A Framework and Application for Efficient Analysis of Peptide Libraries

Eric Riemer Hare

January 7, 2014

My creative component consists of three separate papers, each covering a different component of the overall project I worked on. The layering of the three components is illustrated in Figure 1. The structure of this document is organized in the same manner, beginning with discreteRV, continuing with peptider, and ending with PeLiCa.

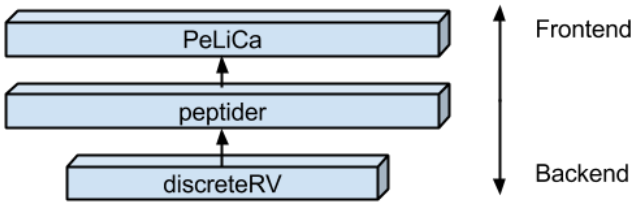


Figure 1: The three components, or layers, of my project.

Manipulation of Discrete Random Variables with discreteRV

Eric Riemer Hare

January 6, 2014

1 Introduction

Dr. Andreas Buja, professor of Statistics at the University of Pennsylvania, created a set of R functions implementing discrete random variables[REF]. These functions were compiled and documented in a package called discreteRV. discreteRV is available for download on the Comprehensive R Archive Network (CRAN)[REF].

The functions of discreteRV are organized into two logical areas, termed probabilities and simulations.

2 Probabilities

discreteRV includes a suite of functions to create, manipulate, and compute distributional quantities for the random variables defined. A list of these functions and brief discriptions of their functionality is available in Table 1.

Name	Description
as.RV	Turn a probability vector with possible outcome values in the names() attribute into a random variable
E	Expected value of a random variable
KURT	Kurtosis of a random variable
make.RV	Make a random variable consisting of possible outcome values and their probabilities or odds
margins	Marginal distribution of a joint random variable
mult	Joint probability mass function of random variables X and Y
multN	Probability mass function of X^n
P	Calculate probabilities of events
plot.RV	Plot a random variable of class RV
print.RV	Print a random variable of class RV
probs	Probability mass function of random variable X
qqnorm.RV	Normal quantile plot for RVs to answer the question how close to normal it is
SD	Standard deviation of a random variable
SKEW	Skewness of a random variable
SofI	Sum of independent random variables
SofIID	Sum of independent identically distributed random variables
V	Variance of a random variable

Table 1: List of the probability functions contained in discreteRV.

2.1 Creation

The centerpiece of discreteRV is a set of functions to create and manipulate discrete random variables. A random variable maps a set of possible outcomes to a set of probabilities that sum to one. In discreteRV, a random variable is defined through the use of the make.RV function. make.RV accepts a vector of probabilities and a vector of outcome values, and returns an RV object.

```
make.RV(vals = 1:6, probs = rep("1/6", times = 6))


## random variable with 6 outcomes
##
##   1   2   3   4   5   6
## 1/6 1/6 1/6 1/6 1/6 1/6
```

2.2 Structure

The syntactic structure of the included functions lends itself both to a natural presentation in an introductory probability course, as well as more advanced modeling of discrete random variables. The object is constructed by setting a standard R vector object to the possible values that the random variable can take (the sample space). It is preferred, though not required, that these be encoded as integers, since this allows for expected values, variances, and other measures of distributional tendency to be computed. This vector of outcomes is then named by the respective probability of each outcome. The probability can be encoded as a string, such as "1/6", if this aids in readability, but the string must be coercable to a numeric.

The choice to encode the probabilities in the names of the vector may seem counterintuitive. However, it is this choice which allows for the familiar syntax employed by introductory statistics courses and textbooks to be seamlessly replicated. Consider, for instance, an RV object "X" which we will use to represent a single roll of a fair die.

```
X <- make.RV(1:6, rep("1/6", 6))
X
```

```
## random variable with 6 outcomes
##
##   1   2   3   4   5   6
## 1/6 1/6 1/6 1/6 1/6 1/6
```

Note that although the print method does not illustrate the inherent structure of the object, the probabilities (1/6, for each of the 6 outcomes of the die roll) are actually stored in the names of the object "X".

```
names(X)
```

```
## [1] "1/6" "1/6" "1/6" "1/6" "1/6" "1/6"
```

2.3 Probabilities

By storing the outcomes as the principle component of the object X, we can now make a number of probability statements in R. For instance, we can ask what the probability of obtaining a roll greater than 1 is by using the code *P(X ¿ 1)*. R will check which values in the vector X are greater than 1. In this case, these are the outcomes 2, 3, 4, 5, and 6. Hence, R will return TRUE for these elements of X, and then we can encode a function P to compute the probability of this occurrence by simply summing over the probability values stored in the names of these particular outcomes. Likewise, we can make slightly more complicated probability statements such as *P(X ¿ 5 — X == 1)*.

Several other distributional quantities are computable, including the expected value and the variance of a random variable. As in notation from probability courses, expected values can be found with the "E" function. To compute the expected value for a single roll of a fair die, we run the code *E(X)*.

