Marker Finder Program Report

Xu Luze

September 19, 2015

Contents

1 Overview

To increase the accuracy and specificity of the detection, we develop an assay over our Paired dCas9 Reporter System to extract more sequence information out of the target genome. The core as well as the first step of the design of array is to screen over the entire genome and get specific sequences (CRISPR target sites) with high specificity as markers. We develop a method named SSPD to achieve our aim, which is composed of 4 steps:

- Search for guide candidates
- Specificity test for each candidate
- Pair left and right guides with optimal spacer length
- Design PCR fragments

After the target sites having been chosen, we developed an Oligo Generator to turn the target sites into oligonucleotides sequences for following sgRNA construction combined with our gRNA generator (Part). And *Python 3.4.3* and extension module *BioPython-1.65* were then used to find markers.

2 SSPD Method

2.1 Search for guide candidates

Firstly, we choose '*.fasta' as our main file format passing sequence information between files and the GenBank file with '.gb', '.gbk', '.gbff' extension were converted to the fasta file by using SeqIO.convert in Bio module. Then we took advantage of *Python 3.4.3* build-in regular expression module *re* to search for left guide sequences of gRNA ($`(? <= cc).(? = .20)'$) and right guide sequences of gRNA ($`(? <= .20).(? = gg)'$) separately, which would be paired later for PC reporter system to function. As Supplementary 1 states, the two kinds of gRNA can form pairs with any of the four PAM orientations.*Marker_Finder_prepare.py* is the program to realize this step, main functions of it are listed below:

- **gb_to_fasta**: change file format from GenBank to fasta
- **gRNA_candidate_iteration**: return the iteration of gRNA candidate
- **gRNA_candidate**: SeqIO.write the iteration into one file

```
def gb_to_fasta(targetname,target_filename):
    SeqIO.convert(target_filename,'gb',fastafilename,'fasta')
def gRNA_candidate_iteration(dna_record, gRNA_flag):
    dna = str(dna_record.seq).lower()
    if gRNA_flag == 'CCN':  # CCNN20
        regexp = compile("(?<=cc).(?=.{20})")
        for mat in regexp.finditer(dna):
            yield SeqRecord(Seq(dna[mat.start()+1:mat.start()+20+1]),
                id='{}|{}'.format(dna_record.id,mat.start()+2),description='')
# each sequence with a special id composed of the target DNA's id and the start
                                              position of 20bp in the target DNA
    elif gRNA_flag == 'NGG':  # N20NGG
        regexp = compile("(?<=.{20}).(?=gg)")
        for mat in regexp.finditer(dna):
            yield SeqRecord(Seq(dna[mat.start()-20:mat.start()]),
                id='{}|{}'.format(dna_record.id,mat.start()-20+1),description='')
def gRNA_candidates(dna_record,gRNA_flag,out_filename):
    gRNA_iter = gRNA_candidate_iteration(dna_record,gRNA_flag)
    SeqIO.write(gRNA_iter, out_filename, 'fasta')
```

2.2 Specificity test for each guide candidate

2.2.1 PAM-proximal 12bp filtration by BLAST

first_filter.py is the program to realize this step, main functions are listed below:

- **first_filter_blast_db_iteration**: return the iteration of gRNA candidate after first filtration by blasting with database
- **first_filter_blast_sbjct_iteration**: return the iteration of gRNA candidate after first filtration by blasting with subject(can be target itself)
- **first_filter_blast_extract_iteration**: analysis the blast result to test the specificity of gRNA
- **first_filter_spec_test**: if the off-target site has the same 12bp PAM-proximal sequences as guide sequence of gRNA return true else return false
- **first_filter_db**: SeqIO.write the iteration from first_filter_blast_db_iteration into one file
- **first_filter_sbjct**: SeqIO.write the iteration from first_filter_blast_sbjct_iteration into one file

```
def first_filter_blast_db_iteration(in_filename,database_name,gRNA_flag):
    blastn_cline = NcbiblastnCommandline(query=in_filename, db=database_name, out=
                                          blast_out_filename, evalue=10,
                                          word_size=11,reward=1,penalty=-3,
                                          gapopen=5,gapextend=2,outfmt=5,
                                          num_threads=4)
    stdout, stderr = blastn_cline()

def first_filter_blast_sbjct_iteration(in_filename,subject_filename,gRNA_flag):
    blastn_cline = NcbiblastnCommandline(query=in_filename, subject=subject_filename,
                                           out=blast_out_filename, evalue=10,
                                          word_size=11,reward=1,penalty=-3,
                                          gapopen=5,gapextend=2,outfmt=5)
    stdout, stderr = blastn_cline()

def first_filter_blast_extract_iteration(in_filename, blast_out_filename,gRNA_flag,
                                      self_flag=0):
    gRNA_dict = SeqIO.index(in_filename,'fasta',key_function=get_position)
    blast_records = NCBIXML.parse(open(blast_out_filename))
    return (gRNA_dict[position] for position in gRNA_dict if spec_flag_dict[position]
                                          )

def first_filter_spec_test(hsp,gRNA_flag):
    if gRNA_flag == 'CCN':
        return hsp.query_start == 1 and hsp.match.startswith('||||||||||||')
    elif gRNA_flag == 'NGG':
        return hsp.query_end == 20 and hsp.match.endswith('||||||||||||')

def first_filter_db(in_filename, out_filename, database_name,gRNA_flag):
    blast_out_filename = first_filter_blast_db_iteration(in_filename, database_name,
                                          gRNA_flag)
    gRNA_iter = first_filter_blast_extract_iteration(in_filename,blast_out_filename,
                                          gRNA_flag)
    SeqIO.write(gRNA_iter, out_filename, 'fasta')

def first_filter_sbjct(in_filename, out_filename, subject_filename,gRNA_flag):
    blast_out_filename = first_filter_blast_sbjct_iteration(in_filename,
                                          subject_filename, gRNA_flag)
    gRNA_iter = first_filter_blast_extract_iteration(in_filename,blast_out_filename,
                                          gRNA_flag,1)
    SeqIO.write(gRNA_iter, out_filename, 'fasta')
```

