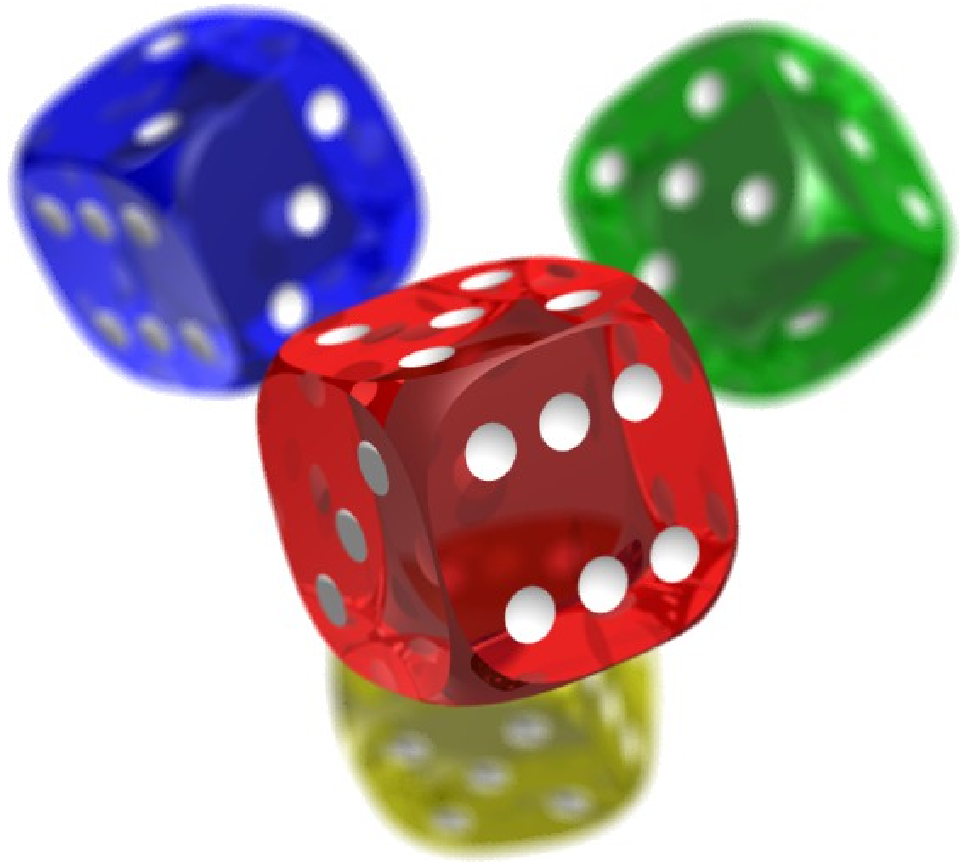
Dice in Perl

Bob Mathews



Dice?

There are a lot of different types of dice.

But I found plenty of weird things to do with ordinary cube-shaped six-sided dice.

Generating random dice rolls is easy.

```
print " ", 1 + int rand 6 for 1 .. 20;
```

```
 2 4 5 3 4 6 1 3 6 5 1 6 3 6 3 1 2 2 6 2
```

So I'm not going to talk about that.

What, Then?

I'm going to show how to determine the probability of rolling various outcomes.

This information can be useful for strategy in dice-based games.

