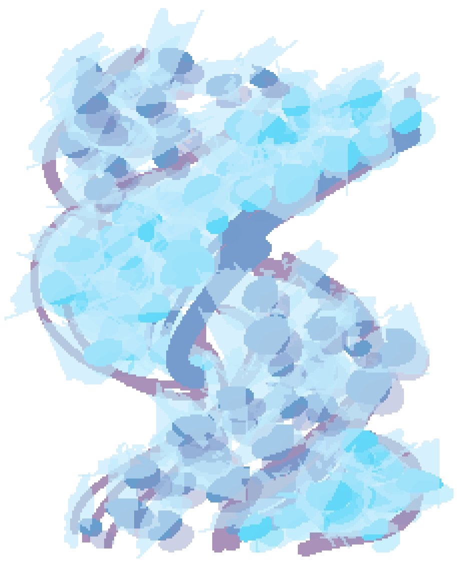
Alignment and Detection of Syntenic Regions of Genes to Identify Horizontally Transferred Islands in Pathogenic Bacteria

by

Stephanie Amber Wilson



A thesis submitted to the Faculty of the Undergraduate School of the University of Colorado in partial fulfillment for the degree of Bachelor of Science

Department of Computer Science

2007

This thesis entitled:

Alignment and Detection of Syntenic Regions of Genes to Identify Horizontally Transferred Islands in Pathogenic Bacteria

written by Stephanie Amber Wilson

has been approved for the Department of Computer Science.

Rob Knight

Clayton Lewis

Michael Main

Date ___________

Abstract

Stephanie Amber Wilson (B. S., Computer Science)

Alignment and Detection of Syntenic Regions of Genes to Identify Horizontally Transferred Islands in Pathogenic Bacteria

Thesis directed by Professor Rob Knight

Alignment and detection of syntenic regions of genes can lead to a better understanding of evolution, both at the species level and at the level of individual genes. In particular, understanding the evolution of genomic features can lead to a better understanding of the genetic changes involved in pathogenicity. Knowing when genes were gained or lost, and whether they were gained or lost together, is important for understanding and predicting their association in functional pathways. In this thesis, I compared the compositional statistics of horizontally transferred islands to the species through codon usage biases and GC content. I also investigated the accuracy of algorithms to align circularly and randomly permuted simulated genomic islands. Detection of horizontally transferred genes will aid in phylogenetic research because these genes violate the mathematical models of sequence evolution typically used for phylogenetic reconstruction. Furthermore, since many of the genes horizontally transferred are involved in antibiotic resistance and pathogenesis, better methods of detection will also aid in medical research.

I dedicate this thesis to my grandmother.

Acknowledgements

I would like to thank my advisor Rob Knight for suggesting I look into algorithmic problems such as circular permutations, and for all his help with my thesis. Furthermore, I would like to thank him for helping with me with my UROP, and HHMI grant work. Each one of the fingerprints was possible due tothe groundbreaking research of Nuboru Sueoka and coworkers., I would like to thank Noboru for taking the time to explain his research. Also I would like to thank my other thesis committee members, Clayton Lewis and Michael Main, for their continuous support and time. Additionally, I would like to thank Jesse Zaneveld, Jeremy Widmann, Micah Hamady, Sandra Smit, Mike Robeson, Catherine Lozupone, and the Knight Lab for all their help.

Table of Contents

Introduction

Bacterial pathogens are currently of increasing medical importance and interest because of antibiotic resistance and the threat of bioterrorism. Many of the genes involved in antibiotic resistance and pathogenesis, are horizontally transferred between different species. Because species vary widely in their GC content and codon usages, a transferred gene often differs from the rest of the genes in the species. However, sequences equilibrate to the background composition of the genome relatively quickly, limiting the utility of methods based on compositional differences. Alternative methods include alignment of syntenic regions (regions where homologous genes appear in the same order) through scored dynamic programming algorithms for permuted islands. Better detection of horizontally transferred genes would both aid in medical research and allow a more accurate estimation of the abundance and importance of horizontal gene transfer as an evolutionary process.

Biological Background

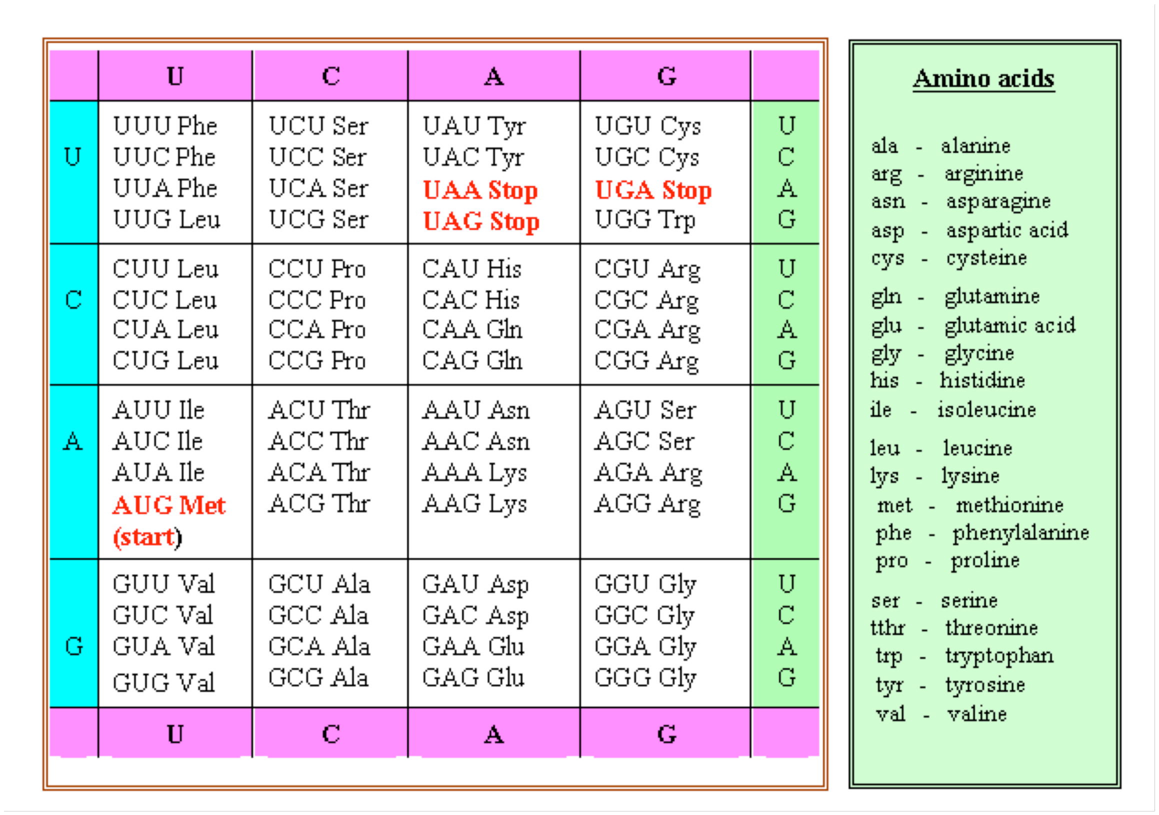
$$\begin{tikzcd}
\text{DNA} \arrow[r, leftrightarrow] & \text{RNA} \arrow[r] & \text{Protein}
\end{tikzcd}$$

Figure 1: The central dogma of biology: DNA is transcribed to RNA and RNA is translated into protein. Some viruses, like HIV, can copy RNA back to DNA, but translation of protein back to nucleic acid has never been observed.

Hereditary information, passed on from one generation to the next, is composed of deoxyribonucleic acid (DNA). The flow of information involved in producing protein from DNA follows what is known as the Central Dogma of Molecular Biology. Specifically, the Central Dogma of Molecular Biology is: DNA is transcribed to ribonucleic acid (RNA), RNA is then translated into protein (Figure 1). Proteins made in this process function in many different ways in organisms including, but not limited to, cell structure, regulators of cell functions, and enzymes. Essentially all phenotypes are influenced by the proteins expressed from the transcription of DNA.

Deoxyribonucleic acid has a simple, repeating structure, and can be considered analogous to a string with a character set of four letters. The letters (AGCT) symbolically represent the repeating nucleotides Adenine, Guanine, Cytosine, and Thymine. DNA in the cell is usually double-stranded, consisting of two antiparallel strands connected in a structure akin to a spiral staircase called a double helix. Each of the characters (nucleotides) is matched with another. Adenine and Thymine form base pairs using specific patterns of hydrogen bonds, as do Cytosine and Guanine. With this simple structure of repeating chemicals, cells can code for every protein needed for the functions of life.

RNA is similar to DNA in that it can be symbolically represented as a string on an alphabet of four characters. However, instead of the nucleotide Thymine, RNA has Uracil, so RNA is represented with the characters A, G, C, and U. Using the chain of RNA, proteins can be translated. Three bases pair with an anticodon on a tRNA molecule (a special kind of RNA that acts as an adaptor), which carries one amino acid, providing the basis for the specificity of the genetic code by linking the amino acid to the antocodon, and hence to the codon. Proteins are chains of amino acids. Each DNA sequence can be translated unambiguously into a protein sequence using the genetic code.



	U	C	A	G	
U	UUU Phe UUC Phe UUA Phe UUG Leu	UCU Ser UCC Ser UCA Ser UCG Ser	UAU Tyr UAC Tyr **UAA Stop** **UAG Stop**	UGU Cys UGC Cys **UGA Stop** UGG Trp	U C A G
C	CUU Leu CUC Leu CUA Leu CUG Leu	CCU Pro CCC Pro CCA Pro CCG Pro	CAU His CAC His CAA Gln CAG Gln	CGU Arg CGC Arg CGA Arg CGG Arg	U C A G
A	AUU Ile AUC Ile AUA Ile **AUG Met (start)**	ACU Thr ACC Thr ACA Thr ACG Thr	AAU Asn AAC Asn AAA Lys AAG Lys	AGU Ser AGC Ser AGA Arg AGG Arg	U C A G
G	GUU Val GUC Val GUA Val GUG Val	GCU Ala GCC Ala GCA Ala GCG Ala	GAU Asp GAC Asp GAA Glu GAG Glu	GGU Gly GGC Gly GGA Gly GGG Gly	U C A G
	U	C	A	G	

Amino acids

ala -	alanine
arg -	arginine
asn -	asparagine
asp -	aspartic acid
cys -	cysteine
gln -	glutamine
glu -	glutamic acid
gly -	glycine
his -	histidine
ile -	isoleucine
leu -	leucine
lys -	lysine
met -	methionine
phe -	phenylalanine
pro -	proline
ser -	serine
tthr -	threonine
trp -	tryptophan
tyr -	tyrosine
val -	valine

Figure 2: This table shows the genetic code, the mapping between specific strings of three nucleotides and amino acids (Cuschieri, n.d.).

Chemically, not all base changes are equal. Since some of the bases are structurally more similar to one another (Figure 3) they are more likely to change from one to another within the DNA sequence (Watson, 1953). Changing purine to pyrimidine and *vice versa*, known as transversions, are more difficult than changing the bases within their purine and pyrimidine groups, known as transitions. Because transversions are chemically less likely (they cannot occur spontaneously by changing the structure of the base *in situ*, and require the polymerase to make a mistake during replication), they occur less than half as often as transitions do, even though twice as many kinds of transversions are possible. (Bos, 2004).

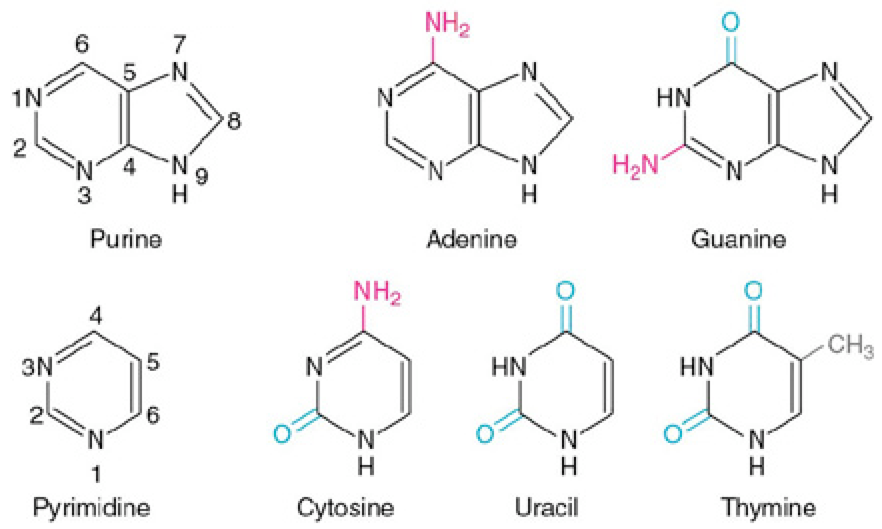


Figure 3: The lewis structure of nucleotides (Weaver, 2005).

Evolution, i.e. mutation of the bases followed by the fixation of the mutant form in the population, is a slow process. For example, the change from UAC to UAA is a change from the amino acid tyrosine to a stop codon that terminates translation (Figure 2). If this happened halfway between the start codon and the original stop codon, the gene sequence (also known as an open reading frame) would now only code for half the protein. This change could lead to a non-functional protein, or to a protein missing parts of the sequence required for normal regulation. The change between UAC and UAA involves a transversion, and thus is less likely to occur than the changes that involve transitions. On the other hand, we can change AGU to AGC and still code for the amino acid serine. In contrast to the previous example, this base change is a transition and therefore more likely to occur. Interestingly, the genetic code is arranged in a way that minimizes the average effects of the mutations, and transitions are less likely to lead to a radical change in amino acid properties than transversions (Freeland, 1998) .

Taking a closer look at the amino acid table, whenever the last nucleotide in the sequence of three changes, the same amino often is coded, especially if the change is a transition. Correspondingly, when sequences in different species are compared, the rate of change is higher at the third codon position because the change at the DNA level often leaves the protein unchanged, and transitions are more likely to be observed (Sueoka, 2002). However, because mutation acts at the DNA level, it is independent of the translated reading frame. Here we see the effects of natural selection, because more changes at the first and second position are lethal and not passed on to the future generations. Different genes also change at different rates, known as substitution rate heterogeneity, which is also due to the combination of mutation and selection (Bos, 2005). These differences in substitution rates lead to codon biases, which can be used to derive a "fingerprint" of the processes operating in a given species (Sueoka, 2002). In other words, different species that share the same genetic code are more likely to use one codon or another for a given amino acid, in part because these codons are produced or destroyed at different rates in different species. Many different factors, including replication, reactive oxygen species, and deamination due to the DNA being single-stranded affect the mutation pattern and hence the codon usage is characteristic for each species. Because of this species-specificity, we can use patterns in codon usage to identify genes that have moved between different species.

Horizontally Transferred Genes

Many bacteria are free-living in the environment. In fact, only one percent of bacteria can be cultured in the laboratory (Pace, 1997). Some bacteria, however, can develop a symbiotic relationship with other hosts. Many bacteria have no effect on their hosts, and others benefit their hosts. For example, our gut bacteria outnumber our own cells by a factor of ten, and provide us with many metabolic capabilities we would otherwise lack. However, some bacteria harm their hosts. These bacteria are known as pathogenic bacteria. Examples of pathogenic bacteria include: *Clostridium botulinum* (responsible for botulism)*, Francisella tularenis* (tularemia)*, Bacillus anthracis* (anthrax), and *Salmonella enterica* (salmonellosis and typhoid fever, depending on strain). In each of these species we can see genes and, consequently, proteins, that allow the bacteria to escape the immune system or antibiotics, or to enter areas harmful to their hosts.

Clostridium botulinum bacteria produce the botulinum neurotoxin (BoNT) that causes botulism (Binz et al.,1994). Some consider BoNT to be the most poisonous substance to humans, estimating BoNT to be 100,000 to 3 million times more poisonous than sarin (a chemical weapon produced, but not used, by Nazi Germany during WWII) (Zubay et al., 2005). BoNT will (irreversibly) bind to the gangliosides and protein receptors on the presynaptic membrane then enters the cell through endocytosis (Hanson et al., 2000). Once in the cell, BoNT toxically cleaves SNARE proteins (Chen et al. 2007). *Clostridium botulinum* is found in the soil and the gastrointestinal tract of human and animals (FDA, 2006). They are classified into four groups based on the BoNT toxin (FDA., 2006). These groups are separated into two types: proteolytic types (i.e. types that can produce enzymes that cut proteins in specific ways that are required to convert the proteins into their active forms), which produce their own BoNT, and nonproteolytic **types, which** use a host protease, such as trypsin, to cleave the inactive holotoxin to make BoNT) (Simpson et al., 2001, Singh, 2000). With treatment, the mortality rate for food-borne botulism has dropped from 25% in the 1950s 1950-1959 to 6% in the 1990s (Zubay et al., 2005).

Francisella tularensis is a bacterium that is potentially very dangerous (The US Centers for Disease Control say it has a “potential for a major public health impact”). Although seen infrequently in the environment, *F. tularensis* is passed through other infected hosts (bugs, animal bites, etc) and contaminated water (CDC, 2003). Concern regarding the bacteria stem from its lethal characteristics. *Francisella tularensis* is able to infect macrophages (a type of white blood cell involved in the immune response) and parenchymal cells (cells that make up organ-specific structures, e.g. in the liver) through unknown mechanism (Fortier, et al., 1994). The availability of antibiotics reduced mortality from 50% to 2.5% (Zubay et al., 2005).

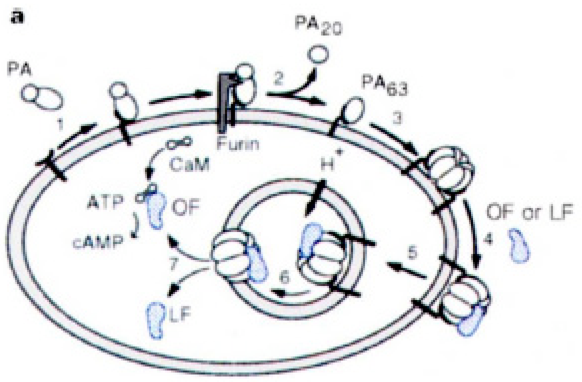


Figure 4: The complex of membrane binding proteins produced by *Bacillus anthracis* (Petosa, et al., 1997)

One of the oldest recorded bacteria (symptoms are referred to in the 5th, 6th, and 10th Egyptian plague, and in the bible) is *Bacillus anthracis,* which causes the disease anthrax (Witkowski, et al., 2002). This species is among the most likely to be used in terrorist attacks due to its hardy spore, which is resilient to cold, heat, radiation, drought, and water (Zubay et al., 2005). *Bacillus anthracis* has a protein capsule for protection against phagocytosis (i.e. the cells in the immune system cannot engulf it for digestion as they do for other bacteria) (Kozel et al., 2007). Also, anthrax forms a transmembrane transporter called PA, and two toxins called LF, and EF. PA binds to the cell membrane (Varughese, et al., 1999). With the addition of LF and EF, PA forms a channel for LF and EF transportation to the cytosol (Petosa, et al., 1997). EF catalyzes the reaction from ATP to cAMP, raising the levels of cAMP in the body (Leppla, 1984). High levels of cAMP limit the inflammation response necessary for the immune system response, and also affect the water homeostasis of the body causing edema (water retention and swelling) (Leppla, 1984). LF toxins cleave proteins in the cells, leading to cell lysis (Klimpel, et al., 1994, Vitale, et al., 2000).

Many diseases previously believed to be independent of bacteria are now being linked to bacterial causes. Once, it was believed that stomach ulcers were solely based on stress on the body. Despite the stomach's low pH, often in the range 0-1 (about the same as 1M hydrochloric acid), it was known that many bacteria could survive in the stomach, including *Helicobacter pylori*. However, bacteria were never linked to ulcers, which were assumed to be caused by stress. In 2005, Barry J Marshall and J. Robin Warren were awarded the Nobel prize for discovering *Helicobacter pylori's* role in gastritis and peptic ulcer disease (Marshall, et al., 2005, Peterson, 1991). In addition to its role in ulcer formation, Helicobacter pylori has also been linked to gastric carcinoma (Parsonnet, 1991).

As discussed earlier, many bacteria are free living, and many others live in symbiosis with their host. For a bacterium to become pathogenic, it must evolve genes that encode proteins that are harmful to the host. To obtain these genes, bacteria must either change their current genes or acquire these genes elsewhere. Mutations that alter a specific phenotype occur at a staggeringly slow rate (Hartwell, 2004). One study of millions of mice found the rate of mutation was 1×10^{-5} **per** nucelotide per generation (Hartwell, 2004), although bacteria are thought to mutate more rapidly and have much larger population sizes. However, rates of horizontal gene transfer (the movement of genes between different genomes) are also thought to be high in bacteria. For example, of the genes in the first three strains of *E. coli* to be sequenced, only 40% of the genes were in all three strains (Welch, et al., 2002). Thus, the motivation to study processes of horizontal gene transfer is strong.

When bacteria acquire new genes from another species it is known as horizontal transfer. To study horizontally transferred genes (HTGs), an island, or contiguous group of genes is most often investigated. Mainly, this is because the larger sample size reduces the chance of seeing extreme composition through sampling error. Thus we can have higher confidence that the island came from another species.