2.4 Joint Distributions

Aside from moments and probability statements, discreteRV includes a powerful set of functions used to create joint probability distributions. Once again letting X be a random variable representing a single die roll, we can use the *multN* function to compute the probability mass function of n trials of X. Table 2 gives the first eight outcomes for n = 2, and Table 3 gives an the first eight outcomes for n = 3. Notice again that the probabilities have been coerced into fractions for readability. Notice also that the outcomes are encoded by the outcomes on each trial separated by a period.

```
## Loading required package:  MASS
```

Outcome	1.1	1.2	1.3	1.4	1.5	1.6	2.1	2.2
Probability	1/36	1/36	1/36	1/36	1/36	1/36	1/36	1/36

Table 2: First eight Outcomes and their associated Probabilities for a variable representing two independent rolls of a die.

Outcome	1.1.1	1.1.2	1.1.3	1.1.4	1.1.5	1.1.6	1.2.1	1.2.2
Probability	1/216	1/216	1/216	1/216	1/216	1/216	1/216	1/216

Table 3: First eight Outcomes and their associated Probabilities for a variable representing three independent rolls of a die.

discreteRV also includes functions to compute the sum of independent random variables. If the variables are identically distributed, the *SofIID* function can be used to compute probabilities for the sum of n independent realizations of the random variable. In our fair die example, *SofIID(X, 2)* would create a random variable object with the representation given in 4

Outcome	2	3	4	5	6	7	8	9	10	11	12
Probability	1/36	1/18	1/12	1/9	5/36	1/6	5/36	1/9	1/12	1/18	1/36

Table 4: Outcomes and their associated Probabilities for a variable representing the sum of two independent rolls of a die.

2.5 Plotting

discreteRV includes a *plot* method for random variable objects so that a visualization of the outcomes and probabilities can be made simply by calling *plot(X)*. The result of plotting the random variable representing a fair die is given in Figure 1. The x axis includes all outcomes, and the y axis includes the probabilities of each particular outcome. The result of plotting a random variable representing the sum of two independent rolls of a die is given in Figure 2. The result of plotting a random variable representing the sum of 100 independent rolls of a die is given in Figure 3.

