PTC_Epitope_ID

August 14, 2023

0.1 Whole Genome Data Analysis - PTC

Written by Ryan V Moriarty, updated August 2023

0.1.1 Purpose:

Merge and map whole-genome sequences, identify variants, and characterize the sequences in viral epitopes. Processes a folder hierarchy of FASTQ files and saves epitope sequences, counts and statistics files to a corresponding folder hierarchy. This script will generate a tsv file with nucleotide sequences and the number of times they are present in the data set. This has been edited from Aliota et al, 2018 to examine MHC-restricted SIV epitopes and identify SNVs.

To make your life easier downstream, format your FASTQ file names as such:

- animal-Xdpi-repX_L001_R1_001.fastq.gz

Python packages needed: - samtools - os - pathlib - tempfile - shutil - subprocess - pandas - glob - re - Bio.Seq

Additional requirements: - BBtools

0.2 Specify dataset arguments

Specify variables that need to be changed depending on the dataset and machine configuration. After these parameters are specified, the rest of the notebook should be runnable automatically.

[ ]:
```
# path to FASTQ files to process
# expects R1 and R2 paired samples
FASTQ_FOLDER_PATH = "/Volumes/titmouse/ViralEvolutionART/raw_fastqs"
exp_id = "WGS" # For naming epitope .txt files

ANIMAL_PATTERN = "cy\d{4}" # Not all animals have the same naming conventioin/
 ↪name length

# path to reference files for read orientation
REF_BASE = '/Volumes/titmouse/ViralEvolutionART/ReferenceFiles'
BBMAP_PATH = REF_BASE + '/bbmap'
REF_FASTA = REF_BASE + '/M33262_barcode_withends.fasta'
EPITOPES=True
```

```
import pandas as pd
epitope_seqs = pd.read_csv(REF_BASE + "/pre_and_post_epitope_seqs.csv")
epitope_seqs['epitopeLength'] = epitope_seqs['epitopeLength'].astype(str)
```

0.3 Functions for epitope identification

Define functions that will be used in the workflow to count epitope sequences from FASTQ files. These functions have not been changed since they were used as published in Aliota et al, 2018, aside from commenting out the print_status command.

[ ]:
```
def create_temp_file():
    '''create named temporary file
    return temporary file object [0] and path to temporary file [1]'''

    import pathlib
    import tempfile

    temp = tempfile.NamedTemporaryFile()
    return [temp, temp.name]

def create_directory(dir_path):
    '''create directory at specified location if one does not already exist'''

    import os

    if not os.path.exists(dir_path):
        os.makedirs(dir_path)

    return dir_path

def find_files_in_path(search_path):
    '''get list of files matching files in search_path'''
    import glob

    f = glob.glob(search_path, recursive=True)

    return f

def derive_file_name(source_file, find_string, replace_string):
    '''create filename string '''

    f = source_file.replace(find_string, replace_string)

    return f
```

```
def run_command(cmd_list, stdout_file = None, stderr_file = None):
    '''run command with subprocess.call
    if stdout or stderr arguments are passed, save to specified file
    '''

    import subprocess

    #print(cmd_list)

    # if neither stdout or stderr specified
    if stdout_file is None and stderr_file is None:
        subprocess.call(cmd_list, stdout=subprocess.DEVNULL)
        #RVM added subprocess.DEVNULL so it would stop printing the java␣
 ↪command

    # if only stdout is specified
    elif stdout_file is not None and stderr_file is None:
        with open(stdout_file, 'w') as so:
            subprocess.call(cmd_list, stdout = so)

    # if only stderr is specified
    elif stdout_file is None and stderr_file is not None:
        with open(stderr_file, 'w') as se:
            subprocess.call(cmd_list, stderr = se)

    # if both stdout and stderr are specified
    elif stdout_file is not None and stderr_file is not None:
        with open(stdout_file, 'w') as so:
            with open(stderr_file, 'w') as se:
                subprocess.call(cmd_list, stdout = so, stderr = se)

    else: pass

def split_path(source_file):
    '''get path to enclosing folder and basename for source_file'''

    import os

    source_file_path = os.path.dirname(source_file)
    source_file_basename = os.path.basename(source_file)

    return [source_file_path, source_file_basename]

def create_barcode_tsv(two_column_barcode_count_file,␣
 ↪three_column_barcode_count_file, sample_name):
    '''add a column with sample_name to barcode count TSV output by␣
 ↪kmercountexact'''
```

```
    import pandas as pd
    # import file containing counts to pandas dataframe
    df = pd.read_csv(two_column_barcode_count_file, sep='\t',␣
 ↪names=['barcode_sequence', 'barcode_count'])

    # add column with sample name
    df.insert(0, 'sample_name', sample_name)

    # export TSV with sample names added
    df.to_csv(three_column_barcode_count_file, sep='\t', index=False)

def remove_file(source_file):
    '''remove file'''

    import os

    os.remove(source_file)
```

