

Significant Intratumor Genomic Heterogeneity (SIGH) Program User Manual

By: Sean Wang (seanw@terpmail.umd.edu)

Last Modified: August 22, 2013

Table of Contents

Introduction.....	3
Installation and Preparation.....	4
Change log.....	4
Software Requirements.....	4
Software Troubleshooting.....	4
Hardware Requirements.....	4
Software Usage Instructions.....	5
How to Use the Software.....	5
Command Line Operation.....	8
Output Examples.....	11
Frequently Asked Questions/Bugs.....	12
Creating Simulation Data.....	13

Introduction

Intra tumor heterogeneity is a major confounding factor in studying individual clones that cannot be resolved readily by global sequencing. Systematic efforts to characterize intra tumor genomic heterogeneity must effectively distinguish significant genuine clonal sequences from probabilistic fake derivatives. We describe a Java-C++ software tool for identifying significant intra tumor genomic heterogeneity (SIGH) directly from mixed sequencing reads that can improve genomic analyses in many oncological contexts. This open source software offers several attractive features: (1) it is supported by a well-grounded statistical framework that exploits probabilistic characteristics of sequencing errors; (2) it identifies genuine clonal sequences from raw sequence reads using significance tests; (3) its unbiased strategy allows detecting rare clone(s) despite that clone's relative abundance; and (4) it estimates constituent proportions in each sample. SIGH also provides the users with a parallel computing option utilizing ubiquitous multi-core machines.

We describe a statistical sequence modeling approach for identifying significant intra tumor genomic heterogeneity (SIGH) directly from mixed sequencing reads rather than variant signatures. The Java-C++ SIGH software tool was developed to identify the number and sequences of genuine subclones justified by model-based significance tests. SIGH works by exploiting the statistical differences in both the sequencing error rates at different nucleotide bases and the read counts of fake sequences in relation to genuine clones of variable abundance. We conduct a simulation study to demonstrate the performance of SIGH under a variety of conditions on realistic synthetic data derived from immune T-cell sequences.

Installation and Preparation

Change log

08/21/13

-v1.0 Runnable and semi-functional as a distributed program. Must manually move the barcode file.

07/23/13

-v0.3 GUI shows up on other computers, but is not functional

Software Requirements

-Ubuntu 12.04 LTS or higher

-Kernal Linux 3.5.0-34-generic or higher

-GNOME 3.4.2 or higher

-java version "1.7.0_21" or higher

-OpenJDK Runtime Environment (IcedTea 2.3.9) (7u21-2.3.9-0ubuntu0.12.04.1) or higher

-OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode) or higher

Software Troubleshooting

If this error or something similar appears:

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: /tmp/natives-136539359/libSIGH_Proj.so: /lib/libc.so.6: version
`GLIBC_2.14' not found (required by /tmp/natives-136539359/libSIGH_Proj.so)
    at java.lang.ClassLoader$NativeLibrary.load(Native Method)
    at java.lang.ClassLoader.loadLibrary0(ClassLoader.java:1928)
    at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1854)
    at java.lang.Runtime.loadLibrary0(Runtime.java:845)
    at java.lang.System.loadLibrary(System.java:1084)
    at sigh_proj.SIGH_Proj.<clinit>(SIGH_Proj.java:23)
```

Make sure you are on Ubuntu 12.04 or higher.

Installing libc6-dbg is also advised. This can be done with the following Linux command: **sudo apt-get install libc6-dbg**

You can check your java version with the following command: **java -version**

If you do not have java installed, you can install it using this command: **sudo apt-get install openjdk-7-jre**

Hardware Requirements

-16GB RAM

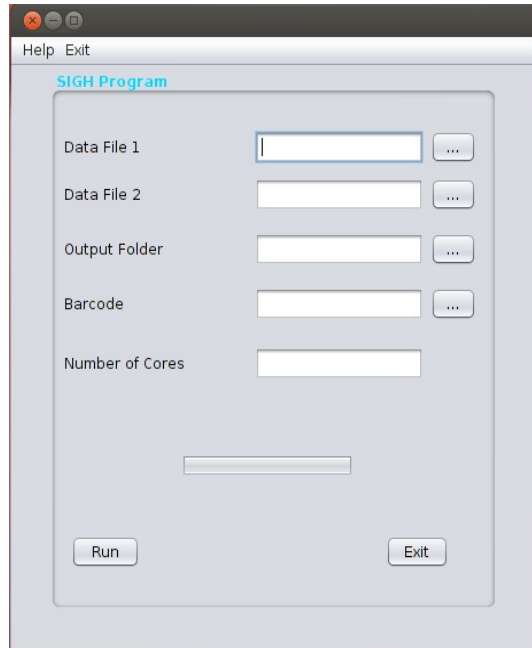
-Intel® Xeon(R) CPU E5-2665 0 @ 2.40GHz × 18 (36 cores)

-1 TB available HD space

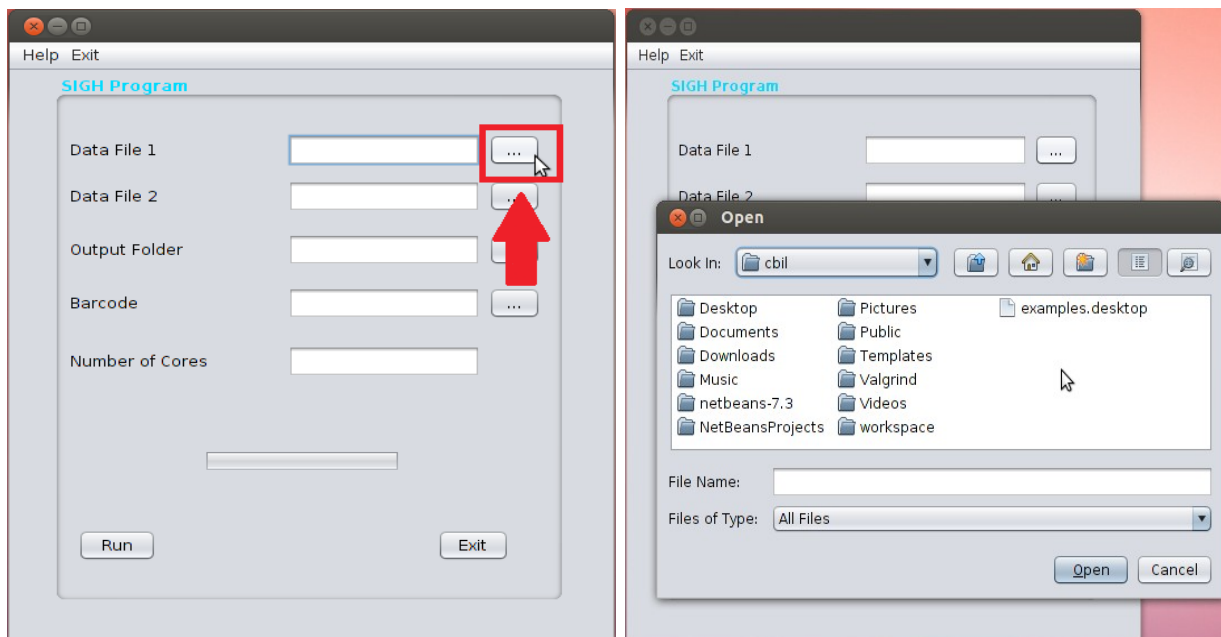
Software Usage Instructions

How to Use the Software

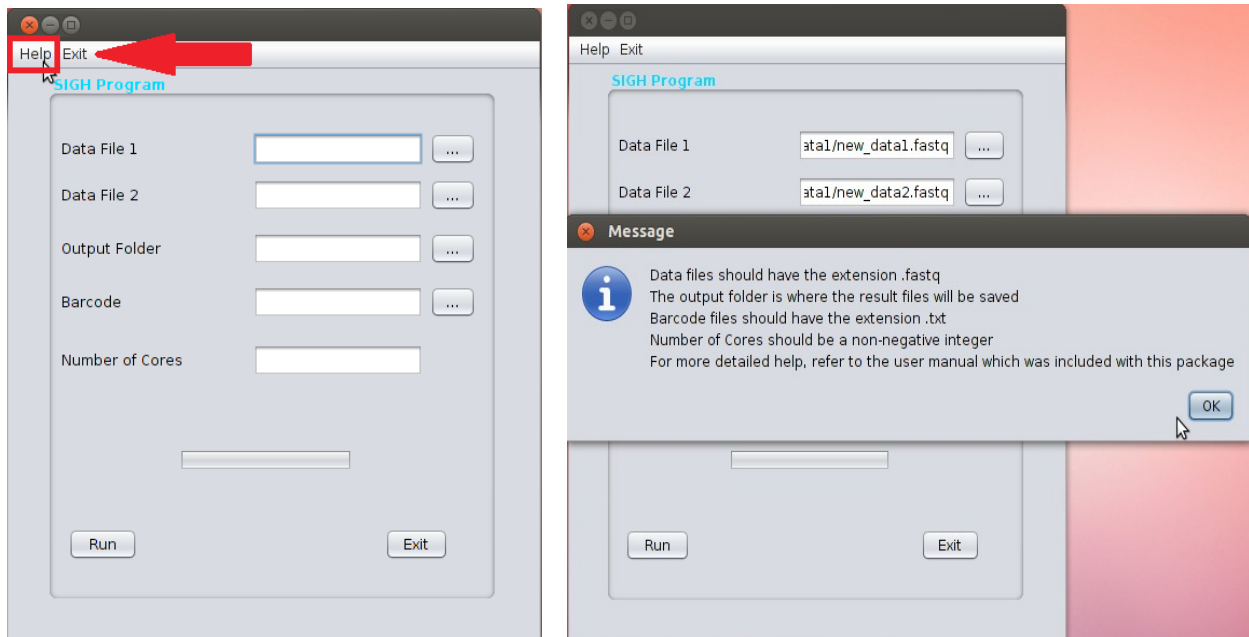
1. Navigate to the location of SIGH.jar on your computer
2. run the JAR as an executable. This can be done with the command: **`java -jar SIGH.jar`**



3. input the corresponding files using the file selection button



- click the “Help” menu item in the top left if you are unsure of which files correspond to which category

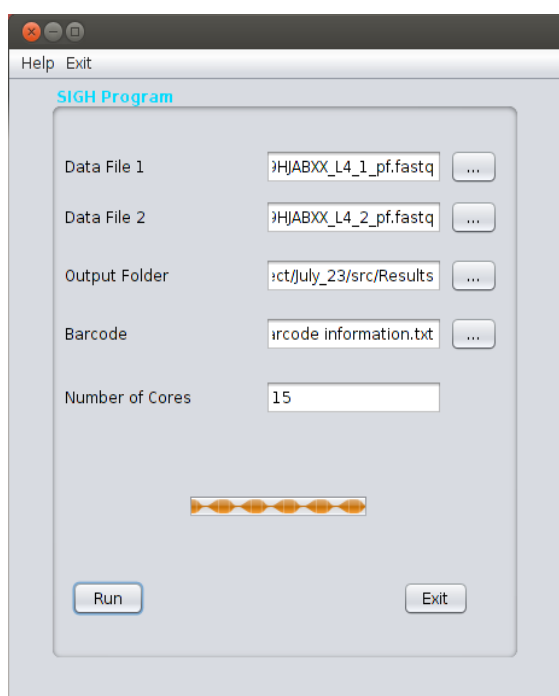


- Enter the number of cores you would like to use. You can find the number of cores your computer has by with this console command: ***grep -c processor /proc/cpuinfo***