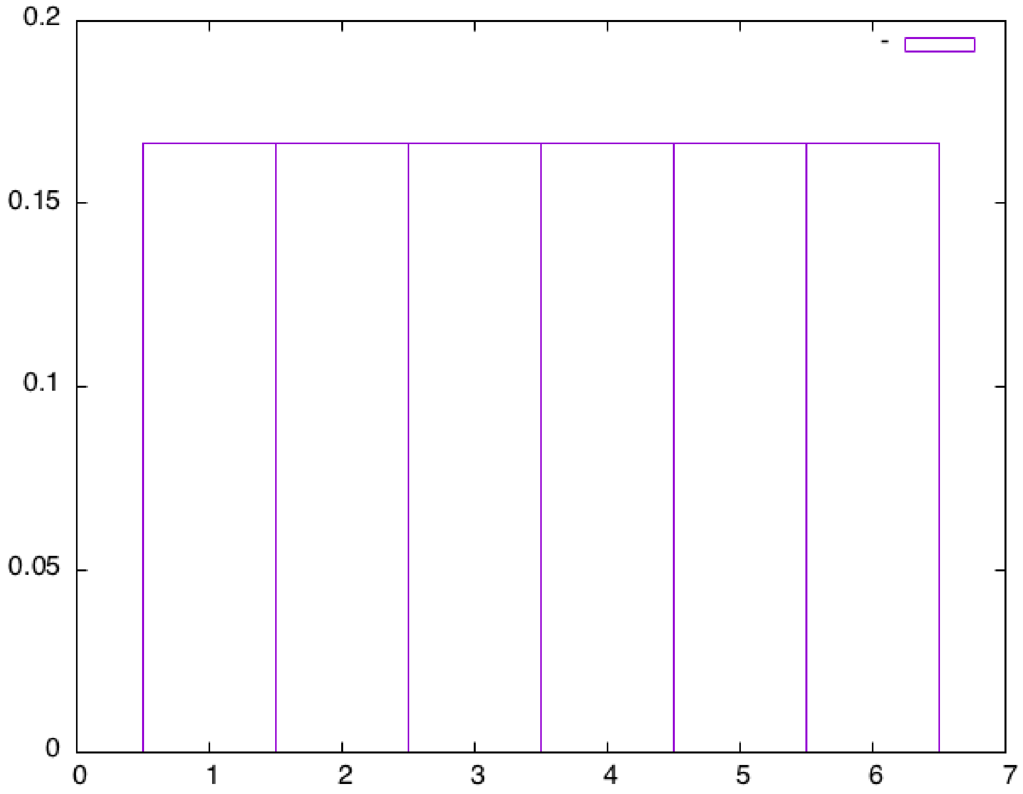
And there are some generally useful programming techniques involved.

Like displaying the results in a graph.

Graphics::GnuplotIF

There are a lot of plotting modules on CPAN, but GnuplotIF seems to be the easiest way to display a plot on-screen.

Rolling one die produces a uniform distribution. Pretty boring.



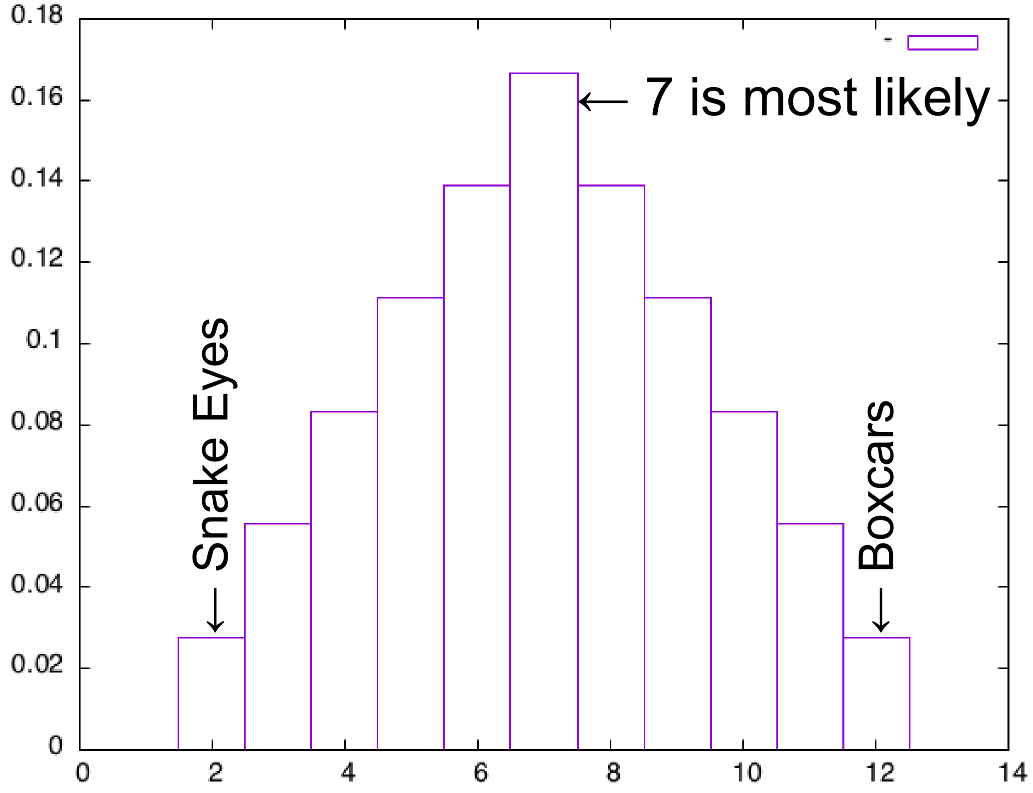
```
use Graphics::GnuplotIF;
my $plot = Graphics::GnuplotIF->new(
   style => 'boxes', yrange => [0, 0.2], persist => 1);
# $plot->gnuplot_hardcopy('plot.png', 'pngcairo');
$plot->gnuplot_plot_xy([1..6], [(1/6)x6]);
```