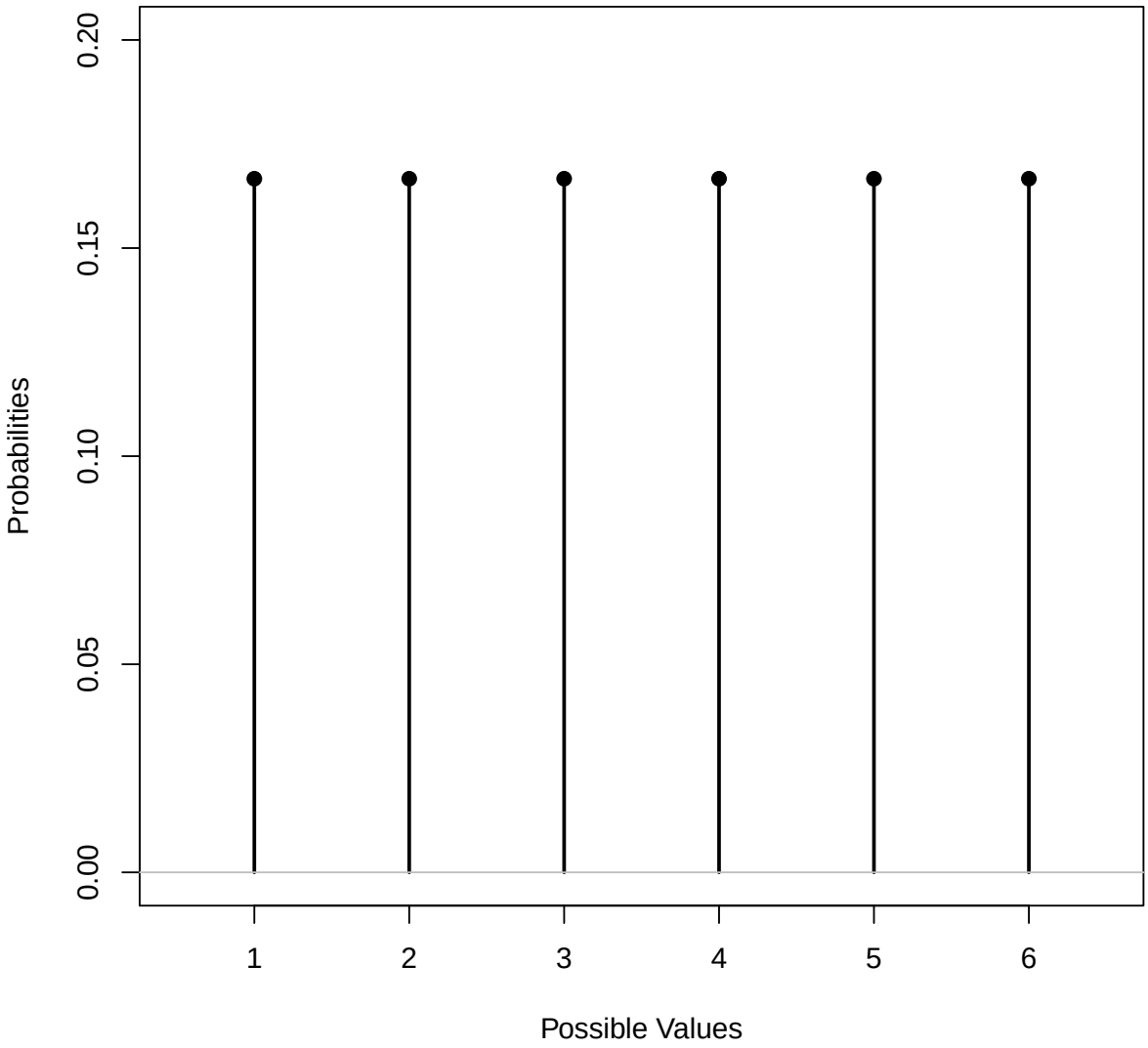


Figure 1: Plot method called on a fair die random variable.

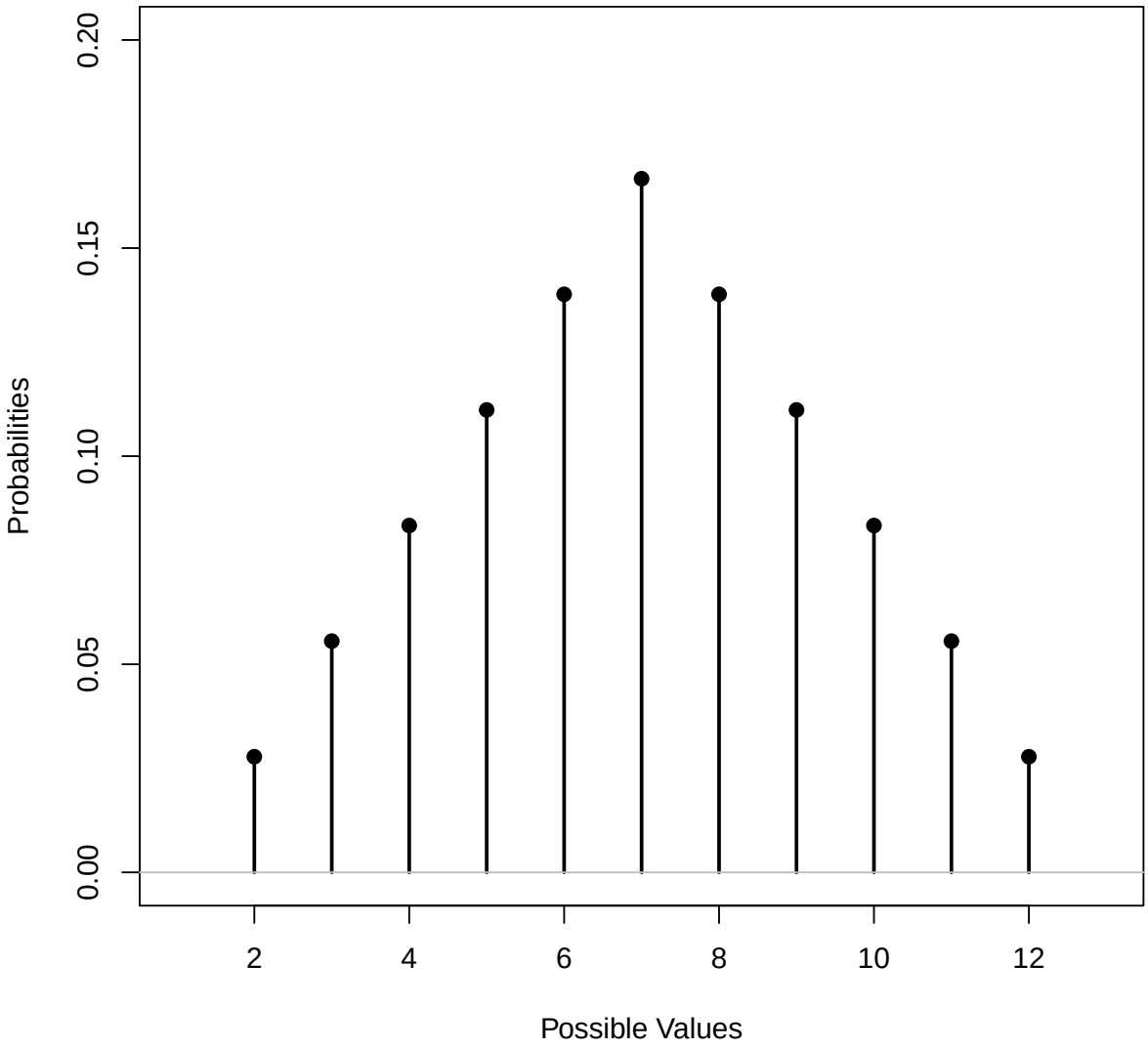


Figure 2: Plot method called on a sum of two fair die random variable.

