RANDOM VARIABLE GENERATION

And now for something completely different...

Prof. Chris Jermaine
cmj4@cs.rice.edu

Prof. Scott Rixner
rixner@cs.rice.edu

Goal In This Class

- Eclectic mix of algorithms and programming
- Remember when I claimed:
  - CS is fundamentally about algorithmics?
  - Programming is relevant only to the extent that it allows computers to run our algs
  - The fact we have to write code is evidence of the epic fail of PL researchers!
- So we won't ever leave algorithms behind
  - And over the next 1.5 classes, will make a significant detour away from PL
  - Will study RV generation
  - Vital to getting our ML algorithm over text to work

What's a Random Variable?

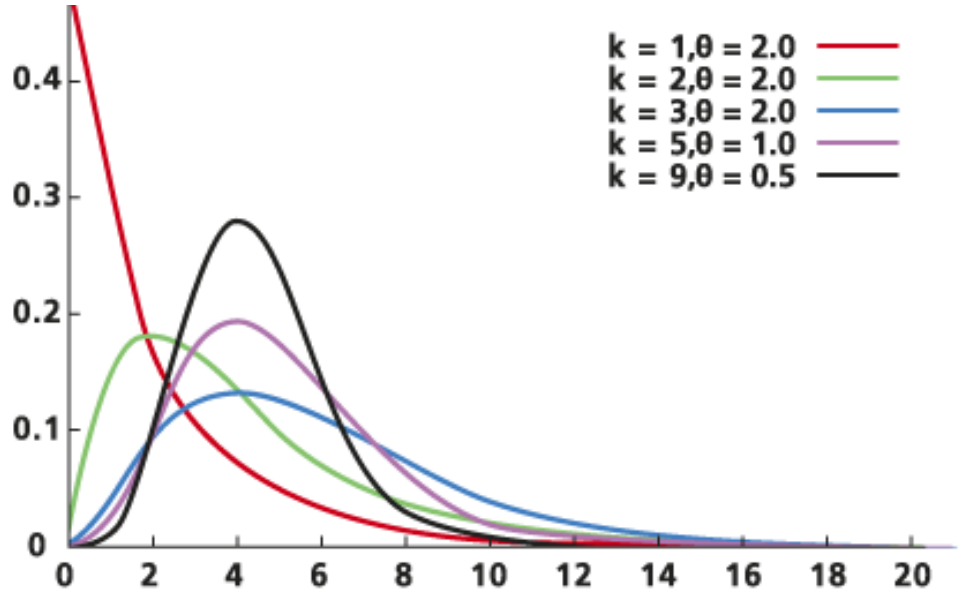
- Think of it like a little machine
- You press a button
- It spits out a random value
- Often, that value is a real number
- But can be anything. You commonly see:
  - — A randomly chosen integer
  - — A randomly chosen item from a set
  - — A random vector of reals
  - — A random sequence of items chosen from a set
  - — And so on...

RVs Whose Domain is Real Numbers

- Meaning you press a button, get out a real
- Are often characterized via a “PDF”
  - — “Probability Density Function”
  - — $f(x)$ denotes the “density” of the RVs distribution at x
- Then $\int_{x_1}^{x_2} f(x)dx$ is...
  - — the probability the machine spits out a value from x_1 to x_2.
  - — Clearly, the total area under the curve f must be one

One Particularly Important Distribution

- Is the “gamma distribution”---will be fundamental to our ML algo
- Takes two parameters:
  - — The shape *k*
  - — The scale θ
- Here’s the PDF for various parameter combos



