

Math 152 Lab Assignment 2

February 15, 2007

This lab assignment shows how to implement sample spaces, events, probability measures and random variables in R.

1 Probability models in R

A set in R is implemented as a vector (or a list). This isn't perfect, since lists and vectors have order whereas sets don't. So we have to ignore the order. Also, vectors can have the same entry more than once, whereas that's not part of the idea of set.

Probability measures in R Here's how to implement probability measures on a set. Start with a set $\Omega = \{1, 2, 3, \dots, 20\}$. Think of an experiment (rolling a strange 20-sided die) with 20 conceivable outcomes.

In R you define Ω like

```
omega <- 1:20
```

then recall that a probability measure P on Ω assigns numbers to each of the 2^{20} (approximately 1 million) subsets of Ω . This is a large number, but even with an old computer we could still explicitly represent this. However to save space, time, and effort, we'll define a probability measure in R as a *function* (taking a subset as input). To do this recall that we can find the measure of a subset as soon as we know the measures of each of the one-element subsets, of which there are only 20 here. We may assign the measures of the one-element subsets to be any non-negative numbers we choose, as long as they add up to 1.

Even more simply, we can choose any 20 nonnegative numbers at all, and then divide them all by their sum. Let's say we define

```
f <- ceiling(10*runif(20))
```

to get 20 integers between 1 and 10. This makes a random choice; I just did that to save typing 20 numbers. Say I got the following vector for f

```
> f
[1] 2 1 3 9 3 4 2 3 8 3 4 1 2 8 4 8 5 3 1 4
```

then we think of an experiment where there are 2 chances of getting a 1 outcome, 1 chance of a 2 outcome, 3 chances of getting a 3 outcome, 9 chances of getting a 4 outcome, etc. This makes sense only relative to the “total number of chances” , ie

```
> sum(f)
[1]
```

so we'll define the probability density p as a function on Ω (i.e., the inputs are elements of Ω rather than subsets) as follows

```
> p <- function(w) f[w]/sum(f)
```

We can check to make sure that the numbers $p(1), p(2), \dots, p(20)$ add up to 1 by

```
> sum(p(omega))
[1] 1
```

Why this works. What's happening here is that R is “vectorized” : to get the vector with entries $p(1), p(2)$, etc we can just give a vector with entries 1, 2, etc as input and R computes the function p for each entry as output. R functions usually work this way, as long as it makes sense. Our R function works this way because “indexing” (getting entries in f by their position) works that way. Try typing

```
> f[c(1,3,5)]
```

and see what you get. So what is the output of

```
> p[omega]
```

going to be?

From density to measure. Now to define the measure of a subset, we just need to add up the probabilities of each element in the subset. We can get these probabilities just as above. If E is a subset of Ω , then in R we can type $p(E)$ to get the probabilities of the elements of E :

```
> E <- c(2,4,5,8,9,16,19)
> p(E)
```

What do you think $p(E)$ will be? Try it.
So the measure of E will just be

```
> sum(p(E))
```

We can make this into a function so we don't need to keep writing it (not that it's that long)

```
> P <- function(E) sum(p(E))
```

So, starting from a vector of nonnegative numbers defining the “chances” of a particular element of the sample space, we’ve defined a probability measure which will tell us the probability of any event.

1.0.1 Questions

- Find the probability of rolling an even number on this 20-sided die (with whatever probabilities you assigned for each face). What event is this (what subset of the sample space)?
- Things are intended to be set up so that if you change your vector f , the functions p and P will change accordingly. Try this and make sure it works. How can you tell?

2 Generating random outcomes (sampling)

It’s very useful to be able to generate random choices which follow a specified probability measure. That is, for a given probability model, we’d like to simulate the experiment it’s modelling on a computer. Here’s how to do that.

We’ll use the basic random-number generator in R, `runif()`.

