# Bactome, I: Python in DNA Fingerprinting

**Chin-How Lee[1], Kun-Cheng Lee[1], Jack Si-Hao Oon[1], Maurice HT Ling[1,2,3]**
[1]School of Chemical and Life Sciences, Singapore Polytechnic
[2]Department of Zoology, The University of Melbourne, Australia
[3]Corresponding author: mauriceling@acm.org

## Abstract

Bactome is a collection of Python functions to find primers suitable for DNA fingerprinting, determine restriction digestion profile, and analyse the resulting DNA fingerprint features as migration distance of the bands in gel electrophoresis. An actual use case will be presented as a case study. These codes are licensed under Lesser General Public Licence version 3.

## 1. Brief Case Scenario

Bactome is a collection of Python functions implemented out of need. Our research project in question aimed to examine the genetic changes of a bacterium, *Escherichia coli*, at different phases of an extended culture in a number of different chemicals (Lee et al., 2010). The genetic changes can be identified using DNA fingerprinting methods and can be visualized as different banding patterns on an agarose gel. The DNA fingerprinting method chosen is Polymerase Chain Reaction (PCR) (Welsh and McClelland, 1990), followed by Restriction Fragment Length Polymorphism (RFLP). PCR is a method used to multiply the number of DNA copies and its specificity (which part of the DNA to multiple) is determined by the primers. Hence, we need to know what primers to use. After PCR, the amplified DNA is cut using specific enzymes (known as restriction endonucleases) to generate a banding pattern (known as a DNA fingerprint). This step is known as RFLP and the types of enzymes to use, within a collection of more than 500, determines the quality of the DNA fingerprint. After visualization, statistical analysis of the banding patterns to illustrate genetic distance is needed.

There are a number of tools available for selecting common primers to amplify different DNA templates, such as PrimerSNP (Yao et al., 2008) and SOP3v2 (Ringquist et al., 2005) or within a single template, such as Lowe et al. (1990), OSP (Hillier and Green, 1991), BatchPrimer3 (You et al., 2008) and ConservedPrimers 2.0 (You et al., 2009). However, none of them fits our requirements as our strategy calls for primers to generate a specific number of amplicons so that the resulting restriction endonuclease digestion will be resolvable, and of visually resolvable sizes so that the PCR product itself can be used as a fingerprint profile. Hence, we decided to develop Bactome from BioPython (Cock et al., 2009).

## 2. Primer Selection for Polymerase Chain Reaction

Primer design was performed to determine the suitable primer for Polymerase Chain Reaction (PCR). The primer sequence obtained is used as forward and reverse primer in each reaction. A string of DNA sequence is read using SeqIO class of BioPython (Cock et al., 2009).

```
def fasta_seq(fasta='ATCC8739.fasta'):
    """
    Open fasta format file and returns the sequence.
    """
    f = SeqIO.parse(open(fasta, 'rU'), 'fasta').next()
    return f.seq
```

Genome of the bacteria was scanned and cut into fragments of equal length.

```
def generate_fragments(seq, tablefile, fragment_size=15):
    """
    Slice an entire sequence into fragments of equal lengths

    Parameters:
       seq = sequence to be sliced (output from fasta_seq function)
            tablefile = name of file to store the sliced sequence
       fragment_size = length of each sequence to be sliced into.
            This will be the maximum primer/probe length.
            Default value is 15 bases.
    """
    genome_fragment = anydbm.open(tablefile, 'c')
    last_start= len(seq) - fragment_size
    count = 0
    for start in range(0, last_start, fragment_size):
        genome_fragment[str(count)] = seq[start:15+start].data
        count = count + 1
        if (count % 1000) == 0:
            print str(count) + ' fragments inserted into ' \
                  + tablefile
    print str(count) + ' fragments inserted into ' + tablefile
    genome_fragment.close()
```

The longest common substring (LCS) can be defined as the longest string that is found in two or more strings (Bergroth et al., 2000) which is used to identify common DNA sequences flanking a length of DNA. The length of DNA to be flanked determines the size of amplicon (number of intervening fragments).

For example in a case scenario in Figure 1, the DNA flanked is 15 base pairs and is represented by one block.

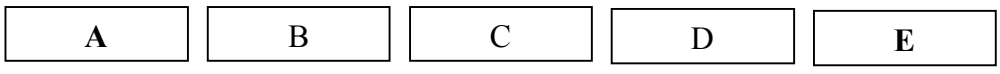



Figure 1: Schematic diagram to illustrate the amplicon size determination with respect to the length of flanked DNA.

Given that after LCS determination, block A and the inverse complement of block E contain a LCS, the sequence of A and the inverse complement of E can be used as primers. Therefore, the size of the amplicon would be 15 x 3 (B, C and D) = 45 base pairs.