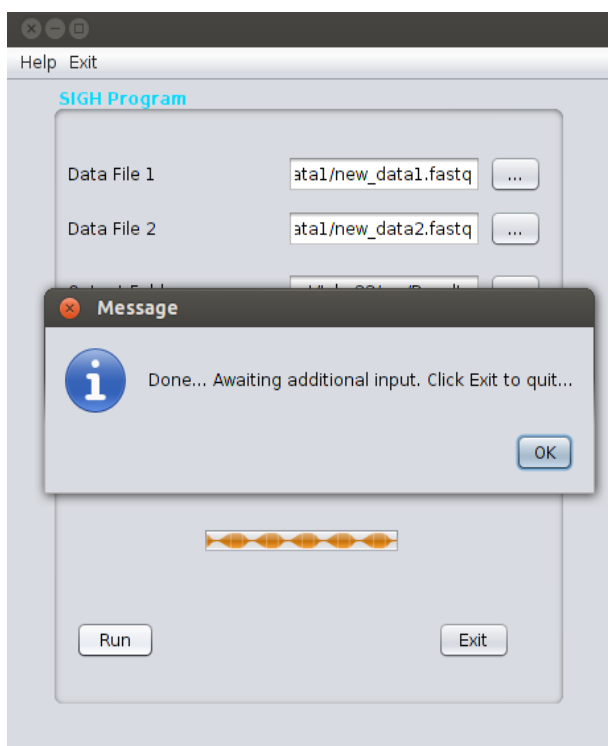


Remember that this must be a non negative integer!

- Click “Run” and the working bar will start moving, indicating that the program is analyzing the data.



- After the program has analyzed the data, a prompt should appear saying that the program has finished running. **If this prompt does not appear, there is an error. Contact Virginia Tech.**



Command Line Operation

Files:

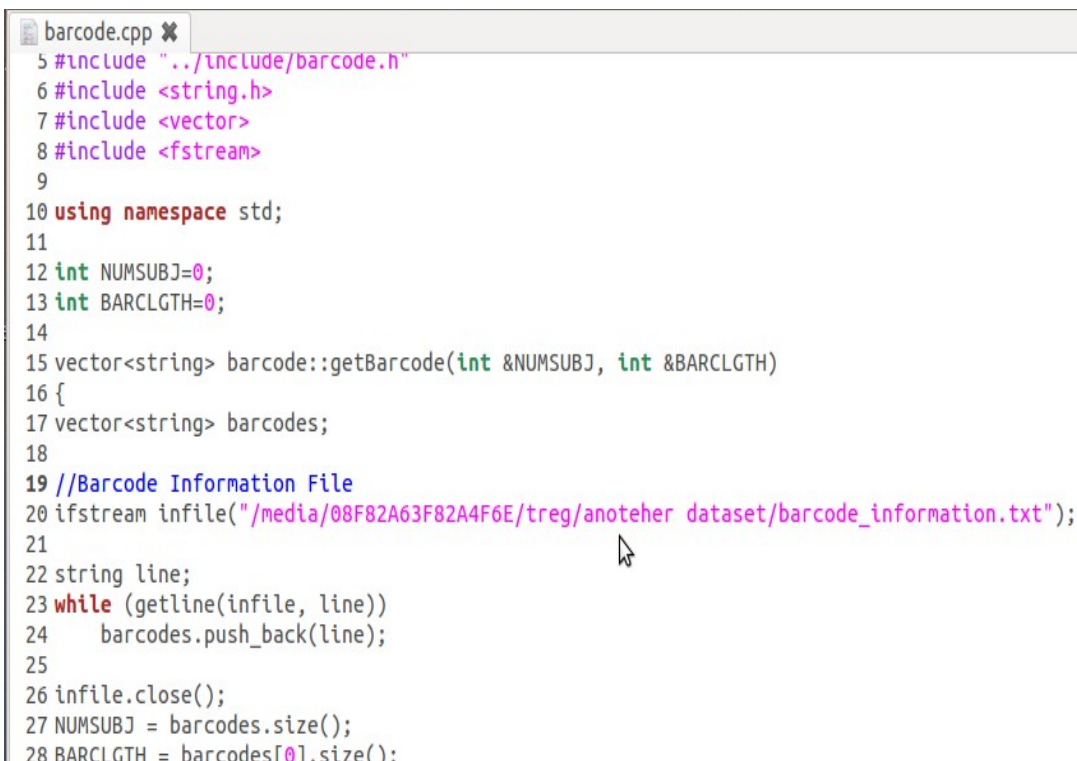
Inside the main program folder, make sure that you have a directory named “src” that contains barcode.cpp, gcode.cpp, JReg.cpp, peseq.cpp, tcr.cp, and VReg.cpp. You also need the directory named “include” that contains barcode.h, gcode.h, Jreg.h, peseq.h, and Vreg.h. Finally, make sure you also have a text file called “Makefile”, which includes compiling directions.

Also, check that you have both forward and backward read .fastq files and your barcode information file (generated by sequencing machine). You should also know the barcode step, which is the number of dummy reads before the barcode of a subject appears in the raw data (this value varies depending on which sequencing machine was used to generate the raw data).

Setup:

The user will have to change the barcode information file path and the barcode step.

-Changing barcode information file path: Navigate to the “src” folder and open “barcode.cpp”. Once you have opened barcode.cpp, find the function called 'barcode::getBarcode'; there is an input filestream(ifstream) type called “infile” that is defined in this function (you can find it by looking for the comment “//Barcode Information File”). Replace what is inside the quotation marks with the full pathname to your barcode information file. Save and exit.