Compositional Statistics

Homologous genes of *Salmonella* and *Escherichia* average 85 percent sequence identity (Bräumler 1997, Sharp 1991), suggesting that about 100-160 million years ago their lineages diverged (Bräumler 1997, Ochman 1987). 10-20 percent of the *Salmonella* genome is absent from *E. coli,* also suggesting horizontal transfer (Bräumler 1997, Ochman et al. 1996 (1), Lan et al. 1996). Some specific examples of transferred genes include type three secretion system (T3SS) proteins, which are often found in pathogenicity islands in the genome or in plasmids, and which are often similar in species whose genes are otherwise dissimilar (Collmer et al. 2002), suggesting that HGT explains their distribution (Gophna et al. 2002). Often the transferred genes differ from the rest of the genome in their GC content (Ochman 1996 (2)) Codon fingerprints differences due to mutational pressures are specifiespecific species (Sueoka 2002), and might provide more power than GC content to distinguish among genes from different sources.

GC Content

Because DNA is base-paired in the cell, one simple measure of the composition of double-stranded DNA is the GC content (the fraction of all the base pairs in the genome that are GC or CG pairs, as opposed to AT or TA pairs). GC pairs are stronger than AT pairs because three hydrogen bonds hold them together rather than two. Thus, it is harder to separate the strands of DNA with more GC pairs, leading to a difference in melting temperature. The GC content of a nucleic acid could thus be measured biochemically in the 1960s, almost two decades before sequencing technologies became available.

Different species vary widely in GC content. This variation (from about 25% to 75% GC) is probably driven by mutational processes, and is so consistent in different species that extremely accurate predictions about the usage of different amino acids and different codons in the genome can be made from the GC content along (Knight et al., 2001). Because of these differences in different organisms, the GC content has been widely used as a marker for genes that might have come from another genome.

A paradigmatic example of horizontal transfer is that of the Type Three Secretion Systems (T3SS). The genes in SPI-1, one of the Salmonella Pathogenicity Islands, are distinctly different in GC content from the rest of the genome (Ochman

1996). Additionally, some transferred genes involved in the T3SS appear to have been horizontally transferred by bacteriophages that pick up genes from one species and integrate into another, causing not only differences in GC content due to the viral replication mechanism but also allowing identification through the presence of specific flanking sequences that the virus uses for integration into the genome and for packaging (Ehrbar 2005). Similar T3SS components have been found in extremely different types of bacteria, e.g. proteobacteria and the chlamydia group, which further suggests that these genes have been horizontally transferred (Collmer 2002).

CAI

The Codon Adaptation Index (Sharp, 1986) is a measure of how closely a gene's codon usage (i.e. the frequency with which each of the different codons is used in the genome, correcting for the fact that different amino acids are encoded by different numbers of codons) resembles the codon usage of highly expressed genes in the genome. The codon usage is compared to the codon usage of a set of reference genes that are known to be highly expressed, such as the ribosomal proteins. In a given species, there is usually a monotonic relationship between CAI and the GC content of each gene, so that outliers from this plot might indicate that genes came from elsewhere.

Fingerprint

The codon fingerprint (Sueoka 2002) is a method of displaying synonymous codon usage graphically. If only mutational processes were at work, and if mutation at each position were independent of mutations at other positions, we would expect four-fold synonymous sites (i.e. sites of the form XYN where X and Y are each a specific nucleotide and N is any nucleotide, and where all codons XYN encode the same amino acid) to have the same composition at the third position no matter what the first two bases are. The fingerprint plot graphs the relative usage of the purine and the pyrimidine in each type of pair against each other for each amino acid, i.e. plotting (G/(G+C)) against (A/(A+T)). Different species have characteristically different fingerprints, again suggesting that this feature might be useful for identifying horizontally transferred genes.

Permuted Islands

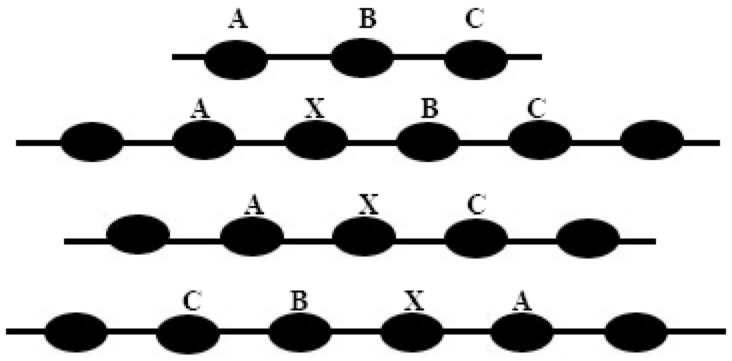


Figure : The top string represents a section of a gene sequence with each letter being a gene. The second strings shows an insertion of the gene ‘X’. The third string show the second string undergoing a deletion of the 'B' gene. In the last string we see a reversal of the second string.

Genomic islands that have been horizontally transferred do not always retain the same arrangement of genes that they had in their originating species. Mutations can occur at the level of whole genes as well as at the level of individual bases. Within sequences, mutations can cause events such as insertion, deletion, and reversals of the genes (Figure).

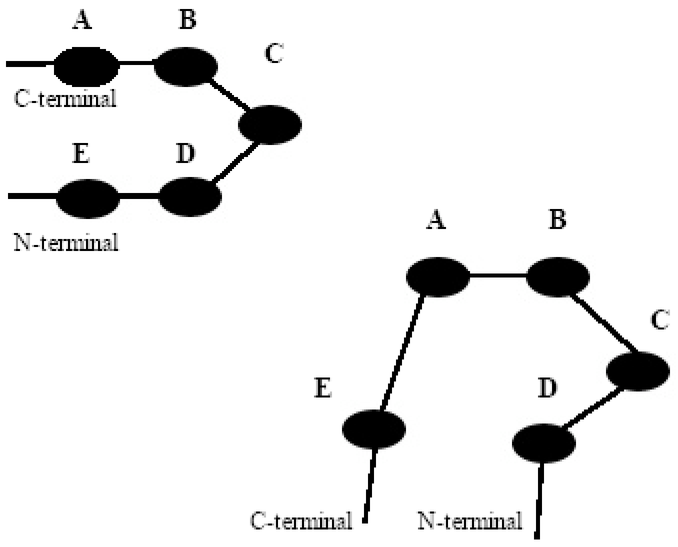


Figure : The top gene sequence is circularly permuted to the bottom moving the gene 'F' from the end of the n-terminal to the beginning of the c-terminal.

DNA sequences, and therefore protein sequences, can also be circularly permuted so the sequence at the end (N-terminus in proteins; 5' end in DNA) can be moved to the beginning of the sequence (C-terminus in proteins' 3' end in DNA) (Figure). This type of circular permutation is especially common in plamids, which usually exist in the cell as circular DNA but can integrate into the larger genomic sequence using several different sites. Interestingly, many proteins and RNAs can maintain their 3D structure and thus their function after undergoing circular permutation, and this process has accordingly been observed in proteins from different species. (Uliel, et al., 1999). Circular permutation is observed both within a single protein sequence, and between groups of genes. Because circular permutation is important for gene transfers that are mediated by plasmids, detecting that one putative genomic island is an approximate circular permutation of another tells us something about the likely mechanism of transfer.

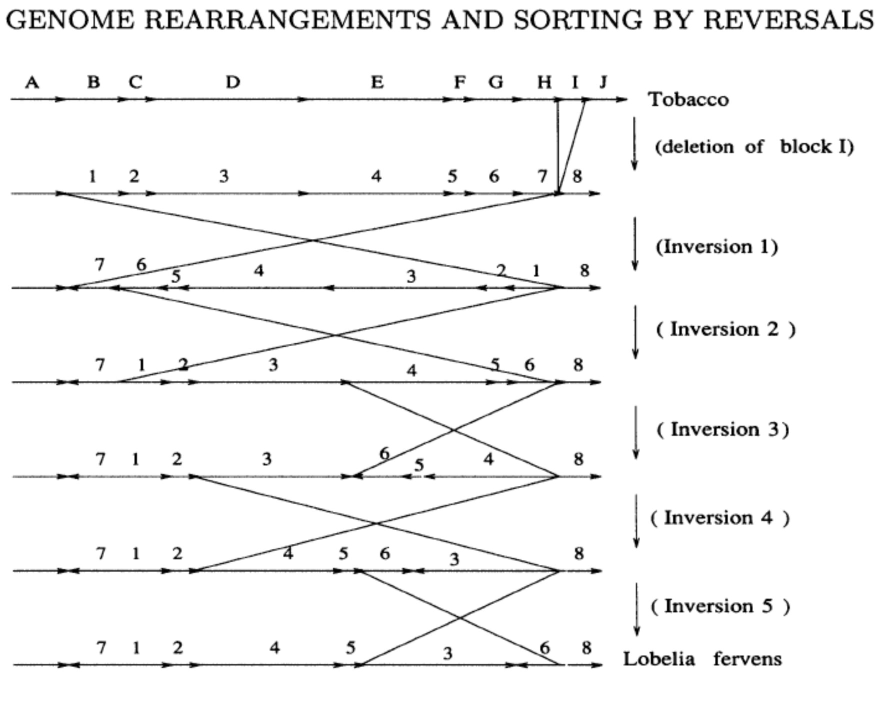


Figure : Each of the lines show insertions or reversal that may have occurred during the evolution of Tobacco to *Lobelia fervens* (Knox, et al., 1993)

Permuted islands are also of interest because they offer a means of tracking the evolutionary history of a species: like sequence characters in an alignment, breakpoint characters can be used for phylogenetic reconstruction. The process of tracing the permutations that occur over time as one species evolves into another is known as breakpoint analysis. Through breakpoint analysis evolutionary paths can be seen in the chloroplast of the tobacco plant to *Lobelia fervens* (Figure) (Knox, et al., 1993).

Existing Methods for Detecting Permuted Syntenic Regions

The Needleman-Wunsch algorithm is a dynamic programming algorithm that can be used to calculate the minimum edit distance between two sequences in terms of insertions, deletions and substitutions. This algorithm performs a global alignment on two sequences that spans the entire length of the sequence. A related algorithm called Smith-Waterman uses a similar technique to perform local alignments, the main difference between the two being the initialization conditions. Dynamic programming algorithms reducing the overall problem to overlapping subproblems for optimization, allowing reduction of the time complexity from exponential to quadratic (Korf, et al., 2003).

The Smith-Waterman and Needleman-Wunsch algorithms are widely used for sequence alignment and are implemented in many bioinformatics software packages. Basically, the way the Smith-Waterman aligns genes is as follows: the matrix is initialized to all zeros. Each cell in the matrix corresponds to an alignment between position i in the first sequence and position j in the second sequence. Starting at the cell one position down and one position to the right from the top-left corner of the matrix, the optimal score for each cell is calculated, working along to the right and downwards. For each cell, the score of aligning the two positions is given by the score for matching the base at position i against the base at position j according to a score matrix (the simplest score matrix consists of a constant match score, given if the two bases are identical, and a constant mismatch penalty, given if the two bases are different, but there are many different score matrices in use for different applications), and adding this score to the score of the cell immediately above and to the left of the current cell. The score of introducing a gap is given by calculating the gap penalty and adding this penalty to the cell above the current cell (for introducing a gap into the first sequence), or to the left of the current cell (for introducing a gap into the second sequence). The maximum of these three possible scores is recorded in the cell, along with a pointer to the cell that gave the best result (in the case of ties, one of the best cells is chosen arbitrarily). When the bottom-right corner of the matrix is reached, the highest cell in the entire matrix is found and used as the start of the alignment. The pointers are then followed and used to introduce gaps into one sequence or the other until the next score drops to 0, at which point the alignment is terminated (Korf, 2003).

To analyze the effectiveness of the Smith-Waterman algorithm for recovering syntenic regions, sequence lengths were tested with one to twenty random insertions one thousand times without a circular permutation. When plotting the data as scatter points (figure 2), a correlation can be seen between the accuracy and sequence length and number of insertions.

Circular Permutation

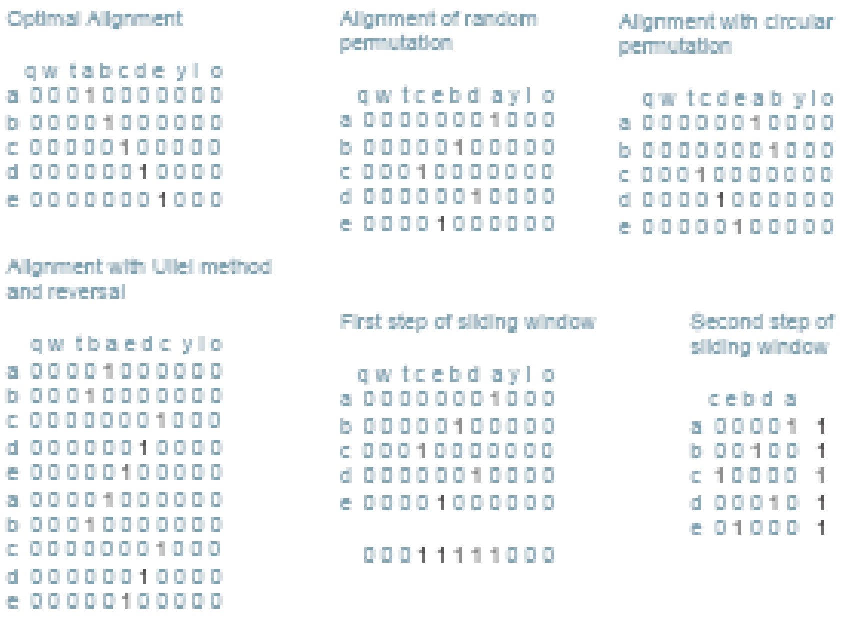
Although the Smith-Waterman algorithm is useful for alignment of sequences with insertions and deletions, it does not account for circular permutations. However, Uliel saw the potential application of this algorithm for sequences with circular permutations. Given the sequence ‘abc’, a circular permutation event could lead to the transformation to the sequence ‘cab’. Comparison of the two sequences through the Smith-Waterman algorithm would lead to the ‘ab’ alignment. However, appending a copy of the original sequence to itself leads to an instance of each possible circular permutation inside the new string (e.g. “abc” and “abc” lead to “ab**cab**c”, which contains the ‘cab’ sequence we are looking for), and thus allows the alignment of a circularly permuted sequence (Uliel, et al., 1999).

Sliding Window

The sliding window approach is also a useful approach to many problems that rely on underlying score matrices. In a sliding window approach, k consecutive elements within a longer sequence of length n are analyzed at a time, advancing one element at each step. For example, the sequence composition analyzer computes the

GC content of each window of a DNA sequence in order to find regions that are of interest because their GC content differs from that of the rest of the genome (SeqComp, n.d.).

To analyse permuted windows, we treat each gene as a single character on a large alphabet, and calculate how many of these characters each window in the first sequence and each window in the second sequence have in common. Windows that contain many of the same genes are likely to represent permuted islands. If the sequences align exactly, the raw score matrix matching each gene against each other gene contains all matches (1s) on the diagonal, and non-matches (0s) elsewhere (Figure first matrix). If the sequence has been circularly permuted, there will be two disconnected diagonal matches (Figure third matrix). if the sequence has been reversed (e.g. because the genes integrated on the opposite strand, or underwent an inversion after integration), a diagonal line in the opposite orientation to the main diagonal will be observed (Figure fourth matrix)


Optimal Alignment

```
  q w t a b c d e y l o
a 0 0 0 1 0 0 0 0 0 0 0
b 0 0 0 0 1 0 0 0 0 0 0
c 0 0 0 0 0 1 0 0 0 0 0
d 0 0 0 0 0 0 1 0 0 0 0
e 0 0 0 0 0 0 0 1 0 0 0
```

Alignment of random permutation

```
  q w t c e b d a y l o
a 0 0 0 0 0 0 0 1 0 0 0
b 0 0 0 0 0 1 0 0 0 0 0
c 0 0 0 1 0 0 0 0 0 0 0
d 0 0 0 0 0 0 1 0 0 0 0
e 0 0 0 0 1 0 0 0 0 0 0
```

Alignment with circular permutation

```
  q w t c d e a b y l o
a 0 0 0 0 0 0 1 0 0 0 0
b 0 0 0 0 0 0 0 1 0 0 0
c 0 0 0 1 0 0 0 0 0 0 0
d 0 0 0 0 1 0 0 0 0 0 0
e 0 0 0 0 0 1 0 0 0 0 0
```

Alignment with Ullel method and reversal

```
  q w t b a e d c y l o
a 0 0 0 0 1 0 0 0 0 0 0
b 0 0 0 1 0 0 0 0 0 0 0
c 0 0 0 0 0 0 0 1 0 0 0
d 0 0 0 0 0 0 1 0 0 0 0
e 0 0 0 0 0 1 0 0 0 0 0
a 0 0 0 0 1 0 0 0 0 0 0
b 0 0 0 1 0 0 0 0 0 0 0
c 0 0 0 0 0 0 0 1 0 0 0
d 0 0 0 0 0 0 1 0 0 0 0
e 0 0 0 0 0 1 0 0 0 0 0
```

First step of sliding window

```
  q w t c e b d a y l o
a 0 0 0 0 0 0 0 1 0 0 0
b 0 0 0 0 0 1 0 0 0 0 0
c 0 0 0 1 0 0 0 0 0 0 0
d 0 0 0 0 0 0 1 0 0 0 0
e 0 0 0 0 1 0 0 0 0 0 0

  0 0 0 1 1 1 1 1 0 0 0
```

Second step of sliding window

```
  c e b d a
a 0 0 0 0 1   1
b 0 0 1 0 0   1
c 1 0 0 0 0   1
d 0 0 0 1 0   1
e 0 1 0 0 0   1
```

Figure : Alignment matrices of various sequences of genes, where each match is denoted as a 1 and each mismatch is denoted as a 0. The last two matrices show the steps involved in the sliding window approach; the first step shows the "or" of the columns at the bottom and the second step shows the 'or of the columns at the right side.

Ideally, we would want to detect all the alignments shown above, and there is one common theme that allows us to dectet these alignments. If the two sequences contain a common, but permuted island, the resulting score matrix (where each match is denoted by a one and mismatch with a zero) contains consecutive rows that are non-empty. Combining the values in each rows using logical or produces a vector with many consecutive oens (Figure fifth matrix). The same is also true for the columns. This characteristic can lead to an efficient method for detecting seemingly random permutations, including insertions and deletions: using a sliding window to sum the nonzero elements in the vector allows us to find regions in which many genes are shared between the two sequences.

Materials and Methods

Compositional Statistics

Matplotlib

The python library Matplotlib was used to graph the results of the statistical analysis of sequence composition. This package was integrated with the Cogent package, developed jointly between the Knight lab at CU Boulder and the Huttley lab at the Australian National University, Canberra. These object-oriented packages allow considerable flexibility for sequence analysis and graphical displays. Other features of the Matplotlib that were used include: LaTeX support, the ability to produce multiple plot types, including histograms, scatter plots, and contour plots, and the ability to programatically generate custom color schemes and legends. The Matplotlib library was written primarily by John Hunter and is an open source project (Hunter, 2007).

Genome Data Sources

Genomic data necessary for calculating compositional statistics came from KEGG: The Kyoto Encyclopedia of Genes and Genomes. KEGG is maintained by the

Bioinformatics Center of Kyoto University and the Human Genome Center of the University of Tokyo(KEGG, 2007). The KEGG database includes the complete genome sequences of many microbial organisms, along with a wealth of additional information useful in interpreting the genome sequence data. Components of the KEGG database include pathways of molecular networks, and information about biochemical reactions and compounds (Kanehisa et al., 2004). KEGG, unlike GenBank, contains the nucleotide sequence of each individual gene paired with the amino acid sequence of the corresponding protein, making the analyses of compositional statistics significantly easier.

Display Construction

One of the challenges of graphing biological data is simplifying it to a readable form. Noboru Sueoka devised a very efficient method, called the fingerprint plot, of displaying the nucleotide biases in the sets of codons encoding amino acids. . In this graph, a colored circle represents each of the eight four-codon amino acids. The radius of each circle is proportional to the ratio of that amino acid to all other amino acids in the gene(s) being investigated. The placement of the circle in the x-axis is dependent on the ratio of G in the third position normalized to the sum of G and C . In the y-axis, the circle is placed depending on the ratio of A in the third position normalized to the sum of A and T in the third position. This fingerprint is characteristic of each species.

Permuted Islands

Circular Permutation Island Algorithm

For alignment of circularly permuted islands the Uliel method was implemented using the Smith-Waterman method from the Cogent library, developed by Jeremy Widmann. The query sequence was concatenated with itself as input, and the output was post-processed to ensure that multiple copies of the same gene were not counted in the result (i.e. that the alignment could use a given gene from the first copy of the sequence or from the second copy, but not from both).

Sliding Window Algorithm

This algorithm was implemented completely from scratch. First, the matrix of matches is created, and then the logical or operation is applied to the columns. Based on variable parameters relating to the number of insertions, deletions, and the desired match length, an ideal length and match ratio is calculated. For example, if the sequence of interest is 10 genes long and we want to account for 2 insertions, the sliding window will be 12 characters long. To account for 2 deletions, if there are 8 or more matches within the window then the index is stored. Each of the indices of interest is stored. However, at this point it is unknown whether the indices represent an alignment or represent a repeat of a single, matching character. The rows must also be examined in order to distinguish these two cases. Each of the indices of interest is used to slice a sub-matrix based on the prior window size calculation. Then, a logical or is performed on the rows of the matrix. If there are enough matches to account for the sequence length minus deletions within the calculated window size, the index of column is stored along with the index of the row in the list of alignments. This list is then returned as a result.

