1 Chapter XXX Codon Alignment

This chapter is about Codon Alignments, which is a special case of nucleotide alignment in which the trinucleotides correspond directly to amino acids in the translated protein product. Codon Alignment carries information that can be used for many evolutionary analysis.

This chapter has been divided into four parts to explain the codon alignment support in Biopython. First, a general introduction about the basic classes in `Bio.CodonAlign` will be given. Then, a typical procedure of how to obtain a codon alignment within Biopython is then discussed. Next, some simple applications of codon alignment, such as dN/dS ratio estimation and neutrality test and so forth will be covered. Finally, IO support of codon alignment will help user to conduct analysis that cannot be done within Biopython.

1.1 X.1 `CodonSeq` Class

`Bio.CodonAlign.CodonSeq` object is the base object in Codon Alignment. It is similar to `Bio.Seq` but with some extra attributes. To obtain a simple `CodonSeq` object, you just need to give a `str` object of nucleotide sequence whose length is a multiple of 3 (This can be violated if you have `rf_table` argument). For example:

```
>>> from Bio.CodonAlign import CodonSeq
>>> codon_seq = CodonSeq("AAATTTCCCGGG")
>>> codon_seq
CodonSeq('AAATTTCCCGGG', Gapped(CodonAlphabet(), '-'))
```

An error will raise up if the input sequence is not a multiple of 3.

```
>>> codon_seq = CodonSeq("AAATTTCCCGG")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/biopython/Bio/CodonAlign/CodonSeq.py", line 81, in __init__
    assert len(self) % 3 == 0, "Sequence length is not a triple number"
AssertionError: Sequence length is not a triple number
```

By default, `Bio.CodonAlign.default_codon_alphabet` will be assigned to `CodonSeq` object if you don't specify any Alphabet. This `default_codon_alphabet` is gapped universal genetic code, which will work in most cases. However, if you are analyzing data from mitochondria, for instance, and are in need of assigning an special codon alphabet by yourself, `Bio.CodonAlign.CodonAlphabet` also provides you an easy solution. All you need is to pick up a `CodonTable` object that is correct for your data. For example:

```
>>> from Bio.CodonAlign import CodonSeq
>>> from Bio.CodonAlign.CodonAlphabet import get_codon_alphabet
>>> from Bio.Data.CodonTable import generic_by_id
# vertebrate mitochondria alphabet
>>> codon_alphabet = get_codon_alphabet(generic_by_id[2], gap_char="-")
>>> codon_seq1 = CodonSeq("AAA---CCCGGG", alphabet=codon_alphabet)
>>> codon_seq1
CodonSeq('AAA---CCCGGG', CodonAlphabet(Vertebrate Mitochondrial))
```

The slice of CodonSeq is exactly the same with Seq and it will always return a Seq object if you sliced a CodonSeq. For example:

```
>>> codon_seq1
CodonSeq('AAA---CCCGGG', CodonAlphabet(Vertebrate Mitochondrial))
>>> codon_seq1[:6]
Seq('AAA---', DNAAlphabet())
>>> codon_seq1[1:5]
Seq('AA--', DNAAlphabet())
```

As you might imagine, CodonSeq is able to be translated into amino acid sequence based on the CodonAlphabet within it. In fact, CodonSeq does more than this. CodonSeq object has a rf_table attribute that dictates how the CodonSeq will be translated (rf_table will indicate the starting position of each codon in the sequence). This is useful if you sequence is known to have frameshift events or pseudogene that has insertion or deletion. You might notice that in the previous example, you haven't specify the rf_table when initiate a CodonSeq object. In fact, CodonSeq object will automatically assign a rf_table to the CodonSeq if you don't say anything about it.

```
>>> codon_seq1 = CodonSeq("AAACCCGGG")
>>> codon_seq1
CodonSeq('AAACCCGGG', CodonAlphabet(Standard))
>>> codon_seq1.rf_table
[0, 3, 6]
>>> codon_seq1.translate()
'KPG'
>>> codon_seq2 = CodonSeq("AAACCCGG", rf_table=[0, 3, 5])
>>> codon_seq2.rf_table
[0, 3, 5]
>>> codon_seq2.translate()
'KPR'
```

In the example, we didn't assign rf_table to codon_seq1. By default, CodonSeq will automatically generate a rf_table to the coding sequence assuming no frameshift events. In this case, it is [0, 3, 6], which means the first codon in the sequence starts at position 0, the second codon in the sequence starts at position 3, and the third codon in the sequence starts at position 6. In codon_seq2, we only have 8 nucleotides in the sequence, but with rf_table option specified. In this case, the third codon starts at the 5th position of the sequence rather than the 6th. And the translate() function will use the rf_table to get the translated amino acid sequence.

