (Bartholomé et al. 2015) using NGM (Sedlazeck, Rescheneder, and Haeseler 2013) and then updated the E. grandis reference genome using bcftools consensus (Li 2011a). We iteratively repeated this procedure until we saw only marginal improvement in the number of unmapped reads and reads that mapped with a mapping quality of zero. The alignment originally contained 67 M unmapped reads and 311 M reads that mapped with zero mapping quality, out of a total of 1792 M reads. After the first iteration, the alignment contained 61 M unmapped reads and 349 M reads that mapped with zero mapping quality. After the last iteration, the alignment contained 59 M unmapped reads and 311 M reads that mapped with zero mapping quality. The consensus of this alignment served as the reference for all further downstream analyses.

2.3.4 Variant Calling for Positive Control

To call variants for the positive control, we mapped each replicate of each branch tip (24 samples in total) to the final pseudo-reference genome using NGM and called genotypes using GATK 4 according to the GATK best practices workflow (Auwera et al. 2013). This resulted in a full genome alignment of all 24 samples (three replicates of eight branches) and produced an initial set of 9,679,544 potential variable sites, a number which includes all heterozygous sites in the genome.

We then filtered variants to minimize the false-positive rate by retaining only those sites in which: (i) genotype calls were identical within all three replicates of each branch tip (see also electronic supplementary material, §1); (ii) at least one branch tip had a different genotype than the other branch tips; (iii) the site is biallelic, since multiple somatic mutations are likely to be extremely rare; (iv) the total depth across all samples is less than or equal to 500 (i.e. roughly twice the expected depth of 240×), since excessive depth is a signal of alignment issues; (v) the ExcessHet

annotation was less than or equal to 40, since excessive heterozygosity at a site is a sign of genotyping errors, particularly in a site that is actually uniformly heterozygous throughout the tree but at which genotyping errors have caused a mutation to be called; and (vi) the site is not in a repetitive region determined by a lift-over of the E. grandis RepeatMasker annotation, as variation in repeat regions is often due to alignment error. This filtering produced a set of 99 high-confidence sites containing putative somatic mutations. The number of mutations that remained after the application of each filter is described in §5 of the electronic supplementary material.

2.3.5 Positive Control

Using the set of 99 high-confidence putative somatic mutations, we use the Phangorn package in R (Schliep 2011) to calculate the parsimony score of all 10,395 possible phylogenetic trees of eight taxa. This estimates the number of somatic mutations that would be required to explain each of the 10,395 phylogenetic trees, using the Fitch algorithm implemented in the Phangorn package. Of these trees, three had the maximum-parsimony score of 78. One of these three trees matched the topology of the physical tree (figure 2.2).

Next, we calculated the path difference (PD) between all 10,395 trees and the physical tree topology. The PD measures differences between two phylogenetic tree topologies (Steel and Penny 1993) by comparing the differences between the path lengths of all pairs of taxa. Here we use the variant of the PD that treats all branch lengths as equal, because we are interested only in topological differences between trees, not branch length differences. Comparing all 10,395 trees to the physical tree topology provides a null distribution of PDs between all trees and the physical tree topology, which we can use to ask whether each of the three maximum-parsimony trees is more similar to the physical tree topology than would be expected by chance.

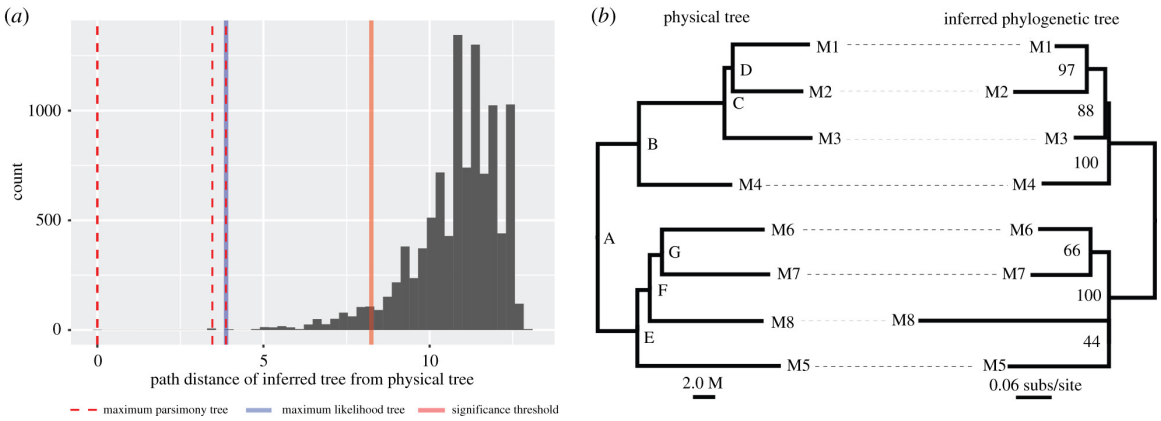


Figure 2.2: Phylogenetic Trees Reconstructed From Somatic Mutations Resemble the Physical Structure of the Tree More Closely Than Expected by Chance | (a) The PD between the physical tree (figure 2.1) and all 10,395 possible phylogenetic trees of eight taxa is shown as a histogram. A tree with the same topology as the physical tree will have a PD of 0. The solid red line represents the boundary of the smallest 5% of the distribution of PDs, such that a tree with a PD lower than this line is more similar to the physical tree than expected by chance. All of the maximum-parsimony trees (dashed red lines) and the one maximum-likelihood tree (solid blue line) are more similar to the physical tree than expected by chance. (b) A side-by-side comparison of the physical tree (left, branch lengths in metres) and the maximum-likelihood tree (right, branch lengths in substitutions per site) inferred with the JC model. Letters on the nodes of the physical tree (left) correspond to the same letters of internal nodes in figure 1. Numbers on the maximum-likelihood tree (right) are bootstrap percentages. There is a single difference between the two trees: the inferred tree groups samples M8 and M5 together with low bootstrap support (44%), which is a grouping that does not occur in the physical tree.

To do this, we simply ask whether the PD of each of the three observed maximum-parsimony trees falls within the lower 5% of the distribution of PDs from all 10,395 trees. This was the case for all three maximum-parsimony trees ($p < 0.001$ in all cases; figure 2.2), suggesting that our data contain biological signal which render the phylogenetic trees reconstructed from somatic mutations more similar than would be expected by chance to the physical tree.

2.3.6 *Variant Calling for Estimating the Rate and Spectrum of Somatic Mutations*

Using the physical tree topology to define the relationship between samples, we called somatic mutations using DeNovoGear's dng-call method (Ramu et al. 2013) compiled from https://github.com/denovogear/denovogear/tree/3ae70ba. Model parameters were estimated from 3-fold degenerate sites in our NGM alignment, via VCFs generated by bcftools mpileup and bcftools call with–pval-threshold = 0. We estimated maximum-likelihood parameters using the Nelder–Mead numerical optimization algorithm implemented in the R package dfoptim (https://cran.r-project.org/package=dfoptim). We then called genotypes using the GATK best practices workflow as above, but with the standard-min-confidence-threshold-for-calling argument set to 0, causing the output VCF to contain every potentially variable site in the alignment. Thus, we used GATK to generate high-quality pileups from our alignments. These pileups were then analysed by dng-call to identify (i) heterozygous sites and (ii) de novo somatic mutations. Since successful haplotype construction in a region indicates a high-quality alignment, we used Whatshap 0.16 (Martin et al. 2016) to generate haplotype blocks from the heterozygous sites.

Next, we filtered our de novo variant set to remove potential false positives. We removed variants that (i) were on a haplotype block with a size less than 500 nucleotides (among other things, this filter will remove many putative variants that fall in long repeat regions); (ii) were within 1000 nucleotides of another de novo variant (indicative of alignment issues such as might occur in repeats and other regions); (iii) had an log likelihood of the data (LLD) score less than -5 (indicative of poor model fit); and (iv) had a de novo mutation probability (DNP) score less than 0.99999 (retaining only the highest confidence variants). This produced a final variant set of 90 variants.

2.3.7 Estimation of the False-negative Rate

To estimate the number of mutations that we were likely to have filtered out in our variant calling pipeline, we used the method of Ness et al. (2012), adapted to the current phylogenetic framework. Specifically, we randomly selected 14,000 sites from the first 11 scaffolds of the pseudo-reference genome and randomly assigned 1000 of these sites to each of the 14 branches on the tree. For each of these sites, we induced *in silico* mutations into the raw reads with a three-step procedure. We first estimated the observed genotype at the root using DeNovoGear call at each site. We then chose a mutant genotype by mutating one of the alleles to a randomly chosen different base using a transition/transversion ratio of 2, reflecting the observed transition/transversion ratio of eucalypts. We edited the raw reads as follows: for each mutation, we defined the samples to be mutated as all of those samples that descend from the branch on which the *in silico* mutation occurred. For example, an *in silico* mutation occurring on branch B $\rightarrow$ C in figure 2.1 would affect all three replicates of samples 1, 2 and 3. We then edited the reads that align to the site in question to reflect the new mutation, depending on whether the reference genotype was homozygous or heterozygous. For homozygous sites, we selected the number of reads to mutate by generating a binomially distributed random number with a probability of 0.5 and a number of observations equal to the number of reads with the reference genotype. We then randomly selected that the number of reads with the reference allele to mutate to the mutant allele and edited the raw reads accordingly. For a heterozygous site, we edited the reads to replace all occurrences of the reference allele to mutant allele. The result of this procedure is the generation of a new set of raw fastq files, which now contain information on 1000 *in silico* mutations for every branch in the physical tree.

2.3.8 Estimation of the False-discovery Rate

To determine the false-discovery rate of the variant calling pipeline, we simulated random trees of our samples (where each of the eight branches is represented by three tips that denote the three replicates of that branch) by shuffling the tip labels until the tree had a maximal Robinson-Foulds distance from the original tree. This 24-taxon tree shares no splits with the original 24-taxon tree, so any phylogenetic information should be removed. We simulated 100 such trees and called variants using the pipeline above, but assuming that these trees were the physical tree, and ignoring any sites we had previously called as variable. Thus, any variants called by the pipeline must be false positives. We recovered 11 false positive calls over 100 simulations (i.e. 0.11 false-positive mutations per simulation), indicating our false-discovery rate is approximately 0.12%. We calculated the false-discovery rate only once, after the details of the pipeline were finalized, to avoid overfitting our pipeline to artefactually reduce the false-discovery rate.

2.4 Results and Discussion

2.4.1 Field Sampling and Sequencing

We selected eight branch tips that maximized the intervening physical branch length on the tree (figure 2.1), reasoning that this would increase our power by maximizing the number of sampled cell divisions and thus somatic mutations. We performed independent DNA extractions from three leaves from each branch tip, prepared three independent libraries for Illumina sequencing and sequenced each library to 10× coverage (assuming a roughly 500 Mbp genome size, as is commonly observed in *Eucalyptus* species (Grattapaglia et al. 2012)) using 100 bp paired-end sequencing on an Illumina HiSeq 2500. Quality control of the sequence data verified that each

sample was sequenced to approximately 10× coverage and that each branch tip was therefore sequenced to approximately 30× coverage.

2.4.2 Positive Control Analysis

We first performed a positive control to confirm that the phylogeny of a set of high-confidence somatic variants matches the physical structure of the tree. This approach relies on being able to infer the ontogeny of the tree with sufficient accuracy that a valid comparison can be made between the ontogeny of the tree and a phylogeny generated from that tree's somatic variants. Documenting a plant's ontogeny with sufficient accuracy may not be possible for all plant species or individuals. Nevertheless, the physical structure of the tree we studied was clear (figure 2.1), and although *Eucalyptus* trees are known to frequently lose branches, branch loss and regrowth should not affect the correlation between ontogeny and phylogeny provided that sufficient mutations accumulate during cell replication. To perform the phylogenetic positive control, we created a pseudo-reference genome using our data to update the genome of *E. grandis* (see methods). We then called variants using GATK (McKenna et al. 2010) in all three replicates of all eight branch tips and used a set of strict filters (see methods and supplementary information) designed to avoid false-positive mutations in order to arrive at an alignment of 99 high-confidence somatic variants. To find the phylogenetic trees that best explain this alignment, we calculated the alignment's parsimony score on all 10,395 possible phylogenetic trees of eight samples. Parsimony is an appropriate method here because we do not expect more than one mutation to occur at any single site on any single branch of the *E. melliodora* tree. We then asked whether the three phylogenetic trees with the most parsimonious scores were more similar to the physical structure of the tree than would be expected by chance. To do this, we calculated the PD between the structure of

the physical tree and each of the three most parsimonious trees. We then compared these differences to the null distribution of PDs generated by comparing the structure of the physical tree to all possible 10,395 trees of eight samples (figure 2.2a). All three maximum-parsimony trees were significantly more similar to the physical tree than would be expected by chance ($p < 0.001$ in all cases; figure 2.2a, dashed red lines). Furthermore, one of the most parsimonious trees is identical to the structure of the physical tree, and a maximum-likelihood tree calculated from the same data shows just one topological difference compared to the structure of the physical tree, in which sample 8 is incorrectly placed as sister to sample 5, but with low bootstrap support of 44% (figure 2.2a, blue line; figure 2.2b). As would be expected if plants accumulate somatic mutations as they grow, there is a significant correlation between the branch lengths of the physical tree measured in metres and the branch lengths of the maximum-parsimony tree of the same topology measured in number of somatic mutations (linear model forced through the origin: $R2 = 0.82$, $p < 0.001$; see also electronic supplementary material, §4). Notably, while various factors such as the difficulty of correctly inferring plant ontogeny may limit the utility of a phylogenetic positive control such as we present here (i.e. may produce false-negative results in which the structure of the tree appears, erroneously, to differ from the phylogeny of the sequenced genomes), it is unlikely that these factors would erroneously cause a close match between the physical structure of the tree and a phylogeny generated from the genomes of eight branches of that tree (i.e. a false positive). We therefore conclude that these analyses demonstrate that the phylogeny recovered from the genomic data matches the physical structure of the tree and confirm that there is a strong biological signal in our data.

2.4.3 Estimation of the Somatic Mutation Rate

We next developed a full maximum-likelihood framework that extends the existing models in DeNovoGear (Ramu et al. 2013) to detect somatic mutations in a phylogenetic context and used this framework to estimate the full rate and spectrum of somatic mutations in the individual *E. melliodora* (see Material and methods). This method improves on the approach we used in our positive control, above, because it increases our power to detect true somatic mutations and avoid false positives by assuming that the phylogenetic structure of the samples follows the physical structure of the tree, an assumption that is validated by the analyses above. It also makes better use of the replicate sampling design than the method we use for our positive control, above, by directly modelling the expected variation in sequencing data across our three biological replicates under the expectation that all three replicates were sequenced from a single underlying genotype (see methods and electronic supplementary material). Using this framework, we identified 90 high-confidence somatic variants.

Of the 90 high-confidence variants we identified, 20 were in genes. Of these, six were in coding regions, with five non-synonymous mutations and one synonymous mutation. The small sample size of synonymous and non-synonymous mutations means that we cannot provide a meaningful estimate of the ratio of non-synonymous to synonymous somatic mutations, although such an estimate would help to understand the extent to which somatic mutations may be under selection. We detected seven mutations on the branch that separates the herbivore-resistant samples from the other samples (branch B $\rightarrow$ C, figure 2.1). Although we lack the functional evidence to determine whether any of these mutations are directly involved in the resistance phenotype, two of the mutations occur near genes that are plausible candi-

dates for further investigation. One mutation occurs near Eucgr.C00081, which is a zinc-binding CCHC-type protein belonging to a small protein family known to bind RNA or ssDNA in *Arabidopsis thaliana* and thus potentially involved in gene expression regulation. Another mutation occurs near Eucgr.I01302, an acid phosphatase that may have as a substrate phosphoenol pyruvate, and therefore may be involved in pathways associated with the production of various secondary metabolites, including those identified in a recent GWAS study in a closely related eucalypt (Kainer et al. 2019).

We used the replicate sampling design of our analysis to estimate the false-negative rate and the false-discovery rate of our approach. It is necessary to estimate both the number of false-negative mutations and the number of false-positive mutations in order to estimate a somatic mutation rate. The former allows one to correct for the number of somatic mutations which a pipeline has failed to detect, while the latter allows one to correct for the number of somatic mutations which a pipeline has erroneously inferred. We estimated the false-negative rate by creating 14,000 *in silico* somatic mutations in the raw reads (Keightley et al. 2014), comprising 1000 *in silico* mutations for each of the 14 branches of the physical tree, and measuring the recovery rate of these *in silico* mutations using our maximum-likelihood approach. We were able to recover 4193 of the *in silico* mutations, suggesting that our recovery rate is 29.95%, and thus our false-negative rate is 70.05%. This false-negative rate was similar across all of the 14 branches in the tree (see electronic supplementary material, §2). Our ability to recover mutations differs substantially between repeat regions and non-repeat regions: we recover 40% of the simulated mutations in non-repeat regions, but just 13% of the simulated mutations in repeat regions (which make up roughly 40% of the genome). This difference is explained primarily by the stringent filters we use, that lead us to screen out many putative somatic mutations

in repeat regions. We then estimated the number of false-positive mutations in our data, and hence the false-discovery rate (the percentage of the observed mutations that are false positives) by repeating our detection pipeline after permuting the labels of samples and replicates to remove all phylogenetic information in the data, and only considering sites that we had not previously identified as variable (see methods). By removing phylogenetic information and previously identified variable sites, we can be sure that any mutations detected by this pipeline are false positives. Across 100 such permutations, we detected 11 false-positive mutations in total, suggesting that our pipeline generates 0.11 false-positive variant calls per experiment, and that the false-discovery rate for our analysis is 0.12%.

Based on these analyses, we can estimate the mutation rate per metre of physical growth and per year. We estimate that the true number of somatic mutations in our samples is 300 (calculated as: (90 high-confidence mutations minus 0.11 false-positive mutations)/the recovery rate of 0.2995)). Since we sampled a total of 90.1 m of physical branch length, this equates to 3.3 somatic mutations per diploid genome per metre of branch length, or 2.75×10^{-9} somatic mutations per base per metre of physical branch length. Although the exact age of this individual is unknown and difficult to estimate—it lives in a temperate climate and does not produce growth rings—its age is nevertheless almost certainly between 50 and 200 years old. Given that the physical branch length connecting each sampled branch tip to the ground varies between 8.4 m and 20.3 m, we estimate that the mutation rate per base per year for a single apical meristem lies in the range 1.16×10^{-10} to 1.12×10^{-9} (i.e. $8.4 \times 2.75 \times 10^{-9}/200$ to $20.3 \times 2.75 \times 10^{-9}/50$). It is important to note that it remains unclear whether mutations in growing plants accumulate linearly with the amount of physical growth. Indeed, evidence is accumulating that in at least some (and perhaps most) species, mutations may accumulate primarily at branching events rather than

during elongation of individual branches (Burian, Barbier de Reuille, and Kuhlemeier 2016; Watson et al. 2016). If this is the case, then the correlation we observe between the physical branch length and the number of inferred somatic mutations (see above, and electronic supplementary material, §4) may be due to a correlation between the physical length of a branch and the number of branching events that occurred along that branch during the plant's development. It is not possible to directly estimate the number of branching events along each branch in the individual tree we used in this study, because we expect that the tree will have regularly lost branches throughout its life, leaving no accurate record of the number of branching events.

2.4.4 What Drives Differences in Somatic Mutation Rates Among Species?

With some additional assumptions, it is also possible to estimate the mutation rate per generation and to compare this to estimates from other plants. The average height of an adult *E. melliodora* individual is between 15 m and 30 m (Boland and McDonald 2006), so if we assume that all somatic mutations are potentially heritable (about which there is limited evidence (Plomion et al. 2018) and ongoing discussion (Lanfear 2018)), we can estimate the per-generation mutation rate. To do this, we assume that a typical seed will be produced from a branch that has experienced 15–30 m of linear growth from the seed (Boland and McDonald 2006), and that mutations will have accumulated along that branch at 2.75×10^{-9} somatic mutations per base per metre of physical branch length, estimated above. We therefore estimate that the heritable somatic mutation rate per generation is between 4.13×10^{-8} and 8.25×10^{-8} mutations per base. For comparison the roughly 20 cm tall *Arabidopsis thaliana* has a per-generation mutation rate of 7.1×10^{-9} mutations per base (Ossowski et al. 2010). To the extent that such a comparison is accurate, which will be somewhat limited because the former estimate considers only somatic mutations and the latter considers all

heritable mutations including those caused during meiosis, we can then compare these estimates. Comparing the estimates suggests that despite being roughly 100 times taller than *Arabidopsis thaliana*, the per-generation mutation rate of *E. melliodora* is just approximately 10 times higher, which is achieved by a roughly fifteen-fold reduction in the mutation rate per physical metre of plant growth.

Our work adds to a growing body of evidence that low somatic mutation rates per unit of growth are a general feature of many large plant species (Bobiwash, Schultz, and Schoen 2013; Hanlon, Otto, and Aitken 2019; Plomion et al. 2018; Schmid-Siegert et al. 2017; Wang et al. 2019). For example, a recent study of the Sitka spruce estimated a per-generation somatic mutation rate of 2.7×10^{-8}, with confidence intervals that overlap ours (Hanlon, Otto, and Aitken 2019). While this per-generation rate is very similar to the one we estimate here, the rate of somatic mutation per metre of growth is around an order of magnitude lower in the Sitka spruce than our estimate for *E. melliodora* (2.75×10^{-9} somatic mutations per base pair per metre of growth for *E. melliodora* estimated here, versus 3.5×10^{-10} somatic mutations per base pair per metre of growth for Sitka spruce, estimated by dividing the per-generation mutation rate of 2.7×10^{-8} mutations per base by the average height of studied individuals of 76 m (Hanlon, Otto, and Aitken 2019), an appropriate calculation because the somatic mutation rate was estimated from paired samples taken from the base and the top of a collection of individual trees). Lower somatic mutation rates per unit of growth in larger plants may be the result of selection for reduced somatic mutation rates in response to the accumulation of increased genetic load in larger individuals (Hanlon, Otto, and Aitken 2019; Klekowski 1988; Lanfear et al. 2013; Plomion et al. 2018; Schmid-Siegert et al. 2017; Scofield 2014; Zhong et al. 2014). This pattern could also explain why larger plants tend to have lower average rates of molecular evolution than their smaller relatives (Lanfear et al. 2013; Smith and Donoghue 2008).

Several possible mechanisms might account for a reduction in accumulation of mutations per unit of growth in larger plants. Selection may favour reduction in the mutation rate per cell division through enhanced DNA repair to reduce the lifetime mutation risk. Alternatively, it may be that the reduction in the mutation rate is due to slower cell division. For example, plant meristems contain a slowly dividing population of cells in the central zone of the apical meristem, and these cells are known to divide more slowly in trees than in smaller plants (Romberger, Hejnowicz, and Hill 1993). Indeed, the rate of cell division in the central zone is so low that one estimate put the total number of cell divisions per generation in large trees as low as one hundred (Romberger, Hejnowicz, and Hill 1993). Regardless of the underlying mechanism, the surprisingly low rates of somatic mutation in large plants reported here and elsewhere suggest an emerging picture in which there is a strong link between the somatic mutation rates and life history across the plant kingdom. Longevity and size are two aspects of plant life history likely to be of central importance to the evolution of somatic mutation rates. Larger plants may suffer from a higher accumulation of somatic mutations because of the necessity for additional cell divisions. Plants that live longer may suffer from a higher accumulation of somatic mutations because of the accumulation of DNA damage over time and/or increased cell turnover in long-lived tissues. The relative importance of these two factors may differ among clades, species, and individual tissues and is likely to also depend on the balance between DNA damage and repair between cell divisions (Gao et al. 2016), the accuracy of DNA replication, cell size, and the rate of cell division. We hope that the approach we describe here will help in further understanding how these and other factors contribute to the accumulation or avoidance of somatic mutations in plants.

2.5 Data Accessibility

All of the bioinformatic workflows we describe here are provided in full detail in the GitHub repository available at https://github.com/adamjorr/somatic-variation. The raw short-read data are available online at https://www.ncbi.nlm.nih.gov/bioproject/553104.

2.6 Authors' Contributions

R.L. conceived the study; D.K., A.P., C.K., L.B., W.F., T.H. and R.L. planned the study; A.J.O., D.K., A.M.-S., R.A.C. and R.L. involved in bioinformatic analysis; A.J.O., D.K. and R.A.C. involved in bioinformatic methods development; A.P. helped in laboratory work; A.P., D.K., C.B.-S., T.H. and J.-F.H. involved in fieldwork; R.L.wrote the paper; all authors discussed and interpreted results; All authors edited many drafts of the paper.

2.7 Competing Interests

We declare we have no competing interests.

2.8 Funding

This work was supported by a Hermon-Slade grant to R.L. and an Australian Research Council Future Fellowship to R.L.

2.9 Footnotes

Electronic supplementary material is available online at https://doi.org/10.6084/m9.figshare.c.4869747.

REFERENCES

Ally, Dilara, Kermit Ritland, and Sarah P. Otto. 2010. "Aging in a Long-Lived Clonal Tree." *PLoS Biology* 8 (8): e1000454. https://doi.org/10.1371/journal.pbio.1000454.

Auwera, Geraldine A., Mauricio O. Carneiro, Christopher Hartl, Ryan Poplin, Guillermo del Angel, Ami Levy-Moonshine, Tadeusz Jordan, et al. 2013. "From FastQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline." *Current Protocols in Bioinformatics* 43 (1). https://doi.org/10.1002/0471250953.bi1110s43.

Bartholomé, Jérôme, Eric Mandrou, André Mabiala, Jerry Jenkins, Ibouniyamine Nabihoudine, Christophe Klopp, Jeremy Schmutz, Christophe Plomion, and Jean-Marc Gion. 2015. "High-resolution genetic maps of *Eucalyptus* improve *Eucalyptus grandis* genome assembly." *New Phytologist* 206 (4): 1283–1296. https://doi.org/10.1111/nph.13150.

Bobiwash, K, S T Schultz, and D J Schoen. 2013. "Somatic deleterious mutation rate in a woody plant: estimation from phenotypic data." *Heredity* 111 (4): 338–344. https://doi.org/10.1038/hdy.2013.57.

Boland, D. J, and M. W McDonald. 2006. *Forest trees of Australia.* OCLC: 181586979. Collingwood, Victoria: CSIRO Pub. http://public.eblib.com/choice/publicfullrecord.aspx?p=282659.

Burian, Agata, Pierre Barbier de Reuille, and Cris Kuhlemeier. 2016. "Patterns of Stem Cell Divisions Contribute to Plant Longevity." *Current Biology* 26 (11): 1385–1394. https://doi.org/10.1016/j.cub.2016.03.067.

Buss, L. W. 1983. "Evolution, development, and the units of selection." *Proceedings of the National Academy of Sciences* 80 (5): 1387–1391. https://doi.org/10.1073/pnas.80.5.1387.

Edwards, Penelope B., W. J. Wanjura, W. V. Brown, and John M. Dearn. 1990. "Mosaic resistance in plants." *Nature* 347 (6292): 434–434. https://doi.org/10.1038/347434a0.

Gao, Ziyue, Minyoung J. Wyman, Guy Sella, and Molly Przeworski. 2016. "Interpreting the Dependence of Mutation Rates on Age and Time." *PLOS Biology* 14 (1): e1002355. https://doi.org/10.1371/journal.pbio.1002355.

Gill, D E, L Chao, S L Perkins, and J B Wolf. 1995. "Genetic Mosaicism in Plants and Clonal Animals." *Annual Review of Ecology and Systematics* 26 (1): 423–444. https://doi.org/10.1146/annurev.es.26.110195.002231.

Grattapaglia, Dario, René E. Vaillancourt, Merv Shepherd, Bala R. Thumma, William Foley, Carsten Külheim, Brad M. Potts, and Alexander A. Myburg. 2012. "Progress in Myrtaceae genetics and genomics: Eucalyptus as the pivotal genus." *Tree Genetics & Genomes* 8 (3): 463–508. https://doi.org/10.1007/s11295-012-0491-x.

Hanlon, Vincent C. T., Sarah P. Otto, and Sally N. Aitken. 2019. "Somatic mutations substantially increase the per-generation mutation rate in the conifer *Picea sitchensis*." *Evolution Letters* 3 (4): 348–358. https://doi.org/10.1002/evl3.121.

Kainer, David, Amanda Padovan, Joerg Degenhardt, Sandra Krause, Prodyut Mondal, William J. Foley, and Carsten Külheim. 2019. "High marker density GWAS provides novel insights into the genomic architecture of terpene oil yield in *Eucalyptus*." *New Phytologist* 223 (3): 1489–1504. https://doi.org/10.1111/nph.15887.

Keightley, Peter D., Rob W. Ness, Daniel L. Halligan, and Penelope R. Haddrill. 2014. "Estimation of the Spontaneous Mutation Rate per Nucleotide Site in a *Drosophila melanogaster* Full-Sib Family." *Genetics* 196 (1): 313–320. https://doi.org/10.1534/genetics.113.158758.

Khoury, Colin, Brigitte Laliberté, and Luigi Guarino. 2010. "Trends in ex situ conservation of plant genetic resources: a review of global crop and regional conservation strategies." *Genetic Resources and Crop Evolution* 57 (4): 625–639. https://doi.org/10.1007/s10722-010-9534-z.

Klekowski, Edward J, Jorge E Corredor, Julio M Morell, and Carlos A Del Castillo. 1994. "Petroleum pollution and mutation in mangroves." *Marine Pollution Bulletin* 28 (3): 166–169. https://doi.org/10.1016/0025-326X(94)90393-X.

Klekowski, Edward J. 1988. *Mutation, Developmental Selection, and Plant Evolution.* Columbia University Press. https://doi.org/10.7312/klek92068.

Klekowski, Edward J., and Paul J. Godfrey. 1989. "Ageing and mutation in plants." *Nature* 340 (6232): 389–391. https://doi.org/10.1038/340389a0.

Lanfear, Robert. 2018. "Do plants have a segregated germline?" *PLOS Biology* 16 (5): e2005439. https://doi.org/10.1371/journal.pbio.2005439.

Lanfear, Robert, Simon Y. W. Ho, T. Jonathan Davies, Angela T. Moles, Lonnie Aarssen, Nathan G. Swenson, Laura Warman, Amy E. Zanne, and Andrew P. Allen. 2013. "Taller plants have lower rates of molecular evolution." *Nature Communications* 4 (1): 1879. https://doi.org/10.1038/ncomms2836.

Li, Heng. 2011a. "A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data." *Bioinformatics* 27 (21): 2987–2993. https://doi.org/10.1093/bioinformatics/btr509.

Martin, Marcel, Murray Patterson, Shilpa Garg, Sarah O Fischer, Nadia Pisanti, Gunnar W Klau, Alexander Schöenhuth, and Tobias Marschall. 2016. *WhatsHap: fast and accurate read-based phasing.* Preprint. Bioinformatics. https://doi.org/10.1101/085050.

McKenna, A., M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, et al. 2010. "The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data." *Genome Research* 20 (9): 1297–1303. https://doi.org/10.1101/gr.107524.110.

Michel, Albrecht, Renee S. Arias, Brian E. Scheffler, Stephen O. Duke, Michael Netherland, and Franck E. Dayan. 2004. "Somatic mutation-mediated evolution of herbicide resistance in the nonindigenous invasive plant hydrilla (Hydrilla verticillata): HYDRILLA SOMATIC MUTANTS and HERBICIDE RESISTANCE." *Molecular Ecology* 13 (10): 3229–3237. https://doi.org/10.1111/j.1365-294X.2004.02280.x.

Ness, Rob W., Andrew D. Morgan, Nick Colegrave, and Peter D. Keightley. 2012. "Estimate of the Spontaneous Mutation Rate in *Chlamydomonas reinhardtii*." *Genetics* 192 (4): 1447–1454. https://doi.org/10.1534/genetics.112.145078.

Ossowski, S., K. Schneeberger, J. I. Lucas-Lledo, N. Warthmann, R. M. Clark, R. G. Shaw, D. Weigel, and M. Lynch. 2010. "The Rate and Molecular Spectrum of Spontaneous Mutations in Arabidopsis thaliana." *Science* 327 (5961): 92–94. ht tps://doi.org/10.1126/science.1180677.

Padovan, Amanda, Andras Keszei, William J Foley, and Carsten Külheim. 2013. "Differences in gene expression within a striking phenotypic mosaic Eucalyptus tree that varies in susceptibility to herbivory." *BMC Plant Biology* 13 (1): 29. https://doi.org/10.1186/1471-2229-13-29.

Plomion, Christophe, Jean-Marc Aury, Joëlle Amselem, Thibault Leroy, Florent Murat, Sébastien Duplessis, Sébastien Faye, et al. 2018. "Oak genome reveals facets of long lifespan." *Nature Plants* 4 (7): 440–452. https://doi.org/10.1038/s41477-018-0172-3.

Ramu, Avinash, Michiel J Noordam, Rachel S Schwartz, Arthur Wuster, Matthew E Hurles, Reed A Cartwright, and Donald F Conrad. 2013. "DeNovoGear: de novo indel and point mutation discovery and phasing." *Nature Methods* 10 (10): 985–987. https://doi.org/10.1038/nmeth.2611.

Romberger, J. A., Z. Hejnowicz, and J. F. Hill. 1993. "Plant structure: function and development. A treatise on anatomy and vegetative development with special reference to woody plants." *Plant structure: function and development. A treatise on anatomy and vegetative development with special reference to woody plants.,* https://www.cabdirect.org/cabdirect/abstract/19940600619.

Schliep, Klaus Peter. 2011. "phangorn: phylogenetic analysis in R." *Bioinformatics* 27 (4): 592–593. https://doi.org/10.1093/bioinformatics/btq706.

Schmid-Siegert, Emanuel, Namrata Sarkar, Christian Iseli, Sandra Calderon, Caroline Gouhier-Darimont, Jacqueline Chrast, Pietro Cattaneo, et al. 2017. "Low number of fixed somatic mutations in a long-lived oak tree." *Nature Plants* 3 (12): 926–929. https://doi.org/10.1038/s41477-017-0066-9.

Schoen, D. J., J. L. David, and T. M. Bataillon. 1998. "Deleterious mutation accumulation and the regeneration of genetic resources." *Proceedings of the National Academy of Sciences* 95 (1): 394–399. https://doi.org/10.1073/pnas.95.1.394.

Schultz, Stewart T., and Douglas G. Scofield. 2009. "Mutation Accumulation in Real Branches: Fitness Assays for Genomic Deleterious Mutation Rate and Effect in Large-Statured Plants." *The American Naturalist* 174 (2): 163–175. https://doi.org/10.1086/600100.

Scofield, D G. 2014. "A definitive demonstration of fitness effects due to somatic mutation in a plant." *Heredity* 112 (4): 361–362. https://doi.org/10.1038/hdy.2013.114.

Sedlazeck, Fritz J., Philipp Rescheneder, and Arndt von Haeseler. 2013. "NextGenMap: fast and accurate read mapping in highly polymorphic genomes." *Bioinformatics* 29 (21): 2790–2791. https://doi.org/10.1093/bioinformatics/btt468.

Smith, S. A., and M. J. Donoghue. 2008. "Rates of Molecular Evolution Are Linked to Life History in Flowering Plants." *Science* 322 (5898): 86–89. https://doi.org/10.1126/science.1163197.

Steel, M. A., and D. Penny. 1993. "Distributions of Tree Comparison Metrics–Some New Results." *Systematic Biology* 42 (2): 126–141. https://doi.org/10.1093/sysbio/42.2.126.

Sutherland, William J., and Andrew R. Watkinson. 1986. "Somatic mutation: Do plants evolve differently?" *Nature* 320 (6060): 305–305. https://doi.org/10.1038/320305a0.

Walbot, Virginia. 1985. "On the life strategies of plants and animals." *Trends in Genetics* 1:165–169. https://doi.org/10.1016/0168-9525(85)90071-X.

Wang, Long, Yilun Ji, Yingwen Hu, Huaying Hu, Xianqin Jia, Mengmeng Jiang, Xiaohui Zhang, et al. 2019. "The architecture of intra-organism mutation rate variation in plants." *PLOS Biology* 17 (4): e3000191. https://doi.org/10.1371/journal.pbio.3000191.

Watson, J. Matthew, Alexander Platzer, Anita Kazda, Svetlana Akimcheva, Sona Valuchova, Viktoria Nizhynska, Magnus Nordborg, and Karel Riha. 2016. "Germline replications and somatic mutation accumulation are independent of vegetative life span in *Arabidopsis*." *Proceedings of the National Academy of Sciences* 113 (43): 12226–12231. https://doi.org/10.1073/pnas.1609686113.

Whitham, Thomas G., and C. N. Slobodchikoff. 1981. "Evolution by individuals, plant-herbivore interactions, and mosaics of genetic variability: The adaptive significance of somatic mutations in plants." *Oecologia* 49 (3): 287–292. https://doi.org/10.1007/BF00347587.

Zhong, B., R. Fong, L. J. Collins, P. A. McLenachan, and D. Penny. 2014. "Two New Fern Chloroplasts and Decelerated Evolution Linked to the Long Generation Time in Tree Ferns." *Genome Biology and Evolution* 6 (5): 1166–1173. https://doi.org/10.1093/gbe/evu087.

Chapter 3

KBBQ: A REFERENCE-FREE METHOD FOR BASE QUALITY SCORE RECALIBRATION

3.1 Introduction

Next generation sequencing has a plethora of applications in biology. While the technology is widely applied, the sequencing process is inherently erroneous and errors are common in sequencing data, with base substitution error rate estimates ranging between .1 and 1%, depending on the sequencing technology used (Fox et al. 2014). To help mitigate this, DNA molecules are usually copied and sequenced multiple times to ensure the sequence is correct, since multiple independent measurements are unlikely to all contain the same error at the same position. While this works well in general, it can be costly and additional sample manipulation can damage samples and insert mutations, further increasing the amount of technical error in the data (Ma et al. 2019; Schirmer et al. 2015).

3.1.1 Quality Scores

In addition to increased sequencing depth, quality scores are an important part of sequencing data that helps identify erroneous sequences. Sequencing data is usually presented along with quality scores in the Phred scale (Ewing et al. 1998; Ewing and Green 1998). These quality scores measure the confidence the instrument has in its determination that any particular base in the sequence is correct. Specifically, the quality score is equal to $-10 \log_{10} P(e)$ (Ewing et al. 1998; Ewing and Green 1998), where $P(e)$ is the probability the base is an error. For example, a base with a quality

score of 40 has a .0001 probability of being incorrect while a base with a quality score of 10 has probability .1 of being incorrect. Generally, bases with scores lower than 10 are considered bad quality and bases with scores 30 and above are considered to be good quality. However, the score allows finer resolution than "good" and "bad", and is therefore more nuanced.

The usefulness of quantitative scores over categorical can be illustrated by considering how variant calling algorithms utilize quality scores. Variant calling is a task to identify genetic variation in a sample from sequencing data. Variant calling models use quality scores to differentially weight the observed data. Generally, these models attempt to find the sample genotype most consistent with the observed data and recognize that low quality bases provide less reliable evidence for one genotype over another. The BCFTools multiallelic caller (H. Li et al. 2009), GATK's HaplotypeCaller (Poplin et al. 2018), and FreeBayes (Garrison and Marth 2012)—three of the most popular variant callers—all take quality score into consideration when calling variants. Since the quality score is an exactly defined probability, it is straightforward to integrate these scores into a model.

If so desired, quantitative scores can be collapsed into coarser categories, such as "good" and "bad". This is sometimes done as a heuristic; *ie.* scores less than 10 are untrustworthy and filtered out of a dataset and other scores are trusted and retained. The practice of binning quality scores together with neighboring scores is commonly performed to reduce file size (Malysa et al. 2015; *Reducing Whole-Genome Data Storage Footprint* 2014; Shibuya and Comin 2019; Yu et al. 2015). However, it's important to note that this process cannot be simply reversed as information is lost when the scores are binned.

3.1.2 Base Quality Score Recalibration

While base quality scores are important for identifying reliable data, base quality scores are often incorrect. Base quality scores are exactly defined as a probability. They can be interpreted as a prediction giving the probability the reported base is an error. In general, probabilities are called *calibrated* if the reported probability accurately predicts the frequency of an event. Base quality scores in Illumina sequencing reads are not well-calibrated (Callahan et al. 2016; Ni and Stoneking 2016). A few alternative base calling models have been developed for Illumina machines that improve base call accuracy and quality score calibration; however, these are difficult to use because they require the raw output from the sequencing machine, which is unavailable to most users of sequencing as they are usually disposed of after a sequencing run due to the large cost associated with storing that data (Kao and Song 2011; Massingham and Goldman 2012).

Since base quality scores are used to some degree by most variant calling methods, it is probable that poorly calibrated reads reduce the quality of resulting variant calls. Similarly, the reduced amount of information in binned quality scores may also impact the quality of variant calls made using the data. Poor calibration combined with poor resolution resulting from quality score binning may have an important impact on algorithms that rely on these scores to function, but the exact effect of these phenomena are unknown.

Though increased sequencing depth can help mitigate the impact of random sequencing errors, sequencing is affected by non-random biases. For these types of errors, increasing sequencing depth counterintuitively *increases* the effect of these errors, as they by definition occur preferentially at the same location. Thus, increased sequencing depth at that location adds more errors than are expected by chance,

making those erroneous reads seem trustworthy. These biases can be due to the nature of the DNA sequence itself, with errors induced during library preparation or the sequencing reaction likely due to secondary structure (Ma et al. 2019; Meacham et al. 2011; Nakamura et al. 2011; Schirmer et al. 2015). At the same time, the sequencing reaction is also non-randomly biased. Bases at the end of a read are much more likely to be erroneous than bases at the beginning of the read, and the identity of the base and adjacent bases also affect the error rate (Fox et al. 2014; Schirmer et al. 2016). Thus while random errors are troublesome, their impacts can be somewhat mitigated by more sequencing. Systematic errors cannot be addressed in the same way, but they can be modeled and incorporated into quality scores.

Base quality score recalibration (BQSR) is the process of modeling errors in sequencing data and using the created model to update quality scores such that they reflect an accurate probability of error of any base. GATK BQSR is the most popular method for BQSR and is recommended before variant calling by the GATK best practices (Auwera et al. 2013). The model integrates many covariates of error such that the output quality score reflects an accurate, independent measure of the probability of error. The algorithm takes reads aligned to a reference and a database of potentially variable sites in the genome as input.

The algorithm proceeds in 3 phases. In the first phase, the algorithm compares each read to the aligned reference. Potentially variable sites are ignored and mismatches from the reference sequence and non-mismatches are counted. The numbers of matching and mismatching bases are categorized according to the model covariates, which are: read group, assigned quality score, sequencing cycle, and the base identity along with the identity of the previous base (the base context). In the second phase, a bayesian hierarchical model is trained with the count data. Using a normal distribution of the mean probability of error as a prior, the *maximum a posteriori*

(MAP) quality score of the read group is calculated assuming the errors are binomially distributed. The estimated error probability for each read group $\hat{Q}_{rg}$ can be written in terms of the binomial probability mass function $\mathcal{B}$ with error probability Q_{rg} and number of observations Observations_{rg} and the normal probability density function $\mathcal{N}$ with standard deviation of 1 around the mean estimate of the quality score for the entire dataset, $\bar{Q}$

$$\hat{Q}_{rg} = \text{argmax}_{Q_{rg}} P(Q_{rg} | \text{Errors}_{rg}) \tag{3.1}$$

$$= \text{argmax}_{Q_{rg}} P(\mathcal{B}(\text{Errors}_{rg} | \text{Observations}_{rg}, Q_{rg}) \times P(\mathcal{N}(Q_{rg} | \bar{Q})) \tag{3.2}$$

This score is then used as the prior for calculating the *maximum a posteriori* quality score for each assigned quality score in that read group, $\hat{Q}_{\text{assigned quality score}}$, using a similar formula. In turn, this score is used as the prior for calculating the *maximum a posteriori* scores for the sequencing cycle and context covariates of bases with that score. The difference between the MAP estimate and the prior is used to calculate the final score, which is the sum of the MAP estimate of the assigned quality score and these two differences. That is,

$$\Delta Q_{\text{cycle}} = \hat{Q}_{\text{cycle}} - \hat{Q}_{\text{assigned quality score}} \tag{3.3}$$

$$\Delta Q_{\text{context}} = \hat{Q}_{\text{context}} - \hat{Q}_{\text{assigned quality score}} \tag{3.4}$$

$$Q_{\text{recalibrated}} = \hat{Q}_{\text{assigned quality score}} + \Delta Q_{\text{cycle}} + \Delta Q_{\text{context}} \tag{3.5}$$

In the third phase, the quality score of each read is adjusted based on the four covariates for each base in the read according to values calculated in the previous phase. See Figure 3.1 for a visual example.

BQSR improves quality score calibration in many cases, especially when there is moderate sequencing depth and the database of variable sites is nearly complete. While BQSR is recommended in GATK's Best Practices, its effect on the resulting

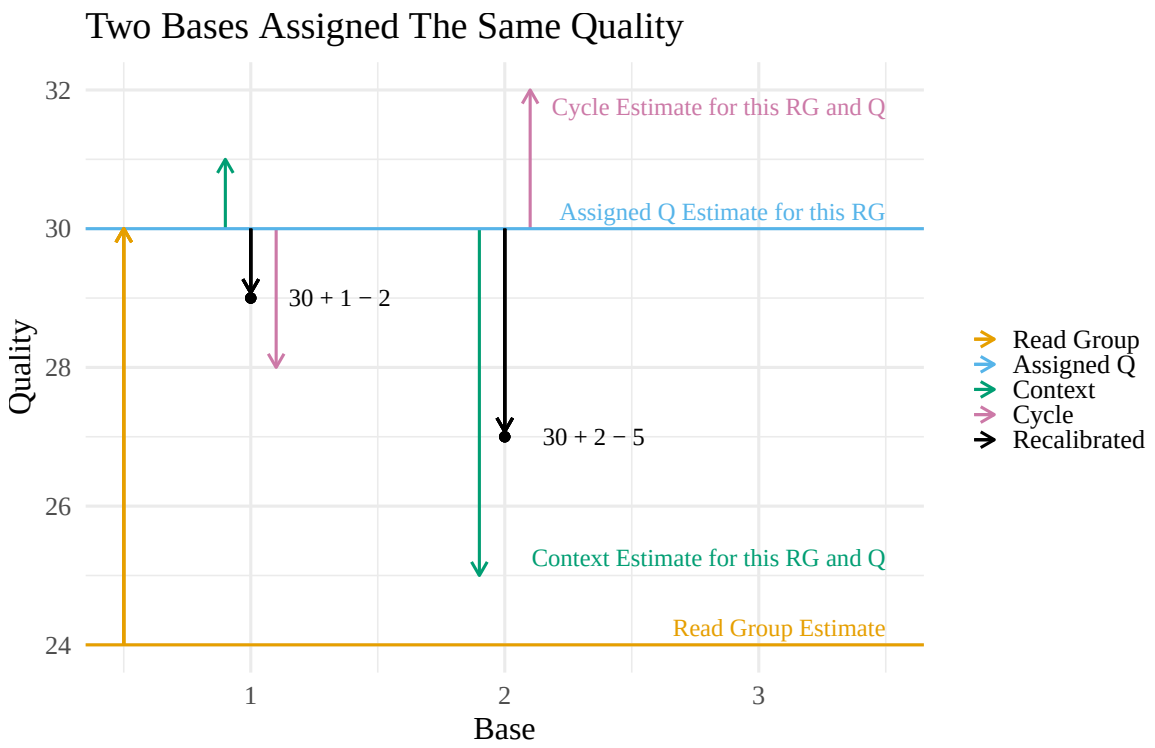


Figure 3.1: Base Quality Score Recalibration Example | An example of two bases that were assigned identical quality scores. Here, the read group has an estimated quality of 24 and the assigned quality score in that read group has an estimated quality of 30. The estimated quality of bases with the same context as the first base is 31 while the estimated quality of bases with the same cycle is 28. This results in an overall estimated quality score of 29. Despite having the same read group and assigned score, because of the difference in context and cycle, the second base is assigned a different score.

variant calls is not well-characterized, and there is ongoing debate about whether to continue recommending BQSR. (Van der Auwera 2020a, 2020b) However, Ni and Stoneking (2016) find that improved quality score calibration aids detection of minor alleles in high coverage datasets by increasing sensitivity and reducing the number of false positive calls. Thus, the need to perform BQSR and the performance of GATK's BQSR algorithm may vary from study to study. Two objectives of this work are to help elucidate when GATK BQSR performs well and when it fails and to provide a method that works in situations that GATK BQSR struggles with. As an example, the GATK developers recommend BQSR for use in cancer variant discovery

(Cibulskis et al. 2013), but the tumor genome is likely much different from the human genome, and the database of variable sites will likely miss many truly variable sites due to the large mutation rate present in cancer genomes. Additionally, mismatches in reads misaligned due to chromosomal rearrangements may be mistakenly counted as evidence of sequencing error. The number of these errors required to significantly impact the performance of the algorithm is not clear.

3.1.3 Alternate Approaches for BQSR

As illustrated above, the most problematic aspect of GATK BQSR occurs in the first phase of the algorithm: counting erroneous and non-erroneous bases. This is especially true when considering non-model organisms, where alignment errors may be common. Furthermore, in non-model organisms a database of variable sites is likely unavailable or largely incomplete. However, there are methods that attempt to overcome this deficiency. While there exist alternative approaches that implement different error models such as Lacer (Chung and Chen 2017), the biggest problem for analyzing data without reliable reference information is the method of counting erroneous and non-erroneous bases.

Many alternative algorithms have been developed to avoid providing a database of variable sites. However, these approaches all still require a reference and alignment, and many require extra reagents and sequencing spike-ins that increase the cost of analysis and cannot be used to reanalyze existing sequencing data that hasn't been specially prepared. ReQON (Cabanski et al. 2012), like GATK, considers bases that do not match the reference as errors but limits the number of acceptable errors at a position to minimize the effect of unknown variants. It then uses a logistic regression to recalibrate the quality scores. SOAP2 (R. Li et al. 2009) contains a model for consensus sequence construction that performs BQSR during construction. The

methods of Zook et al. (2012) and Ni and Stoneking (2016) use synthetic spike-ins of known composition and GATK's model (Zook et al. 2012) or piecewise regression (Ni and Stoneking 2016) to recalibrate quality scores. Since the sequence of the spike-in is known before hand, errors are easy to identify as there should be no biological variation in the spiked-in sample. Crucially, these methods require a reference and possibly other information to recalibrate reads that may not be available.

To effectively recalibrate base quality scores with as little auxiliary information as possible, I developed KBBQ, a software package to recalibrate quality scores of whole genome sequencing data without a reference or database of variable sites. The only required input is the set of reads to be recalibrated. Rather than excluding variation and comparing to the reference like GATK does, KBBQ uses k-mer subsampling to find likely errors. Once the number of errors and nonerrors are counted and categorized according to their covariates, it uses the same model GATK uses to recalibrate the reads. Here, I show that this difference in how KBBQ and GATK detect sequencing errors impacts the resultant base quality scores and that KBBQ produces superior calibrations in a variety of scenarios.

3.2 Methods

KBBQ performs BQSR by adjusting how errors in the dataset are discovered. Instead of looking at the reference, KBBQ implements the error-correction algorithm described in Song, Florea, and Langmead (2014) and uses the errors detected by that procedure to train and apply the standard GATK model. Note that the sequenced bases are not actually changed; the detected errors are used only to train the model. If there is evidence according to the model that the base is erroneous, its base quality will be decreased according to the strength of that evidence.

Briefly, the algorithm subsamples k-mers from the dataset. Since erroneous k-mers

are expected to be unique, erroneous k-mers are less likely to be sampled than error-free k-mers. A binomial test is then conducted for every nucleotide in the dataset; if a sufficient number of k-mers contain the nucleotide, the base is likely not erroneous and called trusted. If k of these trusted bases appear next to each other, that k-mer is stored as a trusted k-mer. Once the trusted k-mers have all been stored, each read is iterated through once again and any bases on the edge of an island of trusted k-mers are changed such that the change produces the maximal number of trusted k-mers in the read. These changes are marked as errors and used to train the model. The model training and recalibration procedure are the same as those described above for GATK's BQSR method.

3.2.1 KBBQ Program Input and Parameters

KBBQ requires as input a set of reads in FASTQ (Cock et al. 2010) or BAM (H. Li et al. 2009) format. If the input data is FASTQ formatted and consists of multiple read groups, the read group each read belongs to should be annotated in the name of each read. Alternatively, if the user has an original data file and a data file that has already been corrected using an error correction program, the user may supply the corrected file with the `--fixed` option to obtain errors from that correction rather than performing the error correction algorithm included in KBBQ. The program's only required parameter is the approximate length of the sequenced region in base-pairs, and this information can be taken from the BAM header if it is present. It may be set with the `--genomelen` option. If BAM input is provided, the `--set-oq` and `--use-oq` flags can be used to set the OQ flag on the read before recalibrating or to use the quality scores encoded in the OQ flag rather than the quality scores in the primary quality score field.

Optionally, the approximate sequencing coverage may also be provided with the

`--coverage` option. If it is not, it will be estimated by finding the length of the sequenced data divided by the provided genome length; Coverage $= \frac{\text{Sequence Length}}{\text{Genome Length}}$. An α parameter may also be provided with the `--alpha` option; this is the same α parameter that Lighter uses, and is the fraction of reads sampled from the input data (Song, Florea, and Langmead 2014). If not provided, the recommended value of $\alpha = \frac{7}{\text{Coverage}}$ is used. A `-k` parameter may also be provided, which changes the k-mer size used for the error detection algorithm. The maximum value of 32 is recommended and is the default value. A summary of flags and options the program supports is listed in table 3.1.

The genome length parameter, in addition to being used to estimate sequencing coverage, is also used to estimate the number of k-mers that will be sampled. This is used to parameterize the bloom filter that stores the sampled and trusted k-mers. This parameterization is different than that used in Lighter; there, the number of sampled k-mers is estimated to be $1.5 \times$ Genome length. However, the expected number of sampled k-mers K_{sampled} is bounded by the expected value of the binomial distribution parameterized by Genome length $\times$ Coverage and α. Assuming *every* k-mer is unique, the number of possible k-mers in the dataset K_{total} is less than Coverage $\times$ Genome length. Since the number of k-mers in each read is Read length $-k+1$ and assuming equal read lengths, the total number of k-mers is:

$$
\begin{aligned}
K_{\text{total}} &= \sum_{\text{Reads}} \text{Read length} - k + 1 && (3.6)\\
&= (\text{Read length} - k + 1) \times \text{Number of reads} && (3.7)\\
&< \text{Read length} \times \text{Number of reads} && (3.8)\\
&< \text{Read length} \times \frac{\text{Coverage} \times \text{Genome length}}{\text{Read length}} && (3.9)\\
&< \text{Coverage} \times \text{Genome length} && (3.10)
\end{aligned}
$$

So the expected number of sampled k-mers is

$$E[K_{\text{sampled}}] = E[\mathcal{B}(x; \alpha, K_{\text{total}})] \tag{3.11}$$

$$= \alpha \times K_{\text{total}} \tag{3.12}$$

$$< \alpha \times \text{Coverage} \times \text{Genome length} \tag{3.13}$$

Notably, if α is the recommended value of $\frac{7}{\text{Coverage}}$, the expected number of sampled k-mers is less than $7 \times$ Genome length, a bound over 4 times larger than the estimate used by Lighter. For the data analyzed here, this provides a much better estimate of the true number of sampled k-mers. Ultimately, the larger estimate of elements inserted into the bloom filter causes an increase in size of the bloom filter but a smaller false positive rate.

3.2.2 Testing and Validation

To test the performance of KBBQ, I reanalyzed the synthetic diploid CHM1-CHM13 dataset from H. Li et al. (2018) (SRA accession ERR1341796, aligned to hg19). Like other benchmarking datasets, this dataset includes a BED file describing confident regions in which the genotype of any site that differs from homozygous reference within those regions are included in a VCF file. For the purposes of this work, I assume these confident regions and associated VCF entries are correct and represent the true genotype of the sequenced sample.

This dataset was specifically designed to study the impact of deep sequencing on variant calling and has a coverage of approximately 45x. Thus, the effect of sequence specific errors and other non-random biases in sequencing should be pronounced in this data. The dataset was constructed by adding equal concentrations of DNA of CHM1 and CHM13 human complete hydatidiform mole cell lines and sequencing the

Parameter	Short Option	Default Value	Summary
`--ksize`	`-k`	32	Size of k-mer to use for correction
`--use-oq`	`-u`	Off	Use BAM OQ tag values as quality scores
`--set-oq`	`-s`	Off	Set BAM OQ tag values before recalibration
`--genomelen`	`-g`	Estimated for BAM, required for FASTQ	The approximate size of the sequenced region in base-pairs.
`--coverage`	`-c`	Estimated from data	Approximate sequencing coverage
`--fixed`	`-f`	Off	Treat changes to reads in the given file as errors and recalibrate.
`--alpha`	`-a`	7 / coverage	Rate to sample k-mers
`--threads`	`-t`	1	Number of CPU threads to use

Table 3.1: KBBQ Parameters | Provides the short options for each long parameter name, the default value of the parameter and a summary of how each parameter changes the behavior of the program.

mixture. These moles are formed when a sperm combines with an egg containing no nucleus; the sperm then undergoes mitosis to generate a completely homozygous cell mass. They are effectively haploid, and it is significantly easier to genotype a haploid cell than a diploid one. Thus, the mixture simulates a diploid human cell but the genotype of each haplotype is known. This means there should be very few, if any, errors in the declared genotypes included with the data, and this was confirmed by H. Li et al. (2018).

