Lecture 4

Sampling:

Inverse Transform, Rejection Sampling, and Importance Sampling

Last Time:

- Expectations and some notation
- The Law of large numbers
- Simulation and Monte Carlo for Integration
- Sampling and the CLT
- Errors in Monte Carlo

Expectation $E_f[X]$

$$E_f X = \int x dF(x) = \begin{cases} \sum_x x f(x) & \text{if X is discrete} \\ \int x f(x) dx & \text{if X is continuous} \end{cases}$$

LOTUS, if $Y = r(X)$:

$$E[Y] = \int r(x) dF(x)$$

If $r(X) = I_A(X)$, Indicator for event A, $p(X \in A) = E_F[I_A(X)] =$ frequentist probability

Law of Large numbers (LLN)

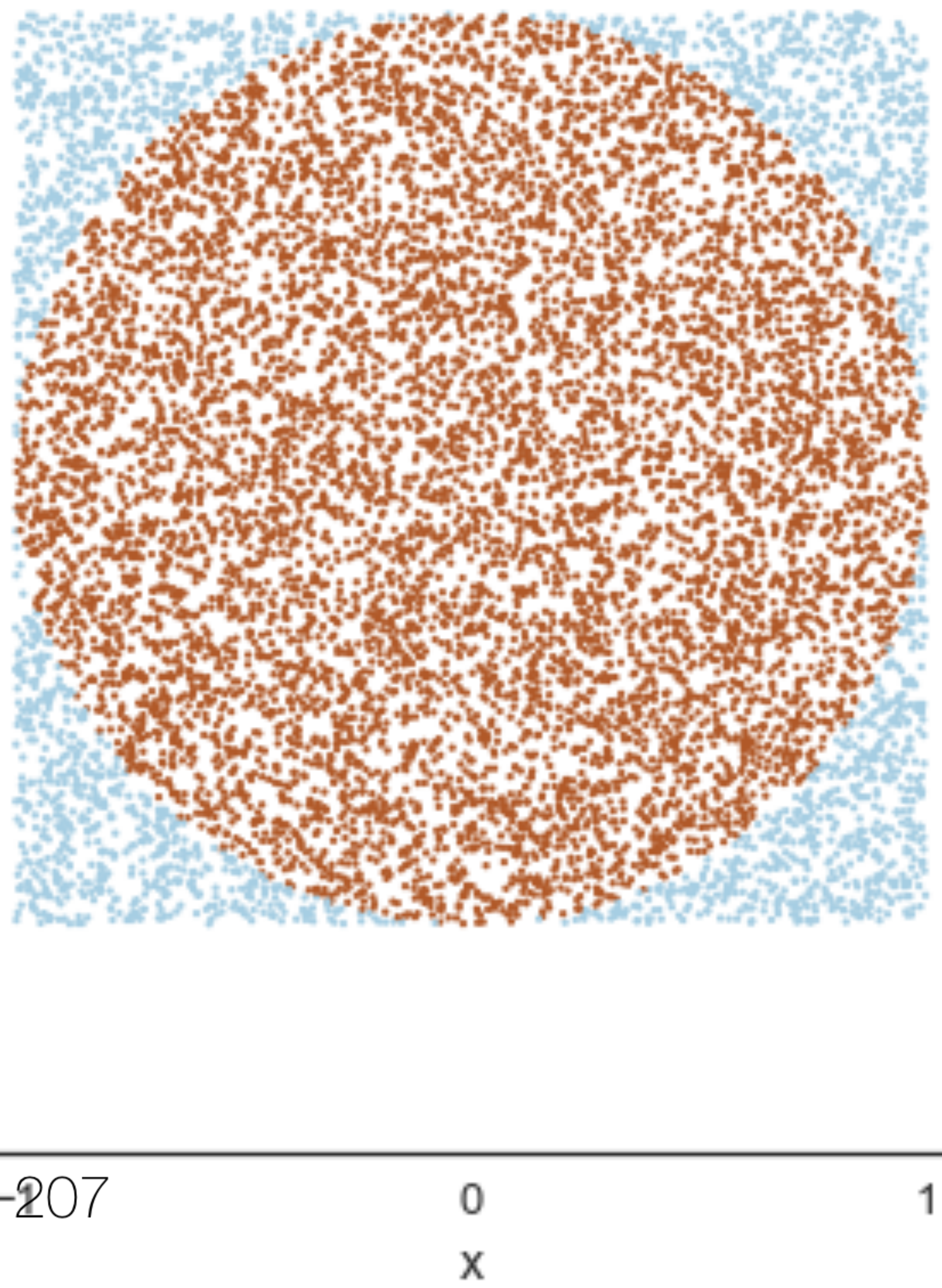
- Expectations become sample averages. Convergence for large N.

$$
\begin{aligned}
E_f[g] &= \int g(x) dF = \int g(x) f(x) dx \\
&= \lim_{n \to \infty} \frac{1}{N} \sum_{x_i \sim f} g(x_i)
\end{aligned}
$$

- foundation of Monte Carlo techniques for expectations and integrals, which allow us to replace integration with summation

Central Limit Theorem

- note that we compute integrals from samples in one replication
- the sample averages are distributes around the true (distribution) expectation in a gaussian distribution with standard error $$s = \frac{\sigma}{\sqrt{n}}$$
- which mean to use depends on the accuracy you desire



Monte Carlo π

- LLN says throw rocks to compute expectation below
- $E_f[I_{\in C}(X,Y)] = \int\int_{\in C} f_{X,Y}(x,y)dxdy$
- which is probability of being in C
- If $f_{X,Y}(x,y) \sim Uniform(V)$:

$$= \frac{1}{V}\int\int_{\in C} dxdy = \frac{A}{V}$$

Formalize Monte Carlo Integration idea

For Uniform pdf: $U_{ab}(x) = 1/V = 1/(b-a)$

$$J = \int_a^b f(x) U_{ab}(x)\, dx = \int_a^b f(x)\, dx / V = I/V$$

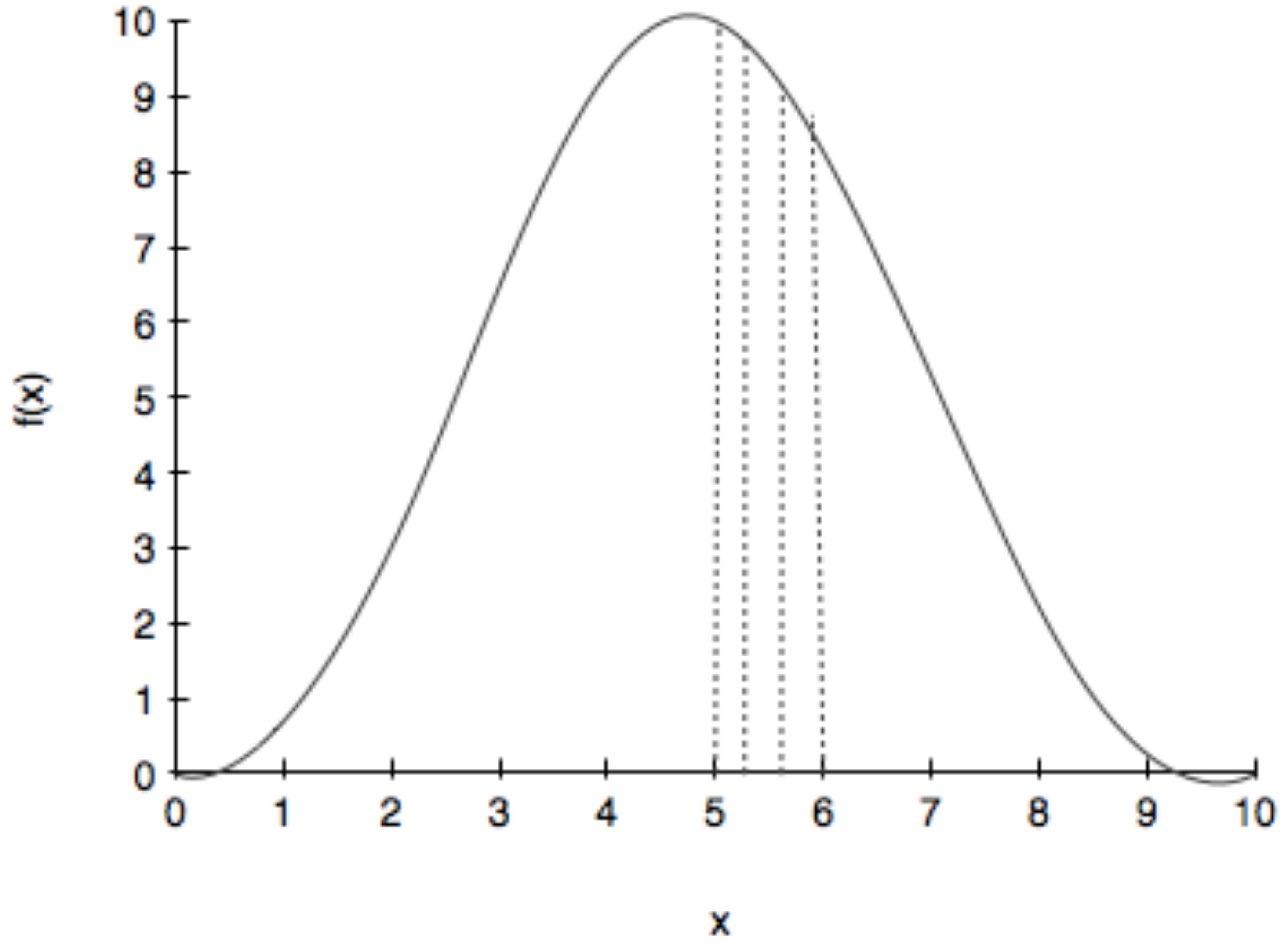
From LOTUS and the law of large numbers:

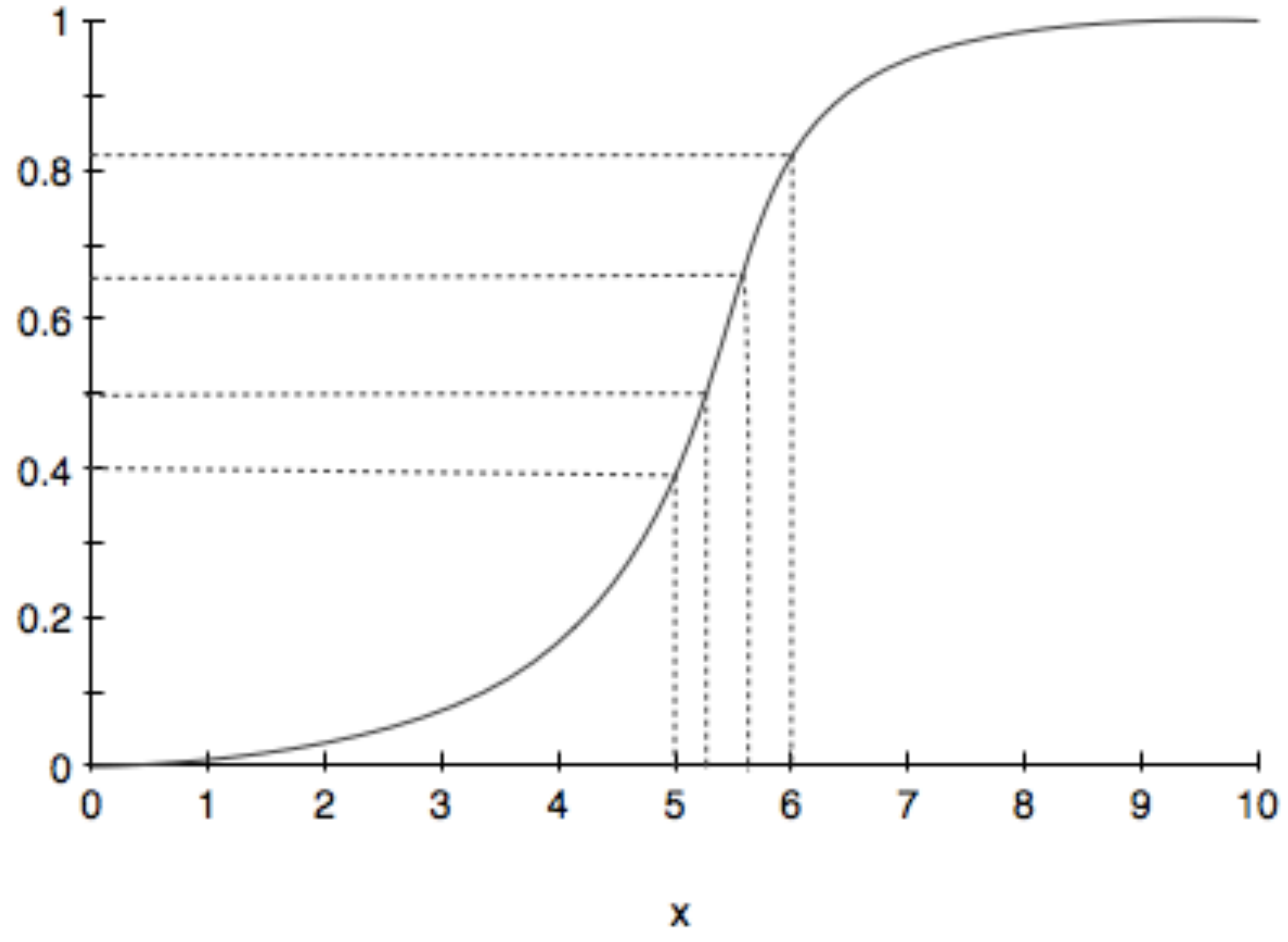
$$I = V \times J = V \times E_U[f] = V \times \lim_{n \to \infty} \frac{1}{N} \sum_{x_i \sim U} f(x_i)$$

Today: We need Samples

- to compute expectations, integrals and do statistics, we need samples
- we start that journey today
- inverse transform
- rejection sampling
- importance sampling: a direct, low-variance way to do integrals and expectations

Inverse transform





algorithm

The CDF F must be invertible!

1. get a uniform sample u from $Unif(0,1)$
2. solve for x yielding a new equation $x = F^{-1}(u)$ where F is the CDF of the distribution we desire.
3. repeat.

Why does it work?

$$F^{-1}(u) = \text{smallest x such that } F(x) >= u$$

What distribution does random variable $y = F^{-1}(u)$ follow?