0.4 Workflow for Epitope Identification

Count unique epitope sequences in FASTQ sequences by:

1. Make dictionary of samples to process.
   - The key is the sample directory path and the value is the sample name (ex. cy1035-14dpi-rep1)
   - Merge the R1 and R2 for each replicate to create combiined R1 and R2 fastq files.
2. Iterate over each sample.

3. Determine sequencing method by identifying the index sequence in the FASTQ header.

4. Merge and quality trim samples
5. Map the merged whole-genome fastq file to the SIVmac239M reference using bbmap.
6. Loop through each epitope if EPITOPES is set to "True"
7. Extracting reads that contain the epitope
   - this is done by searching for the upstream and downstream sequences (allowing for a 1bp mismatch).
8. Orient reads in same direction
   - the file is re-mapped to SIVmac239 using bbmap. The plus and minus strands are split using the "splitsam" utility of bbmap. The minus strand is reverse complemented and the two fastq files are concatenated.

9. Trim reads to contain only the epitope
   - Using bbduk, the upstream and downstream flanking sequence are removed, again allowing for a 1 bp mismatch. By orienting the reads in Step 6, the upstream sequence can be left-trimmed and the downstream sequence can be right-trimmed without needing to account for reverse complements.

10. Count number of times each epitope appears
    - Only sequences exactly matching the nucleotide length (i.e. 24bp for an 8mer) are included to exclude any PCR chimeras that are detected.
11. Create TSV file with counts for each epitope sequence. This .txt file can then be processed further.
    - Temporary files generated during this process are removed.

0.4.1 Create a dictionary of sample names, where the key is the sample directory path and the value is the file name sans forward/reverse designator and file extension.

[ ]:
```
## 1. Make dictionary of samples to process ##
FASTQ_R1 = find_files_in_path(FASTQ_FOLDER_PATH + '/*R1*.fastq.gz')

SAMPLES = {}
for i in FASTQ_R1:

    # get path to directory containing sample FASTQ
    SAMPLE_DIR = split_path(i)[0]

    # get sample name
    FASTQ_R1_BASENAME = split_path(i)[1]
    SAMPLE_NAME = re.sub('_S\d{1,2}_L001_|R1_001.fastq.gz', '',␣
 ↪FASTQ_R1_BASENAME) # RVM adjusted March 2023
    #SAMPLE_NAME = derive_file_name(FASTQ_R1_BASENAME, '_R1_001.fastq.gz', '')
    #print(SAMPLE_NAME[:SAMPLE_NAME.index('_S')])

    SAMPLES[i] = (SAMPLE_DIR, SAMPLE_NAME)
print(SAMPLES) # If you want to verify what samples are there and the absolute␣
 ↪paths
```