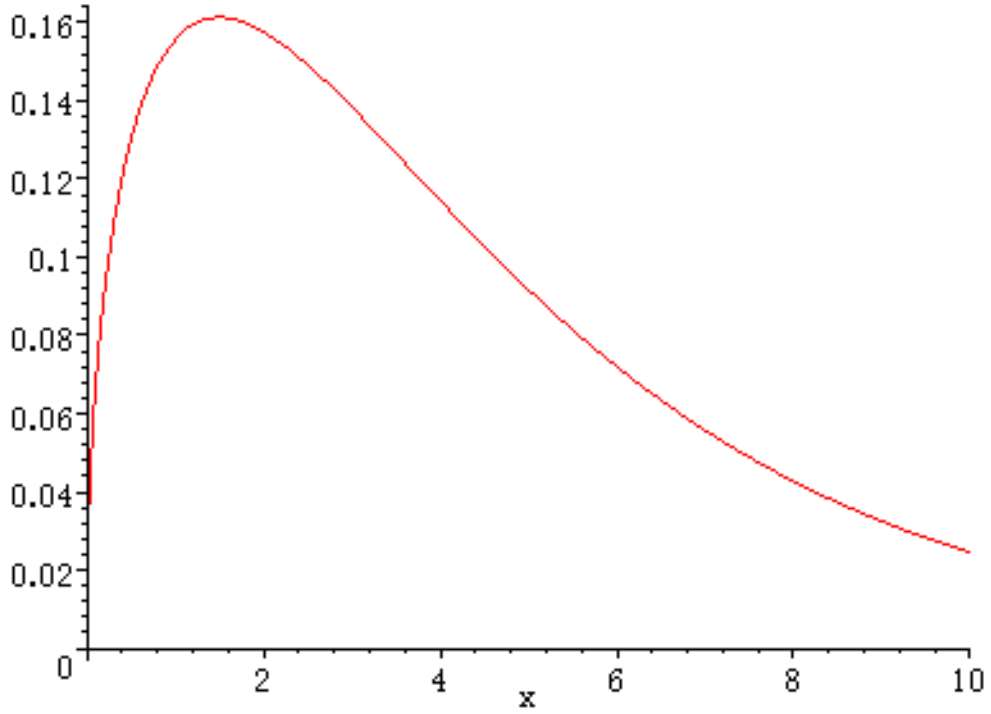
One Particularly Important Distribution

- The mathematical form of the gamma PDF is

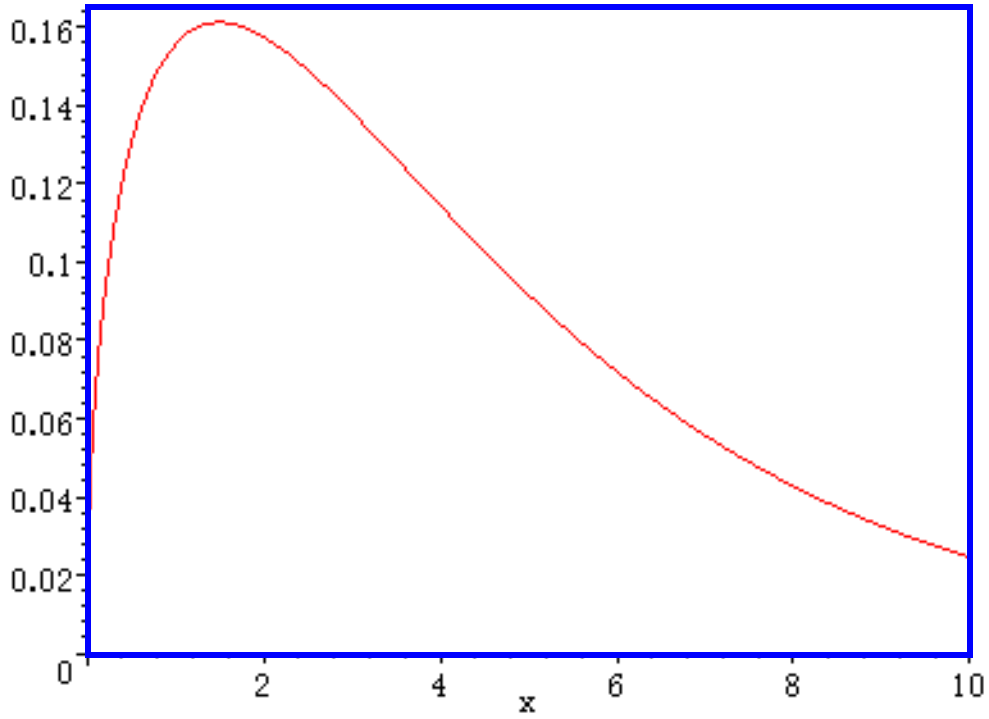
$$f(x|k,\theta) = \frac{x^{k-1}e^{\left(-\frac{x}{\theta}\right)}}{\Gamma(k)\theta^{k}}$$

- An age-old question in CS/comp. statistics:
  - — Given a particular PDF...
  - — how to simulate a RV having exactly that PDF?
  - — In our case, how to simulate a gamma-distributed RV, given the PDF?

The Easiest Way Is “Rejection Sampling”

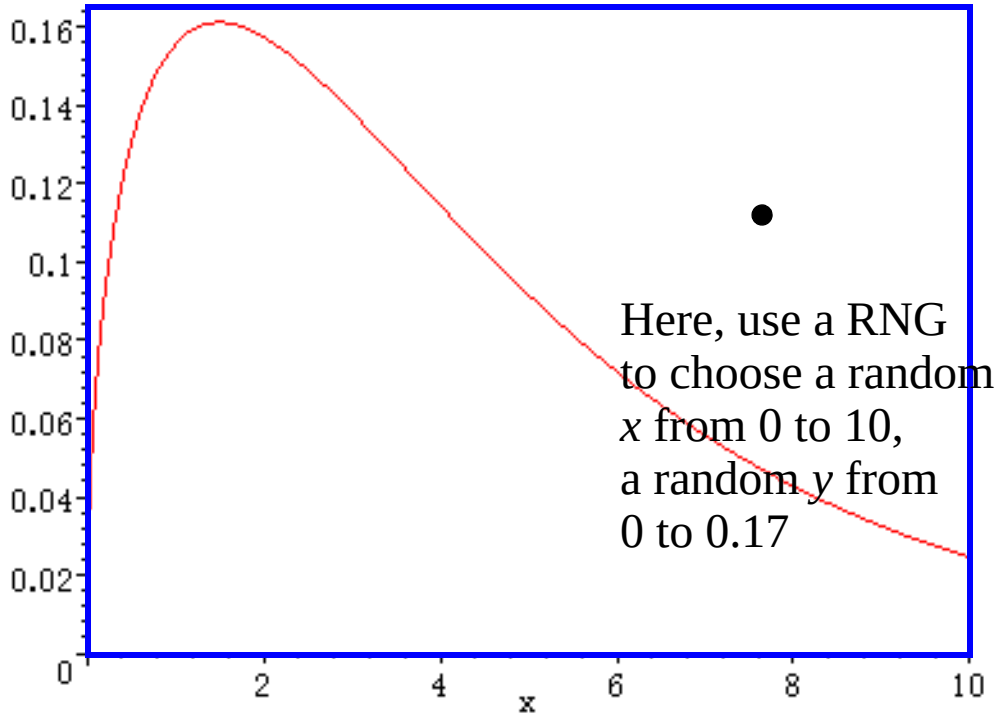


The Easiest Way Is “Rejection Sampling”



