FISHR Programming Project Documentation

System Overview:

The FISHR program consists of 4 component programs, each with its own set of tasks:

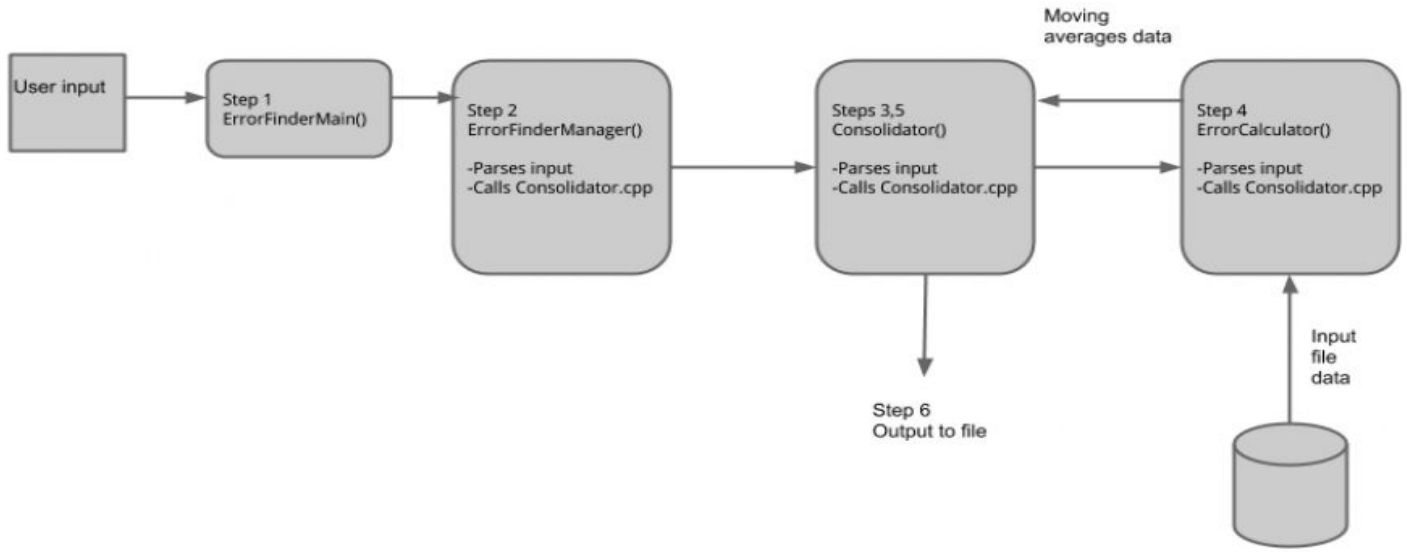
ErrorFinderMain.cpp: This is the program that has the actual main() program from which all other tasks are spawned.

ErrorFinderManager.cpp: This program is responsible for parsing user input from the command line, and for calling the appropriate function sequences based on that user input.

Consolidator.cpp: This program performs a variety of tasks, such as consolidatoring SHs (performConsolidation()) if they differ by a length that is less than a user supplied gap, or trimming SHs based on a variety of parameters (performTrim()).

ErrorCalculator.cpp: This program focuses on mathematically calculating a variety of the parameters that Consolidator.cpp uses to trim or consolidate SHs, such as finding moving averages (getMovingAverages()), etc.

A diagram of how the system works is shown below:



FISHR has several key algorithms that it uses to produce results. One such algorithm is used when finding the implied error averages over a shared haplotype. This algorithm is ErrorCalculator::getMovingAverages(). When we are attempting to caclulate the trim positions on a SH, we use the implied error average at each SNP on that SH to determine whether or not we trim at that point. For example, suppose we have a empirical error threshold of 0.4. This means that we start at the midpoint of the SH, and work our way towards the start and the end of the SH until we find a point that has an error average greater than or equal to 0.4. If we find one, we "trim" the SH at that point, and it becomes the new start or end point for that SH. If we don't find any, then the SH is not trimmed. It's worth pointing out that deciding whether or not to trim actually occurs in Consolidator::performTrim(), but it helps to know why we actually calculate moving averages in the first place.

Moving Averages:

File: ErrorCalculator.cpp

Class: ErrorCalculator
Function-Signature: vector<float> ErrorCalculator::getMovingAverages(vector<int> errors, int snp1, int snp2, int width)
Description: The inputs to this function are thus:
snp1 and snp2: The start and end of the Shared Haplotype segment
errors: This can take on a range of integer values(say, between 2 and 530 for example), so it is most probably the locations of SNPs within the SH, where errors are occurring.
width: remains constant, during one run was set to 10. Perhaps this is the window size.