To measure the performance of KBBQ and compare to GATK's BaseRecalibrator and ApplyBQSR tools, I subset the full dataset to only reads aligned on Chromosome 1 and overlapping the BED of confident regions using the samtools view command (H. Li et al. 2009) and the -L parameter. I then used the samtools (version 1.10) view command with the -F 3844 flag to remove unmapped, secondary, and supplementary alignments as well as reads marked as failing quality checks and as PCR duplicates. I then used the fixmate and view commands with flag -f 1 to remove any singleton reads. I then ran KBBQ (commit ID 9167b72599892a33493d8ebdc01fac33990c1738) on the dataset with the options `--use-oq -g 214206308 -a .15`. I also ran GATK (version v4.1.8.0-5-g1836ab0-SNAPSHOT) BaseRecalibrator with the provided variant data as the known sites file and the `--use-original-qualities` flag set. I then used the ApplyBQSR tool with the `--use-original-qualities` flag to recalibrate the input file. The `--use-original-qualities` flag is appropriate here because the data has already been recalibrated and the original assigned quality scores are stored in the OQ tag on each of the reads. This flag tells GATK to use and recalibrate those quality scores, rather than using the scores that are already present in the QUAL field for the read.

I then compared the two recalibrated files by running the BaseRecalibrator tool again on both output datasets. In this invocation of BaseRecalibrator, `--use-`

`original-qualities` was not set as the quality scores in the QUAL field of the recalibrated reads are the quality scores calculated using the models trained above. I then used GATK's AnalyzeCovariates tool to create CSV files containing the number of errors for each assigned quality score and plotted these values.

To simulate a situation where a researcher has sequenced a non-model organism, is using a reference that may not closely match the sample, and doesn't have a database of variable sites, I aligned the test data to the chimp reference genome (Waterson et al. 2005) using NextGenMap (Sedlazeck, Rescheneder, and Haeseler 2013) version 0.5.2 with the -p flag as the reads are paired. Prior to realignment, the original qualities in the OQ tag were placed into the QUAL field using GATK's RevertBaseQualityScores tool to ensure the raw assigned quality scores were used to align the reads.

In this situation, GATK recommends calling an initial set of variants with high confidence and using these variants as the database of variable sites. To see how this affects the results of GATK's BQSR, I called variants using HaplotypeCaller with the `-stand-call-conf 50` argument. The default value for this argument is 40, and is a phred-scaled confidence threshold for reporting a variant, so with this flag the model's confidence in the emitted calls should be very high. I then used the resulting variants as the database of variable sites with BaseRecalibrator to train a recalibration model. Here, the `--use-original-qualities` flag was not used as the qualities in the QUAL field for each read were already the raw uncalibrated scores.

To evaluate this model using the truth set, I used the model and ApplyBQSR to recalibrate the reads as they were aligned to the human reference. Here, the `--use-original-qualities` flag was used, as the quality scores that represent the original assigned scores for the reads (the same scores used to train the model) are given in the OQ tag rather than the QUAL field. Since the recalibration was performed after

the reads were already aligned to the human reference, the sole difference between this recalibration method and the standard using the correct reference and database of variable sites is the trained model; the alignment has no impact on benchmarking the calibration. I then used AnalyzeCovariates as above to obtain the calibration data and plot it.

As an additional test of KBBQ's performance, particularly in situations where the user has no quality reference, I simulated reads from the unscaffolded contigs of the initial release (v1.0) of the *Eucalyptus grandis* genome, SRA accession AUSX00000000.1. GATK recommends using at least 100 million base pairs for calibration, so I aimed to simulate twice that many bases. To simulate a situation where a user has a medium-coverage dataset of approximately 20X coverage, I therefore simulated a haploid genome of 5 million base pairs by randomly sampling contigs from the full set of *E. grandis* contigs using shuf, awk, and the samtools faidx program.

I then simulated heterozygous sites in the genome using simuG version 1.0.0 (Yue and Liti 2019) with a transition/transversion ratio of 2 to mirror the approximate ratio of mutations in real eucalypts. I simulated 125,000 SNPs in the reference to reflect the large level of heterozygosity in eucalypts (Külheim et al. 2009). I then concatenated the subsampled reference and reference containing the SNPs, which I used to simulate 101-bp long paired end reads using ART (version MountRainier-2016-06-05; Huang et al. 2012) using the HiSeq 2500 error profile, a mean fragment length of 300, a fragment length standard deviation of 20, and a depth of coverage of 20. Since this concatenated genome is twice the size as the haploid genome, at 20 fold coverage it would yield approximately 200 million base-pairs of sequence. I then aligned the simulated reads to the genome with NextGenMap (Sedlazeck, Rescheneder, and Haeseler 2013).

This reflects a scenario where a researcher sequences a highly heterozygous in-

dividual, assembles contigs, and aligns the sequenced reads back to the assembly. However, in this case the locations of all heterozygous sites are known. To compare the performance of KBBQ and GATK recalibration, I built the recalibration model with GATK BaseRecalibrator using the true location of all the heterozygous sites and applied the model with with GATK ApplyBQSR, both without the `--use-original-qualities` flag since the assigned quality scores were present in the QUAL field of the read. Since most researchers don't have knowledge of all the true variable sites in the data, I also repeated this procedure but using the initial calls made by HaplotypeCaller with the `--stand-call-conf` parameter set to 50, which makes HaplotypeCaller emit only variants it is highly confident in. I then ran KBBQ using a k-mer size of 32 and a genome length of 5 million to recalibrate the reads. Finally, I used BaseRecalibrator again and GATK's AnalyzeCovariates tool with the CSV output option to check the calibration of each of the four recalibrated alignments.

Scripts to replicate the above analyses are available in Appendix E.

3.3 Results

Using GATK's recommended method of using high-confidence variants when a database of variable sites is unavailable produced poorly calibrated data with a RMSE of 3.89. The shape of this recalibrated data is also interesting; it shows that the quality scores are underconfident for mid-range quality scores and overconfident for high quality scores. This calibration and the calibration using the true set of variants, using KBBQ, and the raw read quality are plotted in Figure 3.2 and the RMSEs of each data set is listed in Table 3.2

The resulting calibration of each recalibration method applied to the simulated data is shown in Figure 3.3. The raw data as simulated is fairly well-calibrated and almost all quality scores are on the 1-to-1 line. GATK recalibration with an initial

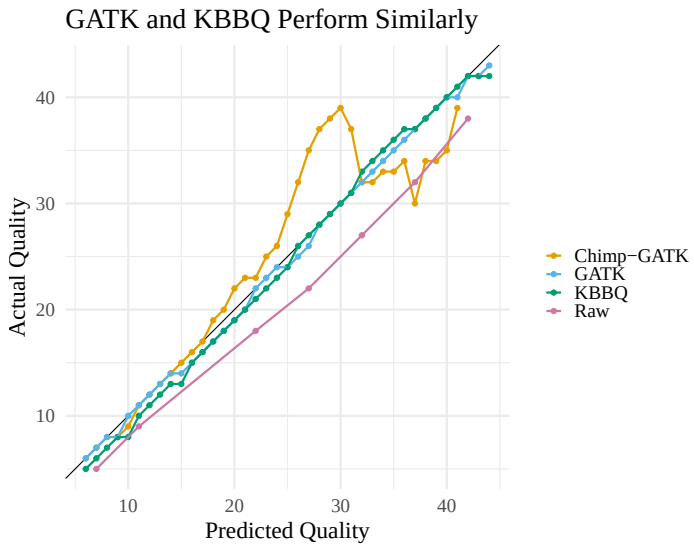


Figure 3.2: Comparison of Calibration Methods | Chimp-GATK is the result of calibrating the reads using the model trained on the reads aligned to the chimp genome along with the variants called using that alignment. GATK is the result of using GATK's BaseRecalibrator with the truth set of variants. KBBQ is the result of using the KBBQ tool, which requires only reads and no reference or variant set. Raw is the calibration of the uncalibrated data.

Calibration Method	RMSE
Chimp-GATK	3.89
GATK	0.60
KBBQ	0.96
Raw	4.05

Table 3.2: Errors of Different Calibration Methods | Root mean squared error of quality score for reads recalibrated using different methods. Chimp-GATK is the result of calibrating the reads using the model trained on the reads aligned to the chimp genome along with the variants called using that alignment. GATK is the result of using GATK's BaseRecalibrator with the truth set of variants. KBBQ is the result of using the KBBQ tool, which requires only reads and no reference or variant set. Raw is the calibration of the uncalibrated data.

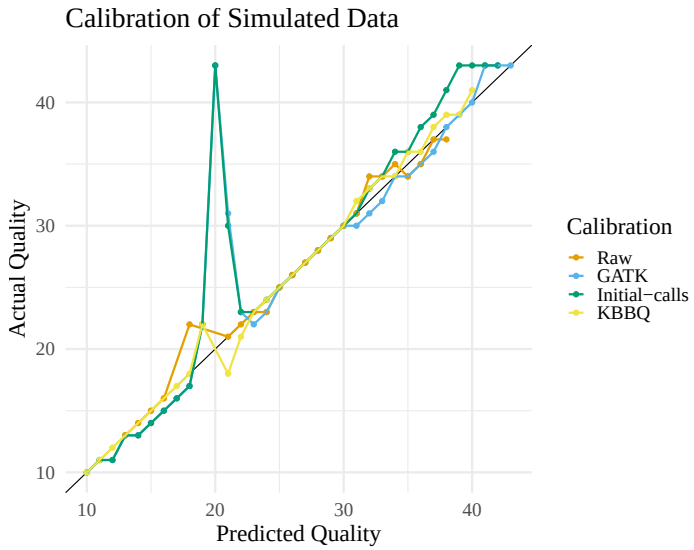


Figure 3.3: Comparison of Calibration Methods on Simulated Data | Plot of calibration errors before and after recalibrating reads simulated with ART. Raw is the unmodified simulated quality scores. GATK is the result of using BaseRecalibrator and ApplyBQSR using the set of true simulated variants as the known sites file. Initial-calls is the result of using BaseRecalibrator and ApplyBQSR but using an initial HaplotypeCaller callset rather than the true set of variants. KBBQ is the result of using KBBQ, a reference-free recalibration method. Note that the raw calibration lies almost all on the 1-to-1 line, meaning the data is already fairly well-calibrated.

callset caused calibration errors in high scoring bases. GATK recalibration caused a large calibration error for bases assigned a score of 20, which contained very few errors. However, there were also relatively few bases assigned a score of 20, as seen in Figure 3.4. The errors are summarized in Table 3.3; KBBQ has the lowest RMSE, while GATK calibration using an initial HaplotypeCaller callset has the highest RMSE.

3.4 Discussion

As the chimp-aligned calibration shows, GATK BQSR struggles to accurately recalibrate quality scores when the reference doesn't closely match the sample. This could cause issues when attempting to use GATK BQSR on data from a non-model organism if the reference contains errors or is otherwise distant to the sampled organism. False negatives are likely to be numerous in almost all but the most well-studied

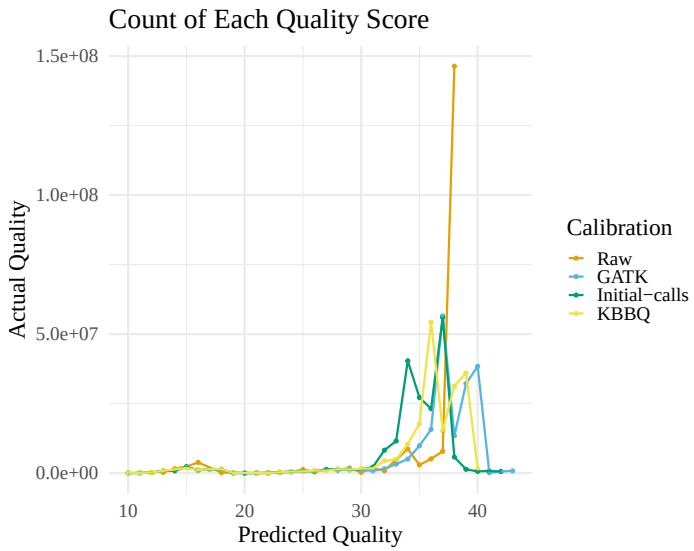


Figure 3.4: Quality Score Distribution of Recalibrated Simulated Data | Plot of the counts of each quality score in each dataset recalibrated by different methods. Raw is the unmodified simulated quality scores. GATK is the result of using BaseRecalibrator and ApplyBQSR using the set of true simulated variants as the known sites file. Initial-calls is the result of using BaseRecalibrator and ApplyBQSR but using an initial HaplotypeCaller callset rather than the true set of variants. KBBQ is the result of using KBBQ, a reference-free recalibration method. Most quality scores are above 30, and all the recalibration methods yield a distribution near that mode.

Calibration Method	RMSE
Raw	1.06
GATK	4.40
Initial-calls	4.54
KBBQ	0.93

Table 3.3: Errors of Different Calibration Methods on Simulated Data | Root mean squared error of quality score for simulated reads recalibrated using different methods. Raw is the unmodified simulated quality scores. GATK is the result of using BaseRecalibrator and ApplyBQSR using the set of true simulated variants as the known sites file. Initial-calls is the result of using BaseRecalibrator and ApplyBQSR but using an initial HaplotypeCaller callset rather than the true set of variants. KBBQ is the result of using KBBQ, a reference-free recalibration method. See Figure 3.3 for a visualization of the calibration errors.

samples (Bobo et al. 2016), and false positives are similarly likely when using poor quality, draft reference genomes that cause alignment errors. This means in many datasets, while its performance may be acceptable if the false negative and false positive rate are sufficiently low, GATK BaseRecalibrator may not be the best recalibration method. This is shown in Figure 3.2, which shows KBBQ performing nearly as well as GATK BQSR with perfect knowledge of variable sites. However, KBBQ doesn't use any reference or any variant site information to do recalibration. KBBQ also performs much better than GATK's recommended procedure when a database of variable sites is not available, as shown in the Chimp-GATK calibration.

Recalibration of the simulated reads shows that KBBQ is able to recalibrate reads even if they are already well-calibrated without severely damaging the calibration of the quality scores. In contrast, using GATK—even with an accurate database of variable sites—introduces additional error in quality score calibration to the reads, with both GATK-based methods having much higher RMSE than the raw quality scores. Additionally, even though the reads were already fairly well-calibrated, KBBQ recalibration confers a slight improvement to the RMSE of the quality scores. All methods yielded quality scores near 38, the mode of the raw quality data.

3.5 Conclusion

Base quality score recalibration is an important procedure to ensure base quality scores are accurate before variant calling. However, the most popular method for doing BQSR is not easy to do if the sequenced organism is a non-model organism. I developed the software tool KBBQ to recalibrate base quality scores without a reference or database of variable sites to overcome these deficiencies. Since it doesn't use a database of variable sites or a reference, the quality of these resources is immaterial to the quality of the resulting calibration.

To evaluate KBBQ, I emulated GATK's procedure for calibration when a database of variable sites is unavailable by aligning a synthetic diploid human benchmarking dataset to the chimp genome and calling variants to use as the database of variable sites. This method produces a calibration worse than KBBQ. I further simulated reads from a draft assembly to simulate sequencing of a highly heterozygous non-model organism. Though the simulated data was fairly well-calibrated, GATK calibration with perfect knowledge of the heterozygous sites and using an initial set of calls introduced more error into the quality score calibration than was in the original data while KBBQ recalibration slightly reduced it. Thus, when a database of variable sites or reference is unavailable or of poor quality, KBBQ is an effective method for base quality score recalibration.

REFERENCES

Auwera, Geraldine A., Mauricio O. Carneiro, Christopher Hartl, Ryan Poplin, Guillermo del Angel, Ami Levy-Moonshine, Tadeusz Jordan, et al. 2013. "From FastQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline." *Current Protocols in Bioinformatics* 43 (1). https://doi.org/10.1002/0471250953.bi1110s43.

Bobo, Dean, Mikhail Lipatov, Juan L. Rodriguez-Flores, Adam Auton, and Brenna M. Henn. 2016. *False Negatives Are a Significant Feature of Next Generation Sequencing Callsets.* Preprint. Bioinformatics. https://doi.org/10.1101/066043.

Cabanski, Christopher R., Keary Cavin, Chris Bizon, Matthew D. Wilkerson, Joel S. Parker, Kirk C. Wilhelmsen, Charles M. Perou, J. S. Marron, and D. Neil Hayes. 2012. "ReQON: a Bioconductor package for recalibrating quality scores from next-generation sequencing data." *BMC bioinformatics* 13:221. https://doi.org/10.1186/1471-2105-13-221.

Callahan, Benjamin J., Paul J. McMurdie, Michael J. Rosen, Andrew W. Han, Amy Jo A. Johnson, and Susan P. Holmes. 2016. "DADA2: High-resolution sample inference from Illumina amplicon data." *Nature Methods* 13 (7): 581–583. https://doi.org/10.1038/nmeth.3869.

Chung, Jade C. S., and Swaine L. Chen. 2017. "Lacer: accurate base quality score recalibration for improving variant calling from next-generation sequencing data in any organism." *bioRxiv,* 130732. https://doi.org/10.1101/130732.

Cibulskis, Kristian, Michael S. Lawrence, Scott L. Carter, Andrey Sivachenko, David Jaffe, Carrie Sougnez, Stacey Gabriel, Matthew Meyerson, Eric S. Lander, and Gad Getz. 2013. "Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples." *Nature Biotechnology* 31 (3): 213–219. https://doi.org/10.1038/nbt.2514.

Cock, Peter J. A., Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. 2010. "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants." *Nucleic Acids Research* 38 (6): 1767–1771. https://doi.org/10.1093/nar/gkp1137.

Ewing, B., L. Hillier, M. C. Wendl, and P. Green. 1998. "Base-calling of automated sequencer traces using phred. I. Accuracy assessment." *Genome Research* 8 (3): 175–185. https://doi.org/10.1101/gr.8.3.175.

Ewing, Brent, and Phil Green. 1998. "Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities." *Genome Research* 8 (3): 186–194. https://doi.org/10.1101/gr.8.3.186.

Fox, Edward J., Kate S. Reid-Bayliss, Mary J. Emond, and Lawrence A. Loeb. 2014. "Accuracy of Next Generation Sequencing Platforms." *Next generation, sequencing & applications* 1. https://doi.org/10.4172/jngsa.1000106.

Garrison, Erik, and Gabor Marth. 2012. "Haplotype-based variant detection from short-read sequencing." *arXiv:1207.3907 [q-bio],* arXiv: 1207.3907. http://arxiv.org/abs/1207.3907.

Huang, Weichun, Leping Li, Jason R. Myers, and Gabor T. Marth. 2012. "ART: a next-generation sequencing read simulator." *Bioinformatics* 28 (4): 593–594. https://doi.org/10.1093/bioinformatics/btr708.

Kao, Wei-Chun, and Yun S. Song. 2011. "naiveBayesCall: an efficient model-based base-calling algorithm for high-throughput sequencing." *Journal of Computational Biology: A Journal of Computational Molecular Cell Biology* 18 (3): 365–377. https://doi.org/10.1089/cmb.2010.0247.

Külheim, Carsten, Suat Hui Yeoh, Jens Maintz, William J. Foley, and Gavin F. Moran. 2009. "Comparative SNP diversity among four Eucalyptus species for genes from secondary metabolite biosynthetic pathways." *BMC Genomics* 10 (1): 1–11. https://doi.org/10.1186/1471-2164-10-452.

Li, Heng, Jonathan M. Bloom, Yossi Farjoun, Mark Fleharty, Laura Gauthier, Benjamin Neale, and Daniel MacArthur. 2018. "A synthetic-diploid benchmark for accurate variant-calling evaluation." *Nature Methods* 15 (8): 595–597. https://doi.org/10.1038/s41592-018-0054-7.

Li, Heng, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. 2009. "The Sequence Alignment/Map format and SAMtools." *Bioinformatics (Oxford, England)* 25 (16): 2078–2079. https://doi.org/10.1093/bioinformatics/btp352.

Li, R., C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. 2009. "SOAP2: an improved ultrafast tool for short read alignment." *Bioinformatics* 25 (15): 1966–1967. https://doi.org/10.1093/bioinformatics/btp336.

Ma, Xiaotu, Ying Shao, Liqing Tian, Diane A. Flasch, Heather L. Mulder, Michael N. Edmonson, Yu Liu, et al. 2019. "Analysis of error profiles in deep next-generation sequencing data." *Genome Biology* 20 (1): 50. https://doi.org/10.1186/s13059-019-1659-6.

Malysa, Greg, Mikel Hernaez, Idoia Ochoa, Milind Rao, Karthik Ganesan, and Tsachy Weissman. 2015. "QVZ: lossy compression of quality values." *Bioinformatics* 31 (19): 3122–3129. https://doi.org/10.1093/bioinformatics/btv330.

Massingham, Tim, and Nick Goldman. 2012. "All Your Base: a fast and accurate probabilistic approach to base calling." *Genome Biology* 13 (2): R13. https://doi.org/10.1186/gb-2012-13-2-r13.

Meacham, Frazer, Dario Boffelli, Joseph Dhahbi, David IK Martin, Meromit Singer, and Lior Pachter. 2011. "Identification and correction of systematic error in high-throughput sequence data." *BMC Bioinformatics* 12 (1): 451. https://doi.org/10.1186/1471-2105-12-451.

Nakamura, Kensuke, Taku Oshima, Takuya Morimoto, Shun Ikeda, Hirofumi Yoshikawa, Yuh Shiwa, Shu Ishikawa, et al. 2011. "Sequence-specific error profile of Illumina sequencers." *Nucleic Acids Research* 39 (13): e90. https://doi.org/10.1093/nar/gkr344.

Ni, Shengyu, and Mark Stoneking. 2016. "Improvement in detection of minor alleles in next generation sequencing by base quality recalibration." *BMC genomics* 17:139. https://doi.org/10.1186/s12864-016-2463-2.

Poplin, Ryan, Valentin Ruano-Rubio, Mark A. DePristo, Tim J. Fennell, Mauricio O. Carneiro, Geraldine A. Van der Auwera, David E. Kling, et al. 2018. "Scaling accurate genetic variant discovery to tens of thousands of samples." *bioRxiv,* https://doi.org/10.1101/201178.

Reducing Whole-Genome Data Storage Footprint. 2014. White paper 970-2012-013. Illumina, Inc.

Schirmer, Melanie, Rosalinda D'Amore, Umer Z. Ijaz, Neil Hall, and Christopher Quince. 2016. "Illumina error profiles: resolving fine-scale variation in metagenomic sequencing data." *BMC Bioinformatics* 17 (1): 125. https://doi.org/10.1186/s12859-016-0976-y.

Schirmer, Melanie, Umer Z. Ijaz, Rosalinda D'Amore, Neil Hall, William T. Sloan, and Christopher Quince. 2015. "Insight into biases and sequencing errors for amplicon sequencing with the Illumina MiSeq platform." *Nucleic Acids Research* 43 (6): e37. https://doi.org/10.1093/nar/gku1341.

Sedlazeck, Fritz J., Philipp Rescheneder, and Arndt von Haeseler. 2013. "NextGenMap: fast and accurate read mapping in highly polymorphic genomes." *Bioinformatics* 29 (21): 2790–2791. https://doi.org/10.1093/bioinformatics/btt468.

Shibuya, Yoshihiro, and Matteo Comin. 2019. "Better quality score compression through sequence-based quality smoothing." *BMC Bioinformatics* 20 (9): 302. https://doi.org/10.1186/s12859-019-2883-5.

Song, Li, Liliana Florea, and Ben Langmead. 2014. "Lighter: fast and memory-efficient sequencing error correction without counting." *Genome Biology* 15 (11): 509. https://doi.org/10.1186/s13059-014-0509-9.

Van der Auwera, Geraldine A. 2020a. "Geraldine Van der Auwera on Twitter." Twitter. https://twitter.com/VdaGeraldine/status/1311000033127550976.

Van der Auwera, Geraldine A. 2020b. "Geraldine Van der Auwera on Twitter." Twitter. https://twitter.com/VdaGeraldine/status/1296181178534440963.

Waterson, Robert H., Eric S. Lander, Richard K. Wilson, and The Chimpanzee Sequencing and Analysis Consortium. 2005. "Initial sequence of the chimpanzee genome and comparison with the human genome." *Nature* 437 (7055): 69–87. https://doi.org/10.1038/nature04072.

Yu, Y. William, Deniz Yorukoglu, Jian Peng, and Bonnie Berger. 2015. "Quality score compression improves genotyping accuracy." *Nature Biotechnology* 33 (3): 240–243. https://doi.org/10.1038/nbt.3170.

Yue, Jia-Xing, and Gianni Liti. 2019. "simuG: a general-purpose genome simulator." *Bioinformatics* 35 (21): 4442–4444. https://doi.org/10.1093/bioinformatics/btz424.

Zook, Justin M., Daniel Samarov, Jennifer McDaniel, Shurjo K. Sen, and Marc Salit. 2012. "Synthetic spike-in standards improve run-specific systematic error analysis for DNA and RNA sequencing." *PloS One* 7 (7): e41356. https://doi.org/10.1371/journal.pone.0041356.

Chapter 4

EVALUATING THE IMPACT OF QUALITY SCORE CALIBRATION ON VARIANT CALLING

4.1 Introduction

DNA sequencing is a powerful tool applied in many fields of biology. Understanding how DNA and changes in DNA influence organisms and how they interact with their environment is a fundamental goal of genetics. Correctly identifying the composition of a sampled DNA molecule is therefore an important first step for many genetic studies. However, this task is not trivial; sequencing technology is inherently prone to errors which can make it difficult to discern true biological effects from technical errors (Fox et al. 2014; Wu et al. 2017).

Because of this error, the same piece of DNA is usually sequenced multiple times to attempt to get multiple measurements and a model is applied to infer the sample genotype (Garrison and Marth 2012; Li 2011a; Poplin et al. 2018). While a few sequencing errors do not impede accurate inference of the genotype of a monoploid sample, sequencing errors can be difficult to distinguish from heterozygosity in diploid organisms. This problem is even more difficult for samples with higher ploidy. However, using a statistical model enables a systematic approach to genotyping every site in the genome.

Additionally, a model allows the incorporation of auxiliary information when deciding the genotype at a site in a sample. For example, variant calling models often incorporate population genetic parameters that can help distinguish between sequencing errors and biological variation. The expected heterozygosity parameter θ is par-

ticularly important; it usually parameterizes some prior on the probability of a base matching the reference genome. If this deviates significantly from the true value, the model may mistakenly classify truly heterozygous sites as errors or vice versa.

The goal of using a model to call variants is to integrate a wide variety of information to make a principled decision about whether or not a variant exists. An important source of information on the trustworthiness of each sequenced base is the base quality score (Ewing et al. 1998; Ewing and Green 1998). This is a score, usually between 2 and 43, that encodes the probability the sequence is an error in the Phred scale. That is, the score $Q = -10 \log_{10} P(\text{Base is an error})$. This number is then rounded to the nearest integer and encoded as a character, shifted by 33, in the sequencing data.

Since these numbers can vary significantly in the same line with few patterns, quality scores can't usually be compressed very well by common compression algorithms. That gives sequencing data large file sizes that make it expensive to store the data for a lengthy period of time. Thus, these quality scores are often binned to reduce the amount of variation in the scores, enabling improved compression (Malysa et al. 2015; *Reducing Whole-Genome Data Storage Footprint* 2014; Shibuya and Comin 2019; Yu et al. 2015). Most binning methods attempt to do so in a way that minimizes the difference between the error rate of the bin and the error rate the bin should have according to its assigned score. However, there can still be a disconnect between the predicted error rate and the actual error rate. Furthermore, as the resolution of quality scores is reduced, some information about the true score at each base is lost. Interestingly, Yu et al. (2015) find that compressing scores by removing quality scores from all sites that are not likely to vary based on the k-mer composition of the data, but keeping quality scores intact at sites that may contain errors or true variants increases the quality of output genotype calls.

To restore quality score resolution before analysis and to ensure quality scores accurately reflect the probability of error at every base, a process known as base quality score recalibration (BQSR) is sometimes performed (Auwera et al. 2013; Pfeifer 2017). This attempts to use known information about sequencing errors to calibrate the quality scores. The most common method for doing so implemented in the Genome Analysis Toolkit (GATK) requires aligned reads and a set of variable sites that are removed from analysis. The algorithm then looks for bases that do not match the reference, assumes they are errors, and uses those errors to recalibrate the quality scores. See Chapter 3 for more information on quality scores and base quality score recalibration.

The ultimate goal of quality scores and base quality score recalibration is to make it easier to identify erroneous bases, which should improve the set of variants a variant caller emits. However, evidence supporting the use of base quality score recalibration is limited and whether base quality score recalibration is worthwhile is subject to debate. The procedure has aided in detecting rare variants (Ni and Stoneking 2016), and a recalibration procedure that integrates mapping errors into quality scores seems to improve variant detection (Li 2011b). The goal of this chapter is to investigate the degree that base quality score recalibration affects variant calling to enable informed decisions for constructing variant calling pipelines, especially for non-model organisms.

4.2 Methods

In order to determine how misspecification of the database of variable sites affects the calibration of GATK's BQSR procedure (Auwera et al. 2013), I simulated datasets with various levels of false negative and false positive variants. In this case, false negative variants in the database of variable sites causes a site that should be ignored

to not be ignored, greatly increasing the number of bases that GATK classifies as sequencing errors. On the other hand, false positive variants remove a site from GATK classification, so the effect is likely to be small except for large false positive rates.

To create each simulated dataset, I began with the high coverage synthetic diploid dataset described in H. Li et al. (2018) with reads aligned to hg19. This dataset was constructed by mixing DNA from the CHM1 and CHM13 hydatidiform mole cell lines to create a synthetic diploid sample. It includes a VCF of variant calls and a BED file representing regions where the authors are confident the calls in the VCF are correct. I subset the full dataset to only reads aligned on Chromosome 1 and overlapping the BED of confident regions using the samtools view command (H. Li et al. 2009) and the -L parameter. I then used the samtools (version 1.10) view command with the -F 3844 flag to remove unmapped, secondary, and supplementary alignments as well as reads marked as failing quality checks and as PCR duplicates. I then used the fixmate and view commands with flag -f 1 to remove any singleton reads.

To create each false negative dataset, an appropriate number of sites from the VCF were randomly sampled with BCFTools (version 1.10.2) and the shuf (version 8.32) program. To create each false positive dataset, all sites from the VCF were extracted in BED format with the BCFTools query command, then subtracted from the BED of confident regions with bedtools (version v2.27.1) subtract (Quinlan and Hall 2010). The appropriate number of sites were then sampled with the shuf program and appended to the sites from the VCF to generate the BED file of all sites to exclude. Thus, I created variable site sets that were artificially crafted to have different rates of false positive and false negative calls, from 0 to 100 in steps of 20. These files were then provided as input to the GATK (version v4.1.8.0-5-g1836ab0-SNAPSHOT) BaseRecalibrator tool and calibration was evaluated with GATK's AnalyzeCovariates

tool.

As a control, I compared each result to the raw, uncalibrated data. To see how a reference-free recalibration method affected variant calling, I also recalibrated the data with KBBQ (commit ID 9167b72599892a33493d8ebdc01fac33990c1738), with the options --genomesize 214206308 and -a .15 .

I then called variants on each dataset using GATK HaplotypeCaller (Poplin et al. 2018). The false positive rate of 100% produced no output calls except in the case of 0% false negative rate, which produced fewer calls than any other set. All datasets containing a 100% false positive rate were therefore discarded. I evaluated these output SNP calls using RealTimeGenomics' RTG Tools (version 39691f9f) vcfeval program (Cleary et al. 2015) and the confident region set along with the confident call set. I first ran the tool using the default settings, which also produces a ROC curve for the calls' GQ annotation. I also ran the tool using the --vcf-score-field=QUAL option, which produces ROC curves for the site's QUAL annotation. This enabled comparison of the sensitivity and false positive rate trade-off of the two scores for each dataset. These two annotations were chosen because they represent an overall summary of the quality of the called variants.

The false positive rate for the ROC curves was found by dividing the number of false positive calls in each dataset at each value of QUAL or GC and below by the number of true negative sites. The number of true negative sites was obtained by subtracting each true positive variant from the BED file containing the confident regions of the callset. This rate had to be computed, as vcfeval does not report a false positive rate. As the RTG-tools manual states, the program doesn't try to compute the possible number of true negatives, as calls could in theory occupy many or no reference bases. However, as I analyze only SNP data here, estimating the true number of negatives as the number of sites that are within the confident regions but

outside variant sites specified by the truth set should be sufficient. The true positive rate was calculated by taking the number of true positive calls in each dataset for each value of QUAL or GC and dividing by the number of positive calls in the truth set, as reported by the vcfeval output.

To evaluate the output calls, I plotted the number of false positive and true positive SNP calls for each dataset. I also constructed a heat map showing the F-statistic of each dataset. This is the harmonic mean of the sensitivity and precision of the calls and is one way to summarize the accuracy of the calls. All plots and heatmaps were made using R (R Core Team 2020) and ggplot2 (Wickham 2016) using the viridis color palette (Garnier 2018) and the Okabe-Ito palette implemented in colorblindr to ensure the plots remain interpretable by those with colorblindness and after photocopying.

To see how filtering the calls affected their quality, I then identically filtered each variant set with two filters using bcftools view (H. Li et al. 2009). The first filter is a depth filter, only accepting variants that have more than 35 aligned reads and fewer than 65. This is within two standard deviations of the mean sequencing depth of 50, assuming the read depths are Poisson distributed. The second filter is a QUAL filter, accepting only calls with a QUAL of over 75. This threshold was chosen because it is approximately the bottom 1% of QUAL values in each dataset and it is the lowest QUAL value that maximizes the F-statistic of all the datasets. I then calculated the same statistics and made the same plots using the filtered calls.

In order to investigate how calibration affected variant calling performance on a medium-coverage dataset from a highly heterozygous sample with a low-quality reference, as might be the case when sequencing a non-model organism, I created a simulated dataset. I first randomly sampled contigs from the first release of the *Eucalyptus grandis* reference (SRA accession AUSX00000000.1) until I sampled 5 million

base-pairs, which at 20X coverage for a diploid genome would yield approximately 200 million base-pairs of synthetic sequence data. I then simulated heterozygous sites in the genome with simuG (version v1.0.0) (Yue and Liti 2019) with a transition/transversion ratio of 2 and 125,000 SNPs to emulate the conditions observed in eucalypts (Külheim et al. 2009). I concatenated the subsampled and simulated genomes to create a diploid reference and simulated reads from it using ART (version MountRainier-2016-06-05; Huang et al. 2012). I simulated 101-bp long reads with a mean fragment length of 300 and standard deviation of 20, with 20X coverage. I then aligned the simulated reads to the genome with NextGenMap version 0.5.2 (Sedlazeck, Rescheneder, and Haeseler 2013) and recalibrated the data 3 ways: 1) with GATK using the true set of heterozygous sites as the database of variable sites, 2) with GATK using a set of variant calls generated with HaplotypeCaller with the `--stand-call-conf` set to 50 as the database of variable sites, and 3) using KBBQ (see Chapter 3) with the genome size parameter set to 5 million.

Including the simulator-assigned raw scores, this yielded four datasets with different quality scores to call variants with and compare the results. HaplotypeCaller was used on each dataset and given the subsampled genome as a reference and called with the `--heterozygosity 0.025` option to represent the approximate heterozygosity of the data. As simuG doesn't output sample genotypes and the simulated sample is heterozygous at every site in the truth VCF, a sample column was added and the genotype of the sample at every site set to be 0|1 using sed. This VCF was then used as the truth comparison for rtg-tools vcfeval command and used to evaluate each set of variant calls. vcfeval was run in annotate mode and the output parsed using hapdip.js (version r66; Li 2014) to generate summaries of each callset. Finally, vcfeval was also run in roc-only mode, once with the `--vcf-score-field` set to GQ and once set to QUAL to produce ROC curves for each of these scores. The false

negative rate was obtained by dividing the number of output false negatives by the length of the genome minus the number of simulated heterozygous sites.

Finally, I ran KBBQ with the specified genome size 605951511 and coverage 240 to recalibrate a set of reads sequenced from an individual of the non-model species *Eucalyptus melliodora* (Orr et al. 2020). This dataset contains reads from 3 leaves each from 8 branches, each leaf sequenced to a depth of approximately 10X. I then called variants using HaplotypeCaller and filtered the calls using the same filters from Orr et al. (2020). I use BCFTools view to record the number of variable sites that were also recorded in the previously identified set of calls found using DeNovoGear (Ramu et al. 2013). I then compared these values to those found using the raw, uncalibrated reads and those found using the GATK recalibrated reads in Orr et al. (2020).

Scripts to reproduce these analyses are included in Appendix G.

4.3 Results

To identify how errors in the database of variable sites affects variant calling, I simulated different datasets with known false positive rates, plotted the calibration and calculated the root mean squared error (RMSE) of the data recalibrated with the model trained using each database as the known sites input to GATK BaseRecalibrator. These plots are shown in figure 4.1, and the RMSE of the quality score for each dataset is shown in table 4.1. For all these datasets, the false negative rate is 0. As the false positive rate increases, the degree of miscalibration doesn't change significantly except for the 100% false positive rate dataset, which is very poorly calibrated; however, this calibration is likely an artifact (see section 4.4).

I also simulated different databases of variable sites with differing false negative rates and similarly used it to recalibrate the CHM1-CHM13 data. In these datasets, the false positive rate is 0%. The plotted calibration and RMSE of the recalibrated

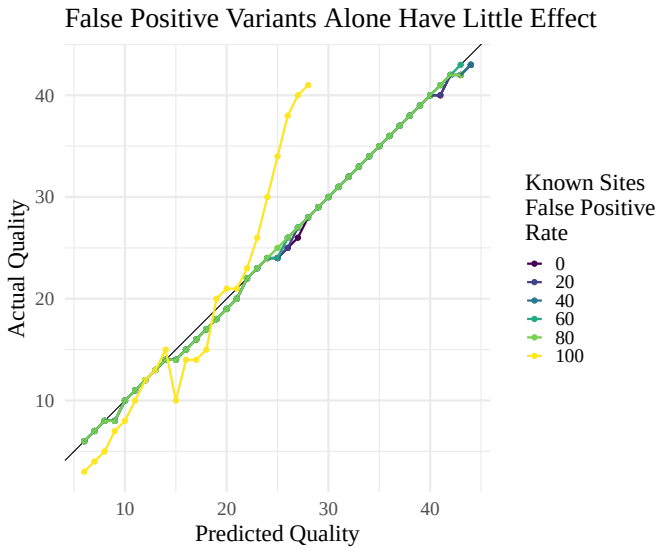


Figure 4.1: False Positive Only Calibration | Base quality score calibration for a range of false positives in the database of variable sites. The false negative rate for all datasets is zero. Increasing the false positive rate does not significantly impact the quality of the calibration. The poor calibration at a 100% false positive rate is likely an artifact (see Section 4.4).

False Positive Rate	RMSE
0	0.60
20	0.58
40	0.53
60	0.49
80	0.49
100	5.50

Table 4.1: False Positive Calibration Errors | The root mean squared error of quality score for reads recalibrated using a database of variable sites with different false positive rates. The false negative rate for each dataset is 0%.

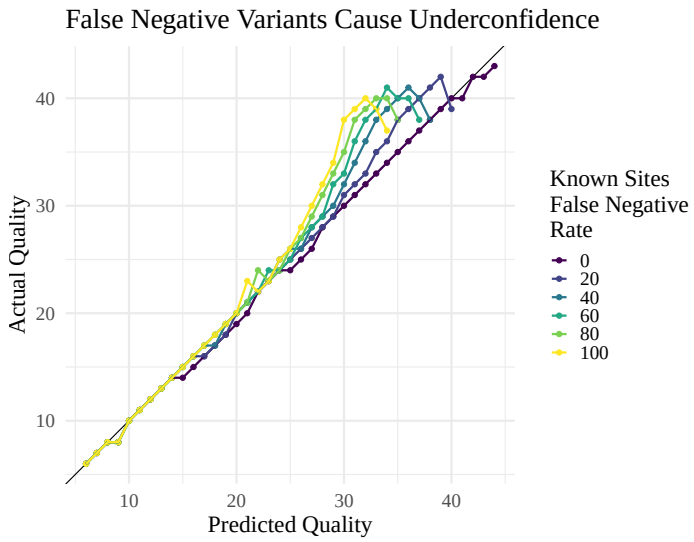


Figure 4.2: False Negative Only Calibration | Base quality score calibration for a range of false negatives in the database of variable sites. The false positive rate for all datasets is zero. Increasing the false negative rate significantly decreases the quality of the calibration, causing increasing underconfidence in quality scores as the false negative rate rises.

data is shown in figure 4.2 and table 4.2. As the false negative rate increases, the degree of miscalibration also steadily increases.

To see if there were any interactive effects of false positive rate and false negative rate, I also simulated datasets with varying false positive and false negative rates. The RMSE of the calibrated scores is reported in Table 4.3 and summarized in Figure 4.3. These datasets were simulated separately from the above datasets, so there are slight differences in the calibration of the resulting reads. As before, the number of false negatives significantly impacted calibration quality. In contrast to the false positive only dataset with a 0% false negative rate, increasing the false positive rate also increased the amount of error in the calibration. Thus, false positives in the database of variable sites seem to enhance miscalibration caused by false negatives.

To evaluate the output calls, I plotted the number of true positives and false positives for each dataset recalibrated using sets of variable sites with differing false

False Negative Rate	RMSE
0	0.60
20	1.32
40	2.10
60	2.57
80	2.90
100	3.20

Table 4.2: False Negative Calibration Errors | Root mean squared error of quality score for reads recalibrated with a database of variable sites simulated with the given false negative rate. The false positive rate for each dataset is 0%.

FPR	FNR					
	0	20	40	60	80	100
0	.641	1.45	2.08	2.73	2.99	3.38
20	.599	1.73	2.54	2.96	3.39	3.76
40	.555	2.02	2.71	3.50	3.84	4.16
60	.531	2.58	3.37	4.27	4.51	4.73
80	.593	3.52	4.62	5.38	5.73	6.32
100	8.05	22.0	24.3	26.4	25.8	28.9

Table 4.3: Combined Calibration Errors | Root mean squared error of base quality score for data calibrated with databases of variable sites containing different levels of false positives and false negatives. The columns indicate false positive rates, the rows indicate false negative rates. The values in each cell are the RMSE of the quality scores for the reads recalibrated with the database of variable sites with false positive and false negative rate appropriate for its row and column. See Figure 4.4 for a graphical representation. The data in the 100% false positive rows are likely artifacts; see section 4.4 for more information.

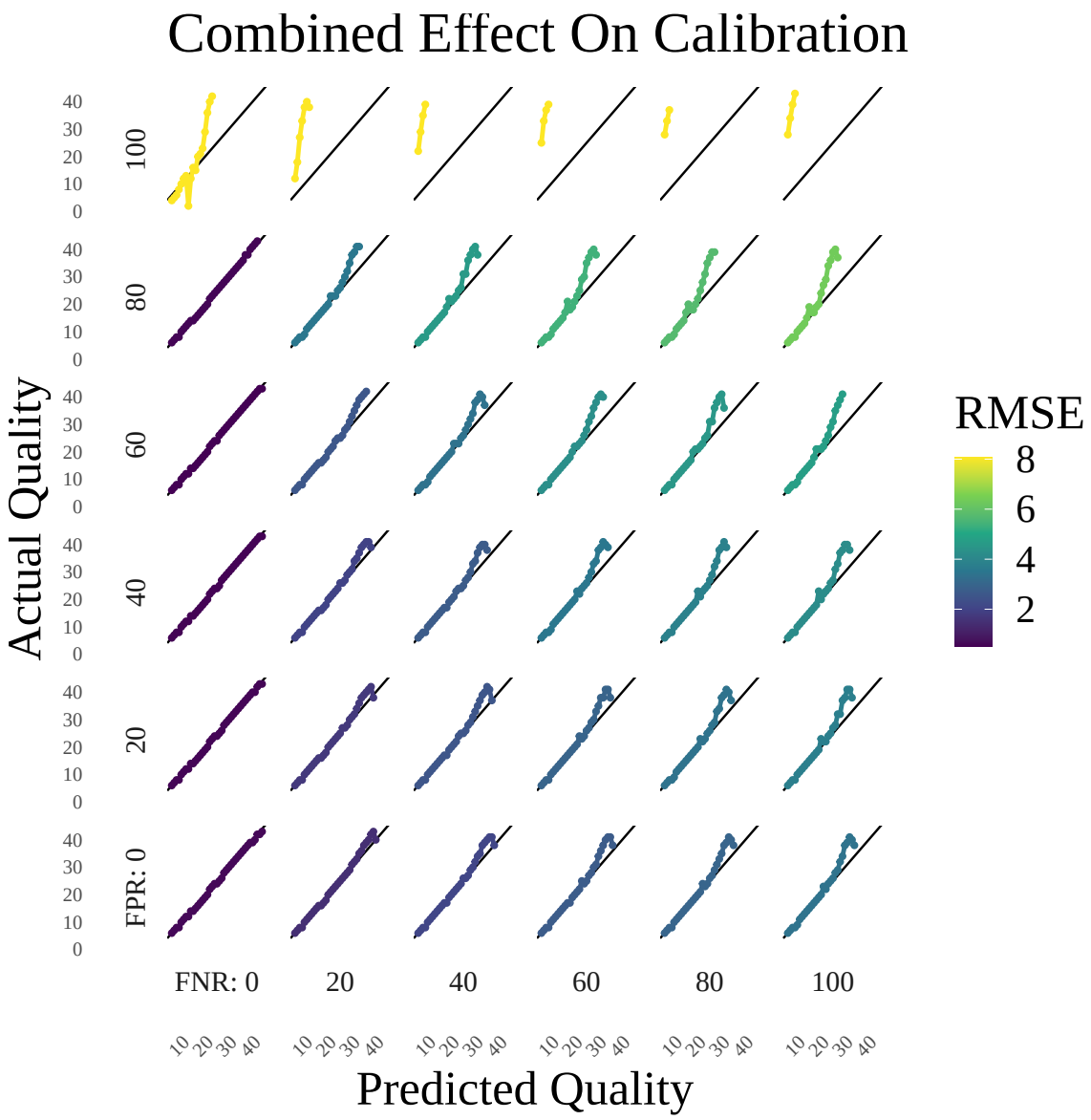


Figure 4.3: Combined Calibration | Calibration of base quality scores of reads recalibrated using a database of variable sites with varying ranges of false negatives and false positives. The RMSE of the quality scores of the resulting calibration are used to color each line. Across the columns are each false negative rate, and each row represents a false positive rate. Except for at a false negative rate of 0, increasing either the false positive rate or the false negative rate increases the RMSE. See Table 4.3 for the RMSE values and the discussion in Section 4.4 about false positive rates of 100%

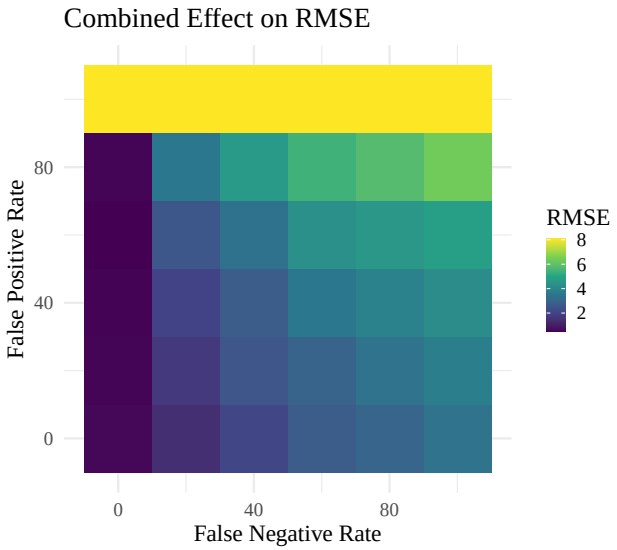


Figure 4.4: Combined Calibration Heat Map | Heat map of base quality calibration of varying ranges of false negatives and false positives. Databases of variable sites with differing false positive and false negative rates were constructed and the RMSE of the quality scores of the resulting calibration were calculated. Across the columns are each false negative rate, and each row represents a false positive rate. Except for at a false negative rate of 0, increasing either the false positive rate or the false negative rate increases the RMSE. See Table 4.3 for the RMSE values and the discussion in Section 4.4 about false positive rates of 100%

negative and false negative rates (Figure 4.5). This shows that after recalibration, improved calibration increases the number of true positive variants called. However, improved calibration also increases the number of false positives. More detail about the relationship between false positives and negatives in the database of calibrated sites are shown in the heat maps in Figure 4.6. Interestingly, the raw uncalibrated data exhibited the most positive calls of any other callset.

To determine the impact of these variants on the overall quality of the callsets, I constructed heatmaps of the sensitivity and precision of each dataset (Figure 4.7). As the data become more well-calibrated, the sensitivity of the caller increases; however, the precision of the caller also decreases. This is also seen in the raw calibration: it has the highest sensitivity and lowest specificity of all the datasets. This means that the caller is able to detect more true variants, but a higher proportion of the calls it

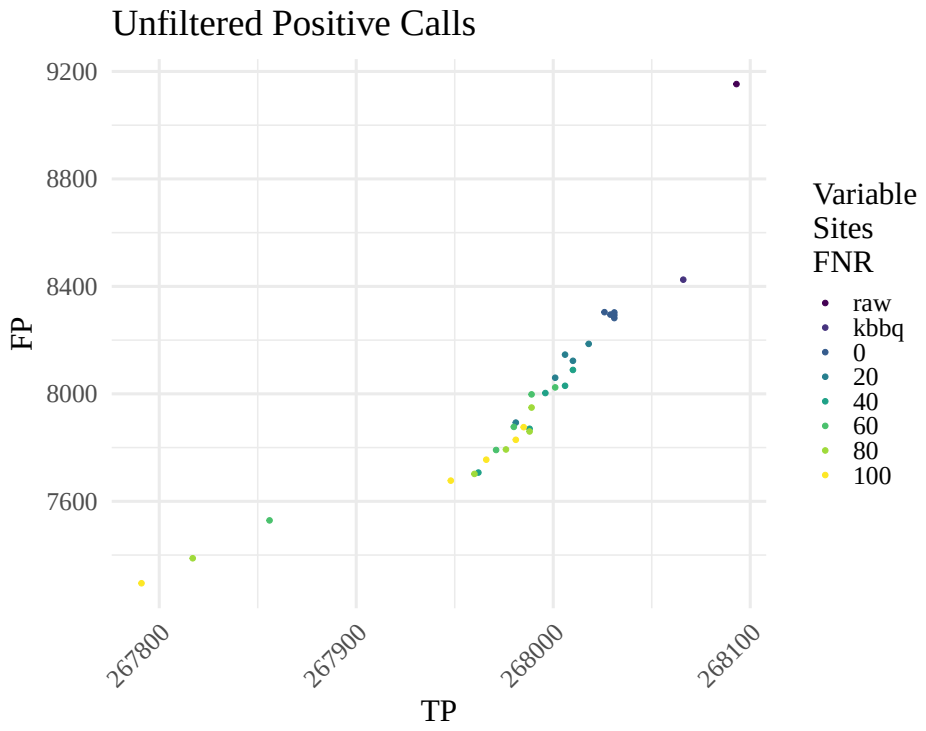


Figure 4.5: More Accurate Known Sites Increases Number of Unfiltered Positive Calls | The output number of positive calls for each dataset. The false negative rate of the set of variable sites used to calibrate the reads used to produce each callset is shown. As the false negative rate decreases, the number of both true positive and false positive calls increases. The raw, uncalibrated data exhibits the highest number of positive calls.

makes are false positives. To visualize the trade-off between sensitivity and precision, I plotted the precision against the sensitivity of each dataset, shown in Figure 4.12. To summarize the overall effect on accuracy, I show how the F-statistic changes across each dataset in Figure 4.8. The callset made from the raw, uncalibrated data have the lowest F-statistic.

Since the base quality score of the reads used to support each genotype call evidently plays a role in whether to emit a variant, I wanted to see how the differences in calibration affected the annotations output by the caller. To that end, I plotted a receiver operating characteristic (ROC) curve showing the trade-off between false positive and true positive rates for variants ordered according to their QUAL and

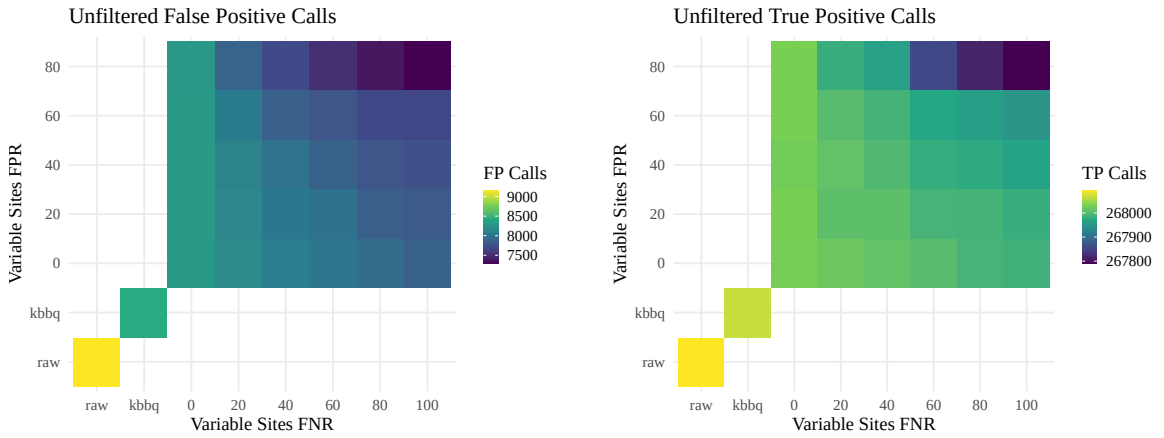


Figure 4.6: Both False Negative and False Positive Rate Contribute to Increased Number of Unfiltered Positive Calls | The output number of false positive calls for each dataset. The false negative and false positive rate of the set of variable sites used to calibrate the input reads or the name of the dataset for non-simulated datasets is shown on the X and Y axis. The color of each cell represents the number of unfiltered false positive (left) or true positive (right) SNP calls. As the false negative and positive rates of the database of variable sites decrease, the base quality scores become more calibrated, and more calibrated data produces more false positive and true positive calls. However, the raw data produces the most true positives and the most false positives.

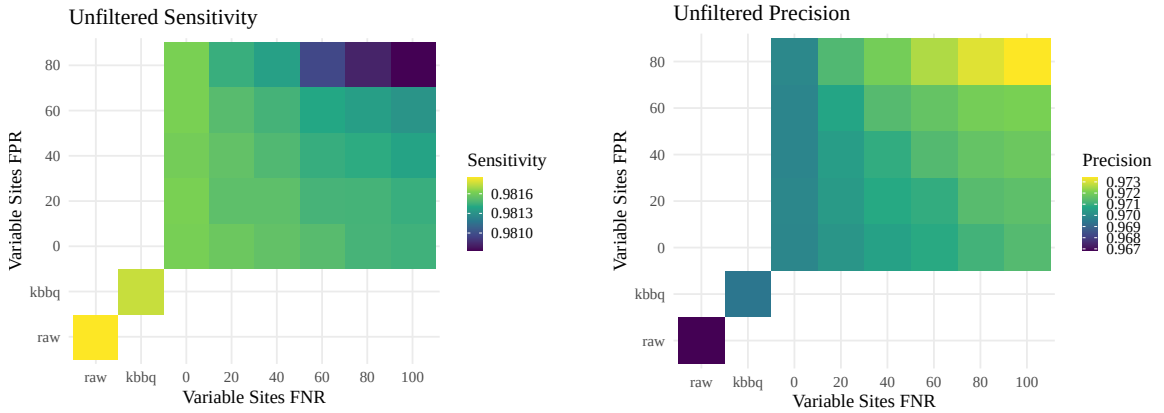


Figure 4.7: Better Calibration Increases Sensitivity and Reduces Precision | The sensitivity (left) and precision (right) of the variant caller on each dataset with no filtering. As the base quality score calibration of the input reads increases, the sensitivity of the caller increases and the precision decreases. The data with raw quality scores has the highest sensitivity and lowest precision.

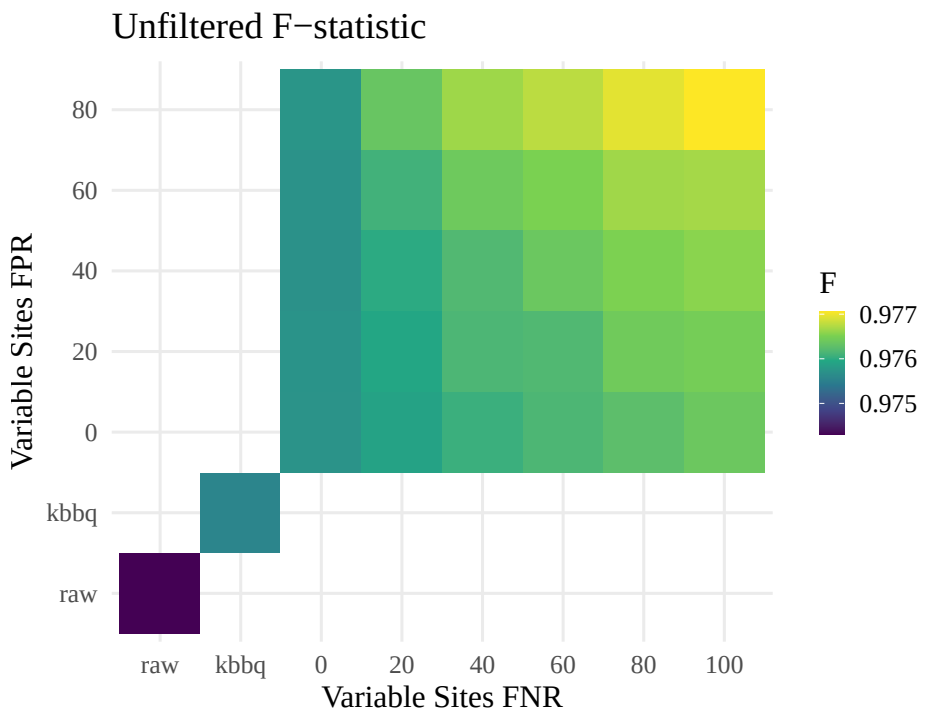


Figure 4.8: F-statistic of Unfiltered Calls | The F-statistic of the unfiltered calls of each dataset. As the base quality score calibration of the input reads increases, the F-statistic of the caller decreases. This is driven by the fact that the increase in sensitivity of the caller is much smaller than the decrease in precision. The raw, uncalibrated quality scores have the lowest F-statistic.

