*kodonz*: A Comprehensive C**odon** Analysis Toolbox

2/9/2019

Can **Z**hou

Texas A&M University, College Station, TX, USA, 77840

Email: [eidotog@gmail.com](mailto:eidotog@gmail.com)

Lingming **K**ong

Shandong University, Weihai, Shandong, China

Email: [xx@xx.edu](mailto:xx@xx.edu)

# Introduction

It is well known that genetic code has redundancy and most of the amino acids can be translated from multiple codons. The synonymous codons encode the same genetic information, but provide additional information on the efficiency of protein production (Grantham, Gautier, Gouy, Mercier, & Pave, 1980).

Codon analysis specifically examines the usage bias of these synonymous codons. Thus, the subject of codon analysis should be independent of the ordering of the genetic information on a DNA sequence (Figure 1). It should be clear from Figure 1 that a complete approach towards the analysis of the production and function of a protein involves both synonymous codon usage bias (SCUB) and amino acid sequence (AAS) information. The complete approach would be a much larger project and requires more resources than is available to the authors, and therefore, we restrict the scope of this package is to only the direct effects of SCUB. In fact, most of the codon analysis are invariant under permutation transformations of codons, and it is advised to remove the influence of other aspects of the DNA information, for example, removing non-degenerate codons from compositional analysis, such that the calculated statistic is not subject to changes in amino acid composition. Meanwhile, there are other methods that partially depend on the ordering of codons, for example, the %MaxMin method (Clarke IV & Clark, 2008), which examines the indirect effect of SCUB on protein production through AAS.

*kodonz* is a comprehensive toolbox to simplify codon analysis procedures, including compositional analysis, relative synonymous codon usage (RSCU) analysis, codon adaptation index (CAI) analysis, effective number of codons (ENC) analysis, %MaxMin analysis and many more. In the following, we present a demonstration of the capabilities of the *kodonz* package using an example DNA dataset.

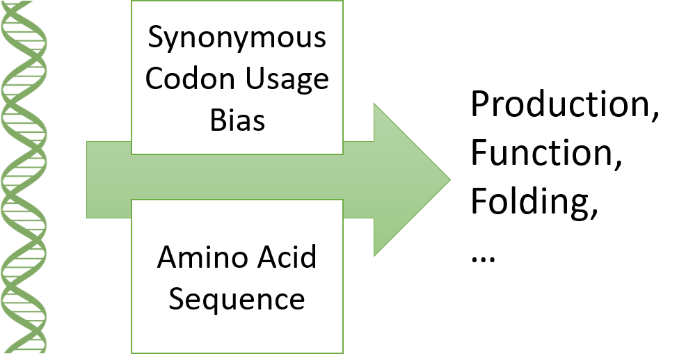


Figure 1 Genetic sequence of a protein encodes two types of information: synonymous codon usage bias (SCUB) and amino acid sequence (AAS), which jointly determine the production and function of the protein.

# Importing the example DNA dataset

There are multiple way to import a DNA sequence into the workspace. Here, ‘KZsqns’ is a class of *kodonz* package for DNA codon sequences, and it is automatically created when the user import sequences using the “load.fasta” method. This import method depends on “read.fasta” from package *seqinr* to interact with external files, checks the sequence for errors, and removes ambiguous bases, such as symbol “N” or “-”, which may appear in a fasta file. Alternatively, the user may directly provide a character array to “load.fasta” for error checking:

## Start of R code 1 ##

ex1=load.fasta(file = " dna.fasta")

dna2 = c("TTTGWXATG", "GWXATTXNNTTTATA")

s2 = load.fasta(x=dna2)

# Warning messages:

# 1: In load.fasta(x = dna2) :

# Sequence1 contains ambiguous base(s), which will be removed from further analysis.

#

# 2: In load.fasta(x = dna2) :

# Sequence2 contains ambiguous base(s), which will be removed from further analysis.

s2

# [[1]]

# [1] "TTT" "ATG"

# attr(,"class")

# [1] "KZsqns"

#

# [[2]]

# [1] "ATT" "TTT" "ATA"

# attr(,"class")

# [1] "KZsqns"

## End of R code ##

In this example, codons with ambiguous nucleotide or any other symbol other than “A”, “T”, “G” and “C” were removed from the “KZsqns” objects. Further, an error will be raised when the number of nucleotides of a sequence is not a multiple of three. However, “load.fasta” only checks some simple types of errors in the sequence data, and the user should do additional checks for errors, and this task is outside the task of *kodonz* as of now. A third way to import data to *kodonz* is to bypass “load.fasta” and create “KZsqns” objects through a loop like in R code 2.

# Compositional analysis

The following demonstrations will be based on a simulated DNA dataset with alternative yeast nuclear codon table (#12). For both nuclear and mitochondrial DNA codons, “A”, “T”, “G” and “C” symbols are used with their usual meaning in genetics in this package to denote the nucleotide composition in the sequence. A list of DNA codon sequences is generated based on CodonTable12,

## Start of R code 2 ##

# All the code present in this tutorial will be continuously numbered, and they will be referenced in the form of “R code #” in the text when needed.