Results

Compositional Statistics

Type III Secretion System

The fingerprint graphs were all produced using the graphing code I created. The data for the graphs came from KEGG, which was tied to the CodonExplorer web interface created by Micah Hamady. CodonExplorer uses my graphing code as a front end for interactive analysis of genomes or user-specified gene sets from KEGG. For each of the islands described below, the location, length, and function were obtained from the review article “Lateral gene transfer in Salmonella” (Pollwollik and McClelland, 2003). The T3SS is of particular interest because it is widely believed to have been horizontally transferred, and because it plays a direct role in *Salmonella’s* ability to act as a pathogen. The T3SS forms a needle complex that allows *Salmonella*, and certain other pathogens, to deliver proteins that they make across the membrane of the host cell, allowing them to manipulate the host cell directly. *Salmonella enterica* serovar Typhimurium (*S.* Typhimurium), the causative agent of salmonellosis (food poisoning), also causes a disease in mice that is equivalent to typhoid fever in humans (the causative agent of typhoid fever is a closely related strain, *Salmonella enterica* serovar Typh). *S.* Typhimurium has two distinct T3SSs: one is used to invade the gut during enteric infections, and the other is used to aid survival inside cells in the body. The two T3SSs are encoded on pathogenicity islands called SPI-1 and SPI-2 respectively; several other pathogenicity islands encode other genes that are involved in pathogenicity for reasons unrelated to type III secretion.

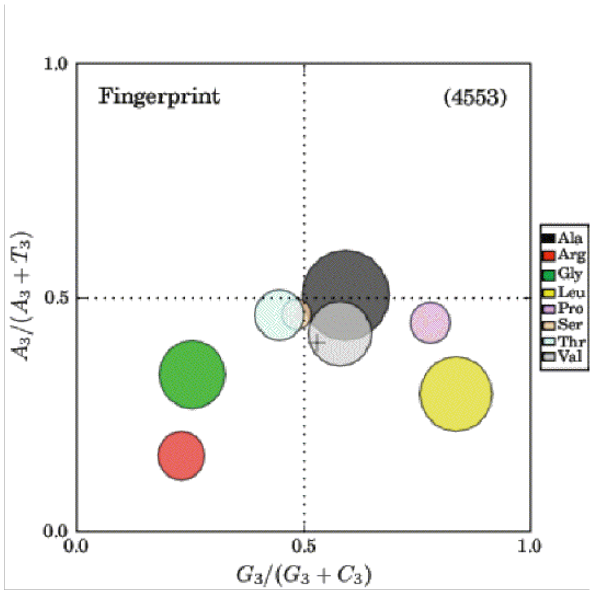


Figure : Fingerprint of the entire genome of *S.* Typhimurium.

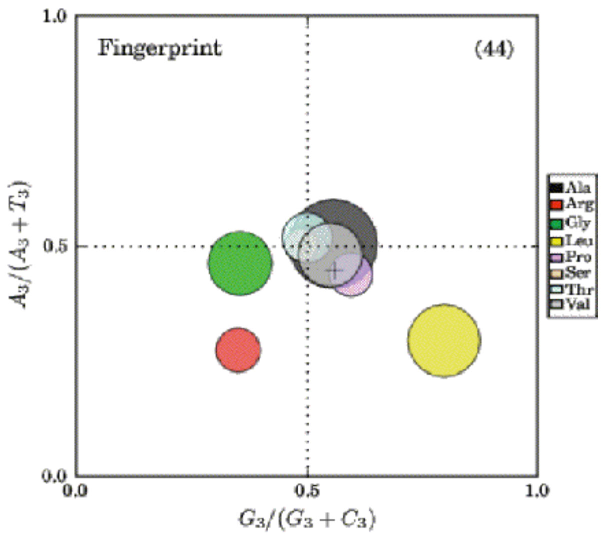


Figure : Genomic island in *S.* Typhimurium starting at gene marker location: STM1379. The genes in this island are involved with SPI2: type III secretion system. Specific functions include: ttr, sse, ssa.

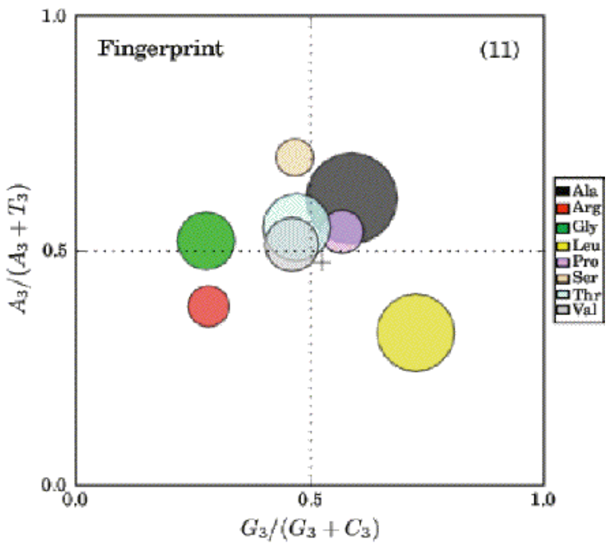


Figure : Genomic island in *S.* Typhimurium starting at STM1087. SPI5: virulence genes, effector protein primarily pip, sopB

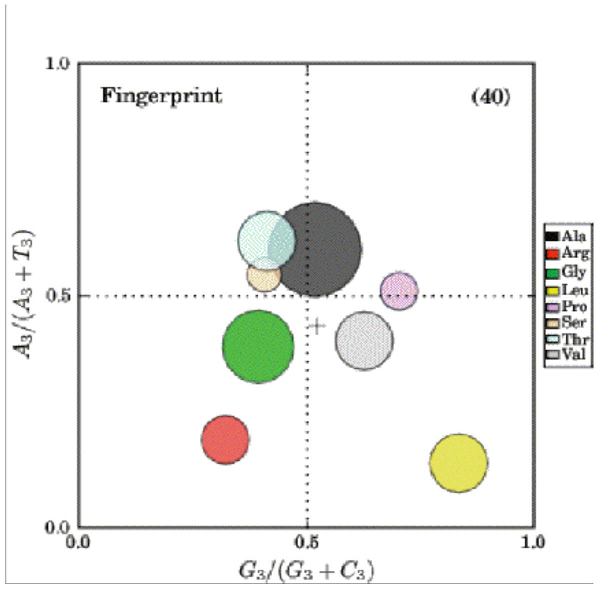


Figure : Genomic island in *S.* Typhimurium starting at STM0893. Prophage Fels-1: superooxide dismutase, neuraminidase primarily sodCIII and nanH.

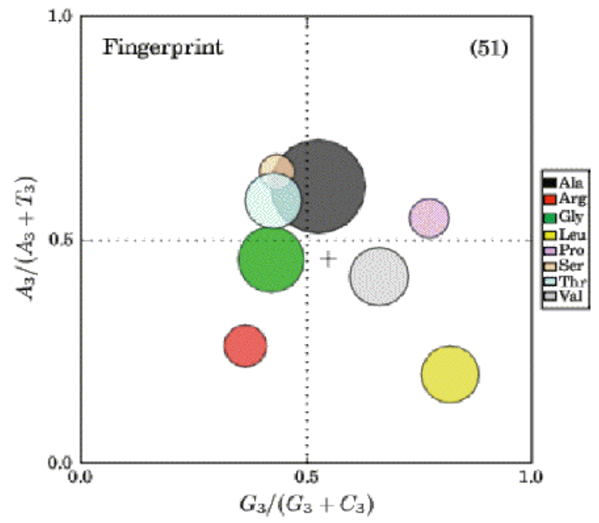


Figure : Genomic island in *S.* Typhimurium starting at STM1005\Prophase Gifsy-2: superoxide dismutase, virulence genes primarily sodCI, grvA, gtgE.

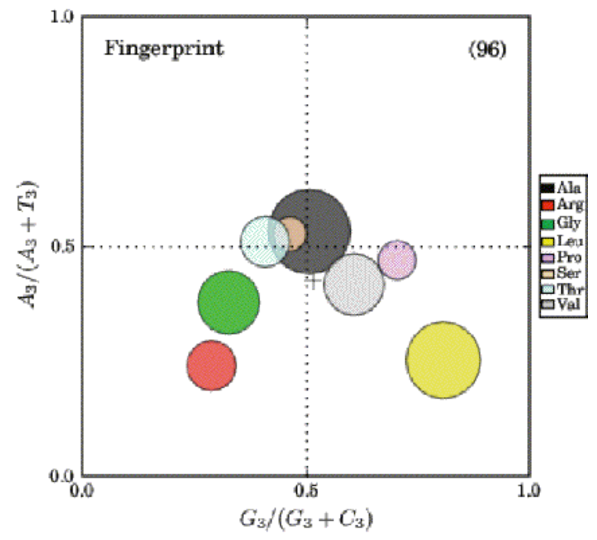


Figure : Genomic island in *S.* Typhimurium starting at STM2689 Prophage Fels-2: PTS system, phase 2 flagellin, H inversion, virulence genes, transporters, siderophores? primarily iro,fljAB,hin,tct

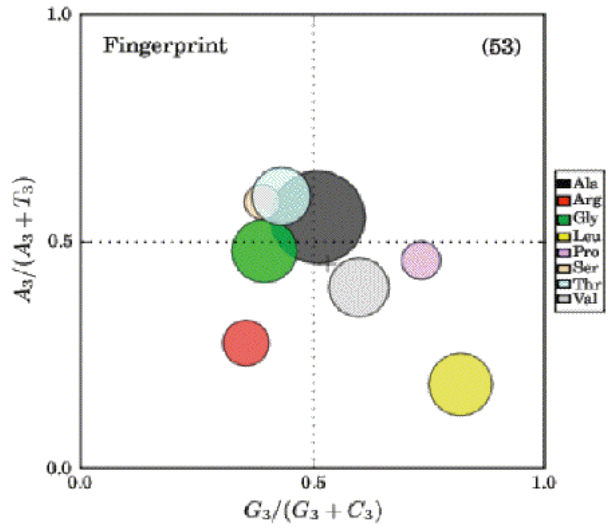


Figure : Genomic island in *S.* Typhimurium starting at STM2584 Prophase Gifsy-1 primarily gipA, gogB

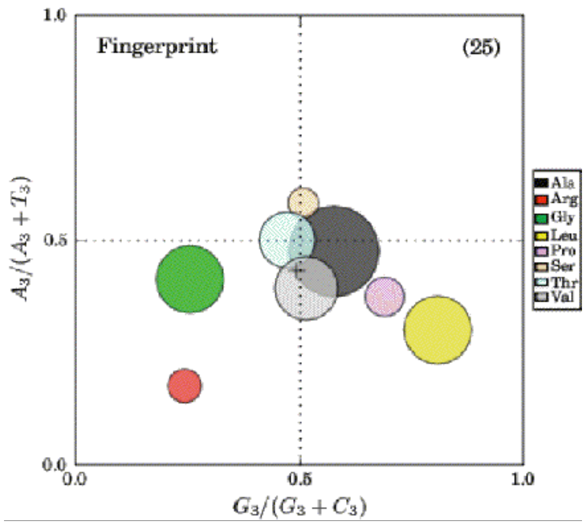


Figure : Genomic island in *S.* Typhimurium starting at STM4195 Prophage

These graphs show a clear difference between the overall species fingerprint and the horizontally transferred genes. Island 1 and 3, and island 2 and 4 show the similarities of similar prophages (these two pairs of islands are associated with particular kinds of viruses that replicate in different ways; a prophage is a virus that has integrated into the genome). Islands 1-4 also differ from island 5, which has a different prophage orgin. Islands 6 and 7 also deviate from the overall species fingerprint. The statistical significance of these results was verified by Jesse Zaneveld in the Knight lab using Monte Carlo techniques (J. Zaneveld, unpublished data). From these examples, we can see that fingerprint difference in genes may serve as an excellent method to detect horizontal transfer.

Permuted Islands

Runtime of Algorithms

Given sequences m and n. the runtime for the Smith-Waterman algorithm is O(mn). From this it can be deduced that the Uliel algorithm is also O(mn), with an additional twofold constant factor. Also, because one of the sequences is repeated, postprocessing is required in cases in which the same gene from both copies of the sequence contributes to the alignments..

Not including creation of the matrices and applying logical or to the matrices, the runtime for the sliding window algorithm is also O(nm) (the sliding window analysis contributes an O(n) step). Implementing the arrays using the Python Numeric libraries rather than using the built-in Python data types resulted in order-of-magnitude improvements in speed.

Algorithm Accuracy

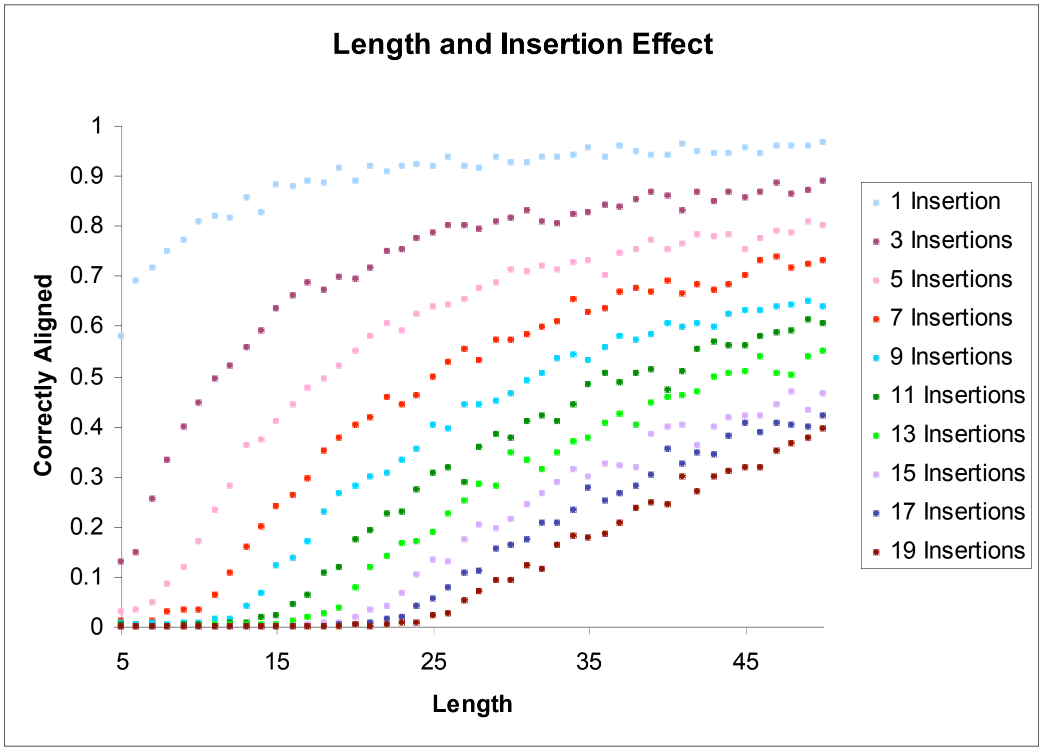


Figure : This graph was generated using the code based on the Uliel method for aligning circularly permuted sequences. Each point represents the correct alignment of an entire island, taken as an average over 1000 randomly circularly permuted samples for a given length of sequence and set number of random insertions.

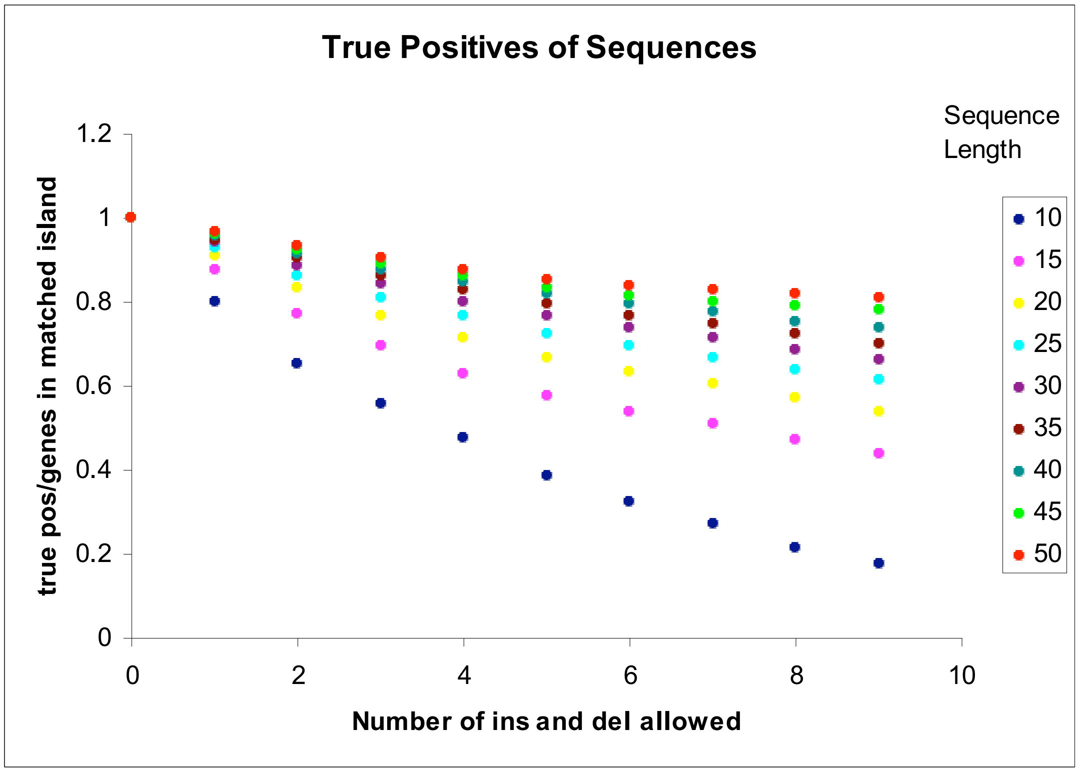


Figure : This graph is based on the sliding window approach to aligning permuted sequence. It shows the true positives ratio of genes found in the aligned island to genes in the actual island. Each point represents an average taken over 1000 samples with a set number of insertions and deletions allowed, random insertions and deletions based up to the allotted number of insertions and deletions, and the set sequence length.

Discussion

Compositional Statistics

Significance of findings

The striking similarities between the fingerprint plots for genes derived from the Gifsy-1 and Gifsy-2 phages, and the Fels-1 and Fels-2 phages, indicate that the mechanism of transfer affects codon usage in a way that can be detected with the fingerprint plots. Furthermore, the similarity of the Prophage islands to each other and the Gifsy to each other suggest that the codon biases are specific to the mechanism of horizontal transfer. The contrast between the T3SS-encoding islands and the genome as a whole also suggests that these techniques may be broadly useful for detecting foreign genes. These results are interesting because the paradigm for most research into horizontal gene transfer to date has been to assume that all the transferred genes will fit the same statistical model that differs from the model for the whole genome. If genes that were transferred by different mechanisms have substantially different compositional signatures, we may be able to detect horizontal gene transfer in a much more sensitive and specific manner by using models that apply to particular mechanisms rather than by treating all transferred genes the same way.

Permuted Islands

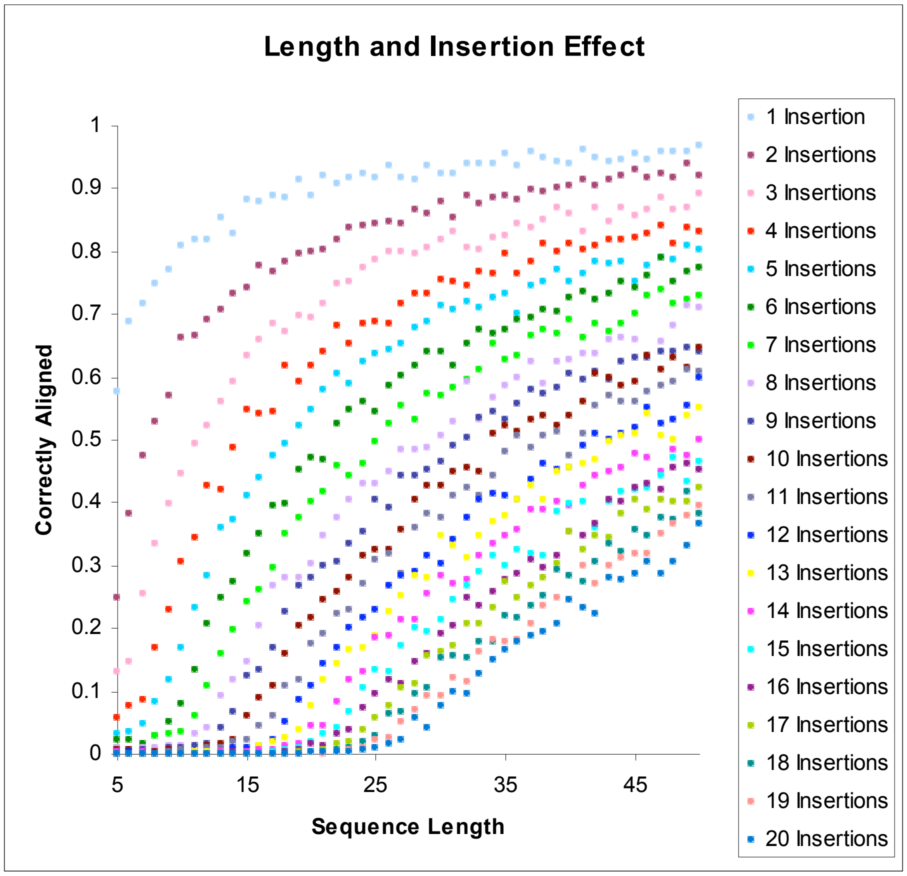
Correlation of Variation and Detection

In the case of the Uliel method of alignment, fewer insertions equated to a better chance of finding the alignment of the island. Because the Smith-Waterman algorithm is score-based, longer sequences improved the accuracy of island detection. The score increases with the length of the sequence, so the impact on the overall score of a single insertion is lower in longer sequneces. This behavior is similar to that of the Smith-Waterman algorithm, so the only real cost of the Uliel method is additional factor of two in the runtime.