2.2.2 Score filtration

second_filter.py is the program to realize this step, main functions are listed below:

- **single_score**: calculate the score of one single off-target site
- **before_query_analysis, align_analysis,after_query_analysis**: analyze the mismatch within and beyond the range of align
- **second_filter_prepare**: upload to BLAST again to find off-targets
- **score_blast_extract,score_blast_db_extract**: score guide sequences of gRNA partially on self or on database
- **gRNA_score**: score guide sequences of gRNA
- **gRNA_score_filter**: filter guide sequences of gRNA with low score or max off-target score
- **second_filter**: SeqIO.write the reserved guide sequences of gRNA after score filtration into one file

```
def single_score(mismatch_lst, gRNA_flag):
    singlescore = 100*w / ((19-mean_distance)/19.0*4+1.0)/(mismatch_num**2)
    return singlescore

def before_query_analysis(hsp, target_sequence, subject_sequence, mismatch_list, rev=
                                              False):
def align_analysis(hsp, mismatch_list, gap_list):
def after_query_analysis(hsp, target_sequence, subject_sequence, mismatch_list, rev=
                                              False):

def score_blast_extract(score_gRNA_dict, maxscore_gRNA_dict, choose_gRNA_dict,
                                              subject_filename, blast_outfile_dir,
                                              gRNA_flag):

def score_blast_db_extract(score_gRNA_dict, maxscore_gRNA_dict, choose_gRNA_dict,
                                              subject_filename, blast_outfile_dir,
                                              gRNA_flag):

def second_filter_prepare(filename_str, target_filename, database_name):
    for gRNA_flag in ['CCN', 'NGG']:
    # upload to BLAST again
        first_filter_blast_db_iteration(in_filename, database_name, gRNA_flag)
        first_filter_blast_sbjct_iteration(
            in_filename, target_filename, gRNA_flag)

def gRNA_score(choose_gRNA_filename, target_filename, target_blast_outfile_dir,
                                              subject_filename,
                                              subject_blast_outfile_dir, out_filename,
                                              gRNA_flag):
    score_blast_db_extract(score_gRNA_dict, maxscore_gRNA_dict,
                               choose_gRNA_dict, subject_filename,
                                                                    subject_blast_outfile_dir
                                                                    , gRNA_flag
                                                                    )
    score_blast_extract(score_gRNA_dict, maxscore_gRNA_dict,
                        choose_gRNA_dict, target_filename, target_blast_outfile_dir,
                                                             gRNA_flag)

def gRNA_score_filter(infname, outfname, gRNA_score_threshold=45, max_score_threshold
                                              =2):
    if score > gRNA_score_threshold and max_score <= max_score_threshold:
            lst.append(line)
```

```
def second_filter(choose_gRNA_filename, target_filename, target_blast_outfile_dir,
                                          subject_filename,
                                          subject_blast_outfile_dir, gRNA_flag,
                                          gRNA_score_threshold=45,
                                          max_score_threshold=2):
    filedir = dirname(choose_gRNA_filename)
    out_filename = '{}\score_{}_gRNA.txt'.format(filedir, gRNA_flag)
    gRNA_score(choose_gRNA_filename, target_filename, target_blast_outfile_dir,
               subject_filename, subject_blast_outfile_dir, out_filename, gRNA_flag)
    choose_filename = '{}\choose_{}_gRNA.txt'.format(filedir, gRNA_flag)
    gRNA_score_filter(
        out_filename, choose_filename, gRNA_score_threshold, max_score_threshold)
```

