Documentation for SICER1.1

# GenomeData.py

* The script is in:  
  */home/data/SICER1.1/SICER/lib*
* This module contains:  
  1. Chromosomes (List) and their length (Dictionary):  
    
  i.e.  
  hg19\_chroms = [‘chr1’, ‘chr2’, … , ‘chrY’]  
  hg19\_chroms\_length = {‘chr1’: 249250621, …}  
    
  2. Species (Dictionary) containing all defined chromosomes and their length  
    
  i.e.  
  species\_chroms = [‘mm8’: mm8\_chroms, …]  
  species\_chroms\_length = [‘mm8’: mm8\_chroms\_length, …]

# BED.py

* The script is in:  
  */home/data/SICER1.1/SICER/lib*
* BED types:   
  BED2: start, strand   
  BED3: chrom, start and end   
  BED6: BED3 + name + score + strand ('+' or '-')   
  BED\_GRAPH: bed graph format to mimic wiggle format: chrom, start, end, value  
    
  The above formats are all defined as Class.
* Class BED  
    
  1. def \_\_init\_\_:   
  Arguments:  
    
  species=’hg18’  
    
  file=None -> The input file is .bed format.  
    
  bed\_type=”BED3” -> Specify the data format in input file, which can be the above BED2, 3, 6, and BED\_GRAPH.  
    
  val\_threshold=0 -> Only considers the bed elements with score (if have) higher than this value.  
    
  Return:  
  self.bed\_vals (Dictionary) -> keys: chromosomes for this species. values: list of bed data. Each data should belong to either of above defined BED Class  
    
  i.e.  
  bed\_vals[‘chr3’] = [<BED2 Class Type Element>, <BED\_Graph Class Type Element>, …]   
    
  2. def keys:  
  Return -> bed\_vals.keys()  
    
  3. def getStarts\_consider\_strands, getStarts, getEnds:  
  Argument -> chrom  
    
  Return -> Return a list of starts / ends positions for shit chromosome

# make\_graph\_file.py

* The script is in:  
  */home/data/SICER1.1/SICER/lib*
* def get\_bed\_coords(file, chrom\_length, fragment\_size)   
    
  1. We know the input .bed file typically has the following format for each read:  
  chrom, start, end, name, score, strand  
  We know the start position is always the end of this sequence, so depending on the fragment\_size, we can do the shift of position to get the middle position of this sequence.  
  The chrom\_length helps us to confirm the shift will make the position exceed the range of chromosome.  
    
  2. Return -> taglist (List), with sorted position for all the sequences. The format is taglist = [position1, position2, …]  
    
  Note: the input file is already only for one chromosome (done by SeperateByChrom.py)  
    
  3. Print out the total tag count (Also shows “+” and “-” count individually)
* def Generate\_windows\_and\_count\_tags(taglist, chrom, chrom\_length, window\_size, file)  
    
  1. taglist can be figured out by Function get\_bed\_coords. Then we define window\_size to count how many tags located in different window.  
    
  2. Return -> bed\_vals (Dictionary)  
  bed\_vals[current\_window\_start] = tag\_count\_in\_current\_window  
    
  3. output file (.graph file) saves the log which records the tag count in each window. The format is: chrom, window\_start, window\_end, tag\_count
* def make\_graph\_file(tagfile, chrom, chrom\_length, window\_size, fragment\_size, outfile)  
    
  1. Runs get\_bed\_coords() and Generate\_window\_and\_count\_tags(). The output file (.graph file) saves the log which records the tag count in each window (each window is identified by its start and end positions).

# SeparateByChrom.py

* The script is in:  
  */home/data/SICER1.1/SICER/lib*
* def separateByChrom(chroms, file, extension)   
    
  The idea of this function is, if in the input file, there are tags for a particular chrom\_1, it will generate a temp file called chrom\_1.extension (.bed file) which saves all the tag item that contains chrom\_1 (copy the line of that tag item).  
    
