Supplementary Data: Evolution and Functional Information

Matthew K. Matlock[1] and S. Joshua Swamidass[1]

[1]Department of Pathology and Immunology, Washington University School of Medicine, Saint Louis, MO, 63110.

September 18, 2016

List of Figures

Notes

This python notebook contains code for generating all the figures in the paper *Evolution and Functional Information*, as well as additional experiments and supplementary figures referenced in the text.

This notebook is best run using the `jupyter notebook` command in the free python bioinformatics environment Anaconda. Anaconda can be downloaded at https://www.continuum.io/downloads

```
In [1]: %matplotlib nbagg
        from matplotlib import pyplot as plt
        from math import factorial
        import numpy as np
        import scipy.stats
        import scipy
        import random
```

Helper Functions

```
In [83]: gray_colors = ['0.0', '0.15', '0.3', '0.45', '0.6']
         markers = ['o', 's', '^', 'x', 'v']

         # A helper to get the last element of an iterator
         def last(iterator):
             rval = None
             for val in iterator:
                 rval = val
             return rval

         # A helper to flatten a list of lists to a single list
```

```
def flatten(L):
    return sum(L, [])

# A helper to iterate a simulation, and compute MISA
def run_simulation(m, L, iters, simcls):
    S=simcls(L=L, mutrate=m)
    for i in xrange(0, iters):
        nseq = S.extant_sequence()
        ndiff = S.diff(S.ancestral(), nseq)
        S.accumulate_sequence(nseq)
        yield (i+1, ndiff, S.MISA())

# Define a base class for simulations which
# defines a PSSM and MISA calculations
class Simulation(object):
  def __init__(self, L):
    self.L = L
    self.PSSM = scipy.zeros((self.L, 20))

  def reset(self):
    self.PSSM = scipy.zeros((self.L, 20))

  def accumulate_sequence(self, S):
    #add sequence to PSSM
    for n, aa in enumerate(S): self.PSSM[n, aa] += 1

  def MISA(self):
    #compute MISA of PSSM following Kirk Durston's formula.
    P = self.PSSM

    MISA = 0
    for i in xrange(self.L):
      # normalization PSSM column into probabilities
      p = P[i] / float(P[i].sum())
      # ignore zeros because 0 * log 0 -> 0 in information calculations
      p = p[p!=0]
      # accumulate the mutual information (using uniform aa background)
      MISA += -scipy.log(1./20) + (p * scipy.log(p)).sum()

    return MISA / scipy.log(2) #convert to bits per protein

  # A helper to compute the number of AA
  # differences between two sequences
  def diff(self, S1, S2):
    return sum([ 1 if x!=y else 0 for x, y in zip(S1,S2)])

  # Define placeholders to be filled in by specific models
  def random(self):
    raise NotImplementedError()

  def mutate(self, seq):
    raise NotImplementedError()
```

Equations

Definition of functional information (FI) (Equation 1):

$$FI = -\log_2(F/W)$$

F is the number of functional sequences and W is the total number of sequences under consideration.
Definition of Mutual Information of Sequence Alignment (MISA) (Equation 2):

$$MISA = L \cdot \log_2\left(\frac{1}{20}\right) - \sum_i \sum_a P(a \text{ at } i) \log_2 P(a \text{ at } i)$$

L is the length of the sequence alignment, $P(a \text{ at } i)$ is the probability of observing amino acid a at position i in the sequence alignment.

Uniform Model of Sequence Evolution

The first and most important simulation demonstrates that a mindless evolutionary process can produce sequences with very high MISA, even when there are no functional constraints on the sequences. In this simulation, several extant sequences a fixed number of mutational events away from a single ancestral sequence are sampled. There are no functional constraints, so the FI equals zero (Equation 1).

Under the unform model, mutations are modeled as a poisson process with an event rate $\lambda =$ mutrate. Transitions between each pair of amino acids occur with uniform, equal probability. Extant sequences are generated by mutating a common ancestral sequence and then the extant sequences are accumulated in a Position Specific Scoring Matrix (PSSM). MISA is calculated using equation 2.

```
In [103]: # Defines a class for simulating neutral evolution of
          # non-functional sequences under a uniform mutational model.
          # Transtitions between all amino acids are equally likely.
          class UniformSimulation(Simulation):
            def __init__(self, L = 300, mutrate = .5):
              super(UniformSimulation, self).__init__(L)

              # Initialize a random ancestral sequence.
              self.ANCESTRAL = self.random()
              # Initialize a poisson mutation model
              self.POISSON = scipy.stats.poisson(mutrate)

            def random(self):
              S = scipy.stats.randint(0, 20).rvs(self.L)
              return np.array(S)

            def mutate(self, seq):
              # Sample mutation events from poisson distribution
              S = self.POISSON.rvs(self.L)
              # Random vector of AA selected uniformly from AA
              R = scipy.stats.randint(0, 20).rvs(self.L)
              # Apply mutations to sites
              return np.array([(r if s>0 else aa) \
                               for aa, r, s in zip(seq, R, S)])

            def extant_sequence(self):
```

```
            return self.mutate(self.ANCESTRAL)

        def ancestral(self):
            return self.ANCESTRAL
```

In [4]:
```
# We simulate evolution of non-functional sequences using the
# uniform model and several different mutation rates.

MISA_by_mutation_rate = {}
for m in [0.0,0.25,0.5,0.75,1.0]:
    print "m = %.2f" % (m),
    sim_iter = run_simulation(m, L=150, iters=10000, \
                              simcls=UniformSimulation)
    MISA_by_mutation_rate[m] = list(sim_iter)
print
```

```
m = 0.00 m = 0.25 m = 0.50 m = 0.75 m = 1.00
```

In [5]:
```
# Plot data from simulations of neutral evolution
# Plot iterations v. MISA using a log scaled X-axis
plt.figure()
ax = plt.subplot(111)
for i, m in enumerate(sorted(MISA_by_mutation_rate)):
    iters, _, MISAs = zip(*MISA_by_mutation_rate[m])
    ax.semilogx(iters, MISAs, color=gray_colors[i])
    ax.text(9000, MISAs[-1]+10, "%.2f events / aa" % (m), \
            horizontalalignment='right', color=gray_colors[i])

ylim = 700
aa_ylim = ylim/float(150)

# Show a second Y-axis for units of bits / amino acid
ax2 = ax.twinx()
ax2.set_ylim(0, aa_ylim)
ax2.set_ylabel("MISA (bits / amino acid)")
ax2.set_yticks(np.arange(0.5,aa_ylim, 0.5))

ax.set_ylim(0, ylim)
ax.set_ylabel("MISA (bits / protein)")
ax.set_xlabel("Number of Samples")
plt.savefig('figures/MISA_uniform_mutation_rate.pdf',
            format='pdf')
```

```
<IPython.core.display.Javascript object>
```