# The code section has a different formatting, such that code can be easily distinguished from normal text, and easily copy-and-pasted from this tutorial.

# When you copy and paste code into your R environment, pay attention to the quotation marks as they may change depending on your text editor, and they need to be replaced with the correct quotation marks.

data(CodonTable5) # Alternative yeast nuclear codon table

x = vector(‘list’, 100) # Creating an empty list

set.seed(7) # For those who crave sanity checks

for(i in 1:100){

x[[i]] = CodonTable12[sample(1:64, 50\*i, TRUE),1]

attr(x[[i]], ‘class’) = ‘KZsqns’

}

## End of R code ##

Composition analysis is easy with *kodonz*. Just one line of code can calculate 13 compositional properties,

## Start of R code 3 ##

d = comp\_ly(x)

head(d)

# A C G T A3 C3 G3 T3 AT GC GC1 GC2

# s\_1 0.1866667 0.2733333 0.3133333 0.2266667 0.2000000 0.2600000 0.3000000 0.2400000 0.4133333 0.5866667 0.540 0.6600000

# s\_2 0.2966667 0.2233333 0.2266667 0.2533333 0.2600000 0.2000000 0.2200000 0.3200000 0.5500000 0.4500000 0.450 0.4800000

# s\_3 0.2555556 0.2488889 0.2733333 0.2222222 0.2266667 0.2466667 0.3200000 0.2066667 0.4777778 0.5222222 0.480 0.5200000

# s\_4 0.2300000 0.2483333 0.2566667 0.2650000 0.2250000 0.2050000 0.3150000 0.2550000 0.4950000 0.5050000 0.420 0.5750000

# s\_5 0.2373333 0.2466667 0.2653333 0.2506667 0.2400000 0.2640000 0.2760000 0.2200000 0.4880000 0.5120000 0.464 0.5320000

# s\_6 0.2422222 0.2533333 0.2622222 0.2422222 0.2566667 0.2366667 0.2566667 0.2500000 0.4844444 0.5155556 0.550 0.5033333

# GC3

# s\_1 0.5600000

# s\_2 0.4200000

# s\_3 0.5666667

# s\_4 0.5200000

# s\_5 0.5400000

# s\_6 0.4933333

## End of R code ##

Note that in R code 3, the column names of the result in Word are misaligned with their corresponding columns. For the result, each row represents a sequence, and they are named consecutively as ‘s\_1’, ‘s\_2’ and so on with respect to their order in the input; each column represents a compositional property. For left to right, these properties are respectively overall “A”, “C”, “G” and “T” compositions, “A”, “C”, “G” and “T” at the 3rd codon position, overall “A” and “T” content, overall “G” and “C” content, “G” and “C” content at the 1st, 2nd and 3rd codon positions. Method “comp\_ly” is a wrapper function to conveniently pack all 13 properties of a sequence into one single vector, and as we’ll see later, this greatly facilitates subsequent multivariate analyses, for example, PCA. However, if the user is only interested in calculating some of the properties, specific methods are also available (See the package manual for specific methods).

In the preceding analysis, we passed in the entire codon sequence, including stop codons and non-degenerate amino acid coding codons when they exist. While stop codons may be included in the compositional analysis when multiple stop codons exist, it is generally advised to exclude non-degenerate codons from the analysis, because the inclusion of these codons would mask any codon preference information present in the data due to differences in amino acid composition among sequences. For the standard codon table, codons “ATG” and “TGG”, which only encodes Met(M) and Trp(W), are non-degenerate; for other codon tables, non-degenerate codons can be different. To exclude non-degenerate codons, we need to additionally pass in the codon table and a helper method,

## Start of R code 4 ##

d\_1 = comp\_ly(rm\_ndc(x,12,T))