0.4.2 Iterate over each sample and merge paired reads, map to reference, and find epitope sequences

[ ]:
```
#### 2. Iterate over every sample to process and: ##
for key,val in SAMPLES.items():

    # get sample path and sample name
    SAMPLE_PATH = val[0]
    SAMPLE_NAME = val[1]

    R1 = key
    R2 = re.sub("L001_R1_001", "L001_R2_001", R1)

    # Alert uer to where you are in the loops
```

```
   print("Processing sample " + str(list(SAMPLES.keys()).index(key) + 1 ) + "␣
↪of " + str(len(SAMPLES)) + ": " + SAMPLE_NAME)

   # create directory to hold statistics files
   STATS_PATH = create_directory(SAMPLE_PATH + '/stats')
   MERGED_FQ_PATH = create_directory(FASTQ_FOLDER_PATH + '/merged_fastqs')

   ## 3. Quality trimming and merging R1 and R2 FASTQ sequences ##
   print("Merging FASTQ files")
   run_command([BBMAP_PATH + '/bbmerge.sh',
                'in=' + R1,
                'in2=' + R2,
                'out=' + MERGED_FQ_PATH + '/' + SAMPLE_NAME + '.merged.fastq.
↪gz',
                'qtrim=t',
                'ow=t'],
                stderr_file = STATS_PATH + '/' + SAMPLE_NAME + '.merging.stats.
↪txt')

   print("Mapping full genome")
   OUTPUT_BAM = create_directory(FASTQ_FOLDER_PATH + '/bamfiles')
   run_command([BBMAP_PATH + '/bbmap.sh',
                    'in=' + MERGED_FQ_PATH + '/' + SAMPLE_NAME + '.merged.fastq.
↪gz',
                    'outm=' + OUTPUT_BAM + '/' + SAMPLE_NAME + '.fullmapped.
↪sam',
                    'ref=' + REF_FASTA,
                    'ow=t'],
                    stderr_file=STATS_PATH + '/' + SAMPLE_NAME + '.fullmapping.
↪stats.txt')

   #print("Converting to BAM file, indexing, and generating coverage file")
   # Samtools must be version 1.9 for quasitools to later work
   #run_command(['samtools', 'sort',
   #              OUTPUT_BAM + '/' + SAMPLE_NAME + '.fullmapped.sam',
   #              '-o',
   #             OUTPUT_BAM + '/' + SAMPLE_NAME + '.bam'])

   #run_command(['samtools', 'index', '-b', OUTPUT_BAM + '/' + SAMPLE_NAME + '.
↪bam'])


   if EPITOPES == True:
       EPITOPE_OUT_PATH = create_directory(FASTQ_FOLDER_PATH + '/EpitopeFiles')
       print("Finding Epitope Changes")
       ## 4. Extracting reads that contain the barcode ##
```

```
        for e in range(len(epitope_seqs)): # RVM added this loop

            UPSTREAM_FLANKING = epitope_seqs['preEpitopeSeq'][e]
            DOWNSTREAM_FLANKING = epitope_seqs['PostEpitopeSeq'][e]
            EPITOPE_NAME = epitope_seqs['EpitopeName'][e]

            print("Starting epitope: " + EPITOPE_NAME)

            # extract reads containing barcode
            # bbduk command to find reads containing upstream flanking sequence␣
↪
            #print("extracting reads")
            run_command([BBMAP_PATH + '/bbduk.sh',
                         'in=' + MERGED_FQ_PATH + '/' + SAMPLE_NAME + '.merged.
↪fastq.gz',
                         'outm=' + EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪containing_upstream_flank.fastq.gz',
                         'literal=' + UPSTREAM_FLANKING,
                         'k=20',
                         'hdist=1',
                         'ow=t'],
                         stderr_file = STATS_PATH + '/' + SAMPLE_NAME + '-' +␣
↪EPITOPE_NAME + '.containing_upstream_flank.stats.txt')

            # bbduk command to find reads containing downstream flanking␣