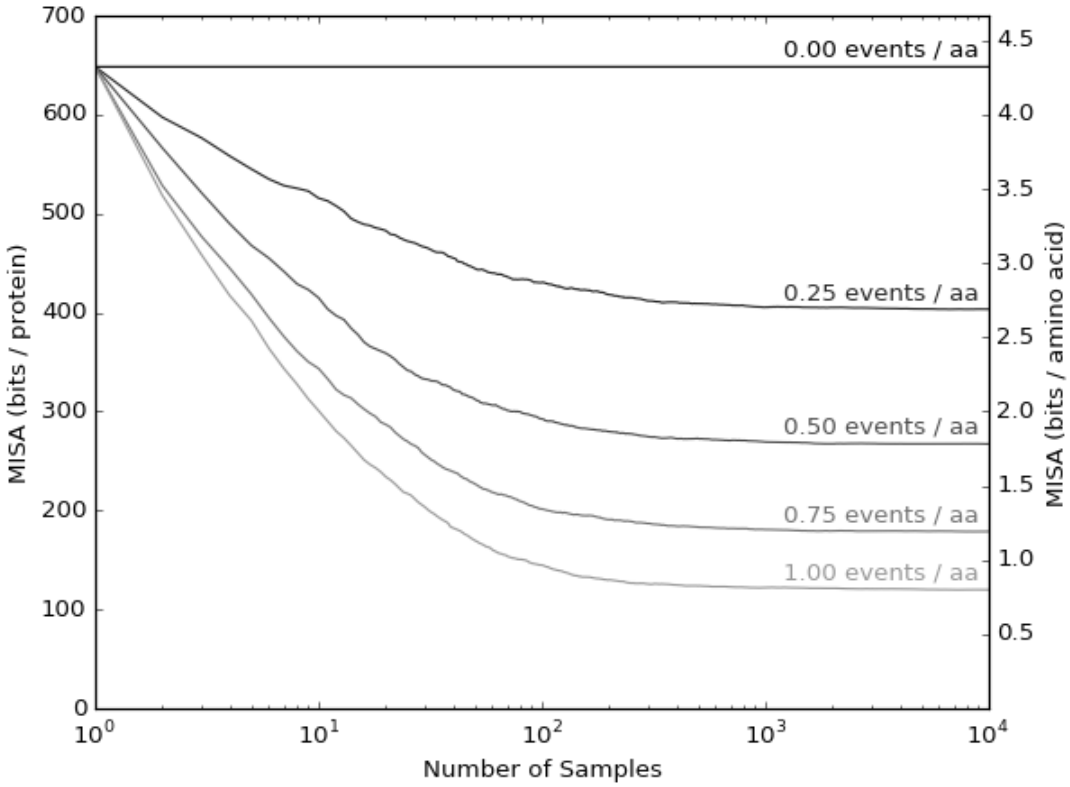


Figure 1A

Modeling amino acid sequence evolution using uniform amino acid transition probabilities shows that the evolution of non-functional sequences produce alignments with arbitrarily high MISA. Here, extant sequences 150 amino acids long, and a fixed number of mutations away from a single ancestral sequence are sampled, added to the sequence alignment, and the MISA is reported as a function of the number of samples. (A) MISA converges to a very high value after about 1000 samples. If MISA were a good estimate of functional information, all estimates should converge to zero bits.

```
In [6]: # Run a uniform simulation with multiple mutation rates and
        # sequence lengths. Store MISA at the last (1000th) iteration.
        MISA_by_mr_by_length = {}
        for m in [0.1, 0.5, 0.9]:
            print "m = %.2f:" % (m),
            MISA_by_mr_by_length[m] = []
            for L in [50, 100, 150, 200, 250, 300, 350]:
                print "L =", L,
                sim_iter = run_simulation(m, L, iters=1000, \
                                          simcls=UniformSimulation)
                _, _, MISA = last(sim_iter)
                MISA_by_mr_by_length[m].append((L, MISA))
            print
```

```
m = 0.10: L = 50 L = 100 L = 150 L = 200 L = 250 L = 300 L = 350
m = 0.50: L = 50 L = 100 L = 150 L = 200 L = 250 L = 300 L = 350
m = 0.90: L = 50 L = 100 L = 150 L = 200 L = 250 L = 300 L = 350
```

```
In [7]: # Plot MISA vs. sequence length for multiple mutation rates
        # MISA is arbitrarily high and linearly dependent upon sequence
        # length
```

```
        plt.figure()
        for i, m in enumerate(sorted(MISA_by_mr_by_length)):
            L, MISAs = zip(*MISA_by_mr_by_length[m])
            plt.plot(L, MISAs, marker=markers[i], color=gray_colors[i])
            plt.text(345, MISAs[-1]+10, "%.1f events / aa" % (m), \
                     horizontalalignment='right', color=gray_colors[i])

        ylim = 1500

        plt.ylim(0, ylim)
        plt.xlim(50, 350)
        plt.ylabel("Functional Information (bits / protein)")
        plt.xlabel("Sequence Length")
        plt.savefig('figures/MISA_uniform_length.pdf', format='pdf')
```

```
<IPython.core.display.Javascript object>
```

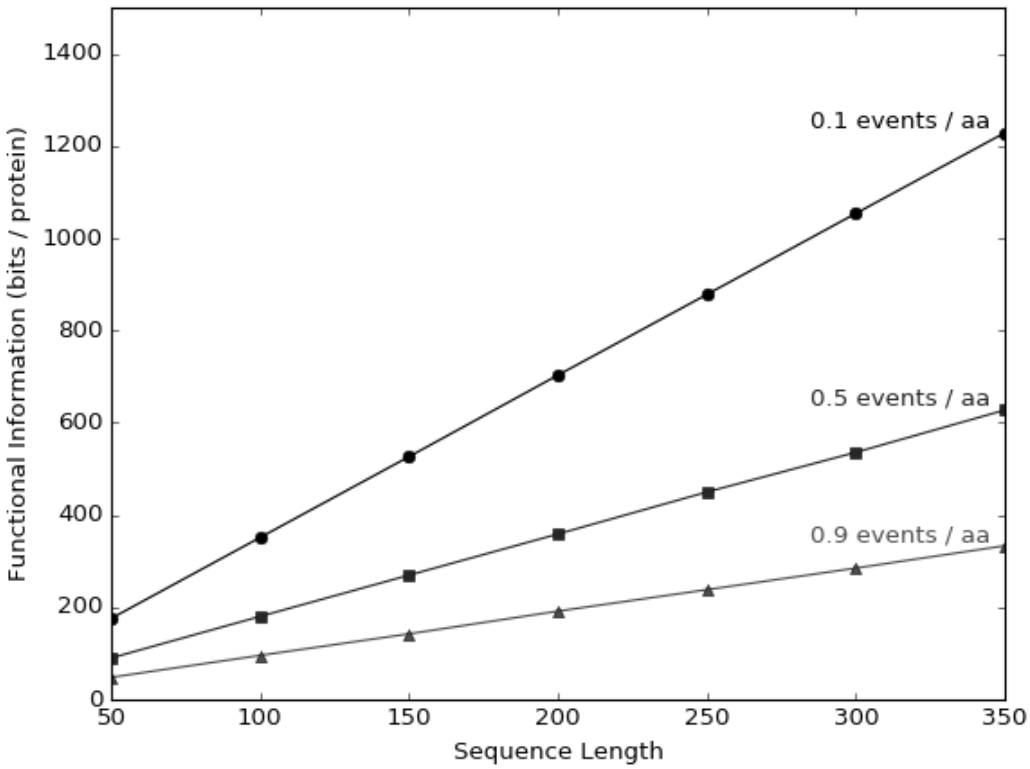


Figure 1B

The MISA of this uniform mutation process scales linearly with sequence length, even though the true FI is zero.