A minimum length of LCS needed is determined biologically as the LCS is to be used as primers for PCR (Dieffenbach et al., 1993). The LCS results are tabulated as a tuple of start fragment number, end fragment position and primer sequence.

```
def LCS(seq, test):
    """
    Finds the longest common substring between the 2 inputs (seq and
    test).

    Adapted from http://en.wikibooks.org/wiki/Algorithm
    _Implementation/Strings/Longest_common_substring
    """
    m = len(seq)
    n = len(test)
    L = [[0] * (n+1) for i in xrange(m+1)]
    LCS = set()
    longest = 0
    for i in xrange(m):
        for j in xrange(n):
            if seq[i] == test[j]:
                v = L[i][j] + 1
                L[i+1][j+1] = v
                if v > longest:
                    longest = v
                    LCS = set()
                if v == longest:
                    LCS.add(seq[i-v+1:i+1])
    return LCS

def generate_all_LCS(fragfile, LCSfile, min_primer=7,
                     max_interval=198, min_interval=6):
    """
    Generates primers where forward primer sequence = reverse primer
    Sequence given the size of amplicon and minimum length of
    primers.

    Parameters:
        fragfile = file of sliced sequence (tablefile parameter of
            generate_fragments function)
        LCSfile = name of file to store least common substrings'
            (primers) data
        min_primer = smallest acceptable length for primers
        max_interval = number of maximum fragment intervals between
            2 primers. This value is determines the maximum length
            of amplicon generated and is determined by the length of
            each sliced fragment.
            Default value is 198 as 3000bp amplicon is the maximum
            Normal Taq polymerase can amplify. 198 intervals of 15
            bases per fragment + 2 flanking fragment of maximum of
            15 bases (primer length) gives a total of 200 fragments
```

```
            of 15 bases = 3000 bases.
        min_interval = number of minimum fragment intervals between
            2 primers. This value is determines the maximum length
            of amplicon generated and is determined by the length of
            each sliced fragment.
            Default value is 6 as 120bp amplicon is the minimum
            length visible on 1.5% agarose gel that is also suitable
            for 3000 bp. 6 intervals of 15 bases per fragment + 2
            flanking fragment of maximum of 15 bases (primer length)
            gives a total of 8 fragments of 15 bases = 120 bases.
    """
    genome_fragment = anydbm.open(fragfile, 'c')
    num_of_frag = len(genome_fragment)
    num_analyzed = 0
    result = {}
    fragment_size = len(genome_fragment['0'])
    for size in range(max_interval, min_interval, -1):
        com_seq = anydbm.open(LCSfile, 'c')
        print 'Analysing for ' + str(fragment_size * size) + \
              'bp amplicons'
        num_com = 0
        for frag in range(0, num_of_frag - size):
            p1 = genome_fragment[str(frag)]
            p2 = genome_fragment[str(frag+size)]
            p2 = Seq(p2, generic_dna).reverse_complement().data
            try: common_seq = LCS(p1, p2).pop()
            except : common_seq = ''
            num_analyzed = num_analyzed + 1
            if len(common_seq) > min_primer:
                com_seq['|'.join([str(frag), str(frag+size),
                                  str(size)])] = common_seq
                num_com = num_com + 1
            if (num_analyzed % 10000) == 0:
                print str(num_analyzed) + ' pairs analyzed'
                print '     ' + str(num_com) + ' LCS more than ' + \
                      str(min_primer) + 'bp for ' + \
                      str(15 * size) + 'bp amplicons'
        com_seq.close()
        result[size] = str(num_com)
    print str(num_analyzed) + ' pairs analyzed'
    print '     ' + str(num_com) + ' LCS more than ' +
       str(min_primer)+ 'bp for ' + str(15 * size) + 'bp amplicons'
    genome_fragment.close()
    return result
```

