Table of Contents for Supplementary Information

Figure_S1. An alignment of the 6868 CEN180 repeats, where each sequence is a row and each site is represented as a column. Sequences are drawn in the cluster order and dashed lines separate clusters. Only segregating sites with respect to the consensus sequence are colored with green, blue, purple, red and grey representing A, G, C, T and – respectively.

Figure_S2. Proportion of ChIP-seq reads assigned to a single cluster or multiple cluster by K-mer size.

Figure_S3. Read length distribution of CENH3 ChIP DNA fragments. A) Bioanalyzer traces for both replicates for each CENH3 ChIP library. B) Density plot of merged paired-end reads for each CENH3 ChIP-seq library. Vertical dashed lines correspond to median values.

Figure_S4. Colocalization of cluster 6 and chromosome 1 specific BAC-signals

Figure_S5. Fraction of total ChIP signal associated with CEN180 sequences for the CENH3 ChIP and their respective antibody-specificity negative controls.

Figure_S6. Pearson correlation analysis between CENH3 ChIP experiments.

Figure_S7. Confirmation of antibody specificity by immunostaining. Antibodies against CENH3 of *A. thaliana*, *L. oreraceum* and *Z. mays* were tested.

Table_S1. Table with number of K-mers and signature Kmers by cluster

Table_S2. Number of repeats identified on each chromosome of the TAIR10 reference assembly

Table_S3. *A. thaliana* centromeres built on *Z. mays* CENH3 mis-segregate in crosses.

Table_S4. Quality assessment of CENH3 ChIP experiments based on FRiP metric i.e. fraction of reads in peaks.