As far as the sliding window approach, the more versatile it becomes the more occurring false positive. As the number of insertions and deletions allotted increase accuracy decrease. This makes sense because by allowing insertions and deletions to occur you're diverging from the sequence itself. Similar to the Uliel method, longer sequences lead to better accuracy. In the longer sequences an insertions or deletion has a lower ratio of change to sequence length.

Importance of Parameters

Several parameters affect the sensitivity and specificity of the sliding window approach. The window size was especially important in cases where the matches were relatively short. For example, if the matching sequence were 3 characters long and the window size was 10, the 3 character match would not be not be detected. Similarly, adding the ability to look for additional insertions and deletions increases the sensitivity at the expense of specificity. However, if there are more insertions or deletions than allowed in the search, part or all of the matching island will be descarded. One possible solution to this problem is to fit the parameters based on genomic data, and to adjust the thresholds according to well-established cases where the evolutionary histories of the genomic islands are known. Another possible solution would be to use a smaller window size and to post-process the results.



Conclusions

Horizontal gene transfer is an important process that shapes microbial genomes, and improved techniques for detecting its operation will be useful in many different contexts. In this work, I was able to show that the fingerprint plots are effective for identifying horizontally transferred genes, and to compare several approaches for identifying syntenic regions. This work thus fills in some of the pieces of the large and complex puzzle that is horizontal gene transfer.

Appendix

1. Code implementing the graphs of various compositional statistics

```
#/usr/lib/python2.4/
#AminoGraphPlots.py

from matplotlib import use, rc
use('Agg')  #suppress graphical rendering
rc('text', usetex=True)
rc('font', family='serif')  #required to match latex text and
equations
import Image
import ImageFilter
from Numeric import array, shape, fromstring
from cogent.base.usage import UnsafeCodonUsage as CodonUsage
from cogent.maths.stats.test import regress, correlation
from pylab import *
from math import pi

"""Provides different kinds of codon usage plots.

See individual docstrings for more info.
"""
#module-level constants

#historical doublet order for fingerprint plot; not currently used,
but
#same order that the colors were entered in. Matches Sueoka 2002.
doublet_order =
['GC','CG','GG','CU','CC','UC','AC','GU','UU','CA','AU',\
                 'AA','AG','GA','UA','UG']
color_order = ["#000000","#FF0000","#00FF00","#FFFF00",
          "#CC99FF","#FFCC99","#CCFFFF","#C0C0C0",
          "#6D6D6D","#2353FF","#00FFFF","#FF8800",
          "#238853","#882353","#EC008C","#000099"]
#map doublets to colors so we can make sure the same doublet always
#gets the same colors
doublets_to_colors = dict(zip(doublet_order, color_order))
#creates a dictionary for the amino acid labels, less to input
aaLabels={'ALANINE':'GCN', 'ARGININE4':'CGN', 'GLYCINE':'GGN',
          'LEUCINE4':'CTN', 'PROLINE':'CCN', 'SERINE4':'TCN',
          'THREONINE':'ACN', 'VALINE':'GTN'}
standard_series_colors=['k','r','g','b', 'm','c']

#Helper functions

def hist(x, bins=10, normed='height', bottom=0, \
    orientation='vertical', width=None, axes=None, **kwargs):
    """Just like the matplotlib hist, but normalizes bar heights to
1.
```

```
    axes uses gca() by default (built-in hist is a method of Axes).

    Original docs from matplotlib:

    HIST(x, bins=10, normed=0, bottom=0, orientiation='vertical',
**kwargs)

    Compute the histogram of x.  bins is either an integer number of
    bins or a sequence giving the bins.  x are the data to be
binned.

    The return values is (n, bins, patches)

    If normed is true, the first element of the return tuple will
    be the counts normalized to form a probability density, ie,
    n/(len(x)*dbin)


    orientation = 'horizontal' | 'vertical'.  If horizontal, barh
    will be used and the "bottom" kwarg will be the left.

    width: the width of the bars.  If None, automatically compute
    the width.

    kwargs are used to update the properties of the
    hist bars
    """
    if axes is None:
        axes = gca()
    if not axes._hold: axes.cla()
    n,bins = norm_hist_bins(x, bins, normed)
    if width is None: width = 0.9*(bins[1]-bins[0])
    if orientation=='horizontal':
        patches = axes.barh(n, bins, height=width, left=bottom)
    else:
        patches = axes.bar(bins, n, width=width, bottom=bottom)
    for p in patches:
        p.update(kwargs)
    return n, bins, silent_list('Patch', patches)

def norm_hist_bins(y, bins=10, normed='height'):
    """Just like the matplotlib mlab.hist, but can normalize by
height.

    normed can be 'area' (produces matplotlib behavior, area is 1),
    any False value (no normalization), or any True value
(normalization).

    Original docs from matplotlib:

    Return the histogram of y with bins equally sized bins.  If bins
    is an array, use the bins.  Return value is
    (n,x) where n is the count for each bin in x

    If normed is False, return the counts in the first element of
the
```

```
    return tuple.  If normed is True, return the probability density
    n/(len(y)*dbin)

    If y has rank>1, it will be raveled
    Credits: the Numeric 22 documentation



    """
    y = asarray(y)
    if len(y.shape)>1: y = ravel(y)

    if not iterable(bins):
        ymin, ymax = min(y), max(y)
        if ymin==ymax:
            ymin -= 0.5
            ymax += 0.5

        if bins==1: bins=ymax
        dy = (ymax-ymin)/bins
        bins = ymin + dy*arange(bins)
    n = searchsorted(sort(y), bins)
    n = diff(concatenate([n, [len(y)]]))
    if normed:
        if normed == 'area':
            db = bins[1]-bins[0]
        else:
            db = 1.0
        return 1/(len(y)*db)*n, bins
    else:
        return n, bins

def as_species(name):
    """Cleans up a filename into a species name, italicizing it in
latex."""
    #trim extension if present
    dot_location = name.rfind('.')
    if dot_location > -1:
        name = name[:dot_location]
    #get rid of _small if present -- used for debugging
    if name.endswith('_small'):
        name = name[:-len('_small')]
    #replace underscores with spaces
    name = name.replace('_', ' ')
    #make sure the first letter of the genus is caps, and not the
first letter
    #of the species
    fields = name.split()
    fields[0] = fields[0].title()
    #assume second field is species name
    if len(fields) > 1:
        fields[1] = fields[1].lower()
    return '\emph{'+' '.join(fields)+'}'

def frac_to_psq(frac, graph_size):
    """Converts diameter as fraction of graph to points squared for
scatter.
```

```
    frac: fraction of graph (e.g. .01 is 1% of graph size)
    graph_size: graph size in inches
    """
    points = frac * graph_size * 72
    return pi * (points/2.0)**2


def init_graph_display(title=None, aux_title=None, size=4.0, \
    graph_shape='sqr', graph_grid=None, x_label='', y_label='', \
    dark=False, with_parens=True, prob_axes=True, axes=None,
num_genes=None):
    """Initializes a range of graph settings for standard plots.

    These settings include:
        - font sizes based on the size of the graph
        - graph shape
        - grid, including lines for x=y or at x and y = 0.5
        - title, auxillary title, and x and y axis labels

    Parameters:
        title: displayed on left of graph, at the top, latex-format
string

        aux_title: displayed on top right of graph, latex-format
string.
        typically used for number of genes.

        size:   size of graph, in inches

        graph_shape: 'sqr' for square graphs, 'rect' for graphs that
include
        a colorbar, 3to1: width 3 to height 1.

        graph_grid: background grid for the graph. Currently
recognized grids
        are '/' (line at x=y) and 't' (cross at x=.5 and y=.5).

        x_label: label for x axis, latex-format string.

        y_label: label for y axis, latex-format string.

        dark: set to True if dark background, reverses text and tick
colors.

        with_parens: if True (default), puts parens around auxillary
title

    returns font, label_font_size (for use in producing additional
labels in
    calling function).
    """
    if dark:
        color='w'
    else:
        color='k'
    min_offset = 0.05           #minimum offset, e.g. for text
    max_offset = 1-min_offset   #center offsets
    rect_scale_factor = 1.28    #need to allow for legend while
keeping graph
                                #square; empirically determined at
1.28
    font_size = int(size*3-1)   #want 11pt font w/ default graph
size 4" sqr
    label_scale_factor = 0.8
    label_font_size = font_size * label_scale_factor
    label_offset = label_font_size * 0.5
    axis_label_font={'fontsize':font_size}
    font={'fontsize':font_size, 'color':color}


    if graph_shape == 'sqr':
        gcf().set_figsize_inches(size,size)
    elif graph_shape == 'rect':
        #scaling for sqr graphs with colorbar
        gcf().set_figsize_inches(size*rect_scale_factor,size)
    elif graph_shape == '3to1':
        gcf().set_figsize_inches(3*size, size)
    elif graph_shape == '2to1':
        gcf().set_figsize_inches(2*size, size)
    else:
        raise ValueError, "Got unknown graph shape %s" % graph_shape

    #set or create axes
    if axes is None:
        axes = gca()

    #draw grid manually: these are in data coordinates.
    if graph_grid == 't':
        #grid lines at 0.5 on each axis, horiz & vertic
        axes.axvline(x=.5, ymin=0, ymax=1, color=color,
linestyle=':')
        axes.axhline(y=.5, xmin=0, xmax=1, color=color,
linestyle=':')
    elif graph_grid == '/':
        #diagonal gridlines from 0,0 to 1,1.
        axes.plot([0,1], color=color, linestyle=':')
    else:
        pass    #ignore other choices

    #remove default grid
    axes.grid(False)

    #set x and y labels
    axes.set_ylabel(y_label, axis_label_font)
    axes.set_xlabel(x_label, axis_label_font)

    #add title/aux_title to graph directly. Note that we want
    #the tops of these to be fixed, and we want the label to be
    #left-justified and the number of genes to be right justified,
    #so that it still works when we resize the graph.
    if title is not None:
        axes.text(min_offset, max_offset, str(title), font, \
            verticalalignment='top', horizontalalignment='left')
    #use num_genes as aux_title by default
    aux_title = num_genes or aux_title
    if aux_title is not None:
        if with_parens:
            aux_title='('+str(aux_title)+')'
        axes.text(max_offset, max_offset, str(aux_title), font,
             verticalalignment='top', horizontalalignment='right')
    if prob_axes:
        init_ticks(axes, label_font_size, dark)
    #set x and y label offsets -- currently though rcParams, but
should be
    #able to do at instance level?
    #rc('xtick.major', pad=label_offset)
    #rc('ytick.major', pad=label_offset)
    return font, label_font_size

def init_ticks(a, label_font_size, dark=False):
    """takes a from (a = gca)
    sets the ticks to span from 0 to 1 with .1 intervals
    changes the size of the ticks and the corresponding number
labels
    """
    a.set_xticks(arange(0,1.01,.1),)
    a.set_yticks(arange(0,1.01,.1))

    #reset sizes for x and y labels
    x = a.get_xticklabels()
    y = a.get_yticklabels()
    for l in a.get_xticklabels() + a.get_yticklabels():
        l.set_fontsize(label_font_size)
    #if dark, need to reset color of internal ticks to white
    if dark:
        for l in a.get_xticklines() + a.get_yticklines():
            l.set_markeredgecolor('white')

def set_axis_to_probs(axes=None):
    """sets the axes to span from 0 to 1
    necessary because order in program changes graph
    """
    #set axis for probabilities (range 0 to 1)
    if axes is None:
        axes = gca()
    axes.set_xlim([0,1])
    axes.set_ylim([0,1])

def plot_regression_line(x_data,y_data,line_color='r', axes=None):
    """Plots the regression line, and returns the equation."""
    if axes is None:
        axes = gca()
    m, b = regress(x_data, y_data)
    r, significance = correlation(x_data,y_data)
    #set the a,b,r values
    r_str = '%0.3g'% (r**2)
    m_str ='%0.3g' % m
    b_str = '%0.3g' % b
    x1=0.0
    y1=b
    x2=1.0
    y2=(m+b)

    #constrain so that y is always in the range (0,1) for plotting
    if(y1<0):
        y1=0.0
        x1=(0-b)/m
    if(y2>=1):
        y2=1.0
        x2=(1-b)/m

    axes.plot([x1,x2],[y1,y2], color=line_color, linewidth=2)

    if b >= 0:
        sign_str = ' + '
    else:
        sign_str = ' '

    equation=''.join(['y=
',m_str,'x',sign_str,b_str,'\n\nr$^2$=',r_str])
    return equation, line_color

def print_regression_equations(equations, axes=None):
    """Writes list of regression equations to graph.

    equations: list of regression equations

    size: size of the graph in inches
    """
    if axes is None:
        axes = gca()
    for i, (eq_text, eq_color) in enumerate(equations):
        axes.text((0.98), (0.02+(.06*i)), str(eq_text), \
            horizontalalignment='right', verticalalignment='bottom',
\
            color=eq_color)


def broadcast(i, n):
    """Broadcasts i to a vector of length n."""
    try:
        i = list(i)
    except:
        i = [i]
    reps, leftovers = divmod(n, len(i))
    return (i * reps) + i[:leftovers]


#scatterplot functions and helpers

def plot_scatter(data, series_names=None, \
    series_color=standard_series_colors,
line_color=standard_series_colors,\
    alpha=0.25, marker_size=.015, scale_markers=True,
    show_legend=True,legend_loc='center right',
    show_regression=True, show_equation=True,
    prob_axes=False, size=8.0, axes=None,
    **kwargs):
    """helper plots one or more series of scatter data of specified
color,
    calls the initializing functions, doesn't print graph

    takes: plotted_pairs, series_names, show_legend, legend_loc, and
        **kwargs passed on to init_graph_display (these include
title,
        aux_title, size, graph_shape, graph_grid, x_label, y_label,
        dark, with_parens).

    plotted_pairs = (first_pos, second_pos, dot_color, line_color,
    alpha, show_regression, show_equation)

    returns the regression str equation (list) if regression is set
true

    suppresses legend if series not named, even if show_legend is
True.
    """
    if not axes:
        axes = gca()
    #initialize fonts, shape and labels
    font,label_font_size=init_graph_display(prob_axes=prob_axes, \
        size=size, axes=axes, **kwargs)
    equations = []
    #figure out how many series there are, and scale vals
accordingly
    num_series = len(data)/2
    series_color = broadcast(series_color, num_series)
    line_color = broadcast(line_color, num_series)
    alpha = broadcast(alpha, num_series)
    marker_size = broadcast(marker_size, num_series)
    if scale_markers:
        marker_size = [frac_to_psq(m, size) for m in marker_size]

    series = []
    for i in range(num_series):
        x, y = data[2*i], data[2*i+1]

series.append(axes.scatter(x,y,s=marker_size[i],c=series_color[i],\
        alpha=alpha[i]))
        #find the equation and plots the regression line if True
        if show_regression:
            equation = plot_regression_line(x,y,line_color[i],
axes=axes)
        if show_equation:
            equations.append(equation)  #will be (str, color) tuple
    #print all the regression equations at once -- need to know how
many
    if show_regression:
        print_regression_equations(equations, axes=axes)
    #clean up axes if necessary
    if show_legend and series_names: #suppress legend if series not
named
        axes.legend(series, series_names, legend_loc)
    if prob_axes:
        set_axis_to_probs(axes)
    return equations, font

def plot_cai_p3_scatter(data, graph_name='cai_p3_scat.png',
**kwargs):
    """Outputs a CAI vs P3 scatter plot.

    expects data as ([P3s_1, CAIs_1, P3s_2, CAIs_2, ...])
    """
    plot_scatter(data, graph_shape='sqr', graph_grid=None,\
        x_label="$P_3$",y_label="CAI", prob_axes=True,**kwargs)
    savefig(graph_name)

def plot_p12_p3(data, graph_name='p12_p3.png', **kwargs):
    """Outputs a P12 versus P3 scatter graph, optionally including
regression.

    expects data as [P3_1, P12_1, P3_2, P12_2, ...n ].
    """
    plot_scatter(data, graph_shape='sqr', graph_grid='/',\
        x_label="$P_3$",y_label="$P_{12}$", prob_axes=True,
**kwargs)
    savefig(graph_name)

def plot_p123_gc(data, graph_name='p123_gc.png', use_p3_as_x=False,
**kwargs):
    """Output a scatter plot of p1,p2,p3 vs gc content

    Expects data as array with rows as GC, P1, P2, P3
