# The Monty Python Method for Generating Gamma Variables

George Marsaglia[1]
The Florida State University
and
Wai Wan Tsang
The University of Hong Kong

**Summary**

The Monty Python Method for generating random variables takes a decreasing density, cuts it into three pieces, then, using area-preserving transformations, folds it into a rectangle of area 1. A random point $(x, y)$ from that rectangle is used to provide a variate from the given density, most of the time as $x$ itself or a linear function of $x$. The decreasing density is usually the right half of a symmetric density.

The Monty Python method has provided short and fast generators for normal, t and von Mises densities, requiring, on the average, from 1.5 to 1.8 uniform variables. In this article, we apply the method to non-symmetric densities, particularly the important gamma densities. We lose some of the speed and simplicity of the symmetric densities, but still get a method for $\gamma_\alpha$ variates that is simple and fast enough to provide beta variates in the form $\gamma_a/(\gamma_a + \gamma_b)$. We use an average of less than 1.7 uniform variates to produce a gamma variate whenever $\alpha \geq 1$. Implementation is simpler and from three to five times as fast as a recent method reputed to be the best for changing $\alpha$'s.

[1]Research supported by the National Science Foundation

# 1 Introduction

We will provide a summary of the Monty Python method here, and then show how it can be applied to provide a method for generating gamma variates for all values of the gamma parameter. The resulting algorithm is simpler and faster than any we are aware of—indeed, simple and fast enough that it can reasonably serve to provide beta variates in the form $\gamma_a / (\gamma_a + \gamma_b)$. Comments on complexity and speed are in Section 6.

The Monty Python method, developed years ago but only recently published in a journal [3], takes a decreasing sigmoid density and folds it into a rectangle of area 1, in such a way that a random point $(x, y)$ from the rectangle can easily provide a variate from that density—most of the time as $x$ itself or as a linear function of $x$. Here is a picture of such a rectangle for the half-normal density:

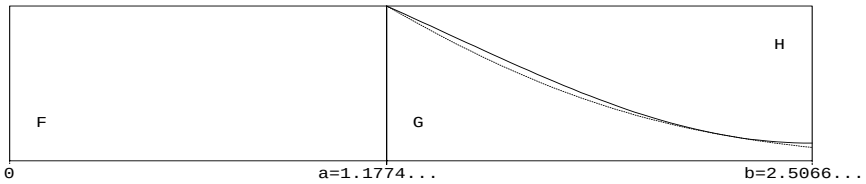


Figure 1: The Monty Python Method applied to the Normal Distribution.

The rectangle has area 1. If a random point $(x, y)$ falls in region F, return a standard normal variate as $\pm x$, if in region G, return $\pm x$, if in region H, return $\pm .88579(b - x)$, and if in the narrow in-between region (1%), return $\pm$ a variate from the normal tail. Note that the presence of $(x, y)$ in region F can be determined without $y$, so about half the time, a normal variate is returned after generating a single uniform variate and a test on magnitude.

To apply the Monty Python method, one chooses a rectangle of area 1, base $0 < x < b$, with $b$ chosen as large as possible, subject to the condition that the 'cap', the portion of the density above the rectangle, can be rotated and stretched so as to fit in the upper-right corner of the rectangle, as shown in Figure 2. Since

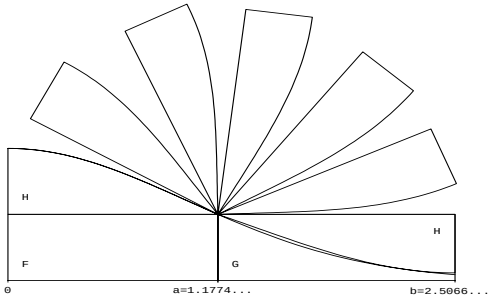


Figure 2: Rotating and stretching the cap.

we require area-preserving transformations, in 'stretching' the cap we must also

'shrink' it vertically, so that area is preserved. And since areas of the rectangle add to 1, the slim in-between area is exactly the tail area. Packing a density into a rectangle in this way leads to relatively simple and fast procedures for generating random variables. Details and examples for normal, t, and von Mises densities are in [3].