Pair of Dice

In many games, you roll two dice together.

We can simply loop through all the possible dice rolls.

```
my @count = (0) x 13;

for my $d1 (1 .. 6) {
   for my $d2 (1 .. 6) {
      $count[$d1 + $d2]++;
   }
}

my @pmf = map { $_ / (6*6) } @count;

my $plot = Graphics::GnuplotIF->new(
   style => 'boxes', xrange => [0, 14], persist => 1);
$plot->gnuplot_plot_y(\@pmf);
```



What Good Is That?

You can answer many questions from the probability mass function (PMF).

Probability of rolling over 7:

```
$p_over_7 += $pmf[$_] for 8 .. $#pdf;
```

Probability of rolling even:

```
for (my $i = 0; $i <= $#pmf; $i += 2) {
   $p_even += $pmf[$i];
}
```

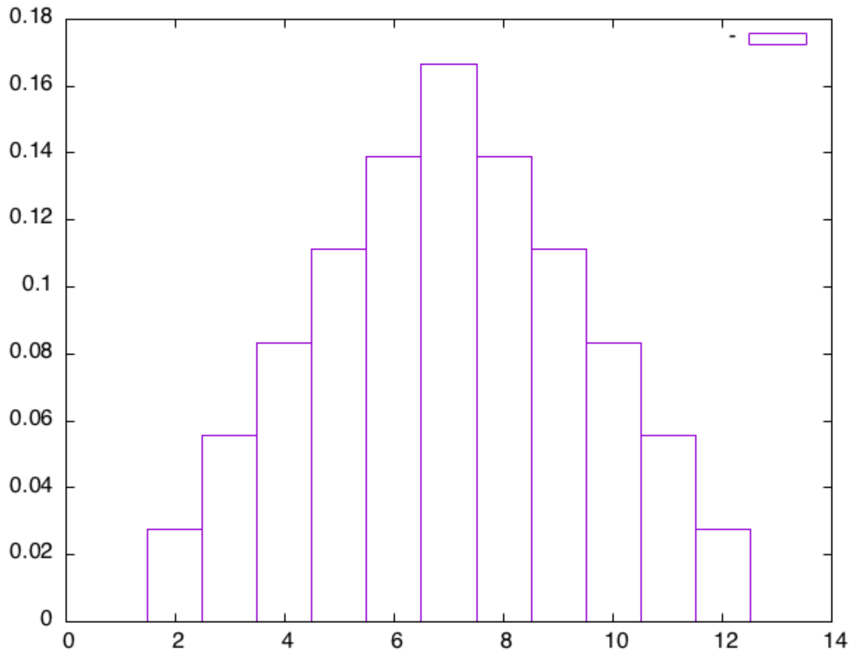
Average:

```
$avg += $_ * $pmf[$_] for 0 .. $#pmf;
```

Sicherman Dice

What's up with these dice?

```
my @sich1 = ( 1, 2, 2, 3, 3, 4 );
my @sich2 = ( 1, 3, 4, 5, 6, 8 );
for my $d1 (@sich1) {
   for my $d2 (@sich2) {
      $count[$d1 + $d2]++;
   }
}
```



Surprisingly, they produce the same probability distribution as an ordinary pair of dice.