  So if file (input file, .bed file) have mixed with tags from different chromosomes, this script can help to separate them all.
* def combineAllGraphFiles(chroms, extension, final\_out)  
    
  Combine all the temp files generated by each chromosome and save into final\_out.extension.
* def cleanup(chroms, extension)  
    
  Clean all the tmp files.

# run-make-graph-by-chrom.py

* The script is in:  
  */home/data/SICER1.1/SICER/src*
* def makeGraphFile(chroms, chrom\_lengths, window, fragment\_size)  
    
  1. If we specify a species, we usually have  
  chroms = GenomeData.species\_chroms[species]  
  chrom\_lengths = GenomeData.species\_chrom\_lengths[species]  
    
  2. It will call Function make\_graph\_file. make\_graph\_file(). We already use SeparateByChrom.separateByChrom() to generate a list of chrom\_x.bed. Now for each bed file and the corresponding chromosome, we will generate a list of chrom\_x.graph files to count the tag in windows.
* def main(argv) -> get the details of argv by typing help  
    
  1. Basically, input is one .bed file, and output in one .graph file. The .graph file consists of counts of tags for all the chromosomes for this species.  
    
  2. The input .bed file has format: chrom, start, end, name, score, strand  
  The output .graph file has format: chrom, window\_start, window\_end, tag\_count

# normalize.py

* The script is in:  
  */home/data/SICER1.1/SICER/src*
* def total\_counts(file, column) -> return total counts in file  
    
  1. For .graph file, column = 3 (tag\_count is at index = 3)
* def normalize\_tag\_count(input\_file, ColumnIndex, total, output\_file)  
    
  1. Normalize the input\_file and the output is written in output\_file
* def main(argv) -> get the details of argv by typing help  
    
  1. Normalize the input\_file and output it.

# remove\_redundant\_reads.py

* The script is in:  
  */home/data/SICER1.1/SICER/src*
* def remove\_redundant\_1chrom\_single\_strand\_sorted(infile, outfile, cutoff)  
    
  1. infile can only contain reads from one chromosome and only one kind of strands (+/-). file must be pre-sorted by column2 (start), then by column3 (end). This will be done in Function strand\_broken\_remove().  
    
  2. Cutoff determines how many reads with exactly the same start and end will be retained.   
    
  3. Those reads that not considered as redundant will be written in outfile.  
    
  4. Return a tuple (total, retained)
* def strand\_broken\_remove(chrom, cutoff)  
    
  1. Before using this function, the BED file which contains all the reads in different chromosome have been separated by function seperateByChrome() (chrom.bed1).   
    
  2. For the specified chrom, generates plus.bed1 file which contains all the “+” strand read and sort by start and end; and also minus.bed1 file which contains all the “-” strand reads and sort by start and end.   
    
  3. Call Function remove\_redundant\_1chrom\_single\_strand\_sorted() to generate the redundancy removed files plus\_removed.bed1 and minus\_removed.bed1. Combine these two files and output in the file chrom.bed2.  
    
  4. Remove the temple files (plus.bed1, plus\_removed.bed1, minus.bed1, minus\_removed.bed1).
* def main(argv)  
    
  Input the raw BED file, call function SeparateByChrom.separateByChrom() to separate into chrom.bed1, and use function strand\_broken\_remove() to generate the chrom.bed2 which removes the redundant reads. Finally call function SeparateByChrom.combineAllGraphFiles() to combine all chrom.bed2 and generate output.bed2.

# get\_total\_counts.py

* The script is in   
  */home/data/SICER1.1/SICER/lib*
* def get\_tag\_counts\_by\_chrom(tag\_bed\_file, chrom)  
    
  1. Get counts of how many reads of a specific chrom in the bed file.
* def get\_total\_tag\_counts(tag\_bed\_file)  
    
  1. Get total counts of how many reads for all chrom in the bed file.
* def get\_total\_tag\_counts\_bed\_graph(summary\_graph\_file, bed\_val={}, threshold = 0):  
    
  1. Providing either the summary\_graph\_file (.graph, format is chrom window\_start, window\_end, tag\_count) or bed\_val (dictionary, format is bed\_vals[window\_start] = tag\_count), count the total tags.