Codon Model of Evolution

A more accurate simulation models the evolution of codons with restrictions imposed by the genetic code. Here, an an- cestral sequence is constructed by uniformly sampling codons. Mutations change a codon into another codon with exactly one nucleotide difference. As would be expected, the MISA of non-functional sequences generated by codon evolution is even higher than that generated with a uniform model. In addition, we show that MISA encodes artifacts of the evolutionary process. In the case of the codon model, amino acids occur in proportion to the number of codons encoding them. This non-uniform amino acid probability results in an

additional 0.183 bits per amino acid of information in MISA estimates. More complicated evolutionary models would result in addition information content unrelated to the functional content of the sequences.

```
In [85]: # Define the known mapping of amino acids to DNA codons
         _codes = {
         'A': ['GCT', 'GCC', 'GCA', 'GCG'],
         'R': ['CGT', 'CGC', 'CGA', 'CGG', 'AGA', 'AGG'],
         'N': ['AAT', 'AAC'],
         'D': ['GAT', 'GAC'],
         'C': ['TGT', 'TGC'],
         'Q': ['CAA', 'CAG'],
         'E': ['GAA', 'GAG'],
         'G': ['GGT', 'GGC', 'GGA', 'GGG'],
         'H': ['CAT', 'CAC'],
         'I': ['ATT', 'ATC', 'ATA'],
         'L': ['TTA', 'TTG', 'CTT', 'CTC', 'CTA', 'CTG'],
         'K': ['AAA', 'AAG'],
         'M': ['ATG'],
         'F': ['TTT', 'TTC'],
         'P': ['CCT', 'CCC', 'CCA', 'CCG'],
         'S': ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'],
         'T': ['ACT', 'ACC', 'ACA', 'ACG'],
         'W': ['TGG'],
         'Y': ['TAT', 'TAC'],
         'V': ['GTT', 'GTC', 'GTA', 'GTG'],
         }

         # Define a helper class to generate codon sequences.
         # Mutate sequences by choosing codons with single amino acid
         # differences
         class GeneticCode:
           def __init__(self):
             # list the codons in sorted order
             self.codons = sorted([codon for a in _codes \
                                   for codon in _codes[a]])
             # list the amino acids in sorted order
             self.aa = sorted(_codes.keys())
             # book keeping
             self.codon2idx = dict((c, n) for n, c in enumerate(self.codons))
             self.aa2idx = dict((a, n) for n, a in enumerate(self.aa))

             # define a mapping of codons to amino acids
             codon2aa = {}
             for a, cs in _codes.items():
               for codon in cs:
                 codon2aa[codon] = a
             self.codon2aa = codon2aa

             # define a mapping of codons to nearby codons with at most
             # 1 nucleotide difference
             neighborhood = {}
             for C1 in self.codons:
                Ns = []
```

```
            for C2 in codon2aa:
              if diff(C1, C2) == 1: Ns.append(self.codon2idx[C2])
            neighborhood[self.codon2idx[C1]] = Ns
          self.neighborhood = neighborhood

          self.codon2aa_idx = [self.aa2idx[self.codon2aa[self.codons[cidx]]] \
                               for cidx in range(61)] # book keeping

        def random(self, length):
          return scipy.stats.randint(0, len(self.codons)).rvs(length)

        def mutate(self, codon_seq, sampling_dist):
          codon_seq = np.array(codon_seq)
          # Sample mutational events from the distribution
          S = sampling_dist.rvs(len(codon_seq))
          # For each mutational event at each codon, choose an adjacent codon
          for i, s in enumerate(S):
            for j in xrange(s):
              codon_seq[i] = np.random.choice(self.neighborhood[codon_seq[i]])
          return codon_seq

        def translate(self, codon_seq):
          return np.array([self.codon2aa_idx[c] for c in codon_seq])
```

In [86]:
```
        # Defines a class for simulating neutral evolution of non-functional
        # sequences under a codon mutational model. Amino acids can mutate to
        # a nearby codon with a difference of one nucleic acid.
        class CodonSimulation(Simulation):
          G = GeneticCode()
          def __init__(self, L = 300, mutrate = .5):
            super(CodonSimulation, self).__init__(L)
            self.ANCESTRAL = self.G.random(L)
            self.POISSON = scipy.stats.poisson(mutrate)

          def extant_sequence(self):
            return self.G.mutate(self.ANCESTRAL, self.POISSON)

          def accumulate_sequence(self, S_codon):
            S_aa = self.G.translate(S_codon)
            super(CodonSimulation, self).accumulate_sequence(S_aa)

          def diff(self, S1, S2):
            T1 = self.G.translate(S1)
            T2 = self.G.translate(S2)
            return super(CodonSimulation, self).diff(T1, T2)

          def ancestral(self):
            return self.ANCESTRAL

          def random(self):
            return self.G.random(self.L)

          def mutate(self, seq):
```

```
            return self.G.mutate(seq, self.POISSON)
```

In [146]:
```
# Simulate neutral evolution of non-functional sequences
# using a codon mutation model. Runs the same simulation
# for several mutational rates.
Codon_MISA_by_mutation_rate = {}
for m in [0.0,0.25,0.5,0.75,1.0]:
    print "m = %.2f" % (m),
    sim_iter = run_simulation(m, L=150, iters=10000, \
                                  simcls=CodonSimulation)
    Codon_MISA_by_mutation_rate[m] = list(sim_iter)
print
```

```
m = 0.00 m = 0.25 m = 0.50 m = 0.75 m = 1.00
```

In [147]:
```
# Plot the data from simulations using the codon mutational model
# Plot iteration vs. MISA using a log scaled X-axis

plt.figure()
ax = plt.subplot(111)
for i, m in enumerate(sorted(Codon_MISA_by_mutation_rate)):
    iters, _, MISAs = zip(*Codon_MISA_by_mutation_rate[m])
    ax.semilogx(iters, MISAs, color=gray_colors[i])
    ax.text(9000, MISAs[-1]+10, "%.2f events / aa" % (m), \
            horizontalalignment='right', color=gray_colors[i])

ylim=700
aa_ylim = ylim / float(150)

ax2 = ax.twinx()
ax2.set_ylim(0, aa_ylim)
ax2.set_ylabel("MISA (bits / amino acid)")
ax2.set_yticks(np.arange(0.5, aa_ylim, 0.5))

ax.set_ylim(0,ylim)
ax.set_ylabel("MISA (bits / protein)")
ax.set_xlabel("Number of Samples")
plt.savefig('figures/MISA_codon_mutation_rate.pdf',
             format='pdf')
```

```
<IPython.core.display.Javascript object>
```