Another thing to keep in mind is that rf_table will only be applied to ungapped nucleotide sequence. This makes rf_table to be interchangeable between CodonSeq with the same sequence but different gaps inserted. For example,

```
>>> codon_seq1 = CodonSeq("AAACCC---GGG")
>>> codon_seq1.rf_table
[0, 3, 6]
>>> codon_seq1.translate()
'KPG'
>>> codon_seq1.full_translate()
'KP-G'
```

We can see that the `rf_table` of `codon_seq1` is still `[0, 3, 6]`, even though we have gaps added. The `translate()` function will skip the gaps and return the ungapped amino acid sequence. If gapped protein sequence is what you need, `full_translate()` comes to help.

It is also easy to convert `Seq` object to `CodonSeq` object, but it is the user's responsibility to ensure all the necessary information is correct for a `CodonSeq` (mainly `rf_table`).

```
>>> from Bio.Seq import Seq
>>> codon_seq = CodonSeq()
>>> seq = Seq('AAAAAA')
>>> codon_seq.from_seq(seq)
CodonSeq('AAAAAA', CodonAlphabet(Standard))
>>> seq = Seq('AAAAA')
>>> codon_seq.from_seq(seq)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/biopython/Bio/CodonAlign/CodonSeq.py", line 264, in from_seq
    return cls(seq._data, alphabet=alphabet)
  File "/biopython/Bio/CodonAlign/CodonSeq.py", line 80, in __init__
    assert len(self) % 3 == 0, "Sequence length is not a triple number"
AssertionError: Sequence length is not a triple number
>>> codon_seq.from_seq(seq, rf_table=(0, 2))
CodonSeq('AAAAA', CodonAlphabet(Standard))
```

1.2 X.2 `CodonAlignment` Class

The `CodonAlignment` class is another new class in `Codon.Align`. It's aim is to store codon alignment data and apply various analysis upon it. Similar to `MultipleSeqAlignment`, you can use numpy style slice to a `CodonAlignment`. However, once you sliced, the returned result will always be a `MultipleSeqAlignment` object.

```
>>> from Bio.CodonAlign import default_codon_alphabet, CodonSeq, CodonAlignment
>>> from Bio.Alphabet import generic_dna
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Alphabet import IUPAC, Gapped
>>> a = SeqRecord(CodonSeq("AAAACGTCG", alphabet=default_codon_alphabet), id="Alpha")
>>> b = SeqRecord(CodonSeq("AAA---TCG", alphabet=default_codon_alphabet), id="Beta")
>>> c = SeqRecord(CodonSeq("AAAAGGTGG", alphabet=default_codon_alphabet), id="Gamma")
>>> codon_aln = CodonAlignment([a, b, c])
>>> print codon_aln
CodonAlphabet(Standard) CodonAlignment with 3 rows and 9 columns (3 codons)
AAAACGTCG Alpha
AAA---TCG Beta
AAAAGGTGG Gamma
>>> codon_aln[0]
ID: Alpha
Name: <unknown name>
Description: <unknown description>
```

```
Number of features: 0
CodonSeq('AAAACGTCG', CodonAlphabet(Standard))
>>> print codon_aln[:, 3]
A-A
>>> print codon_aln[1:, 3:10]
CodonAlphabet(Standard) alignment with 2 rows and 6 columns
---TCG Beta
AGGTGG Gamma
```

You can write out `CodonAlignment` object just as what you do with `MultipleSeqAlignment`.

```
>>> from Bio import AlignIO
>>> AlignIO.write(codon_aln, 'example.aln', 'clustal')
```

An alignment file called `example.aln` can then be found in your current working directory. You can write `CodonAlignment` out in any MSA format that Biopython supports.

