RANDOM VARIABLE GENERATION (PART 2)

And now for something completely different...

Prof. Chris Jermaine
cmj4@cs.rice.edu

The Dirichlet Distribution

- Now that we've done the Gamma distribution...
  - — In reality, we're not interested in the Gamma directly for our project
  - — Rather, we are interested 'cause can be used to generate a Dirichlet RV
- Dirichlet
  - — Real-vector-valued RV
  - — Parameter set is a list of m real values, each larger than zero
  - — Output is a vector-valued list of real numbers from 0 to 1:

$$\langle 0.2, 0.5, 0.2, 0.1 \rangle$$

  - — Constrained so that they sum to one
  - — Super important, 'cause used to model a random vector of probabilities

Turns Out That Generating a Dirichlet is Easy

- Given params $k_1, k_2, ..., k_m$
- Generate m random values, using $rv_i \sim \text{Gamma}(k_i, c)$
- Then ith entry in output vector is simply

$$\frac{rv_i}{\sum_{j=1}^{m} rv_j}$$

- That's it!

Generating a Multinomial

- MN simulates having a large (infinite) number of balls in a bag
- Are *m* colors for the balls
- Proportion of color *i* is p_i
- Then you reach in and select *n* balls at random
- *i*th entry in vector is how many of color *i* that you selected
- So have a vector of ints no less than zero
- Where L1 norm is *n*

How To Implement?

- Discrete, so generally easier than continuous
- Assume $n = 1$
- Just draw a random value *rv* from zero to one
- Compute *i* where

$$rv \geq \sum_{j=1}^{i-1} p_i$$

- And

$$rv \leq \sum_{j=1}^{i} p_i$$

- Then return a vector of all zeros, except for a one at pos *i*

What If *n* exceeds one??

- Just repeat the last process n times
- And add up the n resulting vectors
- Simple! But can you do this really fast?
- Sure! Generate n random values, using $rv_i \sim$ Uniform (0, 1)
- Sort 'em
- Then make a linear pass through random values and the p_i's
- And merge 'em! Will do on the board...
- Running time? $O(n + m + n \log n) = ??$

Some Notes on PRNG

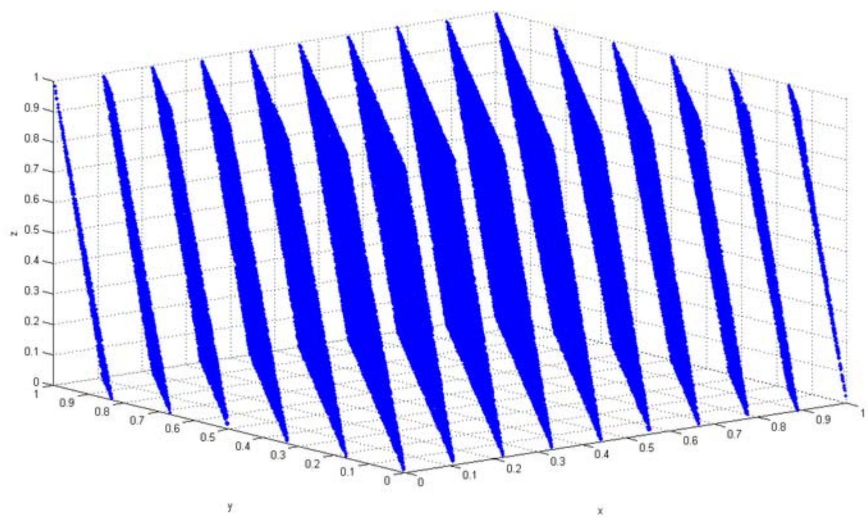
- All of this depends on being able to generate random doubles
  - From zero to one
- Easy if you can generate a random sequence of bits
- But how to generate that sequence of bits? Use a PRNG!
- We’re not asking you to implement one...
  - Java comes with an imp. of a classic PRNG: the “linear congruential”
- Will talk briefly about this. Want to learn more?
  - Chapter 7 of classic CS book “Numerical Recipes”
  - http://books.google.com/books/about/Numerical_recipes.html?id=1aAOdzK3FegC

The Linear Congruential Method

- Defined by $X_{n+1} = (aX_n + c) \bmod m$
  - — In this formula, X_n is the nth pseudo-random number generated
  - — X_0 is known as the “seed”
  - — Note: can always recreate sequence of bits by going back to X_0
- The “period” of a PRNG is the time until it loops back
  - — Obviously, we want a period of m here
  - — Theoretically guaranteed if:
  - — (a) $a - 1$ is divisible by all prime factors of m
  - — (b) if m is a multiple of 4, then $a - 1$ is a multiple of 4
  - — (c) c and m are relatively prime

The Linear Congruential Method

- Before people understood this stuff...
  - — Were some classically bad parameter choices
- Infamous example was IBM's "RANDU" routine
  - — Following picture shamelessly stolen from Wikipedia



- LCM still widely used, though more modern PRNGs available!

Some Closing Notes on A4

- You are asked to provide several implementations for

  `IRandomGenerationAlgorithm <ReturnType>`

- In the end, we felt it was quite challenging...
  - So we are giving you part of the design
  - In particular, the abstract class interface for our various RNG algorithms:

    `ARandomGenerationAlgorithm <ReturnType>`

- Key idea: put the PRNG in the abstract class
  - So abstract away problem of generating uniform numbers...
  - Totally handled by abstract
  - Have two constructors in both abstract and concrete. One that accepts a seed...
  - And one that accepts an “IPRNG” object
  - IPRNG wraps up the PRNG algorithm; we’ve given you “PRNG” class

Some Closing Notes on A4

- In concrete, just call "super()" and then set up local structures
- In abstract...
  - If you get the seed, use it to set up a new PRNG object as the default
  - If you get an IPRNG, then use it
- This is also useful for "linking" RV generation algorithms
  - Example: if Dirichlet uses a bunch of Gammas...
  - They should all use Dirichlet's IPRNG object
  - That way you know they are all using the same PRNG sequence

Some Closing Notes on A4

- How do A4 test cases work?
  — After all, two “correct” imps may spit out different random values
- We instantiate a random variable, then use it to gen many values
- Then compare theoretical vs. observed mean and std. dev.
- Highly unlikely you can have a bad imp that passes the test case
- Downside? Can be difficult to debug
  — Not always a clear path from bad mean/std. dev back to bug in your code
- Sooo... start early!
  — Might be easy, but also could be quite challenging

Questions?