0.1 This notebook is intended to calculate the positions of primers in an alignment, using functions from PrimerProspector.

0.1.1 Import the needed functions, and define the primer sequences

```
# Code modified from PrimerProspector library slice_aligned_region.py (development
    version)

# Imports and definitions
from string import lower, upper
from operator import itemgetter

from cogent import LoadSeqs, DNA
from cogent.core.alphabet import AlphabetError
from cogent.align.align import make_dna_scoring_dict, local_pairwise
from cogent.parse.fasta import MinimalFastaParser
from cogent.core.moltype import IUPAC_DNA_ambiguities

DNA_CODES = ['A', 'C', 'T', 'G', 'R', 'Y', 'M', 'K',
 'W', 'S', 'B', 'D', 'H', 'V', 'N']


# Note that these are all written 5'->3', the reverse primers are reverse complemented
    for the local alignment

# If one wanted to test different primers, they would be defined here.

# 27f/338r = V2 (also includes V1, but generally just referred to as V2)
# 349f/534r = V3
# 515f/806r = V4
# 967f/1046r = V6
# 1391f/1492r = V9

primer_seqs = {
          '27f':'AGAGTTTGATCMTGGCTCAG',
          '338r':DNA.rc('GCTGCCTCCCGTAGGAGT'),
          '349f':'GYGCASCAGKCGMGAAW',
          '534r':DNA.rc('ATTACCGCGGCTGCTGG'),
          '515f':'GTGCCAGCMGCCGCGGTAA',
          '806r':DNA.rc('GGACTACVSGGGTATCTAAT'),
          '967f':'CAACGCGAAGAACCTTACC',
          '1048r':DNA.rc('CGRCRGCCATGYACCWC'),
          '1391f':'TGYACACACCGCCCGTC',
          '1492r':DNA.rc('GGCTACCTTGTTACGACTT'),
          '1391r':'TGYACACACCGCCCGTC' # Need this rather than forward primer to get
              proper 3' position of reverse version
          }

reference_aligned_file = '/home/ubuntu/qiime_software/gg_otus-4feb2011-release/rep_set/
    gg_76_otus_4feb2011_aligned.fasta'
```

0.1.2 Functions from the PrimerProspector code are here.

Should only be calling `get_primer_indices_counts` and `get_final_indices` directly to get aligned indices. Takes a while to run, as this has to do many local alignments for each primer (and takes the mode of the index in the alignment as the most accurate aligned index).

```
# Begin functions

""" This library was written for the purpose of slicing out a region of a reference
    sequence set to
match a target region amplified by a set of primers to improve taxonomic classification
    as seen in
Werner et al. It has been modified here to simply return the positions (3' end of the
    primer) in the
alignment of the given primer pair, as well as the position of simulated short reads of
    150, 250, and
400 base pairs. Note that some of these reads are larger than the size of the amplicon
    itself, and in
this case the simulated read simply isn't created."""

def match_scorer_ambigs(match=1,
                    mismatch=-1,
                    matches=None):
    """ Alternative scorer factory for sw_align, allows match to ambiguous chars

    It allows for matching to ambiguous characters which is useful for
     primer/sequence matching. Not sure what should happen with gaps, but they
     shouldn't be passed to this function anyway. Currently a gap will only match
     a gap.

    match and mismatch should both be numbers. Typically, match should be
    positive and mismatch should be negative.

    Resulting function has signature f(x,y) -> number.