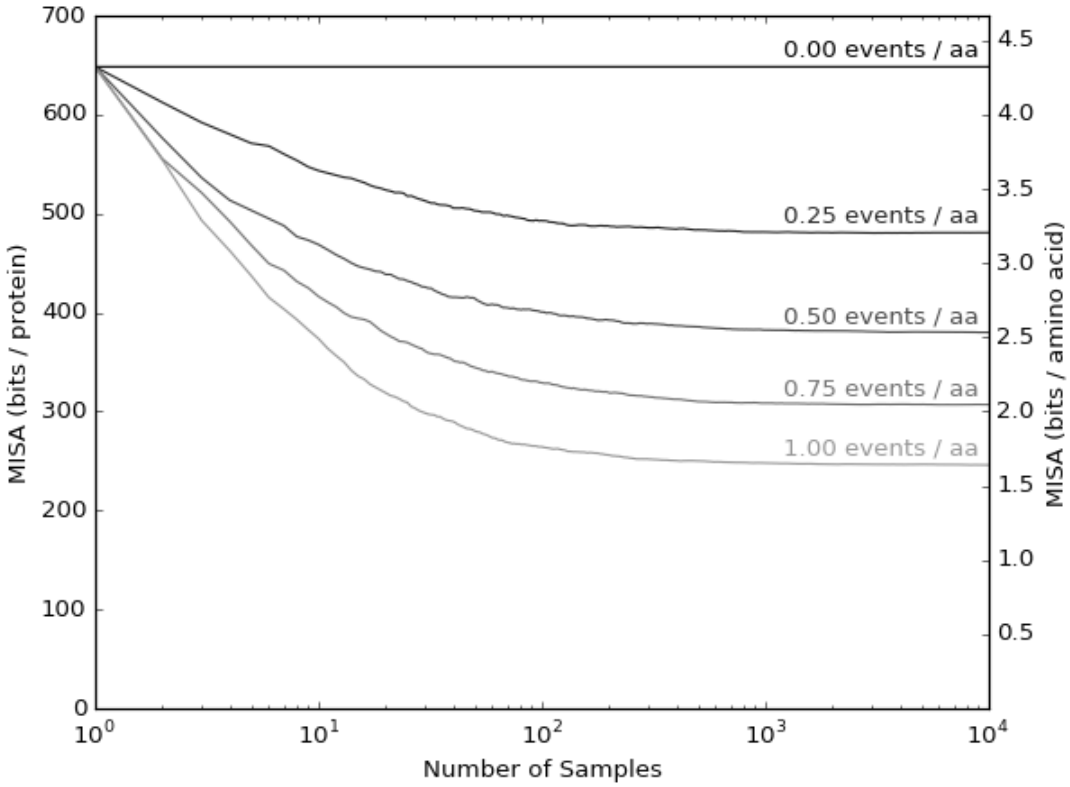


Figure 2

Codon evolution of non functional sequences. The simulation models the evolution of a codon sequence, by limiting mutations to substitution of codons with a single base-pair difference. Extant sequences a fixed number of mutations away from a single ancestral sequence are sampled, added to the sequence alignment, and the MISA is reported as a function of the number of samples. As the number of samples increase, MISA does not converge to zero if the mutation rate is not high enough, even though the true FI is zero. As mutation rate increases, asymptotic MISA does move downwards, but more slowly than uniform evolution and it does not ever reach zero FI even once all amino acids have been substituted.

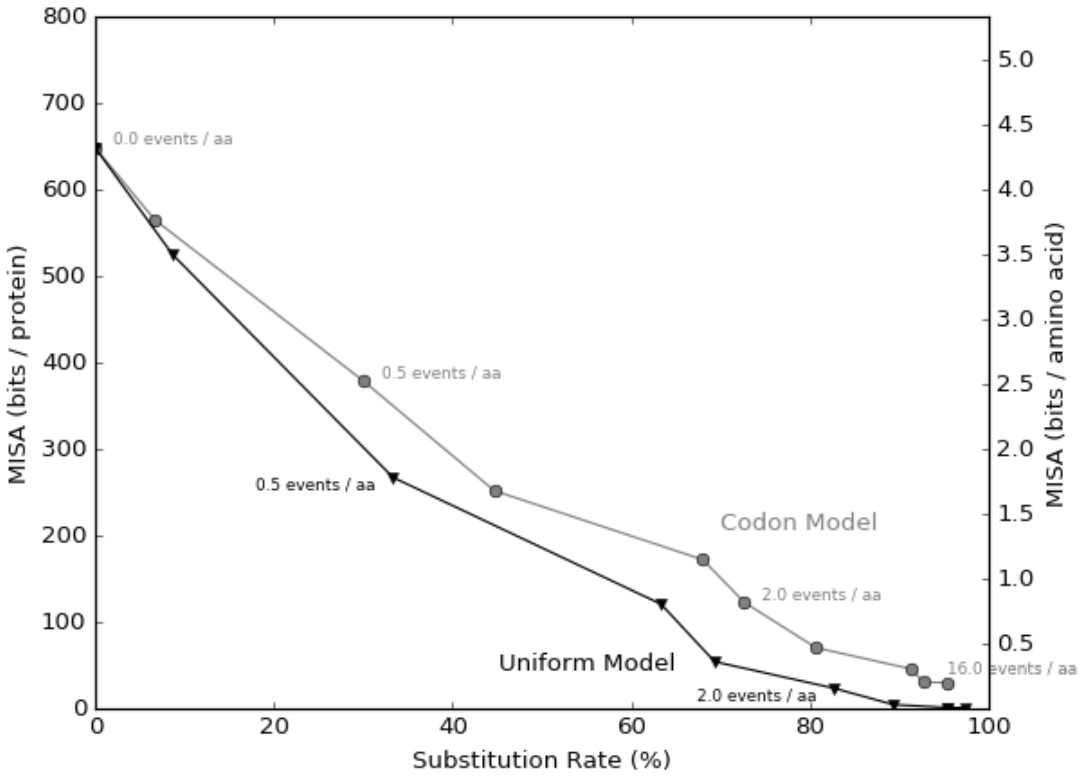
```
In [62]: # For both the codon model and the uniform model:
         # 1. Compare the effect of increasing mutation rates on MISA estimates
         # 2. Compute the observed substitution rate

         AA_MISA_by_substitution_rate = []
         Uniform_MISA_by_substitution_rate = []

         for m in [0.0, 0.1, 0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 8.0, 16.0]:
             print "m = %.2f" % (m),
             sim_iter = run_simulation(m, L=150, iters=5000, \
                                       simcls=CodonSimulation)
             _, DIFF, MISA = last(sim_iter)
             sub_rate = np.mean(DIFF)
             AA_MISA_by_substitution_rate.append((m, sub_rate, MISA))

             sim_iter = run_simulation(m, L=150, iters=5000, \
                                       simcls=UniformSimulation)
             _, DIFF, MISA = last(sim_iter)
             sub_rate = np.mean(DIFF)
             Uniform_MISA_by_substitution_rate.append((m, sub_rate, MISA))
```

```
        print
```

```
m = 0.00 m = 0.10 m = 0.50 m = 1.00 m = 1.50 m = 2.00 m = 3.00 m = 4.00 m = 8.00 m = 16.
```

In [65]:
```
# Plot substitution rate vs. MISA estimate as mutation rate increases
# for both the uniform and codon models

plt.figure()
ax=plt.subplot(111)

M, S, MISAs = zip(*AA_MISA_by_substitution_rate)
S = np.array(S) / 150. * 100.
ax.plot(S, MISAs, marker='o', color='0.5')
ax.text(70, 200, "Codon Model", color='0.5', \
        horizontalalignment='left', verticalalignment='bottom')
ax.text(S[0]+2, MISAs[0], "%.1f events / aa" % (M[0]), size='8', \
        horizontalalignment='left', verticalalignment='bottom', \
        color='0.5')
ax.text(S[2]+2, MISAs[2], "%.1f events / aa" % (M[2]), size='8', \
        horizontalalignment='left', verticalalignment='bottom', \
        color='0.5')
ax.text(S[5]+2, MISAs[5], "%.1f events / aa" % (M[5]), size='8', \
        horizontalalignment='left', verticalalignment='bottom', \
        color='0.5')
ax.text(S[-1], MISAs[7], "%.1f events / aa" % (M[-1]), size='8', \
        horizontalalignment='left', verticalalignment='center', \
        color='0.5')

M, S, MISAs = zip(*Uniform_MISA_by_substitution_rate)
S = np.array(S) / 150. * 100.
ax.plot(S, MISAs, marker='v', color='0.0')
ax.text(65, 65, "Uniform Model", color='0.0', \
        horizontalalignment='right', verticalalignment='top')
ax.text(S[2]-2, MISAs[2], "%.1f events / aa" % (M[2]), size='8', \
        horizontalalignment='right', verticalalignment='top')
ax.text(S[5]-2, MISAs[5], "%.1f events / aa" % (M[5]), size='8', \
        horizontalalignment='right', verticalalignment='top')


ylim = 800
aa_ylim = ylim / float(150)

ax2 = ax.twinx()
ax2.set_ylim(0, aa_ylim)
ax2.set_ylabel("MISA (bits / amino acid)")
ax2.set_yticks(np.arange(0.5,aa_ylim, 0.5))

ax.set_ylim(0, ylim)
ax.set_xlim(0, 100)
ax.set_ylabel("MISA (bits / protein)")
ax.set_xlabel("Substitution Rate (%)")
plt.savefig('figures/MISA_codon_v_uniform_substitution_rate.pdf',
            format='pdf')
```

```
<IPython.core.display.Javascript object>
```