    p1=blue, p2=green, p3=red

    """
    #unpack common x axis, and decide on series names
    if use_p3_as_x:
        series_names = ['$P_1$', '$P_2$']
        colors=['b','g']
        x_label='$P_3$'
        y_label='$P_{12}$'
        xy_pairs = [data[3], data[1], data[3], data[2]]
    else:
        series_names = ['$P_1$', '$P_2$', '$P_3$']
        colors=['b','g','r']
        x_label='GC'
        y_label='$P_{123}$'
        xy_pairs = [data[0], data[1], data[0], data[2], data[0],
data[3]]

    #plot points and write graph
    plot_scatter(xy_pairs,
graph_grid='/',x_label=x_label,y_label=y_label,
        series_names=series_names, prob_axes=True, **kwargs)
    savefig(graph_name)

def plot_fingerprint(data, alpha=0.7, \
    show_legend=True, graph_name='fingerprint.png', has_mean=True,
    which_blocks='quartets', multiple=False, graph_grid='t',
prob_axes=True, \
    **kwargs):
    """Outputs a bubble plot of four-codon amino acid blocks
    labeled with the colors from Sueoka 2002.

    takes: data:  array-elements in the col order x, y, r of
           each of the four codon Amino Acids in the row order:
           ALA, ARG4, GLY, LEU4, PRO, SER, THR, VAL
           (for traditional fingerprint), or:
           UU -> GG (for 16-block fingerprint).
           last row is the mean (if has_mean is set True)

        **kwargs passed on to init_graph_display (these include
        graph_shape, graph_grid, x_label, y_label, dark,
with_parens).

           title: will be printed on graph (default: 'Unknown
Species')

           num_genes (number of genes contributing to graph: default
None)
           NOTE: will not print if None.)

           size: of graph in inches (default = 8.0)

           alpha: transparency of bubbles
           (ranges from 0, transparent, to 1, opaque; default 0.7)

           show_legend: bool, default True, whether to print legend

           graph_name: name of file to write (default
'fingerprint.png')

           has_mean: whether the data contain the mean (default:
True)

           which_blocks: which codon blocks to print (default is
'quartets'
           for the 4-codon amino acid blocks, but can also use 'all'
for all
           quartets or 'split' for just the split quartets.)

           multiple: if False (the default), assumes it got a single
block
           of data. Otherwise, assumes multiple blocks of data in a
list or
           array.

    note: that the data are always expected to be in the range (0,1)
    since we're plotting frequencies. axes, gid, etc. are hard-coded
    to these values.
    """
    #figure out which type of fingerprint plot we're doing, and get
the
    #right colors
    if which_blocks == 'quartets':
        blocks = CodonUsage.SingleAABlocks
    elif which_blocks == 'split':
        blocks = CodonUsage.SplitBlocks
    else:
        blocks = CodonUsage.Blocks

    colors = [doublets_to_colors[i] for i in blocks]

    #formatting the labels in latex
    x_label="$G_3/(G_3+C_3)$"
    y_label="$A_3/(A_3+T_3)$"

    #initializing components of the graph
    font,label_font_size=init_graph_display(graph_shape='sqr', \
        graph_grid=graph_grid, x_label=x_label, \
        y_label=y_label, prob_axes=prob_axes, **kwargs)

    if not multiple:
        data = [data]

    alpha = broadcast(alpha, len(data))

    for al, d in zip(alpha, data):
        #skip this series if no data
        if not d:
            continue
        for i, color in enumerate(colors):
            j = i+1
            #note: doing these as slices because scatter_classic
needs the
            #extra level of nesting
            patches = scatter_classic(d[i:j,0], d[i:j,1],
                        s=(d[i:j,2]/2), c=color)
            #set alpha for the patches manually
            for p in patches:
                p.set_alpha(al)

        #plot mean as its own point -- can't do cross with scatter
        if has_mean:
            mean_index = len(blocks)    #next index after the blocks
            plot([d[mean_index,0]], [d[mean_index,1]],
                 '-k+',markersize=label_font_size, alpha=al)


    abbrev = CodonUsage.BlockAbbreviations

    a = gca()
    #if show_legend is True prints a legend in the right center area
    if show_legend:
        legend_key = [abbrev[b] for b in blocks]
        #copy legend font properties from the x axis tick labels
        legend_font_props = \
            a.xaxis.get_label().get_font_properties().copy()
        legend_font_scale_factor = 0.7
        curr_size = legend_font_props.get_size()

legend_font_props.set_size(curr_size*legend_font_scale_factor)
```

```
        l = figlegend(a.patches,
                  legend_key,
                  prop=legend_font_props,
                  loc='center right',pad=0.1,labelsep=0.0025,
                  handlelen=0.02,handletextsep=0.007, axespad=0.0)
        #fix transparency of patches
        for p in l.get_patches():
            p.set_alpha(1)

    #initialize the ticks
    set_axis_to_probs()
    init_ticks(a, label_font_size)
    a.set_xticks([0, 0.5, 1])
    a.set_yticks([0,0.5,1])

    #output the figure
    savefig(graph_name)

#Contour plots and related functions

def plot_filled_contour(plot_data, xy_data=None,
show_regression=False, \
    show_equation=False, fill_cmap=cm.hot, graph_shape='rect', \
    num_contour_lines=10, **kwargs):
    """helper plots one or more series of contour data
    calls the initializing functions, doesn't output figure

    takes: plot_data, xy_data, show_regression, show_equation,
fill_cmap,
    and **kwargs passed on to init_graph_display.

           plot_data = (x_bin, y_bin, data_matrix dot_colors)
    """
    if show_regression:
        equation = plot_regression_line(xy_data[:,0],xy_data[:,1])
        if show_equation:
            print_regression_equations([equation])
    #init graph display, rectangular due to needed colorbar space
    init_graph_display(graph_shape=graph_shape, **kwargs)
    #plots the contour data
    for x_bin,y_bin,data_matrix in plot_data:
        contourf(x_bin,y_bin,data_matrix, num_contour_lines,
cmap=fill_cmap)
    #add the colorbar legend to the side
    colorbar()

def plot_contour_lines(plot_data, xy_data=None,
show_regression=False, \
        show_equation=False, smooth_steps=0, num_contour_lines=10, \
        label_contours=False, line_cmap=cm.hot,
fill_cmap=cm.gray,dark=True,
        graph_shape='rect', **kwargs):
    """helper plots one or more series of contour line data
    calls the initializing functions, doesn't output figure

    takes: plot_data, xy_data, show_regression, show_equation,
smooth,
        num_contour_lines, label_contours, line_cmap, fill_cmap,
graph_shape,
        and **kwargs passed on to init_graph_display.

           plot_data = (x_bin, y_bin, data_matrix dot_colors)
    """
    #init graph display, rectangular due to needed colorbar space
    init_graph_display(graph_shape=graph_shape,
        dark=dark, **kwargs)
    #plots the contour data
    for x_bin,y_bin,data in plot_data:
        orig_max = max(ravel(data))
        scaled_data = (data/orig_max*255).astype('b')
        if smooth_steps:
            orig_shape = data.shape
            im = Image.fromstring('L', data.shape, scaled_data)
            for i in range(smooth_steps):
                im = im.filter(ImageFilter.BLUR)
            new_data = fromstring(im.tostring(), 'b')
            data = reshape(new_data.astype('i')/255.0 * orig_max,
orig_shape)

        if fill_cmap is not None:
            im = imshow(data, interpolation='bicubic',
extent=(0,1,0,1), \
                origin='lower', cmap=fill_cmap)
        result=contour(x_bin,y_bin,data, num_contour_lines,
                              origin='lower',linewidth=2,
                              extent=(0,1,0,1), cmap=line_cmap)
        if label_contours:
            clabel(result, fmt='%1.1g')

    #add the colorbar legend to the side
    cb = colorbar()
    cb.set_axis_bgcolor('black')

    if show_regression:
        equation=plot_regression_line(xy_data[0],xy_data[1])
        if show_equation:
            print_regression_equations([equation])

def plot_cai_p3_contour(x_bin,y_bin,data,xy_data,
                        graph_name='cai_contour.png',
                        prob_axes=True, **kwargs):
    """Output a contour plot of cai vs p3 with colorbar on side

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'cai_contour.png'
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_filled_contour(plot_data, graph_grid='/',x_label="$P_3$", \
        y_label="CAI", prob_axes=prob_axes, **kwargs)
    set_axis_to_probs()
    savefig(graph_name)

def plot_cai_p3_contourlines(x_bin,y_bin,data,xy_data,
                             graph_name='cai_contourlines.png',
                             prob_axes=True, **kwargs):
    """Output a contour plot of cai

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'cai_contourlines.png'
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_contour_lines(plot_data, graph_grid='/', x_label="$P_3$", \
        y_label="CAI", prob_axes=prob_axes,**kwargs)
    savefig(graph_name)

def plot_p12_p3_contour(x_bin,y_bin,data,xy_data,
                        graph_name='p12_p3_contour.png',
                        prob_axes=True, **kwargs):
    """Outputs a P12 versus P3 contour graph
    and the mean equation of the plot

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'p12_p3_contourlines.png'
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_filled_contour(plot_data, graph_grid='/', x_label="$P_3$",
\
        y_label="$P_{12}$", prob_axes=prob_axes,**kwargs)
    set_axis_to_probs()
    savefig(graph_name)

def plot_p12_p3_contourlines(x_bin,y_bin,data,xy_data,
prob_axes=True,\
    graph_name='p12_p3_contourlines.png', **kwargs):
    """Outputs a P12 versus P3 contourline graph
    and the mean equation of the plot

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')
```

```
            num_genes (default 0 will not print, other numbers will)

            size: of graph in inches (default = 8.0)

            graph_name: default 'p12_p3_contourlines.png
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_contour_lines(plot_data, graph_grid='/', x_label="$P_3$",\
        y_label="$P_{12}$", prob_axes=prob_axes, **kwargs)
    set_axis_to_probs()
    savefig(graph_name)

#Other graphs

def plot_pr2_bias(data, title='ALANINE', graph_name='pr2_bias.png',
\
    num_genes='ignored', **kwargs):
    """Outputs a PR2-Bias plot of:
    -isotypic transversions (base swapping)
    with G3/(G3+C3) and A3/(A3+T3)
    -Transitions (deaminations)
    with G3/(G3+A3) and C3/(C3+T3)
    -Allotypic transversions (G- oxidations)
    with G3/(G3+T3) and C3/(C3+A3)

    takes: an array in the order: x,G3/(G3+C3),A3/(A3+T3),
    G3/(G3/A3),C3/(C3+T3),G3/(G3+T3),C3/(C3+A3)

    label: default 'ALANINE'
    one amino acid written out in caps:
    ALANINE, ARGININE4, GLYCINE, LEUCINE4,
    PROLINE, SERINE4, THREONINE, VALINE
       from one of the amino acids program will add acronym
       C2 type: ala(GCN), pro(CCN), ser4(TCN), thr(ACN)
       G2 type: arg4 (CGN), an gly(GGN)
       T2 type: leu4(CTN), val (GTN)

    size: of graph in inches (default = 8.0)

    graph_name: default 'pr2_bias.png'

    num_genes: number of genes contributing to graph, currently
ignored.
    """
    #we can't put anything in the top right, so print num_genes
after the title
    #if it was supplied
    #initializes the graph display and font
    font,label_font_size=init_graph_display(graph_shape='sqr', \
        graph_grid='/', x_label="$P_3$", y_label="Y axis",
prob_axes=True, \
        title=title, **kwargs)
    #sets the marker_size relative to the font and thus the graph
size
    marker_size = (label_font_size-1)
    #plots the pr2bias in order G3/(G3+C3),A3/(A3+T3),
    #                           G3/(G3/A3),C3/(C3+T3),
    #                           G3/(G3+T3),C3/(C3+A3)
    #colors and symbols coded from Sueoka 2002
    plot([data[:,0]], [data[:,1]], '-ko', c='k',
         markersize=marker_size)
    plot([data[:,0]], [data[:,2]], '-kv', c='k',
         markersize=marker_size)
    plot([data[:,0]], [data[:,3]], '-ro', c='r',
         markersize=marker_size)
    plot([data[:,0]], [data[:,4]], '-rv', c='r',
         markersize=marker_size)
    plot([data[:,0]], [data[:,5]], '-wo', c='k',
         markersize=marker_size)
    plot([data[:,0]], [data[:,6]], '-wv', c='k',
         markersize=marker_size)

    set_axis_to_probs()

    #aaLabel based on the amino acid that is graphed
    #C2 type: ala(GCN), pro(CCN), ser4(TCN), thr(ACN)
    #G2 type: arg4 (CGN), an gly(GGN)
    #T2 type: leu4(CTN), val (GTN) (Sueoka 2002)
    text(.95, .05, aaLabels[title], font,
verticalalignment='bottom',
         horizontalalignment='right')

    #output the figure
    set_axis_to_probs()
    savefig(graph_name)


def plot_histograms(data, graph_name='gene_histogram.png', bins=20,\
        normal_fit=True, normed=True, colors=None, linecolors=None,
\
        alpha=0.75, prob_axes=True, series_names=None,
show_legend=False,\
        y_label=None, **kwargs):
    """Outputs a histogram with multiple series (must provide a list
of series).

    takes:  data: list of arrays of values to plot (needs to be list
of arrays
            so you can pass in arrays with different numbers of
elements)

            graph_name: filename to write graph to
            bins: number of bins to use
            normal_fit: whether to show the normal curve best
fitting the data
            normed: whether to normalize the histogram (e.g. so bars
sum to 1)
            colors: list of colors to use for bars
            linecolors: list of colors to use for fit lines

            **kwargs are pssed on to init_graph_display.
    """
    if y_label is None:
        if normed:
            y_label='Frequency'
        else:
            y_label='Count'
    num_series = len(data)
    if colors is None:
        if num_series == 1:
            colors = ['white']
        else:
            colors = standard_series_colors
    if linecolors is None:
        if num_series == 1:
            linecolors = ['red']
        else:
            linecolors = standard_series_colors

    init_graph_display(prob_axes=prob_axes, y_label=y_label,
**kwargs)
    for i, d in enumerate(data):
        fc = colors[i % len(colors)]
        lc = linecolors[i % len(linecolors)]

        counts, x_bins, patches = hist(d, bins=bins, normed=normed,
\
            alpha=alpha, facecolor=fc)

        if normal_fit:
            mu = mean(d)
            sigma = std(d)
            bin_width = x_bins[-1] - x_bins[-2]
            #want normpdf to extend over the bins, so needs to be
one extra
            #normpdf_bins = x_bins + bin_width/2.0
            normpdf_bins = arange(0,1,0.01)
            y = normpdf(normpdf_bins, mu, sigma)
            orig_area = sum(counts) * bin_width
            y = y * orig_area   #normpdf area is 1 by default
            plot(normpdf_bins, y, linestyle='--', color=lc,
linewidth=1)

    if show_legend and series_names:
        legend(series_names)

    #output figure
    savefig(graph_name)

def plot_scatter_with_histograms(data,
graph_name='histo_scatter.png', \
    graph_grid='/', prob_axes=False, bins=20, frac=0.9,
scatter_alpha=0.5, \
    hist_alpha=0.8, colors=standard_series_colors, normed=True,
**kwargs):
    """Plots a scatter plot with histograms showing distribution of
x and y.
```

```
    Data should be list of [x1, y1, x2, y2, ...].
    """

    #set up subplot coords
    tl=subplot(2,2,1)
    br=subplot(2,2,4)
    bl=subplot(2,2,3, sharex=tl, sharey=br)

    #get_position returns left, bottom, width, height relative to
figure
    tl_coords = tl.get_position()
    bl_coords = bl.get_position()
    br_coords = br.get_position()

    left = tl_coords[0]
    bottom = bl_coords[1]

    width = br_coords[0] + br_coords[2] - left
    height = tl_coords[1] + tl_coords[3] - bottom

    bl.set_position([left, bottom, frac*width, frac*height])
    tl.set_position([left, bottom+(frac*height), frac*width, (1-
frac)*height])
    br.set_position([left+(frac*width), bottom, (1-frac)*width,
frac*height])

    #suppress frame and axis for histograms
    for i in [tl,br]:
        i.set_frame_on(False)
        i.xaxis.set_visible(False)
        i.yaxis.set_visible(False)

    plot_scatter(data=data, alpha=scatter_alpha, axes=bl, **kwargs)

    biggest_x = 0
    biggest_y = 0
    for i in range(0, len(data), 2):
        x, y = data[i], data[i+1]
        color = colors[(i/2)%len(colors)]
        n, bins, patches = hist(x, facecolor=color, bins=bins, \
            alpha=hist_alpha, axes=tl, normed=True)
        biggest_x = max([biggest_x, max(x)])
        n, bins, patches = hist(y, facecolor=color, bins=bins, \
            alpha=hist_alpha, axes=br, normed=normed,
orientation='horizontal')
        biggest_y=max([biggest_y,max(y)])
    bl.set_xlim(0,1)
    bl.set_ylim(0,1)
    savefig(graph_name)
```