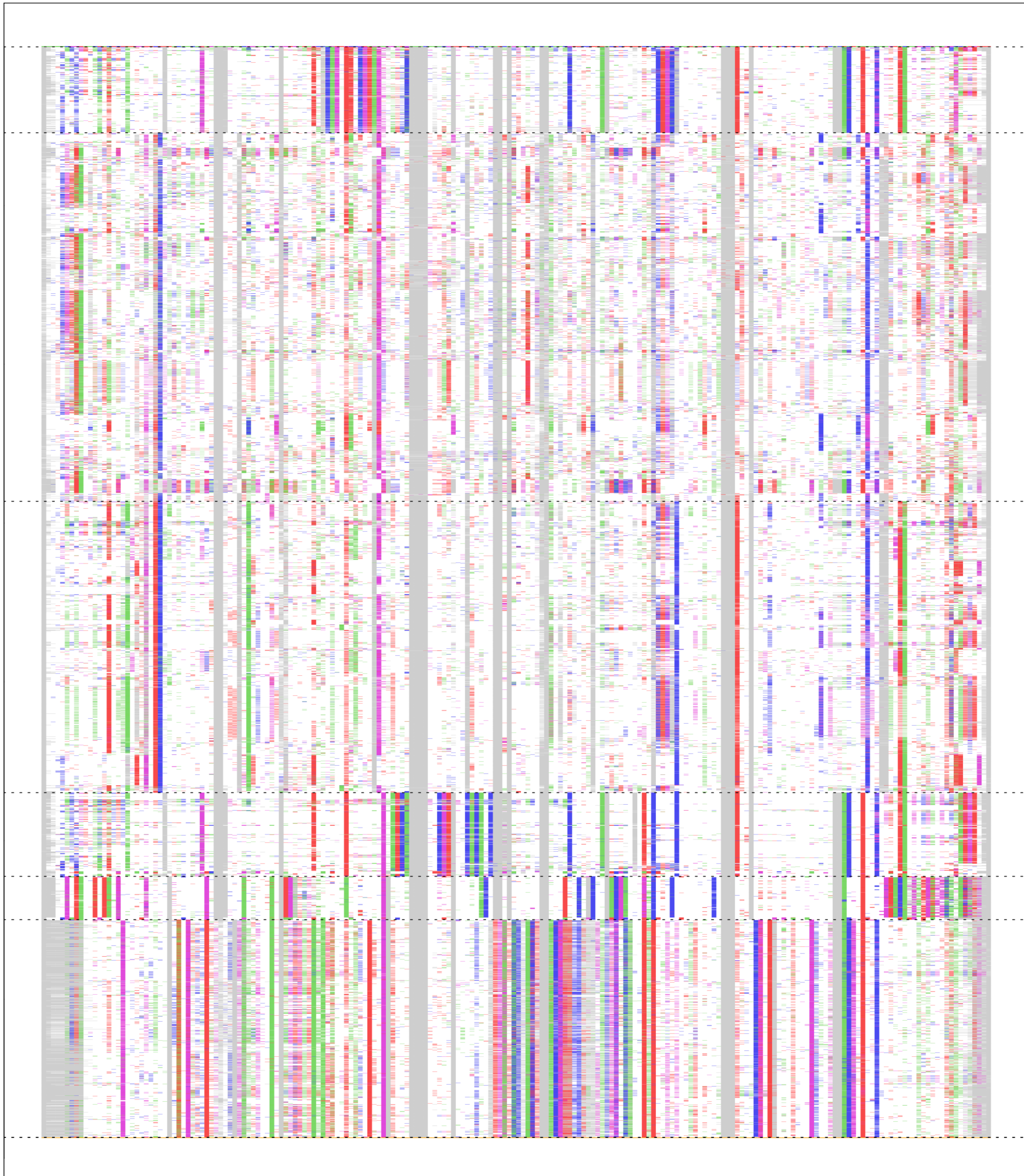
Python scripts

List

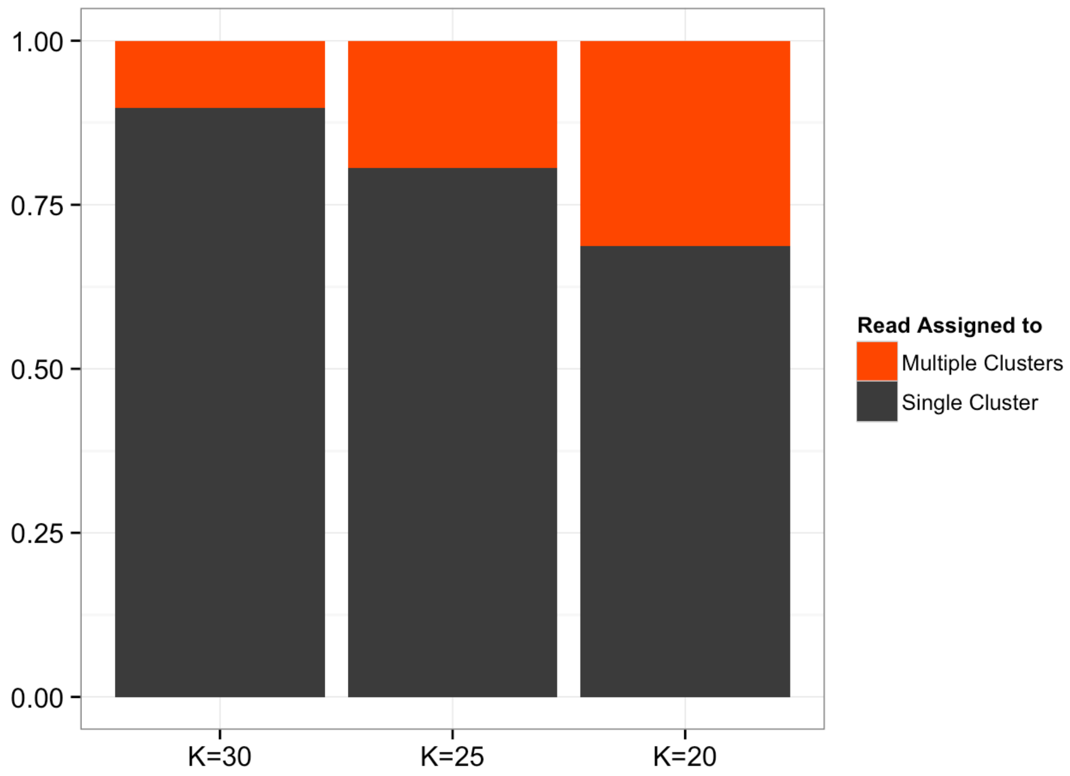
Usage

Script code

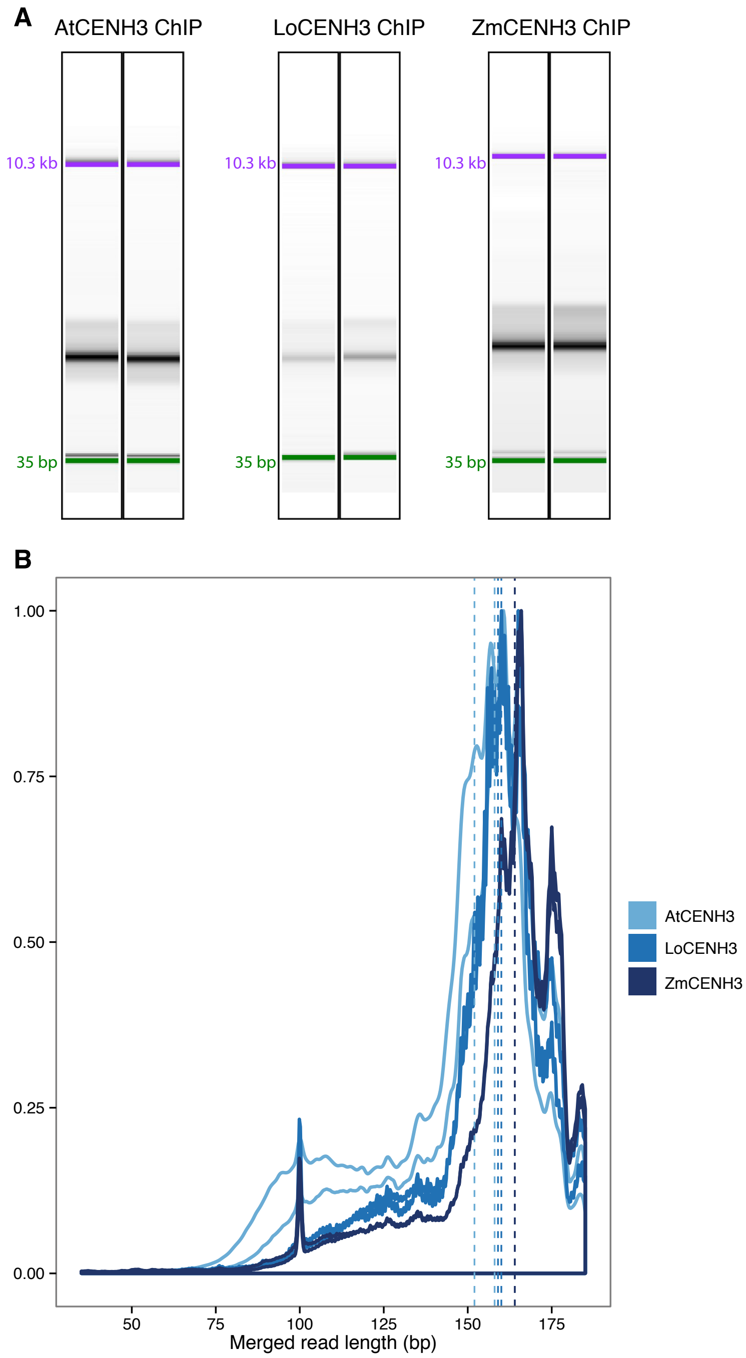
Figure_S1. Segregating sites by cluster



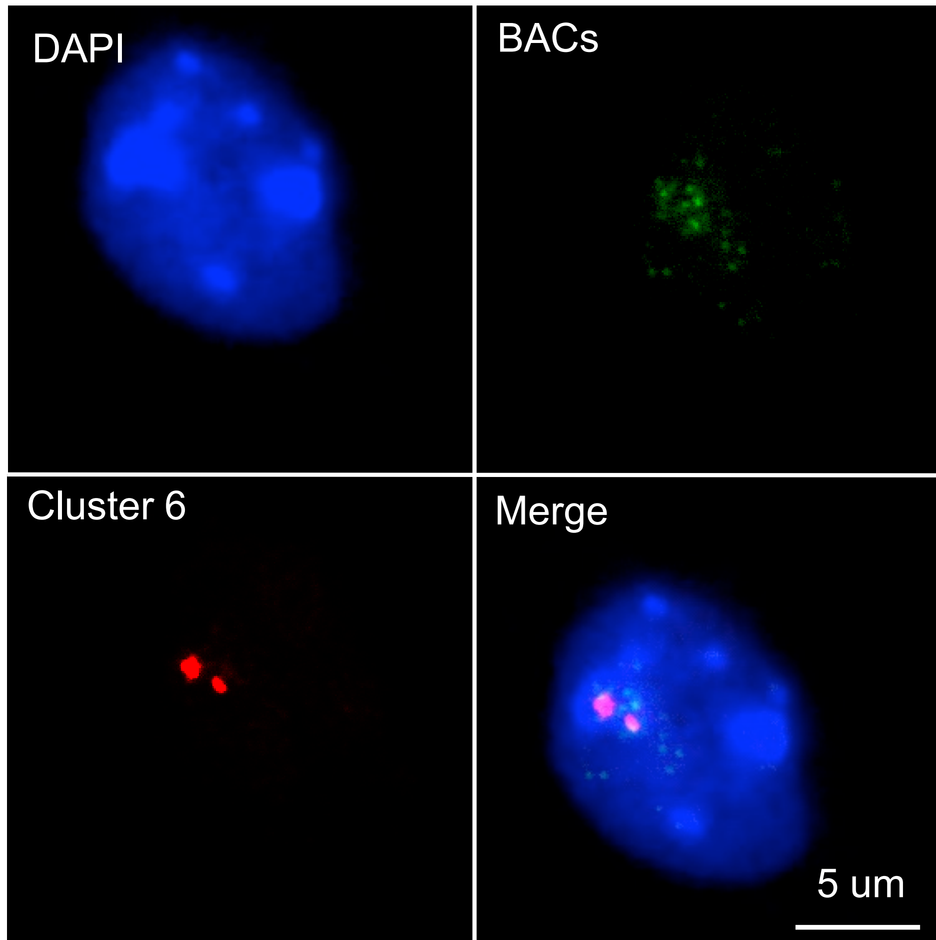
Figure_S2. Proportion of ChIP-seq reads assigned to a single cluster multiple cluster by K-mer size.



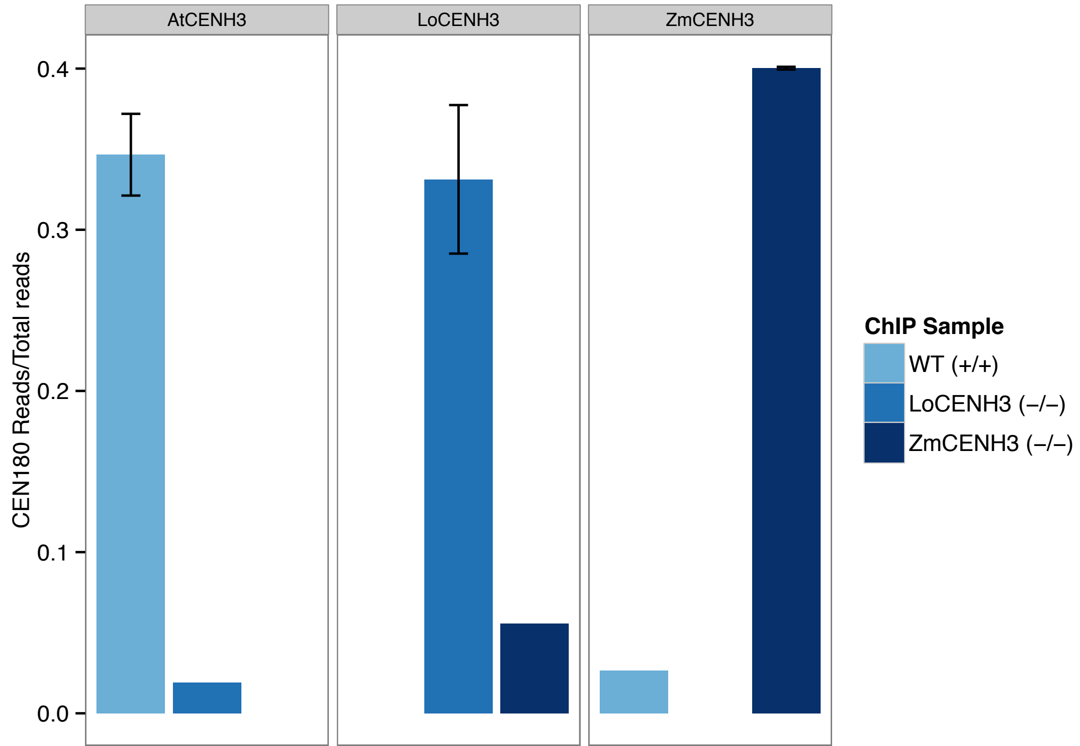
Figure_S3. Read length distribution of CENH3 ChIP DNA fragments.