# Background\_island\_probscore\_statistics.py

* The script is in:  
  */home/data/SICER1.1/SICER/lib*
* class Background\_island\_probscore\_statistics:
* def \_\_init\_\_(self, total\_tags, windowSize, gapSize, window\_pvalue, genomeLength, bin\_size):  
    
  1. self.poisson\_value -> list   
  1) It records all the poisson prob for the number of reads occurred in one window.   
  Self.poisson\_value[index = reads\_in\_window] = prob  
    
  2) The length is self.max.index = max(500, 2\*self.average). self.average is figured out by counting how many reads can be distributed in one window by averaging all the read counts in the effective genome length.  
    
  2. self.window\_score -> list  
  1) The equation to figure out score: s = -log P  
    
  2) Only look at enrichment, so if index < self.average, we assign score = 0  
    
  3) self.window\_scaled\_score = self.window\_score / bin\_size This list is used to do the numerical integration (bin\_size makes every step for integration smaller)  
    
  3. self.min\_tags\_in\_window  
  1) The equation to figure out l0:   
    
  2) Here window\_pvalue = p0  
    
  3) We are not able to sum up to the infinity item, the max = self.max\_index  
    
  4. self.gap\_contribution  
  1) 2) self.gap\_contribution = gap factor G  
    
  5. self.boundary\_contribution  
  1) self.boundary\_contribution = tg+1  
  6. self.island\_expectation -> list  
  1)   
    
  2) This is the list of LM(s) (Expected number of islands have score of s in the total genome length). Function background\_island\_expectation() will use recursive equation defined above to figure out all the terms for s.  
    
  3) The score here is the scaled\_score = score / bin\_size. This is a strategy to shorten the integration step so that increase the accuracy.
* def background\_island\_expectation(self, scaled\_score)  
  1) It will assign the values of list self.island\_expectation[] for all terms below the scaled\_score.  
    
  2) Return the expectation for scaled\_score
* def generate\_cumulative\_dist(self, outfile=""):  
  1) define list self.cumulative[x], where each value of index x means the sum of expectation above the scaled\_score x.  
    
  2) if outfile is not “”, it will write down for each score, the window\_number\_expectations and accumulative numbers to outfile.
* def find\_island\_threshold(self, e\_value\_threshold):  
  1) . The aim of this function is to figure out sT.  
    
  2) return the score\_threshold.
* def find\_asymptotics\_exponent(self, xacc=.00001):  
  1) The asymptotics for the M(s):  
  2) Return the solution for β.

# find\_islands\_in\_pr.py

* The script is in:  
  */home/data/SICER1.1/SICER/src*
* def combine\_proximal\_islands(islands, gap, window\_size\_buffer=3):  
  1) islands are a list of bed graph objects with the format: (chrom, window\_start, window\_end, score). In this case, the input islands are filtered\_bed\_val, which are lists of eligible windows.  
    
  2) Combine the windows to form islands (as long as the distance is smaller than gap + buffer)  
    
  3) return the Final\_islands with each item having the format: (chrom, island\_start, island\_start, total\_score)
* def find\_region\_above\_threshold(island\_list, islands\_minimum\_tags):  
  1) In this case, island\_list.value = socre; island\_minimum\_tags = score\_threshold  
    
  2) return filtered\_islands (score > score\_threshold). The format is (chrom, window\_start, window\_end, score).
* def main(argv):  
  1) window\_pvalue = 0.2; bin\_size = 0.001  
    
  2) call Class Background\_island\_probscore\_statistics \_\_init\_\_ to define background  
    
  3) Input summary graph files (.graph).   
  This file has the format: window\_start, window\_end, tag\_counts  
    
  4) In this case, we call class BED \_\_init\_\_ to define dictionary bed\_val,  
  bed\_val[chrom1]=[<window\_item1>, <window\_item2>,…]  
  Where window\_item1.value=tag\_counts in this window  
    
  5) Judge if the tag\_counts in window is larger than min\_tags\_in\_window. If smaller, score = -1, otherwise, score = -log p. Then replace the window\_item1.value from tag\_counts to the score.  
  Meanwhile, define a new dictionary filtered\_bed\_val, attach the item that score > 0 (enrichment) to this dictionary.  
    