The CDF of y is $p(y <= x)$. Since F is monotonic:

$$p(y <= x) = p(F(y) <= F(x)) = p(u <= F(x)) = F(x)$$

F is the CDF of y, thus f is the pdf.

Example: exponential

pdf: $f(x) = \frac{1}{\lambda} e^{-x/\lambda}$ for $x \geq 0$ and $f(x) = 0$ otherwise.

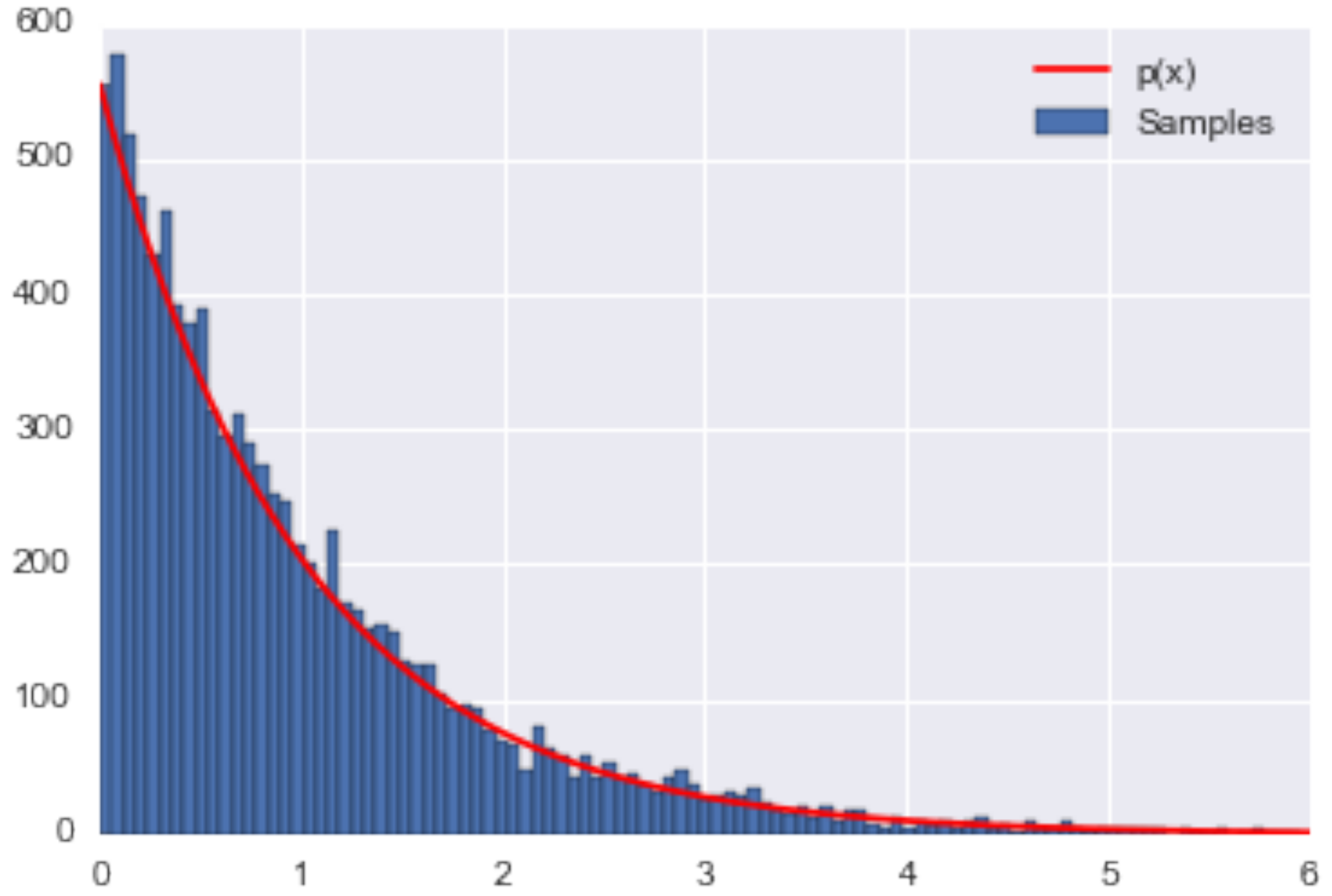
$$u = \int_0^x \frac{1}{\lambda} e^{-x'/\lambda} dx' = 1 - e^{-x/\lambda}$$

Solving for x

$$x = -\lambda \ln(1 - u)$$

code

```
p = lambda x: np.exp(-x)
CDF = lambda x: 1-np.exp(-x)
invCDF = lambda r: -np.log(1-r) # invert the CDF
xmin = 0 # the lower limit of our domain
xmax = 6 # the upper limit of our domain
rmin = CDF(xmin)
rmax = CDF(xmax)
N = 10000
# generate uniform samples in our range then invert the CDF
# to get samples of our target distribution
R = np.random.uniform(rmin, rmax, N)
X = invCDF(R)
hinfo = np.histogram(X,100)
plt.hist(X,bins=100, label=u'Samples');
# plot our (normalized) function
xvals=np.linspace(xmin, xmax, 1000)
plt.plot(xvals, hinfo[0][0]*p(xvals), 'r', label=u'p(x)')
plt.legend()
```



Box-Muller

- how to draw from a normal?
- the CDF integral is not analytically solvable.

$$I = \frac{1}{2\pi} \int_{-\infty}^{x} e^{-x'^2/2} dx'$$

- can do numerical inversion (out of scope) or use box-muller trick.

-trick involves starting with two Normals $N(0,1)$

$$X \sim N(0,1), Y \sim N(0,1) \implies X, Y \sim N(0,1)N(0,1)$$

pdf:

$$f_{XY}(x,y) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \times \frac{1}{\sqrt{2\pi}} e^{-y^2/2} = \frac{1}{2\pi} \times e^{-r^2/2}$$

where $r^2 = x^2 + y^2$.

Using polar co-ordinates r and θ, we have...

$$\Theta \sim Unif(0, 2\pi), S = R^2 \sim Exp(1/2)$$

$$s = r^2 = -2ln(1-u)$$

$$r = \sqrt{-2\,ln(u_1)}, \theta = 2\pi\,u_2$$

where u_1 and $u_2 \sim Unif(0,1)$.

Now, use $x = r\,cos\theta, y = r\,sin\theta$ to obtain Normal samples.

What is $f_{R,\Theta}(r,\theta)$?

General transforms of a pdf

Let $z = g(x)$ so that $x = g^{-1}(z)$

Define the Jacobian $J(z)$ of the transformation $x = g^{-1}(z)$ as the partial derivatives matrix of the transformation.