— Basic idea is to draw a box (“envelope”) around the PDF

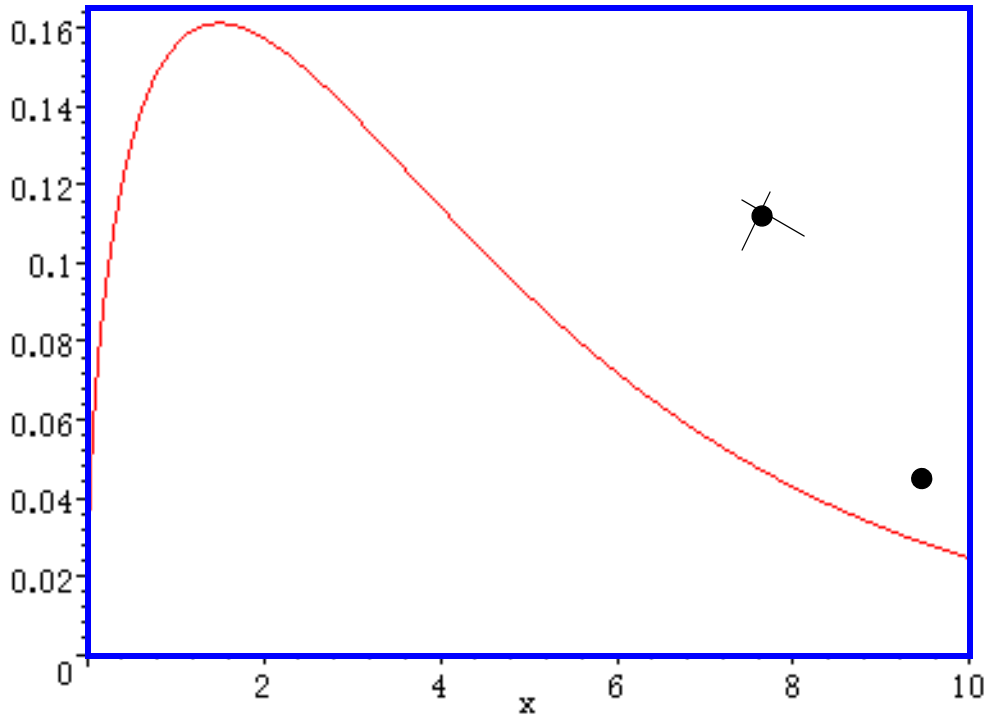
The Easiest Way Is “Rejection Sampling”



— Basic idea is to draw a box (“envelope”) around the PDF

— Then throw a dart randomly into the box

The Easiest Way Is “Rejection Sampling”

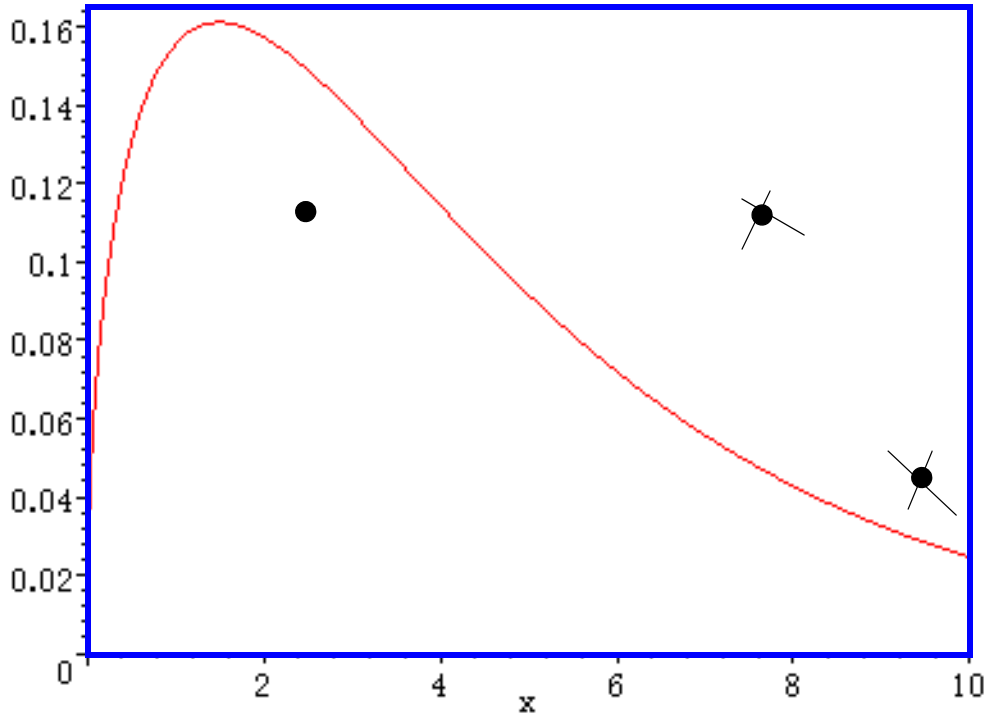


— Basic idea is to draw a box (“envelope”) around the PDF

— Then throw a dart randomly into the box

— If above the PDF, you reject it, and try again

The Easiest Way Is “Rejection Sampling”