# A C G T A3 C3 G3 T3 AT GC GC1

# s\_1 0.1836735 0.2721088 0.3129252 0.2312925 0.1836735 0.2653061 0.3061224 0.2448980 0.4149660 0.5850340 0.5306122

# s\_2 0.2955326 0.2302405 0.2233677 0.2508591 0.2680412 0.2061856 0.2268041 0.2989691 0.5463918 0.4536082 0.4639175

# s\_3 0.2534247 0.2557078 0.2762557 0.2146119 0.2191781 0.2534247 0.3287671 0.1986301 0.4680365 0.5319635 0.4931507

# s\_4 0.2321117 0.2530541 0.2600349 0.2547993 0.2356021 0.2146597 0.3298429 0.2198953 0.4869110 0.5130890 0.4397906

# s\_5 0.2316384 0.2514124 0.2810734 0.2358757 0.2245763 0.2796610 0.2923729 0.2033898 0.4675141 0.5324859 0.4915254

# s\_6 0.2422145 0.2468281 0.2722030 0.2387543 0.2525952 0.2214533 0.2664360 0.2595156 0.4809689 0.5190311 0.5709343

# GC2 GC3

# s\_1 0.6530612 0.5714286

# s\_2 0.4639175 0.4329897

# s\_3 0.5205479 0.5821918

# s\_4 0.5549738 0.5445026

# s\_5 0.5338983 0.5720339

# s\_6 0.4982699 0.4878893

## End of R code ##

Here, we are using the alternative yeast nuclear codon table, which has the same set of non-degenerate codons as the standard codon table, and removed non-degenerate codons while keeping the stop codons if they are degenerate. We can compare the printed results of R code 3 with those from R code 4, and see some small differences between the two. Note that for some codon table, all the codons are degenerate and two results would be the same, for example, the invertebrate mitochondrial codon table (#5) have no non-degenerate codons; the inner helper method call, in addition to removal of non-degenerate codons, also removes the ordering of the codons of a DNA sequence, and it should be used with caution when it is not used in a compositional context, where codon ordering is insignificant.

# Relative synonymous codon usage (RSCU)

RSCU index is widely used to measure synonymous codon usage bias in coding sequences (Sharp & Li, 1986). Each codon has a RSCU index ranging from 0, which denotes complete omission, to the number of synonymous codons for that amino acid, which indicates complete preference. A value of one for all the synonymous codons of an amino acid means no codon usage bias for that amino acid, and codons for that amino acid are uniformly chosen. The  for the *j*-th codon of amino acid *i* is calculated as

,

where  denotes the number of occurrences of codon *j*,  denotes the degeneracy of that amino acid. The value of one can be used to separate positive preference and negative preference, and some authors have developed some range of values, for example from 0.6 to 1.6 for this purpose (Wong, Smith, Rabadan, Peiris, & Poon, 2010). In the following, we will show you how to calculate a upper and lower boundary given a confidence level for your own dataset.

For our simulated dataset, the RSCUs can be easily obtained,

## Start of R code 5 ##

S1 = lapply(rm\_ndc(x, 12), rscu)

# This result consists of 100 64-vectors, and it is not shown here.

# There will be 7 warnings for sequence 1, because 7 amino acids are missing from the sequence due to the short length of that sequence, and those missing codons are filled with NAs

# Simulated data y

y = vector('list', 100) # Creating an empty list

for(i in 1:100){

y[[i]] = CodonTable12[sample(1:64, 5000, TRUE),1]

attr(y[[i]], 'class') = 'KZsqns'

}

## End of R code ##

I have plotted the RSCUs for the last sequence, i.e., x[[100]], in Figure 1. The distribution of the RSCUs are around the value of no bias (1). Note that the distribution in Figure 2 does not look normal, it is composed of the RSCUs of 62 codons (non-degenerate codons are removed), and codons with different degeneracy have a different range of variation from 0 to .

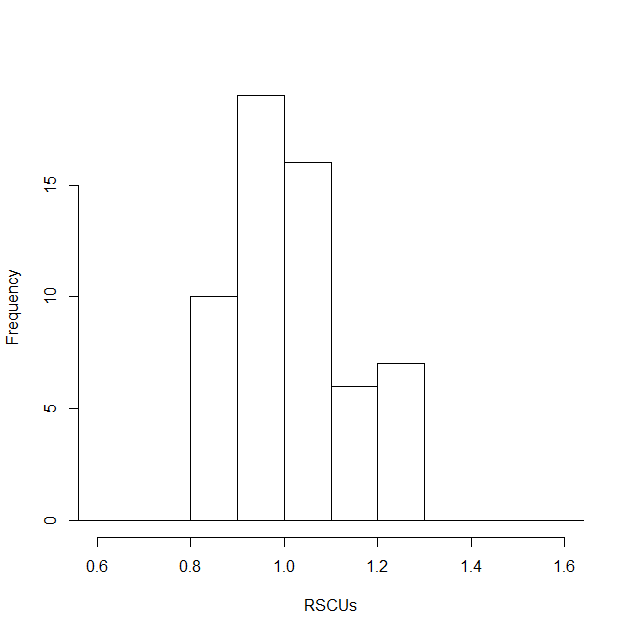


Figure 2 The distribution of RSCUs for the simulated sequence x[[100]]

On the other hand, the distribution for a single codon under the same mutation and selection pressure would have a more normal distributed look. With uniform random sampling, the distribution of RSCU of codon “GGA” is shown in Figure 3. “GGA” has a degeneracy of 4 with codon table #12, it is arbitrarily chosen in this example, and it could have been any other degenerate codons. Now the distribution in Figure 3 has a much more normal look than that in Figure 2. All of the samples used in Figure 3 are i.i.d. (independently identically distributed), but the same can not be said about Figure 2.