A variable called length is set up within the function, which is the length of the SH segment. The algorithm itself is divided into several stages. The first stage deals with calculating the very first element of the moving averages **array(averages[0])**. Assuming a window size of **10**, we can't reasonably calculate **averages[0]** by summing up its **10** neighboring elements, because there are no elements to the left of **e_Places[0](array subscripts can't be negative in c++)**. Thus, for this corner case, the **0th** element of **averages[]** is calculated by summing up the first **width/2(in this case, 5) elements of e_Places(e_Places[0] + … + e_Places[4]), call them e0, e1, …, e4.** Thus:

averages[0] = e0+e1+e2+e3+e4 = s0 (for the **0th** summation, this will come in handy in a minute). now, in the code, a temporary variable called present is initially set to **width/2**(in this case, **5**). This value is incremented until it is equal to width(that is, we have hit the sliding window size. This is done - as far as I can tell - to make sure that we don't understep(access negative indices) of the averages array while we are calculating the elements whose indices are less than the sliding window size). In this example, where width is **10** and present thus starts at **5**, we would see the following sequence of events:

averages[i] = averages[i-1] + e_Places[++present];
averages[0] = e0+e1+e2+e3+e4 = s0
averages[1] = s0 + e_Places[6] = s1, present = 6 Note, the way this is implemented appears to skip e_Places[5].
averages[2] = s1 + e_Places[7] = s2, present = 7
averages[3] = s2 + e_Places[8] = s3, present = 8
averages[4] = s3 + e_Places[9] = s4, present = 9
averages[5] = s4 + e_Places[10] = s5, present = 10

now, **present** is no longer **< width**, so it will fail that conditional and pass into another part of the program for the remainder of the averages array. This is significant because this portion of the algorithm begins to actually "move" the window by introducing another variable called "previous", which is initially set to **0**. It brings up the tail-end of the window, as demonstrated:

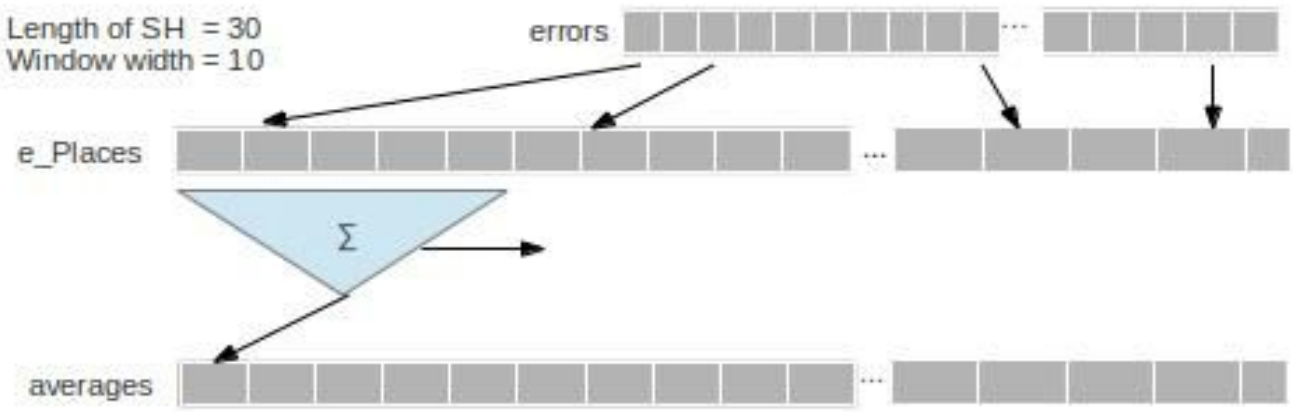
for i such that present >= width (in this case, all i > 5):
averages[i] = averages[i-1] + (e_Places[present] - e_Places[previous])
averages[6] = s5 + (e_Places[10] - e_Places[0]) = s6, present = 11, previous = 1