Currently, you are not able to read MSA data as a `CodonAlignment` object directly (because of dealing with `rf_table` issue for each sequence). However, you can read the alignment data in as a `MultipleSeqAlignment` object and convert them into `CodonAlignment` object using `from_msa()` class method. For example,

```
>>> aln = AlignIO.read('example.aln', 'clustal')
>>> codon_aln = CodonAlignment()
>>> print codon_aln.from_msa(aln)
CodonAlphabet(Standard) CodonAlignment with 3 rows and 9 columns (3 codons)
AAAACGTCG Alpha
AAA---TCG Beta
AAAAGGTGG Gamma
```

Note, the `from_msa()` method assume there is no frameshift events occurs in your alignment. Its behavior is not guaranteed if your sequence contain frameshift events!!

There is a couple of methods that can be applied to `CodonAlignment` class for evolutionary analysis. We will cover them more in X.4.

1.3 X.3 Build a Codon Alignment

Building a codon alignment is the first step of many evolutionary anaysis. But how to do that? `Bio.CodonAlign` provides you an easy funciton `build()` to achieve all. The data you need to prepare in advance is a protein alignment and a set of DNA sequences that can be translated into the protein sequences in the alignment.

`CodonAlign.build` method requires two mandatory arguments. The first one should be a protein `MultipleSeqAlignment` object and the second one is a list of nucleotide `SeqRecord` object. By default, `CodonAlign.build` assumes the order of the alignment and nucleotide sequences are in the same. For example:

```
>>> from Bio import CodonAlign
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Align import MultipleSeqAlignment
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Seq import Seq
>>> nucl1 = SeqRecord(Seq('AAATTTCCCGGG', alphabet=IUPAC.IUPACUnambiguousDNA()), id='nucl1')
>>> nucl2 = SeqRecord(Seq('AAATTACCCGCG', alphabet=IUPAC.IUPACUnambiguousDNA()), id='nucl2')
>>> nucl3 = SeqRecord(Seq('ATATTACCCGGG', alphabet=IUPAC.IUPACUnambiguousDNA()), id='nucl3')
>>> prot1 = SeqRecord(nucl1.seq.translate(), id='prot1')
>>> prot2 = SeqRecord(nucl2.seq.translate(), id='prot2')
>>> prot3 = SeqRecord(nucl3.seq.translate(), id='prot3')
>>> aln = MultipleSeqAlignment([prot1, prot2, prot3])
>>> codon_aln = CodonAlign.build(aln, [nucl1, nucl2, nucl3])
>>> print codon_aln
CodonAlphabet(Standard) CodonAlignment with 3 rows and 12 columns (4 codons)
AAATTTCCCGGG nucl1
AAATTACCCGCG nucl2
ATATTACCCGGG nucl3
```

In the above example, `CodonAlign.build` will try to match `nucl1` with `prot1`, `nucl2` with `prot2` and `nucl3` with `prot3`, i.e., assuming the order of records in `aln` and `[nucl1, nucl2, nucl3]` is the same.

`CodonAlign.build` method is also able to handle key match. In this case, records with same id are paired. For example:

```
>>> nucl1 = SeqRecord(Seq('AAATTTCCCGGG', alphabet=IUPAC.IUPACUnambiguousDNA()), id='nucl1')
>>> nucl2 = SeqRecord(Seq('AAATTACCCGCG', alphabet=IUPAC.IUPACUnambiguousDNA()), id='nucl2')
>>> nucl3 = SeqRecord(Seq('ATATTACCCGGG', alphabet=IUPAC.IUPACUnambiguousDNA()), id='nucl3')
>>> prot1 = SeqRecord(nucl1.seq.translate(), id='prot1')
>>> prot2 = SeqRecord(nucl2.seq.translate(), id='prot2')
>>> prot3 = SeqRecord(nucl3.seq.translate(), id='prot3')
>>> aln = MultipleSeqAlignment([prot1, prot2, prot3])
>>> nucl = {'prot1': nucl1, 'prot2': nucl2, 'prot3': nucl3}
>>> codon_aln = CodonAlign.build(aln, nucl)
>>> print codon_aln
CodonAlphabet(Standard) CodonAlignment with 3 rows and 12 columns (4 codons)
AAATTTCCCGGG nucl1
AAATTACCCGCG nucl2
ATATTACCCGGG nucl3
```

This option is handleful if you read nucleotide sequences using `SeqIO.index` method, in which case the nucleotide dict with be generated automatically.