  6) Use function Background\_island\_probscore\_statistics. find\_island\_threshold to find out the score\_threshold  
    
  7) use function combine\_proximal\_islands() and function find\_region\_above\_threshold() to get the satisfied islands, count the number of total islands and output them.  
    
  8) The output file format: (chrom, island\_start, island\_start, total\_score)

# associate\_tags\_with\_chip\_and\_control\_w\_fc\_q.py

* The script is in:  
  */home/data/SICER1.1/SICER/src*
* def main(argv):  
  1) The purpose of this script.   
  We already have islands files, which has the format (chrom, window\_start, window\_end, score). That’s not enough if we just count all the chip reads which locate within the island region (This applies to the case without control file). We want to filter and remove some islands if the chip reads are not much more significant than control reads in that island region.  
    
  2) Input the islands file, and generate dictionary islands. Islands[chrom] = [island1, island2, …]  
    
  3) Separate the chip bed file into chrom.bed1; control bed file into chrom.bed2   
    
  4) Define dictionary island\_chip\_readcount. Island\_chip\_readcount[chrom] = island\_chip\_readcount\_list. Each index in this list corresponds to an index of island in islands[chrom]. The value of the list is the read counts in that island. So island\_chip\_readcount\_list[index] = chip\_readcount\_in\_island\_of\_index. In the same way, we define dictionary island\_control\_readcount.  
    
  5) define scaling\_factor = chip\_library\_size / control\_library\_size. Where chip\_library\_size and control\_library\_size are the total bed reads (including both in or not in islands). Also define totalchip (located in island) and totalcontrol (located in island).  
    
  6) define list result\_list, and its element item\_dic. Each island term corresponds to one dictionary item\_dic. Item\_dic records (chrom, island\_start, island\_end, chip\_readcount, control\_readcount, pvalue, fc)  
    
  pvalue is figured out by the equation: . Where c is the scaling\_factor. P is poisson function, ns is chip readcount, nc is control readcount.  
    
  fc is the fold change of chip readcount to expected readcount figured out by control countread \* scaling\_factor, for a particular factor.  
    
  7) Also for each island, defines alpha, which will be further use in filter\_islands\_by\_significance.py to filter islands (using FDR method).   
    
  8) Output every item in read\_list as well as alpha to an output file.   
  The output is in the format:   
  (chrom, island\_start, island\_end, chip\_readcount, control\_readcount, pvalue, fc, alpha)

# filter\_islands\_by\_significance.py

* The script is in:  
  */home/data/SICER1.1/SICER/src*
* Input islandsummary files are the output of associate\_tags\_with\_chip\_and\_control\_w\_fc\_q.py
* Retain those island terms that satisfy alpha <= significance (FDR), and output the islandsummary in a file.

# convert\_summary\_to\_bed.py

* The script is in:  
  */home/data/SICER1.1/SICER/utility*
* The purpose of this script is. We know islandsummary file has a lot fields in one line. Now we want to convert to BED file and only retains (chrom, island\_start, island\_end, chip\_readcount)

# filter\_raw\_tags\_by\_islands.py

* The script is in:  
  */home/data/SICER1.1/SICER/utility*
* def tag\_position(sline, fragment\_size):  
  1) for one read line sline, extract the read position information and according to its fragment\_size, get the fragment center position.
* def filter\_tags\_by\_islands(chroms, islands, fragment\_size):  
  1) Judge if the read in chrom.bed1 locate in the islands defined in islands. If so, write down into chrom\_filtered.bed1
* def main(argv):  
  1) The idea of this script is input the bedfile and islandbedfile, filter and retain those bed items that locates in islands.   
    