2. Code implementing the Smith-Waterman, and Uliel algorithms for island detection

```
#/usr/lib/python2.4/
#AminoGraphPlots.py

from matplotlib import use, rc
use('Agg')  #suppress graphical rendering
rc('text', usetex=True)
rc('font', family='serif')  #required to match latex text and
equations
import Image
import ImageFilter
from Numeric import array, shape, fromstring
from cogent.base.usage import UnsafeCodonUsage as CodonUsage
from cogent.maths.stats.test import regress, correlation
from pylab import *
from math import pi

"""Provides different kinds of codon usage plots.

See individual docstrings for more info.
"""
#module-level constants

#historical doublet order for fingerprint plot; not currently used,
but
#same order that the colors were entered in. Matches Sueoka 2002.
doublet_order =
['GC','CG','GG','CU','CC','UC','AC','GU','UU','CA','AU',\
                 'AA','AG','GA','UA','UG']
color_order = ["#000000","#FF0000","#00FF00","#FFFF00",
          "#CC99FF","#FFCC99","#CCFFFF","#C0C0C0",
          "#6D6D6D","#2353FF","#00FFFF","#FF8800",
          "#238853","#882353","#EC008C","#000099"]
#map doublets to colors so we can make sure the same doublet always
#gets the same colors
doublets_to_colors = dict(zip(doublet_order, color_order))
#creates a dictionary for the amino acid labels, less to input
aaLabels={'ALANINE':'GCN', 'ARGININE4':'CGN', 'GLYCINE':'GGN',
          'LEUCINE4':'CTN', 'PROLINE':'CCN', 'SERINE4':'TCN',
          'THREONINE':'ACN', 'VALINE':'GTN'}
standard_series_colors=['k','r','g','b', 'm','c']

#Helper functions

def hist(x, bins=10, normed='height', bottom=0, \
    orientation='vertical', width=None, axes=None, **kwargs):
    """Just like the matplotlib hist, but normalizes bar heights to
1.

    axes uses gca() by default (built-in hist is a method of Axes).

    Original docs from matplotlib:
```

```
    HIST(x, bins=10, normed=0, bottom=0, orientiation='vertical',
**kwargs)

    Compute the histogram of x.  bins is either an integer number of
    bins or a sequence giving the bins.  x are the data to be
binned.

    The return values is (n, bins, patches)

    If normed is true, the first element of the return tuple will
    be the counts normalized to form a probability density, ie,
    n/(len(x)*dbin)


    orientation = 'horizontal' | 'vertical'.  If horizontal, barh
    will be used and the "bottom" kwarg will be the left.

    width: the width of the bars.  If None, automatically compute
    the width.

    kwargs are used to update the properties of the
    hist bars
    """
    if axes is None:
        axes = gca()
    if not axes._hold: axes.cla()
    n,bins = norm_hist_bins(x, bins, normed)
    if width is None: width = 0.9*(bins[1]-bins[0])
    if orientation=='horizontal':
        patches = axes.barh(n, bins, height=width, left=bottom)
    else:
        patches = axes.bar(bins, n, width=width, bottom=bottom)
    for p in patches:
        p.update(kwargs)
    return n, bins, silent_list('Patch', patches)

def norm_hist_bins(y, bins=10, normed='height'):
    """Just like the matplotlib mlab.hist, but can normalize by
height.

    normed can be 'area' (produces matplotlib behavior, area is 1),
    any False value (no normalization), or any True value
(normalization).

    Original docs from matplotlib:

    Return the histogram of y with bins equally sized bins.  If bins
    is an array, use the bins.  Return value is
    (n,x) where n is the count for each bin in x

    If normed is False, return the counts in the first element of
the
    return tuple.  If normed is True, return the probability density
    n/(len(y)*dbin)

    If y has rank>1, it will be raveled
    Credits: the Numeric 22 documentation
    """
    y = asarray(y)
    if len(y.shape)>1: y = ravel(y)

    if not iterable(bins):
        ymin, ymax = min(y), max(y)
        if ymin==ymax:
            ymin -= 0.5
            ymax += 0.5

        if bins==1: bins=ymax
        dy = (ymax-ymin)/bins
        bins = ymin + dy*arange(bins)
    n = searchsorted(sort(y), bins)
    n = diff(concatenate([n, [len(y)]]))
    if normed:
        if normed == 'area':
            db = bins[1]-bins[0]
        else:
            db = 1.0
        return 1/(len(y)*db)*n, bins
    else:
        return n, bins

def as_species(name):
    """Cleans up a filename into a species name, italicizing it in
latex."""
    #trim extension if present
    dot_location = name.rfind('.')
    if dot_location > -1:
        name = name[:dot_location]
    #get rid of _small if present -- used for debugging
    if name.endswith('_small'):
        name = name[:-len('_small')]
    #replace underscores with spaces
    name = name.replace('_', ' ')
    #make sure the first letter of the genus is caps, and not the
first letter
    #of the species
    fields = name.split()
    fields[0] = fields[0].title()
    #assume second field is species name
    if len(fields) > 1:
        fields[1] = fields[1].lower()
    return '\emph{'+' '.join(fields)+'}'

def frac_to_psq(frac, graph_size):
    """Converts diameter as fraction of graph to points squared for
scatter.

    frac: fraction of graph (e.g. .01 is 1% of graph size)
    graph_size: graph size in inches
    """
    points = frac * graph_size * 72
    return pi * (points/2.0)**2


def init_graph_display(title=None, aux_title=None, size=4.0, \
    graph_shape='sqr', graph_grid=None, x_label='', y_label='', \
    dark=False, with_parens=True, prob_axes=True, axes=None,
num_genes=None):
    """Initializes a range of graph settings for standard plots.

    These settings include:
        - font sizes based on the size of the graph
        - graph shape
        - grid, including lines for x=y or at x and y = 0.5
        - title, auxillary title, and x and y axis labels

    Parameters:
        title: displayed on left of graph, at the top, latex-format
string

        aux_title: displayed on top right of graph, latex-format
string.
        typically used for number of genes.

        size:   size of graph, in inches

        graph_shape: 'sqr' for square graphs, 'rect' for graphs that
include
        a colorbar, 3to1: width 3 to height 1.

        graph_grid: background grid for the graph. Currently
recognized grids
        are '/' (line at x=y) and 't' (cross at x=.5 and y=.5).

        x_label: label for x axis, latex-format string.

        y_label: label for y axis, latex-format string.

        dark: set to True if dark background, reverses text and tick
colors.

        with_parens: if True (default), puts parens around auxillary
title

    returns font, label_font_size (for use in producing additional
labels in
    calling function).
    """
    if dark:
        color='w'
    else:
        color='k'
    min_offset = 0.05           #minimum offset, e.g. for text
    max_offset = 1-min_offset   #center offsets
    rect_scale_factor = 1.28    #need to allow for legend while
keeping graph
                                #square; empirically determined at
1.28
```

```
    font_size = int(size*3-1)   #want 11pt font w/ default graph
size 4" sqr
    label_scale_factor = 0.8
    label_font_size = font_size * label_scale_factor
    label_offset = label_font_size * 0.5
    axis_label_font={'fontsize':font_size}
    font={'fontsize':font_size, 'color':color}


    if graph_shape == 'sqr':
        gcf().set_figsize_inches(size,size)
    elif graph_shape == 'rect':
        #scaling for sqr graphs with colorbar
        gcf().set_figsize_inches(size*rect_scale_factor,size)
    elif graph_shape == '3to1':
        gcf().set_figsize_inches(3*size, size)
    elif graph_shape == '2to1':
        gcf().set_figsize_inches(2*size, size)
    else:
        raise ValueError, "Got unknown graph shape %s" % graph_shape

    #set or create axes
    if axes is None:
        axes = gca()

    #draw grid manually: these are in data coordinates.
    if graph_grid == 't':
        #grid lines at 0.5 on each axis, horiz & vertic
        axes.axvline(x=.5, ymin=0, ymax=1, color=color,
linestyle=':')
        axes.axhline(y=.5, xmin=0, xmax=1, color=color,
linestyle=':')
    elif graph_grid == '/':
        #diagonal gridlines from 0,0 to 1,1.
        axes.plot([0,1], color=color, linestyle=':')
    else:
        pass    #ignore other choices

    #remove default grid
    axes.grid(False)

    #set x and y labels
    axes.set_ylabel(y_label, axis_label_font)
    axes.set_xlabel(x_label, axis_label_font)

    #add title/aux_title to graph directly. Note that we want
    #the tops of these to be fixed, and we want the label to be
    #left-justified and the number of genes to be right justified,
    #so that it still works when we resize the graph.
    if title is not None:
        axes.text(min_offset, max_offset, str(title), font, \
            verticalalignment='top', horizontalalignment='left')
    #use num_genes as aux_title by default
    aux_title = num_genes or aux_title
    if aux_title is not None:
        if with_parens:
            aux_title='('+str(aux_title)+')'
        axes.text(max_offset, max_offset, str(aux_title), font,
             verticalalignment='top', horizontalalignment='right')
    if prob_axes:
        init_ticks(axes, label_font_size, dark)
    #set x and y label offsets -- currently though rcParams, but
should be
    #able to do at instance level?
    #rc('xtick.major', pad=label_offset)
    #rc('ytick.major', pad=label_offset)
    return font, label_font_size

def init_ticks(a, label_font_size, dark=False):
    """takes a from (a = gca)
    sets the ticks to span from 0 to 1 with .1 intervals
    changes the size of the ticks and the corresponding number
labels
    """
    a.set_xticks(arange(0,1.01,.1),)
    a.set_yticks(arange(0,1.01,.1))

    #reset sizes for x and y labels
    x = a.get_xticklabels()
    y = a.get_yticklabels()
    for l in a.get_xticklabels() + a.get_yticklabels():
        l.set_fontsize(label_font_size)
    #if dark, need to reset color of internal ticks to white
    if dark:
        for l in a.get_xticklines() + a.get_yticklines():
            l.set_markeredgecolor('white')

def set_axis_to_probs(axes=None):
    """sets the axes to span from 0 to 1
    necessary because order in program changes graph
    """
    #set axis for probabilities (range 0 to 1)
    if axes is None:
        axes = gca()
    axes.set_xlim([0,1])
    axes.set_ylim([0,1])

def plot_regression_line(x_data,y_data,line_color='r', axes=None):
    """Plots the regression line, and returns the equation."""
    if axes is None:
        axes = gca()
    m, b = regress(x_data, y_data)
    r, significance = correlation(x_data,y_data)
    #set the a,b,r values
    r_str = '%0.3g'% (r**2)
    m_str ='%0.3g' % m
    b_str = '%0.3g' % b
    x1=0.0
    y1=b
    x2=1.0
    y2=(m+b)

    #constrain so that y is always in the range (0,1) for plotting
    if(y1<0):
        y1=0.0
        x1=(0-b)/m
    if(y2>=1):
        y2=1.0
        x2=(1-b)/m

    axes.plot([x1,x2],[y1,y2], color=line_color, linewidth=2)

    if b >= 0:
        sign_str = ' + '
    else:
        sign_str = ' '

    equation=''.join(['y=
',m_str,'x',sign_str,b_str,'\n\nr$^2$=',r_str])
    return equation, line_color

def print_regression_equations(equations, axes=None):
    """Writes list of regression equations to graph.

    equations: list of regression equations

    size: size of the graph in inches
    """
    if axes is None:
        axes = gca()
    for i, (eq_text, eq_color) in enumerate(equations):
        axes.text((0.98), (0.02+(.06*i)), str(eq_text), \
            horizontalalignment='right', verticalalignment='bottom',
\
            color=eq_color)


def broadcast(i, n):
    """Broadcasts i to a vector of length n."""
    try:
        i = list(i)
    except:
        i = [i]
    reps, leftovers = divmod(n, len(i))
    return (i * reps) + i[:leftovers]


#scatterplot functions and helpers

def plot_scatter(data, series_names=None, \
    series_color=standard_series_colors,
line_color=standard_series_colors,\
    alpha=0.25, marker_size=.015, scale_markers=True,
    show_legend=True,legend_loc='center right',
    show_regression=True, show_equation=True,
    prob_axes=False, size=8.0, axes=None,
    **kwargs):
    """helper plots one or more series of scatter data of specified
color,
    calls the initializing functions, doesn't print graph
    takes: plotted_pairs, series_names, show_legend, legend_loc, and
        **kwargs passed on to init_graph_display (these include
title,
        aux_title, size, graph_shape, graph_grid, x_label, y_label,
        dark, with_parens).

    plotted_pairs = (first_pos, second_pos, dot_color, line_color,
    alpha, show_regression, show_equation)

    returns the regression str equation (list) if regression is set
true

    suppresses legend if series not named, even if show_legend is
True.
    """
    if not axes:
        axes = gca()
    #initialize fonts, shape and labels
    font,label_font_size=init_graph_display(prob_axes=prob_axes, \
        size=size, axes=axes, **kwargs)
    equations = []
    #figure out how many series there are, and scale vals
accordingly
    num_series = len(data)/2
    series_color = broadcast(series_color, num_series)
    line_color = broadcast(line_color, num_series)
    alpha = broadcast(alpha, num_series)
    marker_size = broadcast(marker_size, num_series)
    if scale_markers:
        marker_size = [frac_to_psq(m, size) for m in marker_size]

    series = []
    for i in range(num_series):
        x, y = data[2*i], data[2*i+1]

series.append(axes.scatter(x,y,s=marker_size[i],c=series_color[i],\
        alpha=alpha[i]))
        #find the equation and plots the regression line if True
        if show_regression:
            equation = plot_regression_line(x,y,line_color[i],
axes=axes)
        if show_equation:
            equations.append(equation)  #will be (str, color) tuple
    #print all the regression equations at once -- need to know how
many
    if show_regression:
        print_regression_equations(equations, axes=axes)
    #clean up axes if necessary
    if show_legend and series_names: #suppress legend if series not
named
        axes.legend(series, series_names, legend_loc)

    if prob_axes:
        set_axis_to_probs(axes)
    return equations, font
```

```
def plot_cai_p3_scatter(data, graph_name='cai_p3_scat.png',
**kwargs):
    """Outputs a CAI vs P3 scatter plot.

    expects data as ([P3s_1, CAIs_1, P3s_2, CAIs_2, ...])
    """
    plot_scatter(data, graph_shape='sqr', graph_grid=None,\
        x_label="$P_3$",y_label="CAI", prob_axes=True,**kwargs)
    savefig(graph_name)

def plot_p12_p3(data, graph_name='p12_p3.png', **kwargs):
    """Outputs a P12 versus P3 scatter graph, optionally including
regression.

    expects data as [P3_1, P12_1, P3_2, P12_2, ...n ].
    """
    plot_scatter(data, graph_shape='sqr', graph_grid='/',\
        x_label="$P_3$",y_label="$P_{12}$", prob_axes=True,
**kwargs)
    savefig(graph_name)

def plot_p123_gc(data, graph_name='p123_gc.png', use_p3_as_x=False,
**kwargs):
    """Output a scatter plot of p1,p2,p3 vs gc content

    Expects data as array with rows as GC, P1, P2, P3
    p1=blue, p2=green, p3=red

    """
    #unpack common x axis, and decide on series names
    if use_p3_as_x:
        series_names = ['$P_1$', '$P_2$']
        colors=['b','g']
        x_label='$P_3$'
        y_label='$P_{12}$'
        xy_pairs = [data[3], data[1], data[3], data[2]]
    else:
        series_names = ['$P_1$', '$P_2$', '$P_3$']
        colors=['b','g','r']
        x_label='GC'
        y_label='$P_{123}$'
        xy_pairs = [data[0], data[1], data[0], data[2], data[0],
data[3]]

    #plot points and write graph
    plot_scatter(xy_pairs,
graph_grid='/',x_label=x_label,y_label=y_label,
        series_names=series_names, prob_axes=True, **kwargs)
    savefig(graph_name)

def plot_fingerprint(data, alpha=0.7, \
    show_legend=True, graph_name='fingerprint.png', has_mean=True,
    which_blocks='quartets', multiple=False, graph_grid='t',
prob_axes=True, \
    **kwargs):
    """Outputs a bubble plot of four-codon amino acid blocks
    labeled with the colors from Sueoka 2002.
    takes: data:  array-elements in the col order x, y, r of
           each of the four codon Amino Acids in the row order:
           ALA, ARG4, GLY, LEU4, PRO, SER, THR, VAL
           (for traditional fingerprint), or:
           UU -> GG (for 16-block fingerprint).
           last row is the mean (if has_mean is set True)

        **kwargs passed on to init_graph_display (these include
        graph_shape, graph_grid, x_label, y_label, dark,
with_parens).

           title: will be printed on graph (default: 'Unknown
Species')

           num_genes (number of genes contributing to graph: default
None)
           NOTE: will not print if None.)

           size: of graph in inches (default = 8.0)

           alpha: transparency of bubbles
           (ranges from 0, transparent, to 1, opaque; default 0.7)

           show_legend: bool, default True, whether to print legend

           graph_name: name of file to write (default
'fingerprint.png')

           has_mean: whether the data contain the mean (default:
True)

           which_blocks: which codon blocks to print (default is
'quartets'
           for the 4-codon amino acid blocks, but can also use 'all'
for all
           quartets or 'split' for just the split quartets.)

           multiple: if False (the default), assumes it got a single
block
           of data. Otherwise, assumes multiple blocks of data in a
list or
           array.

    note: that the data are always expected to be in the range (0,1)
    since we're plotting frequencies. axes, gid, etc. are hard-coded
    to these values.
    """
    #figure out which type of fingerprint plot we're doing, and get
the
    #right colors
    if which_blocks == 'quartets':
        blocks = CodonUsage.SingleAABlocks
    elif which_blocks == 'split':
        blocks = CodonUsage.SplitBlocks
    else:
        blocks = CodonUsage.Blocks
    colors = [doublets_to_colors[i] for i in blocks]

    #formatting the labels in latex
    x_label="$G_3/(G_3+C_3)$"
    y_label="$A_3/(A_3+T_3)$"

    #initializing components of the graph
    font,label_font_size=init_graph_display(graph_shape='sqr', \
        graph_grid=graph_grid, x_label=x_label, \
        y_label=y_label, prob_axes=prob_axes, **kwargs)

    if not multiple:
        data = [data]

    alpha = broadcast(alpha, len(data))

    for al, d in zip(alpha, data):
        #skip this series if no data
        if not d:
            continue
        for i, color in enumerate(colors):
            j = i+1
            #note: doing these as slices because scatter_classic
needs the
            #extra level of nesting
            patches = scatter_classic(d[i:j,0], d[i:j,1],
                        s=(d[i:j,2]/2), c=color)
            #set alpha for the patches manually
            for p in patches:
                p.set_alpha(al)

        #plot mean as its own point -- can't do cross with scatter
        if has_mean:
            mean_index = len(blocks)    #next index after the blocks
            plot([d[mean_index,0]], [d[mean_index,1]],
                 '-k+',markersize=label_font_size, alpha=al)


    abbrev = CodonUsage.BlockAbbreviations

    a = gca()
    #if show_legend is True prints a legend in the right center area
    if show_legend:
        legend_key = [abbrev[b] for b in blocks]
        #copy legend font properties from the x axis tick labels
        legend_font_props = \
            a.xaxis.get_label().get_font_properties().copy()
        legend_font_scale_factor = 0.7
        curr_size = legend_font_props.get_size()

legend_font_props.set_size(curr_size*legend_font_scale_factor)
        l = figlegend(a.patches,
                  legend_key,
                  prop=legend_font_props,
                  loc='center right',pad=0.1,labelsep=0.0025,
                  handlelen=0.02,handletextsep=0.007, axespad=0.0)
        #fix transparency of patches
        for p in l.get_patches():
            p.set_alpha(1)

    #initialize the ticks
    set_axis_to_probs()
    init_ticks(a, label_font_size)
    a.set_xticks([0, 0.5, 1])
    a.set_yticks([0,0.5,1])

    #output the figure
    savefig(graph_name)

#Contour plots and related functions

def plot_filled_contour(plot_data, xy_data=None,
show_regression=False, \
    show_equation=False, fill_cmap=cm.hot, graph_shape='rect', \
    num_contour_lines=10, **kwargs):
    """helper plots one or more series of contour data
    calls the initializing functions, doesn't output figure

    takes: plot_data, xy_data, show_regression, show_equation,
fill_cmap,
    and **kwargs passed on to init_graph_display.

           plot_data = (x_bin, y_bin, data_matrix dot_colors)
    """
    if show_regression:
        equation = plot_regression_line(xy_data[:,0],xy_data[:,1])
        if show_equation:
            print_regression_equations([equation])
    #init graph display, rectangular due to needed colorbar space
    init_graph_display(graph_shape=graph_shape, **kwargs)
    #plots the contour data
    for x_bin,y_bin,data_matrix in plot_data:
        contourf(x_bin,y_bin,data_matrix, num_contour_lines,
cmap=fill_cmap)
    #add the colorbar legend to the side
    colorbar()

def plot_contour_lines(plot_data, xy_data=None,
show_regression=False, \
        show_equation=False, smooth_steps=0, num_contour_lines=10, \
        label_contours=False, line_cmap=cm.hot,
fill_cmap=cm.gray,dark=True,
        graph_shape='rect', **kwargs):
    """helper plots one or more series of contour line data
    calls the initializing functions, doesn't output figure

    takes: plot_data, xy_data, show_regression, show_equation,
smooth,
        num_contour_lines, label_contours, line_cmap, fill_cmap,
graph_shape,
        and **kwargs passed on to init_graph_display.

           plot_data = (x_bin, y_bin, data_matrix dot_colors)
    """
    #init graph display, rectangular due to needed colorbar space
    init_graph_display(graph_shape=graph_shape,
        dark=dark, **kwargs)
    #plots the contour data
    for x_bin,y_bin,data in plot_data:
        orig_max = max(ravel(data))
        scaled_data = (data/orig_max*255).astype('b')
        if smooth_steps:
            orig_shape = data.shape
            im = Image.fromstring('L', data.shape, scaled_data)
            for i in range(smooth_steps):
                im = im.filter(ImageFilter.BLUR)
            new_data = fromstring(im.tostring(), 'b')
            data = reshape(new_data.astype('i')/255.0 * orig_max,
orig_shape)

        if fill_cmap is not None:
            im = imshow(data, interpolation='bicubic',
extent=(0,1,0,1), \
                origin='lower', cmap=fill_cmap)
        result=contour(x_bin,y_bin,data, num_contour_lines,
                              origin='lower',linewidth=2,
                              extent=(0,1,0,1), cmap=line_cmap)
        if label_contours:
            clabel(result, fmt='%1.1g')

    #add the colorbar legend to the side
    cb = colorbar()
    cb.set_axis_bgcolor('black')

    if show_regression:
        equation=plot_regression_line(xy_data[0],xy_data[1])
        if show_equation:
            print_regression_equations([equation])

def plot_cai_p3_contour(x_bin,y_bin,data,xy_data,
                        graph_name='cai_contour.png',
                        prob_axes=True, **kwargs):
    """Output a contour plot of cai vs p3 with colorbar on side

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'cai_contour.png'
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_filled_contour(plot_data, graph_grid='/',x_label="$P_3$", \
        y_label="CAI", prob_axes=prob_axes, **kwargs)
    set_axis_to_probs()
    savefig(graph_name)
```

```
def plot_cai_p3_contourlines(x_bin,y_bin,data,xy_data,
                             graph_name='cai_contourlines.png',
                             prob_axes=True, **kwargs):
    """Output a contour plot of cai

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'cai_contourlines.png'
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_contour_lines(plot_data, graph_grid='/', x_label="$P_3$", \
        y_label="CAI", prob_axes=prob_axes,**kwargs)
    savefig(graph_name)

def plot_p12_p3_contour(x_bin,y_bin,data,xy_data,
                        graph_name='p12_p3_contour.png',
                        prob_axes=True, **kwargs):
    """Outputs a P12 versus P3 contour graph
    and the mean equation of the plot

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'p12_p3_contourlines.png'
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_filled_contour(plot_data, graph_grid='/', x_label="$P_3$",
\
        y_label="$P_{12}$", prob_axes=prob_axes,**kwargs)
    set_axis_to_probs()
    savefig(graph_name)

def plot_p12_p3_contourlines(x_bin,y_bin,data,xy_data,
prob_axes=True,\
    graph_name='p12_p3_contourlines.png', **kwargs):
    """Outputs a P12 versus P3 contourline graph
    and the mean equation of the plot

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)
```

```
            graph_name: default 'p12_p3_contourlines.png
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_contour_lines(plot_data, graph_grid='/', x_label="$P_3$",\
        y_label="$P_{12}$", prob_axes=prob_axes, **kwargs)
    set_axis_to_probs()
    savefig(graph_name)

#Other graphs

def plot_pr2_bias(data, title='ALANINE', graph_name='pr2_bias.png',
\
    num_genes='ignored', **kwargs):
    """Outputs a PR2-Bias plot of:
    -isotypic transversions (base swapping)
    with G3/(G3+C3) and A3/(A3+T3)
    -Transitions (deaminations)
    with G3/(G3+A3) and C3/(C3+T3)
    -Allotypic transversions (G- oxidations)
    with G3/(G3+T3) and C3/(C3+A3)

    takes: an array in the order: x,G3/(G3+C3),A3/(A3+T3),
    G3/(G3/A3),C3/(C3+T3),G3/(G3+T3),C3/(C3+A3)

    label: default 'ALANINE'
    one amino acid written out in caps:
    ALANINE, ARGININE4, GLYCINE, LEUCINE4,
    PROLINE, SERINE4, THREONINE, VALINE
       from one of the amino acids program will add acronym
       C2 type: ala(GCN), pro(CCN), ser4(TCN), thr(ACN)
       G2 type: arg4 (CGN), an gly(GGN)
       T2 type: leu4(CTN), val (GTN)

    size: of graph in inches (default = 8.0)

    graph_name: default 'pr2_bias.png'

    num_genes: number of genes contributing to graph, currently
ignored.
    """
    #we can't put anything in the top right, so print num_genes
after the title
    #if it was supplied
    #initializes the graph display and font
    font,label_font_size=init_graph_display(graph_shape='sqr', \
        graph_grid='/', x_label="$P_3$", y_label="Y axis",
prob_axes=True, \
        title=title, **kwargs)
    #sets the marker_size relative to the font and thus the graph
size
    marker_size = (label_font_size-1)

    #plots the pr2bias in order G3/(G3+C3),A3/(A3+T3),
    #                            G3/(G3/A3),C3/(C3+T3),
    #                            G3/(G3+T3),C3/(C3+A3)
    #colors and symbols coded from Sueoka 2002
    plot([data[:,0]], [data[:,1]], '-ko', c='k',
         markersize=marker_size)
    plot([data[:,0]], [data[:,2]], '-kv', c='k',
         markersize=marker_size)
    plot([data[:,0]], [data[:,3]], '-ro', c='r',
         markersize=marker_size)
    plot([data[:,0]], [data[:,4]], '-rv', c='r',
         markersize=marker_size)
    plot([data[:,0]], [data[:,5]], '-wo', c='k',
         markersize=marker_size)
    plot([data[:,0]], [data[:,6]], '-wv', c='k',
         markersize=marker_size)

    set_axis_to_probs()

    #aaLabel based on the amino acid that is graphed
    #C2 type: ala(GCN), pro(CCN), ser4(TCN), thr(ACN)
    #G2 type: arg4 (CGN), an gly(GGN)
    #T2 type: leu4(CTN), val (GTN) (Sueoka 2002)
    text(.95, .05, aaLabels[title], font,
verticalalignment='bottom',
         horizontalalignment='right')

    #output the figure
    set_axis_to_probs()
    savefig(graph_name)


def plot_histograms(data, graph_name='gene_histogram.png', bins=20,\
        normal_fit=True, normed=True, colors=None, linecolors=None,
\
        alpha=0.75, prob_axes=True, series_names=None,
show_legend=False,\
        y_label=None, **kwargs):
    """Outputs a histogram with multiple series (must provide a list
of series).

    takes:  data: list of arrays of values to plot (needs to be list
of arrays
            so you can pass in arrays with different numbers of
elements)

            graph_name: filename to write graph to
            bins: number of bins to use
            normal_fit: whether to show the normal curve best
fitting the data
            normed: whether to normalize the histogram (e.g. so bars
sum to 1)
            colors: list of colors to use for bars
            linecolors: list of colors to use for fit lines

            **kwargs are pssed on to init_graph_display.

    """
    if y_label is None:
        if normed:
            y_label='Frequency'
        else:
            y_label='Count'
    num_series = len(data)
    if colors is None:
        if num_series == 1:
            colors = ['white']
        else:
            colors = standard_series_colors
    if linecolors is None:
        if num_series == 1:
            linecolors = ['red']
        else:
            linecolors = standard_series_colors

    init_graph_display(prob_axes=prob_axes, y_label=y_label,
**kwargs)
    for i, d in enumerate(data):
        fc = colors[i % len(colors)]
        lc = linecolors[i % len(linecolors)]

        counts, x_bins, patches = hist(d, bins=bins, normed=normed,
\
            alpha=alpha, facecolor=fc)

        if normal_fit:
            mu = mean(d)
            sigma = std(d)
            bin_width = x_bins[-1] - x_bins[-2]
            #want normpdf to extend over the bins, so needs to be
one extra
            #normpdf_bins = x_bins + bin_width/2.0
            normpdf_bins = arange(0,1,0.01)
            y = normpdf(normpdf_bins, mu, sigma)
            orig_area = sum(counts) * bin_width
            y = y * orig_area   #normpdf area is 1 by default
            plot(normpdf_bins, y, linestyle='--', color=lc,
linewidth=1)

    if show_legend and series_names:
        legend(series_names)

    #output figure
    savefig(graph_name)

def plot_scatter_with_histograms(data,
graph_name='histo_scatter.png', \
    graph_grid='/', prob_axes=False, bins=20, frac=0.9,
scatter_alpha=0.5, \
    hist_alpha=0.8, colors=standard_series_colors, normed=True,
**kwargs):
    """Plots a scatter plot with histograms showing distribution of
x and y.

    Data should be list of [x1, y1, x2, y2, ...].
    """

    #set up subplot coords
    tl=subplot(2,2,1)
    br=subplot(2,2,4)
    bl=subplot(2,2,3, sharex=tl, sharey=br)

    #get_position returns left, bottom, width, height relative to
figure
    tl_coords = tl.get_position()
    bl_coords = bl.get_position()
    br_coords = br.get_position()

    left = tl_coords[0]
    bottom = bl_coords[1]

    width = br_coords[0] + br_coords[2] - left
    height = tl_coords[1] + tl_coords[3] - bottom

    bl.set_position([left, bottom, frac*width, frac*height])
    tl.set_position([left, bottom+(frac*height), frac*width, (1-
frac)*height])
    br.set_position([left+(frac*width), bottom, (1-frac)*width,
frac*height])

    #suppress frame and axis for histograms
    for i in [tl,br]:
        i.set_frame_on(False)
        i.xaxis.set_visible(False)
        i.yaxis.set_visible(False)

    plot_scatter(data=data, alpha=scatter_alpha, axes=bl, **kwargs)

    biggest_x = 0
    biggest_y = 0
    for i in range(0, len(data), 2):
        x, y = data[i], data[i+1]
        color = colors[(i/2)%len(colors)]
        n, bins, patches = hist(x, facecolor=color, bins=bins, \
            alpha=hist_alpha, axes=tl, normed=True)
        biggest_x = max([biggest_x, max(x)])
        n, bins, patches = hist(y, facecolor=color, bins=bins, \
            alpha=hist_alpha, axes=br, normed=normed,
orientation='horizontal')
        biggest_y=max([biggest_y,max(y)])
    bl.set_xlim(0,1)
    bl.set_ylim(0,1)
    savefig(graph_name)
```