    match: score for nucleotide match
    mismatch: score for nucleotide mismatch
    matches: dictionary for matching nucleotides, including degenerate bases
    """

    matches = matches or \
     {'A':{'A':None},'G':{'G':None},'C':{'C':None},\
      'T':{'T':None},'-':{'-':None}}
    for ambig, chars in IUPAC_DNA_ambiguities.items():
        try:
            matches[ambig].update({}.fromkeys(chars))
        except KeyError:
            matches[ambig] = {}.fromkeys(chars)

        for char in chars:
            try:
                matches[char].update({ambig:None})
```

```
            except KeyError:
                matches[char] = {ambig:None}

    def scorer(x, y):
        # need a better way to disallow unknown characters (could
        # try/except for a KeyError on the next step, but that would only
        # test one of the characters)
        if x not in matches or y not in matches:
            raise ValueError, "Unknown character: %s or %s" % (x,y)
        if y in matches[x]:
            return match
        else:
            return mismatch
    return scorer


def get_primer_mismatches(primer_hit,
                          target_hit,
                          sw_scorer=match_scorer_ambigs(1, -1)):
    """ Gets mismatches for a given primer sequence and target hit

    Specifically this returns total mismatches

    primer_hit: Alignment object for primer, normally matches primer unless
     gaps were used in the alignment
    target_hit: Alignment object for segment of sequence where primer was
     aligned to. Can contain gaps.
    sw_scorer: Gives scores for mismatches, gap insertions in alignment.

    """

    # Sum mismatches
    mismatches = 0
    for i in range(len(primer_hit)):
        # using the scoring function to check for
        # matches, but might want to just access the dict
        if sw_scorer(target_hit[i], primer_hit[i]) == -1:
            mismatches += 1

    return mismatches

def get_aligned_pos(unaligned_index, aligned_seq):
    """ Returns aligned index given index in degapped sequence

    unaligned_index: int with index in unaligned sequence
    aligned_seq: aligned sequence to find index"""

    nt_counter = 0
    total_counter = 0

    for nt in aligned_seq:
```

```
        if nt in DNA_CODES:
            nt_counter += 1
        total_counter += 1
        if nt_counter == unaligned_index:
            break

    return total_counter

def pair_hmm_align_unaligned_seqs(seqs,
                                  moltype=DNA,
                                  params={}):
    """
        Handles pairwise alignment of given sequence pair

        seqs: list of [primer, target sequence] in string format
        moltype: molecule type tested. Only DNA supported.
        params: Used to set parameters for opening, extending gaps and score
         matrix if something other than the default given in this function
         is desired.
    """

    seqs = LoadSeqs(data=seqs,moltype=moltype,aligned=False)
    try:
        s1, s2 = seqs.values()
    except ValueError:
        raise ValueError,\
         "Pairwise aligning of seqs requires exactly two seqs."

    try:
        gap_open = params['gap_open']
    except KeyError:
        gap_open = 5
    try:
        gap_extend = params['gap_extend']
    except KeyError:
        gap_extend = 2
    try:
        score_matrix = params['score_matrix']
    except KeyError:
        score_matrix = make_dna_scoring_dict(\
         match=1, transition=-1, transversion=-1)

    return local_pairwise(s1, s2, score_matrix, gap_open, gap_extend)

def local_align_primer_seq(primer,
                           sequence):
    """Perform local alignment of primer and sequence

        primer: Current primer being tested
        sequence: Current sequence
```

```
        Returns the Alignment object primer sequence and target sequence,
         and the start position in sequence of the hit.
    """


    query_sequence = sequence

    # Get alignment object from primer, target sequence
    alignment = pair_hmm_align_unaligned_seqs([primer,query_sequence])

    # Extract sequence of primer, target site, may have gaps in insertions
    # or deletions have occurred.
    primer_hit = str(alignment.Seqs[0])
    target_hit = str(alignment.Seqs[1])