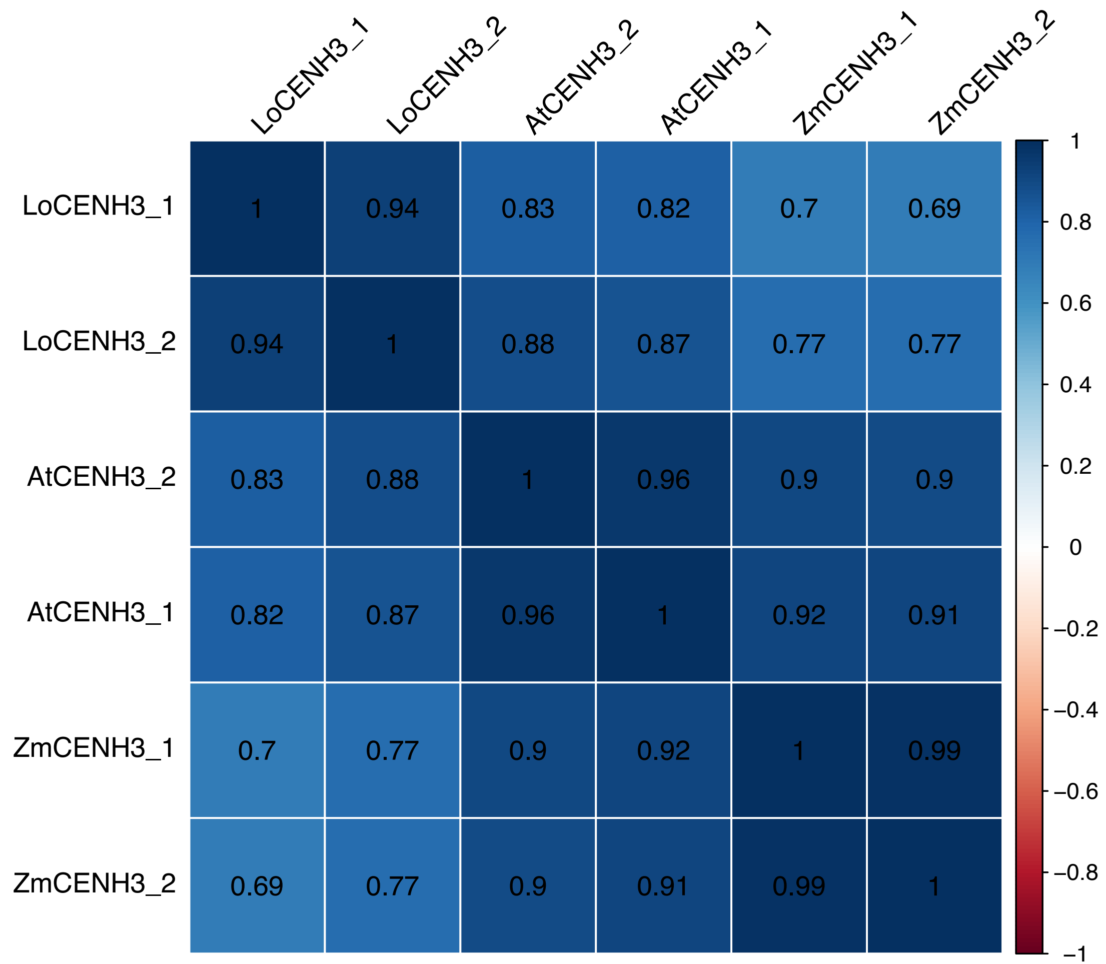
Figure_S4. Colocalization of Cluster 6 and chromosome 1 specific BACs.



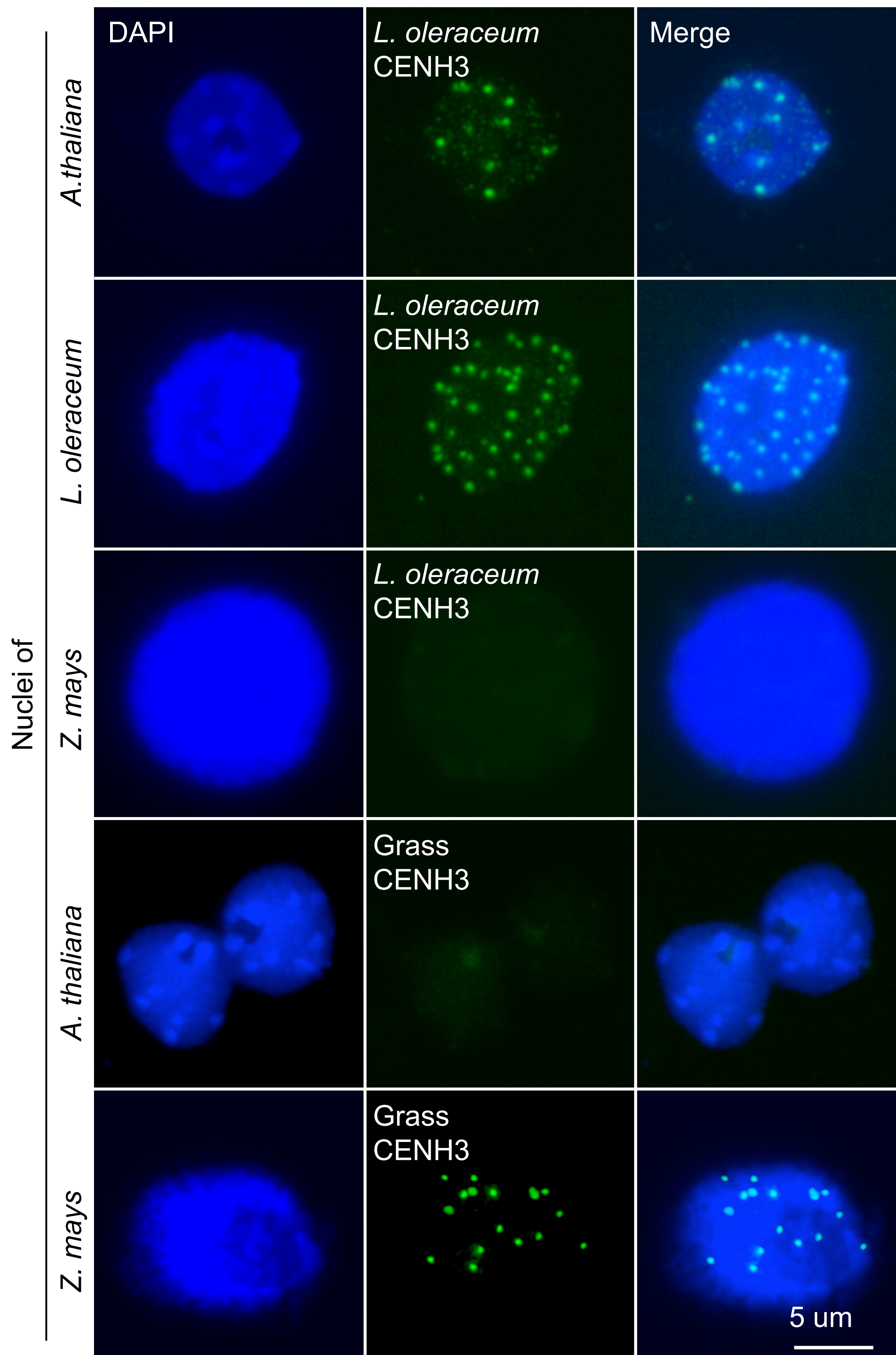
Figure_S5. Fraction of total ChIP signal associated with CEN180 sequences for CENH3 ChIPs and their respective antibody-specificity negative controls.



Figure_S6. Pearson correlation analysis between CENH3 ChIP experiments.



Figure_S7. Confirmation of antibody specificity by immunostaining.