We called this the Monty Python method because, at the time it was developed, a British TV program of that name was being shown in the US, and opening graphics on the program had a stylized head with its top opened and folded over with all sorts of silly images pouring out. And since our research program had investigated many different methods, we needed names to identify them. We also referred to it as the *patchwork method*, (see, for example, Tsang [5]), a term subsequently used by others.

## 2 The Monty Python method for gamma variates

In order to apply the method to generate a gamma variate $\gamma_\alpha$, we use the exact-approximation method of Marsaglia [2], writing $\gamma_\alpha = q(X)$, where $q$ is a monotone function of $X$, and the density of $X$ is such that $q(X)$ is exactly a $\gamma_\alpha$ variate. That density must be $f(x) = q'(x)q(x)^{\alpha-1}e^{-q(x)}/\Gamma(\alpha)$, and we hope to chose $q$ such that $f(x)$ is not necessarily close to a normal density, but rather, is close enough to a symmetric density that we may apply the Monty Python method to it.

So our strategy is: generate a variate $X$ from our nearly symmetric density $f(x)$ by means of the Monty Python method, then return $q(X)$ as the required $\gamma_\alpha$ variate, for any $\alpha \geq 1$. For the rare but difficult case $\alpha < 1$, we boost the parameter to $\alpha + 1$ by means of a method of Marsaglia that goes back to 1961: generate $\gamma_\alpha$ as $\gamma_{\alpha+1}U^{1/\alpha}$, with $U$ independent uniform. See Section 5.

This $q$ works remarkably well for all $\alpha \geq 1$:

$$q(x) = (\alpha - 1/3)(1 + x/\sqrt{16\alpha})^3, \text{ with } -\sqrt{16\alpha} < x < \infty.$$

For that $q$, the resulting density $f(x) = q'(x)q(x)^{\alpha-1}e^{-q(x)}/\Gamma(\alpha)$ is very nearly symmetric. Furthermore, for that choice of $q(x)$, a single rectangle of area 1: $-3.2 < x < 3.2,\ 0 < y < .15625$, may be used to apply the Monty Python method, providing (total) tail areas starting at 2.5% for $\alpha = 1$ and quickly going to less than 2%. Figure 3 shows $f(x)$ for $\alpha = 1, 2, 4, 8$ and the common rectangle for applying the Monty Python method.

## 3 Monty Python for nearly symmetric densities, $\alpha \geq 1$.

We find we are able to apply the Monty Python method only for $\alpha \geq 1$. For $\alpha < 1$ we must to resort to generating a $\gamma_\alpha$ variate as $\gamma_{\alpha+1}U^{1/\alpha}$. If $\alpha \geq 1$ is fixed and we have time to set up the appropriate values, we can generate from a nearly symmetric

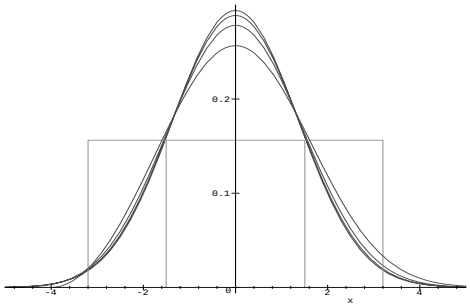


Figure 3: The density $f(x)$ for $\alpha = 1, 2, 4, 8$, with common rectangle.

density by applying the Monty Python method to each side of the density, with virtually the same speed as for symmetric densities. But that requires knowing the exact probabilities for each half, and finding the best rectangle for each half. Since we want our generator to be able to provide $\gamma_\alpha$ variates with $\alpha$ possibly changing from call to call, we cannot afford the luxury of a long, once-only setup time.