↪sequence
            # (among those that contain upstream flanking sequence)
            run_command([BBMAP_PATH + '/bbduk.sh',
                         'in=' + EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪containing_upstream_flank.fastq.gz',
                         'outm=' + EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪containing_barcode.fastq.gz',
                         'literal=' + DOWNSTREAM_FLANKING,
                         'k=20',
                         'hdist=1',
                         'ow=t'],
                         stderr_file=STATS_PATH + '/' + SAMPLE_NAME + '-' +␣
↪EPITOPE_NAME + '.containing_barcode.stats.txt')

            ## 5. Orient reads in same direction ##
            #print("orienting reads")

            # orient reads by mapping to reference
            # use SIV reference with accession M33262
            run_command([BBMAP_PATH + '/bbmap.sh',
```

```
                        'in=' + EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪containing_barcode.fastq.gz',
                        'outm=' + EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪mapped.sam',
                        'ref=' + REF_FASTA,
                        'ow=t'],
                        stderr_file=STATS_PATH + '/' + SAMPLE_NAME + '-' +␣
↪EPITOPE_NAME + '.mapping.stats.txt')

            # split mapped SAM file into plus, minus, and unmapped strand SAM␣
↪temporary files
            PLUS_STRAND_SAM = create_temp_file()
            MINUS_STRAND_SAM = create_temp_file()
            UNMAPPED_STRAND_SAM = create_temp_file()

            #print("Splitting sam")
            run_command([BBMAP_PATH + '/splitsam.sh',
                        EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.mapped.sam',
                        PLUS_STRAND_SAM[1] + '.sam',
                        MINUS_STRAND_SAM[1] + '.sam',
                        UNMAPPED_STRAND_SAM[1] + '.sam'])

            # reformat plus and minus strand SAM files to FASTQ
            PLUS_STRAND_FASTQ = create_temp_file()
            MINUS_STRAND_FASTQ = create_temp_file()

            #print("reformatting files - plus strand")
            run_command([BBMAP_PATH + '/reformat.sh',
                        'in=' + PLUS_STRAND_SAM[1] + '.sam',
                        'out=' + PLUS_STRAND_FASTQ[1] + '.fastq.gz'])
            #print("reformatting files - minus strand")
            run_command([BBMAP_PATH + '/reformat.sh',
                        'in=' + MINUS_STRAND_SAM[1] + '.sam',
                        'out=' + MINUS_STRAND_FASTQ[1] + '.fastq.gz',
                        'rcomp=t'])

            # concatenate oriented plus and minus strand FASTQ
            run_command(['cat',
                        PLUS_STRAND_FASTQ[1] + '.fastq.gz',
                        MINUS_STRAND_FASTQ[1] + '.fastq.gz'],
                        stdout_file = EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪oriented.fastq.gz')

            ## 6. Trim reads to contain only the barcode ##
            # remove upstream flanking sequence from oriented reads
            UPSTREAM_FLANK_REMOVED = create_temp_file()
```

```
            run_command([BBMAP_PATH + '/bbduk.sh',
                         'in=' + EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪oriented.fastq.gz',