Supplementary Figure 1

Once again, MISA scales inversely with mutation rate. In this model, the equilibrium amino acid composition is not uniform but is proportional to the number of codons encoding each amino acid. Consequently, even after a large number of mutations erase all memory of the ancestral sequence, an additional 0.183 bits per amino acid are added to MISA, and it never reaches zero.

```
In [47]: P = np.zeros((len(_codes),))
         for i, aa in enumerate(sorted(_codes)):
             P[i] = len(_codes[aa])

         p = P / P.sum()
         residual_information = -np.log2(1./20.) + \
                                (p * np.log2(p)).sum()

         print "Residual information due to codon evolution:"
         print "%0.3f bits/aa" % (residual_information)
```

```
Residual information due to codon evolution:
0.183 bits/aa
```

MISA is Arbitrarily High Due to Selection Pressure

Selection pressure results in populations that express only highly functional versions of proteins. However, this means that the population does not represent all possible sequences capable of performing a function. This results in a drastic overestimate of functional information using MISA and a huge underestimate of the number of functional sequences.

To demonstrate this effect, we simulated evolution of functional sequences under selection using the codon model. To model the protein family, a single ideal "centroid" sequence of 150 AA was chosen randomly. Then, an ancestral sequence was generated with 49 amino acids randomly replaced in the centroid sequence. Fitness of a sequence was dependent upon the number of amino acid changes compared to the centroid sequence. In particular, for a hamming distance d between 50 and 10 amino acid changes from the centroid sequence, fitness is equal to $2^{50-d} - 1$ for sequences greater that 50 amino acid changes from the centroid, fitness is 0, and for sequences closer than 10 amino acids, fitness is fixed at 2^{40}. We define this latter category of sequences within 10 amino acid changes as highly functional sequences.

In this simulation, we see that MISA is only a good estimate of the number of high functioning sequences, and tells us nothing about the total number of sequences. Additionally, we show that these results are robust to variations in the protein family size and the shape of the fitness function.

In [73]:
```
# Compute the number of possible functional proteins for a protein
# family defined by a single centroid sequence. Any sequence within
# 50 amino acids of the centroid is defined as functional. Any
# sequence within 10 amino acids of the centroid is defined as
# highly functional

G = 1000
L = 150
D = 50

D_HF = 10
Naa = 20
Np = Naa**L

C = lambda n, k: factorial(n) / (factorial(k) * factorial(n-k))

Mx = sum([ (Naa-1)**i * C(L, i) for i in xrange(0,D+1) ])
Mx_HF = sum([ (Naa-1)**i * C(L, i) for i in xrange(0,D_HF+1) ])

FI = -np.log2(Mx / float(Np))
FI_HF = -np.log2(Mx_HF / float(Np))

print "FI Protein Family:"
print FI
print
print "FI of High Function Proteins:"
print FI_HF
```

```
FI Protein Family:
301.96840593

FI of High Function Proteins:
555.749665709
```

In [90]:
```
# Define a simulation class to sample populations of sequences
# according to their fitness. Sample the initial population from
# a sequence at the edge of function (f_lower-1 amino acids away
# from the centroid). Compute MISA using the PSSM defined in
# CodonSimulation. The PSSM is reset between generations.

class SelectionSimulation(object):
```

```
def __init__(self, base_simulation,
                   f_lower=50, f_upper=10, popsize=1000,
                   f_function=lambda x: x):
    self.SIM = base_simulation

    # Choose an "ideal" CENTROID protein that has exactly f_lower
    # mutations from the ANCESTRAL sequence
    self.CENTROID = self.SIM.ancestral().copy()
    L = self.SIM.L
    R = self.SIM.random()
    P = np.random.choice(xrange(0,L), size=f_lower-1, replace=False)
    for i, p in enumerate(P):
        self.CENTROID[p] = R[i]

    # All sequences with at most f_upper different AAs from CENTROID
    # are equally functional
    # All sequences more than f_lower different AAs from CENTROID
    # are non-functional
    # Sequences with between [f_upper, f_lower] different AAs from
    # CENTROID are assigned fitness scores from [0.0, f_lower-f_upper]
    # These fitness scores can be scaled by an optional f_function
    self.FITNESS = lambda dist: f_function( np.clip(f_lower-dist,
                                                    0.0,
                                                    f_lower-f_upper) )

    # Initialize the POPULATION with the ANCESTRAL sequence
    self.popsize = popsize
    self.POPULATION = [ self.SIM.ancestral() ] * self.popsize

def sample(self):
    CENTROID = self.CENTROID
    MUT = self.SIM.mutate
    DIFF = self.SIM.diff
    FIT = self.FITNESS
    POP = self.POPULATION
    POP = flatten([ [seq] + \
                    [MUT(seq) \
                     for i in xrange(10)] \
                        for seq in POP ])

    # Compute the fitness of all individuals in POPULATION
    F = np.array([ FIT( DIFF(seq, CENTROID) ) for seq in POP ])
    N = F.sum()

    # Normalize the fitness to a probability distribution, if all
    # individuals have zero fitness, use a uniform distribution
    if N == 0:
        p = np.ones((len(POP),)) / len(POP)
    else:
        p = F / N

    # Sample a new population using the fitness distribution
    IX = np.arange(len(POP))
```

```
            IX = np.random.choice(IX, size=self.popsize, \
                                  replace=False, p=p)
            self.POPULATION = [POP[i] for i in IX]
            return sum([F[i] for i in IX])
            # INDEX_FIT = sorted(enumerate( F.tolist() ),
            #                    key=lambda IXF: -IXF[1])[:self.popsize]
            # self.POPULATION = [POP[i] for i, _f in INDEX_FIT]
            # return sum([F[i] for i, _f in INDEX_FIT])

        def MISA(self):
            self.SIM.reset()
            ACCUMULATE = self.SIM.accumulate_sequence
            # Compute the PSSM and MISA for the new population
            for S in self.POPULATION:
                ACCUMULATE(S)
            return self.SIM.MISA()
```

In [77]:
```
# NOTICE: These experiments take ~2 hours on a
# MacBook Pro 2.4 Ghz Core i5 Laptop with 8gb memory
# Run a positive selection simulation with L=150,
# f_lower=50 and f_upper=10 using the codon model
# Track population fitness and MISA

S_codon = SelectionSimulation(
                  CodonSimulation(L=L, mutrate=0.01), \
                  popsize=2000, \
                  f_lower=D, f_upper=D_HF, \
                  f_function=lambda x: 1.5**x-1)
MISA_positive_selection_codon = []
for i in xrange(0, G):
    pop_fit = S_codon.sample()
    MISA = S_codon.MISA()
    MISA_positive_selection_codon.append((i+1, pop_fit, MISA))
    if i % 50 == 0: print i,
print
```

```
0 50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950
```