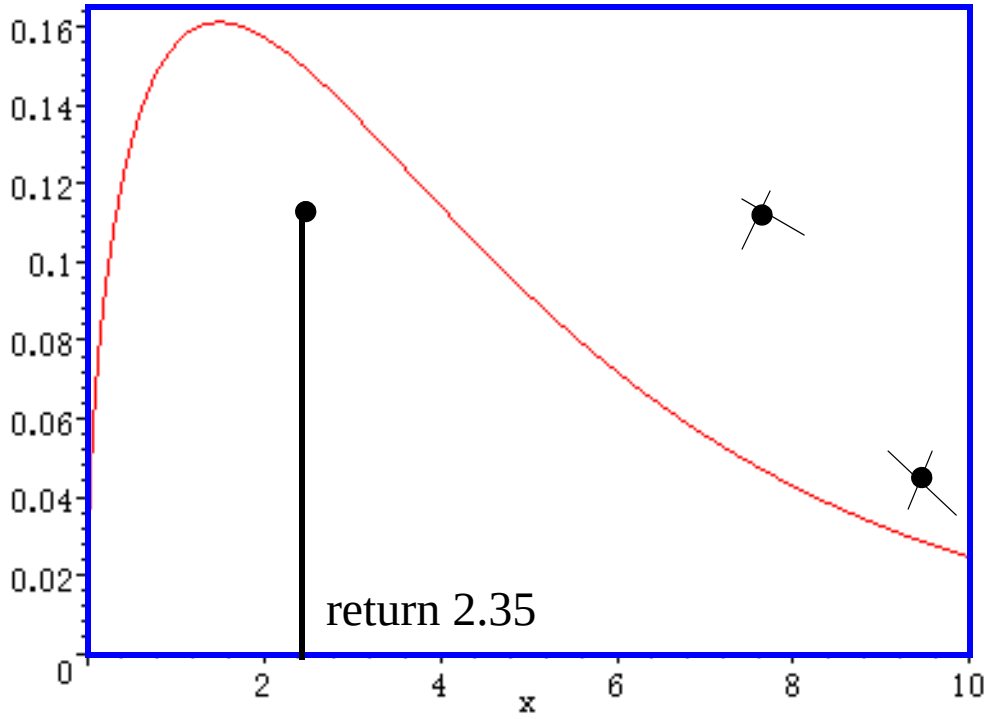


— Basic idea is to draw a box (“envelope”) around the PDF

— Then throw a dart randomly into the box

— If above the PDF, you reject it, and try again, and again

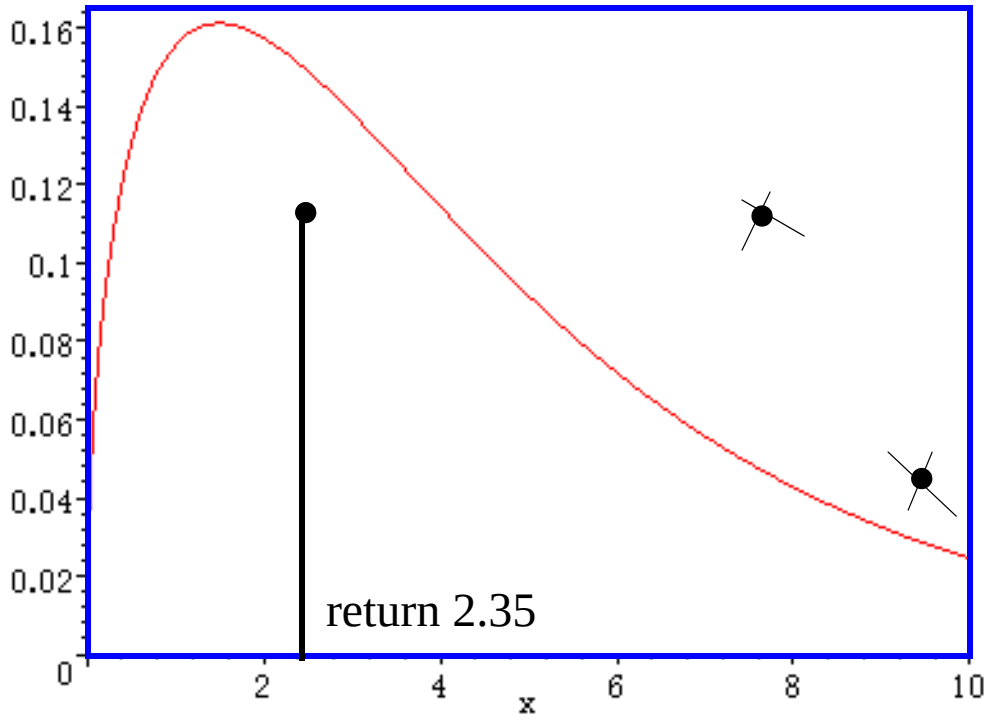
The Easiest Way Is “Rejection Sampling”



— Got one! So the *x* you return is the *x*-val of the first dart under the line

— Can prove this process does in fact sample from a RV whose PDF is the red line

One Super-Cool Thing About RS



— Constant multiplicative factors in PDF can be ignored so for Gamma, kill denom

$$f(x|k,\theta) = \frac{x^{k-1}e^{\left(-\frac{x}{\theta}\right)}}{\Gamma(k)\theta^{k}}$$

Why?