Sometimes, you are neither not able to ensure the same order or the same id. `CodonAlign.build` method provides you an manual approach to tell the program nucleotide sequence and protein sequence correspondance by generating a `corr_dict`. `corr_dict` should be a dictionary that uses protein record id as key and nucleotide record id as item. Let's look at an example:

```
>>> nucl1 = SeqRecord(Seq('AAATTTCCCGGG', alphabet=IUPAC.IUPACUnambiguousDNA()), id='nucl1')
>>> nucl2 = SeqRecord(Seq('AAATTACCCGCG', alphabet=IUPAC.IUPACUnambiguousDNA()), id='nucl2')
>>> nucl3 = SeqRecord(Seq('ATATTACCCGGG', alphabet=IUPAC.IUPACUnambiguousDNA()), id='nucl3')
>>> prot1 = SeqRecord(nucl1.seq.translate(), id='prot1')
>>> prot2 = SeqRecord(nucl2.seq.translate(), id='prot2')
>>> prot3 = SeqRecord(nucl3.seq.translate(), id='prot3')
>>> aln = MultipleSeqAlignment([prot1, prot2, prot3])
>>> corr_dict = {'prot1': 'nucl1', 'prot2': 'nucl2', 'prot3': 'nucl3'}
>>> codon_aln = CodonAlign.build(aln, [nucl3, nucl1, nucl2], corr_dict=corr_dict)
>>> print codon_aln
CodonAlphabet(Standard) CodonAlignment with 3 rows and 12 columns (4 codons)
AAATTTCCCGGG nucl1
AAATTACCCGCG nucl2
ATATTACCCGGG nucl3
```

We can see, even though the second argument of `CodonAlign.build` is not in the same order with `aln` in the above example, the `corr_dict` tells the program to pair protein records and nucleotide records. And we are still able to obtain the correct `CodonAlignment` object.

The underlying algorithm of `CodonAlign.build` method is very similar to `pal2nal` (a very famous perl script to build codon alignment). `CodonAlign.build` will first translate protein sequences into a long degenerate regular expression and tries to find a match in its corresponding nucleotide sequence. When translation fails, it divide protein sequence into several small anchors and tries to match each anchor to the nucleotide sequence to figure out where the mismatch and frameshift events lie. Other options available for `CodonAlign.build` includes `anchor_len` (default 10) and `max_score` (maximum tolerance of unexpected events, default 10). You may want to refer the Biopython build-in help to get more information about these options.

Now let's look at a real example of building codon alignment. Here we will use epidermal growth factor (EGFR) gene to demonstrate how to obtain codon alignment. To reduce your effort, we have already collected EGFR sequences for Homo sapiens, Bos taurus, Rattus norvegicus, Sus scrofa and Drosophila melanogaster. You can download them from here. Uncomressing the `.zip`, you will see three files. `egfr_nucl.fa` is nucleotide sequences of EGFR and `egfr_pro.aln` is EGFR protein sequence alignment in `clustal` format. The `egfr_id` contains id correspondance between protein records and nucleotide records. You can then try the following code (make sure the files are in your current python working directory):

```
>>> from Bio import SeqIO, AlignIO
>>> nucl = SeqIO.parse('egfr_nucl.fa', 'fasta', alphabet=IUPAC.IUPACUnambiguousDNA())
>>> prot = AlignIO.read('egfr_pro.aln', 'clustal', alphabet=IUPAC.protein)
>>> id_corr = {i.split()[0]: i.split()[1] for i in open('egfr_id').readlines()}
>>> aln = CodonAlign.build(prot, nucl, corr_dict=id_corr, alphabet=CodonAlign.default_codon_alphabet
/biopython/Bio/CodonAlign/__init__.py:568: UserWarning: gi|47522840|ref|NP_999172.1|(L 449) does not
  % (pro.id, aa, aa_num, nucl.id, this_codon))
>>> print aln
CodonAlphabet(Standard) CodonAlignment with 6 rows and 4446 columns (1482 codons)
ATGATGATTATCAGCATGTGGATGAGCATATCGCGAGGATTGTGGGACAGCAGCTCC...GTG gi|24657088|ref|NM_057410.3|
---------------------ATGCTGCTGCGACGGCGCAACGGCCCCTGCCCCTTC...GTG gi|24657104|ref|NM_057411.3|
------------------------------ATGAAAAAGCACGAG------------...GCC gi|302179500|gb|HM749883.1|
------------------------------ATGCGACGCTCCTGGGCGGGCGGCGCC...GCA gi|47522839|ref|NM_214007.1|
------------------------------ATGCGACCCTCCGGGACGGCCGGGGCA...GCA gi|41327737|ref|NM_005228.3|
------------------------------ATGCGACCCTCAGGGACTGCGAGAACC...GCA gi|6478867|gb|M37394.2|RATEGFR
```

We can see, while building the codon alignment a mismatch event is found. And this is shown as a UserWarning.