Table_*S*1. Table with number of K-mers and signature Kmers by cluster

	K=30		K=25		K=20	
	Kmers	Signature Kmers	Kmers	Signature Kmers	Kmers	Signature Kmers
Cluster 1	138	7	161	18	176	22
Cluster 2	206	166	249	182	296	182
Cluster 3	387	304	447	335	477	329
Cluster 4	153	21	161	16	169	14
Cluster 5	99	99	127	127	144	138
Cluster 6	350	350	434	434	480	475

Table_*S*2. Number of repeats identified on each chromosome of the TAIR10 reference assembly

	No. of CEN180 repeats	Chromosome size (Mb)	CEN180 Repeat fraction	Median length of repeats (bp)
Chr1	575	30.4	0.32	170
Chr2	145	19.7	0.13	178
Chr3	81	23.4	0.06	177
Chr4	754	18.6	0.71	176
Chr5	995	26.9	0.65	178

Table_S3. *A. thaliana* centromeres built on *Z. mays* CENH3 mis-segregate in crosses.

T1 family	Normal seed %	Total Plants Analysed	Haploids (%)	Diploids (%)	Phenotypic Aneuploids (%)
3	32 (n = 535)	94	18 (19)	61 (65)	15 (16)
17	46 (n = 410)	15	1 (6)	7 (47)	7 (47)
29	44 (n = 519)	258	25 (10)	175 (68)	58 (22)
95	73 (n = 271)	62	0 (0)	54 (87)	8 (13)

Table_S4. Quality assessment of CENH3 ChIP experiments based on FRiP metric i.e. fraction of reads in peaks.

Expt. Name	Factor	Replicate	Peaks	FRiP
AtCENH3_1	AtCENH3	1	443	0.54
AtCENH3_2	AtCENH3	2	443	0.45
LoCENH3_1	LoCENH3	1	443	0.34
LoCENH3_2	LoCENH3	2	443	0.51
ZmCENH3_1	ZmCENH3	1	443	0.56
ZmCENH3_2	ZmCENH3	2	443	0.55

LIST OF SCRIPTS

#Script to parse a very specific LASTZ output format
PROTOCOLS/parse_lastz.py

#Script to refine MSA generated by Clustal Omega
PROTOCOLS/msa_refiner.py

#Script that generates signature Kmers for each cluster and uses that to partition fasta sequences into different clusters.
PROTOCOLS/Seq_2_Clstr.py

#Script to process fastq reads from Illumina
comailab/allprep-13.py

Scripts can also be accessed at
http://doi.org/10.5281/zenodo.160186
https://zenodo.org/record/180498

Brief description of Script use

Raw Read Processing.
1. Demultiplexing, adaptor and quality trimming of the raw reads was performed using the Allprep script (https://github.com/Comai-Lab/allprep).
2. Paired reads were merged using SeqPrep (https://github.com/jstjohn/SeqPrep) with parameters –q 30 (quality) and –L 35 (minimum merged pair length). Read pairs that did not merge were discarded.

Cluster Analysis.
3. LASTZ (http://www.bx.psu.edu/~rsharris/lastz/) was used to identify CEN180 repeats within the assembled TAIR10 genome and PacBio contigs. Several CEN180 sequences were tested as LASTZ queries and we chose to move forward with CEN2 as the query since it identified the maximum number of repeats and a majority of the adjacent repeats were 0-bp apart i.e. in tandem.

```
>CEN2
AAAAGCCTAAGTATTGTTTCCTTGTTAGAAGATACAAAGACAAAGACTCATATGGACTTCGGCTA
CACCATCAAAGCTTTGAGAAGCAAGAAGCTTGGTTAGTGTTTTGGAGTCAAATATGACTTGA
TGTCATGTGTATGATTGAGTATAACAACTTAAACCGCAACCGGATCTT
```

4. The LASTZ output was parsed using the python script **parse_lastz.py** to exclude sequences with >14 gapped positions and lengths outside the 165-185 bp. This step pruned 325 sequences resulting in a total of 7005 repeats (4547 and 2458 from PacBio contigs and TAIR10, respectively).
6. The 7005 repeats were aligned using Clustal omega with default parameters and the alignment was further refined by removing sequences that generated rare indels using the python script **msa_refiner.py**. The final multiple sequence alignment included 6868 sequences and was 204 bp in length.
7. Clustering was performed on this alignment using find.best() function in phyclust, an R package developed for exploring population structure of DNA sequence data using a phylogenetic approach. We evaluated a range of clusters, 2-10, and using the clustergram visualization approach and found that 6 clusters was an optimal solution (data not shown).
8. We assigned ChIPseq reads to clusters using the python script **Seq_2_Clstr.py** which generated sets of signature K-mers for each cluster, i.e. high-frequency K-mers that are unique to that cluster. We arbitrarily required the K-mer to be shared by at least 100 repeats within a cluster and tested K-mers in a range of sizes: 20, 25 and 30-bp (Table S1). If a read contained signature K-mers we annotated it as a CEN180 sequence. If K-mers from a read were restricted to a single cluster we assigned the read to that cluster. We found that the specificity of cluster assignment increased with K-mer size (Fig. S2) and therefore chose 30-bp as the default K-mer length.

ChIPseq read-mapping and peak-calling.
9. Merged paired-end reads were mapped to the TAIR10 reference using the aln and samse algorithms of BWA and upto 10 alignments per read were saved.

10. We performed peak calling using MACS2 with the additional parameters '--nomodel --extsize 165 --keep-dup all -B' and used the appropriate CENH3 ChIP negative controls as input. 11. We evaluated the quality of the ChIP experiments using the modENCODE recommended metric of FRiP i.e. fraction of reads in peaks and calculated correlation between CENH3 ChIP experiments using the R/Bioconductor DiffBind package (Table S4, Fig. S5).