    # Get index of primer hit in target sequence.
    try:
        hit_start = query_sequence.index(target_hit.replace('-',''))
    except ValueError:
        raise ValueError,('substring not found, query string %s, target_hit %s'\
         % (query_sequence, target_hit))


    return primer_hit, target_hit, hit_start

def get_primer_indices_counts(curr_fasta,
                              f_primer_seq,
                              r_primer_seq,
                              mismatch_threshold=3):
    """ Gets counts of primer hit indices for forward and reverse primer

    curr_fasta: current aligned fasta filepath
    f_primer_seq: forward primer sequence
    r_primer_seq: reverse primer sequence (already reverse complemented)
    mismatch_threshold: used to determine which primer hits are counted for
     getting the final aligned index"""

    f_primer_indices = {}
    r_primer_indices = {}

    for label, seq in MinimalFastaParser(open(curr_fasta), "U"):

            unaligned_seq = upper(seq.replace(".","").replace("-",""))

            primer_hit, target_hit, hit_start = \
             local_align_primer_seq(f_primer_seq, unaligned_seq)

            # get mismatches
            mismatches = get_primer_mismatches(primer_hit, target_hit)

            if mismatches <= mismatch_threshold:
```

```
                # Correction for forward primer to get 3' position
                aligned_pos = get_aligned_pos(hit_start + len(target_hit), seq)

                try:
                    f_primer_indices[aligned_pos] += 1
                except KeyError:
                    f_primer_indices[aligned_pos] = 0

            primer_hit, target_hit, hit_start = \
             local_align_primer_seq(r_primer_seq, unaligned_seq)

            # get mismatches
            mismatches = get_primer_mismatches(primer_hit, target_hit)

            if mismatches <= mismatch_threshold:
                # No correction for reverse primer hit
                aligned_pos = get_aligned_pos(hit_start, seq)

                try:
                    r_primer_indices[aligned_pos] += 1
                except KeyError:
                    r_primer_indices[aligned_pos] = 0

    return f_primer_indices, r_primer_indices

def get_final_indices(f_primer_indices,
                      r_primer_indices):
    """ Sorts, retrieves mode of aligning primer hits indices

    f_primer_indices: dictionary of primer_indices:counts
    r_primer_indices: dictionary of primer_indices:counts
    """

    f_primer_final = []
    r_primer_final = []

    for index in f_primer_indices:
        f_primer_final.append((index, f_primer_indices[index]))
    f_primer_final.sort(key=itemgetter(1), reverse=True)
    f_primer_index = f_primer_final[0][0]

    for index in r_primer_indices:
        r_primer_final.append((index, r_primer_indices[index]))
    r_primer_final.sort(key=itemgetter(1), reverse=True)
    r_primer_index = r_primer_final[0][0]


    return f_primer_index, r_primer_index

primer_indices = {}
```

```
primer_f_indices, primer_r_indices = get_primer_indices_counts(reference_aligned_file,
    primer_seqs['27f'], primer_seqs['338r'])
primer_indices['27f'], primer_indices['338r'] = get_final_indices(primer_f_indices,
    primer_r_indices)

primer_f_indices, primer_r_indices = get_primer_indices_counts(reference_aligned_file,
    primer_seqs['349f'], primer_seqs['534r'])
primer_indices['349f'], primer_indices['534r'] = get_final_indices(primer_f_indices,
    primer_r_indices)

primer_f_indices, primer_r_indices = get_primer_indices_counts(reference_aligned_file,
    primer_seqs['515f'], primer_seqs['806r'])
primer_indices['515f'], primer_indices['806r'] = get_final_indices(primer_f_indices,
    primer_r_indices)

primer_f_indices, primer_r_indices = get_primer_indices_counts(reference_aligned_file,
    primer_seqs['967f'], primer_seqs['1048r'])
primer_indices['967f'], primer_indices['1048r'] = get_final_indices(primer_f_indices,
    primer_r_indices)

primer_f_indices, primer_r_indices = get_primer_indices_counts(reference_aligned_file,
    primer_seqs['1391f'], primer_seqs['1492r'])
primer_indices['1391f'], primer_indices['1492r'] = get_final_indices(primer_f_indices,
    primer_r_indices)

primer_f_indices, primer_r_indices = get_primer_indices_counts(reference_aligned_file,
    primer_seqs['515f'], primer_seqs['1391r'])
primer_indices['515f'], primer_indices['1391r'] = get_final_indices(primer_f_indices,
    primer_r_indices)
```

0.1.3 Code for getting 150, 250, and 400 base pair "read" indices from the previously calculated primer indices. There will be some duplicates, such as the forward 150 reads from v2 and v2.v3 regions. We manually deleted the duplicates for our demo.