GQ annotations (Figure 4.9). These are both statistics that summarize the quality of the emitted site in a single, increasing score, and so are well-suited for ROC analysis. According to the VCF standard, the QUAL score is a phred-scaled probability that the ALT allele(s) present in the call are not actually present, $P(\text{ALT is wrong})$. The GQ score is the phred-scaled conditional probability the genotype call is incorrect given the site is variable, $P(\text{genotype is wrong} \mid \text{site is variant})$.

As the ROC plots show that each dataset would respond to filtering differently, I examined how the calls would improve or not after filtration. I used a small QUAL filter filtering out variants with the lower 1% of QUAL scores. This percentage coincides with the smallest QUAL value of the QUAL values that maximize the F-statistic of

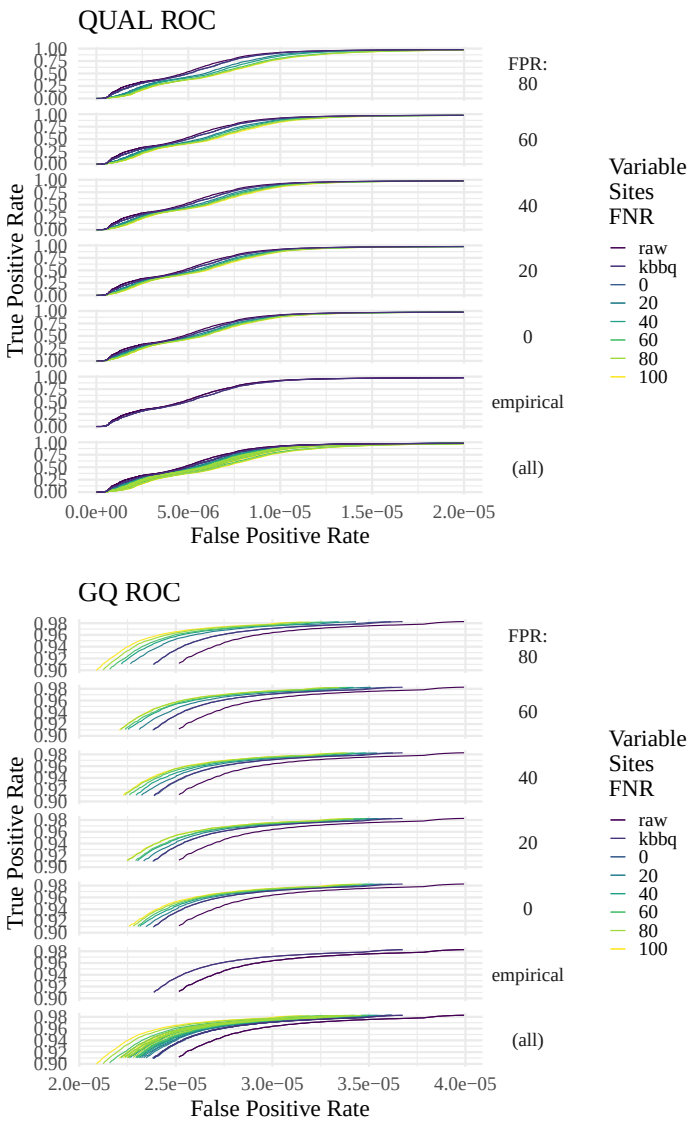


Figure 4.9: Unfiltered ROC Curves | The ROC curves for the QUAL (top) and GQ (bottom) score of an output call for the unfiltered calls of each dataset. The color of each line represents the false negative rate of the set of variant sites used as input to recalibrate the reads used to call variants. Each point in the line represents the number of false positive and false negative calls that have a score above the QUAL or GQ threshold for that point. As this threshold decreases, the number of false positives and true positives increases, though at different rates. These plots show that as the false positive rate increases, the number of true positive calls increases at a faster rate for the well-calibrated data than in other datasets for the QUAL classifier, but at a slower rate for the GQ classifier. The values for the two empirical datasets (raw and KBBQ), are shown on each plot. The other lines on each plot show values from calls made with reads that were recalibrated with variable site sets with the shown false positive rate.

every dataset. These values are shown in Figure 4.8. Depth filters are very common after variant calling, so I also used a depth filter to filter out calls that had unusually high or low depth.

This resulted in a similar pattern of positive calls as the unfiltered data, seen in Figure 4.10; better calibrated data had more false positive and true positive calls, and the raw dataset yielded the most positive calls. However, the difference between the number of true positive variants called from best-calibrated data and worst-calibrated was much larger after filtration. Before filtering, the largest difference between datasets was 313 true positive calls; this difference grows to 3631 after filtering. The opposite is observed for the false positive calls: before filtering, the largest difference in the number of false positive calls was 1858, but after filtering this difference decreases to 678. Ultimately, this alters the sensitivity and precision of each dataset such that the range of the sensitivity is increased and the range of the precision is decreased (see Figure 4.10). This is sufficient to make the best-calibrated data also have the best F-statistic of all the simulated datasets, shown in Figure 4.13. The calls made from the raw quality scores still have the best F-statistic overall.

Each of the three tested recalibration methods had only a small effect on the precision and sensitivity of HaplotypeCaller (see Figure 4.14 and Table 4.4). The Initial-calls calibration did not change the precision compared to the raw quality scores, but did slightly decrease the sensitivity of the caller. GATK calibration using the true known variable sites yielded an increase in precision and sensitivity while the reference-free method KBBQ yielded an increase in precision but a slight decrease in sensitivity. Overall, this lead to an increase in the F-statistic after GATK recalibration with the true known sites and a decrease in the F-statistic after GATK recalibration with initial calls as the known sites, with no change in F-statistic after KBBQ recalibration. The ROC curve on the QUAL field of the called variants does

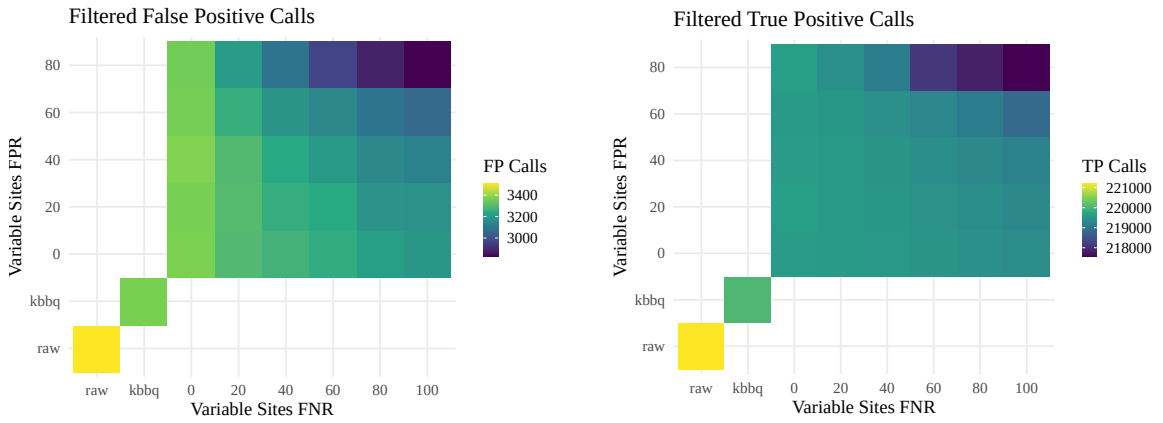


Figure 4.10: Filtered Positive Calls | The output number of false positive and true positive calls for each dataset. The false negative and false positive rate of the set of variable sites used to calibrate the input reads is shown on the x and y axis. The color of each cell represents the number of false or true positive SNPs after filtering. As base quality scores become more calibrated, the number of positive calls increases. Note the range of positive calls has increased in comparison to each unfiltered dataset. Before filtering, the difference in the number of true positives between datasets was at most 240; after filtering it rises to 2044.

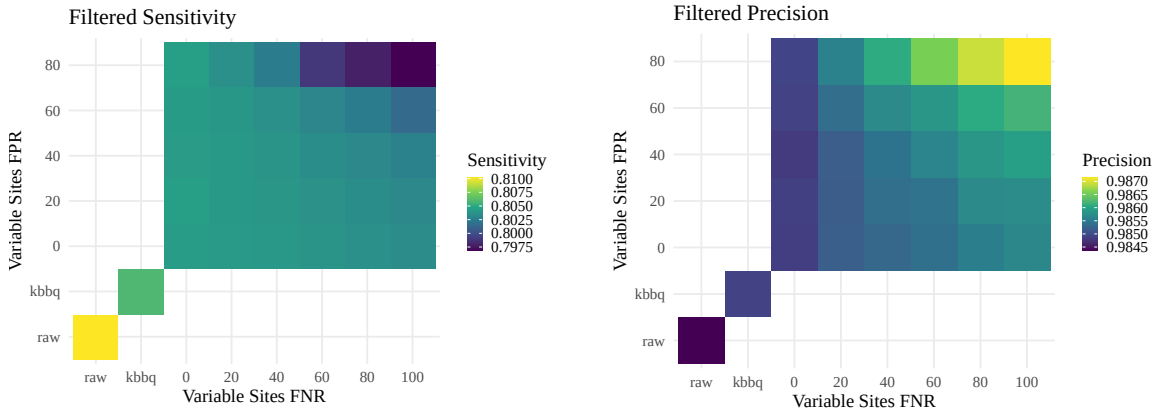


Figure 4.11: Filtered Call Sensitivity and Precision | The sensitivity (left) and precision (right) of the variant caller on each dataset after filtering. As the base quality score calibration increases, the sensitivity of the caller increases and the precision decreases, as seen in the unfiltered data. However, the difference between the largest and smallest precision is slightly smaller than in the unfiltered data, and the difference between the largest and smallest recall is much larger.

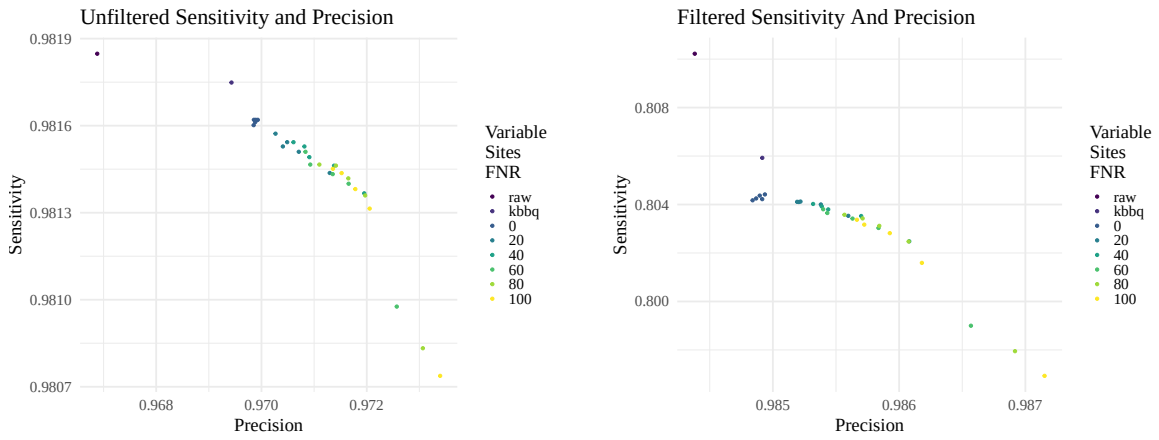


Figure 4.12: Sensitivity and Precision Before and After Filtering | The sensitivity and precision of the variant caller on each dataset with no filtering on the left, and after filters are applied on the right. In the unfiltered calls, as the base quality score calibration of the input reads decrease, the sensitivity of the caller decreases but its precision increases. Once the calls are filtered, the same pattern is observed. However, after filtering the range of precision between datasets is much smaller than before filtering. Conversely, the range of sensitivity increases.

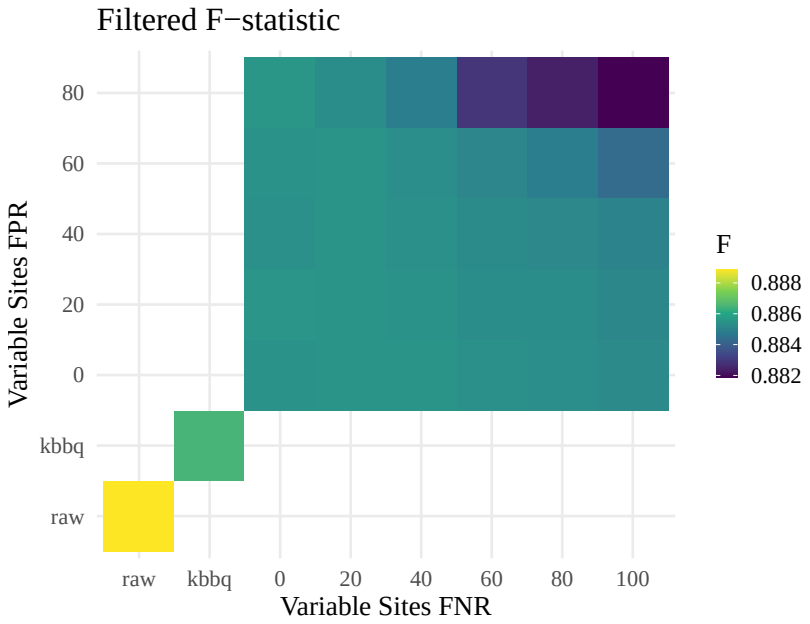


Figure 4.13: Filtered F-statistic | The F-statistic of each dataset. As the calibration of the data improves, so does the F-statistic. This is in contrast to the unfiltered data, in which the worst-calibrated data has the best F-statistic. This difference is driven by an increase in relative precision of the best-calibrated data and a decrease in relative sensitivity of the poorly-calibrated data.

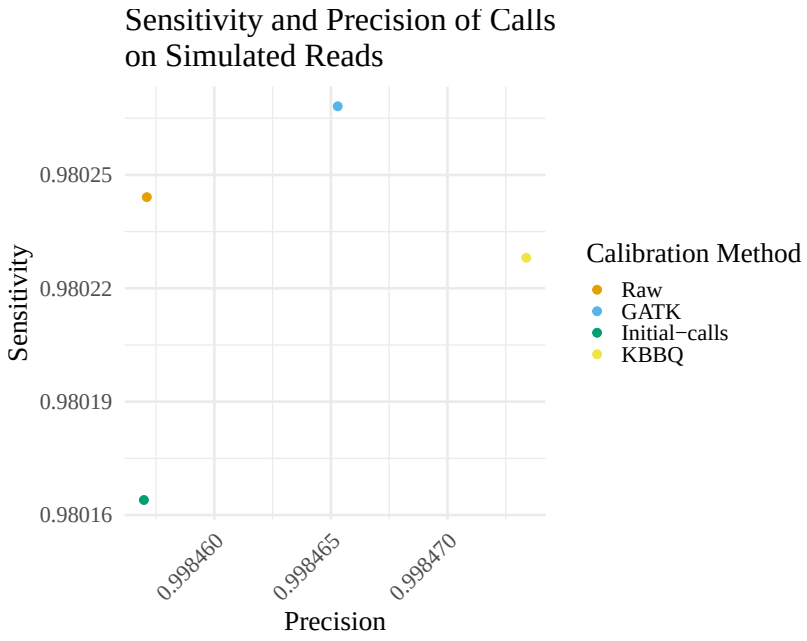


Figure 4.14: Sensitivity and Precision of Calls on Simulated Reads | The sensitivity and precision of calls made using the raw data and after recalibration with 3 methods. See Table 4.4 for values plotted.

not clearly show one calibration method as better than the others, and each one has the best trade-off between false positive and true positive rate at different points along the ROC curve except the Initial-calls calibration, which is closer to the bottom right of the plot at all times. There was no discernable difference between datasets when the GQ score was used to create the ROC curve, so it is not shown.

4.3.1 Variants Detected in E. melliodora

In order to determine the effect of reference-free recalibration on variant calls, I recalibrated the reads with the reference-free base quality score recalibrator KBBQ (see Chapter 3 for more information on KBBQ). I then called variants and filtered the same way they were called and filtered in Orr et al. (2020). The number of variants called at each step in the filtering process is shown in Table 4.5. The number of variants called at each step using uncalibrated reads is shown in Table 4.6. After

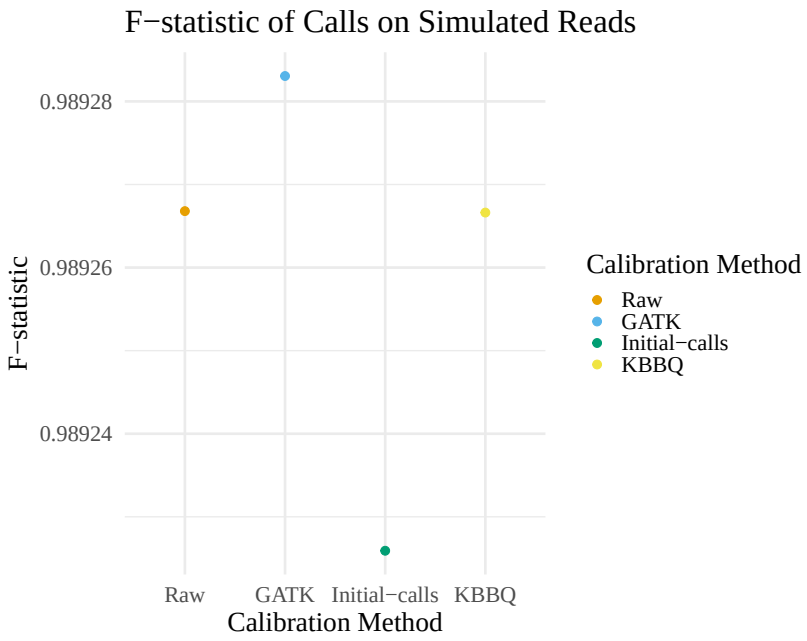


Figure 4.15: F-statistic of Calls on Simulated Reads | The F-statistic of calls made using the raw data and after recalibration with 3 methods. See Table 4.4 for values plotted.

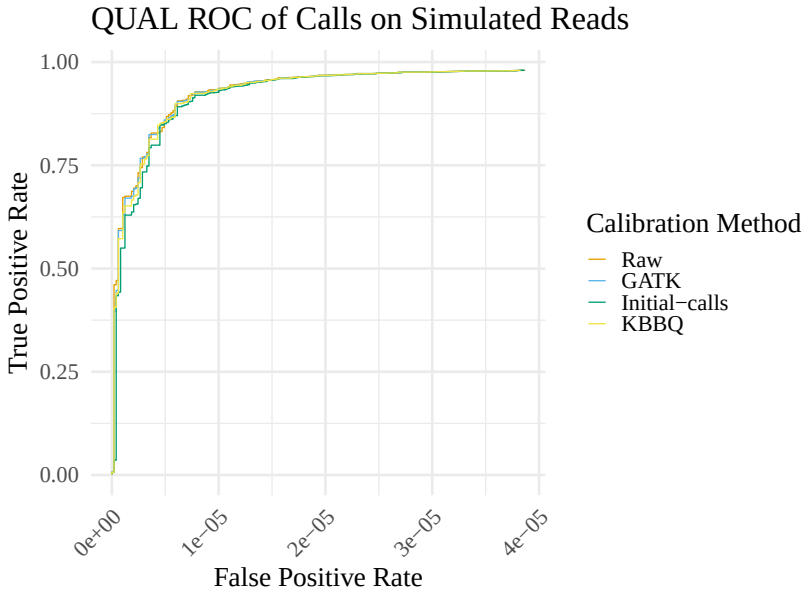


Figure 4.16: ROC of Calls on Simulated Reads | ROC curve on the QUAL score of the variants called by HaplotypeCaller on the uncalibrated simulated reads and after recalibration with 3 methods.

Calibration Method	TP	FP	Sensitivity	Precision	F-statistic
Raw	122308	189	0.98024	0.99846	0.98927
GATK	122311	188	0.98027	0.99847	0.98928
Initial-calls	122298	189	0.98016	0.99846	0.98923
KBBQ	122306	187	0.98023	0.99847	0.98927

Table 4.4: Variant Caller Performance on Simulated Data | A summary of the performance of HaplotypeCaller on simulated data calibrated different ways. TP is the number of true positive calls and FP the number of false positive calls. Raw is the quality scores assigned by the simulator, GATK is recalibration using the known heterozygous sites, Initial-calls is recalibration after calling an initial set of variants, and KBBQ is recalibration with the reference-free recalibrator KBBQ. See Figure 4.14 for a visual comparison.

filtering, 106 variants were detected; 34 of these were also detected in the confident set of variants previously identified. This previous set had a total of 90 variants. Note that the number of detected sites listed in Table 4.5 are not all indicative of mutation, as the number includes heterozygous sites until the last filtering step.

4.4 Discussion

These results show that GATK BaseRecalibrator is particularly vulnerable to false negatives (Figure 4.2) in the database of variable sites, but is robust to false positives if the false negative rate is near 0 (Figure 4.1). At the same time, when the false negative rate is not near 0, false positives will start to impact the calibration quality (Figure 4.3). Thus, when this database is unavailable and construction is required, it may be better to be liberal in deciding which sites may be variable to reduce the false negative rate as much as possible. This is in contrast to the GATK recommendation to use only the most confident sites when a database is unavailable.

The only rate that shows significant deviation from a 0% false positive rate in the

Description	Num. Variants	Previously Identified Variants
Appears in first 11 scaffolds	9838408	88
Total depth <= 500	9190223	87
ExcessHet <= 40	4932628	86
Not within 50bp of an indel	3175128	69
Biallelic SNPs	1913594	67
Outside repeat regions	857810	46
All 3 replicates match	63793	35
Only variable sites	106	34

Table 4.5: Number of Detected Variants After KBBQ Recalibration | The number of variants detected at each step in the filtering pipeline. Each row includes a description of each filter and the number of variants remaining after the filter has been applied. Previously identified variants are the number of variants that overlap the confident set described in Orr et al. (2020). This set contains a total of 90 variants.

false-positive-only data is the 100% false positive rate. Though a 100% false positive rate with a 0% false negative rate implies every site should be considered variable and ignored, the model curiously still has a source of errors it uses to recalibrate. Upon further investigation, this effect is driven by reads with alignments that begin with an insertion, as these inserted bases are not ignored by BaseRecalibrator when the first position of the site that should be ignored is equal to the first aligned position in the read. Thus, this line is a technical artifact. As long as there are enough bases available to analyze, the false positive rate doesn't significantly affect the performance of BaseRecalibrator at a false negative rate of 0. The calibrations of other datasets with a false positive rate of 100% would also be affected by this artifact; however, in a real dataset it's unlikely to ever achieve a false positive rate of 100%, so this artifact

Description	Num. Variants	Previously Identified Variants
Appears in first 11 scaffolds	9823414	88
Total depth <= 500	9177547	87
ExcessHet <= 40	4924104	86
Not within 50bp of an indel	3169769	69
Biallelic SNPs	1911802	67
Outside repeat regions	858383	46
All 3 replicates match	63687	35
Only variable sites	88	34

Table 4.6: Number of Detected Variants Using Uncalibrated Reads | The number of variants detected at each step in the filtering pipeline using uncalibrated reads. See Table 4.5.

is unlikely to significantly affect real data.

Ultimately, if the false negative and false positive rates of the database of variable sites is high, GATK's procedure for BQSR *can* cause miscalibration of the data worse than using raw quality scores, though this can only happen with very large error rates. In this dataset, the raw data has a RMSE of about 4, which is similar to a simulated dataset with a false negative rate between 40-60% and a false positive rate between 60-80%. In a real situation it's unlikely that error rates like this will occur, so BQSR will not severely destroy the input data. However, at even modest false negative and false positive error rates BQSR can cause undesirable miscalibration that could feasibly impact variants called from the data. Interestingly, at all error rates the calibration of quality scores below 25 is almost always correct or 1-off the true value. It seems that errors in the database of variable sites have a larger effect

on higher quality predictions than lower ones.

Interestingly, overconfidence is not displayed in the reads recalibrated using the artificially constructed database of variable sites. This is likely because of how these variant sets were constructed; non-variable sites were selected independently and at random to be added to the set of purported variable sites. Thus a site containing an error and a site not containing an error are selected proportionally. In contrast, on a real dataset a variant calling algorithm would not make false positive errors independently; it is much more likely to classify a site as variable that has a sequencing error than to classify a site as variable that has no errors. That is: in this simulation, $P(\text{classified positive} \mid \text{actual negative})$ is independent of $P(\text{sequencing error})$, but on an empirical dataset $P(\text{classified positive} \mid \text{actual negative})$ is likely not. Thus in reality false positives probably cause overconfidence in a similar manner that false negatives cause underconfidence.

Overall, these results show that as the quality of the calibration increases, the quality of the variant calls also slightly increases. The difference is very small, but better calibration produces more positive calls, with more true positive calls than any other recalibrated dataset. And though I only filtered the data with two fairly lenient filters, the best calibrations still produced the most true positive calls of any dataset except the calls made from data with the raw quality scores, which are not as well-calibrated (see Chapter 3) but produce more positive calls.

Interestingly, filtering seemed to amplify the difference between well-calibrated and poorly-calibrated data, increasing the disparity in the number of true positive variants detected. At the same time, filtering reduced the disparity in the number of false positive variants identified.

This observation makes filtering very important for variant calling. Variant callers generally prefer to include a putative variant rather than exclude it and miss a truly

variant site. Thus, they prefer sensitivity over precision. The consequences of this preference can be seen in Figure 4.12, where filtering reduced the sensitivity of every set of calls but greatly improved precision and decreased the precision disparity between datasets. Filtering raw calls is paramount to getting a variant set with acceptable levels of sensitivity and precision. Ultimately, the impact of base quality score recalibration is not as large as the filters one chooses to use. In this case, better calibration results in an increased precision of .007 before filtering; filtering itself increased precision by about .014, while reducing sensitivity by approximately .18. So the effect of calibration on sensitivity and precision is small in comparison to the effect of filtering. At the same time, important variants that matter for the purposes of the study being conducted could be missed, so this difference in calibration could be relevant.

Unfortunately, this means that filtering plays a big role in the success of a variant calling pipeline. While there is no definitive best way to filter variants in all cases, Figure 4.9 shows that differences in quality score calibration can affect the annotations attached to each call and make the same filters more or less effective, depending on the threshold used. While this is interesting, it doesn't necessarily change how filtering should be done or suggest an optimal filtering strategy.

One limitation of this study is that it uses only a sparse range of false positives and false negatives in the database of variable sites. Especially since the effect of calibration on the output variant calls is small, a different dataset may yield different results. As seen in Chapter 3, the calibration of the raw reads in this dataset is not particularly bad, though it is worse than all other datasets. Replication with other datasets and other variant callers is necessary to further elucidate the role of base quality score calibration on variant caller performance.

Furthermore, the CHM1-CHM13 data is from a human-derived cell line, and pro-

tocols for DNA extraction and sequencing are fairly well-established. Additionally, the original base calling algorithm is likely tuned to work well with human-like data. All together, this means that while the effect of recalibration was not particularly pronounced in this dataset, base quality score recalibration may have a higher impact on data that is messier, contains more errors induced in sample preparation, or results from failed sequencing runs. In light of this, it seems that using the default-assigned base quality scores will offer superior performance unless one has a reason to suspect the data is of poor quality.

The simulated reads similarly show that the effect of base quality score recalibration is fairly small. However, the simulator also assigned fairly well-calibrated scores to begin with, so there is not much room for improvement in those scores and many opportunities to make calibration worse. As shown in the previous chapter, using initial calls for recalibration does degrade the overall quality of the calibration. This is consistent with these results showing that HaplotypeCaller performs worse on reads recalibrated using GATK with a set of initial calls. However, when the set of known sites is the true set of variable sites in the data, GATK recalibration does slightly improve the precision and sensitivity of HaplotypeCaller. At the same time, reference-free recalibration with KBBQ increases precision but slightly decreases sensitivity.

Overall, these simulations show that GATK recalibration slightly increases HaplotypeCaller performance if the set of variable sites is complete and has no false positives, even if the starting data is fairly well-calibrated. However, if the set of variable sites is not complete, GATK recalibration will cause HaplotypeCaller performance to suffer slightly. Thus, if there is a risk that the set of variable sites is inaccurate, GATK recalibration should not be used. At the same time, raw quality scores will provide good performance as long as they are fairly well-calibrated to start. These simulations

also show that even with well-calibrated input data, the reference-free recalibration method KBBQ doesn't significantly degrade the performance of HaplotypeCaller.

4.4.1 KBBQ Recalibration Yielded More E. melliodora Calls

While the simulated datasets revealed a complex interaction between the number of detected variants, sensitivity, precision, and filtering, it appears KBBQ recalibration improved the ability to successfully call variants in a non-model organism. In Chapter 3, I showed that using GATK's recommended approach of calling confident variants and using those as the set of variable sites to use in recalibration results in poor base quality score calibration. I also showed that using KBBQ improved calibration to be comparable to GATK recalibrations with perfect information. The calls on the simulated datasets shown here also imply that KBBQ recalibrated reads behave similarly to or better than reads recalibrated with GATK using perfect information.

Using the same calling and filtering parameters as those used in Orr et al. (2020) yielded more variants at the end of filtering with 106 instead of 99. This is consistent with the results from the simulated datasets, where better-calibrated data yields more variant calls. It is also consistent with Ni and Stoneking (2016), who find that better calibration increases sensitivity. Indeed, this is the case even before other filters are applied, with 9,838,408 variants detected versus 9,679,544 detected in the prior study. This is also slightly more than the 9,823,414 detected using uncalibrated reads before filtering and 88 detected after.

Additionally, the number of variants after filtering that appear in the confident set identified in the previous study increased from 30 to 34. This is identical to the number detected using the raw reads. Thus, using KBBQ to recalibrate the reads before calling seems to have increased the sensitivity of the variant calling pipeline compared to GATK's BaseRecalibrator. Note that the confident call set in Orr et

al. (2020) has a fairly large estimated false negative rate of approximately 30% and many reads did not map to the genome used; thus, it's likely there are many additional true variants in the data that were not previously identified but are nevertheless true variants.

4.5 Conclusion

The simulated data here shows that quality scores do have a small but important effect on the quality of the called variants. This manifests itself in not only the difference in the number of true positive calls and the number of false positive calls, but also in the QUAL and GQ annotations. The simulated data suggest the size of the effect is small, but on an empirical dataset increased calibration yielded a substantial number of additional variants. While effective recalibration will certainly benefit variant calling, these results show BaseRecalibrator can decrease the quality of the called variants if done improperly. If very high sensitivity is critical to the analysis being performed, care must be taken to ensure that base quality scores are properly calibrated.

REFERENCES

Auwera, Geraldine A., Mauricio O. Carneiro, Christopher Hartl, Ryan Poplin, Guillermo del Angel, Ami Levy-Moonshine, Tadeusz Jordan, et al. 2013. "From FastQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline." *Current Protocols in Bioinformatics* 43 (1). https://doi.org/10.1002/0471250953.bi1110s43.

Cleary, John G., Ross Braithwaite, Kurt Gaastra, Brian S. Hilbush, Stuart Inglis, Sean A. Irvine, Alan Jackson, et al. 2015. "Comparing Variant Call Files for Performance Benchmarking of Next-Generation Sequencing Variant Calling Pipelines." *bioRxiv,* 023754. https://doi.org/10.1101/023754.

Ewing, B., L. Hillier, M. C. Wendl, and P. Green. 1998. "Base-calling of automated sequencer traces using phred. I. Accuracy assessment." *Genome Research* 8 (3): 175–185. https://doi.org/10.1101/gr.8.3.175.

Ewing, Brent, and Phil Green. 1998. "Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities." *Genome Research* 8 (3): 186–194. https://doi.org/10.1101/gr.8.3.186.

Fox, Edward J., Kate S. Reid-Bayliss, Mary J. Emond, and Lawrence A. Loeb. 2014. "Accuracy of Next Generation Sequencing Platforms." *Next generation, sequencing & applications* 1. https://doi.org/10.4172/jngsa.1000106.

Garnier, Simon. 2018. *viridisLite: Default Color Maps from 'matplotlib' (Lite Version).* R package version 0.3.0. https://CRAN.R-project.org/package=viridisLite.

Garrison, Erik, and Gabor Marth. 2012. "Haplotype-based variant detection from short-read sequencing." *arXiv:1207.3907 [q-bio]*, arXiv: 1207.3907. http://arxiv.org/abs/1207.3907.

Huang, Weichun, Leping Li, Jason R. Myers, and Gabor T. Marth. 2012. "ART: a next-generation sequencing read simulator." *Bioinformatics* 28 (4): 593–594. https://doi.org/10.1093/bioinformatics/btr708.

Külheim, Carsten, Suat Hui Yeoh, Jens Maintz, William J. Foley, and Gavin F. Moran. 2009. "Comparative SNP diversity among four Eucalyptus species for genes from secondary metabolite biosynthetic pathways." *BMC Genomics* 10 (1): 1–11. https://doi.org/10.1186/1471-2164-10-452.

Li, Heng. 2011a. "A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data." *Bioinformatics* 27 (21): 2987–2993. https://doi.org/10.1093/bioinformatics/btr509.

Li, Heng. 2011b. "Improving SNP discovery by base alignment quality." *Bioinformatics* 27 (8): 1157–1158. https://doi.org/10.1093/bioinformatics/btr076.

Li, Heng. 2014. "Toward better understanding of artifacts in variant calling from high-coverage samples." *Bioinformatics* 30 (20): 2843–2851. https://doi.org/10.1093/bioinformatics/btu356.

Li, Heng, Jonathan M. Bloom, Yossi Farjoun, Mark Fleharty, Laura Gauthier, Benjamin Neale, and Daniel MacArthur. 2018. "A synthetic-diploid benchmark for accurate variant-calling evaluation." *Nature Methods* 15 (8): 595–597. https://doi.org/10.1038/s41592-018-0054-7.

Li, Heng, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. 2009. "The Sequence Alignment/Map format and SAMtools." *Bioinformatics (Oxford, England)* 25 (16): 2078–2079. https://doi.org/10.1093/bioinformatics/btp352.

Malysa, Greg, Mikel Hernaez, Idoia Ochoa, Milind Rao, Karthik Ganesan, and Tsachy Weissman. 2015. "QVZ: lossy compression of quality values." *Bioinformatics* 31 (19): 3122–3129. https://doi.org/10.1093/bioinformatics/btv330.

Ni, Shengyu, and Mark Stoneking. 2016. "Improvement in detection of minor alleles in next generation sequencing by base quality recalibration." *BMC genomics* 17:139. https://doi.org/10.1186/s12864-016-2463-2.

Orr, Adam J., Amanda Padovan, David Kainer, Carsten Külheim, Lindell Bromham, Carlos Bustos-Segura, William Foley, et al. 2020. "A phylogenomic approach reveals a low somatic mutation rate in a long-lived plant." *Proceedings of the Royal Society B: Biological Sciences* 287 (1922): 20192364. https://doi.org/10.1098/rspb.2019.2364.

Pfeifer, S. P. 2017. "From next-generation resequencing reads to a high-quality variant data set." *Heredity* 118 (2): 111–124. https://doi.org/10.1038/hdy.2016.102.

Poplin, Ryan, Valentin Ruano-Rubio, Mark A. DePristo, Tim J. Fennell, Mauricio O. Carneiro, Geraldine A. Van der Auwera, David E. Kling, et al. 2018. "Scaling accurate genetic variant discovery to tens of thousands of samples." *bioRxiv,* https://doi.org/10.1101/201178.

Quinlan, Aaron R., and Ira M. Hall. 2010. "BEDTools: a flexible suite of utilities for comparing genomic features." *Bioinformatics* 26 (6): 841–842. https://doi.org/10.1093/bioinformatics/btq033.

R Core Team. 2020. *R: A Language and Environment for Statistical Computing.* Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Ramu, Avinash, Michiel J Noordam, Rachel S Schwartz, Arthur Wuster, Matthew E Hurles, Reed A Cartwright, and Donald F Conrad. 2013. "DeNovoGear: de novo indel and point mutation discovery and phasing." *Nature Methods* 10 (10): 985–987. https://doi.org/10.1038/nmeth.2611.

Reducing Whole-Genome Data Storage Footprint. 2014. White paper 970-2012-013. Illumina, Inc.

Sedlazeck, Fritz J., Philipp Rescheneder, and Arndt von Haeseler. 2013. "NextGenMap: fast and accurate read mapping in highly polymorphic genomes." *Bioinformatics* 29 (21): 2790–2791. https://doi.org/10.1093/bioinformatics/btt468.

Shibuya, Yoshihiro, and Matteo Comin. 2019. "Better quality score compression through sequence-based quality smoothing." *BMC Bioinformatics* 20 (9): 302. https://doi.org/10.1186/s12859-019-2883-5.

Wickham, Hadley. 2016. *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York. https://ggplot2.tidyverse.org.

Wu, Steven H, Rachel S Schwartz, David J Winter, Donald F Conrad, and Reed A Cartwright. 2017. "Estimating error models for whole genome sequencing using mixtures of Dirichlet-multinomial distributions." *Bioinformatics* 33 (15): 2322–2329. https://doi.org/10.1093/bioinformatics/btx133.

Yu, Y. William, Deniz Yorukoglu, Jian Peng, and Bonnie Berger. 2015. "Quality score compression improves genotyping accuracy." *Nature Biotechnology* 33 (3): 240–243. https://doi.org/10.1038/nbt.3170.

Yue, Jia-Xing, and Gianni Liti. 2019. "simuG: a general-purpose genome simulator." *Bioinformatics* 35 (21): 4442–4444. https://doi.org/10.1093/bioinformatics/btz424.

Chapter 5

CONCLUSION

While noisy sequencing data can be difficult to work with, with care it *is* possible to generate high-quality calls. As both sequencing technology and software tools improve, the amount of useful information that can be extracted from sequencing experiments will continue to grow, and the costs of performing those experiments will continue to shrink. This will enable the generation of an unprecedented amount of data, particularly in organisms that have been historically neglected. This necessitates the continued development of methods that function even without *a priori* information.

All together, the experiments included here show that while calling variants requires effort when reference genomes and other information is not available, it is not impossible. Most methods designed to work with reference-aligned data can work fairly well if a reference from a close relative can be adapted to the data, as shown in Chapter 2. Furthermore, Chapter 3 shows that base quality scores can be successfully calibrated without any auxiliary information. And Chapter 4 shows that this improvement in base quality score calibration can improve variant calling sensitivity and reduce the number of false positive calls.

In these investigations, I have highlighted and developed several methods for detecting mutations in non-model organisms. While the data used here is limited to Illumina short read sequencing, the approaches described are adaptable for even the next generation of sequencing platforms. The principles discussed and their application may prove even more useful on these platforms, as their primary drawback is high error rates. Even now, computational methods are able to overcome the error

rates on these platforms and make them suitable for some applications, like assembly.

Now is an incredible time to be interested in sequencing and sequencing technology. Illumina sequencing and associated methods are relatively mature, but interesting new methods and models are still expanding the limits of the technology and achieving impressive results despite its flaws. On the horizon, long read sequencing is becoming ever more affordable, accurate, and fast, promising new avenues for investigating the natural world along with new challenges. But no matter what the future holds, one thing is clear: software and statistical methods for analyzing sequencing data will continue to be necessary to get the most out of sequencing technology and transform raw data into insights.

REFERENCES

Ally, Dilara, Kermit Ritland, and Sarah P. Otto. 2010. "Aging in a Long-Lived Clonal Tree." *PLoS Biology* 8 (8): e1000454. https://doi.org/10.1371/journal.pbio.1000454.

Audano, Peter A, Shashidhar Ravishankar, and Fredrik O Vannberg. 2018. "Mapping-free variant calling using haplotype reconstruction from k-mer frequencies." *Bioinformatics* 34 (10): 1659–1665. https://doi.org/10.1093/bioinformatics/btx753.

Auwera, Geraldine A., Mauricio O. Carneiro, Christopher Hartl, Ryan Poplin, Guillermo del Angel, Ami Levy-Moonshine, Tadeusz Jordan, et al. 2013. "From FastQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline." *Current Protocols in Bioinformatics* 43 (1). https://doi.org/10.1002/0471250953.bi1110s43.

Bartholomé, Jérôme, Eric Mandrou, André Mabiala, Jerry Jenkins, Ibouniyamine Nabihoudine, Christophe Klopp, Jeremy Schmutz, Christophe Plomion, and Jean-Marc Gion. 2015. "High-resolution genetic maps of *Eucalyptus* improve *Eucalyptus grandis* genome assembly." *New Phytologist* 206 (4): 1283–1296. https://doi.org/10.1111/nph.13150.

Bobiwash, K, S T Schultz, and D J Schoen. 2013. "Somatic deleterious mutation rate in a woody plant: estimation from phenotypic data." *Heredity* 111 (4): 338–344. https://doi.org/10.1038/hdy.2013.57.

Bobo, Dean, Mikhail Lipatov, Juan L. Rodriguez-Flores, Adam Auton, and Brenna M. Henn. 2016. *False Negatives Are a Significant Feature of Next Generation Sequencing Callsets.* Preprint. Bioinformatics. https://doi.org/10.1101/066043.

Boland, D. J, and M. W McDonald. 2006. *Forest trees of Australia.* OCLC: 181586979. Collingwood, Victoria: CSIRO Pub. http://public.eblib.com/choice/publicfullrecord.aspx?p=282659.

Boyle, Alan P., Sean Davis, Hennady P. Shulha, Paul Meltzer, Elliott H. Margulies, Zhiping Weng, Terrence S. Furey, and Gregory E. Crawford. 2008. "High-resolution mapping and characterization of open chromatin across the genome." *Cell* 132 (2): 311–322. https://doi.org/10.1016/j.cell.2007.12.014.

Branton, Daniel, David W Deamer, Andre Marziali, Hagan Bayley, Steven A Benner, Thomas Butler, Massimiliano Di Ventra, et al. 2008. "The potential and challenges of nanopore sequencing." *Nature Biotechnology* 26 (10): 1146–1153. https://doi.org/10.1038/nbt.1495.

Buenrostro, Jason, Beijing Wu, Howard Chang, and William Greenleaf. 2015. "ATAC-seq: A Method for Assaying Chromatin Accessibility Genome-Wide." *Current protocols in molecular biology / edited by Frederick M. Ausubel ... [et al.]* 109:21.29.1–21.29.9. https://doi.org/10.1002/0471142727.mb2129s109.

Burian, Agata, Pierre Barbier de Reuille, and Cris Kuhlemeier. 2016. "Patterns of Stem Cell Divisions Contribute to Plant Longevity." *Current Biology* 26 (11): 1385–1394. https://doi.org/10.1016/j.cub.2016.03.067.

Buss, L. W. 1983. "Evolution, development, and the units of selection." *Proceedings of the National Academy of Sciences* 80 (5): 1387–1391. https://doi.org/10.1073/pnas.80.5.1387.

Cabanski, Christopher R., Keary Cavin, Chris Bizon, Matthew D. Wilkerson, Joel S. Parker, Kirk C. Wilhelmsen, Charles M. Perou, J. S. Marron, and D. Neil Hayes. 2012. "ReQON: a Bioconductor package for recalibrating quality scores from next-generation sequencing data." *BMC bioinformatics* 13:221. https://doi.org/10.1186/1471-2105-13-221.

Callahan, Benjamin J., Paul J. McMurdie, Michael J. Rosen, Andrew W. Han, Amy Jo A. Johnson, and Susan P. Holmes. 2016. "DADA2: High-resolution sample inference from Illumina amplicon data." *Nature Methods* 13 (7): 581–583. https://doi.org/10.1038/nmeth.3869.

Chung, Jade C. S., and Swaine L. Chen. 2017. "Lacer: accurate base quality score recalibration for improving variant calling from next-generation sequencing data in any organism." *bioRxiv,* 130732. https://doi.org/10.1101/130732.

Cibulskis, Kristian, Michael S. Lawrence, Scott L. Carter, Andrey Sivachenko, David Jaffe, Carrie Sougnez, Stacey Gabriel, Matthew Meyerson, Eric S. Lander, and Gad Getz. 2013. "Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples." *Nature Biotechnology* 31 (3): 213–219. https://doi.org/10.1038/nbt.2514.

Cleary, John G., Ross Braithwaite, Kurt Gaastra, Brian S. Hilbush, Stuart Inglis, Sean A. Irvine, Alan Jackson, et al. 2015. "Comparing Variant Call Files for Performance Benchmarking of Next-Generation Sequencing Variant Calling Pipelines." *bioRxiv,* 023754. https://doi.org/10.1101/023754.

Cock, Peter J. A., Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. 2010. "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants." *Nucleic Acids Research* 38 (6): 1767–1771. https://doi.org/10.1093/nar/gkp1137.

Edwards, Penelope B., W. J. Wanjura, W. V. Brown, and John M. Dearn. 1990. "Mosaic resistance in plants." *Nature* 347 (6292): 434–434. https://doi.org/10.1038/347434a0.

Eid, John, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, et al. 2009. "Real-Time DNA Sequencing from Single Polymerase Molecules." *Science* 323 (5910): 133–138. https://doi.org/10.1126/science.1162986.

Ewing, B., L. Hillier, M. C. Wendl, and P. Green. 1998. "Base-calling of automated sequencer traces using phred. I. Accuracy assessment." *Genome Research* 8 (3): 175–185. https://doi.org/10.1101/gr.8.3.175.

Ewing, Brent, and Phil Green. 1998. "Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities." *Genome Research* 8 (3): 186–194. https://doi.org/10.1101/gr.8.3.186.

Fonseca, Rute R. da, Anders Albrechtsen, Gonçalo Espregueira Themudo, Jazmín Ramos-Madrigal, Jonas Andreas Sibbesen, Lasse Maretty, M. Lisandra Zepeda-Mendoza, Paula F. Campos, Rasmus Heller, and Ricardo J. Pereira. 2016. "Next-generation biology: Sequencing and data analysis approaches for non-model organisms." *Marine Genomics* 30:3–13. https://doi.org/10.1016/j.margen.2016.04.012.

Fox, Edward J., Kate S. Reid-Bayliss, Mary J. Emond, and Lawrence A. Loeb. 2014. "Accuracy of Next Generation Sequencing Platforms." *Next generation, sequencing & applications* 1. https://doi.org/10.4172/jngsa.1000106.

Frommer, M., L. E. McDonald, D. S. Millar, C. M. Collis, F. Watt, G. W. Grigg, P. L. Molloy, and C. L. Paul. 1992. "A genomic sequencing protocol that yields a positive display of 5-methylcytosine residues in individual DNA strands." *Proceedings of the National Academy of Sciences* 89 (5): 1827–1831. https://doi.org/10.1073/pnas.89.5.1827.

Gao, Ziyue, Minyoung J. Wyman, Guy Sella, and Molly Przeworski. 2016. "Interpreting the Dependence of Mutation Rates on Age and Time." *PLOS Biology* 14 (1): e1002355. https://doi.org/10.1371/journal.pbio.1002355.

Garnier, Simon. 2018. *viridisLite: Default Color Maps from 'matplotlib' (Lite Version)*. R package version 0.3.0. https://CRAN.R-project.org/package=viridisLite.

Garrison, Erik, and Gabor Marth. 2012. "Haplotype-based variant detection from short-read sequencing." *arXiv:1207.3907 [q-bio]*, arXiv: 1207.3907. http://arxiv.org/abs/1207.3907.

Gauthier, Jérémy, Charlotte Mouden, Tomasz Suchan, Nadir Alvarez, Nils Arrigo, Chloé Riou, Claire Lemaitre, and Pierre Peterlongo. 2020. "DiscoSnp-RAD: de novo detection of small variants for RAD-Seq population genomics." *PeerJ* 8:e9291. https://doi.org/10.7717/peerj.9291.

Gill, D E, L Chao, S L Perkins, and J B Wolf. 1995. "Genetic Mosaicism in Plants and Clonal Animals." *Annual Review of Ecology and Systematics* 26 (1): 423–444. https://doi.org/10.1146/annurev.es.26.110195.002231.

Grattapaglia, Dario, René E. Vaillancourt, Merv Shepherd, Bala R. Thumma, William Foley, Carsten Külheim, Brad M. Potts, and Alexander A. Myburg. 2012. "Progress in Myrtaceae genetics and genomics: Eucalyptus as the pivotal genus." *Tree Genetics & Genomes* 8 (3): 463–508. https://doi.org/10.1007/s11295-012-0491-x.

Hanlon, Vincent C. T., Sarah P. Otto, and Sally N. Aitken. 2019. "Somatic mutations substantially increase the per-generation mutation rate in the conifer *Picea sitchensis*." *Evolution Letters* 3 (4): 348–358. https://doi.org/10.1002/evl3.121.

Huang, Weichun, Leping Li, Jason R. Myers, and Gabor T. Marth. 2012. "ART: a next-generation sequencing read simulator." *Bioinformatics* 28 (4): 593–594. https://doi.org/10.1093/bioinformatics/btr708.

Iqbal, Zamin, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. 2012. "De novo assembly and genotyping of variants using colored de Bruijn graphs." *Nature genetics* 44 (2): 226–232. https://doi.org/10.1038/ng.1028.

Johnson, David S., Ali Mortazavi, Richard M. Myers, and Barbara Wold. 2007. "Genome-Wide Mapping of in Vivo Protein-DNA Interactions." *Science* 316 (5830): 1497–1502. https://doi.org/10.1126/science.1141319.

Kainer, David, Amanda Padovan, Joerg Degenhardt, Sandra Krause, Prodyut Mondal, William J. Foley, and Carsten Külheim. 2019. "High marker density GWAS provides novel insights into the genomic architecture of terpene oil yield in *Eucalyptus*." *New Phytologist* 223 (3): 1489–1504. https://doi.org/10.1111/nph.15887.

Kao, Wei-Chun, and Yun S. Song. 2011. "naiveBayesCall: an efficient model-based base-calling algorithm for high-throughput sequencing." *Journal of Computational Biology: A Journal of Computational Molecular Cell Biology* 18 (3): 365–377. https://doi.org/10.1089/cmb.2010.0247.

Keightley, Peter D., Rob W. Ness, Daniel L. Halligan, and Penelope R. Haddrill. 2014. "Estimation of the Spontaneous Mutation Rate per Nucleotide Site in a *Drosophila melanogaster* Full-Sib Family." *Genetics* 196 (1): 313–320. https://doi.org/10.1534/genetics.113.158758.

Khoury, Colin, Brigitte Laliberté, and Luigi Guarino. 2010. "Trends in ex situ conservation of plant genetic resources: a review of global crop and regional conservation strategies." *Genetic Resources and Crop Evolution* 57 (4): 625–639. https://doi.org/10.1007/s10722-010-9534-z.

Klekowski, Edward J, Jorge E Corredor, Julio M Morell, and Carlos A Del Castillo. 1994. "Petroleum pollution and mutation in mangroves." *Marine Pollution Bulletin* 28 (3): 166–169. https://doi.org/10.1016/0025-326X(94)90393-X.

Klekowski, Edward J. 1988. *Mutation, Developmental Selection, and Plant Evolution.* Columbia University Press. https://doi.org/10.7312/klek92068.

Klekowski, Edward J., and Paul J. Godfrey. 1989. "Ageing and mutation in plants." *Nature* 340 (6232): 389–391. https://doi.org/10.1038/340389a0.

Kodama, Yuichi, on behalf of the International Nucleotide Sequence Database Collaboration, Martin Shumway, on behalf of the International Nucleotide Sequence Database Collaboration, Rasko Leinonen, and on behalf of the International Nucleotide Sequence Database Collaboration. 2012. "The sequence read archive: explosive growth of sequencing data." *Nucleic Acids Research* 40 (D1): D54–D56. https://doi.org/10.1093/nar/gkr854.

Külheim, Carsten, Suat Hui Yeoh, Jens Maintz, William J. Foley, and Gavin F. Moran. 2009. "Comparative SNP diversity among four Eucalyptus species for genes from secondary metabolite biosynthetic pathways." *BMC Genomics* 10 (1): 1–11. https://doi.org/10.1186/1471-2164-10-452.

Lanfear, Robert. 2018. "Do plants have a segregated germline?" *PLOS Biology* 16 (5): e2005439. https://doi.org/10.1371/journal.pbio.2005439.

Lanfear, Robert, Simon Y. W. Ho, T. Jonathan Davies, Angela T. Moles, Lonnie Aarssen, Nathan G. Swenson, Laura Warman, Amy E. Zanne, and Andrew P. Allen. 2013. "Taller plants have lower rates of molecular evolution." *Nature Communications* 4 (1): 1879. https://doi.org/10.1038/ncomms2836.

Leggett, Richard M., and Dan MacLean. 2014. "Reference-free SNP detection: dealing with the data deluge." *BMC Genomics* 15 (4): S10. https://doi.org/10.1186/1471-2164-15-S4-S10.

Lewin, Harris A., Gene E. Robinson, W. John Kress, William J. Baker, Jonathan Coddington, Keith A. Crandall, Richard Durbin, et al. 2018. "Earth BioGenome Project: Sequencing life for the future of life." *Proceedings of the National Academy of Sciences* 115 (17): 4325–4333. https://doi.org/10.1073/pnas.1720115115.

Lewis, Paul O. 2001. "A Likelihood Approach to Estimating Phylogeny from Discrete Morphological Character Data." *Systematic Biology* 50 (6): 913–925. https://doi.org/10.1080/106351501753462876.

Li, Heng. 2011a. "A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data." *Bioinformatics* 27 (21): 2987–2993. https://doi.org/10.1093/bioinformatics/btr509.

Li, Heng. 2011b. "Improving SNP discovery by base alignment quality." *Bioinformatics* 27 (8): 1157–1158. https://doi.org/10.1093/bioinformatics/btr076.

Li, Heng. 2014. "Toward better understanding of artifacts in variant calling from high-coverage samples." *Bioinformatics* 30 (20): 2843–2851. https://doi.org/10.1093/bioinformatics/btu356.

Li, Heng, Jonathan M. Bloom, Yossi Farjoun, Mark Fleharty, Laura Gauthier, Benjamin Neale, and Daniel MacArthur. 2018. "A synthetic-diploid benchmark for accurate variant-calling evaluation." *Nature Methods* 15 (8): 595–597. https://doi.org/10.1038/s41592-018-0054-7.

Li, Heng, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. 2009. "The Sequence Alignment/Map format and SAMtools." *Bioinformatics (Oxford, England)* 25 (16): 2078–2079. https://doi.org/10.1093/bioinformatics/btp352.

Li, R., C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. 2009. "SOAP2: an improved ultrafast tool for short read alignment." *Bioinformatics* 25 (15): 1966–1967. https://doi.org/10.1093/bioinformatics/btp336.

Ma, Xiaotu, Ying Shao, Liqing Tian, Diane A. Flasch, Heather L. Mulder, Michael N. Edmonson, Yu Liu, et al. 2019. "Analysis of error profiles in deep next-generation sequencing data." *Genome Biology* 20 (1): 50. https://doi.org/10.1186/s13059-019-1659-6.

Malysa, Greg, Mikel Hernaez, Idoia Ochoa, Milind Rao, Karthik Ganesan, and Tsachy Weissman. 2015. "QVZ: lossy compression of quality values." *Bioinformatics* 31 (19): 3122–3129. https://doi.org/10.1093/bioinformatics/btv330.

Mantaci, S., A. Restivo, G. Rosone, and M. Sciortino. 2007. "An extension of the Burrows–Wheeler Transform." *Theoretical Computer Science* 387 (3): 298–312. https://doi.org/10.1016/j.tcs.2007.07.014.

Martin, Marcel, Murray Patterson, Shilpa Garg, Sarah O Fischer, Nadia Pisanti, Gunnar W Klau, Alexander Schöenhuth, and Tobias Marschall. 2016. *WhatsHap: fast and accurate read-based phasing.* Preprint. Bioinformatics. https://doi.org/10.1101/085050.

Massingham, Tim, and Nick Goldman. 2012. "All Your Base: a fast and accurate probabilistic approach to base calling." *Genome Biology* 13 (2): R13. https://doi.org/10.1186/gb-2012-13-2-r13.

McKenna, A., M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, et al. 2010. "The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data." *Genome Research* 20 (9): 1297–1303. https://doi.org/10.1101/gr.107524.110.

Meacham, Frazer, Dario Boffelli, Joseph Dhahbi, David IK Martin, Meromit Singer, and Lior Pachter. 2011. "Identification and correction of systematic error in high-throughput sequence data." *BMC Bioinformatics* 12 (1): 451. https://doi.org/10.1186/1471-2105-12-451.

Michel, Albrecht, Renee S. Arias, Brian E. Scheffler, Stephen O. Duke, Michael Netherland, and Franck E. Dayan. 2004. "Somatic mutation-mediated evolution of herbicide resistance in the nonindigenous invasive plant hydrilla (Hydrilla verticillata): HYDRILLA SOMATIC MUTANTS and HERBICIDE RESISTANCE." *Molecular Ecology* 13 (10): 3229–3237. https://doi.org/10.1111/j.1365-294X.2004.02280.x.

Nagalakshmi, Ugrappa, Zhong Wang, Karl Waern, Chong Shou, Debasish Raha, Mark Gerstein, and Michael Snyder. 2008. "The Transcriptional Landscape of the Yeast Genome Defined by RNA Sequencing." *Science (New York, N.Y.)* 320 (5881): 1344–1349. https://doi.org/10.1126/science.1158441.

Nakamura, Kensuke, Taku Oshima, Takuya Morimoto, Shun Ikeda, Hirofumi Yoshikawa, Yuh Shiwa, Shu Ishikawa, et al. 2011. "Sequence-specific error profile of Illumina sequencers." *Nucleic Acids Research* 39 (13): e90. https://doi.org/10.1093/nar/gkr344.

Ness, Rob W., Andrew D. Morgan, Nick Colegrave, and Peter D. Keightley. 2012. "Estimate of the Spontaneous Mutation Rate in *Chlamydomonas reinhardtii*." *Genetics* 192 (4): 1447–1454. https://doi.org/10.1534/genetics.112.145078.

Ni, Shengyu, and Mark Stoneking. 2016. "Improvement in detection of minor alleles in next generation sequencing by base quality recalibration." *BMC genomics* 17:139. https://doi.org/10.1186/s12864-016-2463-2.