2.3 Pair left and right guides with optimal spacer length

Firstly, we sorted the specific guide sequences by start position. Then we scanned the left gRNA and stored the nearest right gRNA with optimal spacer length. Finally, show the paired gRNA as markers on the user interface.

```
def spacer_finder_prepare(self, flag=0):
    orientation_flag = self._Orientation_Var.get()
    if orientation_flag == 1:
        self.spacer_finder(CCN, NGG, 'CCN_NGG', flag)
    elif orientation_flag == 2:
        self.spacer_finder(NGG, CCN, 'NGG_CCN', flag)
    elif orientation_flag == 3:
        self.spacer_finder(CCN, CCN, 'CCN_CCN', flag)
    elif orientation_flag == 4:
        self.spacer_finder(NGG, NGG, 'NGG_NGG', flag)
def spacer_finder(self, CCN, NGG,gRNA_str,sort_flag):
    min_len = self._MinLengthVar.get()
    max_len = self._MaxLengthVar.get()
```

2.4 Design PCR fragments

We provide two methods to determine PCR fragments. For the first, fix pair number k per fragment, search the adjacent but non-overlapped k pairs. The results were sorted by fragment length, and we named it as 'Fix Number Test'. For another, fix maximal PCR fragment length, the results were sorted with pair numbers per fragment, and its name was 'Fix Length Test'. Sorted results would be presented on user interface, enabling users to select fragments by themselves as needed. While selecting overlapping fragment is not allowable for array design.

- **fix_number_cluster**: Fix Number Test
- **fix_length_cluster**: Fix Length Test
- **cluster_analysis**: Show the information of selected cluster

```
def fix_number_cluster(self):
    v_num = len(s)
    group_num = 100
    min_group_list = [[] for i in range(group_num)]
    min_group_subject_len_list = [1000000 for i in range(group_num)]
    k = 0
    while k < v_num:
        group = []
        start = s[k][0]
        current = k
        while current< v_num and len(group) < most_number:
            group_len = s[current][1] - start + 1
            group.append(current)
            next = current+1
            while next<v_num and isoverlap(s[current],s[next]):
                next += 1
            current = next
        k += 1
        if len(group)<most_number:
            continue
        i = group_num-1
        if len(group)==len(min_group_list[i]) and group_len<
                                                   min_group_subject_len_list[i] or
                                                   len(group)>len(min_group_list[i]):
            while i > 0 and \
                    (len(group)>len(min_group_list[i-1]) or
                                         len(group)== len(min_group_list[i-1]) and
                                                                                    group_len
                                                                                    <
                                                                                    min_group_subject_l
                                                                                    [
                                                                                    i
                                                                                    -
                                                                                    1
                                                                                    ]
                                                                                    )
                                                                                    :
                min_group_list[i] = min_group_list[i-1]
                min_group_subject_len_list[i] = min_group_subject_len_list[i-1]
                i -= 1
            min_group_list[i] = group
            min_group_subject_len_list[i] = group_len

def fix_length_cluster(self):
    v_num = len(s)
    group_num = 100
    max_group_list = [[] for i in range(group_num)]
    k = 0
    while k < v_num:
        group = []
        start = s[k][0]
        current = k
        while current< v_num and s[current][1] - start + 1 <= max_len:
            group.append(current)
            next = current+1
            while next<v_num and isoverlap(s[current],s[next]):
                next += 1
            current = next
        i = group_num-1
        if len(group)>len(max_group_list[i]):
            while i > 0 and len(group)>len(max_group_list[i-1]):
                max_group_list[i] = max_group_list[i-1]
                i -= 1
            max_group_list[i] = group
        k += 1
```

3 Oligo Generator

Detecting multi-site on pathogen genome can significantly enhance specificity and sensitivity. Using SSPD method mentioned above, the target sites were designed on genome. However, designing multiple target sites into oligonucleotides sequences for following sgRNA construction manually can be laborious. Thus here we developed a supplementary program to facilitate oligo sequence generation, which is combined with our gRNA generator. Specifically, we used Golden Gate Cloning to make it more convenient to substitute guide sequences for different target sites. Here is the oligo generator and its main function lines.

```
def oligo_generator(self,fix_test_name):
    FW_left_gRNA = 'tagg'+revcmp(left_gRNA)
    RV_left_gRNA = 'aaac'+left_gRNA
    FW_right_gRNA = 'tagg'+right_gRNA
    RV_right_gRNA = 'aaac'+revcmp(right_gRNA)
```