  2) Use function SeparateByChrom.separateByChrom() to separate the input bedfiles into chrom.bed1.  
    
  3) Use function filter\_tags\_by\_islands() to filter all the reads that locate in islands, and write down into chrom\_filtered.bed1.  
    
  4) Combine chrom\_filtered.bed1 into final\_output\_file.  
    
  5) Remove .bed1 and \_filtered.bed1

# UCSC.py

* The script is in:  
  */home/data/SICER1.1/SICER/lib*
* Define Class UCSC, which is for keeping known gene information  
    
  Class UCSC:  
    
  def \_\_init\_\_(self, name, chrom, strand, txStart, txEnd, cdsStart, cdsEnd, exonCount, exonStarts, exonEnds)
* Define Class UCSC\_lite, which only partial information stored.  
    
  Class UCSC\_lite:  
    
  def \_\_init\_\_(self, name, chrom, strand, txStart, txEnd)
* Define Class KnownGenes, which is to read in UCSC known genes files and get all information.  
    
  Class KnownGenes:  
    
  def \_\_init\_\_(self, file=None):  
    
  1) The input UCSC known gene files are in format (.UCSC), and each line contains a Class UCSC instance (name, chrom, strand, txStart, txEnd, cdsStart, cdsEnd, exonCount, exonStarts, exonEnds).  
    
  2) define Dictionary self.gene\_coords. self.gene\_coords[chrom1] = [UCSC type1, UCSC type2, …]. Each element is one known gene item from the input .UCSC file.  
    
  def getPromoters(self, upstream, downstream):  
    
  1) define the promoters which are in the region [prom\_start, prom\_end]:   
  if “+”, [txStart – upstream, txStart + downstream];  
  if “-”, [txEnd - downstream, txEnd + upstream]  
    
  2) define Dictionary self.prom\_coords. self.prom\_coords[chrom1] = [UCSC\_lite type1, UCSC\_lite type2]. It goes through all the known genes elements in self.gene\_coords, and extract the info [name, chrom, strand, prom\_start, prom\_end]  
    
  def getTSS(self):  
    
  1) return a Dictionary of TSS positions keyed by gene name.  
    
  2) if “+”, self.TSS[g.name] = g.txStart;  
   if “-”, self.TSS[g.end] = g.txEnd  
    
  def getGenebodys(self, downstream):  
    
  1) define the genebody which are in the region [GB\_start, GB\_end]:  
  if “+”, [txStart + downstream, txEnd];  
  if “-”, [txStart. txEmd - downstream]  
    
  2) define Dictionary self.genebody. self.genebody[chrom1] = [UCSC\_lite type1, UCSC\_lite type2]. It goes through all the known genes elements in self.gene\_coords, and extract the info [name, chrom, strand, GB\_start, GB\_end]  
    
  def getPromotergenebodys(self, upstream):  
    
  1) define the promoter + genebody which are in the region [pg\_start, pg\_end]:  
  if “+”, [txStart - uptream, txEnd];  
  if “-”, [txStart. txEmd + upstream]  
    
  2) define Dictionary self.pg. self.pg[chrom1] = [UCSC\_lite type1, UCSC\_lite type2]. It goes through all the known genes elements in self.gene\_coords, and extract the info [name, chrom, strand, pg\_start, pg\_end]

# GenerateProfileAroundLocations.py

* The script is in:  
  */home/data/SICER1.1/SICER/extra*
* The idea of this script is to generate profiles data at either TSS or TES. Data format is (profile position, plus\_norm\_score, minus\_norm\_score).
* def breakUpStrands(bed\_list):  
    
  1) here bed\_list contains a list of bed2 format data (strand, start) (Notice here the start for ‘-’ strand already chooses the real start, which is the end position in BED file). This function returns (plus\_starts, minus\_starts), where plus\_starts = [start\_+1, start\_+2, …] (the start of all reads with “+”); minus\_starts = [start\_-1, …] (the start of all reads with “-”).
* def getProfileNearPosition(position, orientation, upstream\_length, downstream\_length, resolution, window\_size, tag\_positions):  
    
  1) The idea of this function is to get the profiles around the position within region length = upstream\_length + downstream\_length.   
    