                         'out=' + UPSTREAM_FLANK_REMOVED[1] + '.fastq.gz',
                         'literal=' + UPSTREAM_FLANKING,
                         'ktrim=l',
                         'rcomp=f',
                         'k=20',
                         'hdist=1',
                         'ow=t'],
                         stderr_file = STATS_PATH + '/' + SAMPLE_NAME + '-' +␣
↪EPITOPE_NAME + '.upstream.flank.removal.stats.txt')

            # remove downstream flanking sequence from oriented reads
            run_command([BBMAP_PATH + '/bbduk.sh',
                         'in=' + UPSTREAM_FLANK_REMOVED[1] + '.fastq.gz',
                         'out=' + EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪barcodes.fastq.gz',
                         'literal=' + DOWNSTREAM_FLANKING,
                         'ktrim=r',
                         'rcomp=f',
                         'k=20',
                         'hdist=1',
                         'minlength=24', # initially these were set to 27 (for a␣
↪9mer epitope)
                         'maxlength=33', # This is now set to look for 8-11mers
                         'ow=t'],
                         stderr_file = STATS_PATH + '/' + SAMPLE_NAME + '-' +␣
↪EPITOPE_NAME + '.downstream.flank.removal.stats.txt')


            ## 7. Count number of times each barcode occurs ##

            # count number of identical barcodes in each sample
            # create temporary count file
            BARCODE_COUNT_TEMP = create_temp_file()

            run_command([BBMAP_PATH + '/kmercountexact.sh',
                  'in=' + EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.barcodes.
↪fastq.gz',
                  'out=' + BARCODE_COUNT_TEMP[1] + '.txt',
                  'fastadump=f',
                  'k=' + epitope_seqs['epitopeLength'][e], # Added by RVM␣
↪because of different epitope lengths
                  'rcomp=f',
                  'ow=t'])
```

```
            ## 8. Create TSV file with barcode counts for each sequence ##
            # create three column barcode count file with sample name

            create_barcode_tsv(BARCODE_COUNT_TEMP[1] + '.txt',
                               EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '-' +␣
↪EPITOPE_NAME +'.barcode_counts.txt',
                               SAMPLE_NAME)

            #print("Epitope " + EPITOPE_NAME + " for sample " + SAMPLE_NAME + "␣
↪txt file is complete!")

            # remove temporary files
            PLUS_STRAND_SAM[0].close()
            PLUS_STRAND_FASTQ[0].close()
            MINUS_STRAND_SAM[0].close()
            MINUS_STRAND_FASTQ[0].close()
            UNMAPPED_STRAND_SAM[0].close()
            UPSTREAM_FLANK_REMOVED[0].close()
            BARCODE_COUNT_TEMP[0].close()

            # remove intermediate files
            remove_file(EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪containing_upstream_flank.fastq.gz')
            remove_file(EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.
↪containing_barcode.fastq.gz')
            remove_file(EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.oriented.fastq.
↪gz')
            remove_file(EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.barcodes.fastq.
↪gz')
            #this is the end of the epitope-specific loop

        remove_file(EPITOPE_OUT_PATH + '/' + SAMPLE_NAME + '.mapped.sam')
```