3. Sliding window code

```
#/usr/lib/python2.4/
#AminoGraphPlots.py

from matplotlib import use, rc
use('Agg')  #suppress graphical rendering
rc('text', usetex=True)
rc('font', family='serif')  #required to match latex text and
equations
import Image
import ImageFilter
from Numeric import array, shape, fromstring
from cogent.base.usage import UnsafeCodonUsage as CodonUsage
from cogent.maths.stats.test import regress, correlation
from pylab import *
from math import pi

"""Provides different kinds of codon usage plots.

See individual docstrings for more info.
"""
#module-level constants

#historical doublet order for fingerprint plot; not currently used,
but
#same order that the colors were entered in. Matches Sueoka 2002.
doublet_order =
['GC','CG','GG','CU','CC','UC','AC','GU','UU','CA','AU',\
                 'AA','AG','GA','UA','UG']
color_order = ["#000000","#FF0000","#00FF00","#FFFF00",
          "#CC99FF","#FFCC99","#CCFFFF","#C0C0C0",
          "#6D6D6D","#2353FF","#00FFFF","#FF8800",
          "#238853","#882353","#EC008C","#000099"]
#map doublets to colors so we can make sure the same doublet always
#gets the same colors
doublets_to_colors = dict(zip(doublet_order, color_order))
#creates a dictionary for the amino acid labels, less to input
aaLabels={'ALANINE':'GCN', 'ARGININE4':'CGN', 'GLYCINE':'GGN',
          'LEUCINE4':'CTN', 'PROLINE':'CCN', 'SERINE4':'TCN',
          'THREONINE':'ACN', 'VALINE':'GTN'}
standard_series_colors=['k','r','g','b', 'm','c']

#Helper functions

def hist(x, bins=10, normed='height', bottom=0, \
    orientation='vertical', width=None, axes=None, **kwargs):
    """Just like the matplotlib hist, but normalizes bar heights to
1.

    axes uses gca() by default (built-in hist is a method of Axes).

    Original docs from matplotlib:

    HIST(x, bins=10, normed=0, bottom=0, orientiation='vertical',
**kwargs)
```

```
    Compute the histogram of x.  bins is either an integer number of
    bins or a sequence giving the bins.  x are the data to be
binned.

    The return values is (n, bins, patches)

    If normed is true, the first element of the return tuple will
    be the counts normalized to form a probability density, ie,
    n/(len(x)*dbin)


    orientation = 'horizontal' | 'vertical'.  If horizontal, barh
    will be used and the "bottom" kwarg will be the left.

    width: the width of the bars.  If None, automatically compute
    the width.

    kwargs are used to update the properties of the
    hist bars
    """
    if axes is None:
        axes = gca()
    if not axes._hold: axes.cla()
    n,bins = norm_hist_bins(x, bins, normed)
    if width is None: width = 0.9*(bins[1]-bins[0])
    if orientation=='horizontal':
        patches = axes.barh(n, bins, height=width, left=bottom)
    else:
        patches = axes.bar(bins, n, width=width, bottom=bottom)
    for p in patches:
        p.update(kwargs)
    return n, bins, silent_list('Patch', patches)

def norm_hist_bins(y, bins=10, normed='height'):
    """Just like the matplotlib mlab.hist, but can normalize by
height.

    normed can be 'area' (produces matplotlib behavior, area is 1),
    any False value (no normalization), or any True value
(normalization).

    Original docs from matplotlib:

    Return the histogram of y with bins equally sized bins.  If bins
    is an array, use the bins.  Return value is
    (n,x) where n is the count for each bin in x

    If normed is False, return the counts in the first element of
the
    return tuple.  If normed is True, return the probability density
    n/(len(y)*dbin)

    If y has rank>1, it will be raveled
    Credits: the Numeric 22 documentation
    """
    y = asarray(y)
    if len(y.shape)>1: y = ravel(y)

    if not iterable(bins):
        ymin, ymax = min(y), max(y)
        if ymin==ymax:
            ymin -= 0.5
            ymax += 0.5

        if bins==1: bins=ymax
        dy = (ymax-ymin)/bins
        bins = ymin + dy*arange(bins)
    n = searchsorted(sort(y), bins)
    n = diff(concatenate([n, [len(y)]]))
    if normed:
        if normed == 'area':
            db = bins[1]-bins[0]
        else:
            db = 1.0
        return 1/(len(y)*db)*n, bins
    else:
        return n, bins

def as_species(name):
    """Cleans up a filename into a species name, italicizing it in
latex."""
    #trim extension if present
    dot_location = name.rfind('.')
    if dot_location > -1:
        name = name[:dot_location]
    #get rid of _small if present -- used for debugging
    if name.endswith('_small'):
        name = name[:-len('_small')]
    #replace underscores with spaces
    name = name.replace('_', ' ')
    #make sure the first letter of the genus is caps, and not the
first letter
    #of the species
    fields = name.split()
    fields[0] = fields[0].title()
    #assume second field is species name
    if len(fields) > 1:
        fields[1] = fields[1].lower()
    return '\emph{'+' '.join(fields)+'}'

def frac_to_psq(frac, graph_size):
    """Converts diameter as fraction of graph to points squared for
scatter.

    frac: fraction of graph (e.g. .01 is 1% of graph size)
    graph_size: graph size in inches
    """
    points = frac * graph_size * 72
    return pi * (points/2.0)**2
```

```
def init_graph_display(title=None, aux_title=None, size=4.0, \
    graph_shape='sqr', graph_grid=None, x_label='', y_label='', \
    dark=False, with_parens=True, prob_axes=True, axes=None,
num_genes=None):
    """Initializes a range of graph settings for standard plots.

    These settings include:
        - font sizes based on the size of the graph
        - graph shape
        - grid, including lines for x=y or at x and y = 0.5
        - title, auxillary title, and x and y axis labels

    Parameters:
        title: displayed on left of graph, at the top, latex-format
string

        aux_title: displayed on top right of graph, latex-format
string.
        typically used for number of genes.

        size:   size of graph, in inches

        graph_shape: 'sqr' for square graphs, 'rect' for graphs that
include
        a colorbar, 3to1: width 3 to height 1.

        graph_grid: background grid for the graph. Currently
recognized grids
        are '/' (line at x=y) and 't' (cross at x=.5 and y=.5).

        x_label: label for x axis, latex-format string.

        y_label: label for y axis, latex-format string.

        dark: set to True if dark background, reverses text and tick
colors.

        with_parens: if True (default), puts parens around auxillary
title

    returns font, label_font_size (for use in producing additional
labels in
    calling function).
    """
    if dark:
        color='w'
    else:
        color='k'
    min_offset = 0.05           #minimum offset, e.g. for text
    max_offset = 1-min_offset   #center offsets
    rect_scale_factor = 1.28    #need to allow for legend while
keeping graph
                                #square; empirically determined at
1.28
    font_size = int(size*3-1)   #want 11pt font w/ default graph
size 4" sqr
```

```
    label_scale_factor = 0.8
    label_font_size = font_size * label_scale_factor
    label_offset = label_font_size * 0.5
    axis_label_font={'fontsize':font_size}
    font={'fontsize':font_size, 'color':color}


    if graph_shape == 'sqr':
        gcf().set_figsize_inches(size,size)
    elif graph_shape == 'rect':
        #scaling for sqr graphs with colorbar
        gcf().set_figsize_inches(size*rect_scale_factor,size)
    elif graph_shape == '3to1':
        gcf().set_figsize_inches(3*size, size)
    elif graph_shape == '2to1':
        gcf().set_figsize_inches(2*size, size)
    else:
        raise ValueError, "Got unknown graph shape %s" % graph_shape

    #set or create axes
    if axes is None:
        axes = gca()

    #draw grid manually: these are in data coordinates.
    if graph_grid == 't':
        #grid lines at 0.5 on each axis, horiz & vertic
        axes.axvline(x=.5, ymin=0, ymax=1, color=color,
linestyle=':')
        axes.axhline(y=.5, xmin=0, xmax=1, color=color,
linestyle=':')
    elif graph_grid == '/':
        #diagonal gridlines from 0,0 to 1,1.
        axes.plot([0,1], color=color, linestyle=':')
    else:
        pass    #ignore other choices

    #remove default grid
    axes.grid(False)

    #set x and y labels
    axes.set_ylabel(y_label, axis_label_font)
    axes.set_xlabel(x_label, axis_label_font)

    #add title/aux_title to graph directly. Note that we want
    #the tops of these to be fixed, and we want the label to be
    #left-justified and the number of genes to be right justified,
    #so that it still works when we resize the graph.
    if title is not None:
        axes.text(min_offset, max_offset, str(title), font, \
            verticalalignment='top', horizontalalignment='left')
    #use num_genes as aux_title by default
    aux_title = num_genes or aux_title
    if aux_title is not None:
        if with_parens:
            aux_title='('+str(aux_title)+')'
        axes.text(max_offset, max_offset, str(aux_title), font,
             verticalalignment='top', horizontalalignment='right')
    if prob_axes:
        init_ticks(axes, label_font_size, dark)
    #set x and y label offsets -- currently though rcParams, but
should be
    #able to do at instance level?
    #rc('xtick.major', pad=label_offset)
    #rc('ytick.major', pad=label_offset)
    return font, label_font_size

def init_ticks(a, label_font_size, dark=False):
    """takes a from (a = gca)
    sets the ticks to span from 0 to 1 with .1 intervals
    changes the size of the ticks and the corresponding number
labels
    """
    a.set_xticks(arange(0,1.01,.1),)
    a.set_yticks(arange(0,1.01,.1))

    #reset sizes for x and y labels
    x = a.get_xticklabels()
    y = a.get_yticklabels()
    for l in a.get_xticklabels() + a.get_yticklabels():
        l.set_fontsize(label_font_size)
    #if dark, need to reset color of internal ticks to white
    if dark:
        for l in a.get_xticklines() + a.get_yticklines():
            l.set_markeredgecolor('white')

def set_axis_to_probs(axes=None):
    """sets the axes to span from 0 to 1
    necessary because order in program changes graph
    """
    #set axis for probabilities (range 0 to 1)
    if axes is None:
        axes = gca()
    axes.set_xlim([0,1])
    axes.set_ylim([0,1])

def plot_regression_line(x_data,y_data,line_color='r', axes=None):
    """Plots the regression line, and returns the equation."""
    if axes is None:
        axes = gca()
    m, b = regress(x_data, y_data)
    r, significance = correlation(x_data,y_data)
    #set the a,b,r values
    r_str = '%0.3g'% (r**2)
    m_str ='%0.3g' % m
    b_str = '%0.3g' % b
    x1=0.0
    y1=b
    x2=1.0
    y2=(m+b)

    #constrain so that y is always in the range (0,1) for plotting
    if(y1<0):
        y1=0.0
        x1=(0-b)/m
    if(y2>=1):
        y2=1.0
        x2=(1-b)/m

    axes.plot([x1,x2],[y1,y2], color=line_color, linewidth=2)

    if b >= 0:
        sign_str = ' + '
    else:
        sign_str = ' '

    equation=''.join(['y=
',m_str,'x',sign_str,b_str,'\n\nr$^2$=',r_str])
    return equation, line_color

def print_regression_equations(equations, axes=None):
    """Writes list of regression equations to graph.

    equations: list of regression equations

    size: size of the graph in inches
    """
    if axes is None:
        axes = gca()
    for i, (eq_text, eq_color) in enumerate(equations):
        axes.text((0.98), (0.02+(.06*i)), str(eq_text), \
            horizontalalignment='right', verticalalignment='bottom',
\
            color=eq_color)


def broadcast(i, n):
    """Broadcasts i to a vector of length n."""
    try:
        i = list(i)
    except:
        i = [i]
    reps, leftovers = divmod(n, len(i))
    return (i * reps) + i[:leftovers]


#scatterplot functions and helpers

def plot_scatter(data, series_names=None, \
    series_color=standard_series_colors,
line_color=standard_series_colors,\
    alpha=0.25, marker_size=.015, scale_markers=True,
    show_legend=True,legend_loc='center right',
    show_regression=True, show_equation=True,
    prob_axes=False, size=8.0, axes=None,
    **kwargs):
    """helper plots one or more series of scatter data of specified
color,
    calls the initializing functions, doesn't print graph

    takes: plotted_pairs, series_names, show_legend, legend_loc, and
```

```
        **kwargs passed on to init_graph_display (these include
title,
        aux_title, size, graph_shape, graph_grid, x_label, y_label,
        dark, with_parens).

    plotted_pairs = (first_pos, second_pos, dot_color, line_color,
    alpha, show_regression, show_equation)

    returns the regression str equation (list) if regression is set
true

    suppresses legend if series not named, even if show_legend is
True.
    """
    if not axes:
        axes = gca()
    #initialize fonts, shape and labels
    font,label_font_size=init_graph_display(prob_axes=prob_axes, \
        size=size, axes=axes, **kwargs)
    equations = []
    #figure out how many series there are, and scale vals
accordingly
    num_series = len(data)/2
    series_color = broadcast(series_color, num_series)
    line_color = broadcast(line_color, num_series)
    alpha = broadcast(alpha, num_series)
    marker_size = broadcast(marker_size, num_series)
    if scale_markers:
        marker_size = [frac_to_psq(m, size) for m in marker_size]

    series = []
    for i in range(num_series):
        x, y = data[2*i], data[2*i+1]

series.append(axes.scatter(x,y,s=marker_size[i],c=series_color[i],\
        alpha=alpha[i]))
        #find the equation and plots the regression line if True
        if show_regression:
            equation = plot_regression_line(x,y,line_color[i],
axes=axes)
        if show_equation:
            equations.append(equation)  #will be (str, color) tuple
    #print all the regression equations at once -- need to know how
many
    if show_regression:
        print_regression_equations(equations, axes=axes)
    #clean up axes if necessary
    if show_legend and series_names: #suppress legend if series not
named
        axes.legend(series, series_names, legend_loc)

    if prob_axes:
        set_axis_to_probs(axes)
    return equations, font

def plot_cai_p3_scatter(data, graph_name='cai_p3_scat.png',
**kwargs):
    """Outputs a CAI vs P3 scatter plot.

    expects data as ([P3s_1, CAIs_1, P3s_2, CAIs_2, ...])
    """
    plot_scatter(data, graph_shape='sqr', graph_grid=None,\
        x_label="$P_3$",y_label="CAI", prob_axes=True,**kwargs)
    savefig(graph_name)

def plot_p12_p3(data, graph_name='p12_p3.png', **kwargs):
    """Outputs a P12 versus P3 scatter graph, optionally including
regression.

    expects data as [P3_1, P12_1, P3_2, P12_2, ...n ].
    """
    plot_scatter(data, graph_shape='sqr', graph_grid='/',\
        x_label="$P_3$",y_label="$P_{12}$", prob_axes=True,
**kwargs)
    savefig(graph_name)

def plot_p123_gc(data, graph_name='p123_gc.png', use_p3_as_x=False,
**kwargs):
    """Output a scatter plot of p1,p2,p3 vs gc content

    Expects data as array with rows as GC, P1, P2, P3
    p1=blue, p2=green, p3=red

    """
    #unpack common x axis, and decide on series names
    if use_p3_as_x:
        series_names = ['$P_1$', '$P_2$']
        colors=['b','g']
        x_label='$P_3$'
        y_label='$P_{12}$'
        xy_pairs = [data[3], data[1], data[3], data[2]]
    else:
        series_names = ['$P_1$', '$P_2$', '$P_3$']
        colors=['b','g','r']
        x_label='GC'
        y_label='$P_{123}$'
        xy_pairs = [data[0], data[1], data[0], data[2], data[0],
data[3]]

    #plot points and write graph
    plot_scatter(xy_pairs,
graph_grid='/',x_label=x_label,y_label=y_label,
        series_names=series_names, prob_axes=True, **kwargs)
    savefig(graph_name)

def plot_fingerprint(data, alpha=0.7, \
    show_legend=True, graph_name='fingerprint.png', has_mean=True,
    which_blocks='quartets', multiple=False, graph_grid='t',
prob_axes=True, \
    **kwargs):
    """Outputs a bubble plot of four-codon amino acid blocks
    labeled with the colors from Sueoka 2002.

    takes: data:  array-elements in the col order x, y, r of
```

```
            each of the four codon Amino Acids in the row order:
            ALA, ARG4, GLY, LEU4, PRO, SER, THR, VAL
            (for traditional fingerprint), or:
            UU -> GG (for 16-block fingerprint).
            last row is the mean (if has_mean is set True)

        **kwargs passed on to init_graph_display (these include
        graph_shape, graph_grid, x_label, y_label, dark,
with_parens).

            title: will be printed on graph (default: 'Unknown
Species')

            num_genes (number of genes contributing to graph: default
None)
            NOTE: will not print if None.)

            size: of graph in inches (default = 8.0)

            alpha: transparency of bubbles
            (ranges from 0, transparent, to 1, opaque; default 0.7)

            show_legend: bool, default True, whether to print legend

            graph_name: name of file to write (default
'fingerprint.png')

            has_mean: whether the data contain the mean (default:
True)

            which_blocks: which codon blocks to print (default is
'quartets'
            for the 4-codon amino acid blocks, but can also use 'all'
for all
            quartets or 'split' for just the split quartets.)

            multiple: if False (the default), assumes it got a single
block
            of data. Otherwise, assumes multiple blocks of data in a
list or
            array.

    note: that the data are always expected to be in the range (0,1)
    since we're plotting frequencies. axes, gid, etc. are hard-coded
    to these values.
    """
    #figure out which type of fingerprint plot we're doing, and get
the
    #right colors
    if which_blocks == 'quartets':
        blocks = CodonUsage.SingleAABlocks
    elif which_blocks == 'split':
        blocks = CodonUsage.SplitBlocks
    else:
        blocks = CodonUsage.Blocks

    colors = [doublets_to_colors[i] for i in blocks]
    #formatting the labels in latex
    x_label="$G_3/(G_3+C_3)$"
    y_label="$A_3/(A_3+T_3)$"

    #initializing components of the graph
    font,label_font_size=init_graph_display(graph_shape='sqr', \
        graph_grid=graph_grid, x_label=x_label, \
        y_label=y_label, prob_axes=prob_axes, **kwargs)

    if not multiple:
        data = [data]

    alpha = broadcast(alpha, len(data))

    for al, d in zip(alpha, data):
        #skip this series if no data
        if not d:
            continue
        for i, color in enumerate(colors):
            j = i+1
            #note: doing these as slices because scatter_classic
needs the
            #extra level of nesting
            patches = scatter_classic(d[i:j,0], d[i:j,1],
                        s=(d[i:j,2]/2), c=color)
            #set alpha for the patches manually
            for p in patches:
                p.set_alpha(al)

        #plot mean as its own point -- can't do cross with scatter
        if has_mean:
            mean_index = len(blocks)    #next index after the blocks
            plot([d[mean_index,0]], [d[mean_index,1]],
                 '-k+',markersize=label_font_size, alpha=al)


    abbrev = CodonUsage.BlockAbbreviations

    a = gca()
    #if show_legend is True prints a legend in the right center area
    if show_legend:
        legend_key = [abbrev[b] for b in blocks]
        #copy legend font properties from the x axis tick labels
        legend_font_props = \
            a.xaxis.get_label().get_font_properties().copy()
        legend_font_scale_factor = 0.7
        curr_size = legend_font_props.get_size()

legend_font_props.set_size(curr_size*legend_font_scale_factor)
        l = figlegend(a.patches,
                  legend_key,
                  prop=legend_font_props,
                  loc='center right',pad=0.1,labelsep=0.0025,
                  handlelen=0.02,handletextsep=0.007, axespad=0.0)
        #fix transparency of patches
        for p in l.get_patches():
            p.set_alpha(1)

    #initialize the ticks
    set_axis_to_probs()
    init_ticks(a, label_font_size)
    a.set_xticks([0, 0.5, 1])
    a.set_yticks([0,0.5,1])

    #output the figure
    savefig(graph_name)

#Contour plots and related functions

def plot_filled_contour(plot_data, xy_data=None,
show_regression=False, \
    show_equation=False, fill_cmap=cm.hot, graph_shape='rect', \
    num_contour_lines=10, **kwargs):
    """helper plots one or more series of contour data
    calls the initializing functions, doesn't output figure

    takes: plot_data, xy_data, show_regression, show_equation,
fill_cmap,
    and **kwargs passed on to init_graph_display.

           plot_data = (x_bin, y_bin, data_matrix dot_colors)
    """
    if show_regression:
        equation = plot_regression_line(xy_data[:,0],xy_data[:,1])
        if show_equation:
            print_regression_equations([equation])
    #init graph display, rectangular due to needed colorbar space
    init_graph_display(graph_shape=graph_shape, **kwargs)
    #plots the contour data
    for x_bin,y_bin,data_matrix in plot_data:
        contourf(x_bin,y_bin,data_matrix, num_contour_lines,
cmap=fill_cmap)
    #add the colorbar legend to the side
    colorbar()

def plot_contour_lines(plot_data, xy_data=None,
show_regression=False, \
        show_equation=False, smooth_steps=0, num_contour_lines=10, \
        label_contours=False, line_cmap=cm.hot,
fill_cmap=cm.gray,dark=True,
        graph_shape='rect', **kwargs):
    """helper plots one or more series of contour line data
    calls the initializing functions, doesn't output figure

    takes: plot_data, xy_data, show_regression, show_equation,
smooth,
        num_contour_lines, label_contours, line_cmap, fill_cmap,
graph_shape,
        and **kwargs passed on to init_graph_display.

           plot_data = (x_bin, y_bin, data_matrix dot_colors)
    """
    #init graph display, rectangular due to needed colorbar space
    init_graph_display(graph_shape=graph_shape,
        dark=dark, **kwargs)
    #plots the contour data
    for x_bin,y_bin,data in plot_data:
        orig_max = max(ravel(data))
        scaled_data = (data/orig_max*255).astype('b')
        if smooth_steps:
            orig_shape = data.shape
            im = Image.fromstring('L', data.shape, scaled_data)
            for i in range(smooth_steps):
                im = im.filter(ImageFilter.BLUR)
            new_data = fromstring(im.tostring(), 'b')
            data = reshape(new_data.astype('i')/255.0 * orig_max,
orig_shape)

        if fill_cmap is not None:
            im = imshow(data, interpolation='bicubic',
extent=(0,1,0,1), \
                origin='lower', cmap=fill_cmap)
        result=contour(x_bin,y_bin,data, num_contour_lines,
                              origin='lower',linewidth=2,
                              extent=(0,1,0,1), cmap=line_cmap)
        if label_contours:
            clabel(result, fmt='%1.1g')

    #add the colorbar legend to the side
    cb = colorbar()
    cb.set_axis_bgcolor('black')

    if show_regression:
        equation=plot_regression_line(xy_data[0],xy_data[1])
        if show_equation:
            print_regression_equations([equation])

def plot_cai_p3_contour(x_bin,y_bin,data,xy_data,
                        graph_name='cai_contour.png',
                        prob_axes=True, **kwargs):
    """Output a contour plot of cai vs p3 with colorbar on side

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'cai_contour.png'
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_filled_contour(plot_data, graph_grid='/',x_label="$P_3$", \
        y_label="CAI", prob_axes=prob_axes, **kwargs)
    set_axis_to_probs()
    savefig(graph_name)

def plot_cai_p3_contourlines(x_bin,y_bin,data,xy_data,
                             graph_name='cai_contourlines.png',
                              prob_axes=True, **kwargs):
    """Output a contour plot of cai

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'cai_contourlines.png'
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_contour_lines(plot_data, graph_grid='/', x_label="$P_3$", \
        y_label="CAI", prob_axes=prob_axes,**kwargs)
    savefig(graph_name)

def plot_p12_p3_contour(x_bin,y_bin,data,xy_data,
                        graph_name='p12_p3_contour.png',
                        prob_axes=True, **kwargs):
    """Outputs a P12 versus P3 contour graph
    and the mean equation of the plot

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'p12_p3_contourlines.png'
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_filled_contour(plot_data, graph_grid='/', x_label="$P_3$",
\
        y_label="$P_{12}$", prob_axes=prob_axes,**kwargs)
    set_axis_to_probs()
    savefig(graph_name)

def plot_p12_p3_contourlines(x_bin,y_bin,data,xy_data,
prob_axes=True,\
    graph_name='p12_p3_contourlines.png', **kwargs):
    """Outputs a P12 versus P3 contourline graph
    and the mean equation of the plot

    takes: x_bin, y_bin, data (data matrix)

           label (default 'Unknown Species')

           num_genes (default 0 will not print, other numbers will)

           size: of graph in inches (default = 8.0)

           graph_name: default 'p12_p3_contourlines.png
    """
    plot_data =[(x_bin,y_bin,data)]
    plot_contour_lines(plot_data, graph_grid='/', x_label="$P_3$",\
        y_label="$P_{12}$", prob_axes=prob_axes, **kwargs)
    set_axis_to_probs()
    savefig(graph_name)

#Other graphs

def plot_pr2_bias(data, title='ALANINE', graph_name='pr2_bias.png',
\
    num_genes='ignored', **kwargs):
    """Outputs a PR2-Bias plot of:
    -isotypic transversions (base swapping)
    with G3/(G3+C3) and A3/(A3+T3)
    -Transitions (deaminations)
    with G3/(G3+A3) and C3/(C3+T3)
    -Allotypic transversions (G- oxidations)
    with G3/(G3+T3) and C3/(C3+A3)

    takes: an array in the order: x,G3/(G3+C3),A3/(A3+T3),
    G3/(G3/A3),C3/(C3+T3),G3/(G3+T3),C3/(C3+A3)

    label: default 'ALANINE'
    one amino acid written out in caps:
    ALANINE, ARGININE4, GLYCINE, LEUCINE4,
    PROLINE, SERINE4, THREONINE, VALINE
       from one of the amino acids program will add acronym
       C2 type: ala(GCN), pro(CCN), ser4(TCN), thr(ACN)
       G2 type: arg4 (CGN), an gly(GGN)
       T2 type: leu4(CTN), val (GTN)

    size: of graph in inches (default = 8.0)

    graph_name: default 'pr2_bias.png'

    num_genes: number of genes contributing to graph, currently
ignored.
    """
    #we can't put anything in the top right, so print num_genes
after the title
    #if it was supplied
    #initializes the graph display and font
    font,label_font_size=init_graph_display(graph_shape='sqr', \
        graph_grid='/', x_label="$P_3$", y_label="Y axis",
prob_axes=True, \
        title=title, **kwargs)
    #sets the marker_size relative to the font and thus the graph
size
    marker_size = (label_font_size-1)

    #plots the pr2bias in order G3/(G3+C3),A3/(A3+T3),
    #                           G3/(G3/A3),C3/(C3+T3),
    #                           G3/(G3+T3),C3/(C3+A3)
    #colors and symbols coded from Sueoka 2002
    plot([data[:,0]], [data[:,1]], '-ko', c='k',
         markersize=marker_size)
    plot([data[:,0]], [data[:,2]], '-kv', c='k',
         markersize=marker_size)
    plot([data[:,0]], [data[:,3]], '-ro', c='r',
         markersize=marker_size)
    plot([data[:,0]], [data[:,4]], '-rv', c='r',
         markersize=marker_size)
    plot([data[:,0]], [data[:,5]], '-wo', c='k',
         markersize=marker_size)
    plot([data[:,0]], [data[:,6]], '-wv', c='k',
         markersize=marker_size)

    set_axis_to_probs()

    #aaLabel based on the amino acid that is graphed
    #C2 type: ala(GCN), pro(CCN), ser4(TCN), thr(ACN)
    #G2 type: arg4 (CGN), an gly(GGN)
    #T2 type: leu4(CTN), val (GTN) (Sueoka 2002)
    text(.95, .05, aaLabels[title], font,
verticalalignment='bottom',
         horizontalalignment='right')

    #output the figure
    set_axis_to_probs()
    savefig(graph_name)


def plot_histograms(data, graph_name='gene_histogram.png', bins=20,\
        normal_fit=True, normed=True, colors=None, linecolors=None,
\
        alpha=0.75, prob_axes=True, series_names=None,
show_legend=False,\
        y_label=None, **kwargs):
    """Outputs a histogram with multiple series (must provide a list
of series).

    takes:  data: list of arrays of values to plot (needs to be list
of arrays
            so you can pass in arrays with different numbers of
elements)

            graph_name: filename to write graph to
            bins: number of bins to use
            normal_fit: whether to show the normal curve best
fitting the data
            normed: whether to normalize the histogram (e.g. so bars
sum to 1)
            colors: list of colors to use for bars
            linecolors: list of colors to use for fit lines

            **kwargs are pssed on to init_graph_display.

    """
    if y_label is None:
        if normed:
            y_label='Frequency'
        else:
            y_label='Count'
    num_series = len(data)
    if colors is None:
        if num_series == 1:
            colors = ['white']
        else:
            colors = standard_series_colors
    if linecolors is None:
        if num_series == 1:
            linecolors = ['red']
        else:
            linecolors = standard_series_colors

    init_graph_display(prob_axes=prob_axes, y_label=y_label,
**kwargs)
    for i, d in enumerate(data):
        fc = colors[i % len(colors)]
        lc = linecolors[i % len(linecolors)]

        counts, x_bins, patches = hist(d, bins=bins, normed=normed,
\
            alpha=alpha, facecolor=fc)

        if normal_fit:
            mu = mean(d)
            sigma = std(d)
            bin_width = x_bins[-1] - x_bins[-2]
            #want normpdf to extend over the bins, so needs to be
one extra
            #normpdf_bins = x_bins + bin_width/2.0
            normpdf_bins = arange(0,1,0.01)
            y = normpdf(normpdf_bins, mu, sigma)
            orig_area = sum(counts) * bin_width
            y = y * orig_area   #normpdf area is 1 by default
            plot(normpdf_bins, y, linestyle='--', color=lc,
linewidth=1)

    if show_legend and series_names:
        legend(series_names)

    #output figure
    savefig(graph_name)

def plot_scatter_with_histograms(data,
graph_name='histo_scatter.png', \
    graph_grid='/', prob_axes=False, bins=20, frac=0.9,
scatter_alpha=0.5, \
    hist_alpha=0.8, colors=standard_series_colors, normed=True,
**kwargs):
    """Plots a scatter plot with histograms showing distribution of
x and y.

    Data should be list of [x1, y1, x2, y2, ...].
    """

    #set up subplot coords
    tl=subplot(2,2,1)
    br=subplot(2,2,4)
    bl=subplot(2,2,3, sharex=tl, sharey=br)
    #get_position returns left, bottom, width, height relative to
figure
    tl_coords = tl.get_position()
    bl_coords = bl.get_position()
    br_coords = br.get_position()

    left = tl_coords[0]
    bottom = bl_coords[1]

    width = br_coords[0] + br_coords[2] - left
    height = tl_coords[1] + tl_coords[3] - bottom

    bl.set_position([left, bottom, frac*width, frac*height])
    tl.set_position([left, bottom+(frac*height), frac*width, (1-
frac)*height])
    br.set_position([left+(frac*width), bottom, (1-frac)*width,
frac*height])

    #suppress frame and axis for histograms
    for i in [tl,br]:
        i.set_frame_on(False)
        i.xaxis.set_visible(False)
        i.yaxis.set_visible(False)

    plot_scatter(data=data, alpha=scatter_alpha, axes=bl, **kwargs)

    biggest_x = 0
    biggest_y = 0
    for i in range(0, len(data), 2):
        x, y = data[i], data[i+1]
        color = colors[(i/2)%len(colors)]
        n, bins, patches = hist(x, facecolor=color, bins=bins, \
            alpha=hist_alpha, axes=tl, normed=True)
        biggest_x = max([biggest_x, max(x)])
        n, bins, patches = hist(y, facecolor=color, bins=bins, \
            alpha=hist_alpha, axes=br, normed=normed,
orientation='horizontal')
        biggest_y=max([biggest_y,max(y)])
    bl.set_xlim(0,1)
    bl.set_ylim(0,1)
    savefig(graph_name)
```