Start with the sample space Ω and probability measure P you used above. Recall that when we do

```
x <- runif(1)
```

we get a random number between 0 and 1, and all numbers in this range have an equal likelihood. Strictly speaking, we can say that the probability that x lies in a certain range (a, b) is just the length of this interval. We used this to get equally-likely door choices: basically we divided the interval $(0, 1)$ into three equal pieces. Here we’ll divide the interval into 20 pieces, and we want the first piece to have length $p(1)$, the second piece to have length $p(2)$, etc. Then we want to find which piece x falls in. This can be done with the R function `cut()`. So the challenge is to figure out how to divide up the interval. To get you started, try first to generate a random integer between 1 and 4, with probabilities $1/3, 1/4, 1/4, 1/6$ respectively. How should the interval be divided up? Where are the breaks?

Once you’ve determined what the breaks should be (of course, you can use R for this), define a vector called “breaks”. It should have 21 elements, the first 0 and the last 1. Then define

```
randomChoice <- function() {  
  x <- runif(1)  
  w <- cut(x, breaks=breaks, labels=FALSE)  
  return(w)  
}
```

Now `randomChoice()` should produce random outcomes in Ω with probabilities according to P . You can modify the function to make many random choices at once: just change it to

```
randomChoice(n=1) {
  x <- runif(n)
  w <- cut(x,breaks=breaks,labels=FALSE)
  return(w)
}
```

To check that the frequencies of different choices agree with the probability density p just ask for a large number of choices, say 10000, and make a table to count the frequencies of different outcomes:

```
X <- randomChoice(10000)
table(X)
```

Divide the entries in the table by 10000; the results should be close to p .

2.0.2 Questions

- If you roll this 20-sided die many times, how often (what percent of the time) do you get an even number? Does this agree with your calculation in the previous section of what the probability of getting an even number is?

3 Cartesian products and independence

We can create Cartesian products of sets in R. Recall that the Cartesian product of a set S and a set T is the set $S \times T$ of all pairs of elements of S with elements of T . To create this in R we can do the following

First, think about how we'd list the elements. Say $S = \{1, 2, 3\}$ and $T = \{a, b\}$. If we write the pairs in dictionary order, we have

1	a
1	b
2	a
2	b
3	a
3	b

The first column (the x coordinate, so to speak) consists of the vector

1,2,3 where we duplicate each entry. We can do this in R as follows

```
xcoords <- rep(c(1,2,3),c(2,2,2))
```

This says to repeat each element of the 1,2,3 vector 2 times. The 2 is because T has two elements; so in general we want to repeat each element as many times as there are elements in T . So we do the following

```

m <- length(T)
times <- rep(m,length=length(S))
xcoords <- rep(S,times)

```

This gives us the first column; the second column is easier. We just replicate the vector a,b three times. In general

```

n <- length(S)
ycoords <- rep(T,n)

```

Now we can just put these vectors side-by-side to make the table above. Let's make it into a function so we don't have to do these steps again:

```

cartesianProduct <- function(S,T) {
  n <- length(S)
  m <- length(T)
  xcoords <- rep(S,rep(m,n))
  ycoords <- rep(T,n)
  prd <- cbind(xcoords,ycoords)
  return(prd)
}

```

This returns a matrix, which is how we listed the elements above, but for R to see it as a set (where the elements are rows) we need to modify it a bit. We'll just define a function to do the conversion:

```

mat2list <- function(M) {
  n <- nrow(M)
  result <- vector(length=n,mode='list')
  for (i in 1:n) result[[i]] <- M[i,]
  return(result)
}

```

This converts from a matrix to a vector (list) whose elements are the rows of the original matrix.

To define a product measure $P \otimes Q$, recall that we can just make the measure whose density is the product of the densities of the measures P and Q .

```

prodDensity <- function(p,q) {
  function(w) p(w[,1])*q(w[,2])
}

```

This works for the matrix version of the Cartesian product. To make it work for the set version, we have to define functions giving us the coordinates.

4 Random variables

Recall that a random variable is just a function defined on the sample space.