note that **s6 = e0+e1+e2+e3+e4+e6+e7+e8+e9+e10+e10 - e0,**

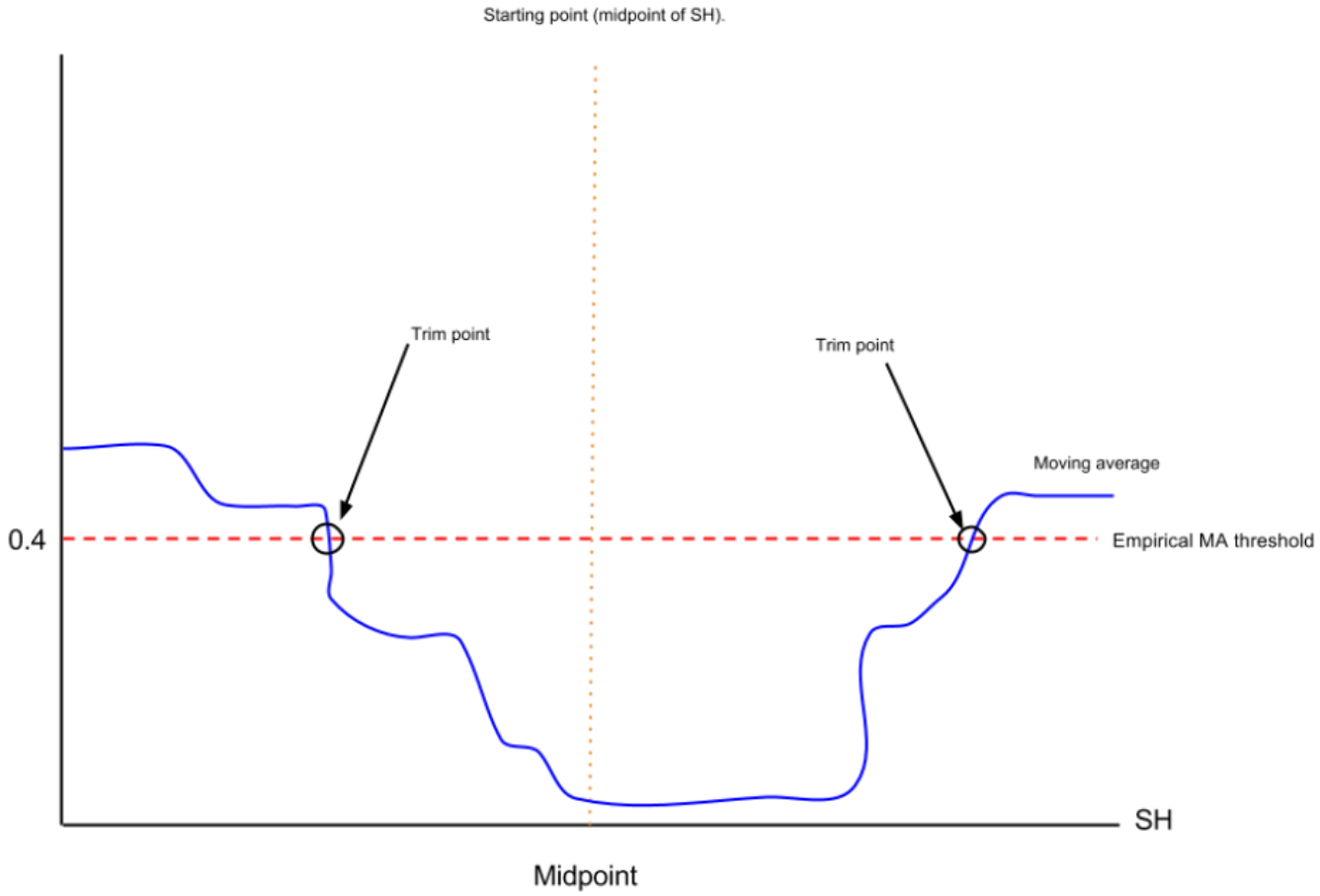
so the algorithm has subtracted out the bottom-most element, and is therefore moving the algorithm.

Note, if the expansion of s6 looks weird, that's because it should. I'm not sure if e5 is being skipped, and if e10 is calculated twice, but I will look into it now-nate

Once a full pass has been made through the averages array, the algorithm loops through it again, performs the following set of operations:
In the (unusual) event that any element of the array has a value of less than 0, it is set to 0.
From there, for the first **i<(width/2)** elements(the first **5**, in our running example), the calculated value is set to:
averages[i] = (averages[i] / (width/2 + i)) This appears to be averaging over the first few elements, once again handling the corner case of being too close to the start of the array.
Similarly, for the remaining elements, it simply averages them in parts over the window size, until it hits the last few elements that would be within a window size. There, it averages them over a smaller window size**(width/2)**.