Orr, Adam J., Amanda Padovan, David Kainer, Carsten Külheim, Lindell Bromham, Carlos Bustos-Segura, William Foley, et al. 2020. "A phylogenomic approach reveals a low somatic mutation rate in a long-lived plant." *Proceedings of the Royal Society B: Biological Sciences* 287 (1922): 20192364. https://doi.org/10.1098/rspb.2019.2364.

Ossowski, S., K. Schneeberger, J. I. Lucas-Lledo, N. Warthmann, R. M. Clark, R. G. Shaw, D. Weigel, and M. Lynch. 2010. "The Rate and Molecular Spectrum of Spontaneous Mutations in Arabidopsis thaliana." *Science* 327 (5961): 92–94. ht tps://doi.org/10.1126/science.1180677.

Padovan, Amanda, Andras Keszei, William J Foley, and Carsten Külheim. 2013. "Differences in gene expression within a striking phenotypic mosaic Eucalyptus tree that varies in susceptibility to herbivory." *BMC Plant Biology* 13 (1): 29. https://doi.org/10.1186/1471-2229-13-29.

Paradis, E., and K. Schliep. 2019. "ape 5.0: an environment for modern phylogenetics and evolutionary analyses in R." *Bioinformatics* 35:526–528.

Peterlongo, Pierre, Chloé Riou, Erwan Drezen, and Claire Lemaitre. 2017. "DiscoSnp++: de novo detection of small variants from raw unassembled read set(s)." *bioRxiv,* 209965. https://doi.org/10.1101/209965.

Pfeifer, S. P. 2017. "From next-generation resequencing reads to a high-quality variant data set." *Heredity* 118 (2): 111–124. https://doi.org/10.1038/hdy.2016.102.

Plomion, Christophe, Jean-Marc Aury, Joëlle Amselem, Thibault Leroy, Florent Murat, Sébastien Duplessis, Sébastien Faye, et al. 2018. "Oak genome reveals facets of long lifespan." *Nature Plants* 4 (7): 440–452. https://doi.org/10.1038/s41477-018-0172-3.

Poplin, Ryan, Valentin Ruano-Rubio, Mark A. DePristo, Tim J. Fennell, Mauricio O. Carneiro, Geraldine A. Van der Auwera, David E. Kling, et al. 2018. "Scaling accurate genetic variant discovery to tens of thousands of samples." *bioRxiv,* https://doi.org/10.1101/201178.

Prezza, Nicola, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. 2019. "SNPs detection by eBWT positional clustering." *Algorithms for Molecular Biology* 14 (1): 3. https://doi.org/10.1186/s13015-019-0137-8.

Quinlan, Aaron R., and Ira M. Hall. 2010. "BEDTools: a flexible suite of utilities for comparing genomic features." *Bioinformatics* 26 (6): 841–842. https://doi.org/10.1093/bioinformatics/btq033.

R Core Team. 2020. *R: A Language and Environment for Statistical Computing.* Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Ramu, Avinash, Michiel J Noordam, Rachel S Schwartz, Arthur Wuster, Matthew E Hurles, Reed A Cartwright, and Donald F Conrad. 2013. "DeNovoGear: de novo indel and point mutation discovery and phasing." *Nature Methods* 10 (10): 985–987. https://doi.org/10.1038/nmeth.2611.

Reducing Whole-Genome Data Storage Footprint. 2014. White paper 970-2012-013. Illumina, Inc.

Richards, Stephen. 2015. "It's more than stamp collecting: how genome sequencing can unify biological research." *Trends in Genetics* 31 (7): 411–421. https://doi.org/10.1016/j.tig.2015.04.007.

Romberger, J. A., Z. Hejnowicz, and J. F. Hill. 1993. "Plant structure: function and development. A treatise on anatomy and vegetative development with special reference to woody plants." *Plant structure: function and development. A treatise on anatomy and vegetative development with special reference to woody plants.,* https://www.cabdirect.org/cabdirect/abstract/19940600619.

Schirmer, Melanie, Rosalinda D'Amore, Umer Z. Ijaz, Neil Hall, and Christopher Quince. 2016. "Illumina error profiles: resolving fine-scale variation in metagenomic sequencing data." *BMC Bioinformatics* 17 (1): 125. https://doi.org/10.1186/s12859-016-0976-y.

Schirmer, Melanie, Umer Z. Ijaz, Rosalinda D'Amore, Neil Hall, William T. Sloan, and Christopher Quince. 2015. "Insight into biases and sequencing errors for amplicon sequencing with the Illumina MiSeq platform." *Nucleic Acids Research* 43 (6): e37. https://doi.org/10.1093/nar/gku1341.

Schliep, Klaus Peter. 2011. "phangorn: phylogenetic analysis in R." *Bioinformatics* 27 (4): 592–593. https://doi.org/10.1093/bioinformatics/btq706.

Schmid-Siegert, Emanuel, Namrata Sarkar, Christian Iseli, Sandra Calderon, Caroline Gouhier-Darimont, Jacqueline Chrast, Pietro Cattaneo, et al. 2017. "Low number of fixed somatic mutations in a long-lived oak tree." *Nature Plants* 3 (12): 926–929. https://doi.org/10.1038/s41477-017-0066-9.

Schoen, D. J., J. L. David, and T. M. Bataillon. 1998. "Deleterious mutation accumulation and the regeneration of genetic resources." *Proceedings of the National Academy of Sciences* 95 (1): 394–399. https://doi.org/10.1073/pnas.95.1.394.

Schones, Dustin E., Kairong Cui, Suresh Cuddapah, Tae-Young Roh, Artem Barski, Zhibin Wang, Gang Wei, and Keji Zhao. 2008. "Dynamic Regulation of Nucleosome Positioning in the Human Genome." *Cell* 132 (5): 887–898. https://doi.org/10.1016/j.cell.2008.02.022.

Schultz, Stewart T., and Douglas G. Scofield. 2009. "Mutation Accumulation in Real Branches: Fitness Assays for Genomic Deleterious Mutation Rate and Effect in Large-Statured Plants." *The American Naturalist* 174 (2): 163–175. https://doi.org/10.1086/600100.

Scofield, D G. 2014. "A definitive demonstration of fitness effects due to somatic mutation in a plant." *Heredity* 112 (4): 361–362. https://doi.org/10.1038/hdy.2013.114.

Sedlazeck, Fritz J., Philipp Rescheneder, and Arndt von Haeseler. 2013. "NextGenMap: fast and accurate read mapping in highly polymorphic genomes." *Bioinformatics* 29 (21): 2790–2791. https://doi.org/10.1093/bioinformatics/btt468.

Shibuya, Yoshihiro, and Matteo Comin. 2019. "Better quality score compression through sequence-based quality smoothing." *BMC Bioinformatics* 20 (9): 302. https://doi.org/10.1186/s12859-019-2883-5.

Smith, S. A., and M. J. Donoghue. 2008. "Rates of Molecular Evolution Are Linked to Life History in Flowering Plants." *Science* 322 (5898): 86–89. https://doi.org/10.1126/science.1163197.

Song, Li, Liliana Florea, and Ben Langmead. 2014. "Lighter: fast and memory-efficient sequencing error correction without counting." *Genome Biology* 15 (11): 509. https://doi.org/10.1186/s13059-014-0509-9.

Stamatakis, Alexandros. 2014. "RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies." *Bioinformatics* 30 (9): 1312–1313. https://doi.org/10.1093/bioinformatics/btu033.

Steel, M. A., and D. Penny. 1993. "Distributions of Tree Comparison Metrics–Some New Results." *Systematic Biology* 42 (2): 126–141. https://doi.org/10.1093/sysbio/42.2.126.

Sutherland, William J., and Andrew R. Watkinson. 1986. "Somatic mutation: Do plants evolve differently?" *Nature* 320 (6060): 305–305. https://doi.org/10.1038/320305a0.

Taub, Margaret A, Hector Corrada Bravo, and Rafael A Irizarry. 2010. "Overcoming bias and systematic errors in next generation sequencing data." *Genome Medicine* 2 (12): 87. https://doi.org/10.1186/gm208.

Uricaru, Raluca, Guillaume Rizk, Vincent Lacroix, Elsa Quillery, Olivier Plantard, Rayan Chikhi, Claire Lemaitre, and Pierre Peterlongo. 2015. "Reference-free detection of isolated SNPs." *Nucleic Acids Research* 43 (2): e11–e11. https://doi.org/10.1093/nar/gku1187.

Van der Auwera, Geraldine A. 2020a. "Geraldine Van der Auwera on Twitter." Twitter. https://twitter.com/VdaGeraldine/status/1311000033127550976.

Van der Auwera, Geraldine A. 2020b. "Geraldine Van der Auwera on Twitter." Twitter. https://twitter.com/VdaGeraldine/status/1296181178534440963.

Walbot, Virginia. 1985. "On the life strategies of plants and animals." *Trends in Genetics* 1:165–169. https://doi.org/10.1016/0168-9525(85)90071-X.

Wang, Long, Yilun Ji, Yingwen Hu, Huaying Hu, Xianqin Jia, Mengmeng Jiang, Xiaohui Zhang, et al. 2019. "The architecture of intra-organism mutation rate variation in plants." *PLOS Biology* 17 (4): e3000191. https://doi.org/10.1371/journal.pbio.3000191.

Waterson, Robert H., Eric S. Lander, Richard K. Wilson, and The Chimpanzee Sequencing and Analysis Consortium. 2005. "Initial sequence of the chimpanzee genome and comparison with the human genome." *Nature* 437 (7055): 69–87. https://doi.org/10.1038/nature04072.

Watson, J. Matthew, Alexander Platzer, Anita Kazda, Svetlana Akimcheva, Sona Valuchova, Viktoria Nizhynska, Magnus Nordborg, and Karel Riha. 2016. "Germline replications and somatic mutation accumulation are independent of vegetative life span in *Arabidopsis*." *Proceedings of the National Academy of Sciences* 113 (43): 12226–12231. https://doi.org/10.1073/pnas.1609686113.

Whitham, Thomas G., and C. N. Slobodchikoff. 1981. "Evolution by individuals, plant-herbivore interactions, and mosaics of genetic variability: The adaptive significance of somatic mutations in plants." *Oecologia* 49 (3): 287–292. https://doi.org/10.1007/BF00347587.

Wickham, Hadley. 2016. *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York. https://ggplot2.tidyverse.org.

Wu, Steven H, Rachel S Schwartz, David J Winter, Donald F Conrad, and Reed A Cartwright. 2017. "Estimating error models for whole genome sequencing using mixtures of Dirichlet-multinomial distributions." *Bioinformatics* 33 (15): 2322–2329. https://doi.org/10.1093/bioinformatics/btx133.

Yu, Y. William, Deniz Yorukoglu, Jian Peng, and Bonnie Berger. 2015. "Quality score compression improves genotyping accuracy." *Nature Biotechnology* 33 (3): 240–243. https://doi.org/10.1038/nbt.3170.

Yue, Jia-Xing, and Gianni Liti. 2019. "simuG: a general-purpose genome simulator." *Bioinformatics* 35 (21): 4442–4444. https://doi.org/10.1093/bioinformatics/btz424.

Zhong, B., R. Fong, L. J. Collins, P. A. McLenachan, and D. Penny. 2014. "Two New Fern Chloroplasts and Decelerated Evolution Linked to the Long Generation Time in Tree Ferns." *Genome Biology and Evolution* 6 (5): 1166–1173. https://doi.org/10.1093/gbe/evu087.

Zook, Justin M., Daniel Samarov, Jennifer McDaniel, Shurjo K. Sen, and Marc Salit. 2012. "Synthetic spike-in standards improve run-specific systematic error analysis for DNA and RNA sequencing." *PloS One* 7 (7): e41356. https://doi.org/10.1371/journal.pone.0041356.

APPENDIX A

PERMISSION TO INCLUDE PREVIOUSLY PUBLISHED WORK

All authors of the paper *A phylogenomic approach reveals a low somatic mutation rate in a long-lived plant*, doi:10.1098/rspb.2019.2364 have given me permission via e-mail to incorporate the paper as a chapter in this dissertation.

APPENDIX B

SUPPLEMENTARY MATERIAL FOR CHAPTER 2

Filter	Mutations called	False Negative Rate	False Discovery Rate (Number of false positive mutations)
All 3 replicates must agree (main ms)	99	77.3%	56.3% (55.71)
At least 2 replicates must agree	335,867	68%	97.5% (327,351)
Majority rule for any number of replicates	335,867	68%	97.5% (327,351)

Table B.1: The Effects of the Replicate Filter on the Detection of Variants

B.1 Effect of Biological Replicate Filters on Variant Calling

In our original positive control analysis (see main manuscript), we required all three biological replicates (consisting of independently extracted and sequenced samples from three neighboring leaves on a single branch) from a single branch to have the same genotype in order for that site in that site to be used in downstream analyses. We did this to minimize the false positive mutation calls, because we hypothesized when planning the study that false positive mutation calls would be severely detrimental to our ability to perform the positive control. During the review process we were asked to quantify how many false-positive mutation calls were eliminated by requiring all three genotypes to agree.

To do this, we repeated the "variant calling for positive control" part of our analysis described in the main body of the manuscript twice. In the first analysis, we required just two replicates to agree, i.e. we retained all sites where at least two genotypes were identical. In the second analysis we performed a simple majority rule analysis. That is, we retained, as in the first analysis, all sites where at least two genotypes were identical. We additionally retained all sites in which just one of the three genotypes was called (i.e. we excluded sites in which there were two genotypes called but the genotypes differed, and those in which three genotypes were called but all three differed from one another). We call this the 'majority rule' analysis below.

The results of these analyses are shown in supplementary table B.1, with the results from the original analysis shown for comparison.

We note that the results from the majority rule filter are identical to those from the filter where at least two replicates must agree - this is because sites at which just a single replicate was genotyped are removed by other filters in our pipeline. These additional analyses suggest that the use of three replicates was vital to the success of our positive control analysis. Requiring all biological replicates to have matching genotypes produced a somewhat higher false negative rate (77.3% versus 68%) than less stringent approaches, but a dramatically lower false discovery rate (56.3%) than either of the less stringent filters (97.5%). Of note - the number of false positive mutations described in supplementary table 1 for the analysis when all three

replicates must match (55.71 false positive mutations on average over 100 replicates) is much higher than the false positive rate in our full Maximum Likelihood analysis using DeNovoGear described in the main manuscript (0.11 false positive mutations on average over 100 replicates). This is expected - the analysis we report above is based on the positive control analysis in the main manuscript, which uses a relatively crude replicate filter that only retains sites at which all three replicates were genotyped and the genotypes agree. In contrast, the full Maximum Likelihood model implemented in DeNovoGear that we describe in the second part of the manuscript leverages the information contained in the biological replicates more effectively than those that use the GATK pipeline. The point of the GATK analysis was to provide a positive control on which to base downstream analyses.

For the positive control analyses in which we used the replicate filter, the numbers in table B.1 show the importance of using three replicate filters. For example, our phylogenetic positive control would be extremely unlikely to be informative if we used an alignment of 335,867 mutations (as recovered when we require only two genotypes to agree) of which 327,351 are likely to be false positives, simply because the phylogenetic signal from the 2.5% of the data which are true mutations is likely to be swamped by the 97.5% of the data which are false positives. On the other hand, a dataset of 99 variable sites of which roughly 56 are false positives (and thus roughly 43 are likely to be true positives) has a much better chance of recovering the expected phylogenetic tree.

B.2 False Negative Rates Estimated for Each Branch of the Tree

To estimate the false negative rate for the whole dataset, we simulated 14,000 in-silico somatic mutations (1000 on each branch), and used our DeNovoGear pipeline (described in the section of the methods entitled "Variant calling for estimating the rate and spectrum of somatic mutations") to measure how many of these in-silico mutations we could recover. From these data it is also possible to examine whether our false negative rate differs among branches, simply by counting for each branch how many of the 1000 simulated mutations were correctly recovered by our pipeline. These data are provided in table B.2.

The false negative rates range from 65.0% to 76.6%. The standard deviation is 2.96%. One value is more than two standard deviations away from the mean (76.6% false negative rate in branch G->7, which is higher than the mean plus two standard deviations which is 75.96%), suggesting that our ability to infer mutations on the branch that leads from node G to branch tip 7 is slightly lower than would be expected from the distribution of false negative rates. This is most likely explained by the lower coverage of sequencing data for sample 7. Of the 8 samples, the mean sequencing coverage at sites that had a coverage of at least 1 sequenced base (i.e. excluding all sites which had no coverage) were: Sample 1, 42.6x; Sample 2, 38.8x; Sample 3, 37.2x; Sample 4, 39.5x; Sample 5, 46.1x; Sample 6, 35.3x; Sample 7, 25.9x; Sample 8, 29.2x. Sample 7 had the lowest coverage (25.9x) of all sequenced samples. Indeed, sequencing coverage appears to explain roughly 90% of the variation in the false negative rate among samples: the Spearman's of the correlation between the false negative rate of the 8 tip branches and their sequence coverage is -0.90.

Branch Mutated	Mutations Recovered from 1000 simulated	False negative rate
E->F	317	68.30%
D->1	301	69.90%
D->2	285	71.50%
C->3	276	72.40%
B->4	306	69.40%
E->5	307	69.30%
G->6	263	73.70%
G->7	234	76.60%
A->E	328	67.20%
F->G	316	68.40%
A->B	306	69.40%
F->8	281	71.90%
C->D	323	67.70%
B->C	350	65.00%

Table B.2: False Negative Rate in Each Branch of the Tree

B.3 Effects of the ExcessHet Filter on Variant Calling

In our analysis we use the ExcessHet filter to remove sites from the analysis in an attempt to reduce the false positive rate. The motivation for this was that we suspected we would often see false positive mutations at sites which were actually heterozygous in all samples, but at which genotyping errors might have caused a false-positive mutation to be called. A reviewer asked us to investigate the effect of removing this filter, so we repeated our analysis without it. As expected, removing the filter increases the number of mutations detected to 520, suggesting that applying this filter excluded 430 putative mutations. Re-calculating the false negative rate and the number of false positives for this data suggest that the false negative rate is 80.3%, and the number of false positive mutations is 286.78 (implying a false discovery rate of 55.2%). The slight increase in the false negative rate (80.3% without the ExcessHet filter, vs. 77.3% with it) was unexpected - naively one expects that filtering fewer variants should lead to a lower false negative rate. Further examination of the data suggests that this effect occurs because the ExcessHet filter removes a number of small and likely spurious indels (specifically, it removes 643,383 small indels). Removing the filter therefore leads to a small number of variants near these likely spurious indels being removed by our indel proximity filter, leading to an increased false negative rate when not using the ExcessHet filter. The large increase in the number of false positive mutations is expected - it implies that when not using the ExcessHet filter roughly half of the inferred mutations are false positives. The inclusion of these false positives in our analysis would make many of our downstream analyses more problematic, and would make it particularly difficult to judge whether any given mutation was of biological origin or a technical artefact.

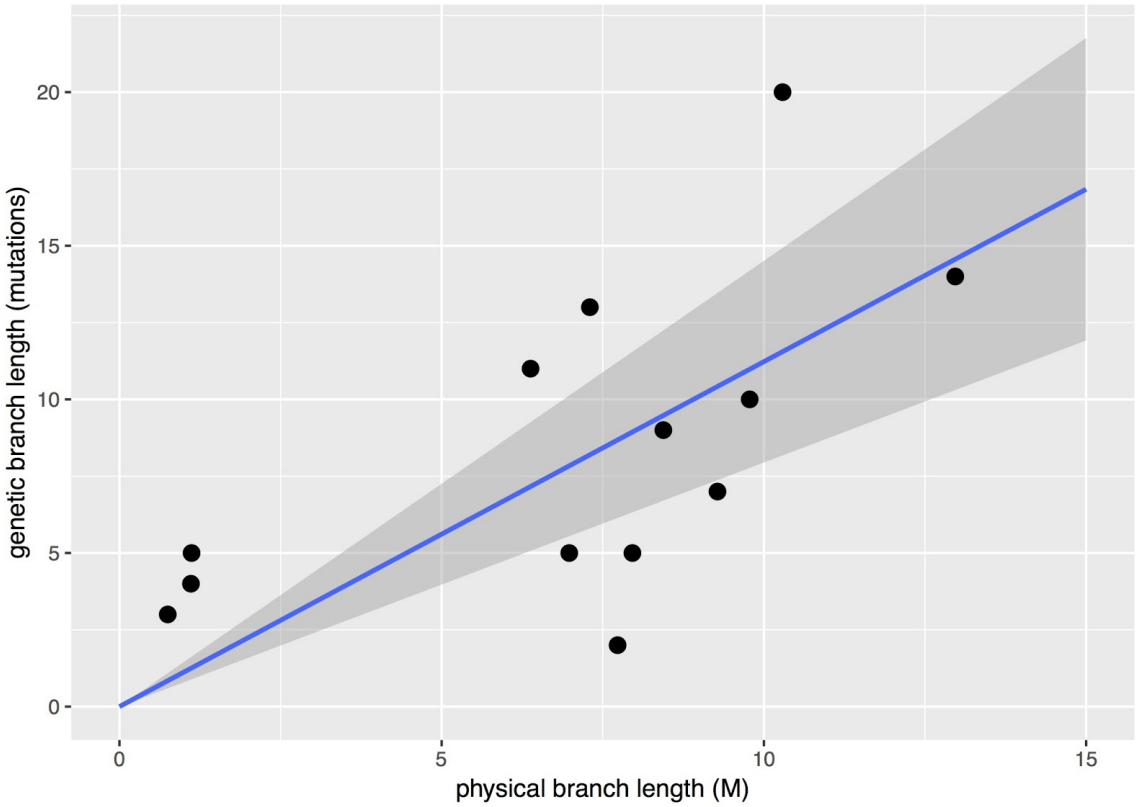


Figure B.1: The Relationship Between Physical Branch Length and Genetic Branch Length | The line of best fit is calculated from a regression forced through the origin.

B.4 Correlation Between Physical and Genetic Branch Lengths

Figure B.1 shows a plot of the linear model forced through the origin that relates the physical branch length in meters to the genetic branch length in inferred somatic mutations, as reported in the main manuscript. This plot shows a significant positive relationship such that longer physical branches tend to have more mutations (R2 = 0.82, p<0.001). The plot was forced through the origin because by definition, a physical branch of length 0 cannot have accumulated somatic mutations. To check whether forcing the origin of the plot through zero is reasonable for this data, we repeated the regression without forcing the origin through zero, allowing us to check whether the uncertainty of the intercept includes zero. This analysis confirms that the uncertainty on the intercept includes zero: the intercept is estimated as 3.19, with 95% confidence intervals of -0.10 to 6.48. Figure B.2 shows a plot of the linear model that is not forced through the origin. This analysis also shows a significant positive relationship such that longer physical branches tend to have more mutations, but as expected the strength of the relationship is weaker (R2 = 0.38, p<0.025).

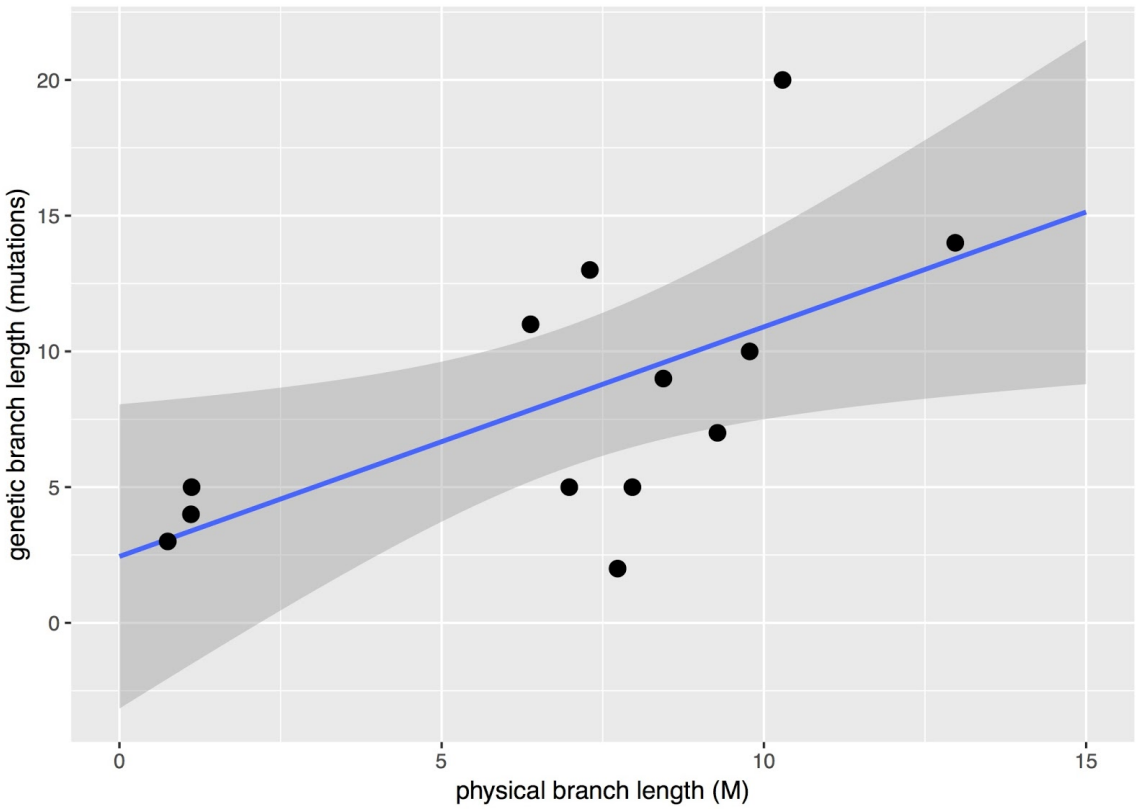


Figure B.2: Physical Branch Length and Genetic Branch Length Not Forced Through Origin | The relationship between physical branch length and genetic branch length. The line of best fit is calculated from a regression that was not forced through the origin.

B.5 Effect of Filtering on the Number of Variants

Table B.3 shows the number of variants remaining after each filtering step in the "Variant Calling for Positive Control" section of the main manuscript. Note that the filters are applied sequentially, such that each row in the table implies that the filter in that row, in addition to the filters in all preceding rows, have been applied. We do not include an estimate of the False Discovery Rate here, because calculating it requires the estimation of a denominator that represents a number of inferred mutations. All of the numbers prior to the application of the final filter (in which we only include variable sites) include both constant heterozygous sites as well as sites that vary among the 8 branches of the tree, meaning that none of them (except the one in the final row) can be used to calculate a meaningful False Discovery Rate.

The table shows, as expected, that the application of each new filter reduces the total number of sites being considered, which in turn increases the false negative rate and decreases the number of false positive mutations that we detect.

Filter	Total Sites	False Negative Rate	Number of false positive mutations
GATK output on 11 scaffolds	9679544	55.40%	1033.18
Site depth <= 500	9109287	56.10%	918.39
ExcessHet <= 40	4958664	56.30%	164.86
Not within 50bp of an indel	3245141	62.00%	99.71
Biallelic SNPs	2049795	62.20%	92.35
Outside repeats	967888	70.10%	55.71
All 3 replicates match	65949	77.30%	55.71
Only include variable sites	99	77.30%	55.71

Table B.3: The Effect of Different Filters on the Number of Sites Remaining Under Consideration

APPENDIX C

SEARCHING FOR PHYLOGENETIC SIGNAL IN THE OUTPUT OF A REFERENCE-FREE VARIANT CALLER

C.1 Introduction

Genotyping a sample given a set of next generation sequencing reads is a common task in biology. However, genotyping accurately and sensitively can be difficult due to errors in sequencing reads (Fox et al. 2014; Pfeifer 2017; Wu et al. 2017). This problem is exacerbated when genotyping non-model organisms because they may lack a polished annotated genome and other prior information that is sometimes used in genotyping pipelines. Additionally, it may be difficult to reliably extract undamaged DNA from the sample material, compounding the error problem (Fonseca et al. 2016). Once obtained, these genotypes can be used for many things; detecting selection in a population, discovering genes and alleles that affect a particular phenotype, or comparing them to another population to attempt to understand their evolutionary history.

While accurate genotype calls are important for all these purposes, accurate detection of variation is difficult even with plenty of time, money, and other auxiliary resources available. Without a high-quality genome and a large number of reads, the task becomes much harder. However, several methods have been developed to call variants directly from sequencing reads, with no additional information necessary. These methods traditionally use De Bruijn graphs to create contiguous sequences representing the genome, much like a genome assembly (Leggett and MacLean 2014). They also keep track of which samples the sequences in the graph came from; once the graph is completed, it is traversed to find bubble-like structures in the graph. These are locations where the sequences specific to different samples diverge briefly, then come back together. This indicates the presence of a single nucleotide polymorphism (SNP) or short insertion-deletion in one of the samples. Then, the sequences that differ can be extracted from the graph and aligned to a reference, if one exists (Iqbal et al. 2012; Uricaru et al. 2015).

Reference-free methods do not necessarily need to use De Bruijn graphs, however. As De Bruijn graphs can be computationally expensive to create, particularly in the amount of computational time and memory required, alternative approaches have also been proposed that utilize different data structures. One method uses an extension of the Burrows-Wheeler transform to find sample divergence in similar sets of reads (Prezza et al. 2019), another does so via k-mer counting (Audano, Ravishankar, and Vannberg 2018). These methods all operate under a common principle: that clustering similar reads together and finding the points of divergence can reveal divergence between samples, even without a reference genome.

There are a variety of software packages that implement reference-free variant calling. One of the first developed is an extension of the Cortex assembler and is called cortex_var (Iqbal et al. 2012). A more recent method called DiscoSNP++ was developed which doesn't require as much RAM or CPU and allows direct VCF output after mapping the predicted variants to a genome (Peterlongo et al. 2017). DiscoSNP++ has recently been updated to include a tool specifically for use with RAD-seq data rather than whole genome sequencing data (Gauthier et al. 2020). The recently developed method, Kestrel, attempts to find variants without a graph and solely considers k-mer counts (Audano, Ravishankar, and Vannberg 2018), while eBWT2SNP (Prezza et al. 2019) utilizes an extended Burrows-Wheeler transform of reads (Mantaci et al. 2007).

I previously called genotypes in the non-model organism *Eucalyptus melliodora* using both out-of-the-box and custom methods to genotype leaves sampled across the tree (Orr et al. 2020). I leveraged the sample structure to estimate false positive rates for each step in the pipeline I used. When I began that work, the available reference-free methods were not sufficiently developed to call variants with the necessary accuracy I wanted to be confident in the results. Since then, the methods have undergone more development, so I was interested in whether they would work for the same data. In this appendix, I use the reference-free variant caller DiscoSNP++ (Peterlongo et al. 2017) construct phylogenies to see whether the output calls capture the branching genetic structure of the tree. I also compare the output of the caller with the set of confident variants detected using the maximum likelihood approach described in (Orr et al. 2020) and implemented with DeNovoGear (Ramu et al. 2013).

C.2 Methods

To obtain variants to compare to the variant set obtained in Orr et al. (2020), I ran the DiscoSNP++ (version 2.3.X) program on the raw sequencing reads. DiscoSNP++ was run with the included run_discoSnp++.sh shell script with the -G, -T, and -R flags set along with the pseudo-reference genome from Orr et al. (2020). This enables alignment of the detected variants to the reference and creation of a VCF from the aligned haplotypes and places k-mers from the reference in the graph to aid in variant calling.

To understand how each filtering step previously applied to this data affected the number of variants in this set, I applied 7 of 8 of these filters to the set of variants. Prior to these filters, however, I used BCFTools (Li 2011a) to remove any site which included a "." genotype, since every sample is diploid; "./." genotypes were permitted until the second to last step. The first filter only permits variants that appear on the first 11 scaffolds of the genome; these represent the chromosomes of *E. melliodora*. Variants with depth > 500 were then excluded, as higher than expected depth is a signal of alignment problems. The ExcessHet annotation could not be used for filtering, as this filter is specific to GATK. I then filtered out variants within 50bp of an indel, another indicator of alignment issues. I then removed any variant that was not a biallelic SNP, as these are more likely to be errors than true mutations. Note that this makes it impossible for this pipeline to detect heterozygous to heterozygous mutations. I then removed variants that were in repetitive regions determined by a liftover of the *E. grandis* RepeatMasker annotation (Bartholomé et al. 2015) to the pseudo-reference. Finally, I removed all sites where any set of three replicates did not have the same genotype. I then included only sites that included a mutation (*ie.* not all heterozygous) to get the final set of variants.

To visualize how each filtering step affected the overall quality of the called variants, I concatenated the variants found in each step to create a FASTA alignment of sites and used RAxML (Stamatakis 2014) with the -f a option to search for the best-scoring maximum likelihood tree, the -m ASC_GTRGAMMA model to use a GTR mixture model with gamma distributed rate heterogeneity and correct for ascertainment bias, and the --asc-corr lewis option to use Lewis' correction for ascertainment bias (Lewis 2001). Variant concatenation is not the optimal way to infer accurate phylogenies; however, for the purposes of a visual summary of the data it is sufficient.

Description	Num Variants	Previously Detected Variants
Appears in first 11 scaffolds	951924	27
Total depth <= 500	910361	27
Not within 50bp of an indel	893535	26
Biallelic SNPs	815925	26
Outside repeat regions	432414	20
All 3 replicates match	2150	0
Only variable sites	4	0

Table C.1: Number of Detected Variants After Each Filtering Step | The number of variants detected at each filtering step. Each row introduces a new filter that applies in addition to all the filters in the above rows. A description of each filter, the number of variants remaining after the filter has been applied, and the number of variants that overlap the previously estimated confident set of variants, which contains 90 variants, are shown.

Trees were visualized in R (R Core Team 2020) with the ape package (Paradis and Schliep 2019).

I also compare the output calls to the confident set of mutation calls made in Orr et al. (2020). Note that, as found in Orr et al. (2020), this set has an approximate false negative rate of 30%, so there are likely many real mutations in the output DiscoSNP++ calls that are not present in the confident call set.

Scripts to reproduce these analyses are shown in Appendix D.

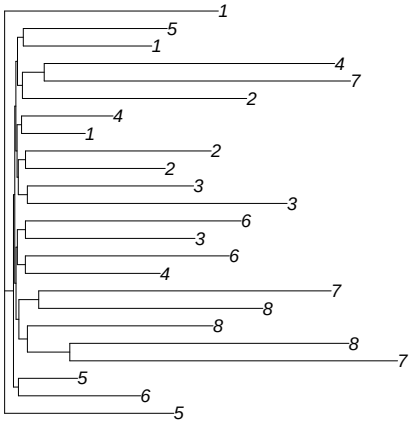
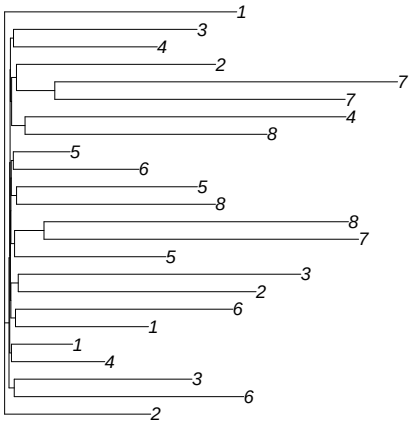
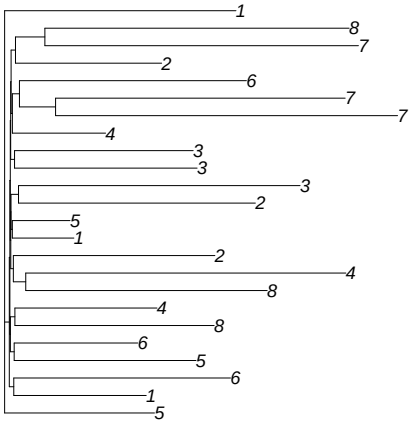
C.3 Results

DiscoSnp++ emitted a total of 1,114,434 potential variants; however, only 951,924 of these mapped to one of the first 11 scaffolds. After applying all filters, only 4 variants were ultimately detected (see Table C.1). Before filtering, 27 of the 90 possible confident set variants were detected; after filtering, none were.

The trees inferred from the set of variants detected at each site are shown in Figure C.1. The second to last filtering step where variants only pass filtering if they are consistent in all three replicates and the last filtering step where variants only pass filtering if there is a mutation (*ie.* not all heterozygous sites of the same genotype) both yielded the same tree, as the nonvariable sites are removed in the tree construction pipeline. Each tree structure differs significantly from the true tree, and there are few examples of all three replicates neighboring one another.

C.4 Discussion

Based on the trees constructed here (Figure C.1), DiscoSNP++ was not able to capture the phylogenetic signal in this data. There are many fewer detected variants at the start of the filtering pipeline compared to those in Orr et al. (2020), with 951,924

Filtering Step	Tree
First 11 Scaffolds	
Depth <= 500	
Not within 50bp of an indel	

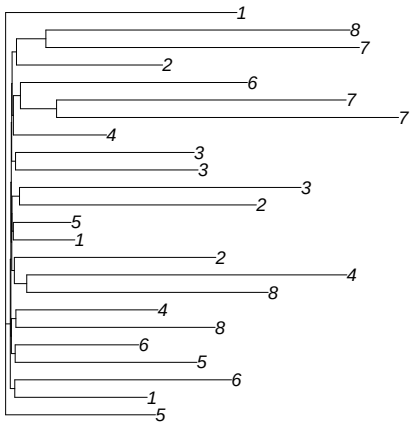
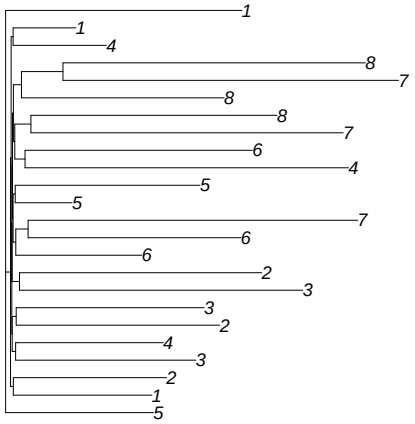
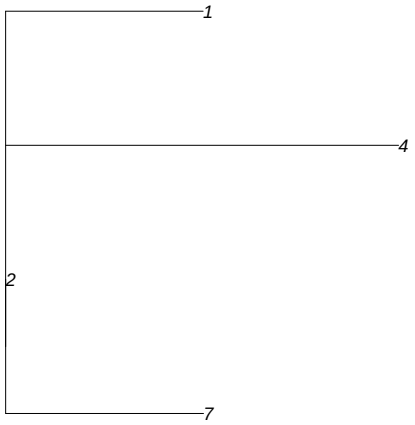
Filtering Step	Tree
Biallelic SNPs	
Outside repeat regions	
All 3 replicates match and only variable sites	

Figure C.1: Trees Inferred at Each Filtering Step | Since each sample was derived from the branch of a tree, its genetic structure should match the physical branching structure. Due to mutations accumulating independently in each branch meristem after branching, the phylogeny should match the true structure.

Filter	Number of sites with a missing genotype	Percent
Appears in first 11 scaffolds	639441	67
Total depth <= 500	636012	70
Not within 50bp of an indel	623544	70
Biallelic SNPs	547636	67
Outside repeat regions	425544	98
All 3 replicates match	309	14
Only variable sites	0	0

Table C.2: Number of Sites With a Missing Genotype After Each Filtering Step | The number of sites with a missing genotype after each filtering step. Having many missing genotypes makes it difficult to accurately resolve the tree. In the second to last step, when there are relatively few missing genotypes, there aren't sufficient informative sites to accurately construct the tree.

variants on the first 11 scaffolds compared to 9,679,544 detected with HaplotypeCaller. This reduced number of potential variants is consistent throughout filtering steps. In all the constructed trees, replicates of the same sample should cluster closely together; however, this is not the case.

One reason for this is the variant calls emitted by DiscoSNP++ are very sparse, and many sites have at least one missing (./.) genotype (see Table C.2). The presence of a missing genotype for any site eliminates the ability for it to contribute useful information during phylogeny construction. Thus, due to the large amount of missing data, the number of informative sites in the alignment is small.

There are a variety of reasons the algorithm is unable to call genotypes at so many sites. One reason is that the algorithm lacks a reference, so when it detects a variant it may not be able to map it back to the genome. For complex or clustered variants, this may be a sample-specific effect that could contribute to differential calling ability on some samples. This is an especially difficult problem with a repetitive and low-quality genome, such as the one used here. Another major reason is the lack of per-sample depth in this data. Since each replicate was only sequenced to 10X depth, the reads are relatively short, and the genome is repetitive, the De Bruijn graph construction may not be well-connected enough to detect variant bubbles.

C.5 Conclusion

Reference-free variant callers have the potential to be powerful tools for detecting variants in non-model organisms. They are also free of reference bias that may impact other variant callers. However, the reference-free caller used here was unable to capture sufficient phylogenetic signal in a structured dataset. Larger per-sample depths and smaller genomes with few repetitive regions would likely improve these results.

Audano, Peter A, Shashidhar Ravishankar, and Fredrik O Vannberg. 2018. "Mapping-free variant calling using haplotype reconstruction from k-mer frequencies." *Bioinformatics* 34 (10): 1659–1665. https://doi.org/10.1093/bioinformatics/btx753.

Bartholomé, Jérôme, Eric Mandrou, André Mabiala, Jerry Jenkins, Ibouniyamine Nabihoudine, Christophe Klopp, Jeremy Schmutz, Christophe Plomion, and Jean-Marc Gion. 2015. "High-resolution genetic maps of *Eucalyptus* improve *Eucalyptus grandis* genome assembly." *New Phytologist* 206 (4): 1283–1296. https://doi.org/10.1111/nph.13150.

Fonseca, Rute R. da, Anders Albrechtsen, Gonçalo Espregueira Themudo, Jazmín Ramos-Madrigal, Jonas Andreas Sibbesen, Lasse Maretty, M. Lisandra Zepeda-Mendoza, Paula F. Campos, Rasmus Heller, and Ricardo J. Pereira. 2016. "Next-generation biology: Sequencing and data analysis approaches for non-model organisms." *Marine Genomics* 30:3–13. https://doi.org/10.1016/j.margen.2016.04.012.

Fox, Edward J., Kate S. Reid-Bayliss, Mary J. Emond, and Lawrence A. Loeb. 2014. "Accuracy of Next Generation Sequencing Platforms." *Next generation, sequencing & applications* 1. https://doi.org/10.4172/jngsa.1000106.

Gauthier, Jérémy, Charlotte Mouden, Tomasz Suchan, Nadir Alvarez, Nils Arrigo, Chloé Riou, Claire Lemaitre, and Pierre Peterlongo. 2020. "DiscoSnp-RAD: de novo detection of small variants for RAD-Seq population genomics." *PeerJ* 8:e9291. https://doi.org/10.7717/peerj.9291.

Iqbal, Zamin, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. 2012. "De novo assembly and genotyping of variants using colored de Bruijn graphs." *Nature genetics* 44 (2): 226–232. https://doi.org/10.1038/ng.1028.

Leggett, Richard M., and Dan MacLean. 2014. "Reference-free SNP detection: dealing with the data deluge." *BMC Genomics* 15 (4): S10. https://doi.org/10.1186/1471-2164-15-S4-S10.

Lewis, Paul O. 2001. "A Likelihood Approach to Estimating Phylogeny from Discrete Morphological Character Data." *Systematic Biology* 50 (6): 913–925. https://doi.org/10.1080/106351501753462876.

Li, Heng. 2011a. "A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data." *Bioinformatics* 27 (21): 2987–2993. https://doi.org/10.1093/bioinformatics/btr509.

Mantaci, S., A. Restivo, G. Rosone, and M. Sciortino. 2007. "An extension of the Burrows–Wheeler Transform." *Theoretical Computer Science* 387 (3): 298–312. https://doi.org/10.1016/j.tcs.2007.07.014.

Orr, Adam J., Amanda Padovan, David Kainer, Carsten Külheim, Lindell Bromham, Carlos Bustos-Segura, William Foley, et al. 2020. "A phylogenomic approach reveals a low somatic mutation rate in a long-lived plant." *Proceedings of the Royal Society B: Biological Sciences* 287 (1922): 20192364. https://doi.org/10.1098/rspb.2019.2364.

Paradis, E., and K. Schliep. 2019. "ape 5.0: an environment for modern phylogenetics and evolutionary analyses in R." *Bioinformatics* 35:526–528.

Peterlongo, Pierre, Chloé Riou, Erwan Drezen, and Claire Lemaitre. 2017. "DiscoSnp++: de novo detection of small variants from raw unassembled read set(s)." *bioRxiv,* 209965. https://doi.org/10.1101/209965.

Pfeifer, S. P. 2017. "From next-generation resequencing reads to a high-quality variant data set." *Heredity* 118 (2): 111–124. https://doi.org/10.1038/hdy.2016.102.

Prezza, Nicola, Nadia Pisanti, Marinella Sciortino, and Giovanna Rosone. 2019. "SNPs detection by eBWT positional clustering." *Algorithms for Molecular Biology* 14 (1): 3. https://doi.org/10.1186/s13015-019-0137-8.

R Core Team. 2020. *R: A Language and Environment for Statistical Computing.* Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Ramu, Avinash, Michiel J Noordam, Rachel S Schwartz, Arthur Wuster, Matthew E Hurles, Reed A Cartwright, and Donald F Conrad. 2013. "DeNovoGear: de novo indel and point mutation discovery and phasing." *Nature Methods* 10 (10): 985–987. https://doi.org/10.1038/nmeth.2611.

Stamatakis, Alexandros. 2014. "RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies." *Bioinformatics* 30 (9): 1312–1313. https://doi.org/10.1093/bioinformatics/btu033.

Uricaru, Raluca, Guillaume Rizk, Vincent Lacroix, Elsa Quillery, Olivier Plantard, Rayan Chikhi, Claire Lemaitre, and Pierre Peterlongo. 2015. "Reference-free detection of isolated SNPs." *Nucleic Acids Research* 43 (2): e11–e11. https://doi.org/10.1093/nar/gku1187.

Wu, Steven H, Rachel S Schwartz, David J Winter, Donald F Conrad, and Reed A Cartwright. 2017. "Estimating error models for whole genome sequencing using mixtures of Dirichlet-multinomial distributions." *Bioinformatics* 33 (15): 2322–2329. https://doi.org/10.1093/bioinformatics/btx133.

APPENDIX D

CODE TO REPRODUCE ANALYSES IN APPENDIX C

```
# FILE:./missing_sites/Makefile


GATKVARIANTS = ../gatk4/repeat-filter.vcf
SCRIPTDIR = ~/bin/zypy/scripts
FLTWITHREP = $(SCRIPTDIR)/filt_with_replicates.pl
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
VCFTOFA = $(SCRIPTDIR)/vcf2fa.sh
PLOTTREE = $(SCRIPTDIR)/plot_tree.R
GATKFPR = $(SCRIPTDIR)/gatkfpr.py

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz \
        04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
        07-replicate.vcf.gz 08-variable.vcf.gz

TXTS = $(addsuffix .txt, $(basename $(basename $(ALL))))

.DEFAULT: n_incomplete.txt

.PHONY: pdfs

%.vcf.gz: %.vcf
  bgzip -c $< >$@

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

%.txt: ../%.vcf.gz
  bcftools view -e 'GT="."' -Ou $< | bcftools view -H -i'GT="./."' |
  ↪  wc -l > $@

.SECONDARY:

n_sites_missing.txt: $(TXTS)
  parallel --tag -I{} 'cat {}' ::: $^ > $@

# FILE:./tp_fn/Makefile



#VARIANTSDIR = ../../2018-07-24_callability/
#REF = ../../e_mel_3/e_mel_3.fa
#ALIGNMENT = $(VARIANTSDIR)/alignment.bam
#REPEATBED = ../../liftover/e_mel_3_repeatmask.bed
#RAWVARIANTS = $(VARIANTSDIR)/gatk/var-calls.vcf.gz
#INREPEATS = $(VARIANTSDIR)/gatk/inrepeats.txt
DNGCALLS = ../../dng/filter/goodsites.vcf
```

```
SCRIPTDIR = ~/bin/zypy/scripts
FLTWITHREP = $(SCRIPTDIR)/filt_with_replicates.pl
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
VCFTOFA = $(SCRIPTDIR)/vcf2fa.sh
PLOTTREE = $(SCRIPTDIR)/plot_tree.R
CALLABILITYRES = $(SCRIPTDIR)/spiked_mutation_results.py

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz \
  04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
  07-replicate.vcf.gz 08-variable.vcf.gz

RESULTS = $(addsuffix .result, $(basename $(basename $(ALL))))
FNRESULTS = $(addsuffix .fn.result, $(basename $(basename $(ALL))))
TPRESULTS = $(addsuffix .tp.result, $(basename $(basename $(ALL))))

results: fn.txt tp.txt

.PHONY: results

.SECONDARY:

%.vcf.gz: %.vcf
  bgzip -c $< >$@

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

#Calls not in the callset but in DNGCALLS are FN
%.fn.result: ../%.vcf.gz
  bcftools view -H -T ^$< $(DNGCALLS) | wc -l > $@

#Calls that match DNGCALLS are TPs
%.tp.result: ../%.vcf.gz
  bcftools view -H -T $(DNGCALLS) $< | wc -l > $@

fn.txt: $(FNRESULTS)
  parallel -I{} --tag 'cat {}' ::: $^ > $@

tp.txt: $(TPRESULTS)
  parallel -I{} --tag 'cat {}' ::: $^ > $@

# FILE:./Makefile


VARIANTSDIR = ../gatk4/
REF = ../e_mel_3/e_mel_3.fa
```

```
REFDICT = $(basename $(REF)).dict
RAWALIGNMENT = ../alignment.dedup.bam
REPEATBED = ../liftover/e_mel_3_repeatmask.bed
RAWVARIANTS = discoRes_k_31_c_3_D_100_P_3_b_0_coherent.vcf
SCAFFOLDS = $(addprefix scaffold_,$(shell seq 1 11))
SCAFFOLDEDVARIANTS = $(addsuffix .var-calls.vcf.gz, $(SCAFFOLDS))
DATADIR = ../data/

SCRIPTDIR = ~/bin/zypy/scripts
FLTWITHREP = $(SCRIPTDIR)/filt_with_replicates.pl
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
VCFTOFA = $(SCRIPTDIR)/vcf2fa.sh
PLOTTREE = $(SCRIPTDIR)/plot_tree.R

SAMPLENUMS := 1 2 3 4 5 6 7 8
SAMPLEREPS := a b c
SAMPLES := $(foreach NUM,$(SAMPLENUMS),$(addprefix
 ↪  M$(NUM),$(SAMPLEREPS)))
SAMPLECMD := $(foreach SAM,$(SAMPLES),echo $(wildcard
 ↪  $(DATADIR)/*$(SAM)*/$(SAM)*.fastq | tr ' ' '\t' >
 ↪  $(SAM)_fof.txt))
SAMPLEFOFS := $(addsuffix _fof.txt,$(SAMPLES))

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz \
        04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
        07-replicate.vcf.gz 08-variable.vcf.gz

all: $(ALL) num_variants.txt

.SHELL: bash

.PHONY: all disco

%.vcf.gz: %.vcf
  bgzip -c $< >$@

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

%.vcf.gz.tbi: %.vcf.gz
  bcftools index -t $<

%.bam.bai: %.bam
  samtools index $<

.SECONDARY:
```

```
.PRECIOUS:

%_fof.txt:
  echo $(wildcard $(DATADIR)/*$**/$**.fastq) $% |\
  xargs -n1 > $@

fof.txt: $(SAMPLEFOFS)
  echo $^ | xargs -n1 > $@

disco: fof.txt $(REF)
  ~/bin/DiscoSnp/run_discoSnp++.sh -r $< -T -G $(word 2, $^) -R

disco_sample_names.txt: discoRes_read_files_correspondance.txt
  sed 's/C_/G/; s/[[:space:]].*\(M[[:digit:]][abc]\).*/ \1/g' $< >$@

#sorted.vcf.gz: $(RAWVARIANTS).gz disco_sample_names.txt
#  bcftools reheader --samples $(word 2, $^) $< | \
#  bcftools sort -m64G -Oz -o $@ -

01-first-11-scaffolds.vcf.gz: $(RAWVARIANTS).gz
↪  disco_sample_names.txt $(REF).fai
  bcftools reheader --samples $(word 2, $^) -f $(word 3, $^) $< | \
  bcftools view -t
  ↪  scaffold_1,scaffold_2,scaffold_3,scaffold_4,scaffold_5,scaffold
  ↪  _6,scaffold_7,scaffold_8,scaffold_9,scaffold_10,scaffold_11 -Ou
  ↪  | \
  bcftools sort -m 64G -Oz -o $@ -

02-depth.vcf.gz: 01-first-11-scaffolds.vcf.gz
↪  01-first-11-scaffolds.vcf.gz.csi
  bcftools filter -i 'SUM(FORMAT/DP) <= 500' -O z -o $@ $<

#03-excesshet.vcf.gz: 02-depth.vcf.gz 02-depth.vcf.gz.csi
#  bcftools filter -i 'ExcessHet <= 40' -O z -o $@ $<

#testing shows -g is applied last
04-near-indel.vcf.gz: 02-depth.vcf.gz 02-depth.vcf.gz.csi
  bcftools filter -g 50 -O z -o $@ $<

05-biallelic.vcf.gz: 04-near-indel.vcf.gz 04-near-indel.vcf.gz.csi
  bcftools view -m2 -M2 -v snps -O z -o $@ $<

06-repeat.vcf.gz: 05-biallelic.vcf.gz $(REPEATBED)
↪  05-biallelic.vcf.gz.csi
  vcftools --gzvcf $< --exclude-bed $(word 2, $^)
  ↪  --remove-filtered-all --recode --recode-INFO-all --stdout |
  ↪  bgzip -c > $@
```

```
07-replicate.vcf.gz: 06-repeat.vcf.gz 06-repeat.vcf.gz.csi
  zcat $< | $(FLTWITHREP) -s -g 3 | bgzip -c > $@

08-variable.vcf.gz: 07-replicate.vcf.gz
  zcat $< | $(DIPLOIDIFY) -t vcf -v | bgzip -c > $@

num_variants.txt: $(ALL)
  parallel --tag -I{} "bcftools view -e 'GT=\".\"' -H {} | wc -l" :::
  ↪  $(ALL) > $@



# FILE:./fp/Makefile


GATKVARIANTS = ../01-first-11-scaffolds.vcf.gz
CALLEDVARIANTS = ../08-variable.vcf.gz
JOBNUMS=$(shell seq 1 100)
VCFNUMS = $(addsuffix .vcf.gz, $(JOBNUMS))
JOBCALLS=$(addprefix %/,$(VCFNUMS))

SCRIPTDIR = ~/bin/zypy/scripts
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
FPRFILT = $(SCRIPTDIR)/gatkfpr.py
SHUF_LABELS = $(SCRIPTDIR)/label_permutation.R
SHELL=bash
DNGCALLS = ../../dng/filter/goodsites.vcf
############################################################

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz \
        04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
        07-replicate.vcf.gz 08-variable.vcf.gz

RESULTS = $(addsuffix .result, $(basename $(basename $(ALL))))
RESFOLDERS = $(addsuffix /, $(basename $(basename $(ALL))))
ALLVCFS = $(foreach folder, $(RESFOLDERS), $(addprefix $(folder),
↪  $(VCFNUMS)))

default: all

all: $(RESULTS)

.PHONY: all default

.DELETE_ON_ERROR:
```

```
.INTERMEDIATE:

.SECONDARY:

############################################################
# Helper Rules

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

#

trees/:
  mkdir -p $@

trees/%.ped: trees/
  echo "##PEDNG v1.0" > $@; echo -e "M\t.\t.\t0" | paste - <(
  ↪  $(SHUF_LABELS) ) >> $@

$(RESFOLDERS):
  mkdir -p $@

vpath %.ped trees/

$(ALLVCFS): | $(RESFOLDERS)
  $(FPRFILT) -n 3 -v ../$(subst /,,$(dir $@)).vcf.gz -p
  ↪  trees/$(notdir $(basename $(basename $@))).ped | \
  $(DIPLOIDIFY) -t vcf -v | bgzip > $@

$(RESULTS): %.result : $(DNGCALLS) $(JOBCALLS)
  rm -f $@; for f in $(wordlist 2,$(words $^),$^) ; do bcftools view
  ↪  -H -T ^$< $$f | wc -l >> $@; done

#FP, FP/P (FDR)
results.txt: $(RESULTS)
  parallel -I{} --tag 'cat {} | awk "{x+=\$$1}END{print x,x/NR}"' :::
  ↪  $^ > $@


# FILE:./trees/Makefile


GATKVARIANTS = ../gatk4/repeat-filter.vcf
SCRIPTDIR = ~/bin/zypy/scripts
FLTWITHREP = $(SCRIPTDIR)/filt_with_replicates.pl
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
VCFTOFA = $(SCRIPTDIR)/vcf2fa.sh
```

```
PLOTTREE = $(SCRIPTDIR)/plot_tree.R
GATKFPR = $(SCRIPTDIR)/gatkfpr.py

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz \
        04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
        07-replicate.vcf.gz 08-variable.vcf.gz

PDFS = $(addsuffix .pdf, $(basename $(basename $(ALL))))

pdfs: $(PDFS)

.PHONY: pdfs

%.vcf.gz: %.vcf
  bgzip -c $< >$@

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

%.fa: ../%.vcf.gz
  bcftools view -e 'GT="."' $< | $(VCFTOFA) | $(DIPLOIDIFY) -v >$@

.SECONDARY:

%.nwk: %.fa
  export RXMLTMPDIR=$(shell mktemp -d --tmpdir=$(CURDIR)
  ↪  raxml_XXXXXX); \
  raxml -T 8 -f a -s $< -n nwk -m ASC_GTRGAMMA -w $$RXMLTMPDIR
  ↪  --asc-corr=lewis -p 12345 -x 12345 -# 100 && \
  cat $$RXMLTMPDIR/RAxML_bestTree.nwk >$@ &&\
  rm -rf $$RXMLTMPDIR

%.pdf: %.nwk
  $(PLOTTREE) $< $@

%.png: %.pdf
  gs -sDEVICE=pngalpha -o $@ $<
```