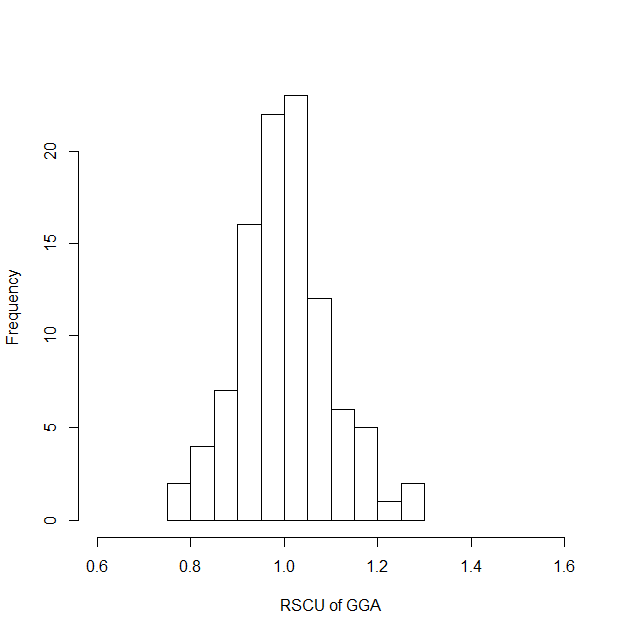


Figure 3 Distribution of the RSCU of GGA of sample size 5000 under the uniform random sampling assumption

The preceding demonstration illustrates another important usage of the *kodonz* package, i.e., to generate small sample critical values of test statistics. For example, suppose we have a 1984-codon long sequence with a calculated RSCU of 1.4 for codon “GGA” based on codon table #12, and we want to know if the selection of codon “GGA” is biased at the 95% confidence interval. The task then is to calculate a small sample distribution of the RSCU for “GGA” under the uniform random null hypothesis,

## Start of R code 6 ##

z = vector('list', 5000)

for(i in 1:5000){

z[[i]] = CodonTable12[sample(1:64, 1984, TRUE),1]

attr(z[[i]], 'class') = "KZsqns"

}

s3 = lapply(rm\_ndc(z, 12), rscu)

t1 = sapply(s3, function(x){x["GGA"]})

quantile(t1, probs = c(0.025, 0.5, 0.975)) # 95% critical values

# 2.5% 50% 97.5%

# 0.7022767 0.9927007 1.3114885

quantile(t1, probs = c(0.005, 0.995)) # 99% critical values

# 0.5% 99.5%

# 0.6031746 1.4339680

## End of R code ##

The user can play with the number of replicas used in the simulation and observe the variation in the generate critical values. For this example, a 5000 replication is sufficient. From the result, we can see that the test statistic 1.4 is significant for the two-sided test at the 95% confidence level, but not significant at the two-sided 99% level. On the other hand, the user may choose to do a less stringent one-sided test, if the user is interested in whether the specific code is being either biased positively or negatively.

# Codon adaptation index (CAI) analysis

CAI is a method to predict the expression level of a coding DNA sequence based on the SCUB of a set of reference coding sequences (Sharp & Li, 1987). It is assumed that a sequence with a similar SCUB with that of a set of highly expressed genes would also have a high expression level. Three build-in reference tables are based on Shields and Sharp (1987) and Sharp and Li (1987). For other species of interest, the user can calculate and supplied a custom reference table. The relative adaptiveness  for each amino acid coding codon *i* can be calculated from a set of highly expressed genes as follows,

,

where *i* and *j* are synonymous codons for an amino acid,  is the observed frequency of codon *i* in the reference set. Stop codons and non-degenerate codons are excluded. The relative adaptiveness has a natural range from 0 to 1, where 0 denotes extremely negative and positive bias. To prevent potentially zeroing out the CAI index of a sequence that included some codon which are absent from the reference set, two approaches has been proposed: 1) in calculating the adaptiveness of an absent codon, use a frequency of 0.5 instead (Sharp & Li, 1987); 2) adjust the adaptiveness of an absent codon to 0.01 (Bulmer, 1988). In the following, I calculate a reference adaptiveness table based on a simulated dataset,

## Start of R code 7 ##

z = vector('list', 10) # Creating an empty list

for(i in 1:10){

z[[i]] = CodonTable12[sample(1:64, 50+20\*i, TRUE),1]

attr(z[[i]], 'class') = 'KZsqns'

}

# For our test, codon “TGC” is missing from the reference set

t2 = cai\_crt(z, 12, "sharp")

# the adaptiveness score of “TGC” is 0.01923, and it is the smallest of the whole set

t3 = cai\_crt(z, 12, "bulmer")

# the adaptiveness score of “TGC” is 0.01, and it is the smallest of the whole set as well

## End of R code ##

In this example, both correcting approaches agree with each other, but it is not guaranteed and in some extreme cases, the results can be very different between these two approaches, and the user’s judgement is required to decide which approach to use. In the authors’ opinion, it is always a good idea to add more genes to the reference set, as more genes are added, the undesired behavior is less likely to occur. With this custom reference set, we can calculate CAIs for other sequences:

## Start of R code 8 ##

x = vector('list', 1000) # Creating an empty list

set.seed(7)

for(i in 1:1000){

x[[i]] = CodonTable12[sample(1:64, 50\*i, TRUE),1]

attr(x[[i]], 'class') = 'KZsqns'