Envelope Can Be Any Shape

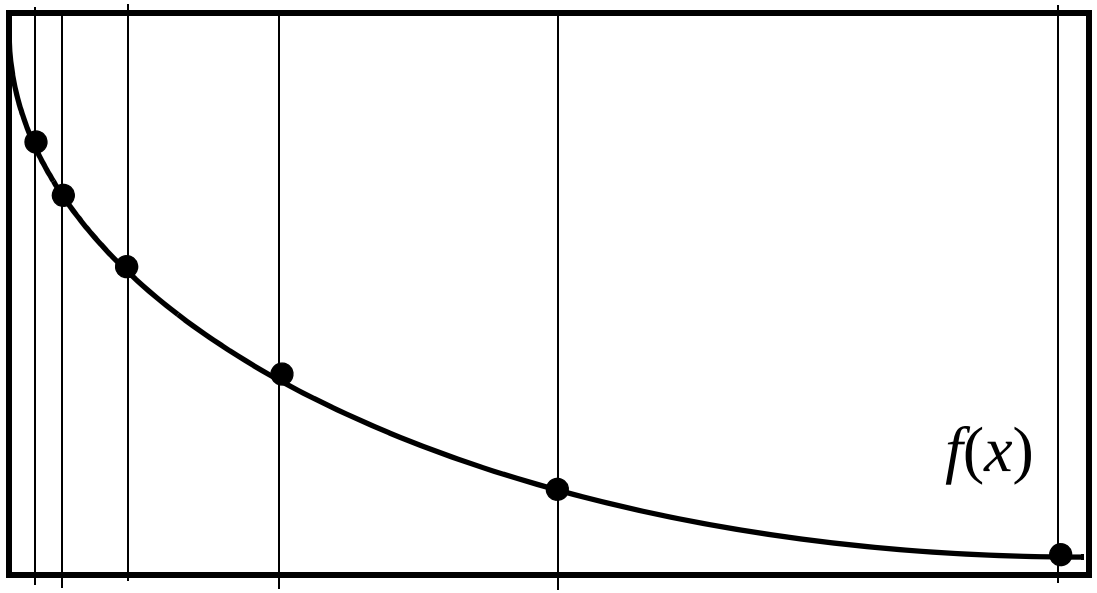
- As long as it fully encloses the PDF
- And as long as your darts hit uniformly in the envelope
- How to do this (somewhat efficiently) for the gamma PDF?
- First, assume the shape is at most one, so concave, decreasing
  - More on what to do if shape exceeds one in a minute
- And assume the scale is one so the PDF becomes

$$f(x|k, \theta) = x^{k-1} e^{(-x)}$$

- Also assume we have l and $h = 2^i l$ for some i, which are the lowest and highest values we can ever sample

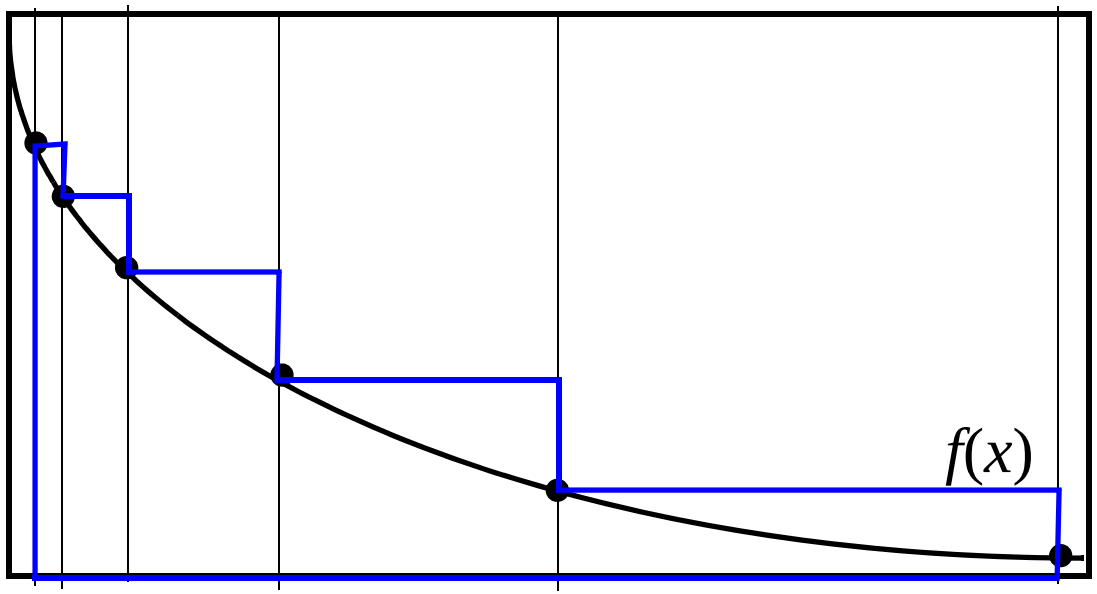
Now What Do We Do?

- First, compute $f(x)$ at l, $2l$, $4l$, $8l$, $16l$, ..., h



Now What Do We Do?

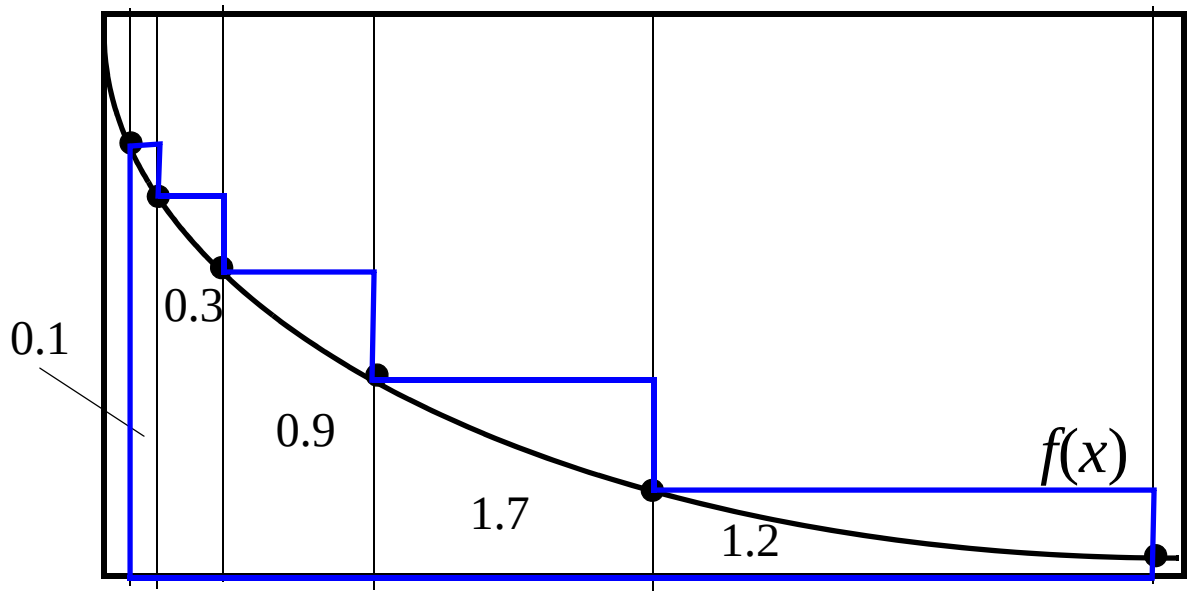
- Your envelope becomes the shape enclosing these dots