Our solution is to choose a single rectangle, from $-3.2 < x < 3.2$, and to shift and stretch each of the caps into its corresponding corner. We cannot provide the maximum possible stretch—the one that makes the tail area as small as possible—but we find that we are able to provide a common rectangle base $-3.2 < x < 3.2$ and a stretch factor $s$ in such a way that the resulting generating procedure is simple and fast. The stretch factor is constant, $s = .94$, for all $\alpha > 2.6$ and a linear function for $1 < \alpha < 2.6$.

The worst case is at $\alpha = 1$, and even that one is pretty good. Here is a picture of the situation:

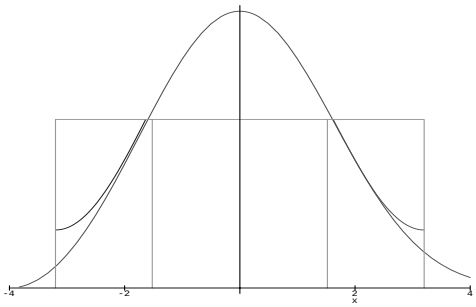


Figure 4: Monty Python for $f(x)$ with $\alpha = 1$.

The density $f(x)$ for $\alpha = 1$ is drawn, with the rectangle $-3.2 < x < 3.2$ (and the universal bounds $|x| < 1.52$, that save need for a random $y$). The caps are rotated and stretched by the stretch factor $s = 1.02$, each rotated and stretched cap put into its corresponding corner. The curves representing the rotated and stretched caps are given by

Left cap: $3.2(1 + s) - sf(s(-3.2 - x)),\ x < 0,$

Right cap: $3.2(1+s) - sf(s(+3.2-x)),\ x>0$,
so that a single formula applies by attaching the sign of $x$ to 3.2.

If Figure 4 is enlarged enough, it becomes clear that the left cap is not quite rotated and stretched in an optimal way; there is a narrow gap that could be closed with a better $s$ for the left side. But the simplicity of $s = 1.02$ for each side in the resulting algorithm is well worth that slight increase in tail area.

Note that for even this worst case, the probability that a variate must be returned from one of the tails is .024. This probability rapidly goes to less than .02 as $\alpha$ increases.

Plots of $f(x)$, the rectangle $-3.2 < x < 3.2$ and the rotated and stretched caps look much the same for $\alpha > 1$, except that the tail areas get smaller. If we vary the unit rectangle on which the Monty Python method is based, as a function of $\alpha$, we find that we can squeeze the tail areas down to nearly 1%. But that compromises the simplicity of using a common rectangle, $-3.2 < x < 3.2$. We have chosen to use that common rectangle, for which the tails are around 2%. As we shall see, providing tail variates is not a very expensive procedure.

We will describe the tail procedure in the next section. Assuming that, we may put our gamma generator, for any $\alpha \geq 1$, in the following simple form, with VNI, UNI representing, resp., uniform variables from
(-1,1) or (0,1) and $q(x) = (\alpha - \frac{1}{3})(1+x/\sqrt{16\alpha})^3$, $f(x) = q'(x)q(x)^{a-1}e^{-q(x)}/\Gamma(\alpha)$, and $s = .94$ if $\alpha > 2.6$, else $.81 + .84/\sqrt{16\alpha}$:

```
x=3.2*VNI
if |x|<1.52 return q(x)
y=.15625*UNI
if y < f(x) return q(x)
z=s*(sign(b,x)-x)
if y >.15625*(1+s)-s*f(z) return q(z)
return a tail variate
```

Those seven lines summarize the generating procedure. Implemented directly, they provide a very concise gamma generator that is still quite fast. And it can be made very fast if quadratic pretests are inserted to avoid, most of the time, evaluation of $f(x)$ in step 4 or $f(z)$ in step 6.