Then:

$$f_Z(z) = f_X(g^{-1}(z)) \times det(J(z))$$

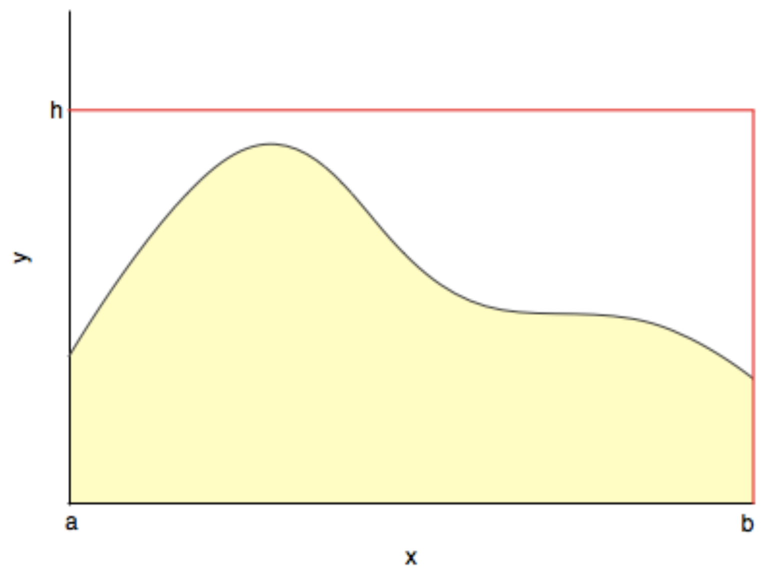
Let $g: r = \sqrt{x^2 + y^2}, tan(\theta) = y/x$. Then $g^{-1}: x = r\, cos(\theta)$, $y = r\, sin(\theta)$

$$J = \begin{pmatrix} cos(\theta) & sin(\theta) \\ -rsin(\theta) & rcos(\theta) \end{pmatrix}, det(J) = r$$

$$f_{R,\Theta}(r, \theta) = f_{X,Y}(rcos(\theta), rsin(\theta)) \times r$$

$$= \frac{1}{\sqrt{2\pi}} e^{-(rcos(\theta))^2/2} \times \frac{1}{\sqrt{2\pi}} e^{-(rsin(\theta))^2/2} = \frac{1}{2\pi} \times e^{-r^2/2} \times r.$$

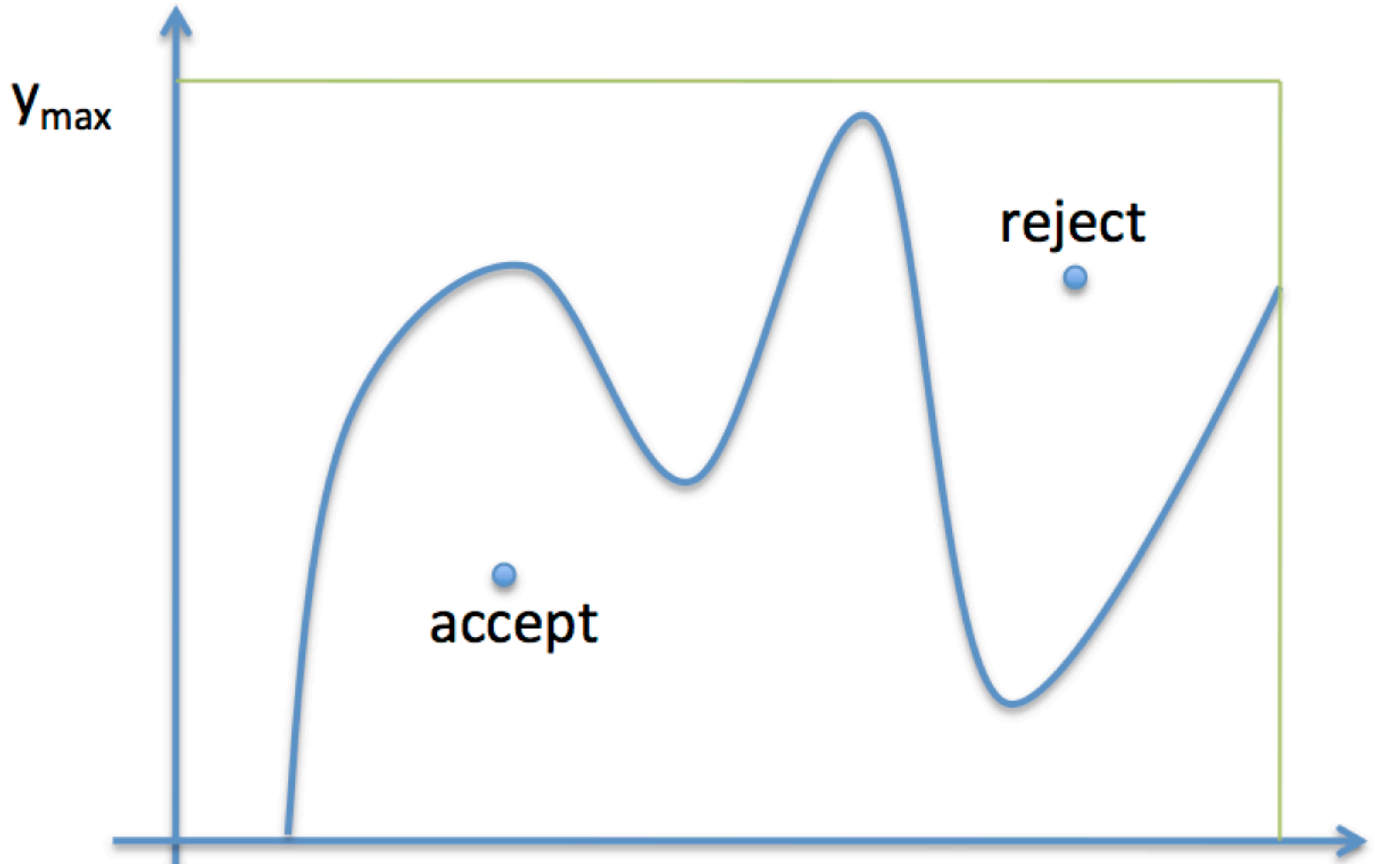
Rejection Sampling



- Generate samples from a uniform distribution with support on the rectangle
- See how many fall below $y(x)$ at a specific x.