For easier analysis of the primer sequence, Inversion of the LCS list (index of LCS) was performed to classify into primer sequence with start fragment position, end fragment position and amplicon size. This requires *marshaldbm* class (Ling, 2010), an object-marshallable *anydbm* module that enables objects to be stored as values in an *anydbm* file.

```
def get_LCS_by_amplicon(LCSfile, amplicon_size):
    """
    Extract all the primers and amplicon positions from the LCSfile
    given the amplicon length.
```

```
    Parameters:
        LCSfile = file of least common substrings' (primers) data
        amplicon_size = size of amplified product in terms of number
            of fragments intervening 2 primers. For example, if the
            fragment size is 15 and the amplicon size is 60, we are
            then looking for primers (forward primer sequence =
            reverse primer sequence) that gives amplicons of 900 bp
            (60*15) excluding the flanking primers.

    Returns:
        (List of primers,
        List of amplicons' data)
        where
        1. each data element in "List of amplicons' data" is in the
           format of '<start fragment position>|<end fragment
           position>|<amplicon size>', <start fragment position> and
           <end fragment position> multiplied by the fragment size
           will give the corresponding position on the fasta
           sequence. For example, '89572|89632|60' represents the
           amplicon from 1343580 bp (889572*15) to 1344480 bp
           (89632*15) of the fasta sequence, corresponding to an
           amplicon size of 900bp (60*15).
        2. the number of primers = number of primers's data. This
           means that in primers =
           get_LCS_by_amplicon('primer.table', 60) primers[0][10]
           sequence will amplify primers[1][10]
    """
    com_seq = anydbm.open(LCSfile, 'c')
    keys = com_seq.keys()
    return ([com_seq[x] for x in keys
             if x.split('|')[2] == str(amplicon_size)],
            [x for x in keys
             if x.split('|')[2] == str(amplicon_size)])

def inverse_LCS(LCSfile, inverseLCSfile, max_interval=198,
                min_interval=6, fragment_size=15):
    """
    Generates the index file of LCSfile.

    Format of LCSfile: <start fragment position>|
                       <end fragment position>|
                       <amplicon size> = <primer sequence>
    Format of inverseLCSfile: <primer sequence> =
                              [<start fragment position>|
                               <end fragment position>|
                               <amplicon size>, ...]

    Parameters:
        LCSfile = file of least common substrings' (primers) data
        inverseLCSfile = name of file to store the index of LCSfile
        max_interval = number of maximum fragment intervals between
            2 primers. This value is determines the maximum length
            of amplicon generated and is determined by the length of
            each sliced fragment. Default value is 198 as 3000bp
            amplicon is the maximum normal Taq polymerase can
            amplify. 198 intervals of 15 bases per fragment + 2
            flanking fragment of maximum of 15 bases (primer length)
```

```
            gives a total of 200 fragments of 15 bases = 3000 bases.
        min_interval = number of minimum fragment intervals between
            2 primers. This value is determines the maximum length
            of amplicon generated and is determined by the length of
            each sliced fragment. Default value is 6 as 120bp
            amplicon is the minimum length visible on 1.5% agarose
            gel that is also suitable for 3000 bp. 6 intervals of 15
            bases per fragment + 2 flanking fragment of maximum of
            15 bases (primer length) gives a total of 8 fragments of
            15 bases = 120 bases.
        fragment_size = length of each sequence to be sliced into.
            This will be the maximum primer/probe length.
            Default value is 15 bases.
    """
    inverseLCSfile = marshaldbm(inverseLCSfile, 'c')
    count = 0
    for amplicon_size in range(max_interval, min_interval, -1):
        seq, pos = get_LCS_by_amplicon(LCSfile, amplicon_size)
        print 'Processing for amplicon size of ' + \
            str(amplicon_size*fragment_size)
        for index in range(len(seq)):
            count = count + 1
            try:
                temp = inverseLCSfile[seq[index]]
                inverseLCSfile[seq[index]] = temp + [pos[index]]
            except KeyError:
                inverseLCSfile[seq[index]] = [pos[index]]
            if (count % 1000) == 0:
                print str(count) + ' LCS processed'
    print str(count) + ' LCS processed'
    inverseLCSfile.close()
```

This function, *generate_primers_from_fasta*, wraps around the above function as a one-step solution to generate all possible primers (LCS).

```
def generate_primers_from_fasta(fasta='ATCC8739.fasta',
                        genome_fragment='genome_fragment.table',
                        max_primer_length=15,
                        min_primer_length=7,
                        max_amplicon_length=3100,
                        min_amplicon_length=300,
                        primer_file='primer.table',
                       inverse_primer_file='invprimer.table'):
    """
    Generate a file of primers with amplicon size and genomic
    position from the Fasta sequence file.

    Parameters:
        fasta = name of fasta file to process
        genome_fragment = name of file to store the sliced sequence.
            Default = 'genome_fragment.table'
        max_primer_length = maximum length of primer (fragment size
            of genome slices). Default = 15 bp
        min_primer_length = minimum length of primer. Default = 7 bp
        max_amplicon_length = maximum size of amplicon. Default =
            3100 bp
```

```
        min_amplicon_length = minimum size of amplicon. Default =
            300 bp
        primer_file = file of least common substrings' (primers)
            data. Default = 'primer.table'
        inverse_primer_file = name of primers index file.
            Default = 'invprimer.table'

    Output files:
        1. genome fragment file from 'generate_fragments' function
        2. primer file from 'generate_all_LCS' function
        3. inverse primer file from 'inverse_LCS' function
    """
    seq = fasta_seq(fasta)
    generate_fragments(seq, genome_fragment,
                       fragment_size=max_primer_length)
    generate_all_LCS(genome_fragment, primer_file,
                     min_primer_length,
                     int(max_amplicon_length/max_primer_length),
                     int(min_amplicon_length/max_primer_length))
    inverse_LCS(primer_file, inverse_primer_file,
                int(max_amplicon_length/max_primer_length),
                int(min_amplicon_length/max_primer_length),
                max_primer_length)
```