SCRIPTS

#Note: each script starts with "#SCRIPT/"

```
#SCRIPT/shamoni-cen-evo-devo-81d1326/PROTOCOLS/msa_refiner.py
#! /usr/bin/env python

import sys, argparse
from Bio import AlignIO
from Bio.Align import MultipleSeqAlignment
from Bio.SeqRecord import SeqRecord

def del_all_gaps(alignment):
	rows = len(alignment)
	n = len(alignment[0])
	n_at_start = n
	i = 0
	while i < n:
		if alignment[:,i].count("-") == rows:
			if i == 0:
				alignment = alignment[:,1:]
			elif i+1 == n:
				alignment = alignment[:,:i]
			else:
				alignment = alignment[:,:i] + alignment[:,i+1:]
			n -= 1
		else:
			i += 1
	return alignment

def del_rows(alignment,t):
	rows = len(alignment)
	pos = len(alignment[0])
	gaps = []
	for i in range(pos):
		if rows - alignment[:,i].count("-") < t:
			gaps.append(i)

	to_remove = set()
	for c in gaps:
		for r in range(rows):
			if alignment[r,c] != "-":
				to_remove.add(alignment[r].id)

	filtered_records = []
	for record in alignment:
		if record.id not in to_remove:
```

```
                    filtered_records.append(SeqRecord(record.seq, record.id,
description=""))

        filtered_align = MultipleSeqAlignment(filtered_records)
        return filtered_align


def main():
        parser = argparse.ArgumentParser(description='This script adjusts a
multiple sequence alignment \
                by removing sequences that insert rare gaps')
        parser.add_argument('input', action="store", type=str, help="Filename of .aln
file in current folder")
        parser.add_argument('-t', '--threshold', action="store", type=int, default=20,
help="Threshold to call rare insertions")
        args = parser.parse_args()

        input_file = args.input
        f = open(input_file)
        alignment = AlignIO.read(f, "fasta")
        print >>sys.stderr, "Recovered msa of lengh %d with %d aligned sequences"
%(len(alignment[0]),len(alignment))

        t = args.threshold
        alignment = del_rows(alignment,t)
        print >>sys.stderr, "Kept %d aligned sequences after deleting sequences with
insertions shared by < %d other sequences" %(len(alignment),t)

        alignment = del_all_gaps(alignment)
        print >>sys.stderr, "Trimmed msa has length %d with %d aligned sequences"
%(len(alignment[0]),len(alignment))

        filename = input_file.split(".")[0]
        AlignIO.write(alignment, filename+".aln2", "fasta")

if __name__ == '__main__':
                main()

#SCRIPT/shamoni-cen-evo-devo-81d1326/PROTOCOLS/parse_lastz.py
#! /usr/bin/env python

#-----------------------------------------------------------------------------
# This script parses a very specific output file format from LASTZ. The command
# for generating that file is the following:
# lastz input.fa[nameparse=darkspace] query.fa --coverage=90 \
```

```
# --
format=general:score,name1,strand1,size1,start1,end1,name2,strand2,identity,lengt
h1,align1 > out.csv
#
# The -o option generates a cleaned-up version of the same csv file that is more
amenable
# to analysis in R.
# The -s option generates a file where each line corresponds to target, query and the
difference between
# the ending position of the previous hit on the target sequence and the starting
position of current hit.
# In effect it is the spacing between the repeats.
# The -p option generates a file where each line corresponds to target, query and the
gaps inserted by
# LASTZ into the target sequence to get an alignment to the query
# The -r option generates a fasta file of target sequences after applying the following
filtering and processing steps:
# 1) Sequences that have gaps introduced by LASTZ greater than the threshold
number are removed.
# 2) The sequnces are ungapped and only sequences that have a length 165-180 bp
are kept.
# 3) If the target sequence matches the "-" strand of the query, the sequence is
reverse complemented
# and this is reflected by a "_R_" prefix added to their IDs.
#---------------------------------------------------------------------------

import sys, csv, argparse
from Bio import SeqIO
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_dna

def extract_fasta(records, gaps_upperlimit):
	records = filter(lambda x: x[11].count("-") < gaps_upperlimit, records)
	print >>sys.stderr, "Recovered %d records with less than %d gaps in target
sequence" %(len(records),gaps_upperlimit)
	ff = []
	for record in records:
		repeat = Seq(record[11], generic_dna)
		if len(repeat.ungap("-")) not in range(165,181):
			pass
		elif record[7] is "-":
				repeat = repeat.reverse_complement()
				ID = "_R_"+str(record[1])+":"+str(record[4])+"-
"+str(record[5])
```

```
					ff.append(SeqRecord(repeat.ungap("-"), id = ID, 
description = ""))
		else:
			ID = str(record[1])+":"+str(record[4])+"-"+str(record[5])
			ff.append(SeqRecord(repeat.ungap("-"), id = ID, description = 
""))
	return ff

def repeat_gaps(records):
	prev_contig = 0
	prev_query = 0
	gaps = []
	for record in records[1:]:
		this_contig = record[1] # establish which contig we are in
		this_query = record[6] # establish which contig we are in

		if (prev_contig != this_contig) or (prev_query != this_query): # if this is 
the start of a new contig or query then resetstop
			prev_contig = this_contig
			prev_query = this_query
			repeat = Seq(record[11], generic_dna)
			g = repeat.count('-')
			gaps.append([this_contig, this_query, g])
		else:
			repeat = Seq(record[11], generic_dna)
			g = repeat.count('-')
			gaps.append([this_contig, this_query, g])
	return gaps

def repeat_spacing(records):
	prev_contig = 0
	prev_query = 0
	spacing = []
	for record in records[1:]:
		this_contig = record[1] # establish which contig we are in
		this_query = record[6] # establish which contig we are in

		if (prev_contig != this_contig) or (prev_query != this_query): # if this is 
the start of a new contig or query then resetstop
			prev_contig = this_contig
			prev_query = this_query
			prev_stop = int(record[5]) + 1
		else:
			this_start = int(record[4])
			spacing.append([this_contig, this_query, this_start - prev_stop])
```

```
			prev_stop = int(record[5]) + 1 # reset the prev_stop to reflect
the end of current repeat
	return spacing

def collect_records(fp):
	records = []

	for line in fp:
		if line[0] == '#':
			header = [line[1:].strip().split('\t')][0]
			records.append(header)
		else:
			record = [line.strip().split('\t')]
			record = record[0] # Going form 'list of a list' to just a list

			if '|' in record[1]:
				record[1] = record[1].split('|')[3] #shortening the
PacBio contig name

			record[9] = record[9].split('%')[0] #removing % sign from
pctIdentity column
			records.append(record)
	return records

def main():
	parser = argparse.ArgumentParser(description='This script parses a specific
"csv" output generated by LASTZ')
	parser.add_argument('input', action="store", type=str, help="Filename of .csv
file in current folder")
	parser.add_argument('-o', '--output', action="store_true", help="Generates a
cleaned up version of the input csv file")
	parser.add_argument('-s', '--spacing', action="store_true", help="Calculates
spacing between repeats")
	parser.add_argument('-p','--gaps', action="store_true", help="Generates a file
with distribution of gaps inserted to get an alignment")
	parser.add_argument('-r','--repeats', action="store_true", help="Generates
fasta file with sequences identified in target(s)")
	parser.add_argument('-g','--gaps_upperlimit', action="store", default=20,
type=int,help="The outer limit of the number of gaps in target sequence")
	args = parser.parse_args()

	lastz_input = args.input
	fp = open(lastz_input)
	records = collect_records(fp)
	print >>sys.stderr, "Recovered %d records" %(len(records) -1)
```

```
	filename = lastz_input.split(".")[0]

	if args.output:
		out_file = open(filename+".forR.csv", "wb")
		writer = csv.writer(out_file, delimiter="\t", quoting =
csv.QUOTE_MINIMAL)
		for record in records:
			writer.writerow(record)
		out_file.close()

	if args.spacing:
		spacing = repeat_spacing(records)