More Dice

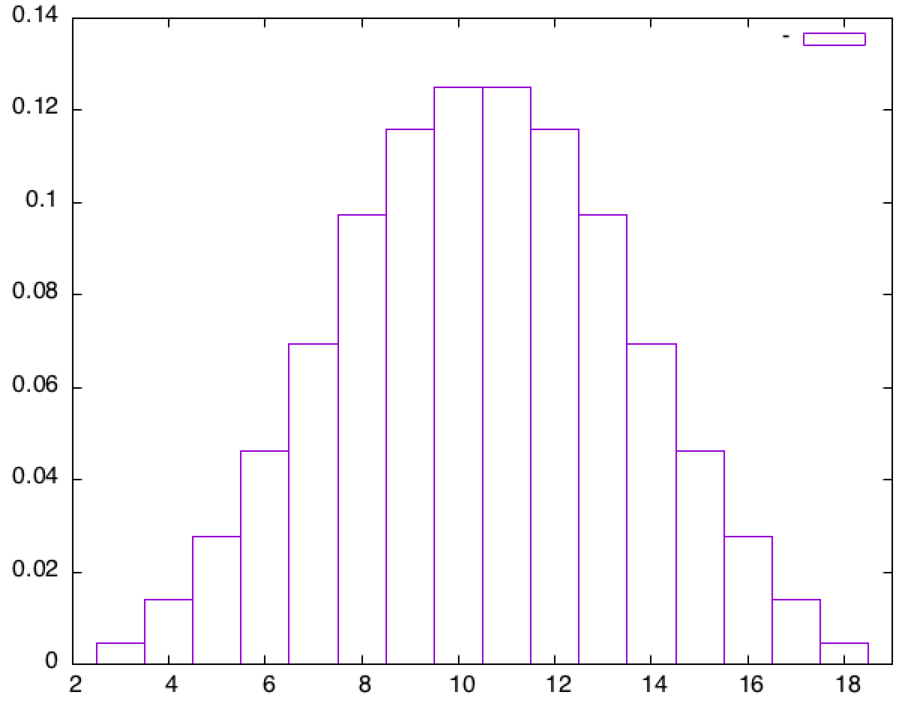
Of course, you might need to roll more than two. We can just add another nested loop...

```
my @count = (0) x 19;
for my $d1 (1 .. 6) {
   for my $d2 (1 .. 6) {
      for my $d3 (1 .. 6) {
         $count[$d1 + $d2 + $d3]++;
      }
   }
}
```

But that's dumb and boring, and what if you don't know the number of dice to roll ahead of time?

Recursive Dice Rolling

The recursive function `roll` calls the anon `sub` repeatedly with different arrays of dice.



```
use List::Util qw( sum );
roll(3, sub {
   $count[&sum]++;
});

sub roll {
   my ($num, $sub, @dice) = @_;
   if (@dice < $num) {
      roll($num, $sub, @dice, $_) for 1 .. 6;
   }
   else {
      $sub->(@dice);
   }
}
```

Perl trick: `&sum` calls the `sum` function with the current `@_` array.

Recursive Call Tree

```
roll(2, $sub)

roll(2, $sub, 1)
    roll(2, $sub, 1, 1) → $sub->(1, 1)
    roll(2, $sub, 1, 2) → $sub->(1, 2)
    roll(2, $sub, 1, 3) → $sub->(1, 3)
    roll(2, $sub, 1, 4) → $sub->(1, 4)
    roll(2, $sub, 1, 5) → $sub->(1, 5)
    roll(2, $sub, 1, 6) → $sub->(1, 6)

roll(2, $sub, 2)
    roll(2, $sub, 2, 1) → $sub->(2, 1)
    roll(2, $sub, 2, 2) → $sub->(2, 2)
    roll(2, $sub, 2, 3) → $sub->(2, 3)
    roll(2, $sub, 2, 4) → $sub->(2, 4)
    roll(2, $sub, 2, 5) → $sub->(2, 5)
    roll(2, $sub, 2, 6) → $sub->(2, 6)

roll(2, $sub, 3)
    roll(2, $sub, 3, 1) → $sub->(3, 1)
    roll(2, $sub, 3, 2) → $sub->(3, 2)
    roll(2, $sub, 3, 3) → $sub->(3, 3)
    roll(2, $sub, 3, 4) → $sub->(3, 4)
    roll(2, $sub, 3, 5) → $sub->(3, 5)
    roll(2, $sub, 3, 6) → $sub->(3, 6)

...
```

Too Slow

This method takes an exponentially increasing amount of time, so it's not practical for large numbers of dice.