Once LCS index file is generated, suitable primers (LCS) can be searched based on the number of amplicons (dictionary values) and the melting point of the primer (deduced from the LCS).

```
def look_for_primers(inverseLCSfile, num_of_amplicons, min_tm,
                     p='yes'):
    """
    Scans the LCSfile index for primer(s) that amplifies a given
    number of amplicons. The primer must not end with adenosine and
    thymidine, and must meet minimum annealing temperature.

    Parameters:
        inverseLCSfile = name of LCSfile index file
        num_of_amplicons = number of amplicons generated by required
            primer(s)
        min_tm = minimum annealing temperature (in degrees
            centigrade) of primers. Calculated as 2AT + 4GC.
        p = flag to determine if data is to be printed. Default =
            yes

    Returns:
        List of primers meeting the criteria
    """
    f = marshaldbm(inverseLCSfile, 'c')
    keys = f.keys()
    primers = []
    for k in keys:
        temperature = 4*(k.count('G')+k.count('C')) + \
                      2*(k.count('A')+k.count('T'))
        if (len(f[k])) == num_of_amplicons and \
           temperature > min_tm-1 and \
           not k.endswith('A') and not k.endswith('T'):
```

```
            primers.append(k)
            if p == 'yes':
                print k
                print f[k]
                print
    return primers
```

We consider the possibility of cases whereby we are interested in primers that only give us one amplicon (*LCS_uniqueness*) or primers giving the same amplicon size (*LCS_tabulate*).

```
def LCS_uniqueness(LCSfile, amplicon_size):
    """
    Extract all the primers and amplicon positions from the LCSfile
    given the amplicon length and groups the primers into 2 groups:
    those that only yield one amplicon and those that yield more
    than one amplicons.

    Parameters:
        LCSfile = file of least common substrings' (primers) data
        amplicon_size = size of amplified product in terms of number
            of fragments intervening 2 primers. For example, if the
            fragment size is 15 and the amplicon size is 60, we are
            then looking for primers (forward primer sequence =
            reverse primer sequence) that gives amplicons of 900 bp
            (60*15) excluding the flanking primers.

    Returns:
        (List of unique primers,
        List of non-unique primers)
        where 'List of unique primers' are primers amplifying only
        one amplicon, and 'List of non-unique primers' is a tuple of
        'number of amplicons' and primer sequence.
    """
    LCS = get_LCS_by_amplicon(LCSfile, amplicon_size)[0]
    unique_LCS = [x for x in LCS
                  if LCS.count(x) == 1]
    non_unique_LCS = list(set([(LCS.count(x), x) for x in LCS
                               if LCS.count(x) > 1]))
    return (unique_LCS, non_unique_LCS)

def LCS_tabulate(LCSfile, max_interval=198, min_interval=6,
                 fragment_size=15, p='yes'):
    """
    Provides a tabulation of the given LCSfile, grouped by amplicon
    size.

    Parameters:
        LCSfile = file of least common substrings' (primers) data
        max_interval = number of maximum fragment intervals between
            2 primers. This value is determines the maximum length
            of amplicon generated and is determined by the length of
            each sliced fragment. Default value is 198 as 3000bp
            amplicon is the maximum normal Taq polymerase can
            amplify. 198 intervals of 15 bases per fragment + 2
            flanking fragment of maximum of 15 bases (primer length)
```

```
        gives a total of 200 fragments of 15 bases = 3000 bases.
    min_interval = number of minimum fragment intervals between
        2 primers. This value is determines the maximum length
        of amplicon generated and is determined by the length of
        each sliced fragment. Default value is 6 as 120bp
        amplicon is the minimum length visible on 1.5% agarose
        gel that is also suitable for 3000 bp. 6 intervals of 15
        bases per fragment + 2 flanking fragment of maximum of
        15 bases (primer length) gives a total of 8 fragments of
        15 bases = 120 bases.
    fragment_size = length of each sequence to be sliced into.
        This will be the maximum primer/probe length. Default
        value is 15 bases.
    p = flag to determine if the tabulation data is to be
        printed. Default = yes
"""
result = {}
for amplicon_size in range(max_interval, min_interval, -1):
    (unique_LCS, non_unique_LCS) = \
                LCS_uniqueness(LCSfile, amplicon_size)
    if p == 'yes':
        print 'Longest Common Substring (primer) for amplicon \
            size of ' + str(amplicon_size*fragment_size)
        print 'Unique Primers (only one amplicon). N = ' + \
            str(len(unique_LCS))
        print '  '.join(unique_LCS)
        print
        print 'Non-Unique Primers (more than one amplicon)'
        print '\t'.join(['# amplicons', 'Primer sequence'])
        for lcs in non_unique_LCS:
            print '\t\t'.join([str(lcs[0]), lcs[1]])
        print
        print
    result[str(amplicon_size)] = (unique_LCS, non_unique_LCS)
return result
```