		out_file = open(filename+".spacing.csv","wb")
		writer = csv.writer(out_file, delimiter="\t", quoting =
csv.QUOTE_MINIMAL)
		for line in spacing:
			writer.writerow(line)
		out_file.close()

	if args.gaps:
		gaps = repeat_gaps(records)
		out_file = open(filename+".gaps.csv","wb")
		writer = csv.writer(out_file, delimiter="\t", quoting =
csv.QUOTE_MINIMAL)
		for line in gaps:
			writer.writerow(line)
		out_file.close()

	gaps_upperlimit = args.gaps_upperlimit

	if args.repeats:
		repeats = extract_fasta(records,gaps_upperlimit)
		print >>sys.stderr, "Recovered %d records of lengths 165-180 bp"
%len(repeats)
		out_handle = open(filename+".fa","w")
		SeqIO.write(repeats, out_handle, "fasta")

if __name__ == '__main__':
	main()

#SCRIPT/shamoni-cen-evo-devo-81d1326/PROTOCOLS/Seq_2_Clstr.py
#! /usr/bin/env python
import argparse, sys, os, csv
from Bio import SeqIO
```

```
def make_kmers(seq,ksize):
	x = []
	for i in range(len(seq) - ksize + 1):
		x.append(seq[i:i+ksize])
	return x

def slurp_kmers(filepath,ksize,cutoff):
	counts = {}
	pos = {}
	fp = open(filepath, "rU")
	for record in SeqIO.parse(fp, "fasta"):
		sequence = str(record.seq)
		sequence = sequence.upper()

		kmers = make_kmers(sequence,ksize)

		for i,kmer in enumerate(kmers):
			counts[kmer] = counts.get(kmer, 0) + 1
			if pos.has_key(kmer):
				pos[kmer].append(i)
			else:
				pos[kmer] = [i,]

	fp.close()
	keylist = counts.keys()
	for kmer in keylist:
		if counts[kmer] < cutoff:
			del counts[kmer]
			del pos[kmer]
	all_kmers = set(counts.keys())
	pos.update((key,list(set(value))) for key,value in pos.items())
	return all_kmers,pos

def parse_signatures(kmers):
	l = len(kmers)
	sigs = []
	for i in range(0,l):
		a = set(kmers[i])
		for j in range(0,l):
			if i != j:
				a = a - kmers[j]
			else:
				pass
		sigs.append(a)
	return sigs
```

```
def sigkmers(path,ksize,cutoff):
	ls = os.listdir(path)
	fasta = filter(lambda x: ".fa" in x,ls)
	fasta.sort()
	print >>sys.stderr, '\nProcessing Kmers from %d Clusters' %len(fasta)
	print >>sys.stderr, 'Identifying Kmers of size %d shared by at least %d
sequences within the cluster' %(ksize,cutoff)
	print >>sys.stderr, ''

	k_by_clstr = []
	kpos_by_clstr = []
	for f in fasta:
		filepath = path+f
		kmers,pos = slurp_kmers(filepath,ksize,cutoff)
		k_by_clstr.append(kmers)
		kpos_by_clstr.append(pos)

	sig_k = parse_signatures(k_by_clstr)
	for i in range(0,len(fasta)): print >>sys.stderr, '%s; No. of Kmers %d; No. of
Signature Kmers %d' \

	%(fasta[i],len(k_by_clstr[i]),len(sig_k[i]))
	print >>sys.stderr, ''

	sigkmer_positions = [{} for i in range(len(sig_k))]
	for i in range(len(sig_k)):
		for kmer in sig_k[i]:
			sigkmer_positions[i][kmer] = kpos_by_clstr[i][kmer]
	return (fasta,sig_k,sigkmer_positions)

def assign_read_to_cluster(read,read_rc,sig_kmers,ksize):
	x = set(make_kmers(read, ksize))
	y = set(make_kmers(read_rc, ksize))
	union = x.union(y)

	isect_list = []
	for index in range(len(sig_kmers)):
		cluster_specific_kmers = sig_kmers[index]
		if union.intersection(cluster_specific_kmers):
			isect_list.append(index)

	return isect_list

def assign_reads_to_clusters(filepath,sig_kmers,ksize,only_reads):
	n_zero = 0
	n_unique = 0
```

```
	n_multi = 0

	cluster_counts = [0]*len(sig_kmers)
	cluster_multi_counts = [0]*len(sig_kmers)
	unique_records = [[] for i in range(len(sig_kmers))]
	for n, record in enumerate(SeqIO.parse(open(filepath),"fasta")):
		if only_reads is not None and n > only_reads:
			break
		read = str(record.seq).upper()
		read_rc = str(record.seq.reverse_complement()).upper()
		isect_list = assign_read_to_cluster(read,read_rc,sig_kmers,ksize)

		if len(isect_list) == 0:
			n_zero += 1
		elif len(isect_list) == 1:
			n_unique += 1
			cluster_counts[isect_list[0]] += 1
			unique_records[isect_list[0]].append(record)
		else:
			assert len(isect_list) > 1
			n_multi += 1
			for i in isect_list:
				cluster_multi_counts[i] += 1

	summary = [n_zero,n_unique,n_multi]
	return summary, cluster_counts, cluster_multi_counts,unique_records

def get_cluster_distribution(path,sig_kmers,ksize,num_reads,outdir):
	ls = os.listdir(path)
	fasta = filter(lambda x: ".fa" in x,ls)
	fasta.sort()
	print >>sys.stderr, 'Processing reads from %d ChIPseq datasets' %len(fasta)

	data = {}
	for f in fasta:
		filepath = path+f
		data[f] =
assign_reads_to_clusters(filepath,sig_kmers,ksize,num_reads)
		print >>sys.stderr, "Finished processing the file %s" %f

	return data

def write_data(data,outdir,clstr_id):
	os.mkdir(outdir)
	os.chdir(outdir)
	with open(outdir+"_summary.csv","wb") as outcsv:
```

```
				writer = csv.writer(outcsv)
				writer.writerow(["filename", "NOT_CEN","Unique_CEN","Multi_CEN"])
				for key,value in data.items():
					writer.writerow([key]+map(str,value[0]))

		with open(outdir+"_results.csv","wb") as outcsv:
				writer = csv.writer(outcsv)
				writer.writerow(["filename","Class"]+clstr_id)
				for key,value in data.items():
					writer.writerow([key]+["Unique"]+map(str,value[1]))
					writer.writerow([key]+["Multi"]+map(str,value[2]))

		for key,value in data.items():
				name = key.split('.')[0]
				for i in range(len(value[3])):
					clstr = clstr_id[i].split('.')[0]
					filename = name+"_"+clstr+".fa"
					SeqIO.write(value[3][i],filename,"fasta")

def write_signatures(sigkmer_positions,outdir,clstr_ids,k_len):
		with open(outdir+"_signatures.csv","wb") as outcsv:
				writer = csv.writer(outcsv)
				for i in range(len(clstr_ids)):
					for key,value in sigkmer_positions[i].items():

		writer.writerow([clstr_ids[i]]+[k_len]+[key]+map(str,value))

def main():
		parser = argparse.ArgumentParser()
		parser.add_argument("-Fc", "--folder-w-clusters", dest="clusterpath",
required=True)
		parser.add_argument("-Fr", "--folder-w-reads", dest="chipseqpath",
required=True)
		parser.add_argument("-O", "--outdir", type=str,default="READ-2-
CLSTR",dest="outdir")
		parser.add_argument("-K","--kmer-length", type=int, default=25,
dest="k_len")
		parser.add_argument("-Kt","--cutoff-kmers",type=int,default=100,
dest="k_co")
		parser.add_argument("-N","--max-no-of-reads-to-
analyze",type=int,default=None, dest="num_reads")
		args = parser.parse_args()

		clstr_ids,sig_kmers,sigkmer_positions =
sigkmers(args.clusterpath,args.k_len,args.k_co)
```

```
        data =
get_cluster_distribution(args.chipseqpath,sig_kmers,args.k_len,args.num_reads,args.
outdir)
        write_data(data,args.outdir,clstr_ids)
        write_signatures(sigkmer_positions,args.outdir,clstr_ids,args.k_len)

if __name__ == '__main__':
        main()


#SCRIPT/comailab-3604971/allprep-13.py
#! /usr/bin/env python2.6

import os, sys, math, time, operator
import subprocess
from itertools import imap
from optparse import OptionParser
from collections import defaultdict

#Comai Lab, Ucdavis Genome Center
#Meric Lieberman, 2016
#This work is the property of UC Davis Genome Center - Comai Lab
#This is shared under a Creative Commons BY-NC-ND 4.0 license
#https://creativecommons.org/licenses/by-nc-nd/4.0/

# Use at your own risk.
# We cannot provide support.
# All information obtained/inferred with this script is without any
# implied warranty of fitness for any purpose or use whatsoever.
#------------------------------------------------------------------------------
#
#Usage:
#For all modes the run command looks like this, with [...] indicating files needed by
specific read type, and {....} indication optional parameters
#allprep-8.py -b barcode-file.txt -f forward-read-file.fq [-r reverse-read-file.fq] [-i
index1-file.fq] [-I index2-file.fq] {-m} {-E} {-n} {-q} {-d}
#
#For a quick view of parameters form the command line, simply use the -h for help.
#allprep-8.py -h
#
#This program that takes a barcode file and splits the lane sequence.txt files into
specified
#library.txt (lib#.txt) files, does barcode check and match, 'N' filtering, primary and
#secondary adapter contamination, quality conversion, quality trimming (mean
quality of 20 over 5 base window),