APPENDIX E

CODE TO REPRODUCE ANALYSES IN CHAPTER 3

```
# FILE:./fnrfpr/Makefile


DATADIR = ../../

VCFIN = $(DATADIR)/chr1.vcf.gz
BAMIN = $(DATADIR)/chr1.bam
BEDIN = $(DATADIR)/chr1_confident.bed.gz
REFIN = $(DATADIR)/chr1.renamed.fa
PROCESSEDBAM = chr1_only_confident.fixmate.sorted.bam

VCFLINES := $(shell bcftools index -n $(VCFIN))
INPUTINCREMENT = 20
INPUTFIRST = 0
INPUTLAST = 100

SHELL=bash
FPRS := $(shell seq $(INPUTFIRST) $(INPUTINCREMENT) $(INPUTLAST))
#FNR_FPR
ADD_FPRS = $(addsuffix _$(FPR),$(FPRS))
ERRATES := $(foreach FPR,$(FPRS),$(ADD_FPRS))
ERBENCHMARKS := $(addsuffix .benchmark,$(ERRATES))
ERPLOTS := $(addsuffix .recal.after.pdf,$(ERRATES))
ERCSVS := $(addsuffix .recal.after.csv,$(ERRATES))

#functions to get FNR and FPR of a given file named FNR_FPR.*
getfnr = $(word 1, $(subst _, ,$(1)))
getfpr = $(word 2, $(subst ., ,$(subst _, ,$(1))))

#actual negative and actual positive files
ANBED = ansites.bed.gz
APBED = apsites.bed.gz
ANLINES = $(shell zcat $(ANBED) | wc -l)
APLINES = $(shell zcat $(APBED) | wc -l)

.PHONY: plots csvs all

plots: $(ERPLOTS)

csvs: $(ERCSVS)

all: plots csvs

.DEFAULT: all
```

```
.PRECIOUS:

.SECONDARY:

#index helpers
%.bed.gz.tbi: %.bed.gz
  tabix -p bed $<

%.vcf.gz.tbi: %.vcf.gz
  bcftools index -t $<

%.bam.bai: %.bam
  samtools index $<

#process raw BAM
chr1_only_confident.namesorted.bam: $(BAMIN) $(BEDIN)
  samtools view -h $< -F 3844 -L $(word 2, $^) | samtools sort -@8 -n
  ↪  -o $@ -O bam -

chr1_only_confident.fixmate.bam: chr1_only_confident.namesorted.bam
  samtools fixmate -r --threads 8 $< $@

chr1_only_confident.fixmate.onlypaired.bam:
↪  chr1_only_confident.fixmate.bam
  samtools view -h -b -o $@ -f 1 $<

$(PROCESSEDBAM): chr1_only_confident.fixmate.onlypaired.bam
  samtools sort -@ 8 -O bam -o $@ $<

#--- BED files containing the appropraite FNR_FPR ---
#false negative rate = FN / AP
$(APBED): $(VCFIN) $(BEDIN)
  bcftools query -R $(word 2, $^) -f '%CHROM\t%POS0\t%END\n' $< | \
  sort -k1,1 -k2,2n | bedtools merge -i - | \
  bedtools intersect -sorted -a - -b $(word 2, $^) | \
  bedtools makewindows -b - -w 1 | \
  sort -k1,1 -k2,2n | bgzip >$@

%.fn.bed.gz: $(APBED)
  zcat $< | \
  shuf -n $(shell echo $$(($(APLINES) * (100 - $(call
  ↪  getfnr,$*))/100))) | \
  sort -k1,1 -k2,2n | bgzip > $@

#false positive rate =  FP / AN
$(ANBED): $(BEDIN) $(APBED)
  bedtools subtract -a $< -b $(word 2, $^) | \
```

```
  bedtools makewindows -b - -w 1 | \
  sort -k1,1 -k2,2n | bgzip > $@

%.fp.bed.gz: $(ANBED)
  zcat $< | shuf -n $(shell echo $$(($(ANLINES) * $(call getfpr,$*) /
  ↪  100))) | \
  sort -k1,1 -k2,2n | bgzip > $@

#combine the false positives with the appropriate number
#of false negatives
%.toskip.bed.gz: %.fp.bed.gz %.fn.bed.gz
  zcat $(word 2, $^) | cat <(zcat $<) - | \
  sort -k1,1 -k2,2n | bedtools merge -i - | bgzip >$@

# --- Do calibration ---
%.recal.txt: $(PROCESSEDBAM) $(REFIN) $(BEDIN) %.toskip.bed.gz
↪  $(PROCESSEDBAM).bai %.toskip.bed.gz.tbi
  gatk BaseRecalibrator -I $< -R $(word 2, $^) -L $(word 3, $^)
  ↪  --known-sites $(word 4, $^) --use-original-qualities -O $@

%.recal.bam: $(PROCESSEDBAM) $(REFIN) %.recal.txt
  gatk ApplyBQSR -I $< -R $(word 2, $^) --bqsr-recal-file $(word 3,
  ↪  $^) --use-original-qualities -O $@

cbbq.recal.bam: $(PROCESSEDBAM)
  ~/bin/cbbq/build/cbbq -t6 -k32 -g 214206308 -a .15 --use-oq $< > $@
  ↪  2> cbbqlog.txt


# --- Benchmark with GATK ---
# Use actual positives for this test.
%.recal.after.txt: %.recal.bam $(REFIN) $(BEDIN) $(APBED)
↪  $(APBED).tbi
  gatk BaseRecalibrator -I $< -R $(word 2, $^) -L $(word 3, $^)
  ↪  --known-sites $(word 4, $^) -O $@

%.recal.after.pdf: %.recal.txt %.recal.after.txt
  gatk AnalyzeCovariates -before $< -after $(word 2, $^) -plots $@

raw.recal.after.pdf: raw.recal.after.txt
  gatk AnalyzeCovariates -bqsr $< -plots $@

%.recal.after.csv: %.recal.txt %.recal.after.txt
  gatk AnalyzeCovariates -before $< -after $(word 2, $^) -csv $@

raw.recal.after.csv: raw.recal.after.txt
  gatk AnalyzeCovariates -bqsr $< -csv $@
```

```
allplots.pdf: $(ERPLOTS)
  gs -dBATCH -dNOPAUSE -q -sDEVICE=pdfwrite -sOutputFile=$@ $^

# FILE:./sims/Makefile


SHELL=bash
EGRANDISURL1 = https://sra-download.ncbi.nlm.nih.gov/traces/wgs03/wgs
↪  _aux/AU/SX/AUSX01/AUSX01.1.fsa_nt.gz
EGRANDISURL2 = https://sra-download.ncbi.nlm.nih.gov/traces/wgs03/wgs
↪  _aux/AU/SX/AUSX01/AUSX01.2.fsa_nt.gz
EGRANDISPT1 = $(notdir $(EGRANDISURL1))
EGRANDISPT2 = $(notdir $(EGRANDISURL2))
SAMPLEDREFSIZE = 5000000
ARTDIR = ./art_bin_MountRainier/

.PRECIOUS:

.SECONDARY:

#index helpers
%.fa.gz.fai: %.fa.gz
  samtools faidx $<

%.bam.bai: %.bam
  samtools index $<

%.vcf.gz: %.vcf
  bgzip -c $< > $@

%.vcf.gz.tbi: %.vcf.gz
  bcftools index -t $<

%.dict: %.fa
  gatk CreateSequenceDictionary -R $<

%.dict: %.fa.gz
  gatk CreateSequenceDictionary -R $<

#download AUSX00000000.1, the e. grandis 1.0 contigs.
#this would represent an assembly where scaffolds haven't been
↪  joined into contigs
$(EGRANDISPT1):
  wget $(EGRANDISURL1) -O $@

$(EGRANDISPT2):
```

```
  wget $(EGRANDISURL2) -O $@

full_grandis_contigs.fa.gz: $(EGRANDISPT1) $(EGRANDISPT2)
  cat <(zcat $<) <(zcat $(word 2, $^)) | bgzip -c > $@

#subsample contigs so we have a reasonable amount of data
#GATK recommends at least 100M bp per read group
#to have 100M bp with 150bp reads, we need 100000000/150 = 666,667
↪  reads
#at 20X coverage, which is high but reasonable for a non-model
↪  organism, this requires
# 20 = sequencedbp / gsize; gsize = sequencedbp/20
#So we need a genome size of 5M with 20x to read 100M bases.
sampled_contigs.txt: full_grandis_contigs.fa.gz.fai
  cat $< | shuf | awk '{if (x < $(SAMPLEDREFSIZE)) {x += $$2; print
  ↪  $$1} else exit}' > $@

sampled_contigs.fa.gz: sampled_contigs.txt full_grandis_contigs.fa.gz
  cat $< | xargs samtools faidx $(word 2, $^) | bgzip -c > $@

#theta 0.025
#5,000,000 bp * 0.025 = 125,000
#titv ratio of 2 is an appropriate estimate for eucs
simug.simseq.genome.fa simug.refseq2simseq.SNP.vcf &:
↪  sampled_contigs.fa.gz
  simuG.pl -seed 20201112 -titv_ratio 2.0 -snp_count $$((
  ↪  $(SAMPLEDREFSIZE) * 25 / 1000 )) -refseq $< -prefix simug

#concatenate both references to generate a file with both
#haplotypes in it
#adds a .s suffix to the sampled contigs so they can be
↪  differentiated
combined_ref.fa: sampled_contigs.fa.gz simug.simseq.genome.fa
  zcat $< | sed '/^>/s/$$/.s/' | cat - $(word 2, $^) > $@

#simulate reads from the reference with ART
art1.fq art2.fq art.sam &: combined_ref.fa
  $(ARTDIR)/art_illumina -sam -na -ss HS25 -p -l 101 -m 300 -s 20 -rs
  ↪  20201112 -i $< -f 20 -o art

#align with ngm
ngm.bam: sampled_contigs.fa.gz art1.fq art2.fq
  ngm -r $< -1 $(word 2, $^) -2 $(word 3, $^) -t 4 --rg-id sim
  ↪  --rg-sm sim --rg-lb sim --rg-pl ILLUMINA --rg-pu sim | samtools
  ↪  sort -m 2G -O bam -o $@ -

#recalibrate with true variants
```

```
%.recal.txt: %.bam sampled_contigs.fa.gz
 ↪  simug.refseq2simseq.SNP.vcf.gz %.bam.bai sampled_contigs.dict
 ↪  simug.refseq2simseq.SNP.vcf.gz.tbi
  gatk BaseRecalibrator -I $< -R $(word 2, $^) --known-sites $(word
   ↪  3, $^) -O $@

#recalibrate with true variants
%.recal.bam: %.bam sampled_contigs.fa.gz %.recal.txt
  gatk ApplyBQSR -I $< -R $(word 2, $^) --bqsr-recal-file $(word 3,
   ↪  $^) -O $@

#benchmark recalibration with true variants
%.recal.after.txt: %.recal.bam sampled_contigs.fa.gz
 ↪  simug.refseq2simseq.SNP.vcf.gz %.recal.bam.bai
 ↪  sampled_contigs.dict simug.refseq2simseq.SNP.vcf.gz.tbi
  gatk BaseRecalibrator -I $< -R $(word 2, $^) --known-sites $(word
   ↪  3, $^) -O $@

%.recal.after.csv: %.recal.txt %.recal.after.txt
  gatk AnalyzeCovariates -before $< -after $(word 2, $^) -csv $@

%.recal.after.pdf: %.recal.txt %.recal.after.txt
  gatk AnalyzeCovariates -before $< -after $(word 2, $^) -plots $@

kbbq-ngm.recal.bam: ngm.bam
  ~/bin/cbbq/build/cbbq -t6 -k32 -g $(SAMPLEDREFSIZE) $< > $@ 2>
   ↪  cbbqlog.txt

kbbq-ngm.recal.after.csv: kbbq-ngm.recal.after.txt
  gatk AnalyzeCovariates -after $< -csv $@

#get initial calls with HC and use them to recalibrate
initial-calls.vcf.gz initial-calls.vcf.gz.tbi &: ngm.bam
 ↪  sampled_contigs.fa.gz ngm.bam.bai sampled_contigs.dict
  gatk HaplotypeCaller -R $(word 2, $^) -I $< -stand-call-conf 50
   ↪  --native-pair-hmm-threads 8 -O $@ --heterozygosity 0.025

initial-calls.recal.txt: ngm.bam sampled_contigs.fa.gz
 ↪  initial-calls.vcf.gz ngm.bam.bai sampled_contigs.dict
 ↪  initial-calls.vcf.gz.tbi
  gatk BaseRecalibrator -I $< -R $(word 2, $^) --known-sites $(word
   ↪  3, $^) -O $@

initial-calls.recal.bam: ngm.bam sampled_contigs.fa.gz
 ↪  initial-calls.recal.txt
  gatk ApplyBQSR -I $< -R $(word 2, $^) --bqsr-recal-file $(word 3,
   ↪  $^) -O $@
```

```
# FILE:./chimpaln/Makefile


DATADIR = ../../

VCFIN = $(DATADIR)/chr1.vcf.gz
BAMIN = ../2020-09-13/chr1_only_confident.fixmate.sorted.bam
BEDIN = $(DATADIR)/chr1_confident.bed.gz
REFIN = $(DATADIR)/chr1.renamed.fa

#this bam has the quality scores reverted since
#NGM can't handle OQ tags
PROCESSEDBAM = ../2020-09-13/variants/raw.bam
CHIMPBAM = chimpaln.bam
CHIMPSKIP = chimpaln.vcf.gz
CHIMPREF = GCF_002880755.1_Clint_PTRv2_genomic.fa

SHELL=bash

.PHONY: all

all: chimpaln.recal.after.pdf chimpaln.recal.after.csv
↪  chimpaln_30.recal.after.pdf chimpaln_30.recal.after.csv

.DEFAULT: all

.PRECIOUS:

.SECONDARY:

#index helpers
%.bed.gz.tbi: %.bed.gz
  tabix -p bed $<

%.vcf.gz.tbi: %.vcf.gz
  bcftools index -t $<

%.bam.bai: %.bam
  samtools index $<

%.fa.fai: %.fa
  samtools faidx $<

%.dict: %.fa
  picard CreateSequenceDictionary R=$< O=$@
```

```
# --- Realign reads to chimp ref and get skips ---
collated_reads.bam: $(PROCESSEDBAM)
  samtools collate -f -u -@4 -o $@ $<

rglines.txt: collated_reads.bam
  samtools view -H collated_reads.bam | grep ^@RG > $@

rgargs.txt: rglines.txt
  cat $< | sed 's/^@RG\t// ; y/\t/ / ;
  ↪  s/\([[:upper:]]\{2\}\):/--rg-\L\1=/g' > $@

rgids.txt: rglines.txt
  cat $< | sed 's/^@RG\t// ; s/ID:\([^[:space:]]*\)\t.*/\1/g' > $@

$(CHIMPBAM): $(CHIMPREF) rgids.txt rgargs.txt collated_reads.bam
  samtools split -f "%!.%." -@8 $(word 4, $^) && \
  parallel --link 'ngm --no-progress -r $< -p -q {1}.bam {2} |
  ↪  samtools sort -m 2G -O bam -o {1}.chimpaln.bam - ' :::: $(word
  ↪  2, $^) :::: $(word 3, $^) && \
  cat $(word 2, $^) | sed 's/$$/.chimpaln.bam/' | xargs samtools
  ↪  merge -@8 -c -p $@

$(CHIMPSKIP): $(CHIMPBAM) $(CHIMPREF) $(CHIMPBAM).bai $(patsubst
↪  %.fa,%.dict,$(CHIMPREF)) $(CHIMPREF).fai
  gatk HaplotypeCaller -R $(word 2, $^) -I $< -stand-call-conf 50
  ↪  --native-pair-hmm-threads 8 -O $@

#use -stand-call-conf 30 instead of 50 or 40 (default)
chimpaln_30.vcf.gz: $(CHIMPBAM) $(CHIMPREF) $(CHIMPBAM).bai
↪  $(patsubst %.fa,%.dict,$(CHIMPREF)) $(CHIMPREF).fai
  gatk HaplotypeCaller -R $(word 2, $^) -I $< -stand-call-conf 30
  ↪  --native-pair-hmm-threads 8 -O $@

# --- Do calibration ---
chimpaln.recal.txt: $(CHIMPBAM) $(CHIMPREF) $(CHIMPSKIP)
↪  $(CHIMPBAM).bai
  gatk BaseRecalibrator -I $< -R $(word 2, $^) --known-sites $(word
  ↪  3, $^) -O $@

chimpaln_30.recal.txt: $(CHIMPBAM) $(CHIMPREF) chimpaln_30.vcf.gz
↪  $(CHIMPBAM).bai
  gatk BaseRecalibrator -I $< -R $(word 2, $^) --known-sites $(word
  ↪  3, $^) -O $@

# Recalibrate the alignment to human using
# the model from the Chimp alignment
%.recal.bam: $(BAMIN) $(REFIN) %.recal.txt
```

```
  gatk ApplyBQSR -I $< -R $(word 2, $^) --bqsr-recal-file $(word 3,
  ↪  $^) --use-original-qualities -O $@

# --- Benchmark with GATK ---
# Use actual positives for this test.
%.recal.after.txt: %.recal.bam $(REFIN) $(BEDIN) $(VCFIN)
↪  $(VCFIN).tbi
  gatk BaseRecalibrator -I $< -R $(word 2, $^) -L $(word 3, $^)
  ↪  --known-sites $(word 4, $^) -O $@

%.recal.after.pdf: %.recal.txt %.recal.after.txt
  gatk AnalyzeCovariates -before $< -after $(word 2, $^) -plots $@

%.recal.after.csv: %.recal.txt %.recal.after.txt
  gatk AnalyzeCovariates -before $< -after $(word 2, $^) -csv $@
```

E.2 Code to Generate Plots

```
#!/usr/bin/env Rscript
library(tidyverse)
library(colorblindr)


# --- Functions to get file paths ---

#return the data directory appended to path
get_data_dir <- function(path){
  return(paste0("../data/", path))
}

#return file name from stems where subpath
#is a dir in the data directory and suffix
#is the part after the stem.
#[../data/subpath/stem1suffix, ../data/subpath/stem2suffix, ...]
get_file <- function(stem, subpath, suffix){
  dir <- paste0(subpath, paste0('/', stem))
  return(paste0(get_data_dir(dir), suffix))
}

#return ../data/subpath/stem.recal.after.csv,
# the default is ../data/stem.recal.after.csv
get_gatk_csv_file <- function(stem, subpath = '.'){
  return(get_file(stem, subpath, ".recal.after.csv"))
}

# --- Utility Functions ---

#turn a probability to a phred-scaled quality score
```

```
p_to_q <- function(p, maxscore = 43){
  return(if_else(p != 0, floor(-10*log10(p)), maxscore))
}

# --- Dataframe Manipulation ---

#Gets only QualityScore and After rows in the dataframe
#and changes CovariateValue title to PredictedQuality
get_quality_rows <- function(df){
  df %>%
    filter(CovariateName == 'QualityScore' & Recalibration ==
    ↪  'After') %>%
    rename(PredictedQuality = CovariateValue)
}

#Calculate an RMSE column, which is the square root of the average
↪  squared
# difference between actual and predicted quality
get_RMSE <- function(df){
  df %>%
    mutate(RMSE = sqrt(mean((ActualQuality - PredictedQuality)^2)))
}

#Returns a df with covariatevalue and reportedquality columns
average_over_rgs_fnr <- function(df){
  df %>%
    get_quality_rows() %>%
    mutate_at(vars(FalseNegativeRate, PredictedQuality), as.numeric)
    ↪  %>%
    group_by(FalseNegativeRate, PredictedQuality) %>%
    summarize(ActualQuality = p_to_q(sum(Errors)/sum(Observations)))
    ↪  %>%
    get_RMSE()
}

average_over_rgs_fpr <- function(df){
  df %>%
    get_quality_rows() %>%
    mutate_at(vars(FalsePositiveRate, PredictedQuality), as.numeric)
    ↪  %>%
    group_by(FalsePositiveRate, PredictedQuality) %>%
    summarize(ActualQuality = p_to_q(sum(Errors)/sum(Observations)))
    ↪  %>%
    get_RMSE()
}

average_over_rgs_fnrfpr <- function(df){
```

```
  df %>%
    get_quality_rows() %>%
    mutate_at(vars(FalsePositiveRate, FalseNegativeRate,
     ↪  PredictedQuality),
              as.numeric) %>%
    group_by(FalseNegativeRate, FalsePositiveRate, PredictedQuality)
     ↪  %>%
    summarize(ActualQuality = p_to_q(sum(Errors)/sum(Observations)))
     ↪  %>%
    get_RMSE()
}

extract_rmse <- function(df){
  df %>%
    summarize(RMSE = unique(RMSE))
}

average_over_rgs_comparison <- function(df){
  df %>%
    mutate_at(vars(PredictedQuality), as.numeric) %>%
    group_by(CalibrationMethod, PredictedQuality) %>%
    summarize(ActualQuality = p_to_q(sum(Errors)/sum(Observations)))
     ↪  %>%
    get_RMSE()
}

# --- Importing CSVs ---

import_fnr_csvs <- function(fnr){
  csv_files <- get_gatk_csv_file(fnr, subpath = 'fnr')
  dfs <- map(csv_files, read_csv)
  names(dfs) <- fnr
  df <- bind_rows(dfs, .id = 'FalseNegativeRate')
  return(df)
}

import_fpr_csvs <- function(fpr){
  csv_files <- get_gatk_csv_file(fpr, subpath = 'fpr')
  dfs <- map(csv_files, read_csv)
  names(dfs) <- fpr
  df <- bind_rows(dfs, .id = 'FalsePositiveRate')
  return(df)
}

import_fnrfpr_csvs <- function(fnr, fpr){
  stems <- unlist(map(fnr, paste0, '_', fpr))
  csv_files <- get_gatk_csv_file(stems, subpath = 'fnrfpr')
```

```
  dfs <- map(csv_files, read_csv)
  names(dfs) <- stems
  df <- bind_rows(dfs, .id = 'FNR_FPR') %>%
    separate(FNR_FPR, into =
    ↪  c("FalseNegativeRate","FalsePositiveRate"),
            sep = "_", convert = TRUE)
}

import_comparison_csvs <- function(){
  kbbq_comparison_df <- get_gatk_csv_file('kbbq') %>%
    read_csv() %>%
    filter(CovariateName == 'QualityScore' & Recalibration == 'After')
  gatk_comparison_df <- get_gatk_csv_file('0', 'fnr') %>%
    read_csv() %>%
    filter(CovariateName == 'QualityScore' & Recalibration == 'After')
  raw_comparison_df <- get_gatk_csv_file('0', 'fnr') %>%
    read_csv() %>%
    filter(CovariateName == 'QualityScore' & Recalibration ==
    ↪  'Before')
  chimp_comparison_df <- get_gatk_csv_file('chimpaln') %>%
    read_csv() %>%
    filter(CovariateName == 'QualityScore' & Recalibration == 'After')
  comparison_dfs <- list( KBBQ = kbbq_comparison_df,
                          GATK = gatk_comparison_df,
                          Raw = raw_comparison_df,
                          "Chimp-GATK" = chimp_comparison_df)
  comparison_df <- bind_rows(comparison_dfs, .id =
  ↪  'CalibrationMethod') %>%
    rename(PredictedQuality = CovariateValue)
  comparison_df
}

# --- Plotting Data ---

plot_fnr_csvs <- function(gatk_df){
  ggplot(gatk_df, aes(PredictedQuality, ActualQuality)) +
    geom_abline(slope = 1, intercept = 0) +
    geom_point(aes(color = factor(FalseNegativeRate)), size = 2) +
    geom_line(aes(color = factor(FalseNegativeRate)), size = 1) +
    # scale_color_brewer('Known Sites\nFalse Negative\nRate',
    ↪  palette = 'PRGn') +
    scale_color_viridis_d('Known Sites\nFalse Negative\nRate') +
    scale_x_continuous("Predicted Quality") +
    scale_y_continuous("Actual Quality") +
    ggtitle('False Negative Variants Cause Underconfidence') +
    coord_fixed(ratio = 1)
}
```

```
plot_fpr_csvs <- function(gatk_df){
  ggplot(gatk_df, aes(PredictedQuality, ActualQuality)) +
    geom_abline(slope = 1, intercept = 0) +
    geom_point(aes(color = factor(FalsePositiveRate)), size = 2) +
    geom_line(aes(color = factor(FalsePositiveRate)), size = 1) +
    # scale_color_brewer('Known Sites\nFalse Positive\nRate',
    ↪  palette = 'PRGn') +
    scale_color_viridis_d('Known Sites\nFalse Positive\nRate') +
    scale_x_continuous("Predicted Quality") +
    scale_y_continuous("Actual Quality") +
    ggtitle('False Positive Variants Alone Have Little Effect') +
    coord_fixed(ratio = 1)
}

plot_comparison_dfs <- function(comparison_dfs){
  ggplot(comparison_dfs, aes(PredictedQuality, ActualQuality)) +
    geom_abline(slope = 1, intercept = 0) +
    geom_point(aes(color = CalibrationMethod), size = 2) +
    geom_line(aes(color = CalibrationMethod), size = 1) +
    # scale_color_brewer('Calibration\nMethod', palette = 'Dark2') +
    # scale_color_viridis_d('Calibration\nMethod') +
    scale_x_continuous("Predicted Quality") +
    scale_y_continuous('Actual Quality') +
    ggtitle('GATK and KBBQ Perform Similarly') +
    coord_fixed(ratio = 1) +
    scale_color_OkabeIto(name = '', use_black = T, order =
    ↪  c(1,2,3,7,8), drop = FALSE)
}

abbreviate_rate <- function(str){
  str_replace_all(str,
                  c("FalsePositiveRate" = "FPR", "FalseNegativeRate"
                  ↪  = "FNR"))
}

rate_labeller <- labeller(
  .cols = function(str){str_replace(str,"^0$","FNR: 0")},
  .rows = function(str){str_replace(str,"^0$","FPR: 0")}
)

plot_fnrfpr_csvs <- function(gatk_df){
  #remove the largest 5 values; these are outliers
  scale_lim <- gatk_df$RMSE %>%
    sort() %>% unique() %>% .[1:(length(.)-5)] %>% range()
  scale_vals <- function(x, ...){
    y <- scales::rescale(x, to = c(0,1), from = scale_lim)
```

```
    y[y>1] <- 1
    y
  }

  ggplot(gatk_df, aes(PredictedQuality, ActualQuality)) +
    geom_abline(slope = 1, intercept = 0) +
    geom_point(aes(color = RMSE), size = 1) +
    geom_line(aes(color = RMSE), size = 1) +
    facet_grid(cols = vars(FalseNegativeRate), rows =
    ↪  vars(FalsePositiveRate),
               switch = "both",
               as.table = FALSE, labeller = rate_labeller) +
    scale_color_viridis_c('RMSE', option = "viridis", rescaler =
    ↪  scale_vals,
                          limits = scale_lim, oob = scales::squish) +
    scale_x_continuous("Predicted Quality") +
    scale_y_continuous("Actual Quality") +
    ggtitle('Combined Effect On Calibration') +
    theme(axis.text = element_text(size = rel(.4)),
          strip.text = element_text(size = rel(.6)),
          axis.text.x = element_text(angle = 45, hjust = 1),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank())
}

plot_fnrfpr_heatmap <- function(gatk_df){
  #remove the largest 5 values; these are outliers
  scale_lim <- gatk_df$RMSE %>%
    sort() %>% unique() %>% .[1:(length(.)-5)] %>% range()
  scale_vals <- function(x, ...){
    y <- scales::rescale(x, to = c(0,1), from = scale_lim)
    y[y>1] <- 1
    y
  }

  ggplot(gatk_df, aes(FalseNegativeRate, FalsePositiveRate)) +
    geom_raster(aes(fill = RMSE)) +
    scale_fill_viridis_c('RMSE', option = "viridis", rescaler =
    ↪  scale_vals,
                          limits = scale_lim, oob = scales::squish) +
    xlab("False Negative Rate") +
    ylab("False Positive Rate") +
    ggtitle('Combined Effect on RMSE') +
    coord_fixed(ratio = 1)
}
```

```
# --- Main ---

fnr <- c(0, 20, 40, 60, 80, 100)
fpr <- c(0, 20, 40, 60, 80, 100)

theme_set(theme_minimal(base_size = 22, base_family = 'Times') +
            theme(plot.margin = margin(0,0,0,0))
)

fnr_df <- average_over_rgs_fnr(import_fnr_csvs(fnr))
pdf('../figures/fnr.pdf', width = 9, height = 7)
plot_fnr_csvs(fnr_df)
dev.off()


fpr_df <- average_over_rgs_fpr(import_fpr_csvs(fpr))
pdf('../figures/fpr.pdf', width = 9, height = 7)
plot_fpr_csvs(fpr_df)
dev.off()

fnrfpr_df <- average_over_rgs_fnrfpr(import_fnrfpr_csvs(fnr, fpr))
pdf('../figures/fnrfpr.pdf', width = 7, height = 7)
plot_fnrfpr_csvs(fnrfpr_df)
dev.off()

pdf('../figures/fnrfpr_heatmap.pdf', width = 9, height = 7)
plot_fnrfpr_heatmap(fnrfpr_df)
dev.off()

comparison_df <- average_over_rgs_comparison(import_comparison_csvs())
pdf('../figures/comparison.pdf', width = 9, height = 7)
plot_comparison_dfs(comparison_df)
dev.off()

  # --- Simulated Data ---
ngm_before_df <- get_gatk_csv_file('ngm') %>% read_csv() %>%
 ↪  filter(CovariateName == 'QualityScore' & Recalibration ==
 ↪  'Before')
ngm_after_df <- get_gatk_csv_file('ngm') %>% read_csv() %>%
 ↪  filter(CovariateName == 'QualityScore' & Recalibration == 'After')
kbbq_ngm_df <- get_gatk_csv_file('kbbq-ngm') %>% read_csv() %>%
 ↪  filter(CovariateName == 'QualityScore' & Recalibration == 'After')
bootstrap_df <- get_gatk_csv_file('initial-calls') %>% read_csv() %>%
 ↪  filter(CovariateName == 'QualityScore' & Recalibration == 'After')


#calibration
```

```
ngm_df <- bind_rows(list(Raw = ngm_before_df, GATK = ngm_after_df,
↪  "Initial-calls" = bootstrap_df, KBBQ = kbbq_ngm_df), .id =
↪  'CalibrationMethod') %>%
  rename(PredictedQuality = CovariateValue) %>%
  average_over_rgs_comparison() %>%
  mutate(CalibrationMethod = factor(CalibrationMethod, levels =
   ↪  c("Raw","GATK","Initial-calls","KBBQ")))

pdf('../figures/sim_comparison.pdf', width = 9, height = 7)
ggplot(ngm_df, aes(PredictedQuality,ActualQuality, color =
↪  CalibrationMethod)) +
  geom_abline(slope = 1, intercept = 0) +
  geom_point(size = 2) +
  geom_line(size = 1) +
  scale_color_OkabeIto(name = 'Calibration', use_black = T, drop =
   ↪  FALSE) +
  xlab("Predicted Quality") +
  ylab("Actual Quality") +
  ggtitle("Calibration of Simulated Data")
dev.off()

#histogram
freq_df <- bind_rows(
    list(Raw = ngm_before_df, GATK = ngm_after_df, "Initial-calls" =
     ↪  bootstrap_df, KBBQ = kbbq_ngm_df),
    .id = 'CalibrationMethod') %>%
  rename(PredictedQuality = CovariateValue) %>%
  mutate_at(vars(PredictedQuality), as.numeric) %>%
  group_by(CalibrationMethod, PredictedQuality) %>%
  mutate(CalibrationMethod = factor(CalibrationMethod, levels =
   ↪  c("Raw","GATK","Initial-calls","KBBQ")))

pdf('../figures/sim_qual_counts.pdf', width = 9, height = 7)
ggplot(freq_df, aes(PredictedQuality, Observations, color =
↪  CalibrationMethod)) +
  geom_point(size = 1.5) +
  geom_line(size = 1) +
  scale_color_OkabeIto(name = 'Calibration', use_black = T, drop =
   ↪  FALSE) +
  xlab("Predicted Quality") +
  ylab("Actual Quality") +
  ggtitle("Count of Each Quality Score")
dev.off()


# --- Print Tables ---
fpr_rmse <- fpr_df %>% extract_rmse()
```

```
fnr_rmse <- fnr_df %>% extract_rmse()
fnrfpr_rmse <- fnrfpr_df %>% extract_rmse()
comparison_rmse <- comparison_df %>% extract_rmse()
sims_rmse <- ngm_df %>% extract_rmse()

write_tsv(fpr_rmse, "../tables/fpr_rmse.txt")
write_tsv(fnr_rmse, "../tables/fnr_rmse.txt")
write_tsv(fnrfpr_rmse, "../tables/fnrfpr_rmse.txt")
write_tsv(comparison_rmse, "../tables/comparison_rmse.txt")
write_tsv(sims_rmse, "../tables/sims_rmse.txt")
```

```
library(tidyverse)
library(colorblindr)

rg_val <- 24
assigned_q_val <- 30
covs <- c("Context", "Cycle")
color_levels <- c("Read Group", "Assigned Q", "Context", "Cycle",
↪  "Recalibrated")

data_df <- data.frame(
  group = factor(c(1, 1, 2, 2)),
  x = c(.9,1.1,1.9,2.1),
  covariate = factor(c(covs,covs), levels = color_levels),
  quality = c(31, 28, 25, 32),
  rg_qual = rg_val,
  assigned_qual = assigned_q_val
)

label_df <- data.frame(label = c("Read Group Estimate","Assigned Q
↪  Estimate for this RG"),
                       # group = factor(c(3)),
                       color = c("Read Group","Assigned Q"),
                       quality = c(rg_val, assigned_q_val)
                       )

label2_df <- data.frame(label = c("Context Estimate for this RG and
↪  Q","Cycle Estimate for this RG and Q"),
                        color = c("Context","Cycle"),
                        quality = c(25,31.5))

adding_df <- data.frame(label = c("30 + 1 - 2","30 + 2 - 5"),
                        color = c("black","black"),
                        x = c(1,2),
                        y = c(29, 27))

pdf("./figures/recalibration_explainer.pdf", width = 11, height = 7)
#show how to go from assigned Q to a new quality score
```

```
data_df %>% ggplot(aes(group, quality, family = 'Times')) +
  # geom_pointrange(aes(ymin = assigned_qual, ymax = quality), size
  ↪  = 1) +
  geom_segment(aes(x = x, xend = x, y = assigned_q_val, yend =
  ↪  quality, color = covariate), size = 1, arrow = arrow(length =
  ↪  unit(.15,"inches"))) +
  geom_text(mapping = aes(x = 3.5, quality+.25, color = color, label
  ↪  = label, hjust = 1),
            data = label_df, show.legend = F, size = 6) +
  geom_text(mapping = aes(x = 3.5, quality+.25, color = color, label
  ↪  = label, hjust = 1), data = label2_df, show.legend = FALSE,
  ↪  size = 6) +
  # geom_text(mapping = aes(x = 3.5, y = 25.25, color = "Context",
  ↪  label = "Context Estimate", hjust = 1), data = data.frame()) +
  # geom_text(mapping = aes(x = 3.5, y = 31.75, color = "Cycle",
  ↪  label = "Cycle Estimate", hjust = 1)) +
  geom_hline(aes(yintercept = quality, color = color), data =
  ↪  label_df, size = 1) +
  geom_segment(aes(x = .5, xend = .5, y = rg_qual, yend =
  ↪  assigned_qual,
                   color = "Read Group"),
               arrow = arrow(length = unit(.15,"inches")),
                ↪  show.legend = FALSE, size = 1) +
  # geom_curve(aes(x = .9, xend = .95, y = assigned_qual, yend =
  ↪  29.05),
  #              arrow = arrow(length = unit(.15,"inches")),
  ↪  show.legend = FALSE, curvature = .5, size = 1, color =
  ↪  "black") +
  # geom_curve(aes(x = 1.9, xend = 1.95, y = assigned_qual, yend =
  ↪  27.05),
  #              arrow = arrow(length = unit(.15,"inches")),
  ↪  show.legend = FALSE, curvature = .5, size = 1, color =
  ↪  "black") +
  geom_segment(aes(x = 1, xend = 1, y = assigned_qual, yend = 29.075),
               arrow = arrow(length = unit(.15,"inches")), show.legend
                ↪  = FALSE, size = 1, color = "black") +
  geom_segment(aes(x = 2, xend = 2, y = assigned_qual, yend = 27.075),
               arrow = arrow(length = unit(.15,"inches")), size = 1,
                ↪  color = "black") +
  geom_point(x = 1, y = 29, color = "black", size = 3) +
  geom_point(x = 2, y = 27, color = "black", size = 3) +
  geom_text(mapping = aes(x = x + .15, y = y, label = label), color =
  ↪  "black", data = adding_df, hjust = 0, show.legend = FALSE, size
  ↪  = 6) +
  xlab("Base") + ggtitle("Two Bases Assigned The Same Quality") +
  ↪  ylab("Quality") +
```

```
  scale_color_OkabeIto(name = '', use_black = T, order =
  ↪  c(1,2,3,7,8), drop = FALSE)

dev.off()
```