— Note: this is *i* steps!

Now What Do We Do?

- Your envelope becomes the shape enclosing these dots

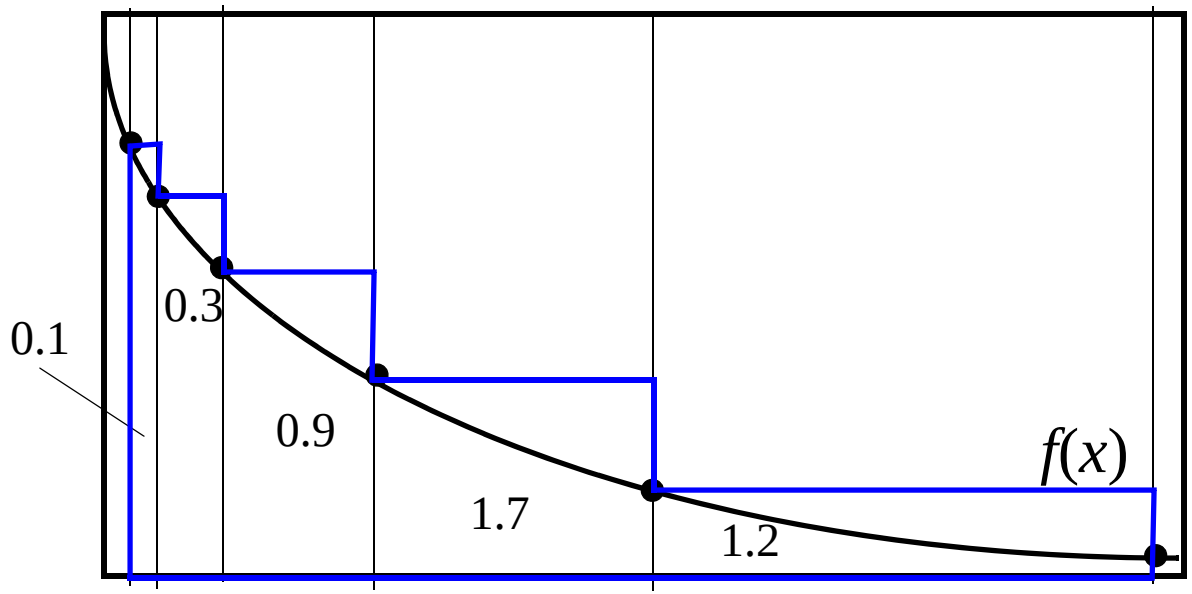


— Note: this is *i* steps!

— Compute the area of each step

Now What Do We Do?

- Your envelope becomes the shape enclosing these dots



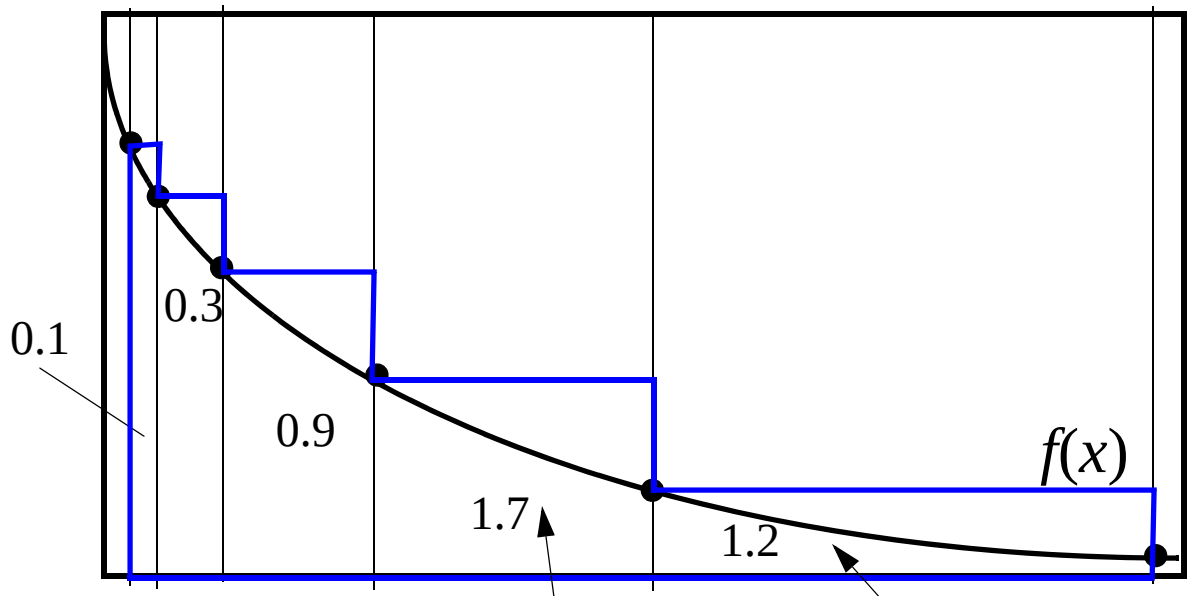
— When it's time to thow a dart, get a random number *r* from 0 to total area

— Choose step *t* if sum of all areas to left of *t* is less than *r*

— But sum of all areas to left of *t*, plus area of *t*, is at least *r*

Now What Do We Do?