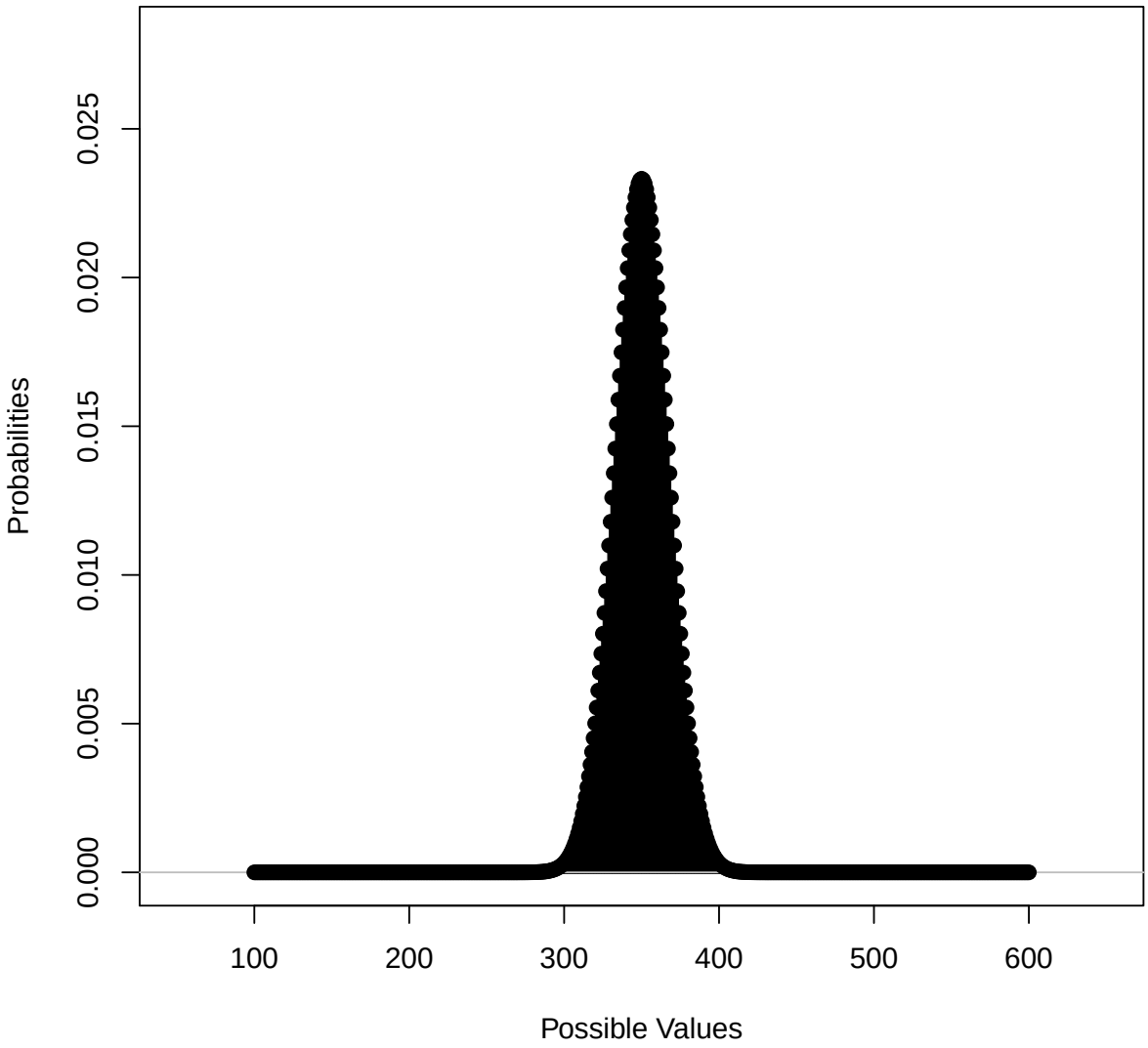


Figure 3: Plot method called on a sum of 100 fair die random variable.

In addition to a plotting method, there is also a method for *qqnorm* to allow assessment of normality for random variable objects, as displayed in Figure 4.