1.4 X.4 Codon Alignment Application

The most important application of codon alignment is to estimate nonsynonymous substitutions per site (dN) and synonymous substitutions per site (dS). `CodonAlign` currently support three counting based methods (NG86, LWL85, YN00) and maximum likelihood method to estimate dN and dS. The function to conduct dN, dS estimation is called `cal_dn_ds`. When you obtained a codon alignment, it is quite easy to calculate dN and dS. For example (assuming you have EGFR codon alignmnet in the python working space):

```
>>> from Bio.CodonAlign.CodonSeq import cal_dn_ds
>>> print aln
CodonAlphabet(Standard) CodonAlignment with 6 rows and 4446 columns (1482 codons)
ATGATGATTATCAGCATGTGGATGAGCATATCGCGAGGATTGTGGGACAGCAGCTCC...GTG gi|24657088|ref|NM_057410.3|
---------------------ATGCTGCTGCGACGGCGCAACGGCCCCTGCCCCTTC...GTG gi|24657104|ref|NM_057411.3|
------------------------------ATGAAAAAGCACGAG------------...GCC gi|302179500|gb|HM749883.1|
------------------------------ATGCGACGCTCCTGGGCGGGCGGCGCC...GCA gi|47522839|ref|NM_214007.1|
------------------------------ATGCGACCCTCCGGGACGGCCGGGGCA...GCA gi|41327737|ref|NM_005228.3|
------------------------------ATGCGACCCTCAGGGACTGCGAGAACC...GCA gi|6478867|gb|M37394.2|RATEGFR
>>> dN, dS = cal_dn_ds(aln[0], aln[1], method='NG86')
>>> print dN, dS
0.0209078305058 0.0178371876389
>>> dN, dS = cal_dn_ds(aln[0], aln[1], method='LWL95')
>>> print dN, dS
0.0203061425453 0.0163935691992
>>> dN, dS = cal_dn_ds(aln[0], aln[1], method='YN00')
>>> print dN, dS
0.0198195580321 0.0221560648799
>>> dN, dS = cal_dn_ds(aln[0], aln[1], method='ML')
>>> print dN, dS
0.0193877676103 0.0217247139962
```

If you are using maximum likelihood methdo to estimate dN and dS, you are also able to specify equilibrium codon frequency to `cfreq` argument. Available options include `F1x4`, `F3x4` and `F61`.

It is also possible to get dN and dS matrix or a tree from a `CodonAlignment` object.

```
>>> dn_matrix, ds_matrix = aln.get_dn_ds_matrxi()
>>> print dn_matrix
gi|24657088|ref|NM_057410.3|    0
gi|24657104|ref|NM_057411.3|    0.0209078305058 0
gi|302179500|gb|HM749883.1|     0.611523924924  0.61022032668   0
gi|47522839|ref|NM_214007.1|    0.614035083563  0.60401686212   0.0411803504059 0
gi|41327737|ref|NM_005228.3|    0.61415325314   0.60182631356   0.0670105144563 0.0614703609541 0
gi|6478867|gb|M37394.2|RATEGFR  0.61870883409   0.606868724887  0.0738690303483 0.0735789092792 0.05
gi|24657088|ref|NM_057410.3|    gi|24657104|ref|NM_057411.3|    gi|302179500|gb|HM749883.1| gi|47522
```

```
>>> dn_tree, ds_tree = aln.get_dn_ds_tree()
>>> print dn_tree
Tree(rooted=True)
    Clade(branch_length=0, name='Inner5')
        Clade(branch_length=0.279185347322, name='Inner4')
            Clade(branch_length=0.00859186651689, name='Inner3')
                Clade(branch_length=0.0258992353629, name='gi|6478867|gb|M37394.2|RATEGFR')
                Clade(branch_length=0.0258992353629, name='gi|41327737|ref|NM_005228.3|')
            Clade(branch_length=0.0139009266768, name='Inner2')
                Clade(branch_length=0.020590175203, name='gi|47522839|ref|NM_214007.1|')
                Clade(branch_length=0.020590175203, name='gi|302179500|gb|HM749883.1|')
        Clade(branch_length=0.294630667432, name='Inner1')
            Clade(branch_length=0.0104539152529, name='gi|24657104|ref|NM_057411.3|')
            Clade(branch_length=0.0104539152529, name='gi|24657088|ref|NM_057410.3|')
```