```
barcode.cpp
5 #include "../include/barcode.h"
6 #include <string.h>
7 #include <vector>
8 #include <fstream>
9
10 using namespace std;
11
12 int NUMSUBJ=0;
13 int BARCLGTH=0;
14
15 vector<string> barcode::getBarcode(int &NUMSUBJ, int &BARCLGTH)
16 {
17     vector<string> barcodes;
18
19     //Barcode Information File
20     ifstream infile("/media/08F82A63F82A4F6E/treg/anoteher dataset/barcode_information.txt");
21
22     string line;
23     while (getline(infile, line))
24         barcodes.push_back(line);
25
26     infile.close();
27     NUMSUBJ = barcodes.size();
28     BARCLGTH = barcodes[0].size();
```


-*Changing the barcode step*: Navigate to the “include” folder and open “barcode.h”. Near the top, you will see “#define BARCSTP” followed by a number. This number is the barcode step. Change it to your data's specific barcode step. Save and exit.

```
barcode.h
1 //Class for mapping triplet of nucleotide to amino acid
2 //Guoqiang Yu @Stanford
3 //Jan.3, 2012
4
5 #include <vector>
6 #include <fstream>
7 #include <string.h>
8
9 //define NUMSUBJ 18
10 #define BARCSTP 1 //Barcode sequence starting position (3 for s1_1.fastq/s
11 //define BARCLGTH 6 //Barcode sequence length
12 #define BARBOOKSZ 4096
13 extern int NUMSUBJ;
```

Compiling SIGH:

Navigate to the main program folder in the command line and type:

```
make -f Makefile
```

and press Enter. If no changes have been made to the source code, you will see a message that states: *make: nothing to be done for 'all'*.

If changes have been made, you will see some output detailing the progress of the compiling. (e.g. `g++ -O -fopenmp -c -o peseq.o src/peseq.cpp`) Many files with a .o extension will be created in the main folder.

Running SIGH:

Navigate to the main program folder in the command line and type:

```
./TCR input1 input2
```

where input1 and input2 are the full file paths to the raw data (.fastq extension). Once you have checked that the file paths are correct, press Enter. You will see some output detailing the properties of your input such as the filenames and the quality scoring system used by the sequencing machine that created the raw data. Next, you will be prompted:

```
If you would like to use the maximum number of cores, enter 999,
otherwise specify.
```

```
Enter cores:
```

followed by a blinking box. Enter the number of cores you would like to use during the program. It is advised to leave at least 1 core free so that the user may use the cores that are not dedicated to the SIGH program for other tasks. You can determine the number of cores your machine has by looking at your computer information, or type 999 to utilize the maximum number of cores your machine has if you are unsure how to check.

Output (Command Line):

After inputting the desired number of cores, the progress of the program will be printed to the console. The preliminary pre-processing of the data uses the most resources, and the time necessary to complete the pre-processing will vary based on how many available cores were specified, random access memory (RAM), and the overall processing speed of the CPUs; on a 12 core machine with ~2.8GHz clockspeed and 20 GB RAM, a 40 GB total raw data file size took about 2-3 hours to complete pre-processing. Pre-processing will be complete after you see this printed to the console:

```
Getting reads...
totreads is: number
getreads() complete
working...
alignall() complete.
```

Afterwards, the remaining computation and result output will occur automatically and you will see more supplementary information printed to the console:

```
begin uqcdr3clone()
mgsort() complete
cmpnucseq() complete.
record uq clone complete.
unique barcodes with at least three differences to others: x
Err Prob from A to C: 0.000000000,    y,
Err Prob from A to G: 0.000000000,    y,
...
Err Prob from T to C: 0.000000000,    y,
Err Prob from T to G: 0.000000000,    y,
14365
calbarerrprob() complete.      //xxxxxxx is probability of error,
calerrprob() complete.        //y is frequency of error in
begin intra correction         // the input data.
begin indel correction
begin inter correction
begin J24FL correction
begin J24LF correction
deletion rate: number
```

After the deletion rate is printed out, the program will be complete and results will be located in a directory named “Results” created in the main program folder. The contents of the results is detailed in the “Output Examples” section of this manual.

Output Examples

cdr3count.txt: Shows the number of subjects in the study

S.1	S.2	S.3	S.4	S.5	S.6	S.7	S.8	S.9	S.10	S.11	S.12	S.13	S.14
	S.15	S.16	S.17	S.18	S.19	S.20	S.21	S.22	S.23	S.24	S.25	S.26	S.27
	S.28	S.29	S.30										
0	0	0	0	0	0	0	0	0	15	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0										
0	0	0	0	0	0	0	0	0	9	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0										

cdr3seq.txt: Shows the nucleotide sequences of the clones

V101	J2401	CGGGGGAAATTCCAG-	RGKFQ-
V101	J2401	TACTGGGGGAAATTCCAG-	YWGKFQ-
V101	J2401	GCCCTAGGGGAATTCCAG-	ALGEFQ-
V101	J2401	GCTGTGGGCTTCAAATTCCAG-	AVGFKFQ-
V101	J2401	GCTGTGAGAGGGAAATTCCAG-	AVRGKFQ-
V101	J2401	GCCAGCTGGGGGAAATTCCAG-	ASWGKFQ-
V101	J2401	GCTTCAAGCTGGGGGAAATTCCAG-	ASSWGKFQ-
V101	J2401	GCCTTAAGCTGGGGGAAATTCCAG-	ALSWGKFQ-
V101	J2401	GCTGACAGCTGGGGGAAATTCCAG-	ADSWGKFQ-
V101	J2401	GCTCCCAGCTGGGGGAAATTCCAG-	APSWGKFQ-
V101	J2401	GCTGTGAGAGCCCTAGGGAAATTCCAG-	AVRALGKFQ-

errv.txt: Shows the error percentages from the sequencing machine

Err Prob from A to C: 0.000405672
to G: 0.00115455
to T: 1e-05
Err Prob from C to A: 0.000105263
to G: 1e-05
to T: 0.000541791

resstat.txt: Shows error statistics of each subject. Smaller threshold implies greater accuracy

```
Subject 1, the expected err: 0.0366175  
threshold: 0.00165802
```

```
Subject 2, the expected err: 0.0405998  
threshold: 0.00110765
```

```
Subject 3, the expected err: 0.0357154  
threshold: 0.00207523
```

stvjp.txt: Shows the genuine clones. The first sequence is the clone, the second is the full read (from input2;reverse read) with V and J regions appended.

```
subject: 1;    V Allele: 101;  J Allele: 2401  
GACCTTAAAGGGGGTCGAATTCCG  
clone size:    1  
CTCCTTCTACAGGAGCTCCAGATGAAAGACTCTGCCTCTTACCTCTGTGCCTTAAGCTGGGGGAAATTCCAGT  
p value: 0.000883647
```

Frequently Asked Questions/Bugs

-My Program outputs the data correctly, but throws an error! Whats wrong?

Make sure that data files are on the same hard drive as the program. That means that data stored on a USB/CD-ROM should be transferred to the computers hard drive.

Creating Simulation Data

We want to create simulation data for the purpose of validating our method by comparing the results with some ground-truth that is not readily available with real sample data from sequencing machines. However, we use the real sample data to strategically extract raw reads and information so that our simulation data will be as close to the real case as possible while still retaining creative control over the ground-truth aspect of the data.

Step 1: Obtaining the reads files

Navigate to the src folder and open pseq.cpp.

-near the top, you will see “`#define DFRAC number`”, it is advisable to change this to a larger number. DFRAC will tell the program to only process $1/\text{DFRAC}$ of the input data. Since we are only creating simulation data, we do not need all of the raw data processed; a DFRAC of ~ 20 is recommended for data sets larger than 20 GB.

-find the `pseq::getreads()` function, a simple way to do this without scrolling through the entire file is to use Ctrl+F and type “getreads” into the search prompt, it should automatically bring you to the function. Uncomment the block of code at the beginning of the function that has the comment: “for the purpose of creating simulation data” next to it by deleting the “`/*`” and “`*/`” symbols surrounding the block. Within the same function, scroll down and uncomment another block of code that is preceded by the “for the purpose of creating simulation data” comment using the same method as before.

First block:

```
/*ofstream out("reads.txt"); //for the purpose of creating simulation
data
ofstream out2("reads2.txt");
ofstream gout("gout.txt");
ofstream gout2("gout2.txt");*/
```

Second block:

```
/*
//for the purpose of creating simulation data.
//This outputs 4 files that remove bad reads and their corresponding
quality lines
////////////////////////////////////
char tmreadln1[READLENGTH];
    ...
    ...
    gout2.write(gread2[i]+j*READLENGTH,READLENGTH);
    gout2 << endl;
}
}
////////////////////////////////////
*/
```

Now setup, compile, and run the program as detailed in the manual. After the program finishes running, there should be 4 .txt files that appear in the main program folder called reads.txt, reads2.txt, qout.txt, and qout2.txt. **All should have the same file size.** **Also**, note the probabilities of error for base pairs (i.e. Err Prob from A to C: *number*, etc) as you will need these for later.

Step 2: Determine characteristics of Simulation Data

Think about how many clones you want in the simulation data and the size of the clones. You will have to create a text file called 'readIndex.txt' that has the index of the reads you would like to be the clones in the simulation data:

e.g.

```
//readIndex.txt
1000
2000
3000
```

The above example will create simulation data with 3 clones, and the sequence for the clones will be extracted from index 1000, 2000, and 3000 from reads.txt and reads2.txt.

*You can use addfunc.exe to see specific reads. Compile using:

```
g++ addfunc.cpp -o addfunc.exe
```

Run using:

```
./addfunc.exe
```

You will be prompted to enter a desired read. This read is the index of your desired read in reads.txt and reads2.txt. For example, if you wanted to use the 300,000th read in reads.txt and reads2.txt, type 300000 and press enter, the program will return:

```
unmodified Fore: sequence1
unmodified Back: sequence2
```

Sequence1 is the forward read from reads.txt and Sequence2 is the backward read from reads2.txt.

In general, you can just use random indexes when creating simulation data; addfunc will allow you to see what the read/sequence actually looks like so you can know the barcode sequence. The location and length of the sequence changes based on the real sample data. You will have to create your own barcode file. Name it "barcode_information.txt" and type each barcode on its own line.

Step 3: Setting up final parameters

Open 'combineArray.cpp'. At the top, you will see a block of #define statements labeled AtoC, AtoG, AtoT, etc. Change these values to the numbers outputted to the console from when you ran the real sample data **multiplied by 10000**. For example, if you see on the console: Err Prob from A to C: 0.000314663, then do:

```
#define AtoC 3.14663
```

Repeat for the other #define statements.

Now, scroll down until you find the main() function or Ctrl+F 'main'. You will see two for loops. The variable j represents the number of clones and the variable, i, represents the size of the clones:

```
for(int j=0;j<3;j++)
{
    ...
    ...
    for(int i=0;i<30000-(j*10000);i++)
    {
        ...
        ...
    }
}
```

The portion highlighted in red is what you change to determine the values for i and j. You must use creative math if you want different clone sizes. For example, in the above example, i is:

```
30000-(j*10000)
```

which will make the first clone 30000-(0*10000), the second clone 30000-(1*10000), and the third clone 30000-(2*10000); or 30000,20000,10000.

Step 4: Run

Compile by typing:

```
g++ combineArray.cpp -o combineArray.exe
```

Then run:

```
./combineArray.exe
```

When it finishes running, you will see two data files called “SimData1.fastq” and “SimData2.fastq” appear in the main folder. These are the 2 simulation data files.