#length trimming (default 35), and library separation.
```

```
#
#Input:
#This script takes a barcode file as specified in the sample sheet in the README, as
well as
#the forward read file, and optionally the reverse, index and secondary index
#These files are loaded with -b, -f, -r, -i, and -I respectively.
#There are also five additional operating options:
#-m for mismatch mode, this allows a 1 bp mismatch between the index / barcode
and the best matching barcode.
#-E for error reads mode, outputs the rejected reads to a file
#-n for N allowed mode, this turns off the check that rejects a read if there are any
'N' nucleotides in the sequence
#-q to convert a file using Illumina 1.5 qualities to Sanger/Illumina1.8 (standard)
#-D for only demultiplexing mode, this only splits the reads by barcode and does no
additional trimming or checks
#
#This program can take a file with most combinations of barcode/indexing. The
possibilities are:
#1. single ended, barcoded (read type 0)
#2. pair ended, barcoded (read type 1)
#3. single ended, one index (read type 2)
#4. single ended, two index (read type 5)
#5. pair ended, one index (read type 3)
#6. pair ended, two index (read type 4)
#
#Please see the README-barcode-file.txt and sample-barcode-file.txt for help in
creating the barcode file for your dataset.


usage = "\npath/%prog -h"
parser = OptionParser(usage=usage)

parser.add_option("-b", "--barcode", dest="bfile", default = False,
help="Barcode/Index file with all lib information. See sample table for details")
parser.add_option("-f", "--forward", dest="f", help="Forward read .fq file")
parser.add_option("-r", "--reverse", dest="r", default = False, help="Reverse read .fq
file")
parser.add_option("-i", "--index", dest="i", default = False, help="Index for forward
read, or index for paired read if only one index")
parser.add_option("-I", "--indexreverse", dest="i2", default = False, help="Reverse
read index file.")
parser.add_option("-m", "--mismatch", dest="miss", action="store_true", default =
False, help="Allow one base difference for indexed barcodes")
parser.add_option("-E", "--error", dest="errfile", action="store_true", default = False,
help="Create a rejected reads file.")
```

```
parser.add_option("-n", "--allowN", dest="ncheck", action="store_true", default =
False, help="Do not reject any sequence with'N' nucleotide automatically.")
parser.add_option("-N", "--Ncut", dest="ntrim", action="store_true", default = False,
help="Do Trim reads by N instead f rejecting them.")
parser.add_option("-q", "--qualities", dest="qual", action="store_true", default =
False, help="Convert Illumina 1.5 read qualities to sanger.")
parser.add_option("-D", "--demultiplex", dest="demulti", action="store_true", default
= False, help="ONLY Demultiplex, not trimming or checking at all")
parser.add_option("-M", "--minseqlen", dest="minSeqLen", type = "int", default=35,
help="Minimum sequence Length")


(opt, args) = parser.parse_args()

minMeanQual = 20.0



# Uses wc to get te number of lines in the file
def file_len(fname):
   p = subprocess.Popen(['wc', '-l', fname], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
   result, err = p.communicate()
   if p.returncode != 0:
      raise IOError(err)
   return int(result.strip().split()[0])

#prints out a status bar
def status(cur, total, good, bad):
     now = time.time()-start
     perdone = cur/float(total-1)
     percent = (perdone)*100
     emin = now/60
     esec = now%60
     rmin = (total-cur)/(cur/now)/60
     rsec = (total-cur)/(cur/now)%60
     sys.stdout.write('\r')
     sys.stdout.write(">%-20s< %i%% Elapsed: %im %is Remaining: %im %is Good:
%i Rejected %i Total: %i   " % ('='*int((perdone*20)),percent,emin,esec,rmin,rsec,
good, bad, good+bad))
     sys.stdout.flush()

#complement sequences
def comp(seq):
   complement = {'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A', 'N': 'N', 'R': 'R', 'Y': 'Y', 'S': 'S', 'W': 'W',
'M': 'M', 'K': 'K'}
```

```
    complseq = [complement[base] for base in seq]
    return complseq

#reverse complemtent
def rev(seq):
    seq = list(seq)
    seq.reverse()
    return ''.join(comp(seq))

#gets 4 reads from all given command parameter files
def get_reads(readset):
  ind = 0
  reads = [[],[],[],[]]
  for fh in readset:
    if fh != '':
      name = fh.readline()
      if name == '':
        break
      seq = fh.readline()
      plus = fh.readline()
      if opt.qual == True:
        qualt = fh.readline()
        qual = "".join(map((lambda x: chr(ord(x)-31)),qualt[:-1]))+'\n'
      else:
        qual = fh.readline()
      reads[ind] = [name, seq, plus, qual]
    ind+=1
  if reads != [[],[],[],[]]:
    return reads
  else:
    return "EOF"

#extracts sequences from all possible reads
def get_codes(readset):
  try:
    code1 = read[0][1][:-1]
  except:
    code1 = ''
  try:
    code2 = read[1][1][:-1]
  except:
    code2 = ''
  try:
    codei1 = read[2][1][:-1]
  except:
    codei1 = ''
```

```
  try:
    codei2 = read[3][1][:-1]
  except:
    codei2 = ''
  if lenindex != 0 and opt.i == False:
    codei1 = read[0][0].split('#')[-1].split('/')[0]
    try:
      codei2 = read[1][0].split('#')[-1].split('/')[0]
    except:
      codei2 = ''
  return [code1, code2, codei1, codei2]

#checks start of sequences against possible barcodes, returns a file handle, the code
type, and the barcode
def check_codes(curcodes):
  cfh = ''
  mtype = ''
  if len(codes['pindex2']) != 0:
    for poss in codes['pindex2'].keys():
      pcode1, pcode2 = poss.split('-')
      if opt.miss == False:
        if pcode1 == curcodes[2][:len(pcode1)] and pcode2 ==
curcodes[3][:len(pcode2)]:
          cfh = codes['pindex2'][poss]
          mtype = 'pindex2'
          break
      else:
        d1 = sum(imap(ne, pcode1, curcodes[2]))
        d2 = sum(imap(ne, pcode2, curcodes[3]))
        if d1 <= 1 and d2 <= 1:
          cfh = codes['pindex2'][poss]
          mtype = 'pindex2'
          break
  if len(codes['sindex2']) != 0 and cfh == '':
    for poss in codes['sindex2'].keys():
      pcode1, pcode2 = poss.split('-')
      if opt.miss == False:
        if pcode1 == curcodes[2][:len(pcode1)] and pcode2 ==
curcodes[3][:len(pcode2)]:
          cfh = codes['sindex2'][poss]
          mtype = 'sindex2'
          break
      else:
        d1 = sum(imap(ne, pcode1, curcodes[2]))
        d2 = sum(imap(ne, pcode2, curcodes[3]))
        if d1 <= 1 and d2 <= 1:
```

```
          cfh = codes['sindex2'][poss]
          mtype = 'sindex2'
          break
  if len(codes['pindex1']) != 0 and cfh == '':
    for poss in codes['pindex1'].keys():
      if opt.miss == False:
        if poss == curcodes[2][:len(poss)]:
          cfh = codes['pindex1'][poss]
          mtype = 'pindex1'
          break
      else:
        d1 = sum(imap(ne, poss, curcodes[2]))
        if d1 <= 1:
          cfh = codes['pindex1'][poss]
          mtype = 'pindex1'
          break
  if len(codes['sindex1']) != 0 and cfh == '':
    for poss in codes['sindex1'].keys():
      if opt.miss == False:
        if poss == curcodes[2][:len(poss)]:
          cfh = codes['sindex1'][poss]
          mtype = 'sindex1'
          break
      else:
        d1 = sum(imap(ne, poss, curcodes[2]))
        if d1 <= 1:
          cfh = codes['sindex1'][poss]
          mtype = 'sindex1'
          break
  if len(codes['old']) != 0 and cfh == '':
    for poss in codes['old'].keys():
      tposs = rev(poss)+overhangs[poss]
      if opt.miss == False:
        if tposs == curcodes[0][:len(tposs)] and (tposs == curcodes[1][:len(tposs)] or
opt.r == False):
          cfh = codes['old'][poss]
          mtype = 'old'
          break
      else:
        d1 = sum(imap(ne, tposs, curcodes[0]))
        d2 = sum(imap(ne, tposs, curcodes[1]))
        if d1 <= 1 and (d2 <= 1 or opt.r == False):
          cfh = codes['old'][poss]
          mtype = 'old'
          break
  return [cfh, mtype, poss]
```

```
#used to output to rejected file
def error_out(read, error):