For evaluation, $f$ takes the form

$$f(x) = e^{(3\alpha-1)\ln(1+x/\sqrt{16\alpha})-(\alpha-\frac{1}{3})(1+x/\sqrt{16\alpha})^3+c},$$

where $c = \alpha\ln(\alpha - \frac{1}{3}) + \ln(3/4) - .5\ln(\alpha) - \ln(\Gamma(a))$. The constant $c$ requires a loggamma routine unless, as we have done, one evaluates it directly to the required accuracy by means of polynomial approximations, as $c$ is very nearly a linear function of $\alpha$ with asymptotic slope 1: $c = \alpha - 1.53995 - 5/(36\alpha) - 1/(81\alpha^2) - 1/(3240\alpha^3) - 1/(1215\alpha^4) + \cdots$. Of course, with quadratic pretests, $c$ is needed only if the random point $(x, y)$ falls between $f(x)$ or $f(z)$ and its quadratic approximation—perhaps 1% of the time. It is not required at all in the tails.

# 4 Sampling from the tails.

When the random point $(x, y)$ from the Monty Python rectangle falls into one of the two 'tail' regions, we must provide an $x$ from one of the two tails. But we don't know which one, since the area of each region is not exactly the probability for its corresponding tail. So we draw the two tails, side by side from zero, as in Figure 5, showing the tails and bounding exponential functions for $\alpha = 2$.

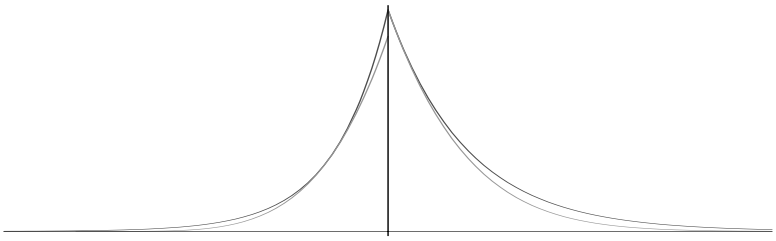

Figure 5: The two tails for $\alpha = 2$, with bounding exponential functions.

We generate a point uniformly from under the two exponential curves until we get one that lies under one of the tail curves, then return our required gamma variate as $q(3.2 + x)$ or $q(-3.2 + x)$, depending on which tail provides the random point. That will ensure that the left or right tail variates are returned with the proper frequencies.

The two tail densities are $f(-3.2+x)/f(3.2), x < 0$ and $f(3.2+x)/f(3.2), x > 0$, standardized at $x = 0$ by dividing both tails by $f(3.2)$. They can then be bounded by curves of the form $e^{c_1 x}$ and $e^{-c_2 x}$. The bounding exponential for the right tail is easy—it has slope that of the right tail: $-r$, where $r = t(3\alpha - 1)((1 + 3.2t)^2 - 1/(1 + 3.2t))$, $t = 1/\sqrt{16\alpha}$.

For the left tail, we use a bounding exponential curve with steeper slope. More numerical work shows that multiplying $r$ by a factor of $k = 124.237 + 206.86r + 117.08r^2 + 22.33r^3$ will provide a bounding exponential curve for the left tail. That is, the bounding exponential for the left tail has slope $kr$.

Thus we have this algorithm for the tails, with given $\alpha$ and bounds $\pm 3.2$ for the Monty Python rectangle, and with UNI representing the result of successive calls to a uniform [0,1) generator:
Put $r = t(3\alpha - 1)((1 + 3.2t)^2 - 1/(1 + 3.2t))$,
$k = 124.237 + 206.86r + 117.08r^2 + 22.33r^3$.
Generate $x = -\ln(\text{UNI})/r$ with probability $k/(1 + k)$ or
$x = \ln(\text{UNI})/(kr)$ with probability $1/(1 + k)$ until either
$x > 0$ and UNI$e^{-rx} < f(b + x)/f(b)$, with $q(b + x)$ returned, or
$x < 0$ and UNI$e^{-krx} < f(-b + x)/f(-b)$, with $q(-b + x)$ returned.

If the $y$ of the random point $(x, y)$ under the two exponential curves is expressed in the form $e^{\ln(\text{UNI})}$, then tests are determined by comparing exponents.