- Your envelope becomes the shape enclosing these dots



— When it's time to throw a dart, get a random number *r* from 0 to total area

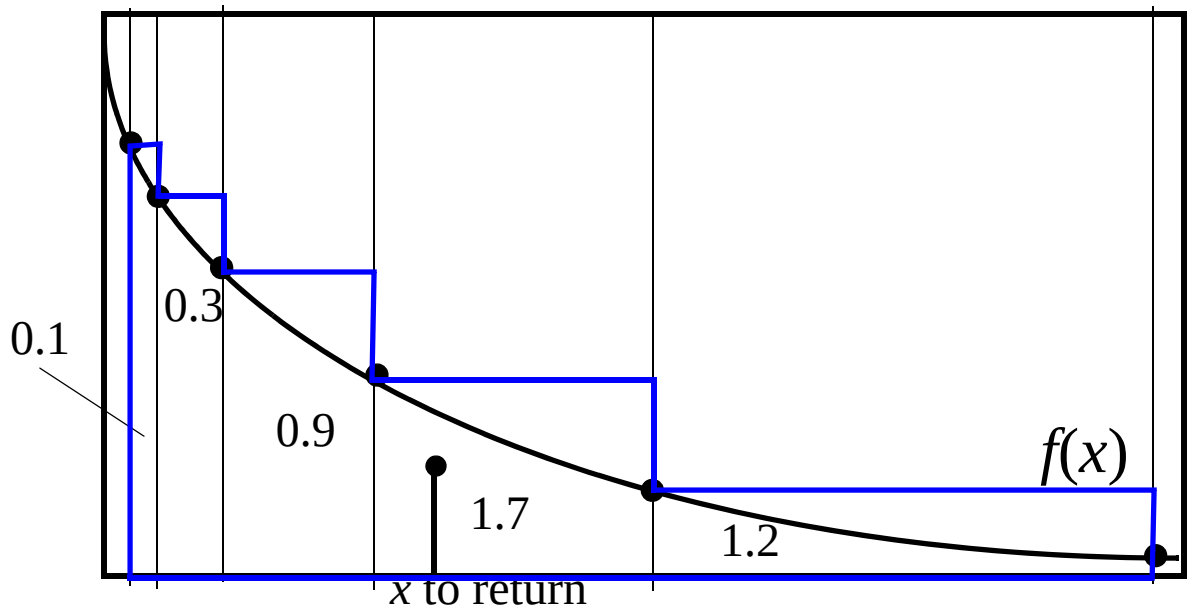
— Choose step *t* if sum of all areas to left of *t* is less than *r*

— But sum of all areas to left of *t*, plus area of *t*, is at least *r*

— Example: *r* = 2.9, choose this one; r = 4.1, choose this one

Now What Do We Do?

- Your envelope becomes the shape enclosing these dots

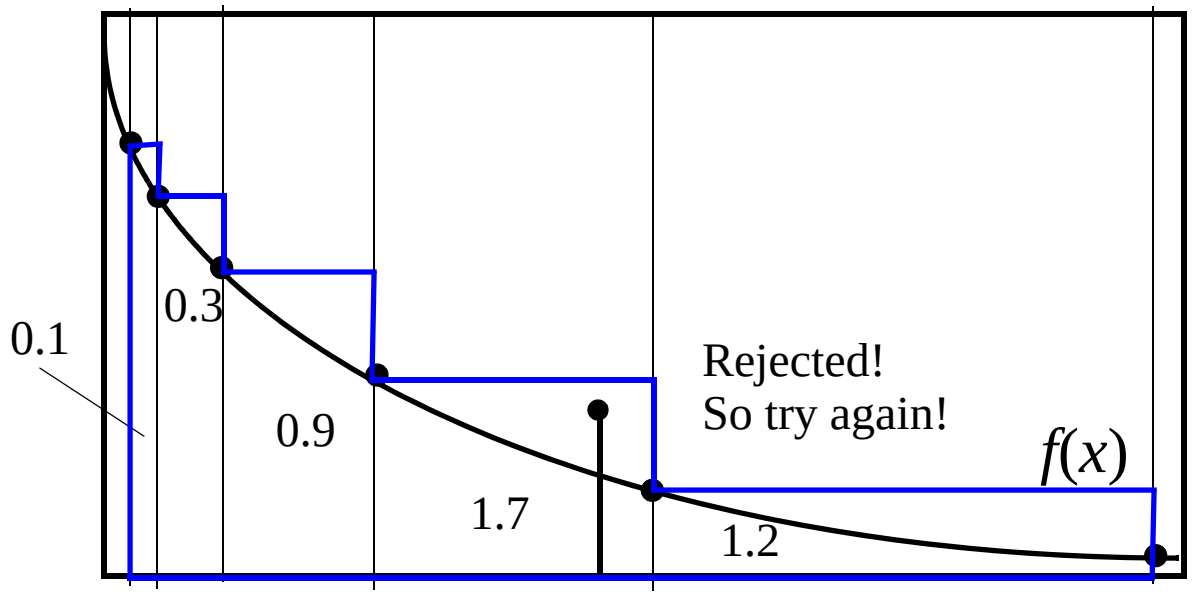


— Whatever step you choose, “throw” a dart uniformly within the step

— If it is under the PDF, then return the *x* of the dart that you chose

Now What Do We Do?