0.5 Reformat the barcode .txt files to organized .csv files

0.5.1 Functions used for .txt file reorganization

[ ]:
```
def remove_empty_files(files):
    """ Take the list of files that were generated through a looped version of␣
↪something Dave made, then
        make a list of files that have sufficient (arbitrary) amount of reads,␣
↪mark those that have epitopes
```

```
        but don't make thresholds, and those that are completely empty

        Args:
            files = list of .txt files output
        Returns:
            four items: one list of the files that had sufficient njumber of␣
↪epitopes, one that has files with
                low number of sequences, one of empties, and then one that␣
↪lists how many sequences are in each
                and every file so we can plot it.
    """

    import pandas as pd

    contains_seqs = []
    empties = []
    low_seqs = []
    num_seqs = []
    for f in range(len(files)):
        df = pd.read_csv(files[f], sep = '\t')
        num_seqs.append(int(pd.DataFrame.sum(df['barcode_count'])))
        if len(df) > 0:
            if pd.DataFrame.sum(df['barcode_count']) > 1000:
                contains_seqs.append(files[f])
            else:
                low_seqs.append(files[f])
        else:
            empties.append(files[f])
    print("There are " + str(len(contains_seqs)) + " files with 1000+ epitope␣
↪sequences detected, \n" +
          str(len(low_seqs)) + " with <1000 epitope sequences detected, \nand "␣
↪+
          str(len(empties)) + " empty files.")
    return contains_seqs, low_seqs, empties, num_seqs

def identify_animals(files):
    """Let's figure out which animals have sequences with epitopes so we can␣
↪later make a list of them

        Args:
            files = list of .txt files from your data set

        Returns:
            list of animals that have been sequenced
    """
    import re
```

```
    animals = set()
    for f in files:
        animals.add(re.search(ANIMAL_PATTERN, f)[0])
    return list(animals)

def translate_and_group(file):
    """Since we care more about the effet on the translated product, translate␣
 ↪the sequences and then combine
       identical amino acid sequences.

       Args:
           file = the file you want translated, in .txt form

       Returns:
           pandas data frame with translated sequences and corresponding counts
    """
    from Bio import Seq

    df = pd.read_csv(file, sep = '\t')
    nts = []
    aas = []
    for seq in df['barcode_sequence']:
        if len(seq) % 3 == 0:
            nts.append(seq)
            transl = Bio.Seq.translate(seq)
            aas.append(transl)
        else:
            print(file + " has a nt sequence length of " + str(len(seq)) + ": "␣
 ↪+ seq)

    df['barcode_sequence'] = pd.DataFrame(aas)
    df.insert(1, 'nt_sequence', nts)

    return df.groupby(['barcode_sequence', 'nt_sequence'], as_index=False).sum()

def sort_by_animal(transl_files, animal_list):
    """Make lists of all files from each animal and return one list of sorted␣
 ↪lists.
       Args:
           transl_files = made from translate_and_group, list of translated␣
 ↪sequences and the counts
           animal_list = made from identify_animals, sets up the unique animals␣
 ↪in the data set
       Returns:
           sorted list of lists for each unique animal
    """
    import re
```

```
    sorted_list = []
    for a in animal_list:
        byanimal = []
        for t in transl_files:
            if re.search(ANIMAL_PATTERN, t[0])[0] == a:
                byanimal.append(t)
        sorted_list.append(byanimal)
    return sorted_list

def write_translated_csv(sorted_file):
    """Take a file from the sorted list and write it as a .csv

        Args:
            sorted_list_name = file from the list, sorted by animal, that you␣
 ↪want replicate from

        Returns:
            csv of the translated file
    """
    import re

    an_id = re.search(ANIMAL_PATTERN, sorted_file[0])
    epitope = sorted_file[0].split('-')[3]
    filename = str(an_id) +  "_" + str(epitope[0:epitope.index('.')]) + "_" +␣
 ↪exp_id + ".csv"
    sorted_file[1].to_csv(filename)

def find_wildtype(dataframe):
    """Since we don't know what the wild type (SIVmac239) epitope sequences are␣
 ↪off the top of our head,
       we can go through and identify which epitopes are and are not the wild␣
 ↪type sequence

        Args:
            dataframe = a pandas data frame that has been filtered based on␣
 ↪frequency

        Returns:
            a data frame with the wild type epitope indicated, with a row added␣
 ↪to the
            data frame in the case that the wild type sequence is not present

    """
    epitope_id = dataframe['epitope'][0]
    epitopes_found = dataframe['barcode_sequence'].tolist()
```

```
    wt_seq = epitope_seqs[epitope_seqs['EpitopeName'] ==␣
↪epitope_id]['EpitopeSeq'].tolist()[0]

    nts_found = dataframe['nt_sequence'].tolist()
    wt_nts = epitope_seqs[epitope_seqs['EpitopeName'] ==␣
↪epitope_id]['epitopeNtSeq'].tolist()[0]
    wt_nts = wt_nts.replace("U", "T")

    if wt_seq in epitopes_found:
        # If there are mutations in the WT nucleotide sequence, we need to make␣
↪sure that all of the WT epitope seqs are identified
        wt_indices = [i for i,d in enumerate(epitopes_found) if d==wt_seq]
        for w in wt_indices:
            new_seq = dataframe.loc[w, 'barcode_sequence'] + " (WT)"
            dataframe.loc[w, 'barcode_sequence'] = new_seq

    else:
        wt_index = len(epitopes_found)
        wt_row = {'barcode_sequence': wt_seq + " (WT)",
                        'nt_sequence': wt_nts,
                        'barcode_count': 0,
                        'animal': dataframe['animal'][0],
                        'sample': dataframe['sample'][0],
                        'epitope':epitope_id,
                        'dpi':dataframe['dpi'][0],
                        'rep':dataframe['rep'][0],
                        'freq':0}
        dataframe = dataframe.append(wt_row, ignore_index = True )

    if wt_nts in nts_found:
        nt_index = nts_found.index(wt_nts)
        new_seq = dataframe.loc[w, 'nt_sequence'] + " (WT)"
        dataframe.loc[w, 'nt_sequence'] = new_seq

    return dataframe
```