}

t4 = lapply(x, cai, y=12, ref=t2)

t5 = lapply(x, cai, y=12, ref=t3)

t4=unlist(t4)

t5=unlist(t5)

par(mfrow=c(1,2))

plot(t4, col='red', type='l', ylab='CAI')

lines(1:1000, t5, col='blue')

legend(500, 0.74, col=c('red', 'blue'), legend=c('Sharp', 'Bulmer'), lty = c(1,1))

plot(t4, t5, xlab='Sharp', ylab='Bulmer')

## End of R code ##

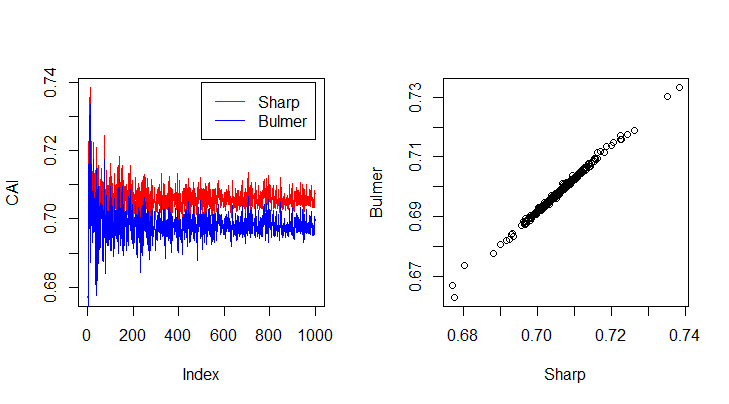


Figure 4 CAI of a simulated set of sequences with increasing length based on custom reference tables with either Sharp type or Bulmer type adjustments

As expected, the CAI based on Sharp correction tends to be larger than the CAI based on Bulmer correction (Figure 4 left panel), because we saw that in R code 7 the adaptiveness value of “TGC” is larger than that from the Bulmer method. But this relationship is not generally true. For small sequences with less than 20k codons, the variability of CAI is due to both the small sample variance and the inherent variability of CAI; for longer sequences, the CAI stabilizes with a narrow band (Figure 4 left panel). The right panel of Figure 4 confirms the strong correlation between two CAIs.

# Effective number of codons (ENC)

ENC is another way to measure SCUB, and it is independent of sequence length and amino acid composition (Wright, 1990). Many such measures have been proposed over the years (Fuglsang, 2004), and two of them is included in *kodonz*: Wright’s formula and Fuglsang’s Ncf4- formula. For Wright’s formula, the expected ENC given GC3 is,

,

where *s* denotes the GC3 content. Note that this formula is only applicable to the standard codon table, and it is an approximation. A more direct method would be to calculate ENC from randomly generated sequences based on the null hypothesis, and compare the calculated ENC from the sample sequence with the simulated ENCs.

## Start of R code 9 ##

t6 = enc(x)

t7 = gc3(x)

gc3 = 1:100/100

t8 = sapply(gc3, enc\_exp)

par(mfrow=c(1,2))

plot(t6, type='l', ylab='ENC')

plot(gc3, t8, type='l', xlab="GC3", ylab="ENC", ylim=c(55, 66), xlim=c(0.3, 0.7))

points(t7,t6)

## End of R code ##

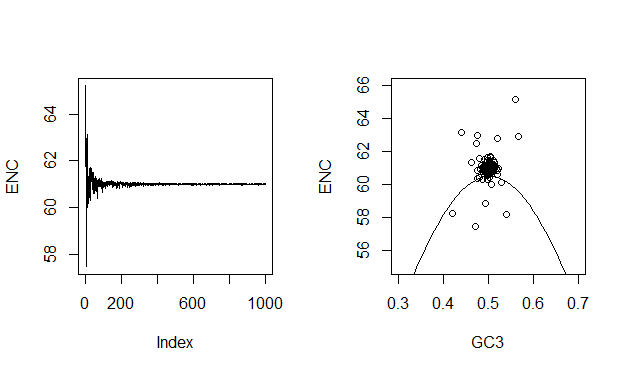


Figure ENC for a list of uniform random generated sequences with increasing length and a scatter plot of calculated ENC vs GC3 and the expected ENC based on GC3

Sequences used in this example were generated using a uniformly, such that there is no codon bias in the sample sequences, and the standard codon table was used in calculating ENC. When the sequence is long enough (more than 10k codons or at index 200), ENC stabilized at value 61. This is expected since there are 3 stop codons in the standard codon table and the coding codons are used without any bias in the generated sequence. The expected ENC based on the formula is 60.5 at  however, emphasizing the fact that the formula is only an approximation. A Monte Carlo simulation method would be more appropriate, and we will further explore that idea in the further usage section.

# %MinMax of a coding sequence

Compared with all other analysis included in this package, %MinMax is special in that ordering is significant. %MinMax calculates the local relative usage frequencies of synonymous codons for a protein coding sequence (Rodriguez, Wright, Emrich, & Clark, 2018). This method calculates a SCUB statistic based on a host reference codon usage table with a sliding window. Sixteen reference codon usage tables were build-in to the *pminmax* method. Additionally, the user may supply a custom reference codon usage table on top of those sixteen. *pminmax* is simple to use.