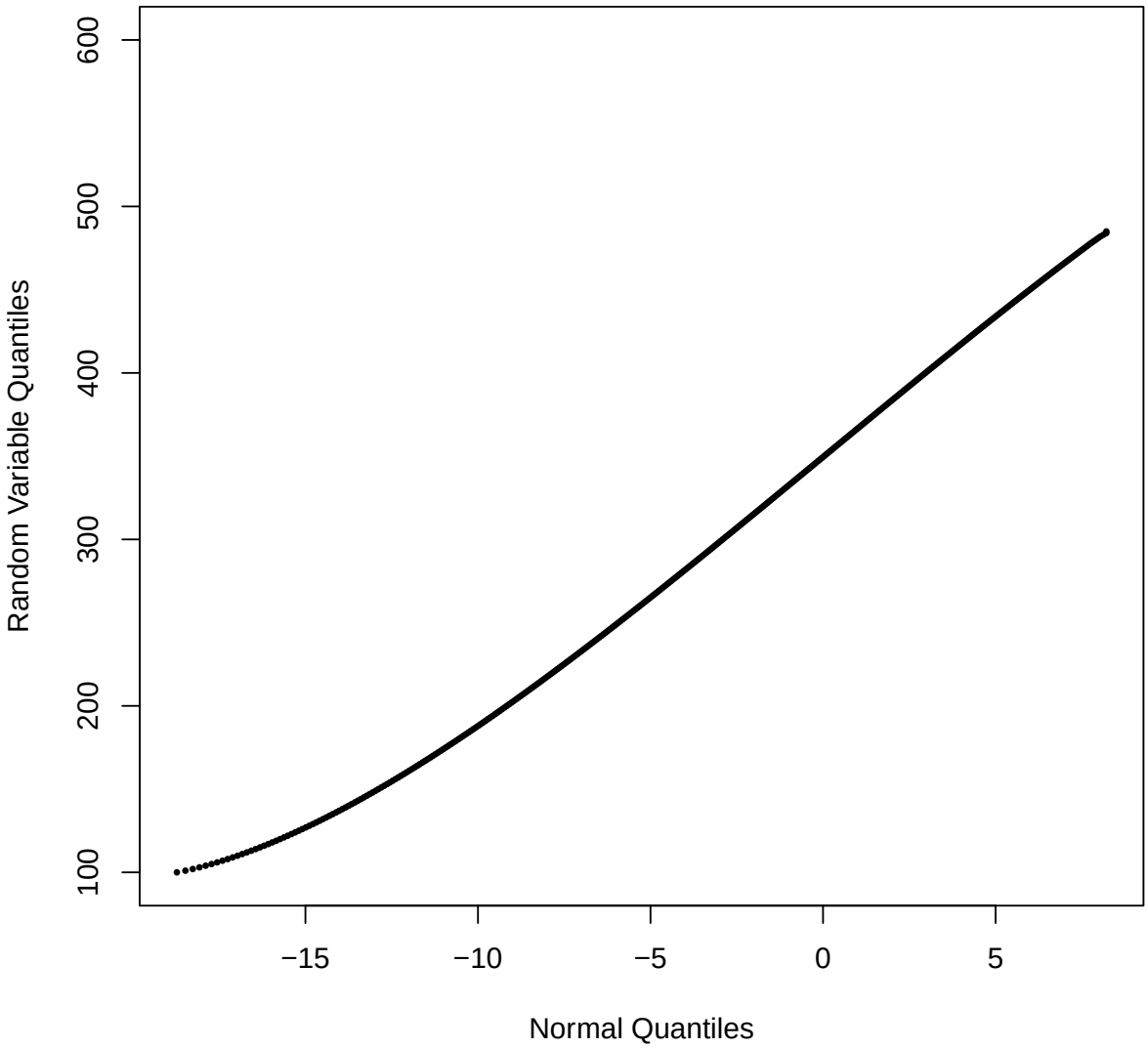


Figure 4: qqnorm method called on a sum of 100 fair die random variable.

3 Simulation

discreteRV also includes a set of functions to simulate trials from a random variable. A list of these functions and brief discriptions of their functionality is available in Table 5.

Name	Description
plot.RVsim	Plot a simulated random vector
Prop	Proportion of an event observed in a vector of simulated trials
props	Proportions of observed outcomes in one or more vectors of simulated trials
rsim	Simulate n independent trials from a random variable X
skewSim	Skew of the empirical distribution of simulated data

Table 5: List of the simulation functions contained in discreteRV.

3.1 Creation

Creating a simulated random vector is done by using the *rsim* function. *rsim* accepts a parameter n representing the number of independent trials to simulate, and a parameter X representing the random variable with which to simulate from. For example, suppose we'd like to simulate ten trials from a fair die. We have already created a random variable object X, so we simply call rsim as follows:

```
X.sim <- rsim(10, X)
X.sim
```

```
## 1/6 1/6 1/6 1/6 1/6 1/6 1/6 1/6 1/6 1/6
##   3   5   1   6   3   3   2   6   5   5
## attr(,"RV")
## random variable with 6 outcomes
##
##   1   2   3   4   5   6
## 1/6 1/6 1/6 1/6 1/6 1/6
## attr(,"class")
## [1] "RVsim"
```

The object returned is a vector of simulated values, with a class attribute to contain the random variable that was used for the simulation. If we would like to retrieve only the simulated values and exclude the attached probabilities, we can coerce the object into a vector using R's built-in *as.vector* function.

```
as.vector(X.sim)
```

```
##  [1] 3 5 1 6 3 3 2 6 5 5
```