Algorithm

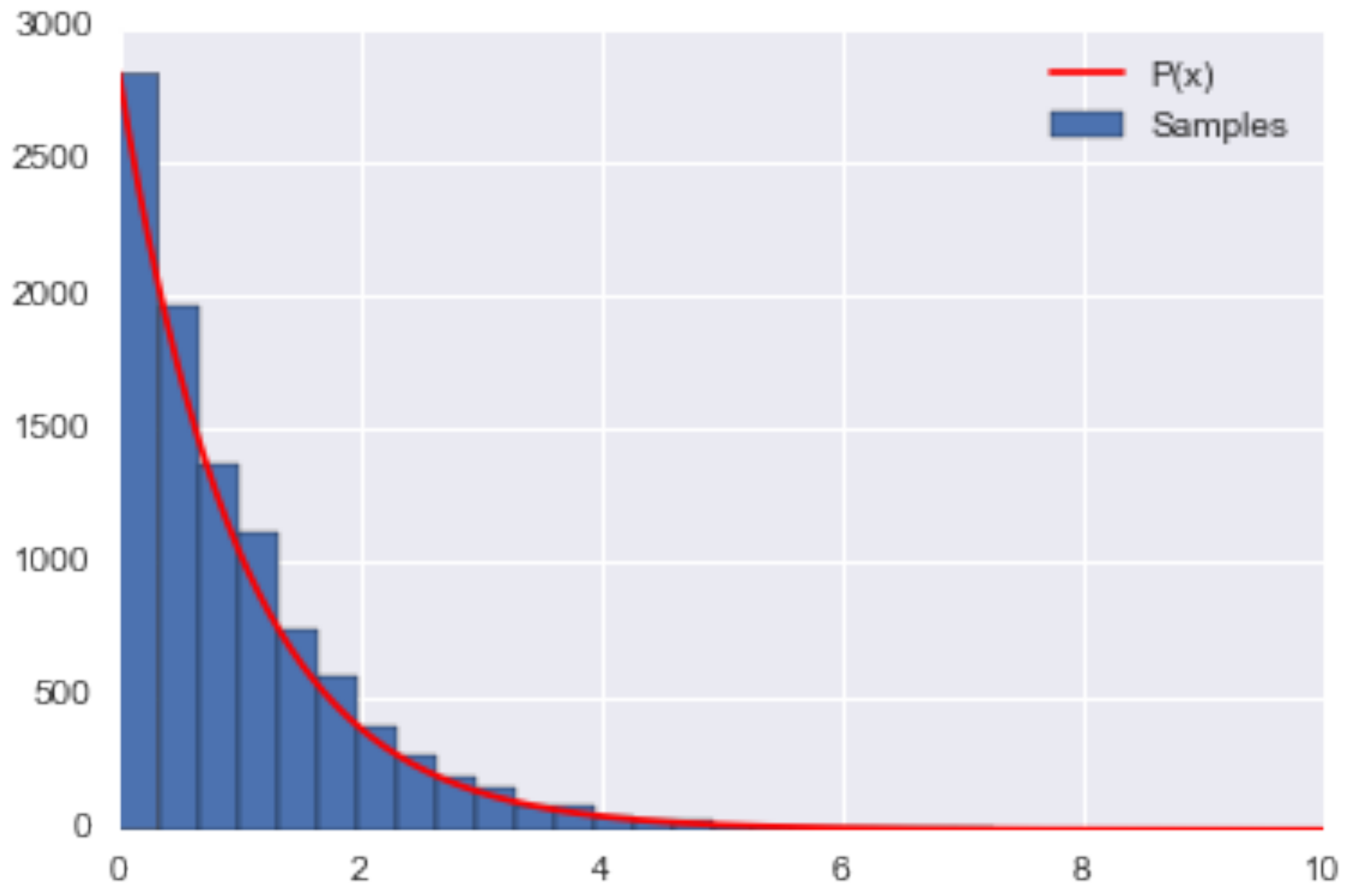
1. Draw x uniformly from $[x_{min}, \, x_{max}]$
2. Draw y uniformly from $[0, \, y_{max}]$
3. if $y < f(x)$, accept the sample
4. otherwise reject it
5. repeat



example

```
P = lambda x: np.exp(-x)
xmin = 0 # the lower limit of our domain
xmax = 10 # the upper limit of our domain
ymax = 1
#you might have to do an optimization to find this.
N = 10000 # the total of samples we wish to generate
accepted = 0 # the number of accepted samples
samples = np.zeros(N)
count = 0 # the total count of proposals

while (accepted < N):
    # pick a uniform number on [xmin, xmax) (e.g. 0...10)
    x = np.random.uniform(xmin, xmax)
    # pick a uniform number on [0, ymax)
    y = np.random.uniform(0,ymax)
    # Do the accept/reject comparison
    if y < P(x):
        samples[accepted] = x
        accepted += 1

    count +=1

print("Count",count, "Accepted", accepted)
hinfo = np.histogram(samples,30)
plt.hist(samples,bins=30, label=u'Samples');
xvals=np.linspace(xmin, xmax, 1000)
plt.plot(xvals, hinfo[0][0]*P(xvals), 'r', label=u'P(x)')
plt.legend()
```

```
Count 100294 Accepted 10000
```



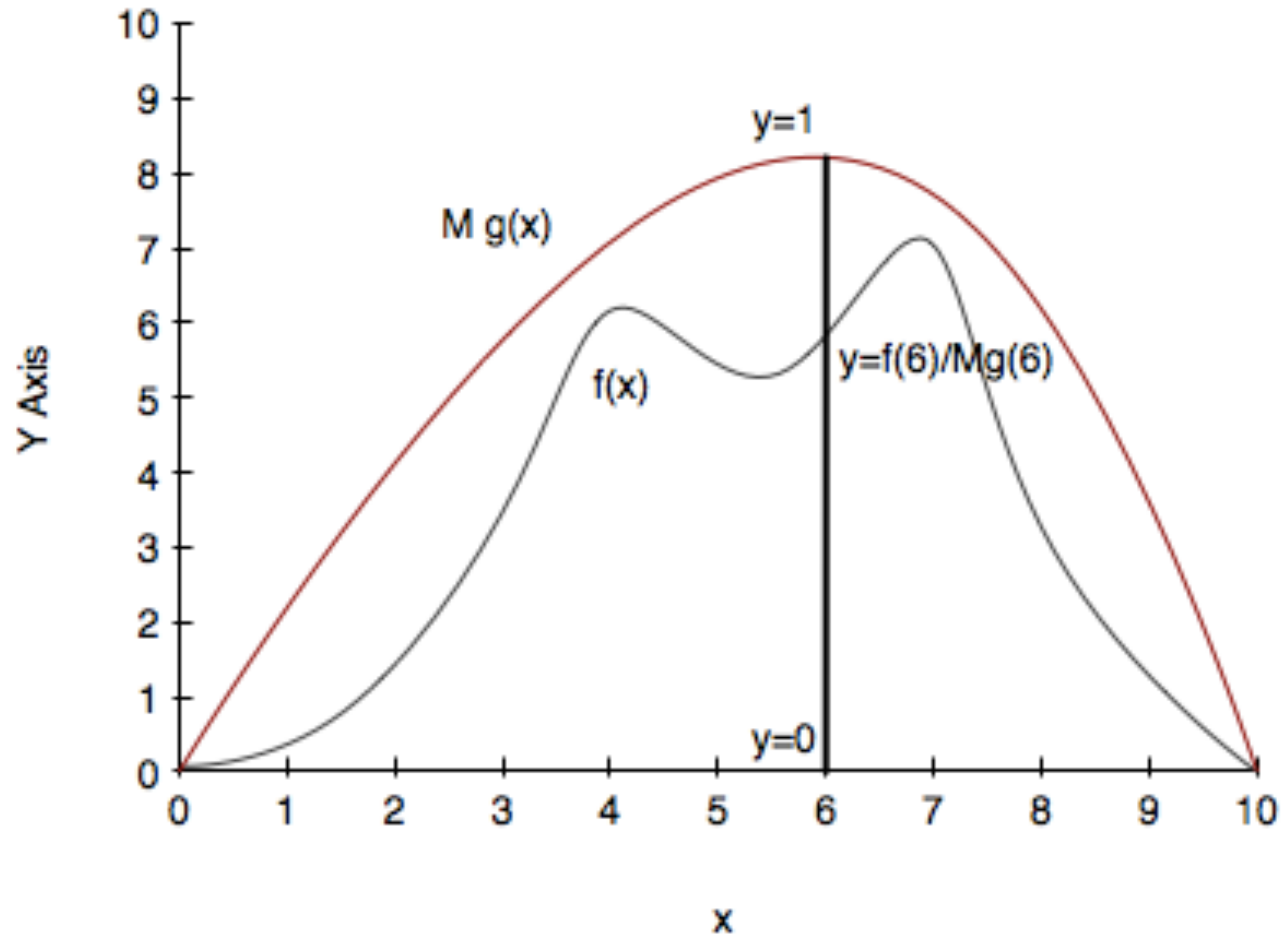
problems

- determining the supremum may be costly
- the functional form may be complex for comparison
- even if you find a tight bound for the supremum, basic rejection sampling is very inefficient: **low acceptance probability**
- infinite support