References

Bafna, V., and Pevner, P. 1996. Genome rearrangement and sorting by reversals. SIAM J Comput. Vol. 25, No. 2, 272-89

Binz, T., Blasi, J., Yamasaki, S., Baumeister, A., Link, E., Sudhof, TC. Jahn, R., and Niemann, H. 1994. Proteolysis of SNAP-25 by types E and A botulinal neurotoxins. J Biol Chem. 269(3):1617-20.

Bos, D, and Posada, D. 2004. Using Models of Nucleotide Evolution to Build Phylogenetic Trees. Development and Comparative Immunology. PubMed.

Bräumler, A. 1997. The record of horizontal gene transfer in Salmonella. Trends in Microbiology. Vol. 5. No. 8. August.

Canchaya, C. Fournous, G., Chibani-Chennoufi, S., Dillmann, M. L., and Brussow, H. 2003. Phage as agents of lateral gene transfer. Curr Opin Microbiol. 6(4):417-24.

Center for Disease Control and Prevention. 2003. Frequently asked questions (FAQ) about *Tularemia.* http://www.bt.cdc.gov/agent/tularemia/faq.asp. Accessed 1 May 2007

Chen, S., Kim, J. J., and Barbieri, J. T. 2007. Mechanisms of the substrate recognition by botulinum neurotoxin serotype A. J Biol Chem. 282(13): 9621-7.

Collmer, A., Lindeberg, M., et al. 2002. Genomic Mining Type II Secretion System Effectors in Pseudomonas Syringae Yields New Picks for all TTSS Prospectors. Trends Microbiology 10(10): 462-9

Cuschieri, Alfred. Cell Biology, Genetics and Embryology. University of Malta. http://staff.um.edu.mt/acus1/4genfunction_files/image005.gif. Accessed 30 April 2007.

Ehrbar, K. and Hardt, W. D. (2005) Bacteriophage-encoded Type III Effectors in Salmonella Enterica Subspecies1 Serovar Typhimurium. Infect Genet Evol 5(1):1-9

Fortier, A. H., Green, S. J., Polsinella, T., Jones, T. R., Crawford, R. M., Leiby, D. A., Elkins, K. L., Meltzer, M. S., and Nacy, C. A. 1994. Life and death of an intracellular pathogen: *Francisella tularansis* and the macrophage. Immunol Ser. 60:349-61.

Freeland, S. J., and Hurst, L. D. 1998. The genetic code is one in a million. J Mol

Evol. 47(3):238-48.

Gophna, U., Ron, E., and Graur, D. 2002. Bacterial type III secretion systems are ancient ad evolved by multiple horizontal-transfer events. Gene 312 pp 151-163.

Hanson, M. A., and Stevens, R. C. 2000. Cocrystal structure of synaptobrevin-II bound to botulinum neurotoxin type B at 2.0 A resolution. Nat Struct Biol. 7(8):697-92.

Hunter, John. 2007. Matplotlib. Wath Works. http://matplotlib.sourceforge.net/. Accessed 26 April 2007.

Kanehisa, M., Goto, S., Kawashima, S., Okono, Y., and Hattori, M. 2004. The KEGG resource for deciphering the genome. Nucleic Acids Res. 32(Database issue):D277-80.

KEGG: Kyoto Encyclopedia of Genes and Genomes. 2007. Kanehisa Laboratories. http://www.genome.jp/kegg/. Accessed 27 April 2007.

Klimpel, K. R., Arora, N., and Leppla, S. H. 1994. Anthrax toxin lethal factor contains a zinc metalloprotease consensus sequence which is required for lethal toxin activity. Mol Microbiol. 13(6):1093-100.

Korf, I., Yandell, M., and Bedell, J. 2003. BLAST. O'Reilly. Sebastopol, CA.

Kozel, T. R., Thorkildson, P., Brandt, S., Welch, W. H., Lovchik, J. A., AuCoin, D. P., Vilai, J., and Lyons, C. R. 2007. Protective and immunochemical activities of monoclonal antibodies reactive with the *Bacillus anthracis* polypeptide capsule. Infect Immun. 75(1):152-63.

Lan, R.T. and Reeves, P.R. 1996. Gene Transfer is a major factor in bacterial evolution. Mol. Microbiol. 15, 749-759.

Leppla, S. H. 1984. Bacillus anthracis calmodulin-dependent adenylate cyclase: chemical and enzymatic properties and interactions with eucaryotic cells. Adv Cyclic Nucleotide Protein Phosphorulation Res. 17:189-98

Lobry, J., and Sueoka, Noboru. 2002. Asymmetric directional mutation pressures in Bacteria. Genome Biology. 3:58(1-0058.14)

Marshall B. and Warren J.R. 2005. The Bacterium *Helicobacter pylori* and its role in Gastric and Peptic Ulcer Disease. Press Release (Nobel Prize). http://nobelprize.org/nobel_prizes/medicine/laureates/2005/press.html Accessed July 2006.

Ochman, H. and Wilson, A.C. 1987. Evolution in bacteria: evidence for a universal substitution rate in cellular genomes. J. Mol. Evol. 26, 74-86.

Ochman, H. and Lawrence, J.G. 1996 (1). Escherichia Coli and Salmonella: Cellular and Molecular Biology. Vol 2. pp. 2627-2637, ASM Press.1111

Ochman, H. et al. 1996 (2). Identification of a Pathogenicity Island Required for Salmonella Survival in Host Cells. Proc Natl Acad Sci USA 93(15): p 7800-4

Pace. 1997. Microbial Diversity and the Biosphere. Science.

Parsonnet, Friedman, et al. (1991 *Helicobacter pylori* infection and the risk of gastric carconoma. N Eng J of Med. Volume 325: 1127-1131.

Peterson WL. (1991) Helicobacter pylori and peptic ulcer disease. N Eng J of Med. 324:1043-1048.

Petosa, C., Collier, R. J., Klimpel, K. R., Leppla, S. H., and Liddington, R. C. 1997. Crystal structure of the anthrax toxin protective antigen. Nature. 385(6619):833-8

Porwollik, S, and McClelland, M. 2003. Lateral gene transfer in Salmonella. Microbes and Infections.

Sequence Composition Analyzer (SeqComp). City of Hope. Dept of Molecular Genetics and Div of Molecular Medicine. http://cityofhope.org/molgen/seqcomp.asp. Accessed 1 May 2007.

Sharp, P.M. 1991. Determinants of DNA sequence divergence between Escherichia coli and Salmonella typhimurium: codon usage, map position, and concerted evolution. J. Mol. Evol. 33, 23-33

Simpson, L. L., Maksymowych, A. B., and Hao, S. 2001. The role of zinc binding in the biological activity of botulinum toxin. J Biol Chem. 276(29):27034-41.

Singh, B. R. 2000. Intimate details of the most poisonous poison. Nat Struct Biol. 7(8)617-9.

Sueoka, Noboru. 2002. Wide intra-genomic G+C heterogeneity in human and chicken is mainly due to strand-symmetric directional mutation pressures: dGTP-oxidation and symmetric cytosine-deamination hypotheses. Gene.

U.S. Food and Drug Administration: Foodborne Pathogenic Microorganisms and Natural Toxins Handbook. 2006. *Clostridium botulinum.* http://www.cfsan.fda.gov/~mow/chap2.html. Accessed 1 May 2007.

Uliel, S., Fliess, A., Amir, A., and Unger, R. 1999. A simple algorithm for detecting circular permutations in proteins. Bioinformatics. Vol. 15 no. 11

Varughese, M., Teixeira, A. V., Lui, S., and Leppla, S. H. 1999. Identification of a receptor-binding region within domain 4 of the protective antigen component of anthrax toxin. Infect Immun. 67(4):1860-5

Vitale, G., Bernardi, L., Napolitani, G., Mock, M., and Montecucco, C. 2000. Susceptibility of mitogen-activated protein kinase kinase family mambers to proteolysis by anthrax lethal factor. Biochem J. Pt 3:739-45.

Watson, J. D., and Crick, F. H. 2003. Molecular structure of nucleic acids. A structure for deoxyribose nucleic acid. Rev Invest Clin. 55(2):108-9.

Weaver, Robert. 2005. Molecular Biology. McGraw Hill. Boston.

Welch, R. A., Burland V., Plunkett, G., Redford, P., Roesch, P., Rasko, D., Buckles, E. L., Liou, S. R., Boutin, A., Hackett, J., Stroud, D., Mayhew, G. F., Rose, D. J., Zhou, S., Schwartz, D. C., Perna, N. T., Mobley, H. L., Donnenberg, M. S., Blattner, F.R. 2002. Extensive mosaic structure revealed by the complete genome sequence of uropathogenic Escherichia coli. Proc Natl Acad Sci U S A. 99(26):17020-4.

Witkowski, J. A., and Parish, L. C. 2002. The story of anthrax from antiquity to the present: a biological weapon of nature and humans. Clin Dermatol. 20(4): 336-42.

Zubay, et al. (2005) Agents of Bioterrorism: Pathogens and their Weaponization. Columbia Press. NY