Another application of codon alignment that `CodonAlign` supports is Mcdonald-Kreitman test. This test compares the within species synonymous substitutions and nonsynonymous substitutions and between species synonymous substitutions and nonsynonymous substitutions to see if they are from the same evolutionary process. The test requires gene sequences sampled from different individuals of the same species. In the following example, we will use Adh gene from fluit fly to demonstrate how to conduct the test. The data includes 11 individuals from D. melanogaster, 4 individuals from D. simulans and 12 individuals from D. yakuba. The data is available from here. A function called `mktest` will be used for the test.

```
>>> from Bio import SeqIO, AlignIO
>>> from Bio.Alphabet import IUPAC
>>> from Bio.CodonAlign import build
>>> from Bio.CodonAlign.CodonAlignment import mktest

>>> pro_aln = AlignIO.read('adh.aln', 'clustal', alphabet=IUPAC.protein)
>>> p = SeqIO.index('drosophilla.fasta', 'fasta', alphabet=IUPAC.IUPACUnambiguousDNA())
>>> codon_aln = build(pro_aln, p)
>>> print codon_aln
CodonAlphabet(Standard) CodonAlignment with 27 rows and 768 columns (256 codons)
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9217|emb|X57365.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9219|emb|X57366.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9221|emb|X57367.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9223|emb|X57368.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9225|emb|X57369.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9227|emb|X57370.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9229|emb|X57371.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9231|emb|X57372.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9233|emb|X57373.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9235|emb|X57374.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9237|emb|X57375.1|
ATGGCGTTTACCTTGACCAACAAGAACGTGGTTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|9239|emb|X57376.1|
ATGGCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATT...ATC gi|9097|emb|X57361.1|
ATGGCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATT...ATC gi|9099|emb|X57362.1|
ATGGCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATT...ATC gi|9101|emb|X57363.1|
ATGGCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATC...ATC gi|9103|emb|X57364.1|
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|156879|gb|M17837.1|DROADHCK
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|156877|gb|M17836.1|DROADHCJ
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|156875|gb|M17835.1|DROADHCI
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|156873|gb|M17834.1|DROADHCH
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|156871|gb|M17833.1|DROADHCG
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATT...ATC gi|156863|gb|M19547.1|DROADHCC
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|156869|gb|M17832.1|DROADHCF
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTGGCCGGTCTGGGAGGCATT...ATC gi|156867|gb|M17831.1|DROADHCE
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATT...ATC gi|156865|gb|M17830.1|DROADHCD
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATT...ATC gi|156861|gb|M17828.1|DROADHCB
ATGTCGTTTACTTTGACCAACAAGAACGTGATTTTCGTTGCCGGTCTGGGAGGCATT...ATC gi|156859|gb|M17827.1|DROADHCA
```

```
>>> print mktest([codon_aln[1:12], codon_aln[12:16], codon_aln[16:]])
0.00206457257254
```

In the above example, `codon_aln[1:12]` belongs to D. melanogaster, `codon_aln[12:16]` belongs to D. simulans and `codon_aln[16:]` belongs to D. yakuba. `mktest` will return the p-value of the test. We can see in this case, 0.00206 << 0.01, therefore, the gene is under strong negative selection according to MK test.

1.5 X.4 Future Development

Because of the limited time frame for Google Summer of Code project, some of the functions in `CodonAlign` is not tested comprehensively. In the following days, I will continue perfect the code and several new features will be added. I am always welcome to hear your suggestions and feature request. You are also highly encouraged to contribute to the existing code. Please do not hesitable to email me (zruan1991 at gmail dot com) when you have novel ideas that can make the code better.