  We have a list of tag\_positions, which is generated by function breakUpStrands(bed\_list) and following a shift of half fragment length considering its strand orientation (plus\_starts + pshift, minus\_starts - mshift).  
    
  We count the scores in a window size, or in other words count how many tag\_positions locate in that window. Every time the window will move forward the resolution size and then count the tags again. numPoints = length / resolution, which defines how many profiles points we have.  
    
  return list scores[index] = count. Index are numPoints.   
  If gene orientation is “+”, index from 0 to numPoints, profile position from [position – upstream\_length, position + downstream\_length];  
  If gene orientation is “-”, index from 0 to numPoints, profile position from [position + upstream\_length, position – downstream\_length]
* def getTSSProfile(coords, upstream\_length, downstream\_length, resolution, window\_size, pshift, mshift, bed\_vals):  
    
  1) Dictionary coords[chrom1] = [UCSC type1, UCSC type2, …]. Dictionary bed\_vals[chrom1] = [BED2 typ1, BED2 typ2]  
    
  2) Call function breakUpStrands(bed\_vals[chrom]) to generate (plus\_starts, minus\_starts), and shift with the pshift and mshift, which then point to the center of the fragment.  
    
  3) Going over all the known genes in Dictionary coords, it counts the tags in each window for both “+” reads and “-” reads for every gene (by calling function getProfileNearPosition). And then sum these values together over all genes. (For gene with “+”, the promoters are in txStart. For genes with “-”, the promoters are in txEnd). And return (plus\_scores, minus\_scores)  
    
  4) All of these are done on one single chrom. In main function, it will go over all the chroms and calling this function each time.
* def getTESProfile(coords, upstream\_length, downstream\_length, resolution, window\_size, pshift, mshift, bed\_vals):  
    
  1) Being very like function getTSSProfile. Only difference is to draw the profiles on the TES (transcription end sites).
* def output(upstream\_length, resolution, plus\_scores, minus\_scores, normalization, outfilename):  
    
  1) Output the profiles data (profile position, plus\_norm\_score, minus\_norm\_score). Both the scores are after normalization.
* def main(argv):

# variableStep.sh

* The script is in:  
  */home/data/SICER1.1/SICER/src*
* The idea of this script is to convert summary .graph file to .wig file. Also output chr.list
* Input argument:  
    
  #$1 Input file name

S=$1

#$2 Output file name

R=$2

#$3 Track label

#$4 window size

SPAN=$4

# bam2wig.sh

* The script is in:  
  */home/data/SICER1.1/SICER/extra/tools/wiggle*

The principle of this script is:  
1. Use bamToBed command in BEDtools to convert bam file to bed file.

1. Use another scripit **bedtowig.sh** (need to specify window size, fragment size, and species) to convert bed file to wig file.
2. Remove temporary bed file.  
     
   Note: Both bedtools and bamToBed source code is in /usr/bin. bamToBed is actually a script that call bamToBed command in BEDtools.

* Input argument:  
    
  SAMPLEDIR=$1

SAMPLE=$2 (file name before suffix .bam)

WINDOW\_SIZE=$3

FRAGMENT\_SIZE=$4

SPECIES=$5

* Note: Before using this script, use cd to go to the .bam directory (otherwise this script cannot find the .bam data file). My own version which is in my own folder has modified so we don’t have to be local in that directory.