## Start of R code 10 ##

t8 = pminmax(x[[100]], z=20, spp='pig')

plot(t8, ylab="%MinMax", type='l')

## End of R code ##

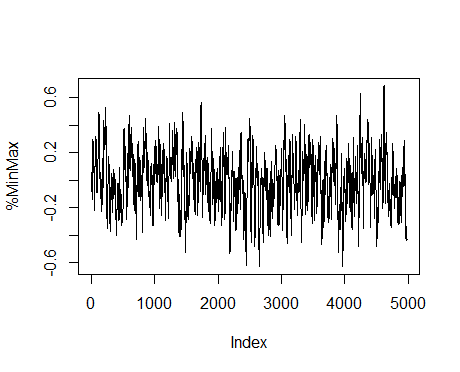


Figure %MinMax of a uniform random generated sequence of 5k codons

The plot of %MinMax index of the example sequence looks random, and I don’t see obvious clusters of either rare codons or common codons (Figure 6). But eyes can be trusted most of the time, and we need a more direct way to judge whether there are rare or common codon clusters. To do that we just need to 1) generate a bunch of sequences, in this case 1000 of them, according to the null hypothesis, which says that there is no difference than the reference set, which is the pig codon usage set, 2) calculate the statistic of interest from those generate data, and 3) calculate the quantiles of interest from the distribution. R code 11 does just that.

## Start of R code 11 ##

plot(t8, ylab="%MinMax", type='l') # This is a continuation from R code 10

sim = vector('list', 500)

probs = cft\_pig[,5]

names(probs) = cft\_pig[,1]

for(i in 1:500){

sim[[i]] = CodonTable0[sample(1:64, 5e3, T, probs[CodonTable0[,1]]),1]

attr(sim[[i]], 'class') <- 'KZsqns'

}

mmsim = sapply(sim, pminmax, z=20, spp='pig') # may take a few minuites

cint = apply(mmsim, 1, quantile, probs=c(0.025, 0.5, 0.975))

gray = rgb(0,0,0,0.2)

polygon(c(1:4981,4981:1),c(cint[1,],rev(cint[3,])), border = NA, col=gray)

lines(1:4981, cint[2,], col='red')

## End of R code ##

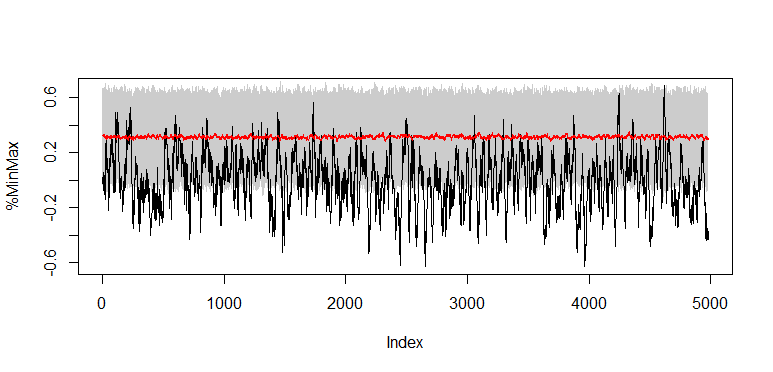


Figure %MinMax of a uniform random generated sequence of 5k codons with simulated 95% confidence band (gray) and median (red)

Now, it is clear! We made a mistake in wrongly identifying high frequency variation of %MinMax value with no clustering of either rare or common codons. In fact, the pig reference set has a median %MinMax value of around 0.3, and the 95% confidence interval covers from -0.05 to 0.66 based on simulation (Figure 7), and distribution of test statistic is not always centered around 0! Now, we see multiple clusters of rare codons (Figure 7), but we will not pursue this further since the sample sequence is simulated and not real. But the analysis of a real sequence would follow the same steps as shown above. In fact, the user may try different sizes of the sliding window in order to get the best result.

# Correlational analysis

Various correlational analyses are possible, once the important SCUB indices were calculated. In the following table, I have listed several correlation analyses. It should be noted that when making this list, I am just demonstrating the additional things that the user can do with the SCUB indices, and I am not trying to be exhaustive here. There might be important correlational analyses that are left out here. Let me know if I am making such a mistake.

Table 1 Correlation analyses and useful kodonz methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Index | X | Y | Genetic significance | Useful *kodonz* methods |
| 1 | CAI | ENC | Determine the relative influence of mutation and selection | cai, enc, cai\_crt |
| 2 | GC12 | GC3 | Neutral evolution analysis | gc3, gcx, comp\_ly |
| 3 | A3/(A3+U3) | G3/(G3+C3) | Parity rule analysis | n3\_freq, comp\_ly |

# Multivariate analysis

Again, once the important SCUB indices were calculated, it is fairly easy to do multivariate analysis, such as classification methods and ordination methods. Here, I demonstrate the use of PCA, which is a popular method to reduce the dimension of the data and capture the major variations in the dataset. In R code 12, I simulate 5 datasets based on the reference codon usage of human, pigs, mouse, rat and insect, and use PCA to see if there are major differences among the datasets.