## 3. Generating Virtual Restriction Digest Profile

Restriction enzyme cuts a specific region in the genome which can indicate the presence of certain sequence in the genome (a string of DNA letters). If there is mutation at the cutting region, the same restriction enzyme cannot cut the same site anymore as the sequence is changed resulting the decrease in number of fragment. In exchange, the restriction enzyme may be able to cut other regions in the genome results in more fragments generation and also the mutated cutting site can now be cut by another restriction site. This function “digests” the genome virtually which allows us to see the expected fragments and also tells us the restriction enzyme that is able to cut the genome, which in turn can be used to select restriction enzyme to purchase for following laboratory experiments. This function uses Biopython (Cock et al., 2009) for the restriction enzyme cutting site then it scans the cutting site in the genome. If it is able to find the cutting site in the genome this indicates the restriction enzyme is able to be used for RFLP. It returns the location and number of cutting site for the restriction enzyme and the length of fragment generated by

restriction enzyme. This tool can also be used for microsatellite and single nucleotide polymorphisms detection and gene mapping.

```
def restriction_digest(seq, enzyme, max_band=23130, min_band=2000,
                       linear=False, p='yes'):
    """
    Performs restriction endonuclease digestion on a sequence and
    group the resulting fragments into 3 groups so simulate
    different agarose gel electrophoresis:
    1. fragment length more than the maximum size
    2. fragment length between the maximum and minimum size
    3. fragment length less than the minimum size

    Parameters:
        seq = DNA sequence for restriction endonuclease digestion
        enzyme = Restriction endonuclease object from
            Bio.Restriction package
        max_band = size of maximum band in basepairs. Default =
            23130
        min_band = size of minimum band in basepairs. Default = 2000
        linear = flag to define if DNA sequence is linear.
            Default = False (DNA is circular)
        p = flag to determine if the data is to be printed. Default
            = yes

    Result:
        (Number of fragments after digestion,
        List of fragments with molecular size above max_band,
        List of fragments with molecular size between max_band and
        min_band, List of fragments with molecular size below
        min_band)
    """
    digest = enzyme.search(seq, linear=linear)
    digest.sort()
    fragment = [digest[x+1] - digest[x]
                for x in range(len(digest) - 1)]
    fragment.sort()
    ogel = [x for x in fragment if x > max_band]
    gel = [x for x in fragment if x <= max_band and x >= min_band]
    ugel = [x for x in fragment if x < min_band]
    ogel.sort()
    gel.sort()
    ugel.sort()
    if p == 'yes':
        print 'Enzyme: ' + str(enzyme)
        print 'Restriction site: ' + enzyme.site
        print 'Number of fragments: ' + str(len(fragment))
        print 'Number of fragments (x > ' + str(max_band) + '): ' \
              + str(len(ogel))
        print 'Number of fragments (' + str(max_band) + ' < x < ' \
              + str(min_band) + '): ' + str(len(gel))
        print 'Number of fragments (x < ' + str(min_band) + '): ' \
              + str(len(ugel))
        print
    return (len(fragment), ogel, gel, ugel)
```

```
def restriction_supplier(seq, max_band=23130, min_band=2000,
                         suppliers='ACEGFIHKJMONQPSRUVX',
                         linear=False, p='yes'):
    """
    Performs restriction endonuclease analysis by batch, based on
    supplier.

    Parameters:
        seq = DNA sequence for restriction endonuclease digestion
        max_band = size of maximum band in basepairs.

            Default = 23130
        min_band = size of minimum band in basepairs.
            Default = 2000
        suppliers = restriction enzyme supplier.
            Default = ACEGFIHKJMONQPSRUVX
            where
                A = Amersham Pharmacia Biotech
                C = Minotech Biotechnology
                E = Stratagene
                G = Qbiogene
                F = Fermentas AB
                I = SibEnzyme Ltd.
                H = American Allied Biochemical, Inc.
                K = Takara Shuzo Co. Ltd.
                J = Nippon Gene Co., Ltd.
                M = Roche Applied Science
                O = Toyobo Biochemicals
                N = New England Biolabs
                Q = CHIMERx
                P = Megabase Research Products
                S = Sigma Chemical Corporation
                R = Promega Corporation
                U = Bangalore Genei
                V = MRC-Holland
                X = EURx Ltd.
        linear = flag to define if DNA sequence is linear.
            Default = False (DNA is circular)
        p = flag to determine if the data is to be printed.
            Default = yes

    Returns:
    {Restriction endonuclease :
        (Total number of fragments after digestion,
        Number of fragments with molecular size above max_band,
        Number of fragments with molecular size between max_band and
        min_band,
        Number of fragments with molecular size below min_band)}
    """
    from Bio.Restriction import RestrictionBatch
    count = 0
    result = {}
    for enzyme in RestrictionBatch(first=[],
                                   suppliers=[x.upper()
                                          for x in suppliers]):
        try:
```

```
            digest = restriction_digest(seq, enzyme, max_band,
                                        min_band, linear, p)
        except MemoryError:
            print 'Memory Error during ' + str(enzyme) + \
                  ' digestion'
        result[str(enzyme)] = (digest[0], len(digest[1]),
                               len(digest[2]), len(digest[3]))
        count = count + 1
        if p != 'yes':
            if count % 10 == 0:
                print str(count) + ' restriction \
                      endonuclease processed'
    return result
```