# bed2wig.sh

* The script is in:  
  */home/data/SICER1.1/SICER/extra/tools/wiggle*
* The principle of this script is:  
  1. Use Python script run-make-graph-file-by-chrom.py to generate the .graph file  
    
  2. Use Python script normalize.py to normalize .graph file  
    
  3. Use Shell script variableStep.sh to convert .graph file to .wig file.
* Input argument:  
    
  SAMPLEDIR=$1

SAMPLE=$2

SAMPLEBED=$SAMPLE.bed

WINDOW\_SIZE=$3

FRAGMENT\_SIZE=$4

SPECIES=$5

SUMMARY=$SAMPLE-W$WINDOW\_SIZE.graph

SUMMARYWIG=$SAMPLE-W$WINDOW\_SIZE.wig

NORMALIZEDSUMMARY=$SAMPLE-W$WINDOW\_SIZE-normalized.graph

NORMALIZEDSUMMARYWIG=$SAMPLE-W$WINDOW\_SIZE-RPKM.wig

# SICER.sh

* The script is in:  
  */home/data/SICER1.1/SICER*
* 11 input arguments:  
  ["InputDir"] ["bed file"] ["control file"] ["OutputDir"] ["Species"] ["redundancy threshold"] ["window size (bp)"] ["fragment size"] ["effective genome fraction"] ["gap size (bp)"] ["FDR"]   
    
  Note: gap size must be multiple of window size.
* The principle of this script is:  
  1) Run remove\_redundant\_reads.py script to remove redundant reads both in chip-seq and control BED files.  
    
  The file name will be -n-removed.bed, where n is the redundancy threshold.  
    
  2) Specify the window\_size and fragment\_size and use run-make-graph-file-by-chrom.py to generate the graph summary file (.graph).  
    
  The file name will be –W(Window\_size).graph  
    
  3) normalize it by running normalize.py and convert to WIG file by running variableStep.sh, which can be uploaded to Genome Browser.  
    
  The file name will be –W(Window\_size)-normalized.wig  
    
  4) run find\_islands\_in\_pr.py to find the candidate islands.  
    
  The file name will be –W(Window\_size)-G(Gap\_size)-islands.scoreisland  
  Each line in this file has the format:  
  chrom, island\_start, island\_end, score.  
    
  Use function Background\_island\_probscore\_statistics. find\_island\_threshold to find out the score\_threshold (determined by e value). And output all the islands with scores > score\_threshold  
    
  5) Calculate significance of candidate islands using the control library (by running associate\_tags\_with\_chip\_and\_control\_w\_fc\_q.py).  
    
  The file name will be –W(Window\_size)-G(Gap\_size)-islands-summary  
    
  Each line in this file have the format:  
  (chrom, island\_start, island\_end, chip\_readcount, control\_readcount, pvalue, fc, FDR)  
    
  6) Identify significant islands using FDR criterion (by running filter\_islands\_by\_significance.py), and convert island summary to island bed file of format chr start end ChIP-read-count (by running convert\_summary\_to\_bed.py).  
    
  After running filter\_islands\_by\_significance.py  
  The file name will be –W(Window\_size)-G(Gap\_size)-islands-summary-FDR(FDRNumber), which has the same format with above.  
    
  After running convert\_summary\_to\_bed.py  
  The file name will be –W(Window\_size)-G(Gap\_size)-FDR(FDRNumber)-island.bed  
    
  7) Filter reads (in BED file) with identified significant islands (by running filter\_raw\_tags\_by\_islands.py) and retain those reads that locate in islands.  
    
  The file name will be –W(Window\_size)-G(Gap\_size)-FDR(FDRNumber)-islandfiltered.bed  
    
  8) Specify the window\_size and fragment\_size and use run-make-graph-file-by-chrom.py to generate the graph summary file (.graph). These are for only island enriched BED reads,  
    
  9) normalize it by running normalize.py and convert to WIG file by running variableStep.sh, which can be uploaded to Genome Browser. These are for only island enriched BED reads.  
    
  The file name will be The file name will be –W(Window\_size)-G(Gap\_size)-FDR(FDRNumber)-islandfiltered-normalized.wig