  if opt.errfile != False:
    eout = read[0][0]+read[0][1]+"+"+error+'\n'+read[0][3]
    if opt.r != False:
      eout += ''.join(read[1])
    if opt.i != False:
      eout += ''.join(read[2])
    if opt.i2 != False:
      eout += ''.join(read[3])
    efile.write(eout)


#Adapter to look for
#secondadapt = "AGATCGGAAG"
#mainadapt = "AGATCGGAAG"



mainadapt = "AGATCGGAAGAGC"


ne = operator.ne
######### READ IN BARCODE FILE
try:
  bt = open(opt.bfile)
  bt.readline()
except:
  parser.error("Missing valid barcode table. Please check your command line
paramters with -h or --help")

codes = {}
codes['old'] = {}
codes['sindex1'] = {}
codes['sindex2'] = {}
codes['pindex1'] = {}
codes['pindex2'] = {}
rembases = {}
overhangs = {}

handles={}
#read in the barcode table, sorting by type
for l in bt:

  x= l[:-1].split('\t')
```

```
  if x == ['']:
    continue
  x = map(lambda y: y.replace(' ',''), x)
  libname, rtype, code1, code2, over, rembase = x
  rembase = rembase.replace('.','0')
  if int(rtype) == 0 or int(rtype) == 1:
    ctype = 'old'
    name = code1
  if int(rtype) == 2:
    ctype = 'sindex1'
    name = code1
  if int(rtype) == 3:
    ctype = 'pindex1'
    name = code1
  if int(rtype) == 4:
    ctype = 'pindex2'
    name = code1+'-'+code2
  if int(rtype) == 5:
    ctype = 'sindex2'
    name = code1+'-'+code2
  codes[ctype][name] = libname
  if libname not in handles:
    handles[libname] = open(libname+'.fq', 'w')
  else:
    print libname
    parser.error("Duplicate Filename !!! :"+libname)
  rembases[name] = int(rembase)
  over = over.replace('.','')
  overhangs[name] = over

bt.close()

lenindex = len(codes['sindex1']) + len(codes['pindex1']) + len(codes['pindex2']) +
len(codes['sindex2'])

#duh, do not run for both
if opt.miss == True:
  if len(codes['old']) != 0 and lenindex != 0:
    parser.error("Missmatch mode cannot be run with both barcodes and indexed
reads. Please check your command line paramters with -h or --help")

#open all command parameter indicated read files
try:
  frlen = file_len(opt.f)/4
  #frlen = 296146516
  ffile = open(opt.f)
```

```
except:
  parser.error("Please specify reads files. Please check your command line
paramters with -h or --help")
if opt.r !=  False:
  rfile = open(opt.r)
else:
  rfile = ''
if opt.i !=  False:
  ifile = open(opt.i)
else:
  ifile = ''
if opt.i2 !=  False:
  i2file = open(opt.i2)
else:
  i2file = ''
if opt.errfile != False:
  efile = open("rejected-reads-"+opt.bfile, 'w')
creadset = [ffile, rfile, ifile, i2file]

# go through all attached files one read at a time
ct = 0
good = 0
bad = 0
start = time.time()
while 1:
  if ct % 20000 == 8:
    status(ct, frlen, good, bad)
  ct+=1

  #pull reads frm files
  read = get_reads(creadset)
  if read == "EOF":
    break

  #get relevant barcodes/indexes
  curcodes = get_codes(read)
  #valid_codes = check_codes(codes)

  cfh, mtype, poss = check_codes(curcodes)
  bad_flag = 0
  error = ''
  if cfh == '':
    bad_flag = 1
    error = "No Barcode Match"
    bad+=1
    error_out(read, error)
```

```
  continue

nameflag = 0
if opt.i == False:
  nameflag = 1

if mtype == 'old':
  forwardname = read[0][0][:-1]+'#'+poss+'\n'
  forwardseq = read[0][1][len(poss)+len(overhangs[poss]):]
  forwardqual = read[0][3][len(poss)+len(overhangs[poss]):]
  if opt.r != False:
    reversename = read[1][0][:-1]+'#'+poss+'\n'
    reverseseq = read[1][1][len(poss)+len(overhangs[poss]):]
    reversequal = read[1][3][len(poss)+len(overhangs[poss]):]
elif mtype == 'sindex1' or mtype == 'pindex1':
  if nameflag == 0:
    forwardname = read[0][0][:-1]+poss+'\n'
  else:
    forwardname = read[0][0][:-1]+'\n'
  forwardseq = read[0][1]
  forwardqual = read[0][3]
  if mtype == 'pindex1':
    if nameflag == 0:
      reversename = read[1][0][:-1]+poss+'\n'
    else:
      reversename = read[1][0][:-1]+'\n'
    reverseseq = read[1][1]
    reversequal = read[1][3]
elif mtype == 'pindex2':
  poss1, poss2 = poss.split('-')
  if nameflag == 0:
    forwardname = read[0][0][:-1]+poss1+'\n'
    reversename = read[1][0][:-1]+poss2+'\n'
  else:
    forwardname = read[0][0][:-1]+'\n'
    reversename = read[1][0][:-1]+'\n'
  forwardseq = read[0][1]
  forwardqual = read[0][3]

  reverseseq = read[1][1]
  reversequal = read[1][3]
elif mtype == 'sindex2':
  poss1, poss2 = poss.split('-')
  if nameflag == 0:
    forwardname = read[0][0][:-1]+poss1+':'+poss2+'\n'
  else:
```

```
    forwardname = read[0][0][:-1]+'\n'
  forwardseq = read[0][1]
  forwardqual = read[0][3]
else:
  parser.error("Error: Check barcode splitting")

#remove bases
if rembases[poss] != 0:
  forwardseq = forwardseq[rembases[poss]:]
  forwardqual = forwardqual[rembases[poss]:]
  if opt.r != False:
    reverseseq = reverseseq[rembases[poss]:]
    reversequal = reversequal[rembases[poss]:]

if opt.demulti == False:

  #check for main type of adapter contamination and trim as needed

  if mainadapt in forwardseq:
    t1 = forwardseq[:forwardseq.index(mainadapt)]
    l1 = len(t1)
    forwardseq = t1+'\n'
    forwardqual = forwardqual[:l1]+'\n'
  if opt.r != False:
    if mainadapt in reverseseq:
      t1 = reverseseq[:reverseseq.index(mainadapt)]
      l1 = len(t1)
      reverseseq = t1+'\n'
      reversequal = reversequal[:l1]+'\n'

  if rev(mainadapt) in forwardseq:
    t1 = forwardseq[:forwardseq.index(rev(mainadapt))]
    l1 = len(t1)
    forwardseq = t1+'\n'
    forwardqual = forwardqual[:l1]+'\n'
  if opt.r != False:
    if rev(mainadapt) in reverseseq:
      t2 = reverseseq[:reverseseq.index(rev(mainadapt))]
      l2 = len(t2)
      reverseseq = t2+'\n'
      reversequal = reversequal[:l2]+'\n'


  #calculate avg quality and trim if lower then threshold (20)

  quals = map(lambda x: ord(x)-33, forwardqual[:-1])
```

```
for x in range(len(quals)-4):
  cut = quals[x:x+5]
  ave = float(sum(cut))/5.0
  if ave < minMeanQual:
    forwardqual = forwardqual[:x]+'\n'
    forwardseq = forwardseq[:x]+'\n'
    break
if opt.r != False:
  quals2 = map(lambda x: ord(x)-33, reversequal[:-1])
  for x in range(len(quals2)-4):
    cut2 = quals2[x:x+5]
    ave2 = float(sum(cut2))/5.0
    if ave2 < minMeanQual:
      reversequal = reversequal[:x]+'\n'
      reverseseq = reverseseq[:x]+'\n'
      break

if opt.ntrim == True:
  if opt.ncheck == False:
    for x in range(len(forwardseq)):
      if forwardseq[x] == 'N':
        forwardqual = forwardqual[:x]+'\n'
        forwardseq = forwardseq[:x]+'\n'
        break
    if opt.r != False:
      for x in range(len(reverseseq)):
        if reverseseq[x] == 'N':
          reversequal = reversequal[:x]+'\n'
          reverseseq = reverseseq[:x]+'\n'
          break
elif opt.ntrim == False:
  #check for 'N's in sequence
  if opt.ncheck == False:
    if 'N' in forwardseq:
      bad_flag = 1
      error += "NinF"
    if opt.r != False:
      if 'N' in reverseseq:
        bad_flag = 1
        error += "NinR"
    if bad_flag != 0:
      bad+=1
      error_out(read, error)
      continue
```

```
      #check that chopped length is not too small
      if len(forwardseq) < opt.minSeqLen+1 or len(forwardqual) < opt.minSeqLen+1:
        bad_flag = 1
        error += "F too short"
      if opt.r != False:
        if len(reverseseq)< opt.minSeqLen+1 or len(reversequal) < opt.minSeqLen+1:
          bad_flag = 1
          error += "R too short"
      if bad_flag != 0:
        bad+=1
        error_out(read, error)
        continue

    outline = forwardname+forwardseq+'+\n'+forwardqual
    if opt.r != False and mtype != 'sindex1' and mtype != 'sindex2':
      outline += reversename+reverseseq+'+\n'+reversequal

    handles[cfh].write(outline)
    good+=1
  status(ct, frlen, good, bad)
  print ""

  #close all library file handles
  for ctype in handles.keys():
    handles[ctype].close()


  ffile.close()

  if opt.r !=  False:
    rfile.close()

  if opt.i !=  False:
    ifile.close()

  if opt.i2 !=  False:
    i2file.close()

  if opt.errfile != False:
    efile.close()


  #THE END
```