## 4. Analysing DNA Fingerprint Profiles

This function analyses the migrated distance of the bands in gel electrophoresis using the method to calculate dissimilarity developed by Nei and Li (1979). Dissimilarity index = 1 - [2 x (number of regions where both species are present)/ [(2 x (number of regions where both species are present)) + (number of regions where only one species is present)]]. There are 2 types of comparison: set and list. Set comparison is used for processing set-based (unordered or nominal) distance of categorical data. List comparison is used for processing list-based (ordered or ordinal) distance of categorical data. For our experiment, list comparison is preferred. It compares the band migrated distance for one sample (original) to another sample (test). If the bands are similar it returns zero dissimilarity index. On the other hand, if the band at that certain distance only appears on of the sample, it will return the Dissimilarity Index (DI).

```
def setCompare(original, test, absent):
    """
    Used for processing set-based (unordered or nominal) distance of
    categorical data.

    Parameters:
        original: list of original data
        test: list of data to test against original
        absent: indicator to define absent data
    """
    original_only = float(len([x for x in original
                               if x not in test]))
    test_only = float(len([x for x in test if x not in original]))
    both = float(len([x for x in original if x in test]))
    return (original_only, test_only, both)

def listCompare(original, test, absent):
    """
    Used for processing list-based (ordered or ordinal) distance of
    categorical data.

    Parameters:
        original: list of original data
```

```
        test: list of data to test against original
        absent: indicator to define absent data
    """
    original = list(original)
    test = list(test)
    original_only = 0.0
    test_only = 0.0
    both = 0.0
    for i in range(len(original)):
        if original[i] == absent and test[i] == absent:
            pass
        elif original[i] == test[i]:
            both = both + 1
        elif original[i] <> absent and test[i] == absent:
            original_only = original_only + 1
        elif original[i] == absent and test[i] <> absent:
            test_only = test_only + 1
        else: pass
return (original_only, test_only, both)

def Nei_Li(original, test, absent=0, type='Set'):
    """
    Nei and Li Distance is distance measure for nominal or ordinal
    data.

    Given 2 lists (original and test), calculates the Nei and Li
    Distance based on the formula,

    1 - [2 x (number of regions where both species are present)/
    [(2 x (number of regions where both species are present)) +
    (number of regions where only one species is present)]]

    Nei M, Li WH (1979) Mathematical models for studying
    Genetic variation in terms of restriction endonucleases.
    Proc Natl Acad Sci USA 76:5269-5273

    Paraemeters:
        original: list of original data
        test: list of data to test against original
        absent: user-defined identifier for absent of region,
            default = 0
        type: (Dieffenbach et al., 1993), define whether use Set
comparison
           (unordered) or list comparison (ordered), default = Set
    """
    if type == 'Set':
        (original_only, test_only, both) = setCompare(original,
                                               test, absent)
    else:
        (original_only, test_only, both) = listCompare(original,
                                               test, absent)
    return 1-((2*both)/((2*both)+original_only+test_only))
```

## 5. Case Study

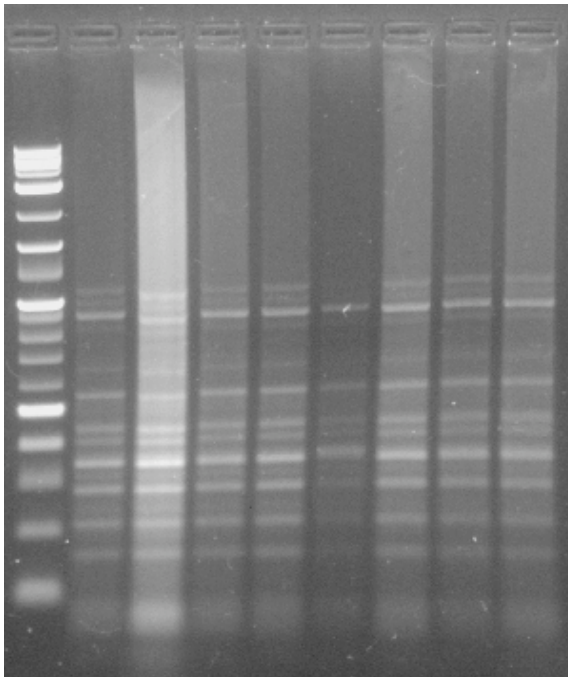
We developed Bactome to cater to the needs of our research work (Lee et al., 2010), briefly described in Section 1, on the effects of food additives on the genome of *Escherichia coli*, a common intestinal bacterium. Three additives were chosen for study, namely, sodium chloride (table salt), benzoic acid (BA; a common food preservative), and monosodium glutamate (MSG; a common taste enhancer). Two concentrations of each additives (H MSG, high MSG; L MSG, low MSG; H BA, high BA; L BA, low BA; H SALT, high salt; L SALT, low salt) and two combination additives (H COMB, combining all the high concentration additives; L COMB, combining all the high concentration additives) were used.

The strain of *E. coli* used was ATCC 8739. It was chosen as its entire genome had been sequenced which enables efficient primer selection for PCR-based DNA fingerprinting. The genome of E. coli ATCC 8739 is about 4.7 megabases.