Yes, some games use a lot of dice.

Dice	Time
3	0.00025 sec
4	0.00173 sec
5	0.00932 sec
6	0.0638 sec
7	0.372 sec
8	2.36 sec
9	15.3 sec
10	94.3 sec
11	~ 10 min?
12	~ 1 hour?
20	~ 200 years?

Part 2: Convolutional Methods

What Went Wrong?

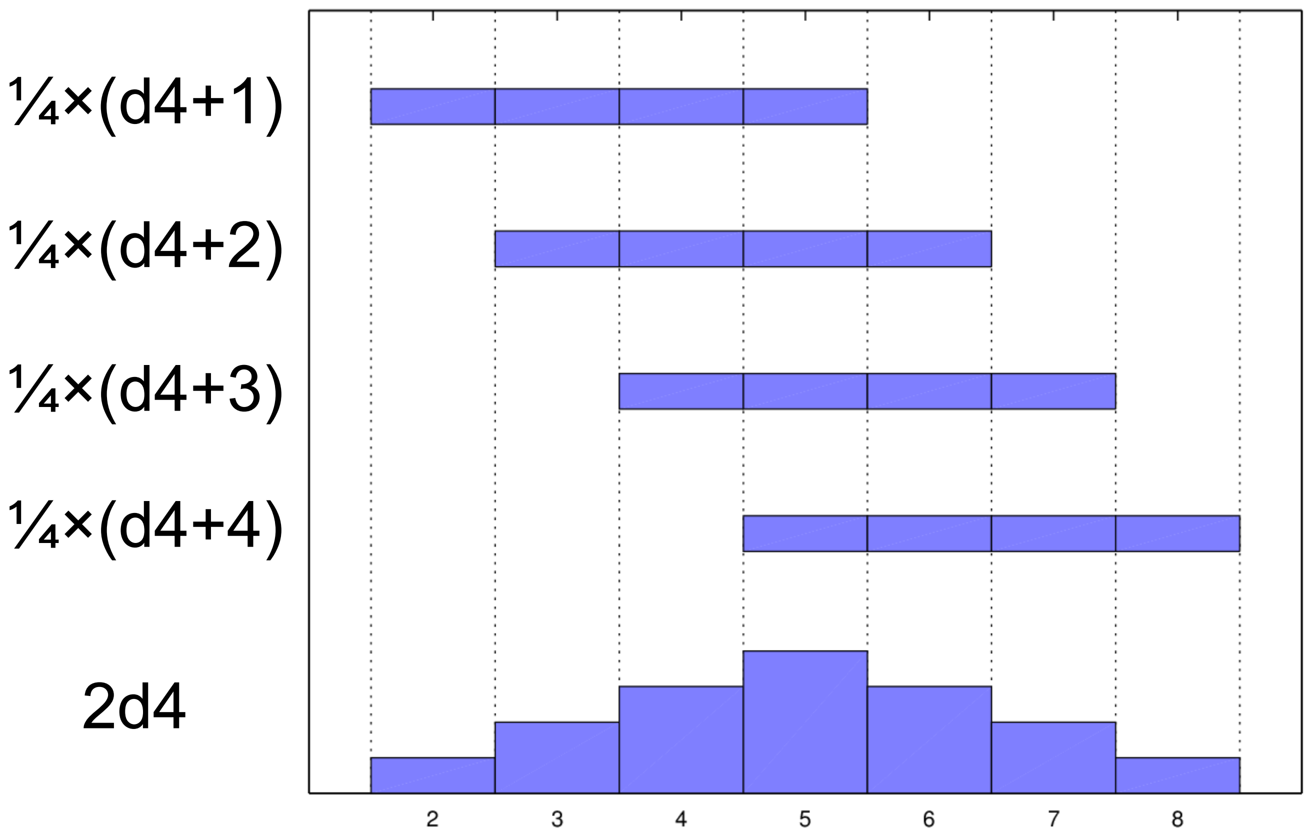
Brute-force looping spent a lot of time working through the individual dice values, but we only care about the sum.