Another one of FISHR's key algorithms is **Consolidator::performTrim()**, which takes input from **movingAverages()** and uses that to determine at which points trimming should occur. The motivation behind this function is that sometimes, a **SH** may contain a significant number of errors (specifically, a number that exceeds the empirical error threshold). When this case occurs, we trim off the offending regions, thereby reducing the size of the **SH** to something that has less errors.



The outputperformTrim is a SH that is less than or equal to the length of the original SH.

How PIE is Calculated:

PIE (percentage of implied errors) is currently calculated by taking the number of implied errors found in a given shared haplotype, and dividing that number by the length of the shared haplotype, with the exception that errors that occur at both the first and last positions of the shared haplotype are dropped from the numerator in this calculation.

For example:
columns: (implied error location, start of SH, end of SH, bp, PIE calculation)

1/20/50 , 1 , 50, 50, 1/50

1/5/20/100/200/300/320/330, 99, 301, 203, 3/203
1/5/20/100/200/300/320/330, 100, 300, 201, 1/201
Note that in the first example, the SH starts at position 1(which has an implied error) and ends at position 50, which also has an implied error. However, neither of these errors are used in the numerator, and the resulting PIE calculation is 1/50 as opposed to 3/50.

Software Documentation

ErrorFinderMain.cpp

This file contains the main function for **FISHR**, and is responsible for creating and activating an **ErrorFinderManager** object that actually runs **FISHR**.

ErrorFinderManager.cpp/.hpp

This file is responsible for parsing user input from the command line, as well as executing various FISHR calculations:

void performConsolidation(int argc,cahr *argv[])
This function is responsible for parsing user input from the command line, and for initializing some of the other parts of FISHR, such as performing SH trims, etc.)

void displayError(std::string argv)
This function simply displays an error message detailing proper FISHR command line usage, should the user provide some sort of erroneous input.

Consolidator.cpp

Consolidator.cpp is one of the largest files in fishr. It contains functions that handle the consolidation of SH, the trimming of SH, as well as several other functions. The API provided by Consolidator.cpp is detailed here:

float Consolidator::average_snp_count()
This function is used with the weighted column calculation. It will sum over every non-zero element in the genome vector, and then divide by the length of the genome vector (minus any zero elements).

bool Consolidator::compareFunction(SNP s1, SNP s2)
This function will compare the start location of s1 with the start location of s2 and will return true if s1 starts before s2. Used with the c++ algorithm library.

void Consolidator::finalOutput(ErrorCalculator &e,float min_cm, int min_snp)const
Use of this function is depricated This function outputs data to stdout in the finalData format. However, this is generally handled by an ErrorCalculator object of the same name, so use of this version is mostly depricated.

int Consolidator::find_genome_max()
This function is used for calculating the weighted column for the weightedOutput option. It will return the maximum element of the genome vector.

int Consolidator::find_genome_min()
This function is used for calculating the weighted column for the weightedOutput option. It will return the minimum element of the genome vector.

void Consolidator::findTruePctErrors(ErrorCalculator &e_obj,int ma_snp_ends, bool holdOut,int window,float ma_threshold, float empirical_ma_threshold)
Basically the same as findTrueSimplePercentErrors, except it uses MOL calculations.

void Consolidator::findTrueSimplePercentErrors(ErrorCalculator &e_obj, float PIElength, bool holdOut,int window, float ma_threshold, float empirical_ma_threshold)
This function calculates over trulyIBD data. Its primary function is to calculate a vector of max moving average values over the trulyIBD data that can then be indexed into to find a moving average threshold. For example, if the ma_threshold argument is 0.8, then it will take the 80th percentile of the vector calculated by this function, and use that value as the cutoff threshold in subsequent moving average calculations.

void Consolidator::findTruePctErrors(ErrorCalculator &e_obj,int ma_snp_ends, bool holdOut,int window,float ma_threshold, float empirical_ma_threshold)
Basically the same as findTrueSimplePercentErrors, except it uses MOL calculations.

float Consolidator::getholdOutThreshold(float threshold)
This function is responsible for returning the moving average threshold corresponding to the percentile (between 0 and 1) specified by the threshold argument.

float Consolidator::getPctErrThreshold(float threshold)
This function is responsible for returning the moving average threshold corresponding to the percentile (between 0 and 1) specified by the threshold argument.