The entire genome is processed by *generate_primers_from_fasta* function. Firstly, a total of 316414 fragments of 15 bases each is generated as the primer length for DNA fingerprinting ranges from 6 to 15 bases. Secondly, longest common substrings were identified as potential primers using a sliding window of 20 (300 bases) to 207 (3105 bases) fragment sizes. This generated a total of 38105 potential primers. Lastly, 8 primers were selected based on one of the two criteria – either the primer has a melting temperature of at least $35^{o}$C and yields 3 amplicons or a melting temperature of at least $33^{o}$C and yields 4 amplicons. The choice of temperature is to enable efficient PCR whereas the choice of number of amplicons is to enable resolution in the visualization after digestion by restriction endonuclease. Each amplicon can be virtually digested using *restriction_digestion* function and combined to simulate the resulting DNA fingerprint.

Each column in Figure 2 is a DNA fingerprint from one sample and each band in within the column represents a feature. Each DNA fingerprint can represented as a Python list of position of each bands. After which, DI can be used to calculate the distances between any 2 samples within all 8 samples as tabulated in Figure 3.

Computationally, the successful generation of DNA fingerprints using primers selected by Bactome demonstrated the reliability and correctness of implementation. Biologically, the distance calculated among any of the samples using DI suggested that *E. coli* underwent genetic changes after extended culture in the presence of MSG, benzoic acid and table salt, both singly and in combination. This is because the initial culture of the treatments originates from the same ancestor and therefore any increment in distance among the samples is indicative of genetic changes.



| Marker | H MSG | L MSG | H BA | L BA | H SALT | L SALT | H COMB | L COMB |
|---|---|---|---|---|---|---|---|---|
| 2.4 | 3.8 | | 3.8 | 3.8 | | 3.8 | 3.8 | 3.8 |
| 2.8 | 4.0 | 4.0 | | 4.0 | | | | |
| 3.2 | 4.1 | 4.1 | | | | | 4.1 | |
| 4.1 | 4.2 | 4.2 | 4.2 | 4.2 | 4.2 | 4.2 | 4.2 | 4.2 |
| 4.3 | 4.4 | | | 4.4 | | 4.4 | 4.4 | |
| 4.6 | 5.0 | 5.0 | 5.0 | 5.0 | | 5.0 | 5.0 | 5.0 |
| 5.7 | 5.4 | 5.4 | 5.4 | 5.4 | 5.4 | 5.4 | 5.4 | 5.4 |
| | 5.9 | 5.9 | 5.9 | 5.9 | 5.9 | 5.9 | 5.9 | 5.9 |
| | 6.0 | 6.0 | 6.0 | 6.0 | 6.0 | 6.0 | 6.0 | 6.0 |
| | 6.4 | 6.4 | 6.4 | 6.4 | 6.4 | 6.4 | 6.4 | 6.4 |
| | 6.5 | | 6.5 | 6.5 | | | 6.5 | 6.5 |
| | 6.8 | 6.8 | 6.8 | 6.8 | 6.8 | 6.8 | 6.8 | 6.8 |
| | 7.2 | 7.2 | 7.2 | | | 7.2 | | |
| | 7.3 | 7.3 | 7.3 | 7.3 | 7.3 | 7.3 | 7.3 | 7.3 |

Figure 2: DNA Fingerprint after PCR and Restriction Endonuclease Digestion. Each vertical column in the image is a fingerprint and its features are tabulated below each visualized profile. The shaded boxes represents a lack of the particular feature

The use of Python has allowed us to save time during data analysis of huge amounts of data. Designing primers of 15 bases to amplify specific number of times and of specific lengths from the entire sequence of *Escherichia coli* ATCC 8739 would be physically impossible to achieve within a short amount of time by skimming through the sequences visually. Our success with primer selection prompted us to use Python to automate tedious calculations of Dissimilarity Index for all 576 DNA fingerprint profiles (2016 permutations) in the entire projects. The calculation was done in a matter of seconds not forgetting that human errors in calculations are totally removed. In summary, Python has been an invaluable tool for us.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| H MSG | 0 | | | | | | | |
| L MSG | 0.120 | 0 | | | | | | |
| H BA | 0.120 | 0.182 | 0 | | | | | |
| L BA | 0.077 | 0.217 | 0.130 | 0 | | | | |
| H SALT | 0.333 | 0.222 | 0.222 | 0.263 | 0 | | | |
| L SALT | 0.120 | 0.182 | 0.091 | 0.130 | 0.222 | 0 | | |
| H COMB | 0.077 | 0.217 | 0.130 | 0.083 | 0.263 | 0.130 | 0 | |
| L COMB | 0.167 | 0.238 | 0.048 | 0.091 | 0.176 | 0.143 | 0.091 | 0 |
| | H MSG | L MSG | H BA | L BA | H SALT | L SALT | H COMB | L COMB |

Figure 3: Nei and Li Distance Matrix Generated from Figure 2.

**Acknowledgements**

We would like to express gratitude to Singapore Totalisation Board and Singapore Polytechnic for funding this research (Grant Account: 11-27801-45-2672).