It is also possible to retrieve some quantities from the simulation. We can retrieve the empirical distribution of simulated values with the *props* function. This will return the outcomes from the original random variable object, and the observed proportion of simulated values for each of the outcomes. We can also compute observed proportions of events by using the *Prop* function. Similar to the *P* function for probability computations on random variable objects, *Prop* accepts a variety of logical statements.

```
props(X.sim)
```

```
## RV
##   1   2   3   4   5   6
## 0.1 0.1 0.3 0.0 0.3 0.2
```

```
Prop(X.sim == 3)
```

```
## [1] 0.3
```

```
Prop(X.sim > 3)
```

```
## [1] 0.5
```

4 Conclusion

The power of discreteRV is truly in its simplicity. Because it uses familiar introductory probability syntax, it can allow students who may not be experienced or comfortable with programming to ease into computer-based computations. Nonetheless, discreteRV also includes several powerful functions for analyzing, summing, and combining discrete random variables which can be of use to the experienced programmer.

Analyzing Peptide Libraries with peptider

Eric Riemer Hare

January 7, 2014

1 Introduction

Libraries of peptides, or amino acid sequences, have a number of applications in the Biological sciences, from studying protein interactions, to vaccine research. Despite their importance, little analysis has been done to assess the statistical properties of different peptide libraries.

Peptider is a newly-released R package which helps to evaluate many important statistical properties of these libraries. It supports a number of built-in library schemes, including NNN, NNB, NNK, NNS, and trimer schemes. It also allows for easy analysis of user-created custom library schemes. Peptider makes use of the R package *discreteRV*, which allows for manipulation and analysis of discrete random variables. By treating each amino acid in a peptide as a realization of an independent draw from the pool of all possible amino acids, probabilities for the occurrence of peptides can easily be formulated.

This paper will focus on two distinct functional areas of peptider. The first is Library Diversity, or statistical measures of the quality of the library itself. The second is Peptide Coverage, or how likely the library is to include particularly desired peptides, or peptides that are most similar to desired peptides. Before proceeding to discuss these measures, we will first discuss the built-in library schemes, and how to define custom schemes.

1.1 Library Schemes

Peptider has several built-in library schemes. The first is the NNN scheme, in which all four bases (Adenine, Guanine, Cytosine, and Thymine) can occur at all three positions in a particular codon, and hence there are 64 possible nucleotides. The second is the NNB scheme, where the first two positions are unrestricted, but the third position can only be three bases, yielding 48 nucleotides. Both NNK and NNS have identical statistical properties in this analysis, with the third position restricted to two bases for a total of 32 nucleotides. Finally, there are trimer-based libraries in which the codons are pre-defined. Each of these scheme definitions can be accessed with the *scheme* function.

```
scheme("NNN")
```

```
##   class    aacids c
## 1     A       SLR 6
## 2     B     AGPTV 4
## 3     C         I 3
## 4     D DEFHKNQYC 2
## 5     E        MW 1
## 6     Z         * 3
```

To build a library of an appropriate scheme, the libscheme function is used. By default, peptides of length one amino acid (k = 1) will be used, but this can be specified.

```
nnk6 <- libscheme("NNK", k = 6)
```

libscheme returns a list containing two elements. The first, data, describes the probability of occurrence of each possible peptide class. The second, info, describes the number of nucleotides, the number of valid nucleotides, and the scheme definition used.

We can also create a custom library scheme by building a data frame of the same format as in Code Example 1. This code creates a custom trimer-based library with peptides of length six.

```
custom <- data.frame(class = c("A", "Z"), aacids = c("SLRAGPTVIDEFHKNQYMW",
    "*"), c = c(1, 0))
custom6 <- libscheme(custom, k = 6)
```

Having created the library of interest, we now turn our attention to assessment of these libraries.

2 Library Diversity

In this section, we introduce a number of properties which can be used to determine the quality of a given peptide library, and which are computable using peptider.

2.1 Functional Diversity

The functional diversity of a library is the overall number of different peptides in the library. Analyzing the peptide sequences directly is complex, so a useful approach is to partition the library into amino acid classes, wherein each amino acid belonging to a particular class has the same number of codon representations as all other amino acids in that class. Letting v represent the number of valid amino acid classes, k represent the number of amino acids in each peptide, b_i represent the number of different peptides in class i, N represent the total size of the peptide library in number of peptides, and p_i represent the probability of peptide class i, then the functional diversity is:

$$D(N,k) = \sum_{i=1}^{v^k} b_i(1 - e^{-Np_i/b_i})$$

To compute this diversity measure in peptider, the *makowski* function can be used.

```
makowski(6, "NNK")
```

```
## [1] 0.2918
```

2.2 Expected Coverage

The expected coverage of the library is directly related to the diversity of the library. It is the percentage of all possible peptides that the library contains. Letting c represent the number of viable amino acids in the library scheme, the expected coverage is:

$$C(N,k) = D(N,k)/c^k$$

Note that the diversity of the library can range between 0 (for a library with no peptides) and c^k (wherein the library includes every possible peptide). Hence, the coverage must range from 0 to 1. Peptider includes a function to compute the coverage, which again takes the peptide length and library scheme as parameters. It also accepts a parameter N representing the overall number of peptides in the library.

```
coverage(6, "NNK", N = 10^8)
```

```
## [1] 0.5229
```