```
# Need to manually add the full length read, ('full.length', 0, 7682), which covers the
    entire Greengenes alignment.

region_boundaries = [
     ('v2', primer_indices['27f'], primer_indices['338r']),
     ('v2.v3', primer_indices['27f'], primer_indices['534r']),
     ('v2.v4', primer_indices['27f'], primer_indices['806r']),
     ('v2.v6', primer_indices['27f'], primer_indices['1048r']),
     ('v2.v8', primer_indices['27f'], primer_indices['1391r']),
     ('v2.v9', primer_indices['27f'], primer_indices['1492r']),
     ('v3', primer_indices['349f'], primer_indices['534r']),
     ('v3.v4', primer_indices['349f'], primer_indices['806r']),
     ('v3.v6', primer_indices['349f'], primer_indices['1048r']),
     ('v3.v8', primer_indices['349f'], primer_indices['1391r']),
     ('v3.v9', primer_indices['349f'], primer_indices['1492r']),
```

```
      ('v4', primer_indices['515f'], primer_indices['806r']),
      ('v4.v6', primer_indices['515f'], primer_indices['1048r']),
      ('v4.v8', primer_indices['515f'], primer_indices['1391r']),
      ('v4.v9', primer_indices['515f'], primer_indices['1492r']),
      ('v6', primer_indices['967f'], primer_indices['1048r']),
      ('v6.v8', primer_indices['967f'], primer_indices['1391r']),
      ('v6.v9', primer_indices['967f'], primer_indices['1492r']),
      ('v9', primer_indices['1391f'], primer_indices['1492r'])
]

region_index = 0
start_index = 1
end_index = 2

# Print in easy format to generate a list
for curr_region in region_boundaries:
    print "('%s', %d, %d)," % (curr_region[region_index], curr_region[start_index],
        curr_region[end_index])

# Manually print full length alignment
print "('full.length', 0, 7682),"

ix_sizes = [150, 250, 400]

names_ix = ['.150', '.250', '.400']

# We used the first sequence in the 97% clustered file for the demonstration, so using
    the
# same file here to recreate exact results
reference97_aligned_file = '/home/ubuntu/qiime_software/gg_otus-4feb2011-release/rep_set
    /gg_97_otus_4feb2011_aligned.fasta'

f = open(reference97_aligned_file, "U")

for label, seq in MinimalFastaParser(f):
    curr_seq = seq
    break

f.close()

nts = ['A', 'T', 'C', 'G', 'N']


for curr_region in region_boundaries:

    for forward_ix in range(len(ix_sizes)):
        curr_name = curr_region[region_index]
        curr_name += names_ix[forward_ix]

        start_ix = curr_region[start_index]
        end_ix = curr_region[end_index]
```

```
        curr_slice = curr_seq[start_ix:end_ix]


        counter = 0
        target_count = ix_sizes[forward_ix]

        curr_forward_ix = start_ix
        for nt in curr_slice:
            if nt in nts:
                counter += 1
            curr_forward_ix += 1
            # Skip if amplicon is smaller than current read size
            if curr_forward_ix == end_ix:
                break
            if counter == target_count:
                print "('%s', %d, %d)," % (curr_name, start_ix, curr_forward_ix)
                break
```

```
('v2', 136, 1868),
('v2.v3', 136, 2232),
('v2.v4', 136, 4051),
('v2.v6', 136, 4932),
('v2.v8', 136, 6426),
('v2.v9', 136, 6791),
('v3', 1916, 2232),
('v3.v4', 1916, 4051),
('v3.v6', 1916, 4932),
('v3.v8', 1916, 6426),
('v3.v9', 1916, 6791),
('v4', 2263, 4051),
('v4.v6', 2263, 4932),
('v4.v8', 2263, 6426),
('v4.v9', 2263, 6791),
('v6', 4653, 4932),
('v6.v8', 4653, 6426),
('v6.v9', 4653, 6791),
('v9', 6450, 6791),
('full.length', 0, 7682),
('v2.150', 136, 702),
('v2.250', 136, 1752),
('v2.v3.150', 136, 702),
('v2.v3.250', 136, 1752),
('v2.v3.400', 136, 2036),
('v2.v4.150', 136, 702),
('v2.v4.250', 136, 1752),
('v2.v4.400', 136, 2036),
('v2.v6.150', 136, 702),
('v2.v6.250', 136, 1752),
('v2.v6.400', 136, 2036),
('v2.v8.150', 136, 702),
('v2.v8.250', 136, 1752),
```

```
('v2.v8.400', 136, 2036),
('v2.v9.150', 136, 702),
('v2.v9.250', 136, 1752),
('v2.v9.400', 136, 2036),
('v3.v4.150', 1916, 2235),
('v3.v4.250', 1916, 2493),
('v3.v4.400', 1916, 4014),
('v3.v6.150', 1916, 2235),
('v3.v6.250', 1916, 2493),
('v3.v6.400', 1916, 4014),
('v3.v8.150', 1916, 2235),
('v3.v8.250', 1916, 2493),
('v3.v8.400', 1916, 4014),
('v3.v9.150', 1916, 2235),
('v3.v9.250', 1916, 2493),
('v3.v9.400', 1916, 4014),
('v4.150', 2263, 3794),
('v4.250', 2263, 4046),
('v4.v6.150', 2263, 3794),
('v4.v6.250', 2263, 4046),
('v4.v6.400', 2263, 4574),
('v4.v8.150', 2263, 3794),
('v4.v8.250', 2263, 4046),
('v4.v8.400', 2263, 4574),
('v4.v9.150', 2263, 3794),
('v4.v9.250', 2263, 4046),
('v4.v9.400', 2263, 4574),
('v6.v8.150', 4653, 5085),
('v6.v8.250', 4653, 5903),
('v6.v8.400', 4653, 6419),
('v6.v9.150', 4653, 5085),
('v6.v9.250', 4653, 5903),
('v6.v9.400', 4653, 6419),
```

0.1.4 Example output and curation for final list of indices

Here is the raw output of the previous steps that can be easily put into the list base_region_boundaries in the main notebook:

```
('v2', 136, 1868),
('v2.v3', 136, 2232),
('v2.v4', 136, 4051),
('v2.v6', 136, 4932),
('v2.v8', 136, 6426),
('v2.v9', 136, 6791),
('v3', 1916, 2232),
('v3.v4', 1916, 4051),
('v3.v6', 1916, 4932),
('v3.v8', 1916, 6426),
('v3.v9', 1916, 6791),
('v4', 2263, 4051),
('v4.v6', 2263, 4932),
```

```
('v4.v8', 2263, 6426),
('v4.v9', 2263, 6791),
('v6', 4653, 4932),
('v6.v8', 4653, 6426),
('v6.v9', 4653, 6791),
('v9', 6450, 6791),
('full.length', 0, 7682),
('v2.150', 136, 702),
('v2.250', 136, 1752),
('v2.v3.150', 136, 702),
('v2.v3.250', 136, 1752),
('v2.v3.400', 136, 2036),
('v2.v4.150', 136, 702),
('v2.v4.250', 136, 1752),
('v2.v4.400', 136, 2036),
('v2.v6.150', 136, 702),
('v2.v6.250', 136, 1752),
('v2.v6.400', 136, 2036),
('v2.v8.150', 136, 702),
('v2.v8.250', 136, 1752),