# 5 The case $\alpha < 1$.

For $\alpha < 1$ we have not found an easy $q(x)$ for which $q'(x)q(x)^{\alpha-1}e^{-q(x)}$ provides a family of nearly symmetric densities for the Monty Python method. So we rely on the fact that a $\gamma_\alpha$ variate can be expressed as the product of a $\gamma_{\alpha+1}$ variate and the $1/\alpha$ power of an independent uniform variate $U$: $\gamma_\alpha \sim \gamma_{\alpha+1}U^{1/\alpha}$. To prove this, take logarithms. The characteristic function of $\ln(\gamma_{\alpha+1})$ is $\Gamma(\alpha+1+it)/\Gamma(\alpha+1)$, and the characteristic function of $\ln(U)/\alpha$ is $\alpha/(\alpha+it)$. The product of those two is $\Gamma(\alpha+it)/\Gamma(\alpha)$, the characteristic function of $\ln(\gamma_\alpha)$.

# 6 Speed and complexity comparisons.

Many methods have been proposed for generating gamma variates. See Devroye's thorough treatment of it and other methods for non-uniform variates [1]. We used an article by Minh [4] as an example of the current best gamma generator when we undertook to see if the Monty Python method might be better. In examining that algorithm, there is little doubt that the Monty Python method leads to a much simpler implementation. As for speed, we find it is also much faster. But comparisons in speed must take into account, to name a few: PC or workstation, Fortran or C, F77 or F95, Microsoft or Borland or Lahey or gnu, optimizing compilers, the way that the necessary uniform variates are provided, which parts are inline, the overhead of subroutine calls, etc.

Nonetheless, for a variety of different elements taken into account, the Monty Python method seems far and away the best we know of. For example, on a fast Silicon Graphics machine, Minh's algorithm [4] averaged 5.234,4.324,5.572,5.473 microseconds for $\alpha = 1+\epsilon, 2, 4, 10$, while the Monty Python method averaged .92,1.02,.844,.814 for those $\alpha$'s. Those are the times without setups. (We must use $\alpha = 1+\epsilon$ for Minh, as it will not work for $\alpha = 1$.)

The times for the MP method are based on an implementation that supplements the simple 7-line algorithm of Section 3 with quadratic pretests to avoid evaluating $f$, and specific code for the tails.

Corresponding averages for repeated calls with $\alpha$ fixed and constants preassigned are: 1.52, .965, .891, .846 for Minh and .64, .637, .626, .622 for Monty Python. In all comparisons, we used a common uniform generator: floating `j=69069*j` to (0,1), produced inline to avoid the overhead of an additional subroutine call.

Another time comparison: on a 120 Mhz PC with Microsoft Fortran, the Minh algorithm took 29.2, 29.2, 28 microsecs for $\alpha = 3, 10, 500$, while corresponding times for Monty Python were 9.6,9.1,9.2.

And another: on an older Sun workstation, for various $\alpha$, Minh averaged 75.3 microsecs, compared to 29.6 for Monty Python.

For readers who may wish to use the Monty Python method, or compare it with others, Fortran and/or C versions are available from either of us, geo@stat.fsu.edu or tsang@cs.hku.hk.

# References

[1] Devroye, Luc (1986), *Non-Uniform Random Variate Generation*, Springer-Verlag, New York.

[2] Marsaglia, George (1984), The exact-approximation method for generating random variables, *Journal Amer. Statist. Assoc.*, **79**, 218–221.

[3] Marsaglia, George and Tsang, Wai Wan (1997) The Monty Python Method for generating random variables, *ACM Transactions on Mathematical Software*, in press.

[4] Minh, Do Le (1988), Generating gamma variates, *ACM Trans. on Math. Software* **14**, 261–266.

[5] Tsang, Wai Wan (1982), Computer generation of random variables, Ph. D. Dissertation, Dept. of Computer Science, Washington State University.