## Start of R code 12 ##

n = 100

l = 5000

human = vector('list', n)

insect = vector('list', n)

rat = vector('list', n)

mouse = vector('list', n)

pig = vector('list', n)

h\_ref = cft\_human[,5]

names(h\_ref) = cft\_human[,1]

i\_ref = cft\_insect[,5]

names(i\_ref) = cft\_insect[,1]

r\_ref = cft\_rat[,5]

names(r\_ref) = cft\_rat[,1]

m\_ref = cft\_mouse[,5]

names(m\_ref) = cft\_mouse[,1]

p\_ref = cft\_pig[,5]

names(p\_ref) = cft\_pig[,1]

for (i in 1:n){

human[[i]] = CodonTable0[sample(1:64, size = l, replace = T, prob = h\_ref[CodonTable0[,1]]),1]

insect[[i]] = CodonTable0[sample(1:64, size = l, replace = T, prob = i\_ref[CodonTable0[,1]]),1]

rat[[i]] = CodonTable0[sample(1:64, size = l, replace = T, prob = r\_ref[CodonTable0[,1]]),1]

mouse[[i]] = CodonTable0[sample(1:64, size = l, replace = T, prob = m\_ref[CodonTable0[,1]]),1]

pig[[i]] = CodonTable0[sample(1:64, size = l, replace = T, prob = p\_ref[CodonTable0[,1]]),1]

attr(human[[i]], 'class') = 'KZsqns'

attr(insect[[i]], 'class') = 'KZsqns'

attr(rat[[i]], 'class') = 'KZsqns'

attr(mouse[[i]], 'class') = 'KZsqns'

attr(pig[[i]], 'class') = 'KZsqns'

}

s1 = list(human, insect, rat, mouse, pig)

t52 = unlist(lapply(s1, function(x){lapply(rm\_ndc(x), rscu)}))

x =matrix(t52, ncol=64, nrow=5\*n, byrow = T)

colnames(x) <- CodonTable0[,1]

y = c(sapply(1:5, rep, n))

ex = which(apply(x, 2, var)==0 | is.na(apply(x, 2, var)) )

pca = prcomp(x[,-ex], scale.=TRUE)

plot(pca$x [,1], pca$x [,2], col=c('red', 'blue', 'green', 'purple', 'black')[y], xlab='PCI1', ylab='PCI2')

## End of R code 12 ##

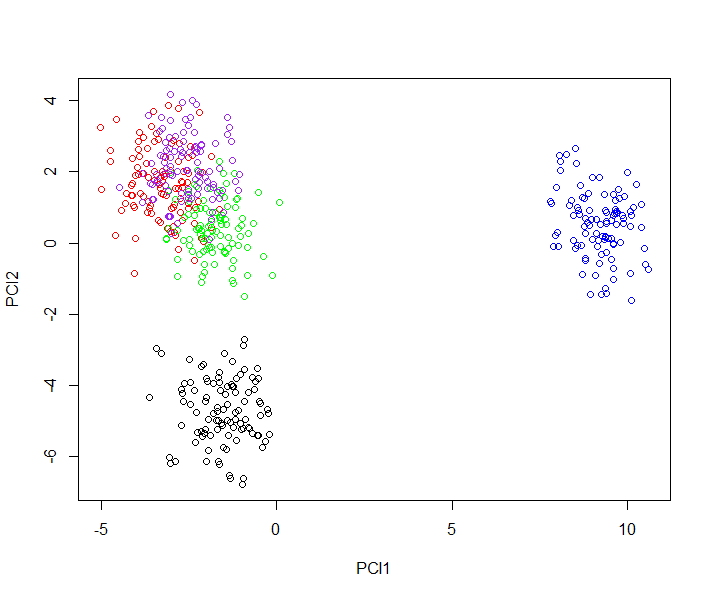


Figure Simulated sequences based on reference codon usage data of human (red), insect (blue), rat (green), mouse (purple) and pig (black).

PCI1 and PCI2 did a pretty good job of separating out three major blobs of sequences (Figure 8). PCI1 clearly separates insect (blue) from the rest of the species, and this is expected since these two groups of species have the longest taxonomical distance. Then PCI2 separates pig (black) from the rest of the species, and there is some overlap between rat (green) and the rest two species, i.e., human (red) and mouse (purple), which are almost not separable based on the first two PCIs. More PCIs are needed to satisfactorily separate sequences of human and mouse in this case.