('v2.v8.400', 136, 2036),
('v2.v9.150', 136, 702),
('v2.v9.250', 136, 1752),
('v2.v9.400', 136, 2036),
('v3.v4.150', 1916, 2235),
('v3.v4.250', 1916, 2493),
('v3.v4.400', 1916, 4014),
('v3.v6.150', 1916, 2235),
('v3.v6.250', 1916, 2493),
('v3.v6.400', 1916, 4014),
('v3.v8.150', 1916, 2235),
('v3.v8.250', 1916, 2493),
('v3.v8.400', 1916, 4014),
('v3.v9.150', 1916, 2235),
('v3.v9.250', 1916, 2493),
('v3.v9.400', 1916, 4014),
('v4.150', 2263, 3794),
('v4.250', 2263, 4046),
('v4.v6.150', 2263, 3794),
('v4.v6.250', 2263, 4046),
('v4.v6.400', 2263, 4574),
('v4.v8.150', 2263, 3794),
('v4.v8.250', 2263, 4046),
('v4.v8.400', 2263, 4574),
('v4.v9.150', 2263, 3794),
('v4.v9.250', 2263, 4046),
('v4.v9.400', 2263, 4574),
('v6.v8.150', 4653, 5085),
('v6.v8.250', 4653, 5903),
('v6.v8.400', 4653, 6419),
('v6.v9.150', 4653, 5085),
('v6.v9.250', 4653, 5903),
```

```
('v6.v9.400', 4653, 6419),
```

But there are some duplicate regions here, such as v2.150 and v2.v3.150. These can be manually removed to get this:

```
('v2', 136, 1868),
('v2.v3', 136, 2232),
('v2.v4', 136, 4051),
('v2.v6', 136, 4932),
('v2.v8', 136, 6426),
('v2.v9', 136, 6791),
('v3', 1916, 2232),
('v3.v4', 1916, 4051),
('v3.v6', 1916, 4932),
('v3.v8', 1916, 6426),
('v3.v9', 1916, 6791),
('v4', 2263, 4051),
('v4.v6', 2263, 4932),
('v4.v8', 2263, 6426),
('v4.v9', 2263, 6791),
('v6', 4653, 4932),
('v6.v8', 4653, 6426),
('v6.v9', 4653, 6791),
('v9', 6450, 6791),
('full.length', 0, 7682),
('v2.150', 136, 702),
('v2.250', 136, 1752),
('v2.v3.400', 136, 2036),
('v3.v4.150', 1916, 2235),
('v3.v4.250', 1916, 2493),
('v3.v4.400', 1916, 4014),
('v4.150', 2263, 3794),
('v4.250', 2263, 4046),
('v4.v6.400', 2263, 4574),
('v6.v8.150', 4653, 5085),
('v6.v8.250', 4653, 5903),
('v6.v8.400', 4653, 6419)
```

Which is the same subset of sequences used to define base_region_boundaries (copied from the main demonstration notebook):

```
('v2', 136, 1868), #27f-338r
('v2.v3', 136, 2232),
('v2.v4', 136, 4051),
('v2.v6', 136, 4932),
('v2.v8', 136, 6426),
('v2.v9', 136, 6791),
('v3', 1916, 2232), #349f-534r
('v3.v4', 1916, 4051),
('v3.v6', 1916, 4932),
('v3.v8', 1916, 6426),
('v3.v9', 1916, 6791),
('v4', 2263, 4051), #515f-806r
```

```
('v4.v6', 2263, 4932),
('v4.v8', 2263, 6426),
('v4.v9', 2263, 6791),
('v6', 4653, 4932), #967f-1048r
('v6.v8', 4653, 6426),
('v6.v9', 4653, 6791),
('v9', 6450, 6791), #1391f-1492r
('full.length', 0, 7682),  # Start 150, 250, 400 base pair reads
('v2.150', 136, 702),
('v2.250', 136, 1752),
('v2.v3.400', 136, 2036), # Skips reads that are larger than amplicon size
('v3.v4.150', 1916, 2235),
('v3.v4.250', 1916, 2493),
('v3.v4.400', 1916, 4014),
('v4.150', 2263, 3794),
('v4.250', 2263, 4046),
('v4.v6.400', 2263, 4574),
('v6.v8.150', 4653, 5085),
('v6.v8.250', 4653, 5903),
('v6.v8.400', 4653, 6419)
```