APPENDIX F

KBBQ PROGRAM CODE

```
// FILE:/home/adam/code/kbbq/include/kbbq/kseq.hh


#ifndef __KBBQ_KSEQ_H
#define __KBBQ_KSEQ_H

#include <htslib/bgzf.h>
#include <htslib/kseq.h>

namespace kseq{
  KSEQ_INIT(BGZF*, bgzf_read)
}

#endif
// FILE:/home/adam/code/kbbq/include/kbbq/readutils.hh


#ifndef READUTILS_H
#define READUTILS_H

#include <vector>
#include <unordered_map>
#include <string>
#include <errno.h>
//
#include <htslib/sam.h>
#include <htslib/kseq.h>
#include <htslib/bgzf.h>
#include "bloom.hh"
#include "covariateutils.hh"
#include "kseq.hh"

#define INFER_ERROR_BAD_QUAL 2

//fwd declare
namespace covariateutils{
  struct dq_t;
}

namespace readutils{

  static int8_t complement[16] = { 0, 8, 4, 12, 2, 10, 6, 14, 1, 9,
  ↪  5, 13, 3, 11, 7, 15 };

  // int correction_len(const bloom::bloomary_t& t, int k);
```

```
//get read sequence as a string in the forward orientation
inline std::string bam_seq_str(bam1_t* bamrecord){
  std::string seq;
  char* s = (char*)bam_get_seq(bamrecord);
  for(size_t i = 0; i < bamrecord->core.l_qseq; ++i){
    seq.push_back(bam_is_rev(bamrecord) ?
      seq_nt16_str[seq_nt16_table['0' + 3-seq_nt16_int[bam_seqi(s,
      ↪  i)]]] :
      seq_nt16_str[bam_seqi(s, i)]);
  }
  if(bam_is_rev(bamrecord)){
    std::reverse(seq.begin(), seq.end());
  }
  return seq;
}

class CReadData{
  public:
    static std::unordered_map<std::string, std::string> rg_to_pu;
    static std::unordered_map<std::string, int> rg_to_int;
    CReadData(){}
    CReadData(bam1_t* bamrecord, bool use_oq = false);
    // hello?
    CReadData(kseq::kseq_t* fastqrecord, std::string rg = "", int
    ↪  second = 2, std::string namedelimiter = "_");
    std::string seq;
    std::vector<uint8_t> qual;
    std::vector<bool> skips;
    std::string name;
    std::string rg;
    bool second;
    std::vector<bool> errors;
    std::string str_qual();
    std::string canonical_name();
    inline int get_rg_int() const{return
    ↪  this->rg_to_int[this->rg];}
    inline std::string get_pu() const{return
    ↪  this->rg_to_pu[this->rg];}
    std::vector<bool> not_skipped_errors() const;
    //fill errors attribute given sampled kmers and thresholds.
    void infer_read_errors(const bloom::Bloom& b, const
    ↪  std::vector<int>& thresholds, int k);
    //fix one error and return the index of the fixed base;
    ↪  std::string::npos if no fixes are found
    size_t correct_one(const bloom::Bloom& t, int k);
    static void load_rgs_from_bamfile(bam_hdr_t* header);
```

```
      //fill errors attribute given trusted kmers
      std::vector<bool> get_errors(const bloom::Bloom& trusted, int
       ↪  k, int minqual = 6, bool first_call = true);
      std::vector<uint8_t> recalibrate(const covariateutils::dq_t&
       ↪  dqs, int minqual = 6) const;
      CReadData substr(size_t pos = 0, size_t count =
       ↪  std::string::npos) const;

  };
}

#endif

// FILE:/home/adam/code/kbbq/include/kbbq/gatkreport.hh


#ifndef KBBQ_GATKREPORT_HH
#define KBBQ_GATKREPORT_HH

#include <iostream>
#include <iomanip>
#include <vector>
#include <fstream>
#include <sstream>

namespace gatkreport{

enum column_type{STRING = 0, FLOAT, INT};

class TableValue{
private:
  column_type type;
  union{
    long double f;
    unsigned long long i;
  }
  std::string s;
public:
  TableValue(): type(), f(), s() {}
  TableValue(unsigned long long i): TableValue() {this->set(i);}
  TableValue(long double f): TableValue() {this->set(f);}
  TableValue(std::string s): TableValue() {this->set(s);}

  inline column_type current_type(){return type;}
  template<typename T> T get();
  template<>
  unsigned long long get<unsigned long long>(){
```

```
    return type == INT ? i : 0;
  }
  template<>
  long double get<long double>(){
    return type == FLOAT ? f : 0;
  }
  template<>
  std::string get<std::string>(){
    return type == STRING ? s : "";
  }
  inline TableValue& set(unsigned long long i){this->i = i;
  ↪  this->type = INT; return this;}
  inline TableValue& set(long double f){this->f = f; this->type =
  ↪  FLOAT; return this;}
  inline TableValue& set(std::string s){this->s = s; this->type =
  ↪  STRING; return this;}
};

class TableRow{
private:
  std::vector<TableValue> columns;
  std::vector<column_type> types;
  std::vector<std::string> headers;
public:
  TableRow(): columns(), types() {}
  TableRow(const std::string& line, const std::vector<column_type>&
  ↪  t, const std::vector<std::string>& h):
    types(t), columns(), headers(h)
  {
    std::istringstream is(line);
    std::string token;
    for(column_type t : types){
      is >> token;
      switch(t){
        case STRING: columns.emplace_back(token);
          break;
        case INT: columns.emplace_back(std::stoull(token));
          break;
        case FLOAT: columns.emplace_back(std::stold(token));
          break;
      }
    }
  }
  //space-delimited values in the row.
  inline explicit operator std::string() const {
    std::string ret;
    for(size_t i = 0; i < columns.size(); ++i){
```

```
      std::string column_val = "";
      switch(types[i]){
        case FLOAT: long double val = columns[i].get<long double>();
          std::string format = headers[i] == "Errors" ? "%.2f" :
          ↪  "%.4f"
          int outsize = std::snprintf(nullptr, 0, format.c_str(),
          ↪  val);
          column_val.resize(outsize + 1);
          std::snprintf(&column_val[0], column_val.size(),
          ↪  format.c_str(), val);
          break;
        case INT: column_val =
        ↪  std::to_string(columns[i].get<unsigned long long>());
          break
        case STRING:
        default:
          column_val = columns[i].get<std::string>;
      }
      ret += column_val + " ";
    }
    ret.pop_back(); //remove trailing space
    return ret;
  }
  template <typename T>
  T get(size_t i){return columns[i].get<T>();}
}

class GATKTable{
private:
  std::vector<TableRow> rows;
  std::string title;
  std::string description;
  std::vector<std::string> headers;
  std::vector<column_type> types;
  std::vector<size_t> widths;
  /*
  const std::map<std::string, std::string> precision = {
    {"EmpiricalQuality", ".4"},
    {"EstimatedQReported",".4"},
    {"Errors",".2"}
  };
  */
public:
  GATKTable(): rows(), title(), description(), headers(), types() {}
  GATKTable(std::string tablestr):
    rows(), title(), description(), headers(), types()
  {
```

```
std::istringstream in(tablestr);
std::string headerline;
std::string headerprefix = "#:GATKTable:"
std::getline(in, headerline); //process first line
if(headerline.substr(0,headerprefix.length()) != headerprefix){
  throw std::invalid_argument("Error: Unable to parse first line
  ↪  of table. " +
    "Ensure input is a valid GATKTable.");
}
headerline.erase(0,headerprefix.length());
std::istringstream hdrin(headerline);
std::string token;
std::getline(hdrin, token, ':');
size_t ncols = std::stoull(token);
std::getline(hdrin, token, ':');
size_t nrows = std::stoull(token);

while(std::getline(hdrin, token, ':')){
  switch(token.pop_back()){
    case 'd': types.push_back(INT);
      break;
    case 'f': types.push_back(FLOAT);
      break;
    case ';': //do nothing
      break;
    case 's':
    default: types.push_back(STRING);
  }
}
if(types.size() != ncols){
  throw std::invalid_argument("Error: Number of types doesn't
  ↪  match header!" +
    "Ensure the type string includes all types.\n");
}

std::getline(in, headerline); //process second line; it's the
↪  title and description
if(headerline.substr(0,headerprefix.length()) != headerprefix){
  throw std::invalid_argument("Error: Unable to parse second line
  ↪  of table. " +
    "Ensure input is a valid GATKTable.");
}
headerline.erase(0,headerprefix.length());
hdrin.str(headerline);
std::getline(hdrin, title, ':');
std::getline(hdrin, description, ':');
```

```
  std::getline(in, headerline); //process the 3rd line; it's the
  ↪  column headers
  hdrin.str(headerline);
  std::copy(std::istream_iterator<std::string>(hdrin),
      std::istream_iterator<std::string>(),
      std::back_inserter(headers));
  if(types.size() != headers.size()){
    throw std::invalid_argument(
      "Error: Number of headers doesn't match the number of
      ↪  columns.\n");
  }
  while(std::getline(in, token)){
    rows.emplace_back(token, types, headers);
  }
  if(rows.size() != nrows){
    throw std::invalid_argument(
      "Error: Number of stated rows doesn't match number of actual
      ↪  rows.\n");
  }
}
//TODO: ctor, default and std::string
//return the header string without a trailing newline
inline std::string headerstring() const{
  std::string headerstr = "#:GATKTable:" +
  ↪  std::to_string(types.size()) +
    ":" + std::to_string(rows.size()) + ":";
  for(size_t i = 0; i < types.size(); ++i){
    column_type t = types[i];
    std::string h = headers[i];
    std::string colcode = "";
    switch(t){
      case STRING: colcode = "%s";
        break;
      case INT: colcode = "%d";
        break;
      case FLOAT: colcode = h == "Errors" ? "%.2f" : "%.4f";
        break;
    }
    headerstr += colcode + ":";
  }
  return headerstr + ";";
}

//the title string without a trailing newline
inline std::string titlestring() const{
  return "#:GATKTable:" + title + ":" + description;
}
```

```
  inline std::vector<size_t> get_col_widths() const{
    std::vector<size_t> widths{headers.size(),0};
    std::transform(headers.begin(), headers.end(), widths.begin(),
      [](std::string str) -> size_t {return str.length();});
    for(size_t i = 0; i < rows.size(); ++i){
      std::istringstream rowin = rows[i];
      for(size_t j = 0; j < widths.size(); ++j){
        std::string colstr;
        rowin >> colstr;
        if(colstr.length() > widths[j]){
          widths[j] = colstr.length();
        }
      }
    }
    return widths;
  }

  inline explicit operator std::string() const {
    std::string ret = this->headerstring() + "\n";
    ret += this->titlestring() + "\n";
    std::ostringstream os();
    std::vector<size_t> widths = this->get_col_widths();
    for(size_t i = 0; i < widths.size(); ++i){
      os << std::setw(widths[i]) << headers[i] << "  ";
    }
    ret += os.str();
    ret.erase(ret.end()-2);
    os.str(""); //reset output stream
    std::istringstream is("");
    std::string token;
    for(const std::string& s : rows){
      is.str(s);
      is >> token;
      os << std::setw(widths[0]) << token;
      for(size_t i = 1; i < widths.size(); ++i){
        is >> token;
        os << "  " << std::setw(widths[i]) << token;
      }
      os << "\n";
    }
    ret += os.str();
    return ret;
  }
  TableRow& get(size_t i){return &rows[i];}
};
```

```
class GATKReport{
private:
  std::vector<GATKTable> tables;
  std::string version;
public:
  GATKReport(){}
  GATKReport(const std::vector<GATKTable>& tables): tables(tables) {}
  GATKReport(const std::string& filename): tables(), version(){
    std::ifstream inf(filename);
    std::string tablestr;
    std::string headerline;
    std::string versionprefix = "#:GATKReport.v";
    std::getline(inf, headerline);
    if(headerline.substr(0,versionprefix.length()) != versionprefix){
      throw std::invalid_argument("Error: Unable to parse first line
      ↪  of input file " +
        filename + " ; Ensure it is a valid GATKReport.");
    }
    headerline.erase(0,versionprefix.length());
    size_t colon_pos = headerline.find(':');
    version = headerline.substr(0, colon_pos);
    size_t ntables = std::stoull(headerline.substr(colon_pos+1,
    ↪  std::string::npos));
    for(std::string line; std::getline(inf, line);){
      if(line.empty()){ //blank line delimits tables
        tablestr.pop_back(); //get rid of ending newline
        tables.emplace_back(tablestr);
        tablestr.clear();
      } else {
        tablestr += line + '\n';
      }
    }
    if(ntables != tables.size()){
      throw std::invalid_argument("Found " + std::to_string(ntables)
      ↪  + " tables in " +
        filename + ", but " + std::to_string(tables.size()) " were
        ↪  declared." +
        "Ensure the file is not truncated and adjust the declared
        ↪  number of tables.");
    }
  }
  //return the header string without a trailing newline
  inline std::string headerstring() const{
    return "#:GATKReport.v" + version + ":" +
    ↪  std::to_string(tables.size());
  }
  inline explicit operator std::string() const {
```

```
    std::string ret = this->headerstring() + "\n";
    for(const std::string& s : tables){ret += s + "\n";}
    return ret;
  }
};

}

#endif
// FILE:/home/adam/code/kbbq/include/kbbq/bloom.hh


#ifndef KBBQ_BLOOM_HH
#define KBBQ_BLOOM_HH
#include <cstdint>
#include <utility>
#include <htslib/hts.h>
#include <minion.hpp>
#include <memory>
#include <functional>
#include <iostream>
#include "bloom_filter.hpp"
#include <stdexcept>
#include <immintrin.h>

#define PREFIXBITS 10
#define KBBQ_MAX_KMER 32

//the number of prefix hashes is 1<<PREFIXBITS - 1

namespace bloom{

class blocked_bloom_filter: public bloom_filter
{
protected:
  typedef unsigned char v32uqi __attribute__ ((__vector_size__
  ↪  (32))); //activate gcc vectorization
  typedef long long base_type;
  typedef base_type v4di __attribute__ ((__vector_size__ (32)));
  typedef v4di cell_type;
  static constexpr cell_type cell_type_zero = {0ULL,0ULL,0ULL,0ULL};
  ↪  //used to initialize
  typedef std::unique_ptr<cell_type,
  ↪  std::function<void(cell_type*)>> table_type;
  // typedef std::unique_ptr<unsigned char,
  ↪  std::function<void(unsigned char*)>> table_type;
public:
```

```
static const size_t block_size = 512; //512 bits = 64 bytes
table_type bit_table_;
//TODO: ensure table size is a multiple of block_size
blocked_bloom_filter(): bloom_filter(){}
blocked_bloom_filter(const bloom_parameters& p){
  projected_element_count_ = p.projected_element_count;
     inserted_element_count_ = 0;
     random_seed_ = (p.random_seed * 0xA5A5A5A5) + 1 ;
     desired_false_positive_probability_ =
     ↪  p.false_positive_probability;
    salt_count_ = std::max(p.optimal_parameters.number_of_hashes,
     ↪  2u);
  table_size_ = p.optimal_parameters.table_size;
  //ensure table fits a full block
  table_size_ += (table_size_ % block_size) != 0 ? block_size -
   ↪  (table_size_ % block_size) : 0;
  generate_unique_salt();
  void* ptr = 0;
  int ret = posix_memalign(&ptr, block_size / bits_per_char,
    table_size_ / bits_per_char);
  if(ret != 0){
    throw std::bad_alloc();
  }
  bit_table_ = table_type(static_cast<cell_type*>(ptr),
    [](cell_type* x){free(x);});
  std::uninitialized_fill_n(bit_table_.get(), table_size_ /
   ↪  bits_per_char / sizeof(cell_type),
    cell_type_zero);
}
//delete copy ctor
blocked_bloom_filter(const blocked_bloom_filter&) = delete;
//delete copy assignment
blocked_bloom_filter& operator=(const blocked_bloom_filter&) =
 ↪  delete;
//move ctor
blocked_bloom_filter(blocked_bloom_filter&& o){
  salt_count_ = std::move(o.salt_count_);
  table_size_ = std::move(o.table_size_);
  bit_table_ = std::move(o.bit_table_);
  salt_ = std::move(o.salt_);
  projected_element_count_ = std::move(o.projected_element_count_);
  inserted_element_count_ = std::move(o.inserted_element_count_);
  random_seed_ = std::move(o.random_seed_);
  desired_false_positive_probability_ =
   ↪  std::move(o.desired_false_positive_probability_);
}
```

```
//move function
inline blocked_bloom_filter& operator=(blocked_bloom_filter&& o){
  if(this != &o){
    salt_count_ = std::move(o.salt_count_);
    table_size_ = std::move(o.table_size_);
    bit_table_ = std::move(o.bit_table_);
    salt_ = std::move(o.salt_);
    projected_element_count_ =
    ↪  std::move(o.projected_element_count_);
    inserted_element_count_ = std::move(o.inserted_element_count_);
    random_seed_ = std::move(o.random_seed_);
    desired_false_positive_probability_ =
    ↪  std::move(o.desired_false_positive_probability_);
  }
  return *this;
}


inline virtual size_t num_blocks() const{
  assert((table_size_ % block_size) == 0);
  return table_size_ / block_size;
}

// inline virtual void compute_indices(const bloom_type& hash,
↪  std::size_t& bit_index, std::size_t& bit) const {
//   bit_index = hash % block_size; //which bit in the block?
//        bit       = bit_index % bits_per_char; //
// }

inline virtual size_t get_block(const bloom_type& hash) const{
  // how lighter does it
  // return (hash % (table_size_ - block_size + 1)) /
  ↪  bits_per_char;
  // we intead pick an aligned block.
  // hash index * sizeof(block) / sizeof(cell)
  return (hash % num_blocks()) * (block_size / bits_per_char) /
  ↪  (sizeof(cell_type));

}

//pick the correct vector inside the block and then the correct
//base type inside the vector.
inline virtual std::pair<size_t,size_t> get_vector_unit(const
↪  size_t& bit_index) const{
  return std::make_pair((bit_index / bits_per_char) /
  ↪  sizeof(cell_type),
```

```
    (bit_index / bits_per_char) %
     ↪  (sizeof(cell_type)/sizeof(base_type)));
}

inline virtual void insert(const unsigned char* key_begin, const
 ↪  size_t& length){
  size_t bit_index = 0;
  size_t bit = 0;
  //index in table with first byte of block
  size_t block_idx = get_block(hash_ap(key_begin, length,
   ↪  salt_[0]));
  cell_type* block = bit_table_.get() + block_idx;
  size_t vec, unit;
  for(size_t i = 1; i < salt_.size(); ++i){
    compute_indices(hash_ap(key_begin, length, salt_[i]),
     ↪  bit_index, bit);
    //the bit index is out of 512, the bit is out of 8.
    //we advance to the proper vector, then subscript to the
     ↪  proper byte
    std::tie(vec, unit) = get_vector_unit(bit_index);
    (*(block + vec))[unit] |= static_cast<base_type>(1) << bit;
  }
  ++inserted_element_count_;
}

template <typename T>
inline void insert(const T& t){
  insert(reinterpret_cast<const unsigned char*>(&t), sizeof(T));
}

inline virtual bool contains(const unsigned char* key_begin, const
 ↪  std::size_t length) const {
  size_t bit_index = 0;
  size_t bit = 0;
  //index in table with first byte of block
  size_t block_idx = get_block(hash_ap(key_begin, length,
   ↪  salt_[0]));
  cell_type* block = bit_table_.get() + block_idx;
  size_t vec, unit;
  for(size_t i = 1; i < salt_.size(); ++i){
    compute_indices(hash_ap(key_begin, length, salt_[i]),
     ↪  bit_index, bit);
    std::tie(vec, unit) = get_vector_unit(bit_index);
    if( ((*(block + vec))[unit] & static_cast<base_type>(1) <<
     ↪  bit) !=
      static_cast<base_type>(1) << bit)
    {
```

```
        return false;
      }
    }
    return true;
  }

  template <typename T>
  inline bool contains(const T& t) const {
    return contains(reinterpret_cast<const unsigned
    ↪  char*>(&t),static_cast<std::size_t>(sizeof(T)));
  }

  inline double effective_fpp() const {
    long double c = size() / element_count() ;
    long double lambda = block_size / c;
    long double fpp = 0;
    for(int i = 0; i < 3 * lambda; ++i){ //var = lambda, so 3*lambda
    ↪  should include most anything
      long double k = i;
      long double p_block = std::pow(lambda, k) * std::exp(-lambda)
      ↪  / std::tgammal(k+1);
      long double fpr_inner = std::pow(1.0 - std::exp(-1.0 *
      ↪  salt_.size() * i / block_size), 1.0 * salt_.size());
      fpp += p_block * fpr_inner;
    }
    return fpp;
  }

protected:
  inline virtual void compute_indices(const bloom_type& hash,
  ↪  std::size_t& bit_index, std::size_t& bit) const {
    bit_index = hash % block_size; //which bit in the block?
        bit       = bit_index % (bits_per_char * sizeof(base_type));
        ↪  //which bit in the base type?
  }
};

class pattern_blocked_bf: public blocked_bloom_filter
{
protected:
  typedef std::unique_ptr<cell_type,
  ↪  std::function<void(cell_type*)>> pattern_type;
  static const size_t num_patterns = 65536; //4MiB
  pattern_type patterns;
public:
  pattern_blocked_bf(): blocked_bloom_filter(){}
```

```
pattern_blocked_bf(const bloom_parameters& p):
↪  blocked_bloom_filter(p){
  void* ptr = 0;
  //we have num_patterns patterns, each with size block_size (in
  ↪  bits)
  int ret = posix_memalign(&ptr, block_size / bits_per_char,
    num_patterns * block_size / bits_per_char);
  if(ret != 0){
    throw std::bad_alloc();
  }
  patterns = pattern_type(static_cast<cell_type*>(ptr),
    [](cell_type* x){free(x);});
  std::uninitialized_fill_n(patterns.get(),
    num_patterns * block_size / bits_per_char / sizeof(cell_type),
    cell_type_zero);
  minion::Random rng;
  rng.Seed(random_seed_); //todo: check that seeding is proper for
  ↪  multiple rng instances
  //ie. the rng in kmersubsampler.
  std::uniform_int_distribution<> d(0, block_size-1);

  std::vector<size_t> possible_bits(block_size);
  std::iota(possible_bits.begin(), possible_bits.end(), 0);
  //begin with a fully shuffled array
  std::shuffle(possible_bits.begin(), possible_bits.end(), rng);
  for(size_t i = 0; i < num_patterns; ++i){
    //first index of the block containing the pattern
    size_t block_start_idx = i * (block_size / bits_per_char) /
    ↪  sizeof(cell_type);
    //sample salt_.size() bits
    for(int j = 0; j < salt_.size(); ++j){
      size_t sampled_bit = d(rng,
        std::uniform_int_distribution<>::param_type{j, block_size -
        ↪  1});
      std::swap(possible_bits[j], possible_bits[sampled_bit]);
    }
    //set the bits in the appropriate pattern
    size_t vec, unit;
    for(size_t j = 0; j < salt_.size(); ++j){
      size_t sampled_bit_number = possible_bits[j]; //the bit out
      ↪  of [0, 511]
      //the index of the correct vector and byte within the block
      std::tie(vec, unit) = get_vector_unit(sampled_bit_number);
      size_t sampled_bit_idx = block_start_idx + vec;
      //set the index of the bit within the char
```

```
      (*(patterns.get() + sampled_bit_idx))[unit] |=
      ↪  (static_cast<base_type>(1) << (sampled_bit_number %
      ↪  (sizeof(base_type) * bits_per_char)));
    }
  }
}
//delete copy ctor
pattern_blocked_bf(const pattern_blocked_bf&) = delete;
//delete copy assignment
pattern_blocked_bf& operator=(const pattern_blocked_bf&) = delete;
//move ctor
pattern_blocked_bf(pattern_blocked_bf&& o):
  blocked_bloom_filter(std::move(o)),
  ↪  patterns(std::move(o.patterns))
  {}

//move function
inline pattern_blocked_bf& operator=(pattern_blocked_bf&& o){
  if(this != &o){
    blocked_bloom_filter::operator=(std::move(o));
    patterns = std::move(o.patterns);
  }
  return *this;
}

inline virtual size_t get_pattern(const bloom_type& hash) const{
  size_t pattern_number = hash & (num_patterns - 1); //which block
  return pattern_number * (block_size / bits_per_char) /
  ↪  sizeof(cell_type); // how lighter does it
}

inline virtual void insert(const unsigned char* key_begin, const
↪  size_t& length){
  //index in table with first byte of block
  size_t block = get_block(hash_ap(key_begin, length, salt_[0]));
  size_t pattern = get_pattern(hash_ap(key_begin, length,
  ↪  salt_[1]));
  static_assert(block_size / bits_per_char / sizeof(cell_type) > 0,
    "Block size must be greater than or equal to size of cell
    ↪  type.");
  cell_type* bit_block = reinterpret_cast<cell_type*>(__builtin_as
  ↪  sume_aligned(bit_table_.get() + block, block_size /
  ↪  bits_per_char));
  cell_type* pattern_block = reinterpret_cast<cell_type*>(__builti
  ↪  n_assume_aligned(patterns.get() + pattern, block_size /
  ↪  bits_per_char));
```

```
  for(size_t i = 0; i < block_size / bits_per_char /
   ↪  sizeof(cell_type); ++i){
    *(bit_block + i) |= *(pattern_block + i);
  }
  ++inserted_element_count_;
}

template <typename T>
inline void insert(const T& t)
{
  // Note: T must be a C++ POD type.
  insert(reinterpret_cast<const unsigned char*>(&t),sizeof(T));
}

inline virtual bool contains(const unsigned char* key_begin, const
 ↪  size_t length) const{
  //index in table with first byte of block
  size_t block = get_block(hash_ap(key_begin, length, salt_[0]));
  size_t pattern = get_pattern(hash_ap(key_begin, length,
   ↪  salt_[1]));
  static_assert(block_size / bits_per_char / sizeof(cell_type) > 0,
    "Block size must be greater than or equal to size of cell
     ↪  type.");
  cell_type* bit_block = reinterpret_cast<cell_type*>(__builtin_as
   ↪  sume_aligned(bit_table_.get() + block, block_size /
   ↪  bits_per_char));
  cell_type* pattern_block = reinterpret_cast<cell_type*>(__builti
   ↪  n_assume_aligned(patterns.get() + pattern, block_size /
   ↪  bits_per_char));
  for(size_t i = 0; i < block_size / bits_per_char /
   ↪  sizeof(cell_type); ++i){
    if(!_mm256_testc_si256((*(bit_block + i) & *(pattern_block +
     ↪  i)), //==
      *(pattern_block + i)))
    {
      return false;
    }
  }
  return true;
}

template <typename T>
inline bool contains(const T& t) const
{
  return contains(reinterpret_cast<const unsigned
   ↪  char*>(&t),static_cast<std::size_t>(sizeof(T)));
}
```

```
  inline virtual bloom_type block_hash(const unsigned char*
  ↪  key_begin, const size_t& length) const{
    return hash_ap(key_begin, length, salt_[0]);
  }

  template <typename T>
  inline bloom_type block_hash(const T& t) const{
    return block_hash(reinterpret_cast<const unsigned
    ↪  char*>(&t),static_cast<std::size_t>(sizeof(T)));
  }

  inline virtual bloom_type pattern_hash(const unsigned char*
  ↪  key_begin, const size_t& length) const{
    return hash_ap(key_begin, length, salt_[1]);
  }

  template <typename T>
  inline bloom_type pattern_hash(const T& t) const{
    return pattern_hash(reinterpret_cast<const unsigned
    ↪  char*>(&t),static_cast<std::size_t>(sizeof(T)));
  }

  inline double effective_fpp() const {
    long double c = size() / element_count() ;
    long double lambda = block_size / c;
    long double fpp = 0;
    for(int i = 0; i < 3 * lambda; ++i){ //var = lambda, so 3*lambda
    ↪  should include most anything
      long double p_block = std::pow(lambda, i) * std::exp(-lambda)
      ↪  / std::tgammal(i+1);
      long double p_collision = 1.0l - std::pow(1.0l - 1.0l /
      ↪  (num_patterns), i);
      long double fpr_inner = std::pow(1.0l - std::exp(-1.0l *
      ↪  salt_.size() * i / block_size), 1.0l * salt_.size());
      fpr_inner = p_collision + (1.0l - p_collision) * fpr_inner;
      fpp += p_block * fpr_inner;
    }
    return fpp;
  }

};
```

```
//a class to hold an encoded kmer
class Kmer{
protected:
  size_t s; //num times kmer added to since last reset
  int k;
  uint64_t x[2]; //fwd and reverse
  uint64_t mask;
  uint64_t shift;
public:
  Kmer(int k): k(k), s(0), mask(k < 32 ? (1ULL<<k*2) - 1 : -1),
  ↪  shift((k-1)*2) {x[0] = x[1] = 0;}
  Kmer(const Kmer& o): k(o.k), s(o.s), mask(o.mask), shift(o.shift),
  ↪  x{o.x[0], o.x[1]} {}
  //add a character and return the number of times the kmer has
  ↪  been added to since last reset
  inline size_t push_back(char ch){
    int c = seq_nt16_int[seq_nt16_table[ch]];
    if (c < 4){
      x[0] = (x[0] << 2 | c) & mask;                  // forward
      ↪  strand
      x[1] = x[1] >> 2 | (uint64_t)(3 - c) << shift;  // reverse
      ↪  strand
      ++s;
    } else this->reset(); // if there is an "N", restart
    return s;
  }
  //get encoded kmer
  inline uint64_t get() const{return x[0] < x[1] ? x[0] : x[1];}
  ↪  //min of x[0] and x[1]
  //get encoded prefix
  inline uint64_t prefix() const{return
  ↪  this->get()&((1<<PREFIXBITS)-1);}
  //empty the kmer and set s to 0
  inline void reset(){s = 0; x[0] = x[1] = 0;}
  //the number of times the kmer has been added to since the last
  ↪  reset
  inline size_t size() const{return s;}
  //whether the kmer has enough bases to be of length k
  inline bool valid() const{return (s >= k);}
  //the length of the kmer
  inline int ksize() const{return k;}
  inline operator std::string() const {
    std::string ret{};
    for(int i = 1; i <= k; ++i){
      ret.push_back( seq_nt16_str[seq_nt16_table['0' + ((x[0] & (3ULL
      ↪  << (2*(k-i)))) >> (2*(k-i)))]] );
    }
```

```
    return ret;
  }
  inline explicit operator bool() const{return this->valid();}
};

//a bloom filter. TODO: make blocked; hold an array of bloom
 ↪  filters and
//delegate insert/query fn's to the appropriate one.
//calculating the right fpr may be a bit involved.
//if we do threads we can also add a mutex for insertion
class Bloom
{
public:
  typedef pattern_blocked_bf bloom_type;
  Bloom(unsigned long long int projected_element_count, double fpr,
   ↪  unsigned long long int seed = 0xA5A5A5A55A5A5A5AULL);
  Bloom(Bloom&& b) noexcept: bloom(std::move(b.bloom)){} //move ctor
  Bloom& operator=(Bloom&& o){bloom = std::move(o.bloom); return
   ↪  *this;} //move assign
  ~Bloom();
  bloom_parameters params;
  bloom_type bloom;
  inline void insert(const Kmer&
   ↪  kmer){if(kmer.valid()){bloom.insert(kmer.get());}}
  template <typename T>
  inline void insert(const T& t){bloom.insert(t);}
  inline bool query(const Kmer& kmer) const {return (kmer.valid() &&
   ↪  bloom.contains(kmer.get()));}
  inline double fprate() const {return bloom.effective_fpp();}
  inline unsigned long long inserted_elements() const {return
   ↪  bloom.element_count();}
  // inline double fprate() const {return bloom.GetActualFP();}
};

// typedef std::array<Bloom,(1<<PREFIXBITS)> bloomary_t;

std::array<std::vector<size_t>,2> overlapping_kmers_in_bf(std::string
 ↪  seq, const Bloom& b, int k = 31);

//return the total number of kmers in b
int nkmers_in_bf(std::string seq, const Bloom& b, int k);

//given a kmer, get the next character (in ACGT order) that would
 ↪  create a trusted
//kmer when appended and return it. Return 0 if none would be
 ↪  trusted.
//Set the flag to test in TGCA order instead.
```

```
char get_next_trusted_char(const bloom::Kmer& kmer, const Bloom&
↪  trusted, bool reverse_test_order = false);

//return the INCLUSIVE indices bounding the largest stretch of
↪  trusted sequence
//if the first value is -1, there are no trusted kmers.
//if the 2nd value is -1 (== std::string::npos), until the end of
↪  the string is trusted.
//thus the whole string being trusted looks like {0,
↪  std::string::npos}
//while no part of the string being trusted looks like
↪  {std::string::npos, std::string::npos}
std::array<size_t,2> find_longest_trusted_seq(std::string seq, const
↪  Bloom& b, int k);

//find the longest possible fix for the kmer at position (k-1)
↪  until the end
//return the best character (multiple in case of a tie), the index
↪  of the next untrusted base,
// and whether multiple corrections were considered for the fix.
//if the length of the fix character vector is 0, no fix was found
↪  and correction should end.
//If the sequence is reverse-complemented, set the flag to test
↪  bases in reverse order
// (TGCA) instead of (ACGT)
std::tuple<std::vector<char>, size_t, bool>
↪  find_longest_fix(std::string seq, const Bloom& t, int k, bool
↪  reverse_test_order = false);

//given the sampling rate, calculate the probability any kmer is in
↪  the array.
long double calculate_phit(const Bloom& bf, long double alpha);

//given the number of inserts and the desired fpr, calculate the
↪  total size of the hash needed
uint64_t numbits(uint64_t numinserts, long double fpr);

//given the desired fpr, calculate the number of hash fn's needed
//assuming we use the optimal number of bits
int numhashes(long double fpr);

//given a sequence and an anchor, see if the anchor could be
↪  improved by moving it to the left.
//this is used in the correction step.
//ensure anchor >= k before this function is called.
//return {anchor, multiple}, the location of the new anchor and
↪  whether multiple corrections
```

```
//were possible during anchor adjustment.
std::pair<size_t, bool> adjust_right_anchor(size_t anchor,
 ↪  std::string seq, const Bloom& trusted, int k);

//get the biggest consecutive trusted block, for creating a trusted
 ↪  anchor when
//one doesn't exist. If there are too many consecutive misses, the
 ↪  procedure
//will end early. 94518
int biggest_consecutive_trusted_block(std::string seq, const Bloom&
 ↪  trusted, int k, int current_len);

}

inline std::ostream& operator<< (std::ostream& stream, const
 ↪  bloom::Kmer& kmer){
  stream << std::string(kmer);
  return stream;
}

#endif
// FILE:/home/adam/code/kbbq/include/kbbq/covariateutils.hh


#ifndef COVARIATEUTILS_H
#define COVARIATEUTILS_H
#define KBBQ_MAXQ 93

#include <vector>
#include <utility>
#include <cmath>
#include <random>
#include <array>
#include <limits>
#include "readutils.hh"
#include "recalibrateutils.hh"

//fwd declare
namespace readutils{
  class CReadData;
}

namespace covariateutils{

typedef std::vector<int> meanq_t;
typedef std::vector<int> rgdq_t;
typedef std::vector<std::vector<int>> qscoredq_t;
```

```
typedef std::vector<std::vector<std::array<std::vector<int>,2>>>
↪  cycledq_t;
typedef std::vector<std::vector<std::vector<int>>> dinucdq_t;

struct dq_t
{
  meanq_t meanq;
  rgdq_t rgdq;
  qscoredq_t qscoredq;
  cycledq_t cycledq;
  dinucdq_t dinucdq;
};

typedef std::vector<int> prior1_t; //1 prior for each rg
typedef std::vector<std::vector<int>> prior2_t; //1 prior for each
↪  rg -> q pair

class NormalPrior{
public:
  static std::vector<long double> normal_prior;
  NormalPrior(){}
  ~NormalPrior(){}
  static long double get_normal_prior(size_t j);
};

/*
def _logpmf(self, x, n, p):
    k = floor(x)
    combiln = (gamln(n+1) - (gamln(k+1) + gamln(n-k+1)))
    return combiln + special.xlogy(k, p) + special.xlog1py(n-k, -p)
*/

inline long double log_binom_pmf(unsigned long long k, unsigned long
↪  long n, long double p){
  // k+=1; //+1/+2 correction
  // n+=2;
  long double coefficient = (std::lgamma(n+1) - (std::lgamma(k+1) +
  ↪  std::lgamma(n-k+1)));
  return coefficient + (long double)k * std::log(p) + (long
  ↪  double)(n-k) * std::log1p(-p);
}

inline std::vector<long double> log_binom_cdf(unsigned long long k,
↪  long double p){
  std::vector<long double> ret(k+1);
  ret[0] = log_binom_pmf(0,k,p);
  for(int i = 1; i <= k; ++i){
```

```
    ret[i] = std::log(std::exp(ret[i-1]) +
    ↪  std::exp(log_binom_pmf(i,k,p)));
  }
  return ret;
}

//return vector of each k (k < n)
inline std::vector<int> calculate_thresholds(unsigned long long k,
↪  long double p, long double quartile = .9951){
  std::vector<int> threshold(k+1,0);
  threshold[0] = 0;
  for(int i = 1; i <= k; ++i){
    std::vector<long double> cdf = log_binom_cdf(i,p);
    for(int j = 0; j < cdf.size(); ++j){
      long double logp = cdf[j];
      if(logp >= std::log(quartile)){
        threshold[i] = j;
        break;
      }
    }
  }
  return threshold;
}

inline bool nt_is_not_n(char c){
  return seq_nt16_int[seq_nt16_table[c]] < 4;
}

//ensure there's no N!!!!
inline int8_t dinuc_to_int(char first, char second){
  int8_t f = seq_nt16_int[seq_nt16_table[first]];
  int8_t s = seq_nt16_int[seq_nt16_table[second]];
  return 15 & ((f << 2) | s); // 1111 & (xx00 | 00xx)
}

inline std::array<char, 2> int_to_dinuc(int8_t dinuc){
  std::array<char,2> x = {std::to_string(dinuc >> 2)[0],
  ↪  std::to_string(dinuc | 3)[0]};
  return x;
}

typedef std::array<unsigned long long, 2> covariate_t;

class CCovariate: public std::vector<covariate_t>
{
public:
  CCovariate(): std::vector<covariate_t>(){}
```

```
  CCovariate(size_t len): std::vector<covariate_t>(len) {}
  void increment(size_t idx, covariate_t value);
  void increment(size_t idx, unsigned long long err, unsigned long
   ↪  long total);
};

class CRGCovariate: public CCovariate
{
public:
  CRGCovariate(){}
  CRGCovariate(size_t len): CCovariate(len){}
  void consume_read(const readutils::CReadData& read);
  rgdq_t delta_q(prior1_t prior);
};

class CQCovariate: public std::vector<CCovariate>
{
public:
  CQCovariate(){}
  CQCovariate(size_t rgs, size_t qlen): std::vector<CCovariate>(rgs,
   ↪  CCovariate(qlen)){}
  void consume_read(const readutils::CReadData& read);
  qscoredq_t delta_q(prior1_t prior);
};

typedef std::array<CCovariate,2> cycle_t;

//The first Covariate is for fwd reads, the 2nd Covariate is for
 ↪  reverse reads.
class CCycleCovariate: public std::vector<std::vector<cycle_t>>
{
public:
  CCycleCovariate(){}
  CCycleCovariate(size_t rgs, size_t qlen, size_t cylen):
    std::vector<std::vector<cycle_t>>(rgs, std::vector<cycle_t>(qlen,
     ↪  cycle_t({CCovariate(cylen),CCovariate(cylen)})))
    {}
  void consume_read(const readutils::CReadData& read);
  cycledq_t delta_q(prior2_t prior);
};

class CDinucCovariate: public std::vector<std::vector<CCovariate>>
{
public:
  CDinucCovariate(){}
  CDinucCovariate(size_t rgs, size_t qlen, size_t dilen):
```

```
    std::vector<std::vector<CCovariate>>(rgs,
    ↪  std::vector<CCovariate>(qlen, CCovariate(dilen)))
    {}
  void consume_read(const readutils::CReadData& read, int minscore =
  ↪  6);
  dinucdq_t delta_q(prior2_t prior);
};

class CCovariateData
{
// protected:
//   CRGCovariate rgcov;
//   CQCovariate qcov;
//   CCycleCovariate cycov;
//   CDinucCovariate dicov;
public:
  CRGCovariate rgcov;
  CQCovariate qcov;
  CCycleCovariate cycov;
  CDinucCovariate dicov;
  CCovariateData(){};
  void consume_read(readutils::CReadData& read, int minscore = 6);
  dq_t get_dqs();
};

}
#endif
// FILE:/home/adam/code/kbbq/include/kbbq/recalibrateutils.hh


#ifndef RECALIBRATEUTILS_H
#define RECALIBRATEUTILS_H

#include <vector>
#include <string>
#include <cmath>
#include "bloom.hh"
#include "htsiter.hh"
#include "covariateutils.hh"

//debug
#include <fstream>


//fwd declare covariateutils stuff
namespace covariateutils{
  class CCovariateData;
```

```
  struct dq_t;
}

namespace htsiter{
  class KmerSubsampler;
  class HTSFile;
}

namespace recalibrateutils{

//subsample kmers, hash them, and add them to the bloom filter
void subsample_kmers(htsiter::KmerSubsampler& s, bloom::Bloom&
↪  sampled);

//get some reads from a file, whether a kmer is trusted and put it
↪  in a cache.
//this can probably be parallelized if needed because the reads are
↪  independent
void find_trusted_kmers(htsiter::HTSFile* file, bloom::Bloom& trusted,
  const bloom::Bloom& sampled, std::vector<int> thresholds, int k);

inline long double q_to_p(int q){return std::pow(10.0l, -((long
↪  double)q / 10.0l));}
inline int p_to_q(long double p, int maxscore = 42){return p > 0 ?
↪  (int)(-10 * std::log10(p)) : maxscore;}

//get covariate data using the trusted kmers
//this can be parallelized easily since each read is independent
covariateutils::CCovariateData get_covariatedata(htsiter::HTSFile*
↪  file, const bloom::Bloom& trusted, int k);

//recalibrate all reads given the CovariateData
void recalibrate_and_write(htsiter::HTSFile* in, const
↪  covariateutils::dq_t& dqs, std::string outfn);

}

#endif
// FILE:/home/adam/code/kbbq/include/kbbq/htsiter.hh


#ifndef KBBQ_HTSITER_H
#define KBBQ_HTSITER_H

#include <iterator>
#include <string>
#include <random>
```

```
#include <algorithm>
#include <htslib/hts.h>
#include <htslib/bgzf.h>
#include <htslib/sam.h>
#include <htslib/kseq.h>
#include <htslib/kstring.h>
#include <htslib/thread_pool.h>
#include <minion.hpp>
#include "readutils.hh"
#include "kseq.hh"
#include "bloom.hh"
#include <cstdlib>

//This is defined starting in version 1.10
#ifndef HTS_VERSION
#define HTS_VERSION 0
#endif

static_assert(HTS_VERSION >= 101000, "Your version of htslib is out
↪  of date. KBBQ requires version >= 1.10.");

#ifndef NDEBUG
#define KBBQ_USE_RAND_SAMPLER
#endif

//fwd declare
namespace readutils{
  class CReadData;
  std::string bam_seq_str(bam1_t* bamrecord);
}

//unsigned char* s = bam_get_seq(bamrecord);
namespace htsiter{

class HTSFile{
public:
  virtual ~HTSFile(){}
  virtual int next()=0;
  virtual std::string next_str()=0;
  virtual readutils::CReadData get()=0;
  virtual void recalibrate(const std::vector<uint8_t>& qual)=0;
  virtual int open_out(std::string filename)=0; //open an output
  ↪  file so it can be written to later.
  virtual int write()=0; //write the current read to the opened file.
};

class BamFile: public HTSFile{
```

```
public:
  samFile *sf;
  // hts_idx_t *idx;
  sam_hdr_t *h;
  // hts_itr_t *itr;
  bam1_t *r;
  samFile *of;
  bool use_oq;
  bool set_oq;
  htsThreadPool* tp;
  BamFile(std::string filename, htsThreadPool* tp, bool use_oq =
  ↪  false, bool set_oq = false):
    use_oq(use_oq), set_oq(set_oq), tp(tp){
    r = bam_init1();
    sf = sam_open(filename.c_str(), "r");
    if(tp->pool && hts_set_thread_pool(sf, tp) != 0){
      std::cerr << "Couldn't attach thread pool to file " << filename
      ↪  << std::endl;
    };
      h = sam_hdr_read(sf);
      //TODO: support iteration with index?
      // idx = sam_index_load(sf, filename.c_str());
       //TODO: throw when index can't be found
      // itr = sam_itr_queryi(idx, HTS_IDX_START, 0, 0); //iterate
      ↪  over whole file;
      of = NULL; //this can be opened later with open_out
  }
  ~BamFile(){
    if(r != NULL){bam_destroy1(r);}
    // if(itr != NULL){sam_itr_destroy(itr);}
    if(sf != NULL){sam_close(sf);}
    if(of != NULL){sam_close(of);}
    if(h != NULL){sam_hdr_destroy(h);}
    // if(idx != NULL){hts_idx_destroy(idx);}
  }

  // to use: while (ret = BamFile.next() >= 0){//do something with
  ↪  this->r}
  // >= 0 on success, -1 on EOF, <-1 on error
  int next();
  // return next read sequence as a string. if there are no more,
  ↪  return the empty string.
  std::string next_str();
  //
  readutils::CReadData get();
  //
  void recalibrate(const std::vector<uint8_t>& qual);
```

```
  // TODO:: add a PG tag to the header
  int open_out(std::string filename);
  //
  int write();
  //
}; //end of BamFile class

class FastqFile: public HTSFile
{
public:
  BGZF* fh;
  kseq::kseq_t* r;
  BGZF* ofh;
  htsThreadPool* tp;
  FastqFile(std::string filename, htsThreadPool* tp): ofh(NULL),
  ↪  tp(tp){
    fh = bgzf_open(filename.c_str(),"r");
    if(tp->pool && bgzf_thread_pool(fh, tp->pool, tp->qsize) < 0){
      std::cerr << "Couldn't attach thread pool to file " << filename
      ↪  << std::endl;
    }
    r = kseq::kseq_init(fh);
  };
  ~FastqFile(){
    if(r != NULL){kseq::kseq_destroy(r);}
    if(fh != NULL){bgzf_close(fh);}
    if(ofh != NULL){bgzf_close(ofh);}
  }
  int next();
  std::string next_str();
  readutils::CReadData get();
  void recalibrate(const std::vector<uint8_t>& qual);
  int open_out(std::string filename);
  int write();
};

class KmerSubsampler{
public:
  HTSFile* file;
  minion::Random rng;
  std::bernoulli_distribution d;
  std::string readseq = "";
  std::vector<bloom::Kmer> kmers;
  bloom::Kmer kmer;
  size_t cur_kmer = 0;
  size_t total_kmers = 0;
  bool not_eof = true;
```

```
  int k;
  KmerSubsampler(HTSFile* file): KmerSubsampler(file, KBBQ_MAX_KMER){}
  KmerSubsampler(HTSFile* file, int k): KmerSubsampler(file, k, .15){}
  KmerSubsampler(HTSFile* file, int k, double alpha):
   ↪  KmerSubsampler(file, k, alpha,
   ↪  minion::create_seed_seq().GenerateOne()){}
  KmerSubsampler(HTSFile* file, int k, double alpha, uint64_t seed):
   ↪  file(file), k(k), kmer(k), d(alpha) {rng.Seed(seed); std::cerr
   ↪  << "p: " << d.p() << std::endl;} //todo remove srand

  //return the next kmer
  //once the file is finished iterating and there are no remaining
   ↪  kmers,
  //return an empty kmer. You should check the not_eof flag.
  //the kmer returned is not guaranteed to be valid.
  bloom::Kmer next_kmer();

  //return the next kmer that survives sampling.
  //it is not gauranteed to be valid.
  bloom::Kmer next();
  inline bloom::Kmer operator()() {return this->next();}
  inline explicit operator bool() const{return not_eof;}
};

}

#endif
```