## 6. References

BERGROTH, L., HANKONEN, H. & RAITA, T. (2000) A survey of longest common subsequence algorithms. Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00), 39–48.

COCK, P. J., ANTAO, T., CHANG, J. T., CHAPMAN, B. A., COX, C. J., DALKE, A., FRIEDBERG, I., HAMELRYCK, T., KAUFF, F., WILCZYNSKI, B. & DE HOON, M. J. (2009) Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics,* 25**,** 1422-3.

DIEFFENBACH, C. W., LOWE, T. M. J. & DVEKSLER, G. S. (1993) General concepts for PCR primer design. *Genome Res.,* 3**,** S30-S37.

HILLIER, L. & GREEN, P. (1991) OSP: a computer program for choosing PCR and DNA sequencing primers. *PCR Methods Appl,* 1**,** 124-8.

LEE, C.H., LEE, K.C., OON, J.S.H. & LING, M.H.T. (2010) Evolution Characterization of Escherichia coli Using RFLP DNA Fingerprinting. Diploma in Biotechnology Final Year Project. School of Chemical and Life Sciences, Singapore Polytechnic, Singapore.

LING, M.H.T. (2010) Recipe 577003: Extending anydbm with marshal. ActiveState Python Cookbook. [Last accessed: 12 January 2010; http://code.activestate.com/recipes/577003/]

LOWE, T., SHAREFKIN, J., YANG, S. Q. & DIEFFENBACH, C. W. (1990) A computer program for selection of oligonucleotide primers for polymerase chain reactions. *Nucleic Acids Res,* 18**,** 1757-61.

NEI, M. & LI, W. H. (1979) Mathematical model for studying genetic variation in terms of restriction endonucleases. *Proc Natl Acad Sci U S A,* 76**,** 5269-73.

RINGQUIST, S., PECORARO, C., GILCHRIST, C. M., STYCHE, A., RUDERT, W. A., BENOS, P. V. & TRUCCO, M. (2005) SOP3v2: web-based selection of oligonucleotide primer trios for genotyping of human and mouse polymorphisms. *Nucleic Acids Res,* 33**,** W548-52.

SEZONOV, G., JOSELEAU-PETIT, D. & D'ARI, R. (2007) Escherichia coli Physiology in Luria-Bertani Broth. *Journal of Bacteriology,* 189**,** 8746–8749.

WELSH, J. & MCCLELLAND, M. (1990) Fingerprinting genomes using PCR with arbitrary primers. *Nucleic Acids Res,* 18**,** 7213-8.

YAO, J., LIN, H., VAN DEYNZE, A., DODDAPANENI, H., FRANCIS, M., LEMOS, E. G. & CIVEROLO, E. L. (2008) PrimerSNP: a web tool for whole-genome selection of allele-specific and common primers of phylogenetically-related bacterial genomic sequences. *BMC Microbiol,* 8**,** 185.

YOU, F. M., HUO, N., GU, Y. Q., LUO, M. C., MA, Y., HANE, D., LAZO, G. R., DVORAK, J. & ANDERSON, O. D. (2008) BatchPrimer3: a high throughput web application for PCR and sequencing primer design. *BMC Bioinformatics,* 9**,** 253.

YOU, F. M., HUO, N., GU, Y. Q., LAZO, G. R., DVORAK, J. & ANDERSON, O. D. (2009) ConservedPrimers 2.0: a high-throughput pipeline for comparative genome referenced intron-flanking PCR primer design and its application in wheat SNP discovery. *BMC Bioinformatics,* 10**,** 331.

## I. Appendix I: Code to Calculate Nei and Li Distance Matrix

```
From Bactome import Nei_Li

profile = [[3.8, 4.0, 4.1, 4.2, 4.4, 5.0, 5.4,
            5.9, 6.0, 6.4, 6.5, 6.8, 7.2, 7.3],
           [4.0, 4.1, 4.2, 5.0, 5.4, 5.9, 6.0, 6.4, 6.8, 7.2, 7.3],
           [3.8, 4.2, 5.0, 5.4, 5.9, 6.0, 6.4, 6.5, 6.8, 7.2, 7.3],
           [3.8, 4.0, 4.2, 4.4, 5.0, 5.4,
            5.9, 6.0, 6.4, 6.5, 6.8, 7.3],
           [4.2, 5.4, 5.9, 6.0, 6.4, 6.8, 7.3],
           [3.8, 4.2, 4.4, 5.0, 5.4, 5.9, 6.0, 6.4, 6.8, 7.2, 7.3],
           [3.8, 4.1, 4.2, 4.4, 5.0, 5.4, 5.9,
            6.0,6.4, 6.5, 6.8, 7.3],
           [3.8, 4.2, 5.0, 5.4, 5.9, 6.0,6.4, 6.5, 6.8, 7.3]]

distance = [(p1, p2, Nei_Li(profile[p1], profile[p2]))
            for p1 in range(len(profile))
            for p2 in range(len(profile))]

print distance
```