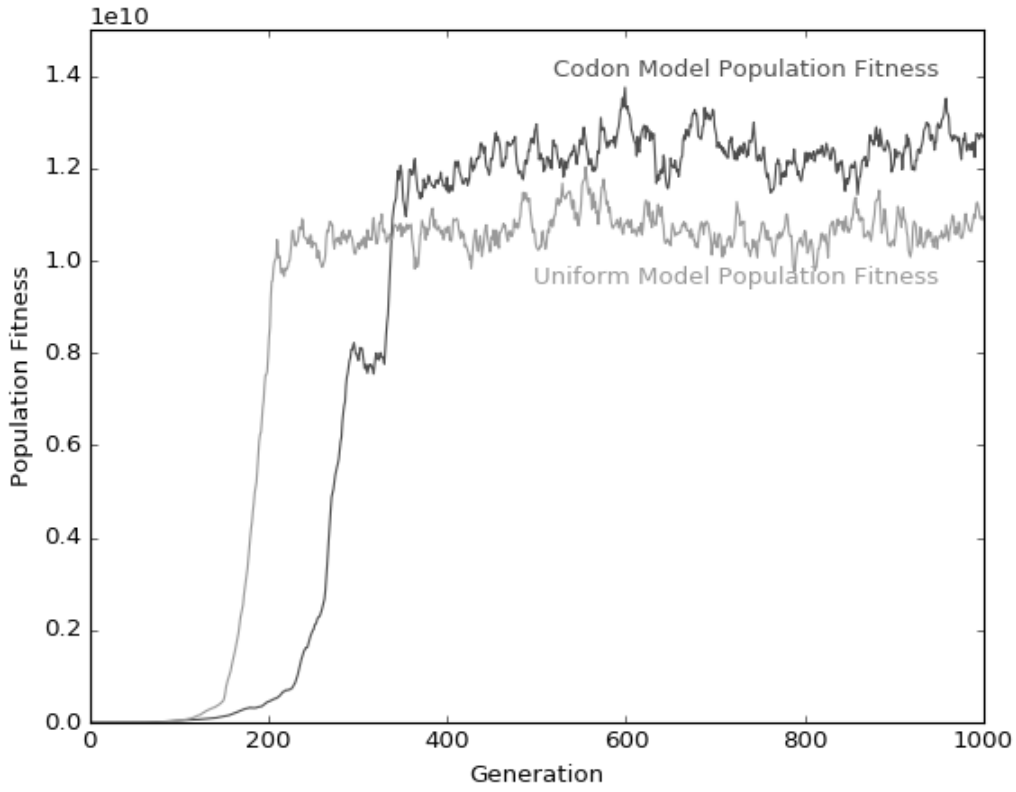
In [106]:
```
# NOTICE: These experiments take ~2 hours on a
# MacBook Pro 2.4 Ghz Core i5 Laptop with 8gb memory
# Run a positive selection simulation with L=150,
# f_lower=50 and f_upper=10 using the uniform model
# Track population fitness and MISA

S_uniform = SelectionSimulation(
                  UniformSimulation(L=L, mutrate=0.01), \
                  popsize=2000, \
                  f_lower=D, f_upper=D_HF, \
                  f_function=lambda x: 1.5**x-1)
MISA_positive_selection_uniform = []
```

```
        for i in xrange(0, G):
            pop_fit = S_uniform.sample()
            MISA = S_uniform.MISA()
            MISA_positive_selection_uniform.append((i+1, pop_fit, MISA))
            if i % 50 == 0: print i,
        print
```

```
0 50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950
```

In [138]:
```
# Plot the generation vs. total population fitness on a log plot.

        plt.figure()

        iters, popfitness, _ = zip(*MISA_positive_selection_codon)
        plt.plot(iters, popfitness, color='0.3')
        plt.text(950, 1.4e10, "Codon Model Population Fitness",
                 horizontalalignment='right', color='0.3')

        iters, popfitness, _ = zip(*MISA_positive_selection_uniform)
        plt.plot(iters, popfitness, color='0.6')
        plt.text(950, 9.5e9, "Uniform Model Population Fitness",
                 horizontalalignment='right', color='0.6')

        plt.xlim(1, G)
        plt.ylim(0,1.5e10)
        plt.ylabel("Population Fitness")
        plt.xlabel("Generation")
```

```
<IPython.core.display.Javascript object>
```




```
Out[138]: <matplotlib.text.Text at 0x11e8d3a90>
```

Supplementary Figure 2

The population of sequences under positive selection converges to a high level of fitness over time. This population fitness indicates an average distance of 12 amino acids from the centroid.

```
In [113]: iters, popfitness, _ = zip(*MISA_positive_selection_codon)

          avg_fitness = np.mean(popfitness[-50:])/2000+1
          print 50 - np.log(avg_fitness) / np.log(1.5)
```

```
11.3665382282
```

```
In [148]: # Plot generation vs. MISA for the positive selection simulation.
          # Also plot FI of the protein family and FI of highly functional
          # proteins within the family

          plt.figure()
          ax = plt.subplot(111)

          iters, _, MISAs = zip(*MISA_positive_selection_codon)
          plt.plot(iters, MISAs, color='0.3')
          plt.text(G-10, 610, "MISA (Codon Model)", \
                   horizontalalignment='right', color='0.3')

          iters, _, MISAs = zip(*MISA_positive_selection_uniform)
          plt.plot(iters, MISAs, color='0.6')
          plt.text(G-10, 581, "MISA (Uniform Model)", \
```

```
                horizontalalignment='right', color='0.6',
                verticalalignment='top')

        plt.axhline(FI_HF, xmin=0, xmax=1000, color='k', linestyle='--')
        plt.text(G-10, FI_HF - 10, "FI (High Function)", \
                horizontalalignment='right', verticalalignment='top')

        plt.axhline(FI, xmin=0, xmax=1000, color='k', linestyle='--')
        plt.text(G-10, FI + 10, "FI (Low Function)", \
                horizontalalignment='right')

        ylim = 800
        aa_ylim = ylim / float(L)

        ax2 = ax.twinx()
        ax2.set_ylim(0, aa_ylim)
        ax2.set_ylabel("Information (bits / amino acid)")
        ax2.set_yticks(np.arange(0.5,aa_ylim, 0.5))

        ax.set_ylim(0, ylim)
        ax.set_xlim(0, G)
        ax.set_ylabel("Information (bits / protein)")
        ax.set_xlabel("Generation")
        plt.savefig('figures/MISA_codon_selection_simulation.pdf',
                    format='pdf')
```

```
<IPython.core.display.Javascript object>
```