F.2 Source

```
// FILE:/home/adam/code/kbbq/src/kbbq/kbbq.cc


#include "htsiter.hh"
#include "bloom.hh"
#include "covariateutils.hh"
#include "recalibrateutils.hh"
#include <memory>
#include <iostream>
#include <fstream>
#include <sstream>
#include <htslib/hfile.h>
#include <htslib/thread_pool.h>
#include <getopt.h>
#include <cassert>
#include <functional>
#include <iomanip>
#include <ctime>
```

```
#ifndef NDEBUG
#define KBBQ_USE_RAND_SAMPLER
#endif

//opens file filename and returns a unique_ptr to the result.
std::unique_ptr<htsiter::HTSFile> open_file(std::string filename,
 ↪  htsThreadPool* tp, bool is_bam = true, bool use_oq = false, bool
 ↪  set_oq = false){
  std::unique_ptr<htsiter::HTSFile> f(nullptr);
  if(is_bam){
    f = std::move(std::unique_ptr<htsiter::BamFile>(new
     ↪  htsiter::BamFile(filename, tp, use_oq, set_oq)));
    // f.reset(new htsiter::BamFile(filename));
  } else {
    f = std::move(std::unique_ptr<htsiter::FastqFile>(new
     ↪  htsiter::FastqFile(filename, tp)));
    // f.reset(new htsiter::FastqFile(filename));
  }
  return f;
}



template<typename T>
std::ostream& operator<< (std::ostream& stream, const
 ↪  std::vector<T>& v){
  stream << "[";
  std::copy(v.begin(), v.end(), std::ostream_iterator<T>(stream, ",
   ↪  "));
  stream << "]";
  return stream;
}

template<typename T>
void print_vec(const std::vector<T>& v){
  std::cerr << v;
}

std::ostream& put_now(std::ostream& os){
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  return os << std::put_time(&tm, "[%F %T %Z]");
}

int check_args(int argc, char* argv[]){
  if(argc < 2){
```

```
    std::cerr << put_now << " Usage: " << argv[0] << "
    ↪  input.[bam,fq]" << std::endl;
    return 1;
  } else {
    std::cerr << put_now << "  Selected file: " <<
    ↪  std::string(argv[1]) << std::endl;
    return 0;
  }
}

//long option, required arg?, flag, value
struct option long_options[] = {
  {"ksize",required_argument,0,'k'}, //default: 31
  {"use-oq",no_argument,0,'u'}, //default: off
  {"set-oq",no_argument,0,'s'}, //default: off
  {"genomelen",required_argument,0,'g'}, //estimated for bam input,
  ↪  required for fastq input
  {"coverage",required_argument,0,'c'}, //default: estimated
  {"fixed",required_argument,0,'f'}, //default: none
  {"alpha",required_argument,0,'a'}, //default: 7 / coverage
  {"threads",required_argument,0,'t'},
#ifndef NDEBUG
  {"debug",required_argument,0,'d'},
#endif
  {0, 0, 0, 0}
};

int main(int argc, char* argv[]){
  int k = 32;
  long double alpha = 0;
  uint64_t genomelen = 0; //est w/ index with bam, w/ fq estimate w/
  ↪  coverage
  uint coverage = 0; //if not given, will be estimated.
  uint32_t seed = 0;
  bool set_oq = false;
  bool use_oq = false;
  int nthreads = 0;
  std::string fixedinput = "";

  int opt = 0;
  int opt_idx = 0;
#ifndef NDEBUG
  std::string kmerlist("");
  std::string trustedlist("");
#endif
  while((opt = getopt_long(argc,argv,"k:usg:c:f:a:t:d:",long_options,
  ↪  &opt_idx)) != -1){
```

```
    switch(opt){
      case 'k':
        k = std::stoi(std::string(optarg));
        if(k <= 0 || k > KBBQ_MAX_KMER){
          std::cerr << put_now << "  Error: k must be <= " <<
          ↪  KBBQ_MAX_KMER << " and > 0." << std::endl;
        }
        break;
      case 'u':
        use_oq = true;
        break;
      case 's':
        set_oq = true;
        break;
      case 'g':
        genomelen = std::stoull(std::string(optarg));
        break;
      case 'c':
        coverage = std::stoul(std::string(optarg));
        break;
      case 'f':
        fixedinput = std::string(optarg);
        break;
      case 'a':
        alpha = std::stold(std::string(optarg));
        break;
      case 't':
        nthreads = std::stoi(std::string(optarg));
        if(nthreads < 0){
          std::cerr << put_now << " Error: threads must be >= 0." <<
          ↪  std::endl;
        }
        break;
#ifndef NDEBUG
      case 'd': {
        std::string optstr(optarg);
        std::istringstream stream(optstr);
        std::getline(stream, kmerlist, ',');
        std::getline(stream, trustedlist, ',');
        break;
      }
#endif
      case '?':
      default:
        std::cerr << put_now << "  Unknown argument " << (char)opt <<
        ↪  std::endl;
        return 1;
```

```
      break;
  }
}

std::string filename("-");
if(optind < argc){
  filename = std::string(argv[optind]);
  while(++optind < argc){
    std::cerr << put_now << " Warning: Extra argument " <<
    ↪  argv[optind] << " ignored." << std::endl;
  }
}

long double sampler_desiredfpr = 0.01; //Lighter uses .01
long double trusted_desiredfpr = 0.0005; // and .0005

//create thread pool
std::unique_ptr<htsThreadPool, std::function<void(htsThreadPool*)>>
↪  tp{
  new htsThreadPool,
  [](htsThreadPool* ptr){
    if(ptr->pool){hts_tpool_destroy(ptr->pool);}
  }
};
if(nthreads > 0 && !(tp->pool = hts_tpool_init(nthreads))){
  std::cerr << put_now << " Unable to construct thread pool." <<
  ↪  std::endl;
  return 1;
}


//see if we have a bam
htsFormat fmt;
hFILE* fp = hopen(filename.c_str(), "r");
if (hts_detect_format(fp, &fmt) < 0) {
  //error
  std::cerr << put_now << " Error opening file " << filename <<
  ↪  std::endl;
  hclose_abruptly(fp);
  return 1;
}
bool is_bam = true;
if(fmt.format == bam || fmt.format == cram){
  is_bam = true;
} else if (fmt.format == fastq_format){
  is_bam = false;
} else {
```

```
    //error
    std::cerr << put_now << " Error: File format must be bam, cram,
    ↪  or fastq." << std::endl;
    hclose_abruptly(fp);
    return 1;
  }
  std::unique_ptr<htsiter::HTSFile> file;
  covariateutils::CCovariateData data;

if(fixedinput == ""){ //no fixed input provided

  if(genomelen == 0){
    if(is_bam){
      std::cerr << put_now << " Estimating genome length" <<
      ↪  std::endl;
      samFile* sf = hts_hopen(fp, filename.c_str(), "r");
      if(tp->pool && hts_set_thread_pool(sf, tp.get()) != 0){
        std::cerr << "Couldn't attach thread pool to file " <<
        ↪  filename << std::endl;
      };
      sam_hdr_t* h = sam_hdr_read(sf);
      for(int i = 0; i < sam_hdr_nref(h); ++i){
        genomelen += sam_hdr_tid2len(h, i);
      }
      sam_hdr_destroy(h);
      hts_close(sf);
      if(genomelen == 0){
        std::cerr << put_now << " Header does not contain genome
        ↪  information." <<
          " Unable to estimate genome length; please provide it on
          ↪  the command line" <<
          " using the --genomelen option." << std::endl;
        return 1;
      } else {
        std::cerr << put_now << " Genome length is " << genomelen <<"
        ↪  bp." << std::endl;
      }
    } else {
      std::cerr << put_now << " Error: --genomelen must be specified
      ↪  if input is not a bam." << std::endl;
    }
  } else {
    if(hclose(fp) != 0){
      std::cerr << put_now << " Error closing file!" << std::endl;
    }
  }
```

```
//alpha not provided, coverage not provided
if(alpha == 0){
  std::cerr << put_now << " Estimating alpha." << std::endl;
  if(coverage == 0){
    std::cerr << put_now << " Estimating coverage." << std::endl;
    uint64_t seqlen = 0;
    file = std::move(open_file(filename, tp.get(), is_bam, use_oq,
    ↪  set_oq));
    std::string seq("");
    while((seq = file->next_str()) != ""){
      seqlen += seq.length();
    }
    if (seqlen == 0){
      std::cerr << put_now << " Error: total sequence length in
      ↪  file " << filename <<
        " is 0. Check that the file isn't empty." << std::endl;
      return 1;
    }
    std::cerr << put_now << " Total Sequence length: " << seqlen <<
    ↪  std::endl;
    std::cerr << put_now << " Genome length: " << genomelen <<
    ↪  std::endl;
    coverage = seqlen/genomelen;
    std::cerr << put_now << " Estimated coverage: " << coverage <<
    ↪  std::endl;
    if(coverage == 0){
      std::cerr << put_now << " Error: estimated coverage is 0." <<
      ↪  std::endl;
      return 1;
    }
  }
  alpha = 7.0l / (long double)coverage; // recommended by Lighter
  ↪  authors
}

if(coverage == 0){ //coverage hasn't been estimated but alpha is
↪  given
  coverage = 7.0l/alpha;
}

file = std::move(open_file(filename, tp.get(), is_bam, use_oq,
↪  set_oq));

std::cerr << put_now << " Sampling kmers at rate " << alpha <<
↪  std::endl;
```

```
  //in the worst case, every kmer is unique, so we have genomelen *
  ↪  coverage kmers
  //then we will sample proportion alpha of those.
  unsigned long long int approx_kmers = genomelen*coverage*alpha;
  bloom::Bloom subsampled(approx_kmers, sampler_desiredfpr);
  ↪  //lighter uses 1.5 * genomelen
  bloom::Bloom trusted(approx_kmers, trusted_desiredfpr);

  if(seed == 0){
    seed = minion::create_seed_seq().GenerateOne();
  }
  std::cerr << put_now << " Seed: " << seed << std::endl ;

  //sample kmers here.
#ifdef KBBQ_USE_RAND_SAMPLER
  std::srand(seed); //lighter uses 17
#endif
  htsiter::KmerSubsampler subsampler(file.get(), k, alpha, seed);
  //load subsampled bf.
  //these are hashed kmers.
  recalibrateutils::subsample_kmers(subsampler, subsampled);

  //report number of sampled kmers
  std::cerr << put_now << " Sampled " <<
  ↪  subsampled.inserted_elements() << " valid kmers." << std::endl;

#ifndef NDEBUG
  //ensure kmers are properly sampled
  if(kmerlist != ""){
    std::ifstream kmersin(kmerlist);
    bloom::Kmer kin(k);
    for(std::string line; std::getline(kmersin, line); ){
      kin.reset();
      for(char c: line){
        kin.push_back(c);
      }
      if(kin.valid()){
        assert(subsampled.query(kin));
      }
    }
  }
#endif


  //calculate thresholds
  long double fpr = subsampled.fprate();
```

```
  std::cerr << put_now << " Approximate false positive rate: " << fpr
   ↪  << std::endl;
  if(fpr > .15){
    std::cerr << put_now << " Error: false positive rate is too high.
     ↪  " <<
      "Increase genomelen parameter and try again." << std::endl;
    return 1;
  }

  long double p = bloom::calculate_phit(subsampled, alpha);
  std::vector<int> thresholds =
   ↪  covariateutils::calculate_thresholds(k, p);
#ifndef NDEBUG
  std::vector<int> lighter_thresholds = {0, 1, 2, 3, 4, 4, 5, 5, 6, 6,
    7, 7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 12, 13, 13, 14, 14, 15,
     ↪  15, 15, 16, 16, 17};

  std::cerr << put_now << " Thresholds: [ " ;
  std::copy(thresholds.begin(), thresholds.end(),
   ↪  std::ostream_iterator<int>(std::cerr, " "));
  std::cerr << "]" << std::endl;
  std::cerr << put_now << " Lighter Th: [ " ;
  std::copy(lighter_thresholds.begin(), lighter_thresholds.end(),
   ↪  std::ostream_iterator<int>(std::cerr, " "));
  std::cerr << "]" << std::endl;
  assert(lighter_thresholds == thresholds);
#endif

  std::vector<long double> cdf = covariateutils::log_binom_cdf(k,p);

  std::cerr << put_now << " log CDF: [ " ;
  for(auto c : cdf){std::cerr << c << " ";}
  std::cerr << "]" << std::endl;

  //get trusted kmers bf using subsampled bf
  std::cerr << put_now << " Finding trusted kmers" << std::endl;

  file = std::move(open_file(filename, tp.get(), is_bam, use_oq,
   ↪  set_oq));
  recalibrateutils::find_trusted_kmers(file.get(), trusted,
   ↪  subsampled, thresholds, k);

#ifndef NDEBUG
// check that all kmers in trusted list are actually trusted in our
 ↪  list.
// it seems that lighter has quite a few hash collisions that end
 ↪  up making
```

```
// it trust slightly more kmers than it should

if(trustedlist != ""){
  std::ifstream kmersin(trustedlist);
  bloom::Kmer kin(k);
  for(std::string line; std::getline(kmersin, line); ){
    // std::cerr << "Trusted kmer: " << line << std::endl;
    kin.reset();
    for(char c: line){
      kin.push_back(c);
    }
    if(!trusted.query(kin)){
      std::cerr << "Trusted kmer not found!" << std::endl;
      std::cerr << "Line: " << line << std::endl;
      std::cerr << "Kmer: " << kin << std::endl;
    }
    assert(trusted.query(kin));
  }
}
#endif

  //use trusted kmers to find errors
  std::cerr << put_now << " Finding errors" << std::endl;
  file = std::move(open_file(filename, tp.get(), is_bam, use_oq,
  ↪  set_oq));
  data = recalibrateutils::get_covariatedata(file.get(), trusted, k);
} else { //use fixedfile to find errors
  std::cerr << put_now << " Using fixed file to find errors." <<
  ↪  std::endl;
  file = std::move(open_file(filename, tp.get(), is_bam, use_oq,
  ↪  set_oq));
  std::unique_ptr<htsiter::HTSFile> fixedfile =
  ↪  std::move(open_file(fixedinput, tp.get(), is_bam, use_oq,
  ↪  set_oq));
  while(file->next() >= 0 && fixedfile->next() >= 0){
    readutils::CReadData read = file->get();
    readutils::CReadData fixedread = fixedfile->get();
    std::transform(read.seq.begin(), read.seq.end(),
     ↪  fixedread.seq.begin(),
      read.errors.begin(), std::not_equal_to<char>{});
    data.consume_read(read);
  }
}
```

```
  std::vector<std::string>
  ↪  rgvals(readutils::CReadData::rg_to_int.size(), "");
  for(auto i : readutils::CReadData::rg_to_int){
    rgvals[i.second] = i.first;
  }

#ifndef NDEBUG
  std::cerr << put_now << " Covariate data:" << std::endl;
  std::cerr << "rgcov:";
  for(int i = 0; i < data.rgcov.size(); ++i){ //rgcov[rg][0] = errors
    std::cerr << i << ": " << rgvals[i] << " {" << data.rgcov[i][0]
    ↪  << ", " << data.rgcov[i][1] << "}" << std::endl;
  }
  std::cerr << "qcov:" << "(" << data.qcov.size() << ")" << std::endl;
  for(int i = 0; i < data.qcov.size(); ++i){
    std::cerr << i << " " << rgvals[i] << "(" << data.qcov[i].size()
    ↪  << ")" << ": [";
    for(int j = 0; j < data.qcov[i].size(); ++j){
      if(data.qcov[i][j][1] != 0){
        std::cerr << j << ":{" << data.qcov[i][j][0] << ", " <<
        ↪  data.qcov[i][j][1] << "} ";
      }
    }
    std::cerr << "]" << std::endl;
  }
#endif

  //recalibrate reads and write to file
  std::cerr << put_now << " Training model" << std::endl;
  covariateutils::dq_t dqs = data.get_dqs();


#ifndef NDEBUG
  std::cerr << put_now << " dqs:\n" << "meanq: ";
  print_vec<int>(dqs.meanq);
  std::cerr << "\nrgdq:" << std::endl;
  for(int i = 0; i < dqs.rgdq.size(); ++i){
    std::cerr << rgvals[i] << ": " << dqs.rgdq[i] << " (" <<
    ↪  dqs.meanq[i] + dqs.rgdq[i] << ")" << std::endl;
  }
  std::cerr << "qscoredq:" << std::endl;
  for(int i = 0; i < dqs.qscoredq.size(); ++i){
    for(int j = 0; j < dqs.qscoredq[i].size(); ++j){
      if(data.qcov[i][j][1] != 0){
        std::cerr << rgvals[i] << ", " << "q = " << j << ": " <<
        ↪  dqs.qscoredq[i][j] << " (" <<
          dqs.meanq[i] + dqs.rgdq[i] + dqs.qscoredq[i][j] << ") " <<
```

```
            data.qcov[i][j][1] << " " << data.qcov[i][j][0] <<
             ↪  std::endl;
        }
      }
    }
    std::cerr << "cycledq:" << std::endl;
    for(int i = 0; i < dqs.cycledq.size(); ++i){
      for(int j = 0; j < dqs.cycledq[i].size(); ++j){
        if(data.qcov[i][j][1] != 0){
          for(size_t k = 0; k < dqs.cycledq[i][j].size(); ++k){
            for(int l = 0; l < dqs.cycledq[i][j][k].size(); ++l){
              std::cerr << rgvals[i] << ", " << "q = " << j << ", cycle
               ↪  = " << (k ? -(l+1) : l+1) << ": " <<
               ↪  dqs.cycledq[i][j][k][l] << " (" <<
                dqs.meanq[i] + dqs.rgdq[i] + dqs.qscoredq[i][j] +
                 ↪  dqs.cycledq[i][j][k][l] << ") " <<
                data.cycov[i][j][k][l][1] << " " <<
                 ↪  data.cycov[i][j][k][l][0] << std::endl;
            }
          }
        }
      }
    }
    std::cerr << "dinucdq:" << std::endl;
    for(int i = 0; i < dqs.dinucdq.size(); ++i){
      for(int j = 0; j < dqs.dinucdq[i].size(); ++j){
        if(data.qcov[i][j][1] != 0){
          for(size_t k = 0; k < dqs.dinucdq[i][j].size(); ++k){
            std::cerr << rgvals[i] << ", " << "q = " << j << ", dinuc =
             ↪  " << seq_nt16_str[seq_nt16_table['0' + (k >> 2)]] <<
             ↪  seq_nt16_str[seq_nt16_table['0' + (k & 3)]] << ": " <<
             ↪  dqs.dinucdq[i][j][k] << " (" <<
              dqs.meanq[i] + dqs.rgdq[i] + dqs.qscoredq[i][j] +
               ↪  dqs.dinucdq[i][j][k] << ") " <<
              data.dicov[i][j][k][1] << " " << data.dicov[i][j][k][0]
               ↪  << std::endl;
          }
        }
      }
    }
  #endif

    std::cerr << put_now << " Recalibrating file" << std::endl;
    file = std::move(open_file(filename, tp.get(), is_bam, use_oq,
     ↪  set_oq));
    recalibrateutils::recalibrate_and_write(file.get(), dqs, "-");
    return 0;
```

```
}

// FILE:/home/adam/code/kbbq/src/kbbq/test.cc


#include <bitset>

int test(){
  bloom_parameters p{};
  p.projected_element_count = 100;
  p.false_positive_probability = .05;
  p.compute_optimal_parameters();
  bloom::pattern_blocked_bf bf(p);
  std::cerr << "Table size: " << bf.size() << std::endl;
  std::cerr << "Block size: " << bf.block_size << std::endl;
  std::cerr << "Block bytes: " << bf.block_size / bits_per_char <<
  ↪  std::endl;
  std::cerr << "Pattern bytes: " << bf.num_patterns * bf.block_size /
  ↪  bits_per_char << std::endl;
  std::cerr << "Num hashes: " << bf.hash_count() << std::endl;
  std::cerr << "Optimal K: " << p.optimal_parameters.number_of_hashes
  ↪  << std::endl;
  for(size_t i = 0; i < 10; ++i){
    // std::cerr << (void*)bf.patterns.get() << std::endl;
    std::cerr << i << "(" << (void*)&bf.patterns.get()[2 * i] << ")"
    ↪  << ": ";
    for(size_t j = 0; j < 4; ++j){
      std::cerr << std::bitset<64>(bf.patterns.get()[2*i][j]) << ", ";
    }
    for(size_t j = 0; j < 4; ++j){
      std::cerr << std::bitset<64>(bf.patterns.get()[2*i+1][j]) << ",
      ↪  ";
    }
    std::cerr << std::endl;
  }
  uint64_t kmer = 21345423534512;
  uint64_t gibberish = 123543451243ull;
  bf.insert(kmer);
  std::cerr << "kmer inserted into block: " <<
  ↪  bf.get_block(bf.block_hash(kmer)) << std::endl;
  size_t kpattern = bf.get_pattern(bf.pattern_hash(kmer));
  std::cerr << "kmer pattern number: " << kpattern << std::endl;
  std::cerr << "gibberish block: " <<
  ↪  bf.get_block(bf.block_hash(gibberish)) << std::endl;
  std::cerr << "gibberish pattern number: " <<
  ↪  bf.get_pattern(bf.pattern_hash(gibberish)) << std::endl;
  std::cerr << "kmerp: ";
```

```
  for(size_t i = kpattern; i < kpattern + 2;++i){
    for(size_t j = 0; j < 4; ++j){
      std::cerr << std::bitset<64>(bf.patterns.get()[i][j]) << ", ";
    }
  }
  std::cerr << std::endl;

  std::cerr << "Table: \n";
  for(size_t i = 0; i < bf.size() / bits_per_char / 64; ++i){
    // if(i % (bf.block_size/bits_per_char) == 0){
    //   std::cerr << "\nBLOCK" << std::endl;
    // }
    std::cerr << "block: ";
    for(size_t j = 0; j < 4; ++j){
      std::cerr << std::bitset<64>(bf.bit_table_.get()[2*i][j]) << ",
      ↪  ";
    }
    for(size_t j = 0; j < 4; ++j){
      std::cerr << std::bitset<64>(bf.bit_table_.get()[2*i+1][j]) <<
      ↪  ", ";
    }
    std::cerr << std::endl;
  }
  std::cerr << std::endl;
  std::cerr << "bf contains kmer ? " << bf.contains(kmer) <<
  ↪  std::endl;
  std::cerr << "bf contains gibberish ? " << bf.contains(gibberish)
  ↪  << std::endl;
  assert(bf.contains(kmer));


  return 0;
}
// FILE:/home/adam/code/kbbq/src/kbbq/htsiter.cc


#include "htsiter.hh"

namespace htsiter{

int BamFile::next(){return sam_read1(sf, h, r);}//return
↪  sam_itr_next(sf, itr, r);
  // return next read as a string. if there are no more, return the
  ↪  empty string.
std::string BamFile::next_str(){return this->next() >= 0 ?
↪  readutils::bam_seq_str(r) : "";}
//
```

```
readutils::CReadData BamFile::get(){return
 ↪  readutils::CReadData(this->r, use_oq);}
//
void BamFile::recalibrate(const std::vector<uint8_t>& qual){
  uint8_t* q = bam_get_qual(this->r);
  if(set_oq){
    std::string qstr;
    std::transform(q, q + this->r->core.l_qseq,
     ↪  std::back_inserter(qstr),
      [](uint8_t c) -> char {return c + 33;}); //qual value to
       ↪  actual str
    //returns 0 on success, -1 on fail. We should consider throwing
     ↪  if it fails.
    if(bam_aux_update_str(this->r, "OQ", qstr.length()+1,
     ↪  qstr.c_str()) != 0){
      if(errno == ENOMEM){
        std::cerr << "Insufficient memory to expand bam record." <<
         ↪  std::endl;
      } else if(errno == EINVAL){
        std::cerr << "Tag data is corrupt. Repair the tags and try
         ↪  again." << std::endl;
      }
      throw std::invalid_argument("Unable to update OQ tag.");
    }
  }
  if(bam_is_rev(this->r)){
    std::reverse_copy(qual.begin(), qual.end(), q);
  } else {
    std::copy(qual.begin(), qual.end(), q);
  }
  // for(int i = 0; i < this->r->core.l_qseq; ++i){
  //   q[i] = (char)qual[i];
  // }
}
// TODO:: add a PG tag to the header
int BamFile::open_out(std::string filename){
  this->of = sam_open(filename.c_str(), "wb");
  if(tp->pool && hts_set_thread_pool(this->of, tp) != 0){
    std::cerr << "Couldn't attach thread pool to file " << filename
     ↪  << std::endl;
  };
  return sam_hdr_write(this->of, this->h);
}
//
int BamFile::write(){return sam_write1(this->of, this->h, this->r);}

// FastqFile class
```

```
int FastqFile::next(){
  return kseq_read(r);
}

std::string FastqFile::next_str(){
  return this->next() >= 0? std::string(this->r->seq.s): "";
}

readutils::CReadData FastqFile::get(){
  return readutils::CReadData(this->r);
}

void FastqFile::recalibrate(const std::vector<uint8_t>& qual){
  for(int i = 0; i < this->r->qual.l; ++i){
    this->r->qual.s[i] = (char)(qual[i]+33);
  }
}

int FastqFile::open_out(std::string filename){
  ofh = bgzf_open(filename.c_str(),"w"); //mode should be "wu" if
  ↪  uncompressed output is desired.
  if(tp->pool && bgzf_thread_pool(fh, tp->pool, tp->qsize) < 0){
    std::cerr << "Couldn't attach thread pool to file " << filename
    ↪  << std::endl;
  }
  return ofh == 0 ? -1 : 0;
}

int FastqFile::write(){
  std::string name = ks_c_str(&this->r->name);
  std::string seq = ks_c_str(&this->r->seq);
  std::string comment = ks_c_str(&this->r->comment);
  std::string qual = ks_c_str(&this->r->qual);
  // std::string name = this->r->name.s ?
  ↪  std::string(this->r->name.s) : "";
  // std::string seq = this->r->seq.s ? std::string(this->r->seq.s)
  ↪  : "";
  // std::string comment = this->r->comment.s ?
  ↪  std::string(this->r->comment.s) : "";
  // std::string qual = this->r->qual.s ?
  ↪  std::string(this->r->qual.s) : "";
  std::string s("@" + name + "\n" + seq + "\n+" + comment + "\n" +
  ↪  qual + "\n");
  return bgzf_write(ofh, s.c_str(), s.length());
}
```

```
// KmerSubsampler
bloom::Kmer KmerSubsampler::next_kmer(){
  if(cur_kmer < kmers.size()){
    return kmers[cur_kmer++]; //return the current kmer and advance
  } else {
    readseq = file->next_str();
    kmer.reset();
    if(readseq.empty()){
      this->not_eof = false;
      return kmer; //no more sequences
    } else {
      kmers.clear();
      for(size_t i = 0; i < readseq.length(); ++i){
        kmer.push_back(readseq[i]);
        if(i >= k-1){
          kmers.push_back(kmer);
        }
      }
      cur_kmer = 0; //reset current kmer
      total_kmers += kmers.size();
      return this->next_kmer(); //try again
    }
  }
}

bloom::Kmer KmerSubsampler::next(){
  bloom::Kmer kmer = this->next_kmer();
  if(this->not_eof){
#ifdef KBBQ_USE_RAND_SAMPLER
    double p = std::rand() / (double)RAND_MAX;
    if(p < this->d.p()){
#else
    if(d(rng)){ //sampled
#endif
      return kmer;
    }
    else{ //try again
      return this->next();
    }
  }
  return kmer; // empty
}
```

```
}

// FILE:/home/adam/code/kbbq/src/kbbq/recalibrateutils.cc


#include "recalibrateutils.hh"

using namespace htsiter;

namespace recalibrateutils{

void subsample_kmers(KmerSubsampler& s, bloom::Bloom& sampled){
  for(bloom::Kmer kmer = s.next(); s.not_eof; kmer = s.next()){
    if(kmer.valid()){
      sampled.insert(kmer);
    }
  }
}

void find_trusted_kmers(HTSFile* file, bloom::Bloom& trusted,
  const bloom::Bloom& sampled, std::vector<int> thresholds, int k)
{
  int n_trusted;
  bloom::Kmer kmer(k);
  //the order here matters since we don't want to advance the
  ↪  iterator if we're chunked out
  while(file->next() >= 0){
    readutils::CReadData read = file->get();
    read.infer_read_errors(sampled, thresholds, k);
    n_trusted = 0;
    kmer.reset();
    for(int i = 0; i < read.seq.length(); ++i){
      kmer.push_back(read.seq[i]);
      if(!read.errors[i]){
        ++n_trusted;
      }
      if(i >= k && !read.errors[i-k]){
        --n_trusted;
      }
      if(kmer.valid() && n_trusted == k){
        trusted.insert(kmer);
        //trusted kmer here
      }
    }
```

```
  }
}

covariateutils::CCovariateData get_covariatedata(HTSFile* file, const
↪  bloom::Bloom& trusted, int k){
  covariateutils::CCovariateData data;
#ifndef NDEBUG
  std::ifstream errorsin("../../adamjorr-Lighter/corrected.txt");
  int linenum = 0;
  std::string line = "";
#endif
  while(file->next() >= 0){
    readutils::CReadData read = file->get();
    read.get_errors(trusted, k, 6);
#ifndef NDEBUG
    //check that errors are same
    std::getline(errorsin, line);
    linenum++;
    std::vector<bool> lighter_errors(line.length(), false);
    std::transform(line.begin(), line.end(), lighter_errors.begin(),
      [](char c) -> bool {return (c == '1');});
    if( lighter_errors != read.errors){
      std::string message("Line num: " + std::to_string(linenum));
      // std::array<size_t,2> anchors =
      ↪  bloom::find_longest_trusted_seq(read.seq, trusted, k);
      // if(anchors[1] - anchors[0] - k + 1 >= k){ //number of
      ↪  trusted kmers >= k
      //   anchors[1] = bloom::adjust_right_anchor(anchors[1],
      ↪  read.seq, trusted, k);
      // }
      // std::cerr << "Anchors: [" << anchors[0] << ", " <<
      ↪  anchors[1] << "]";
      // std::cerr << " (npos is " << std::string::npos << ")\n";
      std::cerr << message << std::endl << "Errors : " ;
      for(const bool& v : read.errors){
        std::cerr << v;
      }
      std::cerr << std::endl;
      std::cerr << "Lighter: " ;
      for(const bool& v : lighter_errors){
        std::cerr << v;
      }
      std::cerr << std::endl;
      std::cerr << "Seq: " << read.seq << std::endl;
      // bloom::Kmer kmer(k);
      // for(const char& c :
      ↪  std::string("CAGAATAGAAAGATTTATAAATTAAATACTC")){
```

```
      //   std::cerr << c << ":" << seq_nt4_table[c] << ":" <<
      ↪  kmer.push_back(c) << "," ;
      // }
      // std::cerr << "Last kmer trusted?" <<
      ↪  trusted[kmer.hashed_prefix()].query(kmer.get_query()) <<
      ↪  std::endl;
    }
    assert(lighter_errors == read.errors);
#endif
    data.consume_read(read);
  }
  return data;
}

void recalibrate_and_write(HTSFile* in, const covariateutils::dq_t&
↪  dqs, std::string outfn){
  if(in->open_out(outfn) < 0){
    //error!! TODO
    return;
  }
  while(in->next() >= 0){
    readutils::CReadData read = in->get();
    std::vector<uint8_t> newquals = read.recalibrate(dqs);
    in->recalibrate(newquals);
    if(in->write() < 0){
      //error! TODO
      return;
    }
  }
}

}

// FILE:/home/adam/code/kbbq/src/kbbq/bloom.cc


#include <stdlib.h>
#include <cstring>
#include <cmath>
#include <random>
#include "bloom.hh"

namespace bloom
{

  constexpr blocked_bloom_filter::cell_type
  ↪  blocked_bloom_filter::cell_type_zero;
```

```
Bloom::Bloom(unsigned long long int projected_element_count,
 ↪  double fpr,
unsigned long long int seed): params(){
  // params(projected_element_count, fpr, seed), bloom(params){
  params.projected_element_count = projected_element_count;
  params.false_positive_probability = fpr;
  params.random_seed = seed;
  if(!params){
    throw std::invalid_argument("Error: Invalid bloom filter
     ↪  parameters. \
      Adjust parameters and try again.");
  }
  params.compute_optimal_parameters();
  bloom = bloom_type(params);
}

Bloom::~Bloom(){}

std::array<std::vector<size_t>,2>
 ↪  overlapping_kmers_in_bf(std::string seq, const Bloom& b, int k){
  bloom::Kmer kmer(k);
  std::vector<bool> kmer_present(seq.length()-k+1, false);
  std::vector<size_t> kmers_in(seq.length(), 0);
  std::vector<size_t> kmers_possible(seq.length(), 0);
  size_t incount = 0;
  size_t outcount = 0;
  for(size_t i = 0; i < seq.length(); ++i){
    kmer.push_back(seq[i]);
    if(i >= k-1){
      kmer_present[i-k+1] = kmer.valid() ? b.query(kmer) : false;
    }
#ifndef NDEBUG
    // if(i >= k-1){std::cerr << kmer << " " <<
     ↪  kmer_present[i-k+1] << std::endl;}
    if(i >= k-1 && seq == "AAGTGGGTTTCTCAGTATTTTATTCTTTTGATATTATCAT
     ↪  ACATGATACTATCGTCTTGATTTCTTCTTCAGAGAGTTTATTGTTGTTGTAGAAATACA
     ↪  ATTGATTTTTGTGTATTGATTTTGTATCCTGCAGCTTTGCTGAATTTTATTT"){
      std::string kmerstr(seq, i-k+1, k);
      std::cerr << kmerstr << " " << kmer_present[i-k+1] <<
       ↪  std::endl;
    }
#endif
  }
  for(size_t i = 0; i < seq.length(); ++i){
    if(i < seq.length() - k + 1){ //add kmers now in our window
      if(kmer_present[i]){
```

```
      ++incount;
    } else {
      ++outcount;
    }
  }
  if(i >= k){ //remove kmers outside our window
    if(kmer_present[i - k]){
      --incount;
    } else {
      --outcount;
    }
  }
  kmers_in[i] = incount;
  kmers_possible[i] = incount + outcount;
 }
 return {kmers_in, kmers_possible};
}

int nkmers_in_bf(std::string seq, const Bloom& b, int k){
 Kmer kmer(k);
 int count = 0;
 for (size_t i = 0; i < seq.length(); ++i) {
  kmer.push_back(seq[i]);
  if (kmer.valid()) { //we have a full k-mer
    if(b.query(kmer)){
      count++;
    }
  }
 }
 return count;
}

char get_next_trusted_char(const Kmer& kmer, const Bloom& trusted,
↪  bool reverse_test_order){
 const std::array<char, 4> test_bases = !reverse_test_order ?
  std::array<char, 4>{'A','C','G','T'}: std::array<char,
  ↪  4>{'T','G','C','A'};
 for(const char& c: test_bases){
  Kmer extra = kmer;
  extra.push_back(c);
  if(trusted.query(extra)){
    return c;
  }
 }
 return 0;
}
```

```
  std::array<size_t, 2> find_longest_trusted_seq(std::string seq,
  ↪  const Bloom& b, int k){
    Kmer kmer(k);
    size_t anchor_start, anchor_end, anchor_best, anchor_current;
    anchor_start = anchor_end = std::string::npos;
    anchor_best = anchor_current = 0;
    for(size_t i = 0; i < seq.length(); ++i){
      if(kmer.push_back(seq[i]) >= k){
        if(b.query(kmer)){
          anchor_current++; //length of current stretch
        } else { //we had a streak but the kmer is not trusted
          if(anchor_current > anchor_best){
            anchor_best = anchor_current;
            anchor_end = i - 1; // always > 0 because i >=
            ↪  kmer.size() >= k
            anchor_start = i + 1 - k - anchor_current;
          }
          anchor_current = 0;
        }
      } else if(anchor_current != 0){ //we had a streak but ran into
      ↪  a non-ATGC base
        if(anchor_current > anchor_best){
          anchor_best = anchor_current;
          anchor_end = i - 1; // always > 0 because i >= kmer.size()
          ↪  >= k
          anchor_start = i + 1 - k - anchor_current;
        }
        anchor_current = 0;
      }
    } //we got to the end
    if(anchor_current > anchor_best){
      anchor_best = anchor_current;
      anchor_end = std::string::npos;
      anchor_start = seq.length() + 1 - k - anchor_current;
    }
    return std::array<size_t,2>{{anchor_start, anchor_end}};
  }

  std::tuple<std::vector<char>,size_t,bool>
  ↪  find_longest_fix(std::string seq, const Bloom& trusted, int k,
  ↪  bool reverse_test_order){
#ifndef NDEBUG
    std::cerr << seq << std::endl;
#endif
    Kmer kmer(k);
    std::vector<char> best_c{};
    size_t best_i = 0;
```

```
    bool single = false; //this pair of flags will be used to
     ↪  determine whether
    bool multiple = false; //multiple corrections were considered
    char unfixed_char = seq[k-1];
    const std::array<char,4> test_bases = !reverse_test_order ?
      std::array<char, 4>{'A','C','G','T'}: std::array<char,
       ↪  4>{'T','G','C','A'};
    for(const char& c: test_bases){
      if(c == unfixed_char){continue;}
      seq[k-1] = c;
      kmer.reset();
      size_t i;
      size_t i_stop = std::max((size_t)2*k-1, seq.length()); //2k-1
       ↪  -> 2k?
      for(i = 0; i < i_stop; ++i){ //i goes to max(2*k-1,
       ↪  seq.length())
        char n = i < seq.length() ? seq[i] :
         ↪  get_next_trusted_char(kmer, trusted, reverse_test_order);
        if(n == 0){ // no next trusted kmer
          break;
        }
        kmer.push_back(n);
        if(i >= k-1){
          // std::cerr << kmer.size() << std::endl;
          if(kmer.valid()){
#ifndef NDEBUG
            std::cerr << std::string(kmer) << " " << i << " " <<
             ↪  trusted.query(kmer) << std::endl;
#endif
            if(!trusted.query(kmer)){
              break;
            } else { //we have a trusted kmer with this fix
              if(i == k-1){ //the first possible trusted kmer
                if(single){multiple = true;} //if we had one
                 ↪  already, set multiple
                single = true;
              }
            }
          } else { //a non-ATCG base must've been added
            break;
          }
        }
      }
      if(i > best_i){
#ifndef NDEBUG
        std::cerr << std::string(kmer) << " L" << std::endl;
#endif
```

```
        best_c.clear();
        best_c.push_back(c);
        best_i = i;
      } else if (i == best_i){
#ifndef NDEBUG
        std::cerr << std::string(kmer) << " T" << std::endl;
#endif
        best_c.push_back(c);
      }
    }
    return std::make_tuple(best_c, best_i, multiple);
  }

  long double calculate_phit(const Bloom& bf, long double alpha){
    long double fpr = bf.fprate();
    double exponent = alpha < 0.1 ? 0.2 / alpha : 2;
    long double pa = 1 - pow(1-alpha,exponent);
    return pa + fpr - fpr * pa;
  }

  uint64_t numbits(uint64_t numinserts, long double fpr){
    //m = - n * log2(fpr) / ln2
    return numinserts * (-log2(fpr) / log(2));
  }

  int numhashes(long double fpr){
    //k = -log2(fpr)
    return ceil(-log2(fpr));
  }

  //ensure anchor >= k - 1 before this.
  std::pair<size_t,bool> adjust_right_anchor(size_t anchor,
   ↪  std::string seq, const Bloom& trusted, int k){
    Kmer kmer(k);
    bool multiple = false; //multiple corrections were considered
    size_t modified_idx = anchor + 1;
    assert((anchor >= k - 1));
    for(size_t i = modified_idx - k + 1; i < modified_idx; ++i){
      kmer.push_back(seq[i]);
    }
    for(const char& c : {'A','C','G','T'}){
      if(seq[modified_idx] == c){continue;}
      bloom::Kmer new_kmer = kmer;
      new_kmer.push_back(c);
      //if this fix works, we don't need to adjust the anchor if it
       ↪  fixes all remaining kmers.
      //modified_idx + 1 to modified_idx + 1 + k - 1
```

```
      for(size_t i = 0; i <= k; ++i){ //668188
#ifndef NDEBUG
        std::cerr << "i: " << i << " anchor + 2 + i: " << anchor + 2
        ↪  + i << " len: " << seq.length() << std::endl;
#endif
        if(!trusted.query(new_kmer)){
          break;
        }
        //if we get to the end of the altered kmers and they're all
        ↪  fixed, the anchor is fine.
        if(modified_idx + i == seq.length()-1 || i == k){
#ifndef NDEBUG
          std::cerr << "First Try!" << std::endl;
#endif
          return std::make_pair(anchor, multiple);
        } else { //we haven't gotten to the end yet, so
        ↪  modified_idx+i+1 is valid.
          new_kmer.push_back(seq[modified_idx+i+1]);
        }
      }
    } // if we make it through this loop, we need to adjust the
    ↪  anchor.
    //we will test fixes starting with halfway through the last
    ↪  kmer to the end.
    //how much we're winding back the anchor; anchor-i-k must be >
    ↪  0.
    for(int i = k/2-1; i >= 0 && anchor > i+k-1; --i){
      kmer.reset();
      modified_idx = anchor-i;
      // size_t modified_idx = anchor+1-i+k-2;
      // for(size_t j = anchor + 1 - i - k; j < anchor + 1 - i;
      ↪  ++j){ //start can be -1 from this
      for(size_t j = modified_idx - k + 1; j < modified_idx; ++j){
        kmer.push_back(seq[j]);
      }
      for(const char& c: {'A','C','G','T'}){
        if(seq[modified_idx] == c){continue;}
        bloom::Kmer new_kmer = kmer;
        new_kmer.push_back(c);
#ifndef NDEBUG
        std::cerr << "i: " << i << " modified_idx: " << modified_idx
        ↪  << " kmer:" << seq.substr(modified_idx-k+1,k-1) << c <<
        ↪  std::endl;
#endif
        if(new_kmer.valid() && trusted.query(new_kmer)){
#ifndef NDEBUG
          std::cerr << "Trusted!" << std::endl;
```

```
#endif
          multiple = true;
          for(size_t j = 0; new_kmer.valid() &&
          trusted.query(new_kmer) &&
          modified_idx+1+j < seq.length() && j <= k/2; ++j){
            new_kmer.push_back(seq[modified_idx+1+j]);
            if(j == k/2 && new_kmer.valid() &&
             ↪  trusted.query(new_kmer)){
              //we went the full length
              return std::make_pair(modified_idx-1, multiple); //the
               ↪  idx before the new base that needs fixing
              //only return here if j == k/2, NOT if you run out of
               ↪  sequence!!!
            }
          }
        }
      }
    }
    //couldn't find a better adjustment
    return std::make_pair(anchor, multiple);
  }

  int biggest_consecutive_trusted_block(std::string seq, const Bloom&
   ↪  trusted, int k, int current_len){
    Kmer kmer(k);
    int in = 0;
    int out = 0;
    int len = 0;
    for (size_t i = 0; i < seq.length(); ++i) {
      kmer.push_back(seq[i]);
      if (i >= k-1){ //we have a full k-mer
        if(trusted.query(kmer)){ //if it's in, increment
          ++in;
        } else { //otherwise, reset but record the miss
          if(in > len){
            len = in;
          }
          in = 0;
          ++out;
          if(k - out < current_len){ //end if we have too many misses
            break;
          }
        }
      }
    }
    if(in > len){
      len = in;
```

```
    }
    return len;
  }

//end namespace
}

// FILE:/home/adam/code/kbbq/src/kbbq/covariateutils.cc


#include "covariateutils.hh"

namespace covariateutils{

  std::vector<long double> NormalPrior::normal_prior{};

  long double NormalPrior::get_normal_prior(size_t j){
    if(j >= normal_prior.size()){
      for(int i = normal_prior.size(); i < j+1; ++i){
        errno = 0;
        // long double prior_linspace = .9l *
        ↪  std::exp(-std::pow((long double)i,2.0l) * 2.0l)
        normal_prior.push_back(std::log(.9l *
        ↪  std::exp(-(std::pow(((long double)i/.5l),2.0l))/2.0l)));
        if(errno != 0){ //if an underflow happens just set the prior
        ↪  to smallest possible #
          normal_prior[i] = std::numeric_limits<long
          ↪  double>::lowest();
        }
      }
    }
    return normal_prior[j];
  }

  void CCovariate::increment(size_t idx, covariate_t value){
    this->increment(idx,value[0],value[1]);
  }

  void CCovariate::increment(size_t idx, unsigned long long err,
  ↪  unsigned long long total){
    this->at(idx)[0] += err;
    this->at(idx)[1] += total;
  }

  void CRGCovariate::consume_read(const readutils::CReadData& read){
    int rg = read.get_rg_int();
    if(this->size() <= rg){this->resize(rg+1);}
```

```
  std::vector<bool> nse = read.not_skipped_errors();
  this->increment(rg,
    std::accumulate(nse.begin(), nse.end(), 0),
    read.skips.size() - std::accumulate(read.skips.begin(),
     ↪  read.skips.end(), 0));
  // for(size_t i = 0; i < read.skips.size(); ++i){
  //   if(!read.skips[i]){
  //     this->increment(rg, !!read.errors[i], 1);
  //   }
  // }
}

rgdq_t CRGCovariate::delta_q(meanq_t prior){
  rgdq_t dq(this->size());
  for(int i = 0; i < this->size(); ++i){ //i is rgs here
    int map_q = 0; //maximum a posteriori q
    long double best_posterior = std::numeric_limits<long
     ↪  double>::lowest();
    for(int possible = 0; possible < KBBQ_MAXQ+1; ++possible){
      int diff = std::abs(prior[i] - possible);
      long double prior_prob = NormalPrior::get_normal_prior(diff);
      long double p = recalibrateutils::q_to_p(possible);
      long double loglike = log_binom_pmf((*this)[i][0] + 1,
       ↪  (*this)[i][1] + 2, p);
      long double posterior = prior_prob + loglike;
      if(posterior > best_posterior){
        map_q = possible;
        best_posterior = posterior;
      }
    }
    dq[i] = map_q - prior[i];
  }
  return dq;
}

void CQCovariate::consume_read(const readutils::CReadData& read){
  int rg = read.get_rg_int();
  int q;
  if(this->size() <= rg){this->resize(rg+1);}
  for(size_t i = 0; i < read.skips.size(); ++i){
    if(!read.skips[i]){
      q = read.qual[i];
      if(this->at(rg).size() <= q){this->at(rg).resize(q+1);}
      (*this)[rg].increment(q, std::array<unsigned long long,
       ↪  2>({read.errors[i], 1}));
    }
  }
```

```
}

qscoredq_t CQCovariate::delta_q(prior1_t prior){
  qscoredq_t dq(this->size());
  for(int i = 0; i < this->size(); ++i){ //i is rgs here
    dq[i].resize((*this)[i].size());
    for(int j = 0; j < (*this)[i].size(); ++j){ //j is q's here
      int map_q = 0; //maximum a posteriori q
      long double best_posterior = std::numeric_limits<long
      ↪  double>::lowest();
      for(int possible = 0; possible < KBBQ_MAXQ+1; possible++){
        int diff = std::abs(prior[i] - possible);
        long double prior_prob =
        ↪  NormalPrior::get_normal_prior(diff);
        long double p = recalibrateutils::q_to_p(possible);
        long double loglike = log_binom_pmf((*this)[i][j][0] + 1,
        ↪  (*this)[i][j][1] + 2, p);
        long double posterior = prior_prob + loglike;
        if(posterior > best_posterior){
          map_q = possible;
          best_posterior = posterior;
        }
      }
      dq[i][j] = map_q - prior[i];
    }
  }
  return dq;
}

void CCycleCovariate::consume_read(const readutils::CReadData&
↪  read){
  int rg = read.get_rg_int();
  int q;
  int cycle;
  if(this->size() <= rg){this->resize(rg+1);}
  size_t readlen = read.skips.size();
  for(size_t i = 0; i < readlen; ++i){
    if(!read.skips[i]){
      q = read.qual[i];
      if((*this)[rg].size() <= q){(*this)[rg].resize(q+1);}
      if((*this)[rg][q][read.second].size() <=
      ↪  i){(*this)[rg][q][read.second].resize(i+1);}
      (*this)[rg][q][read.second].increment(i, read.errors[i], 1);
    }
  }
}
```

```
cycledq_t CCycleCovariate::delta_q(prior2_t prior){
  cycledq_t dq(this->size()); //rg -> q -> fwd(0)/rev(1) -> cycle
  ↪  -> values
  for(int i = 0; i < this->size(); ++i){ //i is rgs here
    dq[i].resize((*this)[i].size());
    for(int j = 0; j < (*this)[i].size(); ++j){ //j is q's here
      for(int k = 0; k < 2; ++k){ //fwd/rev
        dq[i][j][k].resize((*this)[i][j][k].size());
        for(int l = 0; l < (*this)[i][j][k].size(); ++l){ //cycle
        ↪  value
          int map_q = 0; //maximum a posteriori q
          long double best_posterior = std::numeric_limits<long
          ↪  double>::lowest();
          for(int possible = 0; possible < KBBQ_MAXQ+1; possible++){
            int diff = std::abs(prior[i][j] - possible);
            long double prior_prob =
            ↪  NormalPrior::get_normal_prior(diff);
            long double p = recalibrateutils::q_to_p(possible);
            long double loglike =
            ↪  log_binom_pmf((*this)[i][j][k][l][0] + 1,
            ↪  (*this)[i][j][k][l][1] + 2, p);
            long double posterior = prior_prob + loglike;
            if(posterior > best_posterior){
              map_q = possible;
              best_posterior = posterior;
            }
          }
          dq[i][j][k][l] = map_q - prior[i][j];
        }
      }
    }
  }
  return dq;
}

void CDinucCovariate::consume_read(const readutils::CReadData&
↪  read, int minscore){
  int rg = read.get_rg_int();
  if(this->size() <= rg){this->resize(rg+1);}
  int q;
  for(size_t i = 1; i < read.seq.length(); ++i){
    q = read.qual[i];
    if(!read.skips[i] && nt_is_not_n(read.seq[i]) &&
      nt_is_not_n(read.seq[i-1]) && read.qual[i] >= minscore)
    {
      if((*this)[rg].size() <= q){(*this)[rg].resize(q+1);}
      if((*this)[rg][q].size() < 16){(*this)[rg][q].resize(16);}
```

```
      (*this)[rg][q].increment(dinuc_to_int(read.seq[i-1],
       ↪  read.seq[i]), read.errors[i], 1);
    }
  }
  // seq_nt16_table[256]: char -> 4 bit encoded (1/2/4/8)
  // seq_nt16_str[]: 4 bit -> char
  // seq_nt16_int[]: 4 bit -> 2 bits (0/1/2/3)
}

dinucdq_t CDinucCovariate::delta_q(prior2_t prior){
  dinucdq_t dq(this->size()); //rg -> q -> fwd(0)/rev(1) -> cycle
   ↪  -> values
  for(int i = 0; i < this->size(); ++i){ //i is rgs here
    dq[i].resize((*this)[i].size());
    for(int j = 0; j < (*this)[i].size(); ++j){ //j is q's here
      dq[i][j].resize((*this)[i][j].size());
      for(int k = 0; k < (*this)[i][j].size(); ++k){ //k is dinuc
        int map_q = 0; //maximum a posteriori q
        long double best_posterior = std::numeric_limits<long
         ↪  double>::lowest();
        for(int possible = 0; possible < KBBQ_MAXQ+1; possible++){
          int diff = std::abs(prior[i][j] - possible);
          long double prior_prob =
           ↪  NormalPrior::get_normal_prior(diff);
          long double p = recalibrateutils::q_to_p(possible);
          long double loglike = log_binom_pmf((*this)[i][j][k][0]
           ↪  + 1, (*this)[i][j][k][1] + 2, p);
          long double posterior = prior_prob + loglike;
          if(posterior > best_posterior){
            map_q = possible;
            best_posterior = posterior;
          }
        }
        dq[i][j][k] = map_q - prior[i][j];
      }
    }
  }
  return dq;
}

void CCovariateData::consume_read(readutils::CReadData& read, int
 ↪  minscore){
  //TODO: readd skips once the error correction code works
   ↪  properly
  // for(int i = 0; i < read.seq.length(); ++i){
```

```
    //   read.skips[i] = (read.skips[i] ||
    ↪  seq_nt16_int[seq_nt16_table[read.seq[i]]] >= 4 ||
    ↪  read.qual[i] < minscore);
    // }
    rgcov.consume_read(read);
    qcov.consume_read(read);
    cycov.consume_read(read);
    dicov.consume_read(read, minscore);
  }

  dq_t CCovariateData::get_dqs(){
    dq_t dq;
    std::vector<long double> expected_errors(this->qcov.size(),0);
    meanq_t meanq(this->qcov.size(),0);
    for(int rg = 0; rg < this->qcov.size(); ++rg){
      for(int q = 0; q < this->qcov[rg].size(); ++q){
        expected_errors[rg] += (recalibrateutils::q_to_p(q) *
        ↪  this->qcov[rg][q][1]);
      }
      meanq[rg] = recalibrateutils::p_to_q(expected_errors[rg] /
      ↪  this->rgcov[rg][1]);
    }
    dq.meanq = meanq;
    dq.rgdq = this->rgcov.delta_q(meanq);
    prior1_t rgprior(this->qcov.size());
    for(int rg = 0; rg < this->qcov.size(); ++rg){
      rgprior[rg] = meanq[rg] + dq.rgdq[rg];
    }
    dq.qscoredq = this->qcov.delta_q(rgprior);
    prior2_t qprior(this->qcov.size());
    for(int rg = 0; rg < this->qcov.size(); ++rg){
      for(int q = 0; q < this->qcov[rg].size(); ++q){
        qprior[rg].push_back(rgprior[rg] + dq.qscoredq[rg][q]);
      }
    }
    dq.cycledq = this->cycov.delta_q(qprior);
    dq.dinucdq = this->dicov.delta_q(qprior);
    return dq;
  }



}
// FILE:/home/adam/code/kbbq/src/kbbq/readutils.cc


#include "readutils.hh"
```

```
#include <algorithm>
#include <iterator>
#include <iostream>

KSEQ_DECLARE(BGZF*)

namespace readutils{

  std::unordered_map<std::string, std::string> CReadData::rg_to_pu{};
  std::unordered_map<std::string, int> CReadData::rg_to_int{};

  CReadData::CReadData(bam1_t* bamrecord, bool use_oq){ //TODO: flag
   ↪  to use OQ
    this->name = bam_get_qname(bamrecord);
    this->seq = bam_seq_str(bamrecord);
    if(use_oq){
      const uint8_t* oqdata = bam_aux_get(bamrecord, "OQ"); // this
       ↪  will be null on error
      //we should throw in that case
      if(oqdata == NULL){
        std::cerr << "Error: --use-oq was specified but unable to
         ↪  read OQ tag " <<
        "on read " << this->name << std::endl;
        if(errno == ENOENT){
          std::cerr << "OQ not found. Try again without the --use-oq
           ↪  option." << std::endl;
        } else if(errno == EINVAL){
          std::cerr << "Tag data is corrupt. Repair the tags and try
           ↪  again." << std::endl;
        }
        throw std::invalid_argument("Unable to read OQ tag.");
      }
      const std::string oq = bam_aux2Z(oqdata);
      std::transform(oq.cbegin(), oq.cend(),
       ↪  std::back_inserter(this->qual),
        [](const char& c) -> uint8_t {return c - 33;});
    } else {
      std::copy(bam_get_qual(bamrecord), bam_get_qual(bamrecord) +
       ↪  bamrecord->core.l_qseq,
        std::back_inserter(this->qual));
    }
    if(bam_is_rev(bamrecord)){
      //seq is already in fwd orientation
      std::reverse(this->qual.begin(), this->qual.end());
    }
    this->skips.resize(bamrecord->core.l_qseq, 0);
    uint8_t* rgdata = bam_aux_get(bamrecord, "RG");
```

```
  if(rgdata == NULL){
    std::cerr << "Error: Unable to read RG tag on read " <<
    ↪  this->name << std::endl;
    if(errno == ENOENT){
      std::cerr << "RG not found. " <<
      "Every read in the BAM must have an RG tag; add tags with " <<
      "samtools addreplacerg and try again." << std::endl;
    } else if(errno == EINVAL){
      std::cerr << "Tag data is corrupt. Repair the tags and try
      ↪  again." << std::endl;
    }
    throw std::invalid_argument("Unable to read RG tag.");;
  }
  this->rg = bam_aux2Z(bam_aux_get(bamrecord, "RG"));
  if(rg_to_pu.count(this->rg) == 0){
    rg_to_int[this->rg] = rg_to_int.size();
    // we just need something unique here.
    rg_to_pu[this->rg] = rg; //when loaded from the header this is
    ↪  actually a PU
  }
  this->second = bamrecord->core.flag & BAM_FREAD2; // 0x80
  this->errors.resize(bamrecord->core.l_qseq,0);
}
// if second is >1, that means infer.

CReadData::CReadData(kseq::kseq_t* fastqrecord, std::string rg, int
↪  second, std::string namedelimiter):
seq(fastqrecord->seq.s), skips(seq.length(), false),
↪  errors(seq.length(), false)
{
  // this->seq = std::string(fastqrecord->seq.s);
  // this->qual.assign(fastqrecord->qual.s, fastqrecord->qual.l);
  std::string quals(fastqrecord->qual.s);
  std::transform(quals.begin(), quals.end(),
  ↪  std::back_inserter(this->qual),
    [](char c) -> int {return c - 33;});
  // this->skips.resize(this->seq.length(), false);

  std::string fullname(fastqrecord->name.s);
  size_t current_pos = fullname.find(namedelimiter);
  std::string first_name = fullname.substr(0, current_pos);

  while(rg == "" && current_pos != std::string::npos){
    // if we need to find rg
    fullname = fullname.substr(current_pos+1); //get the right
    ↪  part, excluding the delimiter
```

```
    current_pos = fullname.find(namedelimiter); //reset the
    ↪  delimiter; this is npos if no more fields
    if(fullname.substr(0,3) == "RG:"){
      size_t last_colon = fullname.find_last_of(":", current_pos);
      ↪  // current_pos is last char to search
      rg = fullname.substr(last_colon+1, current_pos);
    }
  }
  this->rg = rg;

  if(second > 1){
    std::string tail = first_name.substr(first_name.length() - 2);
    second = (tail == "/2");
    if(second || tail == "/1"){
      first_name = first_name.substr(0, first_name.length() - 2);
    }
  }
  this->name = first_name;
  this->second = second;
  // this->errors.resize(this->seq.length(), false);

  if(rg_to_pu.count(this->rg) == 0){
    rg_to_int[this->rg] = rg_to_int.size();
    rg_to_pu[this->rg] = rg; //when loaded from the header this is
    ↪  actually a PU.
  }
}

void CReadData::load_rgs_from_bamfile(bam_hdr_t* header){
  std::string hdrtxt(header->text);
  size_t linedelim = hdrtxt.find('\n');
  // kstring_t val;
  // while (sam_hdr_find_tag_pos(header, "RG", i, "ID", val) ==
  ↪  0){
  //   std::string rgid(val.s);
  //   sam_hdr_find_tag_pos(header, "RG", i, "PU", val);
  //   std::string pu(val.s);
  //   if(rg_to_pu.count(rgid) == 0){
  //     rg_to_int[rgid] = rg_to_int.size();
  //     rg_to_pu[rgid] = pu;
  //   }
  // }
  while(linedelim != std::string::npos){
    if(hdrtxt.substr(0,3) == "@RG"){
      std::string id("");
      std::string pu("");
      std::string line = hdrtxt.substr(0, linedelim);
```

```
      size_t tokendelim = line.find_first_of("\t ");
      while(tokendelim != std::string::npos){
        if(line.substr(0,3) == "ID:"){
          //id
          id = line.substr(4, tokendelim);
        } else if (line.substr(0, 3) == "PU:"){
          //pu
          pu = line.substr(4, tokendelim);
        }
        line = line.substr(tokendelim);
        tokendelim = line.find_first_of("\t ");
      }
      if(id != ""){
        rg_to_int[id] = rg_to_int.size();
        rg_to_pu[id] = pu;
      }
    }
    hdrtxt = hdrtxt.substr(linedelim);
    linedelim = hdrtxt.find('\n');
  }
}

std::string CReadData::str_qual(){
  std::string str_qual;
  for(size_t i = 0; i < this->qual.size(); i++){
    str_qual.push_back(this->qual[i] + 33);
  }
  return str_qual;
}

std::string CReadData::canonical_name(){
  std::string suffix;
  if (this->second){
    suffix = "/2";
  }
  else{
    suffix = "/1";
  }
  return this->name + suffix;
}

std::vector<bool> CReadData::not_skipped_errors() const{
  std::vector<bool> unskipped_errs;
  std::transform(this->skips.cbegin(), this->skips.cend(),
  ↪  this->errors.cbegin(),
    std::back_inserter(unskipped_errs),
    [](const bool &s, const bool &e) -> bool {return (!s) && e;});
```

```
    return unskipped_errs;
  }

  void CReadData::infer_read_errors(const bloom::Bloom& b, const
   ↪  std::vector<int>& thresholds, int k){
    std::array<std::vector<size_t>,2> overlapping =
     ↪  bloom::overlapping_kmers_in_bf(this->seq, b, k);
    std::vector<size_t> in = overlapping[0];
    std::vector<size_t> possible = overlapping[1];
    for(size_t i = 0; i < errors.size(); ++i){
      this->errors[i] = (in[i] <= thresholds[possible[i]] ||
       ↪  this->qual[i] <= INFER_ERROR_BAD_QUAL);
#ifndef NDEBUG
      if(possible[i] > k){std::cerr << "seq:" << this->seq << " " <<
       ↪  possible[i] << "WARNING: Invalid i: " << i << std::endl;}
      // if(i>=k-1 && std::string(this->seq, i-k+1, k) ==
       ↪  "CCCCCCCCCTCGCCCCCCCCCCCCCCCCCC"){
      //   std::cerr << "seq: " << this->seq << "\n";
      //   std::cerr << "in: ";
      //   std::copy(in.begin()+i-k+1, in.begin()+i+1,
       ↪  std::ostream_iterator<size_t>(std::cerr, ", "));
      //   std::cerr << "\npossible: ";
      //   std::copy(possible.begin()+i-k+1, possible.begin()+i+1,
       ↪  std::ostream_iterator<size_t>(std::cerr, ", "));
      //   std::cerr << "\nerrors: ";
      //   std::copy(this->errors.begin()+i-k+1,
       ↪  this->errors.begin()+i+1,
       ↪  std::ostream_iterator<bool>(std::cerr, ", "));
      //   std::cerr << std::endl;
      // }
#endif
    }
  }

  size_t CReadData::correct_one(const bloom::Bloom& t, int k){
    int best_fix_len = 0;
    char best_fix_base;
    size_t best_fix_pos = std::string::npos;
    for(size_t i = 0; i < this->seq.length(); ++i){
      std::string original_seq(this->seq); //copy original
      for(const char& c : {'A','C','G','T'}){
        if(this->seq[i] == c){continue;}
        original_seq[i] = c;
        size_t start = i > k - 1 ? i - k + 1 : 0;
        //test a kmer to see whether its worth counting them all
        //i'm not sure any performance gain is worth it, but this is
         ↪  how Lighter does it
```

```
        size_t magic_start = i > k/2 - 1 ? std::min(i - k/2 + 1,
        ↪  original_seq.length()-k) : 0;
        bloom::Kmer magic_kmer(k);
        for(size_t j = magic_start; j <= magic_start + k - 1; ++j){
          magic_kmer.push_back(original_seq[j]);
        }
        //
        if(t.query(magic_kmer)){
          //94518
          int n_in = bloom::biggest_consecutive_trusted_block(
            original_seq.substr(start, 2*k - 1),t,k,best_fix_len);
#ifndef NDEBUG
          std::cerr << "Found a kmer: " << magic_kmer << " i: " << i
          ↪  << " Fix len: " << n_in << std::endl;
#endif
          if(n_in > best_fix_len){ //94518
            best_fix_base = c;
            best_fix_pos = i;
            best_fix_len = n_in;
          } else if(n_in == best_fix_len && this->qual[i] <
          ↪  this->qual[best_fix_pos]){
            best_fix_base = c;
            best_fix_pos = i;
          }
        }
      }
    }
    if(best_fix_len > 0){
      this->seq[best_fix_pos] = best_fix_base;
    }
    return best_fix_pos;
  }

  //this is a chonky boi
  std::vector<bool> CReadData::get_errors(const bloom::Bloom&
  ↪  trusted, int k, int minqual, bool first_call){
    std::string original_seq(this->seq);
    size_t bad_prefix = 0;
    size_t bad_suffix = std::string::npos;
    bool multiple = false; //whether there were any ties
#ifndef NDEBUG
    std::cerr << "Correcting seq: " << original_seq << std::endl;
#endif
    std::array<size_t,2> anchor =
    ↪  bloom::find_longest_trusted_seq(this->seq, trusted, k);
#ifndef NDEBUG
```

```
    std::cerr << "Initial anchors: [" << anchor[0] << ", " <<
    ↪  anchor[1] << "]" << std::endl;
#endif
    if(anchor[0] == std::string::npos){ //no trusted kmers in this
    ↪  read.
      multiple = true;
      size_t corrected_idx = this->correct_one(trusted, k);
      if(corrected_idx == std::string::npos){
        return this->errors;
      } else {
        anchor = bloom::find_longest_trusted_seq(this->seq, trusted,
        ↪  k);
#ifndef  NDEBUG
        std::cerr << "Created anchor: [" << anchor[0] << ", " <<
        ↪  anchor[1] << "]" << std::endl;
#endif
        this->errors[corrected_idx] = true;
      }
    }
    if(anchor[0] == 0 && anchor[1] == std::string::npos){ //all
    ↪  kmers are trusted
      return this->errors;
    }
    //we're guaranteed to have a valid anchor now.
    //min is in case anchor[1] is npos.
    size_t anchor_len = std::min(anchor[1], this->seq.length()-1) +
    ↪  1 - anchor[0];
    bool corrected = false; //whether there were any corrections
    //right side
    if(anchor[1] != std::string::npos){
      //number of trusted kmers >= k
      if(anchor_len - k + 1 >= k){ //number of trusted kmers
        bool current_multiple;
        std::tie(anchor[1], current_multiple) =
        ↪  bloom::adjust_right_anchor(anchor[1], this->seq, trusted,
        ↪  k);
#ifndef NDEBUG
        std::cerr << "Adjust R Multiple: " << current_multiple <<
        ↪  std::endl;
#endif
        multiple = multiple || current_multiple;
      }
      for(size_t i = anchor[1] + 1; i < this->seq.length();){
        size_t start = i - k + 1; //seq containing all kmers that
        ↪  are affected
        std::vector<char> fix;
        size_t fixlen;
```

```
        bool current_multiple;
        std::tie(fix, fixlen, current_multiple) =
         ↪  bloom::find_longest_fix(this->seq.substr(start,
         ↪  std::string::npos), trusted, k);
#ifndef NDEBUG
        std::cerr << "R fix Multiple: " << current_multiple <<
         ↪  std::endl;
#endif
        multiple = multiple || current_multiple;
        size_t next_untrusted_idx = start + fixlen;
        if(next_untrusted_idx > i){
          if(fix.size() > 1){
            // multiple = true; //we take care of this in function
            // i = index of erroneous base
            //to = ( i + kmerLength - 1 < readLength ) ? i +
             ↪  kmerLength - 1 : readLength - 1 ;
            //to = inclusive biggest index the fix can possibly go
             ↪  to. (= largest_possible_idx)
            //maxto = largest index the fix actually goes to
             ↪  exclusive; =next_untrusted_idx
            //if( maxTo <= to || to - i + 1 < kmerLength ) ...
            //i + k
            multiple = true;
            size_t largest_possible_idx = std::min(i + k - 1,
             ↪  this->seq.length()-1);
#ifndef NDEBUG
            std::cerr << "next_untrusted_idx: " << next_untrusted_idx
             ↪  << std::endl;
            std::cerr << "largest_possible_idx (to): " <<
             ↪  largest_possible_idx << std::endl;
            std::cerr << "largest_possible_idx-i+1: " <<
             ↪  largest_possible_idx-i+1 << std::endl;
#endif          //20629746
            if(next_untrusted_idx <= largest_possible_idx ||
             ↪  largest_possible_idx - i + 1 < k){
              // size_t trimstart = next_untrusted_idx;
              bad_suffix = i; //readlength - trimstart
#ifndef NDEBUG
              //if there's a tie and we haven't gone the max number
               ↪  of kmers, end correction
              std::cerr << "i: " << i << " next_untrusted_idx " <<
               ↪  next_untrusted_idx << std::endl;
              std::cerr << "Fixlen is " << fixlen << " Fix: ";
              for(char c : fix){
                std::cerr << c << " ";
              }
              std::cerr << std::endl;
```

```
            std::cerr << "Tie and fix not long enough; ending
             ↪  correction early!" << std::endl;
#endif
            break;
          }
        } else {
          this->seq[i] = fix[0];
          this->errors[i] = true;
        }
        corrected = true;
#ifndef NDEBUG
        std::cerr << "Error detected at position " << i << ".
         ↪  Advancing " << fixlen - k + 1 << "." << std::endl;
#endif
        i += fixlen - k + 1; // i = next_untrusted_idx
      } else {
        //couldn't find a fix; skip ahead and try again if long
         ↪  enough
#ifndef NDEBUG
        std::cerr << "Couldn't fix position " << i << " Cutting
         ↪  read and trying again." << std::endl;//". Skipping
         ↪  ahead " << k - 1 << "." << std::endl;
#endif
        bad_suffix = i;
        break;
        // i += k-1; //move ahead and make i = k-1 as the first
         ↪  base in the new kmer
        // if(this->seq.length() - i + k <= (seq.length()/2) ||
         ↪  this->seq.length() - i + k <= 2*k ){
        //   //sequence not long enough. end this side.
        //   break;
        // }
      }
    }
  }
  //left side
  if(anchor[0] != 0){
    //the bad base is at anchor[0]-1, then include the full kmer
     ↪  for that base.
    // std::string sub = this->seq.substr(0, anchor[0] - 1 + k);
    std::string revcomped(this->seq.length(), 'N');
    std::transform(this->seq.rbegin(), this->seq.rend(),
     ↪  revcomped.begin(),
      [](char c) -> char {return seq_nt16_str[seq_nt16_table[('0'
       ↪  + 3-seq_nt16_int[seq_nt16_table[c]])]];});
    //if num of trusted kmers >= k, see if anchor needs adjusting.
    if(anchor_len - k + 1 >= k){
```

```
        size_t left_adjust;
        bool current_multiple;
        std::tie(left_adjust, current_multiple) = bloom::adjust_right
        ↪  _anchor(revcomped.length()-anchor[0]-1, revcomped,
        ↪  trusted, k);
        anchor[0] = revcomped.length()-left_adjust-1; //change back
        ↪  to original coordinates
#ifndef NDEBUG
        std::cerr << "Adjust L Multiple: " << current_multiple <<
        ↪  std::endl;
#endif
        multiple = multiple || current_multiple;
      }
      //
      for(int i = anchor[0] - 1; i >= 0;){ //index of erroneous base
      ↪  in original seq
        int j = revcomped.length()-i-1; //index of erroneous base in
        ↪  reversed seq
        size_t start = j - k + 1; //seq containing all kmers that
        ↪  are affected
        //but [j -k + 1, npos) in reverse space.
        std::string sub = revcomped.substr(start, std::string::npos);
        ↪  //get the right subsequence
        std::vector<char> fix;
        size_t fixlen;
        bool current_multiple;
        std::tie(fix, fixlen, current_multiple) =
        ↪  bloom::find_longest_fix(sub, trusted, k, true); //155392
        ↪  TODO: add reverse_test to adjust_right_anchor
#ifndef NDEBUG
        std::cerr << "L Fix Multiple: " << current_multiple <<
        ↪  std::endl;
#endif
        multiple = multiple || current_multiple;
        //new i is return value + start // i += r -k + 1
        //new j should be return value + start // j += r - k + 1
        // new j should be fixlen + start; j += fixlen - k + 1
        // j = fixlen + start; j = j - k + 1 + fixlen; j += fixlen
        ↪  -k + 1
        size_t next_untrusted_idx = start + fixlen; // next
        ↪  untrusted idx in j space
        if( next_untrusted_idx > j){
          if(fix.size() > 1){
            multiple = true; // we don't have to do this here
            ↪  because we do it in function
            //this value needs to be fixed i think
```

```
            //to = ( i - kmerLength + 1 < 0 ) ? 0 : ( i - kmerLength
            ↪  + 1 ) ;
            //( minTo >= to || i - to + 1 < kmerLength )
            //Line num: 21537
            // if(next_untrusted_idx <= std::max(start + (size_t)2*k
            ↪  - 1, revcomped.length())){
            size_t largest_possible_idx = std::min(j + (size_t)k -
            ↪  1, revcomped.length()-1);
            if(next_untrusted_idx <= largest_possible_idx ||
            ↪  largest_possible_idx - j + 1 < k){ //595573
              //if there's a tie and we haven't gone the max number
              ↪  of kmers, end correction
              bad_prefix = i;
              break;
            }
          } else {
            revcomped[j] = fix[0];
            this->errors[i] = true;
          }
          corrected = true;
#ifndef NDEBUG
          std::cerr << "next_untrusted_idx: " << next_untrusted_idx
          ↪  << " j: " << j << std::endl;
          std::cerr << "Error detected at position " << i << ".
          ↪  Advancing " << next_untrusted_idx-j << " " << (fixlen -
          ↪  k + 1) << "." << std::endl;
#endif
          i -= next_untrusted_idx - j; //fixlen - k + 1;
        } else {
          //couldn't find a fix; skip ahead and try again if long
          ↪  enough
#ifndef NDEBUG
          std::cerr << "Couldn't fix position " << i << "Cutting read
          ↪  and trying again." << std::endl;//". Skipping ahead "
          ↪  << k - 1 << "." << std::endl;
#endif
          bad_prefix = i; //the last position in the new read
          break;
          // i -= k-1;
          // if(i + k <= (seq.length()/2) || i + k <= 2*k ){
          //   //sequence not long enough. end this side.
          //   break;
          // }
        }
      }
    }
#ifndef NDEBUG
```

```
    std::cerr << "Anchors: [" << anchor[0] << ", " << anchor[1] <<
    ↪  "]" << std::endl;
#endif


    // check for overcorrection and fix it
    if(corrected){
      bool adjust = true;
      //check that no trusted kmers were "fixed"
      bloom::Kmer kmer(k);
      size_t trusted_start = std::string::npos;
      size_t trusted_end = std::string::npos;
      for(size_t i = 0; i < original_seq.length() && adjust == true;
      ↪  ++i){
        kmer.push_back(original_seq[i]);
        if(kmer.valid() && trusted.query(kmer)){
          trusted_start = std::min(trusted_start,i-k+1);
          trusted_end = i;
          // std::cerr << "Trusted: " << trusted_start << " " <<
          ↪  trusted_end << std::endl;
        } else {
          if(i > trusted_end){
            //we clear everything from beginning to end
            for(size_t j = trusted_start; j <= trusted_end; ++j){
              if(this->errors[j]){
                adjust = false;
                break;
              }
            }
            trusted_start = std::string::npos;
            trusted_end = std::string::npos;
          }
        }
      }
      //if we get to the end but have a trusted block
      //we don't actually trust that block :|
      // if(trusted_end != std::string::npos){
      //   for(size_t j = trusted_start; j <= trusted_end; ++j){
      //     if(this->errors[j]){
      //       std::cerr << "(2) Problem: j: " << j << " " <<
      ↪  trusted_start << " " << trusted_end << std::endl;
      //       adjust = false;
      //       break;
      //     }
      //   }
      // }
#ifndef NDEBUG
```

```
      std::cerr << "Adjust: " << adjust << " Multiple: " << multiple
      ↪  << std::endl;
#endif
      adjust = adjust && !multiple;//made it through the loop and no
      ↪  ties during correction
      // std::cerr << "Read corrected. Adjust threshold? " <<
      ↪  adjust << std::endl;
#ifndef NDEBUG
      std::cerr << "Errors before adjustment: ";
      for(const bool& b: this->errors){
        std::cerr << b;
      }
      std::cerr << std::endl;
      std::cerr << "Seq After Correction: " << seq << std::endl;
#endif
      int ocwindow = 20;
      int base_threshold = 4;
      int threshold = base_threshold;
      double occount = 0;
      //check for overcorrection
      std::vector<int> overcorrected_idx; //push_back overcorrected
      ↪  indices in order
      //then from overcorrected.begin() - k to overcorrected.end()
      ↪  + k should all be reset.
      for(int i = 0; i < this->seq.length(); ++i){
        if(this->errors[i] &&
        ↪  seq_nt16_int[seq_nt16_table[original_seq[i]]] < 4){
        ↪  //increment correction count
          if(this->qual[i] <= minqual){
            occount += 0.5;
          } else {
            ++occount;
          }
        }
        if(i >= ocwindow && this->errors[i-ocwindow] &&
        ↪  seq_nt16_int[seq_nt16_table[original_seq[i-ocwindow]]] <
        ↪  4){ //decrement count for not in window
          if(this->qual[i-ocwindow] <= minqual){
            occount -= 0.5;
          } else {
            --occount;
          }
        }
        //set threshold
        threshold = adjust && i >= ocwindow && i + ocwindow - 1 <
        ↪  this->seq.length() ?
          base_threshold + 1 : base_threshold;
```

```
        //determine if overcorrected
#ifndef NDEBUG
        std::cerr << "Occount: " << occount << " Threshold: " <<
        ↪  threshold;
        std::cerr << " Seq: " << original_seq[i] << " (" <<
        ↪  seq_nt16_int[seq_nt16_table[original_seq[i]]];
        std::cerr << ")" << " Q: " << +this->qual[i] << std::endl;
#endif
        if(occount > threshold && this->errors[i]){
          overcorrected_idx.push_back(i);
        }

      }
#ifndef NDEBUG
      std::cerr << "Overcorrected indices (" <<
      ↪  overcorrected_idx.size() << "): ";
      std::copy(overcorrected_idx.begin(), overcorrected_idx.end(),
      ↪  std::ostream_iterator<int>(std::cerr, ", "));
      std::cerr << std::endl;
#endif
      //Line num: 23026
      for(int oc_idx : overcorrected_idx){
        if(this->errors[oc_idx]){ //overcorrected idx hasn't been
        ↪  addressed yet
          int start = oc_idx-k+1; //the beginningmost position to
          ↪  check
          start = start >= 0? start : 0;
          int end = oc_idx+k; //the endmost position to check
          end = end < this->seq.length() ? end : this->seq.length();
          //we start iteration but we need to unfix anything within
          ↪  k of an overcorrected position
          //OR within k of one of those fixed positions.
          for(int i = start; i < end; ++i){
            if(this->errors[i]){
              this->errors[i] = false;
              //changint the end must come before the start change
              ↪  because the start
              //change changes i!
              if(i+k > end){ //change the end if we need to
                end = i+k < this->seq.length() ? i+k :
                ↪  this->seq.length();
              }
              //i will be 1 greater than start, so rather than i-k+1
              ↪  we have i-k.
              if(i-k < start){ //go back a bit if we need to; +1
              ↪  comes from the loop
```

```
                i = i-k+1 >= 0 ? i-k : -1; //+1 will come from the
                ↪  loop
                start = i;
              }
            }
          }
        }
      }
    }
    if(first_call && bad_prefix > 0 && (bad_prefix >=
    ↪  this->seq.length() / 2 || bad_prefix >= 2*k)){
#ifndef NDEBUG
      std::cerr << "bad_prefix: " << bad_prefix << std::endl;
#endif
      CReadData subread = this->substr(0, bad_prefix+1); //2nd
      ↪  argument is length
      std::vector<bool> suberrors = subread.get_errors(trusted, k,
      ↪  minqual, false);
      std::copy(suberrors.begin(), suberrors.end(),
      ↪  this->errors.begin());
    }
    if(first_call && bad_suffix < std::string::npos && bad_suffix <
    ↪  this->seq.length() &&
    (this->seq.length()-bad_suffix > this->seq.length()/2 ||
    ↪  this->seq.length()-bad_suffix > 2*k)){
#ifndef NDEBUG
      std::cerr << "bad_suffix: " << bad_suffix << std::endl;
#endif
      CReadData subread = this->substr(bad_suffix, std::string::npos);
      std::vector<bool> suberrors = subread.get_errors(trusted, k,
      ↪  minqual, false);
      std::copy(suberrors.begin(), suberrors.end(),
      ↪  this->errors.begin()+bad_suffix);
    }
    if(std::find(this->errors.begin(), this->errors.end(), true) !=
    ↪  this->errors.end()){
      corrected = true;
    }

    this->seq = original_seq;
    return this->errors;
  }

  std::vector<uint8_t> CReadData::recalibrate(const
  ↪  covariateutils::dq_t& dqs, int minqual) const{
    std::vector<int> recalibrated;
```

```
    std::copy(this->qual.begin(), this->qual.end(),
    ↪  std::back_inserter(recalibrated));
    int rg = this->get_rg_int();
    for(int i = 0; i < this->seq.length(); ++i){
      uint8_t q = this->qual[i];
      if(q >= minqual){
        recalibrated[i] = dqs.meanq[rg] + dqs.rgdq[rg] +
        ↪  dqs.qscoredq[rg][q] +
          dqs.cycledq[rg][q][this->second][i];
        if(i > 0){
          int first = seq_nt16_int[seq_nt16_table[this->seq[i-1]]];
          int second = seq_nt16_int[seq_nt16_table[this->seq[i]]];
          if(first < 4 && second < 4){
            int8_t dinuc = 15 & ((first << 2) | second); //1111 &
            ↪  (xx00|00xx)
            recalibrated[i] += dqs.dinucdq[rg][q][dinuc];
          }
        }
      }
    }
    std::vector<uint8_t> ret;
    std::transform(recalibrated.begin(), recalibrated.end(),
    ↪  std::back_inserter(ret),
      [](int q)->uint8_t {return q < 0 ? 0 : KBBQ_MAXQ < q ?
      ↪  KBBQ_MAXQ : q;}); //std::clamp in c++17
    return ret;
  }

  CReadData CReadData::substr(size_t pos, size_t count) const{
    CReadData ret = (*this);
    ret.seq = ret.seq.substr(pos, count);
    size_t len = ret.seq.size();
    ret.qual.resize(len);
    ret.skips.resize(len);
    ret.errors.resize(len);
    std::copy(this->qual.cbegin()+pos, this->qual.cbegin()+pos+len,
    ↪  ret.qual.begin());
    std::copy(this->skips.cbegin()+pos, this->skips.cbegin()+pos+len,
    ↪  ret.skips.begin());
    std::copy(this->errors.cbegin()+pos,
    ↪  this->errors.cbegin()+pos+len, ret.errors.begin());
    return ret;
  }


}
```

APPENDIX G

CODE TO REPRODUCE ANALYSES IN CHAPTER 4

G.1 Code to Call Variants With CHM1-CHM13 Data

```
# FILE:chm1-chm13_variants/filtering/Makefile


DATADIR = ../../../../
VCFIN = $(DATADIR)/chr1.vcf.gz
BEDIN = $(DATADIR)/chr1_confident.bed.gz
REFIN = $(DATADIR)/chr1.renamed.fa
HAPDIPJS = $(HOME)/bin/hapdip/hapdip.js

INPUTINCREMENT = 20
INPUTFIRST = 0
INPUTLAST = 100

SHELL=bash
FPRS := $(shell seq $(INPUTFIRST) $(INPUTINCREMENT) $(INPUTLAST))
#FNR_FPR
ADD_FPRS = $(addsuffix _$(FPR),$(FPRS))
ERRATES := $(foreach FPR,$(FPRS),$(ADD_FPRS))
ERRATES := $(ERRATES) cbbq
ERVCFS := $(addprefix ../,$(addsuffix .vcf.gz,$(ERRATES)))
FLTVCFS := $(addsuffix _flt.vcf.gz,$(ERRATES))
RTGCALLS := $(addsuffix _flt/calls.vcf.gz,$(ERRATES))
SNPSUMS := $(addsuffix _flt/hapdip_summary.txt,$(ERRATES))
OUTSUMS := $(addsuffix _flt.summary.txt,$(ERRATES))

#index helpers
%.bed.gz.tbi: %.bed.gz
  tabix -p bed $<

%.vcf.gz.tbi: %.vcf.gz
  bcftools index -t $<

%.bam.bai: %.bam
  samtools index $<

%.fa.sdf: %.fa
  rtg format -o $@ $<

.PHONY: all

.DEFAULT: all

.PRECIOUS:

.SECONDARY:
```

```
all: $(OUTSUMS)

#filter: only snps
#filter: DP > 35 & DP < 65 (+-2 SDEV from mean of ~50)
#filter: QUAL > 75; this is ~bottom 1% and the maximum F
#   value from the previous analysis is between 75 and 150.
$(FLTVCFS): %_flt.vcf.gz: ../%.vcf.gz $(BEDIN)
  bcftools view -v snps -i'DP>35 & DP < 65 & QUAL > 75' -R $(word 2,
  ↪  $^) -Oz -o $@ $<

raw_flt.vcf.gz: ../raw.vcf.gz $(BEDIN)
  bcftools view -v snps -i'DP>35 & DP < 65 & QUAL > 75' -R $(word 2,
  ↪  $^) -Oz -o $@ $<

#benchmarking result directories
$(RTGCALLS): %/calls.vcf.gz: %.vcf.gz $(VCFIN) $(BEDIN) $(REFIN).sdf
↪  %.vcf.gz.tbi
  rm -rf $(dir $@) && rtg vcfeval -b $(word 2, $^) -c $< -e $(word 3,
  ↪  $^) -o $(dir $@) -t $(word 4, $^) --output-mode=annotate -T 4

raw_flt/calls.vcf.gz: raw_flt.vcf.gz $(VCFIN) $(BEDIN) $(REFIN).sdf
↪  raw_flt.vcf.gz.tbi
  rm -rf $(dir $@) && rtg vcfeval -b $(word 2, $^) -c $< -e $(word 3,
  ↪  $^) -o $(dir $@) -t $(word 4, $^) --output-mode=annotate -T 4

$(SNPSUMS): %/hapdip_summary.txt: %/calls.vcf.gz
  k8 $(HAPDIPJS) rtgeval $(BEDIN) $(dir $@) > $@

raw_flt/hapdip_summary.txt: raw_flt/calls.vcf.gz
  k8 $(HAPDIPJS) rtgeval $(BEDIN) $(dir $@) > $@

$(OUTSUMS): %.summary.txt: %/hapdip_summary.txt
  cp $< $@

raw_flt.summary.txt: raw_flt/hapdip_summary.txt
  cp $< $@

# FILE:chm1-chm13_variants/Makefile


DATADIR = ../../../
VCFIN = $(DATADIR)/chr1.vcf.gz
BEDIN = $(DATADIR)/chr1_confident.bed.gz
REFIN = $(DATADIR)/chr1.renamed.fa
RAWBAM =  ../chr1_only_confident.fixmate.sorted.bam
PROCESSEDBAM = raw.bam
```

```
HAPDIPJS = $(HOME)/bin/hapdip/hapdip.js

INPUTINCREMENT = 20
INPUTFIRST = 0
INPUTLAST = 100

SHELL=bash
FPRS := $(shell seq $(INPUTFIRST) $(INPUTINCREMENT) $(INPUTLAST))
#FNR_FPR
ADD_FPRS = $(addsuffix _$(FPR),$(FPRS))
ERRATES := $(foreach FPR,$(FPRS),$(ADD_FPRS))
ERRATES := $(ERRATES) cbbq
ERBAMS := $(addprefix ../,$(addsuffix .recal.bam,$(ERRATES)))
ERVCFS := $(addsuffix .vcf.gz,$(ERRATES))
RTGCALLS := $(addsuffix /calls.vcf.gz,$(ERRATES))
SNPSUMS := $(addsuffix /hapdip_summary.txt,$(ERRATES))
OUTSUMS := $(addsuffix .summary.txt,$(ERRATES))

#index helpers
%.bed.gz.tbi: %.bed.gz
  tabix -p bed $<

%.vcf.gz.tbi: %.vcf.gz
  bcftools index -t $<

%.bam.bai: %.bam
  samtools index $<

%.fa.sdf: %.fa
  rtg format -o $@ $<

.PHONY: all

.DEFAULT: all

.PRECIOUS:

.SECONDARY:

all: $(OUTSUMS)

$(ERVCFS): %.vcf.gz: ../%.recal.bam $(REFIN) $(BEDIN)
  gatk HaplotypeCaller -R $(word 2, $^) -I $< -L $(word 3, $^)
  ↪  --native-pair-hmm-threads 16 -O $@ --QUIET true

$(PROCESSEDBAM): $(RAWBAM)
  gatk RevertBaseQualityScores -I $< -O $@
```

```
raw.vcf.gz: $(PROCESSEDBAM) $(REFIN) $(BEDIN)
  gatk HaplotypeCaller -R $(word 2, $^) -I $< -L $(word 3, $^)
   ↪  --native-pair-hmm-threads 16 -O $@ --QUIET true

#benchmarking result directories
$(RTGCALLS): %/calls.vcf.gz: %.vcf.gz $(VCFIN) $(BEDIN) $(REFIN).sdf
  rm -rf $(dir $@) && rtg vcfeval -b $(word 2, $^) -c $< -e $(word 3,
   ↪  $^) -o $(dir $@) -t $(word 4, $^) --output-mode=annotate -T 1

raw/calls.vcf.gz: raw.vcf.gz $(VCFIN) $(BEDIN) $(REFIN).sdf
  rm -rf $(dir $@) && rtg vcfeval -b $(word 2, $^) -c $< -e $(word 3,
   ↪  $^) -o $(dir $@) -t $(word 4, $^) --output-mode=annotate -T 8

$(SNPSUMS): %/hapdip_summary.txt: %/calls.vcf.gz
  k8 $(HAPDIPJS) rtgeval $(BEDIN) $(dir $@) > $@

raw/hapdip_summary.txt: raw/calls.vcf.gz
  k8 $(HAPDIPJS) rtgeval $(BEDIN) $(dir $@) > $@

$(OUTSUMS): %.summary.txt: %/hapdip_summary.txt
  cp $< $@

raw.summary.txt: raw/hapdip_summary.txt
  cp $< $@

# FILE:chm1-chm13_variants/rocs/Makefile


DATADIR = ../../../../
VCFIN = $(DATADIR)/chr1.vcf.gz
BEDIN = $(DATADIR)/chr1_confident.bed.gz
REFIN = $(DATADIR)/chr1.renamed.fa
HAPDIPJS = $(HOME)/bin/hapdip/hapdip.js
TAG = QUAL

INPUTINCREMENT = 20
INPUTFIRST = 0
INPUTLAST = 100

SHELL=bash
FPRS := $(shell seq $(INPUTFIRST) $(INPUTINCREMENT) $(INPUTLAST))
#FNR_FPR
ADD_FPRS = $(addsuffix _$(FPR),$(FPRS))
ERRATES := $(foreach FPR,$(FPRS),$(ADD_FPRS))
ERRATES := $(ERRATES) raw cbbq
RTGROCS := $(addsuffix _$(TAG)/snp_roc.tsv.gz,$(ERRATES))
```

```
OUTROCS := $(addsuffix _$(TAG).snp_roc.tsv.gz,$(ERRATES))

#index helpers
%.bed.gz.tbi: %.bed.gz
  tabix -p bed $<

%.vcf.gz.tbi: %.vcf.gz
  bcftools index -t $<

%.bam.bai: %.bam
  samtools index $<

%.fa.sdf: %.fa
  rtg format -o $@ $<

.PHONY: all

.DEFAULT: all

.PRECIOUS:

.SECONDARY:

all: $(OUTROCS)

$(RTGROCS): %_$(TAG)/snp_roc.tsv.gz: ../%.vcf.gz $(VCFIN) $(BEDIN)
↪  $(REFIN).sdf
  rm -rf $(dir $@) && rtg vcfeval -b $(word 2, $^) -c $< -e $(word 3,
  ↪  $^) -t $(word 4, $^) --output-mode=roc-only
  ↪  --vcf-score-field=$(TAG) -T 4 -o $(dir $@)

$(OUTROCS): %_$(TAG).snp_roc.tsv.gz: %_$(TAG)/snp_roc.tsv.gz
  cp $< $@
```