2.3 Relative Efficiency

In general, it is desireable to achieve a coverage as close as possible to one if we'd like the library to contain all possible peptides. One can achieve this with arbitrarily high probability by increasing the library size (N). However, doing so can drastically increase the cost of the library and may not always be possible. Ideally, the library should contain as high as possible diversity with as small as possible size. Relative Efficiency is a measure to quantify this, and is defined as:

$$R(N, k) = D(N, k)/N$$

As N increases, the relative efficiency subsequently decreases. An ideal peptide library from a cost-benefit perspective would contain both a high diversity (and hence high expected coverage) and still a high relative effeciency. Efficiency can be computed with peptider in the same way as coverage.

```
efficiency(6, "NNK", N = 10^8)
```

```
## [1] 0.3347
```

3 Peptide Coverage

The measures of library diversity included in peptider introduced to this point are useful to broadly assess the library performance. However, applications of peptide libraries often require knowledge of the probability of observing particular peptides in the library[REF].

3.1 Peptide Inclusion

The peptide inclusion probability is the probability of obtaining at least one instance of a particular peptide in the library. If X is defined as a random variable representing the number of occurrences of this peptide, the probability is

$$P(X \geq 1) = 1 - P(X = 0) \approx 1 - e^{-N \sum_i p_i}$$

p_i again is the probability of peptide class i. Because of the dependence on the peptide class, this inclusion probability will vary depending on the particular peptide of interest, and the library scheme. The *ppeptide* function in peptider allows computation of the probability. It accepts a peptide sequence as a character vector, the library scheme, and the library size.

```
ppeptide("HENNING", "NNK", N = 10^10)
```

```
## [1] 0.5166
```

We may also be interested in the range of inclusion probabilities for all peptides rather than just a particular sequence. In this case, the *detect* function is useful. detect differs a bit syntactically as it requires the created library to be passed in as an argument. For an NNK library with peptide lengths 6 and library size 10^7, we have:

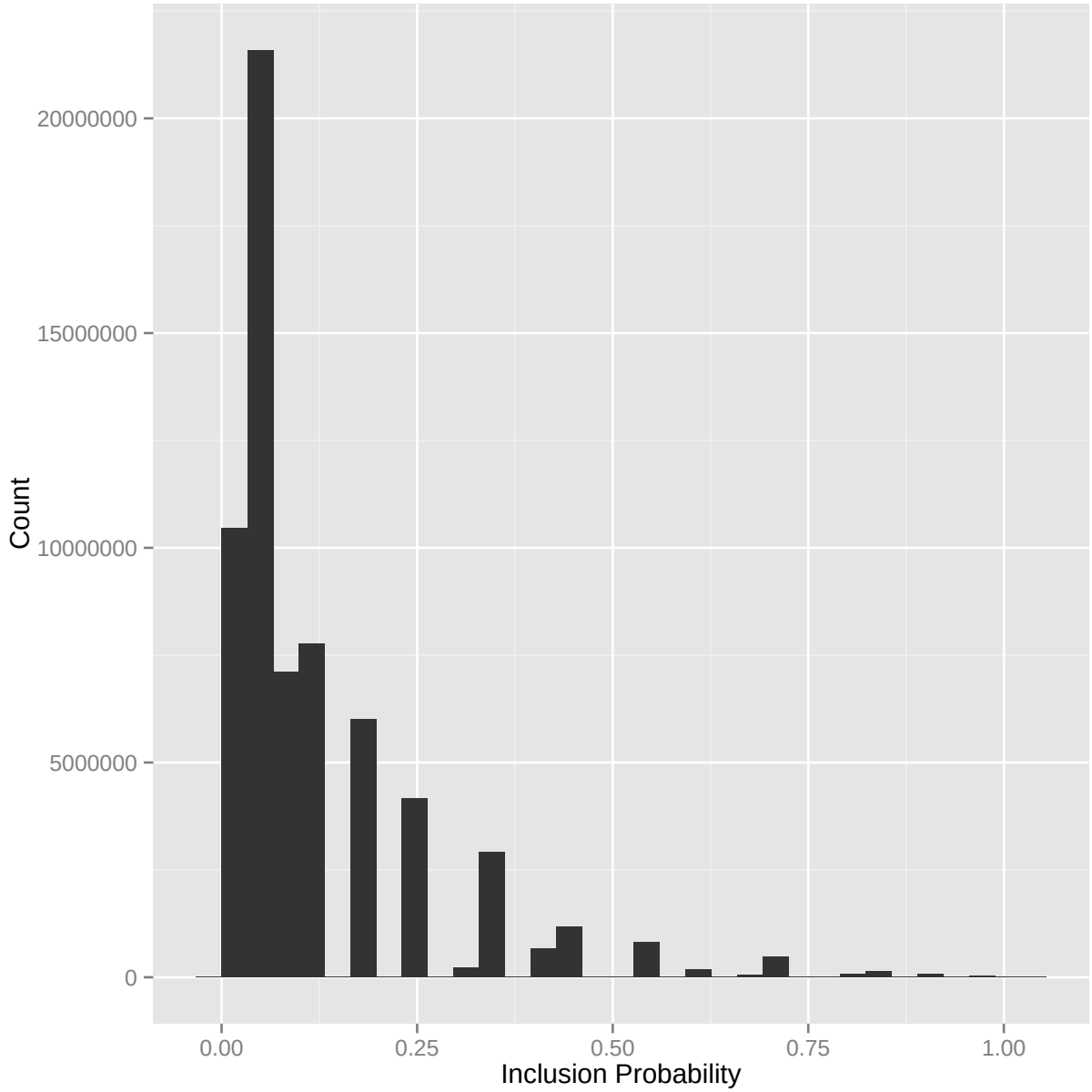
```
my_lib <- libscheme("NNK", 6)

summary(detect(my_lib, 10^7))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0112  0.1840  0.3330  0.3980  0.5560  1.0000
```

```
qplot(detect(my_lib, 10^7), weight = di, geom = "histogram", data = my_lib$data) +
    xlab("Inclusion Probability") + ylab("Count")
```

```
## stat_bin:  binwidth defaulted to range/30.  Use 'binwidth = x' to adjust this.
```