void Consolidator::performTrim(ErrorCalculator& e_obj,int window, int ma_snp_ends, float ma_threshold, int min_snp,float min_cm, float per_err_threshold, string option, float hThreshold, bool holdOut,float empirical_threshold, float empirical_pie_threshold)
This is a major function in the operation of FISHR, so its algorithms will be detailed in a different section of the documentation. From a programmer's standpoint, it works as follows:

e_obj is used for writing various output formats to stdout.

window is the sliding-window size that is used in moving average calculations.

ma_snp_ends is depricated and needs to be removed.

ma_threshold is the cutoff threshold for moving average calculations.

min_snp is the minimum acceptable SNP length. SH below this length are dropped.

similarily, min_cm is the minimum acceptable cM length for a SH.

per_err_threshold is the PIE threshold.

option refers to the ErrorCalculator:: output type.

hThreshold is depricated and needs to be removed.

holdOut would indicate whether or not holdout data is being used.

empirical_threshold is a user-supplied cutoff value for moving average calculations. This cannot be used with ma_threshold.

empirical_pie_threshold is a user-supplied cutoff value for PIE calculations. This cannot be used with per_err_threshold.

This function is generally the last major step that FISHR does. performTrim will calculate moving averages and PIE values for each SH, as well as trimming them. After that, it will check to see whether or not the SH should be dropped, or if it will be kept. The results will be written to stdout using one of the output functions specified in ErrorCalculator::.

void Consolidator::print_genome()
This function is used for calculating the weighted column for the weightedOutput option. It is nothing more than an output device that will print out the contents of the genome vector.

void Consolidator::readMatches(string path,int pers_count, ErrorCalculator& eCalculator, int trueSNP, float trueCM)
This function is responsible for building the m_matches and m_trueMatches vectors that are vital to FISHRs functionality.

bool Consolidator::sortMatches()
This function uses Consolidator::compareFunction(SNP s1, SNP s2) to sort all of the SH matches between persons in a genome into ascending order. These SHs are sorted in-place in the m_matches data structure.

void Consolidator::update_genome(int snp1,int snp2)
This function is used for calculating the weighted column for the weightedOutput option. It will increment each element in the genome vector between snp1 and snp2 (inclusive) by 1.

void Consolidator::update_snp_weight(int snp1, int snp2)
This function is used for calculating the weighted column for the weightedOutput option. This will calculate the weight for a SH that starts at snp1 and ends at snp2. The contents of the genome vector over range [snp1,snp2] will be summed and then divided by the length of the SH.

ErrorCalculator.cpp Documentation

ErrorCalculator.cpp is one of the largest files in fishr. It contains functions for calculating moving averages, PIE, as well as several different output functions. The API provided by ErrorCalculator.cpp is detailed here:

void ErrorCalculator::changeMapFile(std::string path)
This is used during hold out execution. It allowd the underlying map file to be changed to the one specified by path.

vector ErrorCalculator::getTrimPositions(std::vectoraverages,int snp1,int snp2,float threshold,float minLength)
This function is used for calculating the trim positions for the SH specified by snp1, snp2. The threshold argument is used in calculating the start/end points of the trim. Trimmed SH that fall under the minLength parameter are dropped.

bool ErrorCalculator::isInitialCmDrop(int snp1, int snp2, float minLength)
This function checks to see whether or not the SH specified by start and end points snp1 and snp2 has a cM length below the minLength threshold. A value of true is returned if this is the case, indicating to the caller (usually Consolidator::performTrim) that this SH should be dropped. All cM calculations are done using the marker_id[snp].cm_distance value from the ErrorCalculator class.

void ErrorCalculator::createLogFile(std::string path)

This function will create a log file using the path variable for the log file name. If no path is supplied, it will default to FISH.log.

void ErrorCalculator::log(std::string& str)
This function will write str to the log file that was opened wirh createLogFile() using the internal ErrorCalculator::m_logger object.

void ErrorCalculator::readBmidFile(std::string path)
This function opens and parses the BMID file specified by the string path.

void ErrorCalculator::readBsidFile(std::string path)
This function opens and parses the BSID file specified by the string path.

void ErrorCalculator::readPedFile(std::string path, std::string missing)
This function opens and parses the ped file specified by the string path. Missing can be string of characters to look for in the ped file.

void ErrorCalculator::readHPedFile(std::string path, std::string missing)
This function opens and parses the ped file specified by the string path in hold out mode.