- Your envelope becomes the shape enclosing these dots



— Whatever step you choose, “throw” a dart uniformly within the step

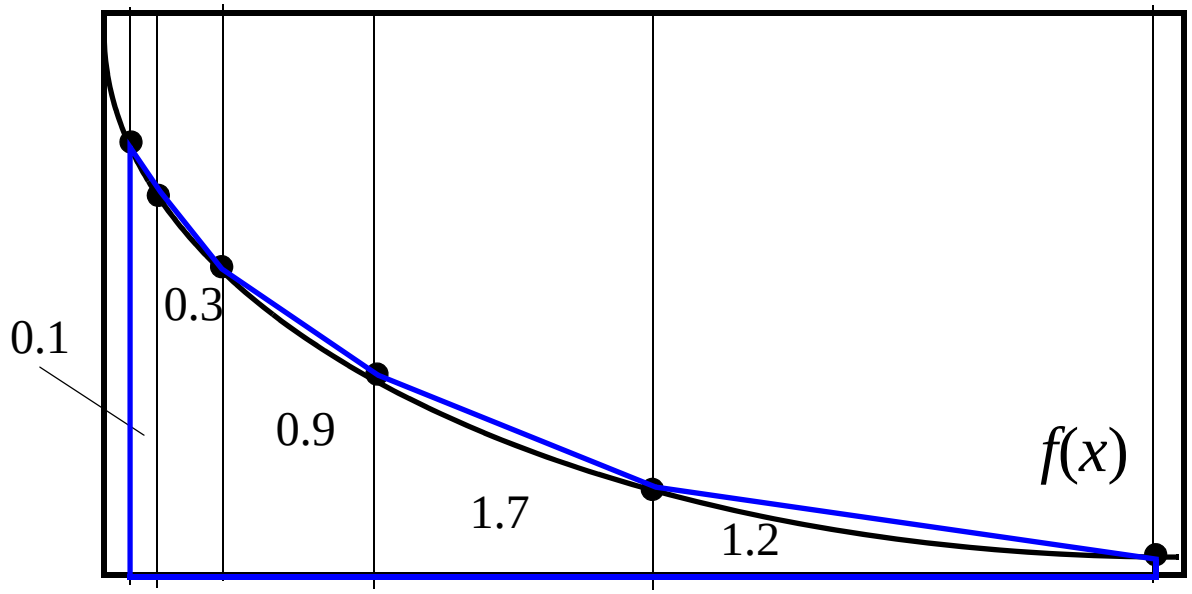
— If it is under the PDF, then return the *x* of the dart that you chose

— If it is not under the PDF, then choose a step using same process, and try again

— Keep going until you get one under the PDF!

Want To Be Really Fancy?

- Use trapezoids instead of steps!
- Will result in less rejections
- But remember, your dart must fall uniformly within the trapezoid



Two More Issues

- What if scale you want is not one?
  - — Can just generate scale one RV, then multiply by theta to get a scale-theta RV
- What if shape you want exceeds one?
  - — Turns out if X_1 and X_2 are gamma RV with same scale
  - — And X_1, has shape k_1, X_2 has shape k_2
  - — Then $X_1 + X_2$ is a gamma RV with shape $k_1 + k_2$
  - — Sooooo...
  - — To generate gamma RV with $k > 1$...
  - — Let j = floor(k)
  - — Generate and sum j gamma RVs with shape 1, and one more with shape (k - j)

The Dirichlet Distribution

- Now that we've done all of that...
  - — 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

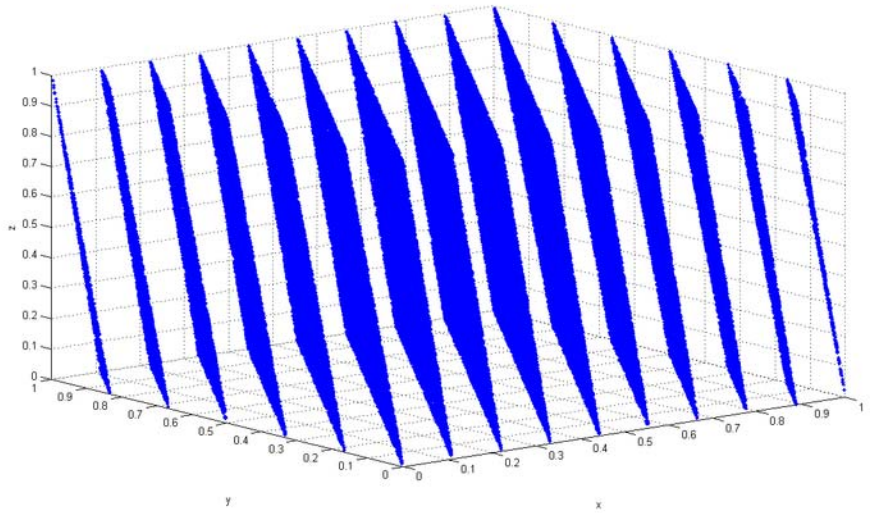
- All of this depends on being able to generate random doubles
  - — From zero to one
- Easy if you can generate a random sequence of bits
- But how to generate that sequence of bits? Use a PRNG!
- We’re not asking you to implement one...
  - — Java comes with an imp. of a classic PRNG: the “linear congruential”
- 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?