# Further usage

Monte Carlo simulations can be used to generate the distribution of test statistics under the null hypothesis, and this distribution can then be used to test the significance of the sample test statistics. This idea has been explored throughout this tutorial, when we calculated RSCU, ENC and %MinMax indices. This approach can be used when the large sample distribution of the test statistics is not known, or the sample size is too small. This approach is both versatile and easy to carry out. Through out this tutorial, I have used two null hypotheses: 1) uniform random distribution over all 64 codons, and 2) distributed according to a reference set. In the following, I explore another null hypothesis on ENC, where the GC­3 is fixed at a preset value, say 0.2, or any other value between 0 and 1. Since we are taking a numerical approach, the specific value does not matter that much. Also, since the hypothesis didn’t mention any other bias other than the GC3 content, it is fair to assume no other bias in choose a nucleotide for a codon. So for the 1st and the 2nd position of the codon, there is no bias, and four bases have the same probability of occurring; for the 3rd position, “A” and “T” each has a 40% chance of occurring, and “G” and “C” each has a 10% chance. In R code 13, we carry out the analysis.

## Start of R code 13 ##

bases = strsplit("ATGC", split = "")[[1]]

h1 <- function(n){

return(paste0(bases[sample(1:4,1)], bases[sample(1:4,1)], bases[sample(1:4, 1, prob = c(0.4, 0.4, 0.1, 0.1))]))

}

s2 = vector('list', 500)

for(i in 1:500){

s2[[i]] = sapply(1:5000, h1)

attr(s2[[i]], "class") = "KZsqns"

}

t34 = enc(x = s2, y = 0, formula = 'w')

hist(t34, main='', xlab='ENC')

## End of R code ##

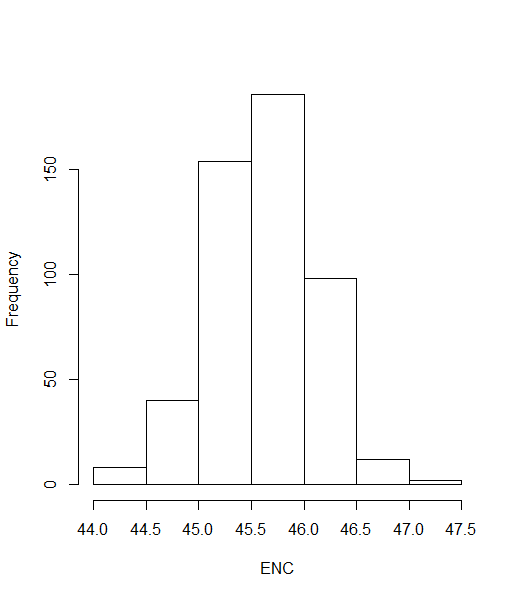


Figure Distribution of simulated ENC with GC3=0.2

The simulated ENC is centered around 45.6, which is slightly larger than the expected ENC based on the formula (R code 9), and its lower and upper 95% quantiles are 44.7 and 46.5. The expected ENC is 44.8, which lies at the lower boundary of the 95% quantile. 5000 nucleotide long sequences were used in the simulation (R code 13), and as longer sequences are used, we would expect the distribution to get thinner and thinner, and the expected ENC based on the formula to fall out of the lower bound of the 95% quantile. We suspect the expected ENC formula has a negative bias, and recommend to use Monte Carlo simulations as demonstrated above to calculate the expected ENC under the null hypothesis.

Bulmer, M. (1988). Are codon usage patterns in unicellular organisms determined by selection‐mutation balance? *Journal of Evolutionary Biology, 1*(1), 15-26.

Clarke IV, T. F., & Clark, P. L. (2008). Rare codons cluster. *PLoS ONE, 3*(10), e3412.

Fuglsang, A. (2004). The ‘effective number of codons’ revisited. *Biochemical and biophysical research communications, 317*(3), 957-964.

Grantham, R., Gautier, C., Gouy, M., Mercier, R., & Pave, A. (1980). Codon catalog usage and the genome hypothesis. *Nucleic acids research, 8*(1), 197-197.

Rodriguez, A., Wright, G., Emrich, S., & Clark, P. L. (2018). % MinMax: A versatile tool for calculating and comparing synonymous codon usage and its impact on protein folding. *Protein Science, 27*(1), 356-362.

Sharp, P. M., & Li, W.-H. (1986). Codon usage in regulatory genes in Escherichia coli does not reflect selection for ‘rare’codons. *Nucleic acids research, 14*(19), 7737-7749.

Sharp, P. M., & Li, W.-H. (1987). The codon adaptation index-a measure of directional synonymous codon usage bias, and its potential applications. *Nucleic acids research, 15*(3), 1281-1295.

Shields, D. C., & Sharp, P. M. (1987). Synonymous codon usage in Bacillus subtilis reflects both translational selection and mutational biases. *Nucleic acids research, 15*(19), 8023-8040.

Wong, E. H., Smith, D. K., Rabadan, R., Peiris, M., & Poon, L. L. (2010). Codon usage bias and the evolution of influenza A viruses. Codon Usage Biases of Influenza Virus. *BMC evolutionary biology, 10*(1), 253.

Wright, F. (1990). The ‘effective number of codons’ used in a gene. *Gene, 87*(1), 23-29.