Rejection on steroids

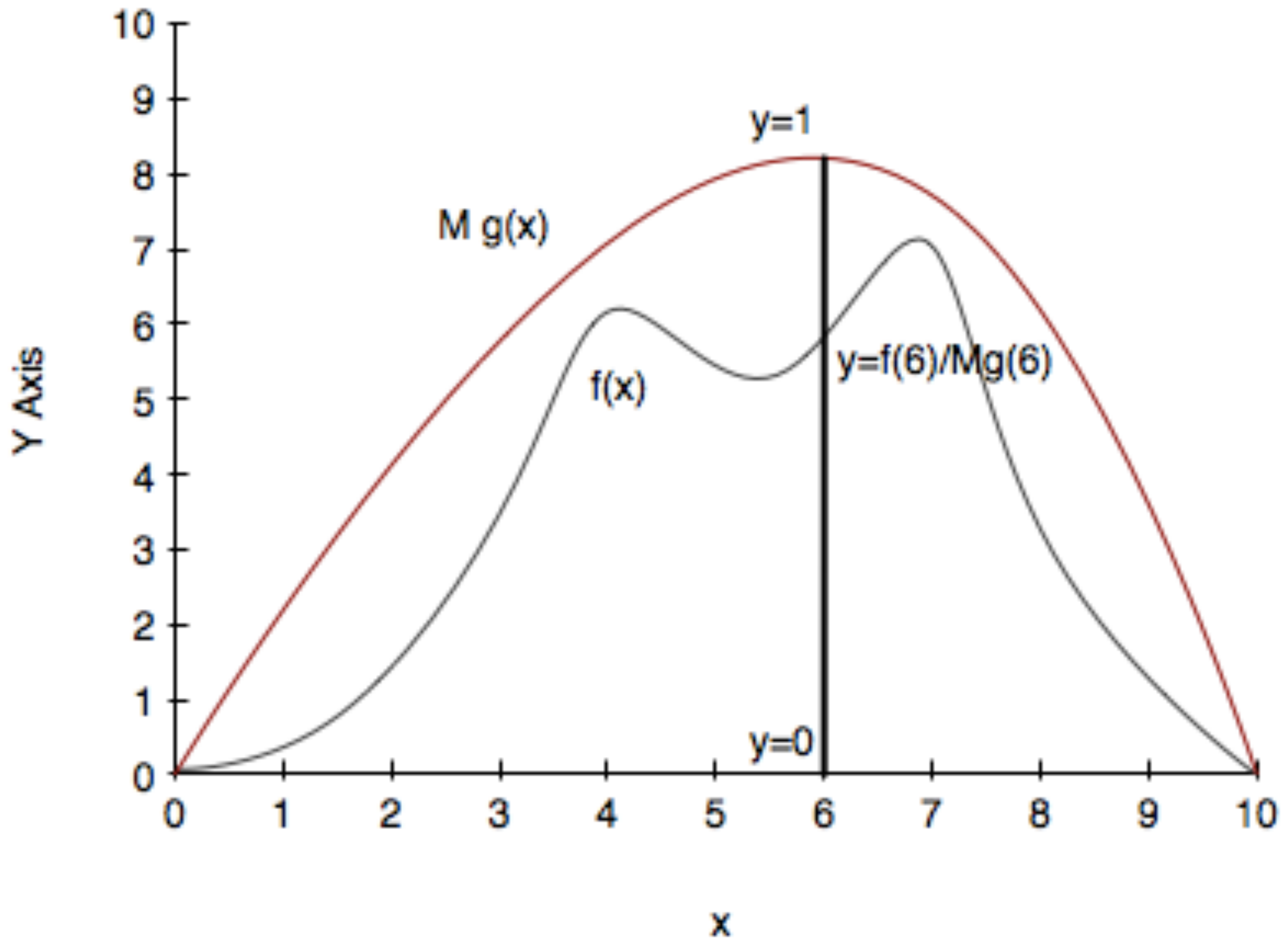
Introduce a **proposal density** $g(x)$.

- $g(x)$ is easy to sample from and (calculate the pdf)
- Some M exists so that $M\, g(x) > f(x)$ in your entire domain of interest
- ideally $g(x)$ will be somewhat close to f
- optimal value for M is the supremum over your domain of interest of f/g.
- probability of acceptance is $1/M$



Algorithm

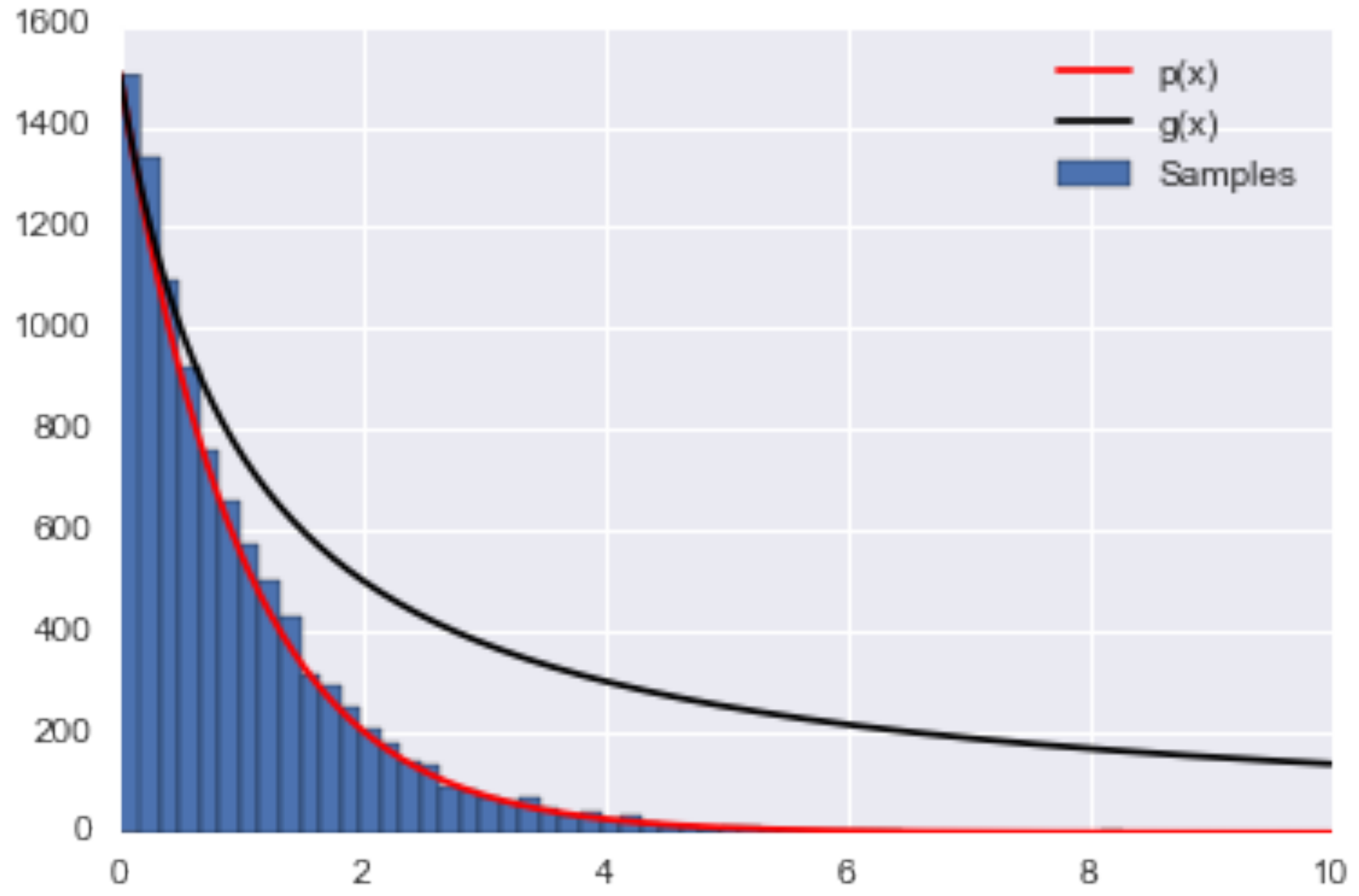
1. Draw x from your proposal distribution $g(x)$
2. Draw y uniformly from [0,1]
3. if $y < f(x)/M\, g(x)$, accept the sample
4. otherwise reject it
5. repeat