0.5.2 Translate epitope sequences, write translated .csv files, and filter out low-frequency epitope sequences.

[ ]:
```
import Bio
import glob
import re

txt_files = [f for f in glob.glob(FASTQ_FOLDER_PATH + "/EpitopeFiles/*.txt")]
viable_files = remove_empty_files(txt_files)[0]
```

```
translated_files = []
for v in viable_files:
    translated_files.append((v, translate_and_group(v)))

animals = identify_animals(txt_files)
sorted_list = sort_by_animal(translated_files, animals)
for s in sorted_list:
    for t in s:
        t_sub = t[0][t[0].rfind("/") +1:]
        t[1]['epitope'] = t_sub[t_sub.rfind("-")+1:t_sub.index("barcode_counts.
 ↪txt")-1]
        t[1]['dpi'] = "None" if re.search("d\d{1,3}", t_sub) is None and re.
 ↪search("\d{1,3}dpi", t_sub) is None else re.search("d\d{1,3}|\d{1,3}dpi",␣
 ↪t_sub)[0]
        t[1]['sample'] = t_sub[t_sub.index("-")+1:t_sub.index("rep")-1]
        t[1]['rep'] = re.search("rep\d{1}", t_sub)[0]
        t[1]['freq'] = t[1]['barcode_count']/t[1]['barcode_count'].sum()
        t[1]['animal'] = re.search(ANIMAL_PATTERN, t_sub)[0]
        t[1].sort_values('barcode_count', ascending=False)
        t[1].to_csv(t[0][:t[0].index(".barcode")] + "_" + exp_id + ".csv")
```

[ ]:
```
# Here we are filtering out any epitope sequence that comprises less than 1% of␣
 ↪the total frequency. These will be
# grouped into an "other" group. The wild type (SIVmac239) sequence is also␣
 ↪indicated.
sorted_list = sort_by_animal(translated_files, animals)

filtered_list = []
merged_list = []
for s in sorted_list[0:1]:
    filtered_animal = []
    for t in s:
        df = t[1].loc[t[1]['freq'] > 0.01].copy()
        other_row = {'barcode_sequence': 'other_0.01',
                     'nt_sequence': 'other_0.01',
                     'barcode_count': sum(t[1]['barcode_count']) -␣
 ↪sum(df['barcode_count']),
                     'animal': t[1]['animal'][0],
                     'epitope': t[1]['epitope'][0],
                     'dpi': t[1]['dpi'][0],
                     'sample': t[1]['sample'][0],
                     'rep': t[1]['rep'][0],
                     'freq': 1 - sum(df['freq'])
                    }
        df = df.append(pd.Series(other_row), ignore_index=True)
        filtered_name = t[0][0:t[0].index(".")] + "_" + exp_id + ".filtered.csv"
```

```
        df = find_wildtype(df)
        df.to_csv(filtered_name)
        filtered_animal.append(df)

    if len(filtered_animal) > 0:
        merged_animal = pd.concat(filtered_animal)
        merged_animal_name = FASTQ_FOLDER_PATH + "/EpitopeFiles/" +␣
↪t[1]['animal'][0] + "_" + exp_id + ".merged.filtered.csv"
        merged_animal.to_csv(merged_animal_name)
        merged_list.append(merged_animal)
```