G.2 Code to Call Variants With Simulated Data

```
# FILE:sim_variants/Makefile


SHELL = bash
REF = ../sampled_contigs.fa.gz
REFDICT = $(basename $(basename $(REF))).dict
TRUESNPS = ../simug.refseq2simseq.SNP.vcf.gz
TARGETVCFS = ngm.vcf.gz ngm.recal.vcf.gz kbbq-ngm.recal.vcf.gz
↪  initial-calls.recal.vcf.gz
RTGTARGETS = $(addsuffix .d/calls.vcf.gz,$(TARGETVCFS))
SUMMARYTARGETS = $(addsuffix .summary.txt,$(TARGETVCFS))
QUALROCS = $(addsuffix _QUAL_ROC/snp_roc.tsv.gz,$(TARGETVCFS))
```

```
GQROCS = $(addsuffix _GQ_ROC/snp_roc.tsv.gz,$(TARGETVCFS))
QUALROCTARGETS = $(addsuffix _QUAL.snp_roc.tsv.gz,$(TARGETVCFS))
GQROCTARGETS = $(addsuffix _GQ.snp_roc.tsv.gz,$(TARGETVCFS))
HAPDIPJS = $(HOME)/bin/hapdip/hapdip.js

.PRECIOUS:

.SECONDARY:

.PHONY: variants summaries rocs

.DEFAULT: summaries

#index helpers
%.fa.gz.fai: %.fa.gz
  samtools faidx $<

%.bam.bai: %.bam
  samtools index $<

%.vcf.gz: %.vcf
  bgzip -c $< > $@

%.vcf.gz.tbi: %.vcf.gz
  bcftools index -t $<

%.dict: %.fa
  gatk CreateSequenceDictionary -R $<

%.dict: %.fa.gz
  gatk CreateSequenceDictionary -R $<

%.fa.gz.sdf: %.fa.gz
  rtg format -o $@ $<


#theta 0.025

#get initial calls with HC and use them to recalibrate
$(TARGETVCFS): %.vcf.gz : ../%.bam $(REF) ../%.bam.bai $(REFDICT)
  gatk HaplotypeCaller -R $(word 2, $^) -I $<
   ↪  --native-pair-hmm-threads 8 -O $@ --heterozygosity 0.025

variants: $(TARGETVCFS)

sim_sample_snps.vcf.gz: $(TRUESNPS)
  bcftools view $< | \
```

```
  sed '/^#[^#]/s/$$/\tFORMAT\tsim/' | \
  sed '/^#/!s/$$/\tGT\t0|1/' | \
  bgzip -c > $@

#evaluate with rtg
$(RTGTARGETS): %.vcf.gz.d/calls.vcf.gz: %.vcf.gz
↪  sim_sample_snps.vcf.gz $(REF).sdf sim_sample_snps.vcf.gz.tbi
  rm -rf $(dir $@) && rtg vcfeval -b $(word 2, $^) -c $< -o $(dir
  ↪  $@) -t $(word 3, $^) --output-mode=annotate -T 8

#we don't have any non-confident regions; use the whole genome
conf.bed: $(REF).fai
  cat $< | awk -v OFS='\t' '{print $$1, 0, $$2}' > $@

%.d/hapdip_summary.txt: %.d/calls.vcf.gz conf.bed
  k8 $(HAPDIPJS) rtgeval $(word 2, $^) $(dir $@) > $@

$(SUMMARYTARGETS): %.summary.txt: %.d/hapdip_summary.txt
  cp $< $@

summaries: $(SUMMARYTARGETS)

#QUALROCS = $(addsuffix _QUAL_ROC/snp_roc.tsv.gz,$(TARGETVCFS))
$(QUALROCS): %.vcf.gz_QUAL_ROC/snp_roc.tsv.gz: %.vcf.gz
↪  sim_sample_snps.vcf.gz $(REF).sdf sim_sample_snps.vcf.gz.tbi
  rm -rf $(dir $@) && rtg vcfeval -b $(word 2, $^) -c $< -t $(word 3,
  ↪  $^) --output-mode=roc-only --vcf-score-field=QUAL -T 4 -o $(dir
  ↪  $@)

$(QUALROCTARGETS): %.vcf.gz_QUAL.snp_roc.tsv.gz:
↪  %.vcf.gz_QUAL_ROC/snp_roc.tsv.gz
  cp $< $@

$(GQROCS): %.vcf.gz_GQ_ROC/snp_roc.tsv.gz: %.vcf.gz
↪  sim_sample_snps.vcf.gz $(REF).sdf sim_sample_snps.vcf.gz.tbi
  rm -rf $(dir $@) && rtg vcfeval -b $(word 2, $^) -c $< -t $(word 3,
  ↪  $^) --output-mode=roc-only --vcf-score-field=GQ -T 4 -o $(dir
  ↪  $@)

$(GQROCTARGETS): %.vcf.gz_GQ.snp_roc.tsv.gz:
↪  %.vcf.gz_GQ_ROC/snp_roc.tsv.gz
  cp $< $@

rocs: $(QUALROCTARGETS) $(GQROCTARGETS)
```

```
# FILE:euc_variants/raw/Makefile


VARIANTSDIR = ../gatk4/
REF = ../e_mel_3/e_mel_3.fa
REFDICT = $(basename $(REF)).dict
RAWALIGNMENT = ../alignment.dedup.bam
REPEATBED = ../liftover/e_mel_3_repeatmask.bed
RAWVARIANTS = var-calls.vcf.gz
SCAFFOLDS = $(addprefix scaffold_,$(shell seq 1 11))
SCAFFOLDEDVARIANTS = $(addsuffix .var-calls.vcf.gz, $(SCAFFOLDS))

SCRIPTDIR = ~/bin/zypy/scripts
FLTWITHREP = $(SCRIPTDIR)/filt_with_replicates.pl
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
VCFTOFA = $(SCRIPTDIR)/vcf2fa.sh
PLOTTREE = $(SCRIPTDIR)/plot_tree.R

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz
↪  03-excesshet.vcf.gz \
        04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
        07-replicate.vcf.gz 08-variable.vcf.gz

all: $(ALL) num_variants.txt

.SHELL: bash

.PHONY: all

%.vcf.gz: %.vcf
  bgzip -c $< >$@

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

%.bam.bai: %.bam
  samtools index $<

.SECONDARY:

.PRECIOUS:

$(SCAFFOLDEDVARIANTS): %.var-calls.vcf.gz: $(RAWALIGNMENT) $(REF)
↪  $(RAWALIGNMENT).bai $(REFDICT)
```

```
  ~/.local/bin/gatk HaplotypeCaller -I $< -R $(word 2, $^)
   ↪  --native-pair-hmm-threads 8 --heterozygosity 0.025 -O $@
   ↪  --QUIET true -L $*

$(RAWVARIANTS): $(REFDICT) $(SCAFFOLDEDVARIANTS)
  picard SortVcf O=$@ SEQUENCE_DICTIONARY=$< $(addprefix
   ↪  I=,$(wordlist 2,$(words $^),$^))

01-first-11-scaffolds.vcf.gz: $(RAWVARIANTS) $(RAWVARIANTS).csi
  bcftools view -r
   ↪  scaffold_1,scaffold_2,scaffold_3,scaffold_4,scaffold_5,scaffold
   ↪  _6,scaffold_7,scaffold_8,scaffold_9,scaffold_10,scaffold_11 -Oz
   ↪  -o $@ $<

02-depth.vcf.gz: 01-first-11-scaffolds.vcf.gz
 ↪  01-first-11-scaffolds.vcf.gz.csi
  bcftools filter -i 'INFO/DP <= 500' -O z -o $@ $<

03-excesshet.vcf.gz: 02-depth.vcf.gz 02-depth.vcf.gz.csi
  bcftools filter -i 'ExcessHet <= 40' -O z -o $@ $<

#testing shows -g is applied last
04-near-indel.vcf.gz: 03-excesshet.vcf.gz 03-excesshet.vcf.gz.csi
  bcftools filter -g 50 -O z -o $@ $<

05-biallelic.vcf.gz: 04-near-indel.vcf.gz 04-near-indel.vcf.gz.csi
  bcftools view -m2 -M2 -v snps -O z -o $@ $<

06-repeat.vcf.gz: 05-biallelic.vcf.gz $(REPEATBED)
 ↪  05-biallelic.vcf.gz.csi
  vcftools --gzvcf $< --exclude-bed $(word 2, $^)
   ↪  --remove-filtered-all --recode --recode-INFO-all --stdout |
   ↪  bgzip -c > $@

07-replicate.vcf.gz: 06-repeat.vcf.gz 06-repeat.vcf.gz.csi
  zcat $< | $(FLTWITHREP) -s -g 3 | bgzip -c > $@

08-variable.vcf.gz: 07-replicate.vcf.gz
  zcat $< | $(DIPLOIDIFY) -t vcf -v | bgzip -c > $@

num_variants.txt: $(ALL)
  parallel --tag -I{} 'bcftools view -H {} | wc -l' ::: $(ALL) > $@



# FILE:euc_variants/raw/fnr/Makefile
```

```
#VARIANTSDIR = ../../2018-07-24_callability/
#REF = ../../e_mel_3/e_mel_3.fa
#ALIGNMENT = $(VARIANTSDIR)/alignment.bam
#REPEATBED = ../../liftover/e_mel_3_repeatmask.bed
#RAWVARIANTS = $(VARIANTSDIR)/gatk/var-calls.vcf.gz
#INREPEATS = $(VARIANTSDIR)/gatk/inrepeats.txt
DNGCALLS = ../../dng/filter/goodsites.vcf

SCRIPTDIR = ~/bin/zypy/scripts
FLTWITHREP = $(SCRIPTDIR)/filt_with_replicates.pl
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
VCFTOFA = $(SCRIPTDIR)/vcf2fa.sh
PLOTTREE = $(SCRIPTDIR)/plot_tree.R
CALLABILITYRES = $(SCRIPTDIR)/spiked_mutation_results.py

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz
↪  03-excesshet.vcf.gz \
  04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
  07-replicate.vcf.gz 08-variable.vcf.gz

RESULTS = $(addsuffix .result, $(basename $(basename $(ALL))))
FNRESULTS = $(addsuffix .fn.result, $(basename $(basename $(ALL))))
TPRESULTS = $(addsuffix .tp.result, $(basename $(basename $(ALL))))

results: fn.txt tp.txt

.PHONY: results

.SECONDARY:

%.vcf.gz: %.vcf
  bgzip -c $< >$@

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

#Calls not in the callset but in DNGCALLS are FN
%.fn.result: ../%.vcf.gz
  bcftools view -H -T ^$< $(DNGCALLS) | wc -l > $@

#Calls that match DNGCALLS are TPs
%.tp.result: ../%.vcf.gz
  bcftools view -H -T $(DNGCALLS) $< | wc -l > $@

fn.txt: $(FNRESULTS)
  parallel -I{} --tag 'cat {}' ::: $^ > $@
```

```
tp.txt: $(TPRESULTS)
  parallel -I{} --tag 'cat {}' ::: $^ > $@

# FILE:euc_variants/raw/fdr/Makefile


GATKVARIANTS = ../01-first-11-scaffolds.vcf.gz
CALLEDVARIANTS = ../08-variable.vcf.gz
JOBNUMS=$(shell seq 1 100)
VCFNUMS = $(addsuffix .vcf.gz, $(JOBNUMS))
JOBCALLS=$(addprefix %/,$(VCFNUMS))

SCRIPTDIR = ~/bin/zypy/scripts
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
FPRFILT = $(SCRIPTDIR)/gatkfpr.py
SHUF_LABELS = $(SCRIPTDIR)/label_permutation.R
SHELL=bash
DNGCALLS = ../../dng/filter/goodsites.vcf
############################################################

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz
↪  03-excesshet.vcf.gz \
        04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
        07-replicate.vcf.gz 08-variable.vcf.gz

RESULTS = $(addsuffix .result, $(basename $(basename $(ALL))))
RESFOLDERS = $(addsuffix /, $(basename $(basename $(ALL))))
ALLVCFS = $(foreach folder, $(RESFOLDERS), $(addprefix $(folder),
↪  $(VCFNUMS)))
ALLTREES = $(addprefix trees/, $(addsuffix .ped,$(JOBNUMS)))

default: all

all: $(RESULTS) results.txt

.PHONY: all default

.DELETE_ON_ERROR:

.INTERMEDIATE:

.SECONDARY:

############################################################
# Helper Rules
```

```
%.vcf.gz.csi: %.vcf.gz
	bcftools index $<

#

trees/:
	mkdir -p $@

trees/%.ped: trees/
	echo "##PEDNG v1.0" > $@; echo -e "M\t.\t.\t0" | paste - <(
	↪ $(SHUF_LABELS) ) >> $@

$(RESFOLDERS):
	mkdir -p $@

vpath %.ped trees/

$(ALLVCFS): | $(RESFOLDERS)
	$(FPRFILT) -n 3 -v ../$(subst /,,$(dir $@)).vcf.gz -p
	↪ trees/$(notdir $(basename $(basename $@))).ped | \
	$(DIPLOIDIFY) -t vcf -v | bgzip > $@

$(RESULTS): %.result : $(DNGCALLS) $(JOBCALLS)
	rm -f $@; for f in $(wordlist 2,$(words $^),$^) ; do bcftools view
	↪ -H -T ^$< $$f | wc -l >> $@; done

#FP, FP/P (FDR)
results.txt: $(RESULTS)
	parallel -I{} --tag 'cat {} | awk "{x+=\$$1}END{print x,x/NR}"' :::
	↪ $^ > $@


# FILE:euc_variants/kbbq/Makefile


VARIANTSDIR = ../gatk4/
REF = ../e_mel_3/e_mel_3.fa
REFDICT = $(basename $(REF)).dict
RAWALIGNMENT = ../alignment.dedup.bam
KBBQALIGNMENT = recalibrated.bam
REPEATBED = ../liftover/e_mel_3_repeatmask.bed
RAWVARIANTS = var-calls.vcf.gz
SCAFFOLDS = $(addprefix scaffold_,$(shell seq 1 11))
SCAFFOLDEDVARIANTS = $(addsuffix .var-calls.vcf.gz, $(SCAFFOLDS))

SCRIPTDIR = ~/bin/zypy/scripts
FLTWITHREP = $(SCRIPTDIR)/filt_with_replicates.pl
```

```
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
VCFTOFA = $(SCRIPTDIR)/vcf2fa.sh
PLOTTREE = $(SCRIPTDIR)/plot_tree.R

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz
↪  03-excesshet.vcf.gz \
        04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
        07-replicate.vcf.gz 08-variable.vcf.gz

all: $(ALL) num_variants.txt

.SHELL: bash

.PHONY: all

%.vcf.gz: %.vcf
  bgzip -c $< >$@

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

%.bam.bai: %.bam
  samtools index $<

.SECONDARY:

.PRECIOUS:

$(KBBQALIGNMENT): $(RAWALIGNMENT)
  ~/bin/cbbq/build/cbbq -t12 -k32 -g 605951511 -c 240 $< > $@ 2>
  ↪  cbbq.log

$(SCAFFOLDEDVARIANTS): %.var-calls.vcf.gz: $(KBBQALIGNMENT) $(REF)
↪  $(KBBQALIGNMENT).bai $(REFDICT)
  ~/.local/bin/gatk HaplotypeCaller -I $< -R $(word 2, $^)
  ↪  --heterozygosity 0.025 -O $@ --QUIET -L $*

$(RAWVARIANTS): $(REFDICT) $(SCAFFOLDEDVARIANTS)
  picard SortVcf O=$@ SEQUENCE_DICTIONARY=$< $(addprefix
  ↪  I=,$(wordlist 2,$(words $^),$^))

01-first-11-scaffolds.vcf.gz: $(RAWVARIANTS) $(RAWVARIANTS).csi
  bcftools view -r
  ↪  scaffold_1,scaffold_2,scaffold_3,scaffold_4,scaffold_5,scaffold
  ↪  _6,scaffold_7,scaffold_8,scaffold_9,scaffold_10,scaffold_11 -Oz
  ↪  -o $@ $<
```

```
02-depth.vcf.gz: 01-first-11-scaffolds.vcf.gz
↪  01-first-11-scaffolds.vcf.gz.csi
  bcftools filter -i 'INFO/DP <= 500' -O z -o $@ $<

03-excesshet.vcf.gz: 02-depth.vcf.gz 02-depth.vcf.gz.csi
  bcftools filter -i 'ExcessHet <= 40' -O z -o $@ $<

#testing shows -g is applied last
04-near-indel.vcf.gz: 03-excesshet.vcf.gz 03-excesshet.vcf.gz.csi
  bcftools filter -g 50 -O z -o $@ $<

05-biallelic.vcf.gz: 04-near-indel.vcf.gz 04-near-indel.vcf.gz.csi
  bcftools view -m2 -M2 -v snps -O z -o $@ $<

06-repeat.vcf.gz: 05-biallelic.vcf.gz $(REPEATBED)
↪  05-biallelic.vcf.gz.csi
  vcftools --gzvcf $< --exclude-bed $(word 2, $^)
  ↪  --remove-filtered-all --recode --recode-INFO-all --stdout |
  ↪  bgzip -c > $@

07-replicate.vcf.gz: 06-repeat.vcf.gz 06-repeat.vcf.gz.csi
  zcat $< | $(FLTWITHREP) -s -g 3 | bgzip -c > $@

08-variable.vcf.gz: 07-replicate.vcf.gz
  zcat $< | $(DIPLOIDIFY) -t vcf -v | bgzip -c > $@

num_variants.txt: $(ALL)
  parallel --tag -I{} 'bcftools view -H {} | wc -l' ::: $(ALL) > $@



# FILE:euc_variants/kbbq/fnr/Makefile


#VARIANTSDIR = ../../2018-07-24_callability/
#REF = ../../e_mel_3/e_mel_3.fa
#ALIGNMENT = $(VARIANTSDIR)/alignment.bam
#REPEATBED = ../../liftover/e_mel_3_repeatmask.bed
#RAWVARIANTS = $(VARIANTSDIR)/gatk/var-calls.vcf.gz
#INREPEATS = $(VARIANTSDIR)/gatk/inrepeats.txt
DNGCALLS = ../../dng/filter/goodsites.vcf

SCRIPTDIR = ~/bin/zypy/scripts
FLTWITHREP = $(SCRIPTDIR)/filt_with_replicates.pl
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
VCFTOFA = $(SCRIPTDIR)/vcf2fa.sh
PLOTTREE = $(SCRIPTDIR)/plot_tree.R
```

```
CALLABILITYRES = $(SCRIPTDIR)/spiked_mutation_results.py

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz
↪  03-excesshet.vcf.gz \
  04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
  07-replicate.vcf.gz 08-variable.vcf.gz

RESULTS = $(addsuffix .result, $(basename $(basename $(ALL))))
FNRESULTS = $(addsuffix .fn.result, $(basename $(basename $(ALL))))
TPRESULTS = $(addsuffix .tp.result, $(basename $(basename $(ALL))))

results: fn.txt tp.txt

.PHONY: results

.SECONDARY:

%.vcf.gz: %.vcf
  bgzip -c $< >$@

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

#Calls not in the callset but in DNGCALLS are FN
%.fn.result: ../%.vcf.gz
  bcftools view -H -T ^$< $(DNGCALLS) | wc -l > $@

#Calls that match DNGCALLS are TPs
%.tp.result: ../%.vcf.gz
  bcftools view -H -T $(DNGCALLS) $< | wc -l > $@

fn.txt: $(FNRESULTS)
  parallel -I{} --tag 'cat {}' ::: $^ > $@

tp.txt: $(TPRESULTS)
  parallel -I{} --tag 'cat {}' ::: $^ > $@

# FILE:euc_variants/kbbq/fdr/Makefile


GATKVARIANTS = ../01-first-11-scaffolds.vcf.gz
CALLEDVARIANTS = ../08-variable.vcf.gz
JOBNUMS=$(shell seq 1 100)
VCFNUMS = $(addsuffix .vcf.gz, $(JOBNUMS))
JOBCALLS=$(addprefix %/,$(VCFNUMS))

SCRIPTDIR = ~/bin/zypy/scripts
```

```
DIPLOIDIFY = $(SCRIPTDIR)/diploidify.py
FPRFILT = $(SCRIPTDIR)/gatkfpr.py
SHUF_LABELS = $(SCRIPTDIR)/label_permutation.R
SHELL=bash
DNGCALLS = ../../dng/filter/goodsites.vcf
############################################################

ALL = 01-first-11-scaffolds.vcf.gz 02-depth.vcf.gz
↪  03-excesshet.vcf.gz \
        04-near-indel.vcf.gz 05-biallelic.vcf.gz 06-repeat.vcf.gz \
        07-replicate.vcf.gz 08-variable.vcf.gz

RESULTS = $(addsuffix .result, $(basename $(basename $(ALL))))
RESFOLDERS = $(addsuffix /, $(basename $(basename $(ALL))))
ALLVCFS = $(foreach folder, $(RESFOLDERS), $(addprefix $(folder),
↪  $(VCFNUMS)))
ALLTREES = $(addprefix trees/, $(addsuffix .ped,$(JOBNUMS)))

default: all

all: $(RESULTS) results.txt

.PHONY: all default

.DELETE_ON_ERROR:

.INTERMEDIATE:

.SECONDARY:

############################################################
# Helper Rules

%.vcf.gz.csi: %.vcf.gz
  bcftools index $<

#

trees/:
  mkdir -p $@

trees/%.ped: trees/
  echo "##PEDNG v1.0" > $@; echo -e "M\t.\t.\t0" | paste - <(
  ↪  $(SHUF_LABELS) ) >> $@

$(RESFOLDERS):
  mkdir -p $@
```

```
vpath %.ped trees/

$(ALLVCFS): | $(RESFOLDERS)
  $(FPRFILT) -n 3 -v ../$(subst /,,$(dir $@)).vcf.gz -p
   ↪  trees/$(notdir $(basename $(basename $@))).ped | \
  $(DIPLOIDIFY) -t vcf -v | bgzip > $@

$(RESULTS): %.result : $(DNGCALLS) $(JOBCALLS)
  rm -f $@; for f in $(wordlist 2,$(words $^),$^) ; do bcftools view
   ↪  -H -T ^$< $$f | wc -l >> $@; done

#FP, FP/P (FDR)
results.txt: $(RESULTS)
  parallel -I{} --tag 'cat {} | awk "{x+=\$$1}END{print x,x/NR}"' :::
   ↪  $^ > $@
```

G.4 ROC Plotting Code

```
#!/usr/bin/env Rscript

#ROCs on ROCs on ROCs!!

library(tidyverse)
library(colorblindr)

# --- Import Data ---
#import rtg-eval roc files.

get_data_dir <- function(path){
  return(paste0("../data/roc/", path))
}

#return file name of the roc tsv
#given the prefix and type. For example, 0_0 QUAL
#returns "../data/roc/0_0_QUAL.snp_roc.tsv.gz"
get_roc_path <- function(prefix, type){
  return(get_data_dir(paste0(prefix, "_", type, ".snp_roc.tsv.gz")))
}

# --- Get Data Frame ---
import_rtg_rocs <- function(fnr, fpr, type){
  prefixes <- unlist(map(fnr, paste0, '_', fpr))
  prefixes <- c(prefixes, "raw","cbbq")
  roc_files <- get_roc_path(prefixes, type)
  dfs <- map(roc_files, read_tsv, comment = "#",
             col_names = c(
```

```
                type,"TPb","FP","TPc", "FN", "precision",
                 ↪  "sensitivity", "f"))
  prefixes[(length(prefixes)-1):length(prefixes)] =
   ↪  c("raw_raw","kbbq_kbbq")
  names(dfs) <- prefixes
  notempty <- map_lgl(dfs, ~dim(.)[1] != 0)
  dfs <- dfs[notempty]
  bind_rows(dfs, .id = 'FNR_FPR') %>%
  separate(FNR_FPR, into = c("FalseNegativeRate","FalsePositiveRate"),
           sep = "_", convert = TRUE) %>%
  mutate(across(c(FalseNegativeRate,FalsePositiveRate),
                ~factor(., levels =
                 ↪  c("raw","kbbq",0,20,40,60,80,100)))) %>%
  group_by(FalseNegativeRate,FalsePositiveRate) %>%
  filter(FalsePositiveRate != 100)
}

rate_labeller <- labeller(
  # .cols = function(str){str_replace(str,"^","FNR: 0")},
  # .rows = function(str){str_replace(str,'^(\\d+)$','FPR:\\1')}
  .rows = function(str){str_replace(str,'^0$','FPR:0')}
)

# --- Make plots ---

r_factor <- function(v){factor({{v}}, levels =
 ↪  c("kbbq","raw",0,20,40,60,80,100))}

plot_roc <- function(df){
  just_empirical <- df %>%
    ungroup() %>%
    filter(FalseNegativeRate == "kbbq" | FalseNegativeRate == "raw")
     ↪  %>%
    select(FalseNegativeRate, FPR, sensitivity)

  df %>%
    unite("FNR_FPR", c(FalseNegativeRate,FalsePositiveRate), remove =
     ↪  FALSE) %>%
    mutate(FalsePositiveRate = factor(FalsePositiveRate)) %>%
    mutate(FalsePositiveRate = fct_collapse(FalsePositiveRate,
      empirical = c('kbbq','raw')
    )) %>%
    ggplot(aes(FPR,sensitivity)) +
    # facet_grid(rows = vars(FalsePositiveRate), as.table = FALSE,
               # margins = TRUE, labeller = rate_labeller) +
    facet_wrap(facets = vars(FalsePositiveRate), as.table = TRUE,
               labeller = rate_labeller) +
```

```
    geom_line(aes(color = factor(FalseNegativeRate, levels =
    ↪  c("raw","kbbq",0,20,40,60,80,100)), group = FNR_FPR)) +
    # geom_line(aes(color = FalseNegativeRate, group = FNR_FPR)) +
    scale_color_viridis_d('Variable\nSites\nFNR', option = 'viridis')
    ↪  +
    xlab("False Positive Rate") +
    ylab("True Positive Rate") +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    coord_fixed(ratio = max(df$FPR)) +
    geom_line(data = just_empirical, aes(color = FalseNegativeRate,
    ↪  group = FalseNegativeRate)) #add empirical on top of every
    ↪  plot
}

plot_single_roc <- function(df){
  df %>%
    unite("FNR_FPR", c(FalseNegativeRate,FalsePositiveRate), remove =
    ↪  FALSE) %>%
    mutate(FalsePositiveRate = factor(FalsePositiveRate)) %>%
    mutate(FalsePositiveRate = fct_collapse(FalsePositiveRate,
                                            empirical =
                                            ↪  c('kbbq','raw')
    )) %>%
    ggplot(aes(FPR,sensitivity)) +
    geom_line(aes(color = factor(FalseNegativeRate, levels =
    ↪  c("raw","kbbq",0,20,40,60,80,100)), group = FNR_FPR)) +
    scale_color_viridis_d('Variable\nSites\nFNR', option = 'viridis')
    ↪  +
    # scale_color_OkabeIto(name = 'Variable\n Sites\nFNR') +
    xlab("False Positive Rate") +
    ylab("True Positive Rate") +
    theme(strip.text.y = element_text(angle = 0)) +
    coord_fixed(ratio = max(df$FPR))
}

plot_sen_prec <- function(df){
  df %>% ggplot(aes(sensitivity,precision)) +
    facet_grid(rows = vars(FalsePositiveRate), as.table = FALSE) +
    geom_line(aes(color = factor(FalseNegativeRate))) +
    scale_color_viridis_d(option = 'viridis')
}

plot_roc_maxf <- function(df){
  df %>%
    summarize(maxf = max(f)) %>%
    ggplot(aes(FalseNegativeRate,FalsePositiveRate)) +
    geom_raster(aes(fill = maxf)) +
```

```
    scale_fill_viridis_c() +
    coord_fixed(ratio = 1)
}

#the value of the statistic where F is maximized
plot_roc_argmaxf <- function(df, var){
  df %>%
    slice_max(order_by = f, with_ties = FALSE) %>%
    ggplot(aes(FalseNegativeRate,FalsePositiveRate)) +
    geom_raster(aes(fill = {{var}})) +
    scale_fill_viridis_c() +
    xlab("Variable Sites FNR") +
    ylab("Variable Sites FPR") +
    coord_fixed(ratio = 1)
}

plot_roc_avgf <- function(df){
  df %>%
    summarize(avgf = median(f)) %>%
    ggplot(aes(FalseNegativeRate,FalsePositiveRate)) +
    geom_raster(aes(fill = avgf)) +
    scale_fill_viridis_c() +
    xlab("Variable Sites FNR") +
    ylab("Variable Sites FPR") +
    coord_fixed(ratio = 1)
}

#emulate filtering on an (exclusive) maximum and minimum value.
# The ROC lists the cumulative TP, FP, FN in DESCENDING ORDER.
#
# Thus, above the maximum:
# FPs and TPs are removed.
# FNs need to be added.
#
# Below the minimum:
# FPs and TPs are removed.
# FNs are calculated correctly.
#
# Precision, sensitivity, and F are then recalculated.
emulate_filter <- function(df, var, min, max){
  ceil <- df %>%
    filter({{var}} > max, .preserve = TRUE) %>%
    slice_min(order_by = {{var}}, with_ties = FALSE) %>%
    select(FalseNegativeRate,FalsePositiveRate,TPb:FN) %>%
    rename(TPbMAX = TPb, FPMAX = FP, TPcMAX = TPc) %>%
    mutate(FNMAX = max(FN)) %>%
    select(-FN)
```

```
  df %>% select(FalseNegativeRate:FN) %>%
    filter({{var}} <= max, .preserve = TRUE) %>%
    left_join(ceil) %>%
    mutate(FP = FP - FPMAX,
           TPb = TPb - TPbMAX,
           TPc = TPc - TPcMAX) %>%
    filter({{var}} >= min, .preserve = TRUE) %>%
    mutate(FN = if_else(FN == max(FN),FNMAX,FN)) %>%
    select(!TPbMAX:FNMAX) %>%
    mutate(sensitivity = TPb/(TPb + FN), precision = TPc/(TPc + FP))
    ↪  %>%
    mutate(f = 2 * precision * sensitivity / (precision +
    ↪  sensitivity))
}


# --- Main ---
fpr <- seq(0,100,20)

#zcat ansites.bed.gz | awk '{x += ($3-$2)}END{print x}'
neg_sites <- 213684509

#set theme
theme_set(theme_minimal(base_size = 22, base_family = 'Times') +
            theme(plot.margin = margin(0,0,0,0))
          )

# I plot the ROC, argmaxf, and avgf

#qualroc
qualroc <- import_rtg_rocs(fpr,fpr,"QUAL") %>% mutate(FPR =
↪  FP/neg_sites)

pdf("../figures/qualroc.pdf", width = 9, height = 7)
qualroc %>% plot_roc() + #xlim(0 , 2e-5) +
  ggtitle("QUAL ROC")
dev.off()

pdf("../figures/0fpr_roc.pdf", width = 9, height = 7)
qualroc %>% filter(FalsePositiveRate == 0 | FalsePositiveRate ==
↪  "kbbq" | FalsePositiveRate == "raw") %>% plot_single_roc() +
  ggtitle("QUAL ROC")
dev.off()

    pdf("../figures/qualmaxf.pdf", width = 9, height = 7)
qualroc %>% plot_roc_argmaxf(QUAL) +
```

```
  ggtitle("QUAL with Best F-statistic")
dev.off()
# qualroc %>% plot_roc_avgf() +
#   ggtitle("Better Calibration Increases Average QUAL F-statistic")

#dproc
# I decided not to use this since it's not well-defined as a score:
#   larger numbers != better. Depth can probably be transformed to
#   a score assuming a Poisson distribution, but that is outside of
#   the scope of this work.

# dproc <- import_rtg_rocs(fpr,fpr,"DP")
# dproc %>% plot_roc()
# dproc %>% plot_roc_maxf()

#gqroc
gqroc <- import_rtg_rocs(fpr,fpr,"GQ") %>% mutate(FPR = FP/neg_sites)
pdf("../figures/gqroc.pdf", width = 9, height = 7)
gqroc %>% plot_roc() +
  ggtitle("GQ ROC")
dev.off()
# gqroc %>% plot_roc_argmaxf(GQ) +
#   ggtitle("")
# gqroc %>% plot_roc_avgf()

# Better calibration reduces the power of GQ, but increases the
↪  power of QUAL
bcftools_data_dir <- function(path){
  return(paste0("../data/bcftools_roc/", path))
}

bcftools_roc_path <- function(prefix, type){
  return(bcftools_data_dir(paste0(prefix, "_", type,
  ↪  ".snp_roc.tsv.gz")))
}

import_bcftools_rocs <- function(fnr, fpr, type){
  prefixes <- unlist(map(fnr, paste0, '_', fpr))
  prefixes <- c(prefixes, "raw","cbbq")
  roc_files <- bcftools_roc_path(prefixes, type)
  dfs <- map(roc_files, read_tsv, comment = "#",
             col_names = c(
               type,"TPb","FP","TPc", "FN", "precision",
               ↪  "sensitivity", "f"))
  prefixes[(length(prefixes)-1):length(prefixes)] =
  ↪  c("raw_raw","kbbq_kbbq")
  names(dfs) <- prefixes
```

```
  notempty <- map_lgl(dfs, ~dim(.)[1] != 0)
  dfs <- dfs[notempty]
  bind_rows(dfs, .id = 'FNR_FPR') %>%
    separate(FNR_FPR, into =
    ↪  c("FalseNegativeRate","FalsePositiveRate"),
             sep = "_", convert = TRUE) %>%
    mutate(across(c(FalseNegativeRate,FalsePositiveRate),
                  ~factor(., levels =
                   ↪  c("raw","kbbq",0,20,40,60,80,100)))) %>%
    group_by(FalseNegativeRate,FalsePositiveRate) %>%
    filter(FalsePositiveRate != 100)
}

# ------- Simulated Data ROC Plots ----------
sim_neg_sites <- 5013478 - 125000

import_sim_rocs <- function(type){
  datanames <-
  ↪  c("ngm","ngm.recal","initial-calls.recal","kbbq-ngm.recal")
  nicenames <- c("Raw","GATK","Initial-calls","KBBQ")
  roc_files <- paste0("../data/sims/",datanames,".vcf.gz_",type,".snp
  ↪  _roc.tsv.gz")
  dfs <- map(roc_files, read_tsv, comment = "#",
             col_names = c(
               type,"TPb","FP","TPc", "FN", "precision",
                ↪  "sensitivity", "f"))
  names(dfs) <- nicenames
  bind_rows(dfs, .id = 'CalibrationMethod') %>%
    group_by(CalibrationMethod) %>%
    mutate(CalibrationMethod = factor(CalibrationMethod, levels =
    ↪  nicenames)) %>%
    mutate(FPR = FP/sim_neg_sites)
}


plot_sim_roc <- function(df){
  df %>%
    ggplot(aes(FPR,sensitivity)) +
    # facet_grid(rows = vars(FalsePositiveRate), as.table = FALSE,
    # margins = TRUE, labeller = rate_labeller) +
    # facet_wrap(facets = vars(CalibrationMethod), as.table = TRUE)
    ↪  +
    # geom_line(aes(color = CalibrationMethod)) +
    geom_line(aes(color = CalibrationMethod)) +
    xlab("False Positive Rate") +
    ylab("True Positive Rate") +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
```

```
    coord_fixed(ratio = max(df$FPR)) +
    scale_color_OkabeIto(name = 'Calibration Method', use_black = T,
    ↪  drop = FALSE) +
    ggtitle("QUAL ROC of Calls on Simulated Reads")
}

sim_qual_df <- import_sim_rocs("QUAL")
pdf("../figures/sims_qualroc.pdf", width = 9, height = 7)
plot_sim_roc(sim_qual_df)
dev.off()
#GQ roc doesn't show any differences.

#sim_gq_df <- import_sim_rocs("GQ")
#plot_sim_roc(sim_gq_df)
```

G.5 Summary Plotting Code

```
#!/usr/bin/env Rscript
library(tidyverse)
library(colorblindr)

# --- Functions to get file paths ---

#return the data directory appended to path
get_data_dir <- function(path){
  return(paste0("../data/", path))
}

#return file name of the roc tsv
#given the prefix. For example, 0_0
#returns "../data/0_0.snp_roc.tsv.gz"
get_summary_path <- function(prefix){
  return(get_data_dir(paste0(prefix, ".summary.txt")))
}

# --- Import Dataframe ---
import_fnrfpr_tsvs <- function(fnr, fpr){
  prefixes <- unlist(map(fnr, paste0, '_', fpr))
  prefixes <- c(prefixes, "raw","cbbq")
  summary_files <- get_summary_path(prefixes)
  dfs <- map(summary_files, read_tsv,
             col_names = c("mode","type","operator","value"))
  prefixes[(length(prefixes)-1):length(prefixes)] =
  ↪  c("raw_raw","kbbq_kbbq")
  names(dfs) <- prefixes
  # notempty <- map_lgl(dfs, ~dim(.)[1] != 0)
  # dfs <- dfs[notempty]
  fnr_var <- sym('%FNR')
```

```
  fdr_var <- sym('%FDR')
  bind_rows(dfs, .id = 'FNR_FPR') %>%
    pivot_wider(names_from = operator, values_from = value) %>%
    separate(FNR_FPR, into =
     ↪  c("FalseNegativeRate","FalsePositiveRate"),
             sep = "_", convert = TRUE) %>%
    rename(FNR = all_of(fnr_var)) %>%
    rename(FDR = all_of(fdr_var)) %>%
    mutate(FDR = FDR/100) %>%
    mutate(FNR = FNR/100) %>%
    mutate(across(c(FalseNegativeRate,FalsePositiveRate),
          ~factor(., levels = c("raw","kbbq",0,20,40,60,80,100))))
}

import_bcftools_tsvs <- function(fnr, fpr){
  prefixes <- unlist(map(fnr, paste0, '_', fpr))
  prefixes <- c(prefixes, "raw","cbbq")
  summary_files <- get_summary_path(paste0("bcftools/",prefixes))
  dfs <- map(summary_files, read_tsv,
             col_names = c("mode","type","operator","value"))
  prefixes[(length(prefixes)-1):length(prefixes)] =
   ↪  c("raw_raw","kbbq_kbbq")
  names(dfs) <- prefixes
  # notempty <- map_lgl(dfs, ~dim(.)[1] != 0)
  # dfs <- dfs[notempty]
  fnr_var <- sym('%FNR')
  fdr_var <- sym('%FDR')
  bind_rows(dfs, .id = 'FNR_FPR') %>%
    pivot_wider(names_from = operator, values_from = value) %>%
    separate(FNR_FPR, into =
     ↪  c("FalseNegativeRate","FalsePositiveRate"),
             sep = "_", convert = TRUE) %>%
    rename(FNR = all_of(fnr_var)) %>%
    rename(FDR = all_of(fdr_var)) %>%
    mutate(FDR = FDR/100) %>%
    mutate(FNR = FNR/100) %>%
    mutate(across(c(FalseNegativeRate,FalsePositiveRate),
                  ~factor(., levels =
                   ↪  c("raw","kbbq",0,20,40,60,80,100))))
}

# --- Manipulate Dataframe ---
only_snps <- function(df){
  df %>%
    filter(mode == "rtgeval-gt") %>%
    filter(type == "SNP") %>%
    select(-type, -mode)
```

```
}

get_f_statistic <- function(df){
  df %>%
    group_by(FalseNegativeRate, FalsePositiveRate) %>%
    # mutate(precision = 1 - FDR) %>%
    # mutate(recall = 1 - FNR) %>%
    mutate(recall = TPc / (TPc + FN)) %>%
    mutate(precision = TPc / (TPc + FP)) %>%
    mutate(f = 2 * precision * recall / (precision + recall))
}

calc_f_and_filter <- function(df){
  df %>% only_snps() %>% filter(FalsePositiveRate != 100) %>%
   ↪  get_f_statistic()
}

# --- Main ---
fnr <- seq(0,100,20)
df <- import_fnrfpr_tsvs(fnr,fnr) %>%
  calc_f_and_filter()

#set theme
theme_set(theme_minimal(base_size = 22, base_family = 'Times') +
            theme(plot.margin = margin(0,0,0,0))
)

# --- Plotting ---

#Better calibrated data has more TPs
pdf("../figures/tp_heatmap.pdf", width = 9, height = 7)
df %>%
  ggplot(aes(FalseNegativeRate,FalsePositiveRate)) +
  geom_raster(aes(fill = TP)) +
  scale_fill_viridis_c("TP Calls") +
  scale_x_discrete("Variable Sites FNR") +
  scale_y_discrete("Variable Sites FPR") +
  # ggtitle("Better Calibration Increases\nTrue Positive Calls") +
  ggtitle("Unfiltered True Positive Calls")
dev.off()

#and more FPs
pdf("../figures/fp_heatmap.pdf", width = 9, height = 7)
df %>%
  ggplot(aes(FalseNegativeRate,FalsePositiveRate)) +
  geom_raster(aes(fill = FP)) +
  scale_fill_viridis_c("FP Calls") +
```

```
  scale_x_discrete("Variable Sites FNR") +
  scale_y_discrete("Variable Sites FPR") +
  ggtitle("Unfiltered False Positive Calls")
  # ggtitle("Better Calibration Increases\nFalse Positive Calls") +
dev.off()

#Shown together
pdf("../figures/tp_fp_plot.pdf", width = 9, height = 7)
df %>% ggplot(aes(TP, FP)) +
  geom_point(aes(color = factor(FalseNegativeRate))) +
  # coord_fixed(ratio = 1) +
  # geom_segment(
  #   aes(x = 267750, y = 7200, xend = 268050, yend = 7200 +
  ↪  (268050-267750))) +
  scale_color_viridis_d("Variable\nSites\nFNR", option = 'viridis') +
  ggtitle("Unfiltered Positive Calls") +
  # ggtitle("Better Calibration Produces\nMore Positive Calls") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
dev.off()

#This leads to an increase in sensitivity
pdf("../figures/sensitivity.pdf", width = 9, height = 7)
df %>%
  ggplot(aes(FalseNegativeRate, FalsePositiveRate)) +
  geom_raster(aes(fill = recall)) +
  scale_fill_viridis_c('Sensitivity', option = "viridis") +
  xlab("Variable Sites FNR") +
  ylab("Variable Sites FPR") +
  # ggtitle('Better Calibration Increases Sensitivity') +
  ggtitle('Unfiltered Sensitivity') +
  coord_fixed(ratio = 1)
dev.off()

#But a decrease in precision
pdf("../figures/precision.pdf", width = 9, height = 7)
df %>%
  ggplot(aes(FalseNegativeRate, FalsePositiveRate)) +
  geom_raster(aes(fill = precision)) +
  scale_fill_viridis_c('Precision', option = "viridis") +
  xlab("Variable Sites FNR") +
  ylab("Variable Sites FPR") +
  ggtitle('Unfiltered Precision') +
  coord_fixed(ratio = 1)
dev.off()

#Shown together:
pdf("../figures/sens_precision.pdf", width = 9, height = 7)
```

```
df %>% ggplot(aes(precision, recall)) +
  geom_point(aes(color = factor(FalseNegativeRate))) +
  # coord_fixed(ratio = 1) +
  scale_color_viridis_d("Variable\nSites\nFNR", option = 'viridis') +
  ggtitle("Unfiltered Sensitivity and Precision") +
  scale_x_continuous("Precision") +
  scale_y_continuous("Sensitivity")
dev.off()
# Taken together, this leads to a reduced F-statistic for the
↪  well-calibrated
# data.
# F line plot
pdf("../figures/f_plot.pdf", width = 9, height = 7)
df %>%
  ggplot(aes(FalsePositiveRate, f)) +
  geom_point(aes(color = FalseNegativeRate)) +
  geom_line(aes(color = FalseNegativeRate, group =
  ↪  FalseNegativeRate)) +
  scale_color_viridis_d("Variable\nSites\nFNR", option = "viridis") +
  ggtitle("Unfiltered F-statistic")
dev.off()
# F Heatmap
pdf("../figures/f_heatmap.pdf", width = 9, height = 7)
df %>%
  ggplot(aes(FalseNegativeRate, FalsePositiveRate)) +
  geom_raster(aes(fill = f)) +
  scale_fill_viridis_c('F', option = "viridis") +
  xlab("Variable Sites FNR") +
  ylab("Variable Sites FPR") +
  ggtitle('Unfiltered F-statistic') +
  coord_fixed(ratio = 1)
dev.off()
#Import filtered DF
# --- Import Dataframe ---
import_flt_tsvs <- function(fnr, fpr){
  prefixes <- unlist(map(fnr, paste0, '_', fpr))
  prefixes <- c(prefixes, "raw","cbbq")
  summary_files <- get_summary_path(paste0('flt/',prefixes,'_flt'))
  dfs <- map(summary_files, read_tsv,
             col_names = c("mode","type","operator","value"))
  prefixes[(length(prefixes)-1):length(prefixes)] =
  ↪  c("raw_raw","kbbq_kbbq")
  names(dfs) <- prefixes
  # notempty <- map_lgl(dfs, ~dim(.)[1] != 0)
  # dfs <- dfs[notempty]
  fnr_var <- sym('%FNR')
  fdr_var <- sym('%FDR')
```

```
  bind_rows(dfs, .id = 'FNR_FPR') %>%
    pivot_wider(names_from = operator, values_from = value) %>%
    separate(FNR_FPR, into =
     ↪  c("FalseNegativeRate","FalsePositiveRate"),
             sep = "_", convert = TRUE) %>%
    rename(FNR = all_of(fnr_var)) %>%
    rename(FDR = all_of(fdr_var)) %>%
    mutate(FDR = FDR/100) %>%
    mutate(FNR = FNR/100) %>%
    mutate(across(c(FalseNegativeRate,FalsePositiveRate),
                  ~factor(., levels =
                   ↪  c("raw","kbbq",0,20,40,60,80,100))))
}

import_bcftools_flt_tsvs <- function(fnr, fpr){
  prefixes <- unlist(map(fnr, paste0, '_', fpr))
  prefixes <- c(prefixes, "raw","cbbq")
  summary_files <-
   ↪  get_summary_path(paste0('bcftools_flt/',prefixes,'_flt'))
  dfs <- map(summary_files, read_tsv,
             col_names = c("mode","type","operator","value"))
  prefixes[(length(prefixes)-1):length(prefixes)] =
   ↪  c("raw_raw","kbbq_kbbq")
  names(dfs) <- prefixes
  # notempty <- map_lgl(dfs, ~dim(.)[1] != 0)
  # dfs <- dfs[notempty]
  fnr_var <- sym('%FNR')
  fdr_var <- sym('%FDR')
  bind_rows(dfs, .id = 'FNR_FPR') %>%
    pivot_wider(names_from = operator, values_from = value) %>%
    separate(FNR_FPR, into =
     ↪  c("FalseNegativeRate","FalsePositiveRate"),
             sep = "_", convert = TRUE) %>%
    rename(FNR = all_of(fnr_var)) %>%
    rename(FDR = all_of(fdr_var)) %>%
    mutate(FDR = FDR/100) %>%
    mutate(FNR = FNR/100) %>%
    mutate(across(c(FalseNegativeRate,FalsePositiveRate),
                  ~factor(., levels =
                   ↪  c("raw","kbbq",0,20,40,60,80,100))))
}

# -- Main ---
fltdf <- import_flt_tsvs(fnr,fnr) %>%
  calc_f_and_filter()
```

```
#The patterns in the data do not significantly change after
↪  filtration;
# the better-calibrated data maintains more TPs, more FPs, higher
↪  sensitivity
# and lower precision.

#Better calibrated data has more TPs
pdf("../figures/flt_tp_heatmap.pdf", width = 9, height = 7)
fltdf %>%
  ggplot(aes(FalseNegativeRate,FalsePositiveRate)) +
  geom_raster(aes(fill = TP)) +
  scale_fill_viridis_c("TP Calls") +
  scale_x_discrete("Variable Sites FNR") +
  scale_y_discrete("Variable Sites FPR") +
  ggtitle("Filtered True Positive Calls") +
  coord_fixed(ratio = 1)
dev.off()

#and more FPs
pdf("../figures/flt_fp_heatmap.pdf", width = 9, height = 7)
fltdf %>%
  ggplot(aes(FalseNegativeRate,FalsePositiveRate)) +
  geom_raster(aes(fill = FP)) +
  scale_fill_viridis_c("FP Calls") +
  scale_x_discrete("Variable Sites FNR") +
  scale_y_discrete("Variable Sites FPR") +
  ggtitle("Filtered False Positive Calls") +
  coord_fixed(ratio = 1)
dev.off()

#Shown together
pdf("../figures/flt_tp_fp_plot.pdf", width = 10, height = 6)
fltdf %>% ggplot(aes(TP, FP)) +
  geom_point(aes(color = factor(FalseNegativeRate))) +
  scale_color_viridis_d("Variable\nSites\nFNR", option = 'viridis') +
  ggtitle("Filtered Positive Calls")
dev.off()

#This leads to an increase in sensitivity
pdf("../figures/flt_sensitivity.pdf", width = 9, height = 7)
fltdf %>%
  ggplot(aes(FalseNegativeRate, FalsePositiveRate)) +
  geom_raster(aes(fill = recall)) +
  scale_fill_viridis_c('Sensitivity', option = "viridis") +
  xlab("Variable Sites FNR") +
  ylab("Variable Sites FPR") +
  ggtitle('Filtered Sensitivity') +
```

```
  coord_fixed(ratio = 1)
dev.off()

#But a decrease in precision
pdf("../figures/flt_precision.pdf", width = 9, height = 7)
fltdf %>%
  ggplot(aes(FalseNegativeRate, FalsePositiveRate)) +
  geom_raster(aes(fill = precision)) +
  scale_fill_viridis_c('Precision', option = "viridis") +
  xlab("Variable Sites FNR") +
  ylab("Variable Sites FPR") +
  ggtitle('Filtered Precision') +
  coord_fixed(ratio = 1)
dev.off()

#Shown together:
pdf("../figures/flt_sens_precision.pdf", width = 9, height = 7)
fltdf %>% ggplot(aes(precision, recall)) +
  geom_point(aes(color = factor(FalseNegativeRate))) +
  # coord_fixed(ratio = 1) +
  scale_color_viridis_d("Variable\nSites\nFNR", option = 'viridis') +
  ggtitle("Filtered Sensitivity And Precision") +
  scale_x_continuous("Precision") +
  scale_y_continuous("Sensitivity")
dev.off()

#Show before/after filtering of sensitivity and precision:
bothdf <- bind_rows(before = df, after = fltdf, .id = "filtering") %>%
  pivot_wider(id_cols = c(FalseNegativeRate, FalsePositiveRate),
              names_from = filtering, values_from = c(recall,
              ↪  precision))
bothdf %>% ggplot(aes(precision_before, recall_before)) +
  geom_segment(aes(xend = precision_after, yend = recall_after,
                   color = factor(FalseNegativeRate)),
               arrow = arrow()) +
  scale_color_viridis_d("Variable\nSites\nFNR", option = 'viridis')

# In contrast to the pre-filtration data, the best-calibrated data
↪  now
# has the best F-statstic
pdf("../figures/flt_f_plot.pdf", width = 9, height = 7)
fltdf %>%
  ggplot(aes(FalsePositiveRate, f)) +
  geom_point(aes(color = FalseNegativeRate)) +
  geom_line(aes(color = FalseNegativeRate, group =
  ↪  FalseNegativeRate)) +
  scale_color_viridis_d("Variable\nSites\nFNR", option = "viridis") +
```

```
  ggtitle("Filtered F-statistic")
dev.off()

# F Heatmap
pdf("../figures/flt_f_heatmap.pdf", width = 9, height = 7)
fltdf %>%
  ggplot(aes(FalseNegativeRate, FalsePositiveRate)) +
  geom_raster(aes(fill = f)) +
  scale_fill_viridis_c('F', option = "viridis") +
  xlab("Variable Sites FNR") +
  ylab("Variable Sites FPR") +
  ggtitle("Filtered F-statistic") +
  coord_fixed(ratio = 1)
dev.off()

# ---- Simulated Data Summaries -----
get_sim_f <- function(df){
  df %>%
    group_by(CalibrationMethod) %>%
    mutate(recall = TPc / (TPc + FN)) %>%
    mutate(precision = TPc / (TPc + FP)) %>%
    mutate(f = 2 * precision * recall / (precision + recall))
}

import_sim_tsvs <- function(){
  datanames <-
  ↪  c("ngm","ngm.recal","initial-calls.recal","kbbq-ngm.recal")
  nicenames <- c("Raw","GATK","Initial-calls","KBBQ")
  summary_files <-
  ↪  paste0("../data/sims/",datanames,".vcf.gz.summary.txt")

  dfs <- map(summary_files, read_tsv,
             col_names = c("mode","type","operator","value"))
  names(dfs) <- nicenames

  fnr_var <- sym('%FNR')
  fdr_var <- sym('%FDR')
  bind_rows(dfs, .id = 'CalibrationMethod') %>%
    pivot_wider(names_from = operator, values_from = value) %>%
    rename(FNR = all_of(fnr_var)) %>%
    rename(FDR = all_of(fdr_var)) %>%
    mutate(FDR = FDR/100) %>%
    mutate(FNR = FNR/100) %>%
    mutate(CalibrationMethod = factor(CalibrationMethod, levels =
    ↪  nicenames)) %>%
    only_snps() %>%
    get_sim_f()
```

```
}

sim_tsvs <- import_sim_tsvs()
pdf("../figures/sims_sens_precision.pdf", width = 9, height = 7)
sim_tsvs %>% ggplot(aes(precision, recall)) +
  geom_point(aes(color = CalibrationMethod), size = 3) +
  ggtitle("Sensitivity and Precision of Calls\non Simulated Reads") +
  scale_x_continuous("Precision") +
  scale_y_continuous("Sensitivity") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_color_OkabeIto(name = 'Calibration Method', use_black = T,
  ↪  drop = FALSE)
dev.off()

pdf("../figures/sims_fstat.pdf", width = 9, height = 7)
sim_tsvs %>%
  ggplot(aes(CalibrationMethod, f)) +
  geom_point(aes(color = CalibrationMethod), size = 3) +
  ylab("F-statistic") +
  xlab("Calibration Method") +
  ggtitle("F-statistic of Calls on Simulated Reads") +
  scale_color_OkabeIto(name = 'Calibration Method', use_black = T,
  ↪  drop = FALSE)
dev.off()

sim_table <- sim_tsvs %>%
  select(CalibrationMethod, TPc, FP, recall, precision,f) %>%
  rename(TP = TPc, Sensitivity = recall, Precision = precision,
  ↪  "F-statistic" = f)

write_tsv(sim_table, "../tables/sims_summary.txt")
```