Example

```python
p = lambda x: np.exp(-x)  # our distribution
g = lambda x: 1/(x+1)  # our proposal pdf (we're thus choosing M to be 1)
invCDFg = lambda x: np.log(x +1) # generates our proposal using inverse sampling
xmin = 0 # the lower limit of our domain
xmax = 10 # the upper limit of our domain
# range limits for inverse sampling
umin = invCDFg(xmin)
umax = invCDFg(xmax)
N = 10000 # the total of samples we wish to generate
accepted = 0 # the number of accepted samples
samples = np.zeros(N)
count = 0 # the total count of proposals

while (accepted < N):

    # Sample from g using inverse sampling
    u = np.random.uniform(umin, umax)
    xproposal = np.exp(u) - 1

    # pick a uniform number on [0, 1)
    y = np.random.uniform(0,1)

    # Do the accept/reject comparison
    if y < p(xproposal)/g(xproposal):
        samples[accepted] = xproposal
        accepted += 1

    count +=1

print("Count", count, "Accepted", accepted)
# get the histogram info
hinfo = np.histogram(samples,50)
plt.hist(samples,bins=50, label=u'Samples');
xvals=np.linspace(xmin, xmax, 1000)
plt.plot(xvals, hinfo[0][0]*p(xvals), 'r', label=u'p(x)')
plt.plot(xvals, hinfo[0][0]*g(xvals), 'k', label=u'g(x)')
plt.legend()
```

```
Count 23809 Accepted 10000
```



Importance sampling

The basic idea behind importance sampling is that we want to draw more samples where $h(x)$, a function whose integral or expectation we desire, is large. In the case we are doing an expectation, it would indeed be even better to draw more samples where $h(x)f(x)$ is large, where $f(x)$ is the pdf we are calculating the integral with respect to.

Unlike rejection sampling we use all samples!!

$$E_f[h] = \int_V f(x)h(x)dx.$$

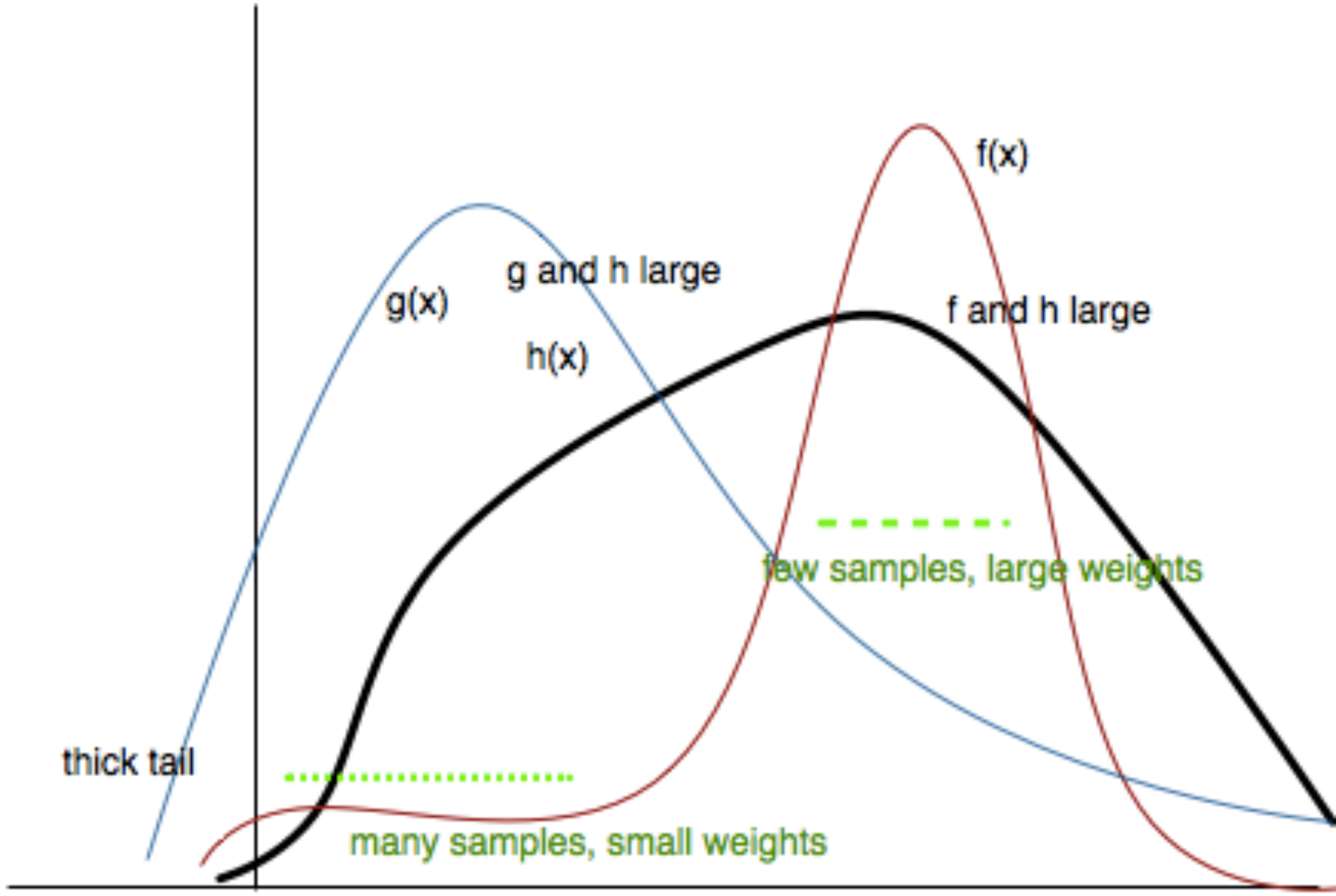
Choosing a proposal distribution $g(x)$:

$$E_f[h] = \int h(x)g(x)\frac{f(x)}{g(x)}dV$$

$$E_f[h] = \lim_{N\to\infty}\frac{1}{N}\sum_{x_i\sim g(.)} h(x_i)\frac{f(x_i)}{g(x_i)}$$