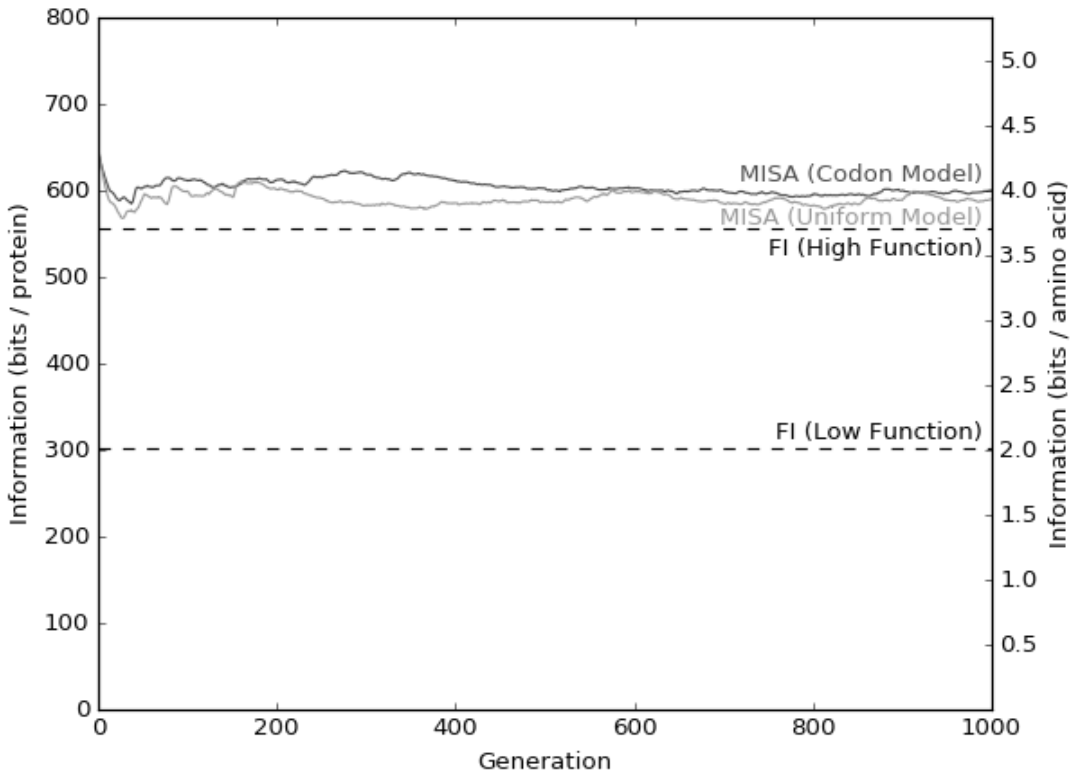


Figure 4

Simulations show that MISA fantastically overestimates the FI of sequences under selection and, therefore, drastically underestimates the number of functional proteins. In this simulation, a functional sequence family is defined as all sequences 150 amino acids long with fewer than 50 amino acids different from a fixed centroid sequence. This space contains 1.8×10^{104} possible functional sequences, and corresponds to a true FI of about 301 bits. Simulating selection, we defined a fitness function that varies exponentially between 50 and 10 differences from the centroid, with sequences closer to the centroid given higher fitness. All sequences within 10 amino acids of the centroid were defined as highly functional and assigned equal fitness. The FI of the high function group is about 555 bits. A population of 2,000 sequences is used for the simulation. Each generation, ten offspring per sequence are generated, using a fixed mutation rate of 0.01 events per amino acid. The next generation is selected from the offspring by random sampling weighted by fitness. The population is initialized to 2,000 copies of a single ancestral sequence exactly 49 amino acids from the centroid sequence, at the edge of functionality. The MISA converges to a value greater than 600 bits, overestimating FI by approximately 307 bits, and underestimating the number of functional sequences by about 90 orders of magnitude.

In [125]:
```
# Compute the MISA predictions of FI and number of functional proteins
# Compare these to the true FI and number of functional proteins

_, _, MISAs = zip(*MISA_positive_selection_codon)
MISA_ps = scipy.stats.gmean(MISAs[-100:])
E_Mx_ps = Np * 2**(-MISA_ps)

print "Number of sequences of lenght %d" % (L)
print "Np    =", float(Np)
print
print "MISA Approximations:"
print "FI    ~", MISA_ps
print "Mx    ~", E_Mx_ps
print
print "True Functional Protein Count"
print "FI    =", FI
print "Mx    =", float(Mx)
print
print "High Function Protein Count"
print "FI    =", FI_HF
print "Mx    =", float(Mx_HF)
```

```
Number of sequences of lenght 150
Np    = 1.42724769271e+195

MISA Approximations:
FI    ~ 598.002925704
Mx    ~ 1.37303376687e+15

True Functional Protein Count
FI    = 301.96840593
Mx    = 1.79040557593e+104

High Function Protein Count
FI    = 555.749665709
Mx    = 7.19747045729e+27
```

Selection simulation with a larger protein family size

Selection simulations using a larger protein family show that MISA still only estimates the number of highly functional proteins even with a protein family containing 10^{62} more functional proteins compared to the previous simulation. This protein class consists of sequences of length L=150, which are at most 100 aa changes away from a single centroid sequence. Again, proteins with fewer than 10 aa changes all have maximum fitness, and proteins with greater than 100 aa changes have zero fitness.

```
In [133]: # Compute the FI of a larger protein family defined as all sequences
          # within 100 amino acids of a centroid sequence of length 150

          D_wide = 100
          Mx_wide = sum([ (Naa-1)**i * C(L, i) for i in xrange(0,D_wide+1) ])
          FI_wide = -np.log2(Mx_wide / float(Np))
          print "FI of Larger Protein Family"
          print FI_wide
```

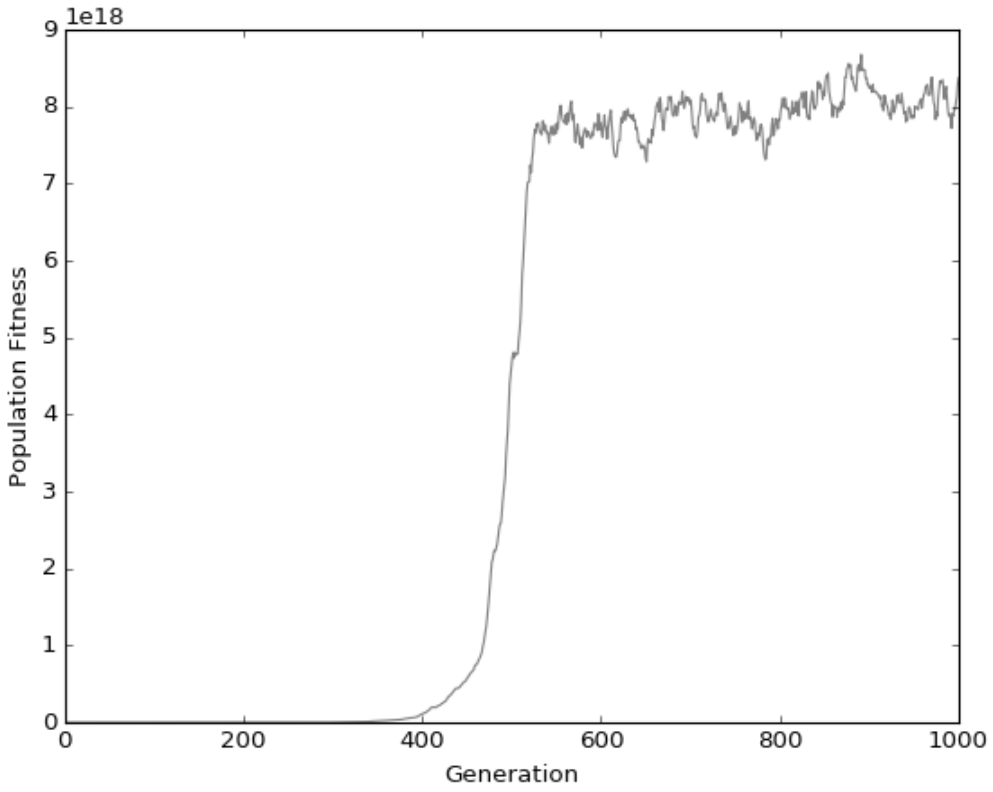
```
FI of Larger Protein Family
89.4534957295
```

```
In [134]: # NOTICE: This experiment takes ~2 hours on a
          # MacBook Pro 2.4 Ghz Core i5 Laptop with 8gb memory
          # Run a selection simulation for the larger protein family with
          # f_lower=100, f_upper=10 and L=150.

          S = SelectionSimulation(CodonSimulation(L=L, mutrate=0.01), \
                                  popsize=2000, \
                                  f_lower=D_wide, f_upper=D_HF, \
                                  f_function=lambda x: 1.5**x-1)
          MISA_positive_selection_wide = []
          for i in xrange(0, G):
              pop_fit = S.sample()
              MISA = S.MISA()
              MISA_positive_selection_wide.append((i+1, pop_fit, MISA))
              if i % 50 == 0: print i,
          print
```

```
0 50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900 950
```

```
In [139]: # Plot generation vs. population fitness on a log plot

          plt.figure()

          iters, popfitness, _ = zip(*MISA_positive_selection_wide)
          plt.plot(iters, popfitness, c='0.5')

          plt.xlim(1, G)
          plt.ylabel("Population Fitness")
          plt.xlabel("Generation")
```

```
<IPython.core.display.Javascript object>
```




```
Out[139]: <matplotlib.text.Text at 0x121eda0d0>
```