3.2 Neighborhoods

A related concern to inclusion of a particular peptide is inclusion of a peptide's "neighbors". A degree-n neighbor of a peptide p is any peptide that differs by at most n amino acids from p. For example, the length 7 peptide HENNING includes degree-1 neighbor QENNING as well as HQNNING, and a degree-2 neighbor YENNLNG. Peptider includes a function for working with neighborhoods, *getNeighbors*, which accepts a peptide sequence as a character vector and returns all degree-1 neighbors.

```
getNeighbors("HENNING")
```

```
##  [1] "HENNING" "QENNING" "YENNING" "HDNNING" "HQNNING" "HKNNING" "HEDNING"
```

```
##  [8] "HENDING" "HENNLNG" "HENNMNG" "HENNVNG" "HENNIDG"
```

Although there is no built-in function for computing degree-2 and greater neighborhoods for a given peptide, the functionality can be emulated by successive calling of getNeighbors, i.e.,

```
unique(unlist(getNeighbors(getNeighbors("HENNING"))))
```

```
##  [1] "HENNING" "QENNING" "YENNING" "HDNNING" "HQNNING" "HKNNING" "HEDNING"
##  [8] "HENDING" "HENNLNG" "HENNMNG" "HENNVNG" "HENNIDG" "RENNING" "EENNING"
## [15] "KENNING" "QDNNING" "QQNNING" "QKNNING" "QEDNING" "QENDING" "QENNLNG"
## [22] "QENNMNG" "QENNVNG" "QENNIDG" "FENNING" "WENNING" "YDNNING" "YQNNING"
## [29] "YKNNING" "YEDNING" "YENDING" "YENNLNG" "YENNMNG" "YENNVNG" "YENNIDG"
## [36] "HNNNING" "HDDNING" "HDNDING" "HDNNLNG" "HDNNMNG" "HDNNVNG" "HDNNIDG"
## [43] "HRNNING" "HHNNING" "HQDNING" "HQNDING" "HQNNLNG" "HQNNMNG" "HQNNVNG"
## [50] "HQNNIDG" "HKDNING" "HKNDING" "HKNNLNG" "HKNNMNG" "HKNNVNG" "HKNNIDG"
## [57] "HEENING" "HEDDING" "HEDNLNG" "HEDNMNG" "HEDNVNG" "HEDNIDG" "HENEING"
## [64] "HENDLNG" "HENDMNG" "HENDVNG" "HENDIDG" "HENNLDG" "HENNMDG" "HENNVDG"
## [71] "HENNIEG"
```

4 Further Work

Although these functions are suitable for working with peptides up to about length ten, they become slower and more problematic for larger peptides. Work has progressed on a new suite of functions which simplify the encoding of peptides. By recognizing the assumption of independence of each amino acid, we can store counts of each peptide class in order to avoid storing all possible permutations of peptide classes. This greatly simplifies the computation time and allows for peptides of length 20 and beyond to be analyzed in the context of peptide libraries.

Replacement functions for most of the previously described functionality using this new encoding scheme is available unexported in peptider. For instance, to compute the coverage of a size 10^25 NNK library with peptides of length 18, one can call:

```
peptider:::coverage_new(18, "NNK", N = 10^25)
```

```
## [1] 0.7923
```

Functions for coverage, efficiency, diversity, and inclusion probabilities have all been written and have the "new" suffix.

5 Conclusion

A Web Application for Efficient Analysis of Peptide Libraries

Eric Riemer Hare

January 6, 2014