If $w(x_i) = f(x_i)/g(x_i)$:

$$E_f[h] = \lim_{N\to\infty}\frac{1}{N}\sum_{x_i\sim g(.)} w(x_i)h(x_i)$$



Variance reduction

Usually: $\hat{V} = \frac{V_f[h(x)]}{N}$

Importance Sampling: $\hat{V} = \frac{V_g[w(x)h(x)]}{N}$

Minimize $V_g[w(x)h(x)]$ (make 0), if:

$$w(x)h(x) = C \implies f(x)h(x) = Cg(x),...$$

Gives us $g(x) = \frac{f(x)h(x)}{E_f[h(x)]}$

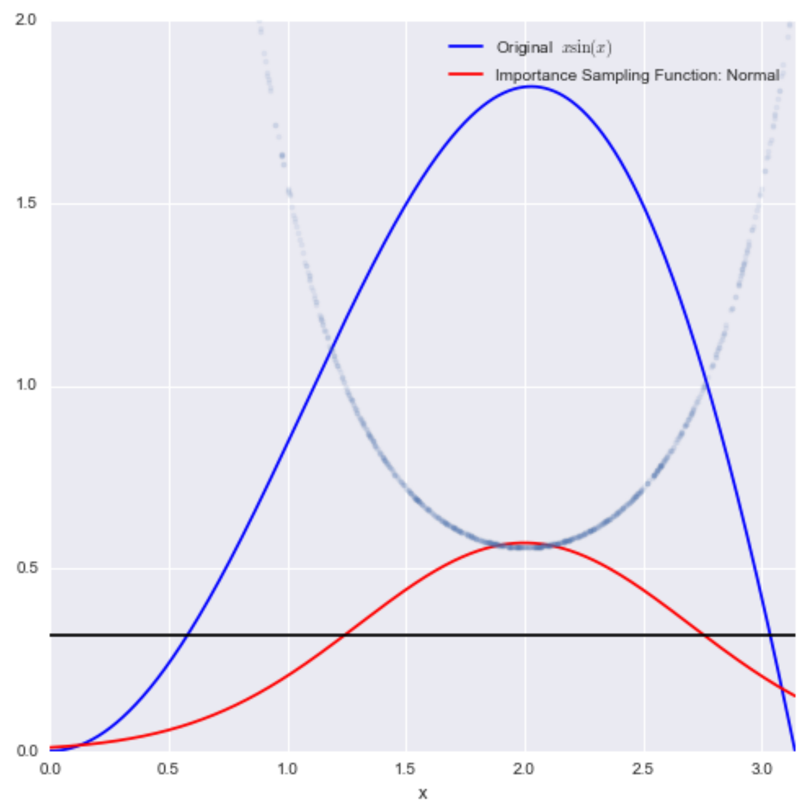
To get low variance, we must have $g(x)$ **large where the product $f(x)h(x)$ is large**.

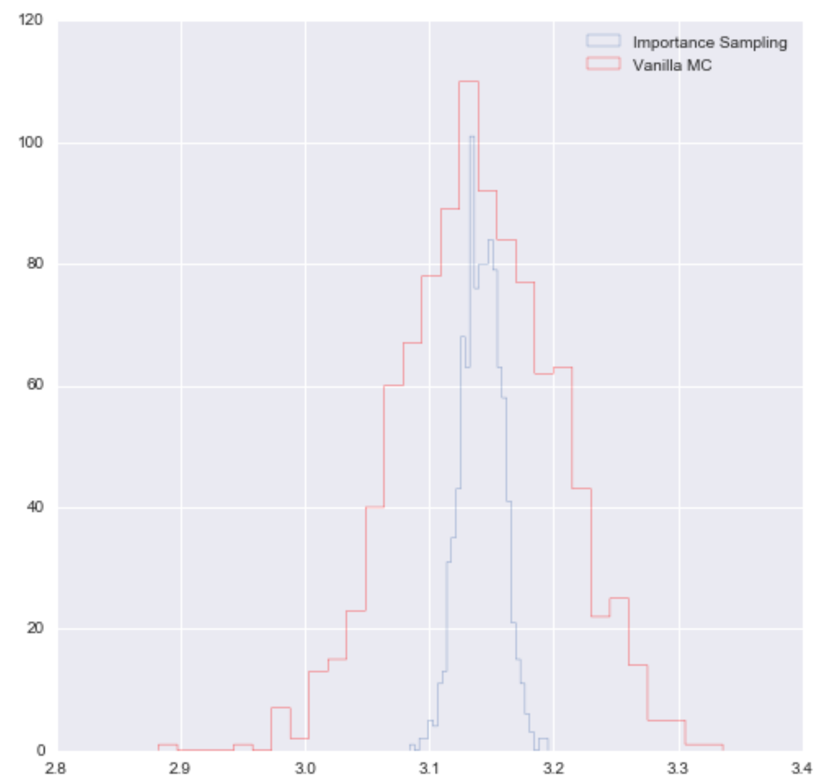
Or, $\frac{g(x)}{f(x)}$ ought to be large where $h(x)$ is large. This means that, as we said earlier, choose more samples near the peak.

Example: integral of x sin(x)

```
mu = 2;
sig =.7;
f = lambda x: np.sin(x)*x
infun = lambda x: np.sin(x)-x*np.cos(x)
p = lambda x: (1/np.sqrt(2*np.pi*sig**2))*np.exp(-(x-mu)**2/(2.0*sig**2))
normfun = lambda x:  norm.cdf(x-mu, scale=sig)
# range of integraion
xmax =np.pi
xmin =0
N =1000 # Number of draws

# Just want to plot the function
x=np.linspace(xmin, xmax, 1000)
plt.plot(x, f(x), 'b', label=u'Original  $x\sin(x)$')
plt.plot( x, p(x), 'r', label=u'Importance Sampling Function: Normal')
plt.plot(x, np.ones(1000)/np.pi,'k')
xis = mu + sig*np.random.randn(N,1);
plt.plot(xis, 1/(np.pi*p(xis)),'.', alpha=0.1)

# IMPORTANCE SAMPLING
Iis = np.zeros(1000)
for k in np.arange(0,1000):
    # DRAW FROM THE GAUSSIAN mean =2 std = sqrt(0.4)
    xis = mu + sig*np.random.randn(N,1);
    xis = xis[ (xis<xmax) & (xis>xmin)] ;
    # normalization for gaussian from 0..pi
    normal = normfun(np.pi)-normfun(0);
    Iis[k] =np.mean(f(xis)/p(xis))*normal;
```

```
Exact solution is:   3.14159265359
Mean basic MC estimate:   3.14068341144
Standard deviation of our estimates:  0.0617743877206
Mean importance sampling MC estimate:  3.14197268362
Standard deviation of our estimates:  0.0161935244302
```


Original $x\sin(x)$
Importance Sampling Function: Normal
x


Importance Sampling
Vanilla MC