Supplementary Figure 3

Even with a larger space of functional sequences, the population converges to a high level of fitness corresponding to an average distance of 11 aa from the centroid sequence.

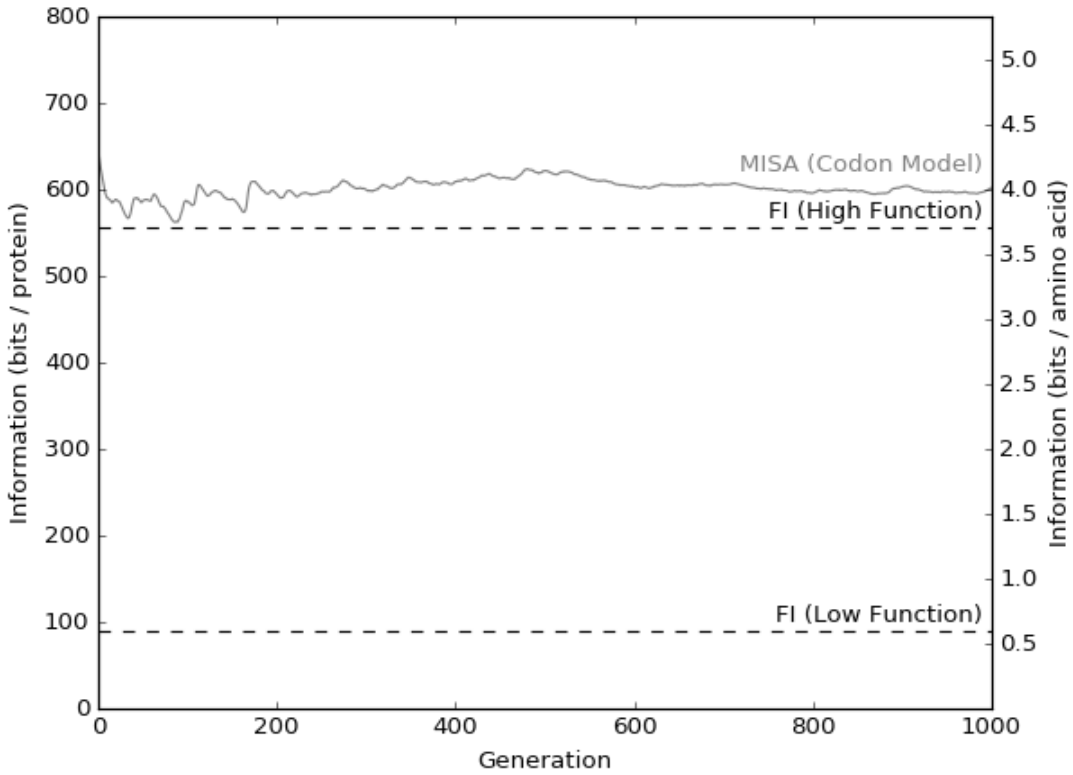
```
In [140]: iters, popfitness, _ = zip(*MISA_positive_selection_wide)

          avg_fitness = np.mean(popfitness[-50:])/2000+1
          print 100 - np.log(avg_fitness) / np.log(1.5)
```

```
11.3679122901
```

```
In [143]: # Plot generation vs. MISA for the positive selection simulation
          # using the larger family with f_lower=100. Also plot FI of the
          # larger protein family and FI of highly functional proteins
          # within the family

          plt.figure()
          ax = plt.subplot(111)

          iters, _, MISAs = zip(*MISA_positive_selection_wide)
          plt.plot(iters, MISAs, c='0.5')
          plt.text(G-10, 620, "MISA (Codon Model)", \
                   horizontalalignment='right', color='0.5')

          plt.axhline(FI_HF, xmin=0, xmax=1000, color='k', linestyle='--')
          plt.text(G-10, FI_HF + 10, "FI (High Function)", \
```

```
                horizontalalignment='right')

        plt.axhline(FI_wide, xmin=0, xmax=1000, color='k', linestyle='--')
        plt.text(G-10, FI_wide + 10, "FI (Low Function)", \
                horizontalalignment='right')

        ylim = 800
        aa_ylim = ylim / float(L)

        ax2 = ax.twinx()
        ax2.set_ylim(0, aa_ylim)
        ax2.set_ylabel("Information (bits / amino acid)")
        ax2.set_yticks(np.arange(0.5,aa_ylim, 0.5))

        ax.set_ylim(0, ylim)
        ax.set_xlim(0, G)
        ax.set_ylabel("Information (bits / protein)")
        ax.set_xlabel("Generation")
```

```
<IPython.core.display.Javascript object>
```




```
Out[143]: <matplotlib.text.Text at 0x11efa9f90>
```

Supplementary Figure 4

Under positive selection, MISA estimates the FI of only high function proteins even when the protein family size is increased by 64 orders of magnitude. In this case, MISA underestimates the total number of functional proteins by over 150 orders of magnitude.

```
In [145]: # For the larger protein family (f_lower=100)
          # Compute MISA predictions of FI and number of functional proteins
          # Compare to the true FI and number of functional proteins

          _, _, MISAs = zip(*MISA_positive_selection_wide)
          MISA_ps = scipy.stats.gmean(MISAs[-100:])
          E_Mx_ps = Np * 2**(-MISA_ps)

          print "Np    =", float(Np)
          print
          print "MISA Approximations:"
          print "FI    ~", MISA_ps
          print "Mx    ~", E_Mx_ps
          print "P(Fx) ~", E_Mx_ps / float(Np)
          print
          print "True Functional Protein Count"
          print "FI    =", FI_wide
          print "Mx    =", float(Mx_wide)
          print "P(Fx) =", Mx_wide / float(Np)
          print
          print "High Function Protein Count"
          print "FI    =", FI_HF
          print "Mx    =", float(Mx_HF)
```

```
Np    = 1.42724769271e+195

MISA Approximations:
FI    ~ 598.190588246
Mx    ~ 1.20556092688e+15
P(Fx) ~ 8.44675337741e-181

True Functional Protein Count
FI    = 89.4534957295
Mx    = 1.68389077623e+168
P(Fx) = 1.17981677941e-27

High Function Protein Count
FI    = 555.749665709
Mx    = 7.19747045729e+27
```