With 10 dice, there are over 60 million different rolls, but the sum can only go up to 60.

Illustrations stolen from Gunnerkrigg Court by Tom Siddell

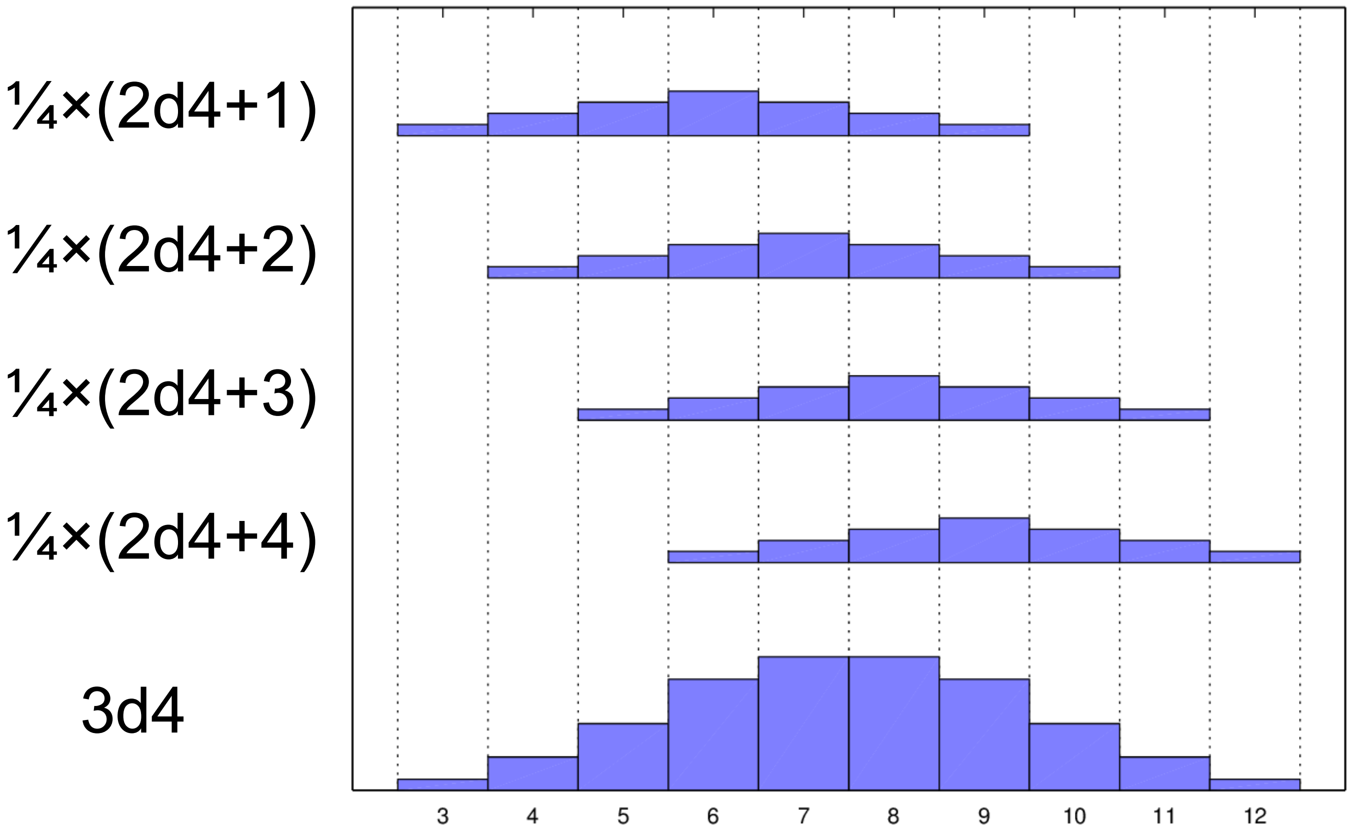
What Else Can We Do?

Work with the PMFs directly. Using 4-sided dice to keep the diagram simple, start with four shifted copies of the boring one-die PMF. They add up to the 2d4 PMF.



Then What?

Add up four copies of the 2d4 PMF to get the 3d4 PMF.
Repeat as needed.



Convolution Algorithm

This operation is called “convolution.”

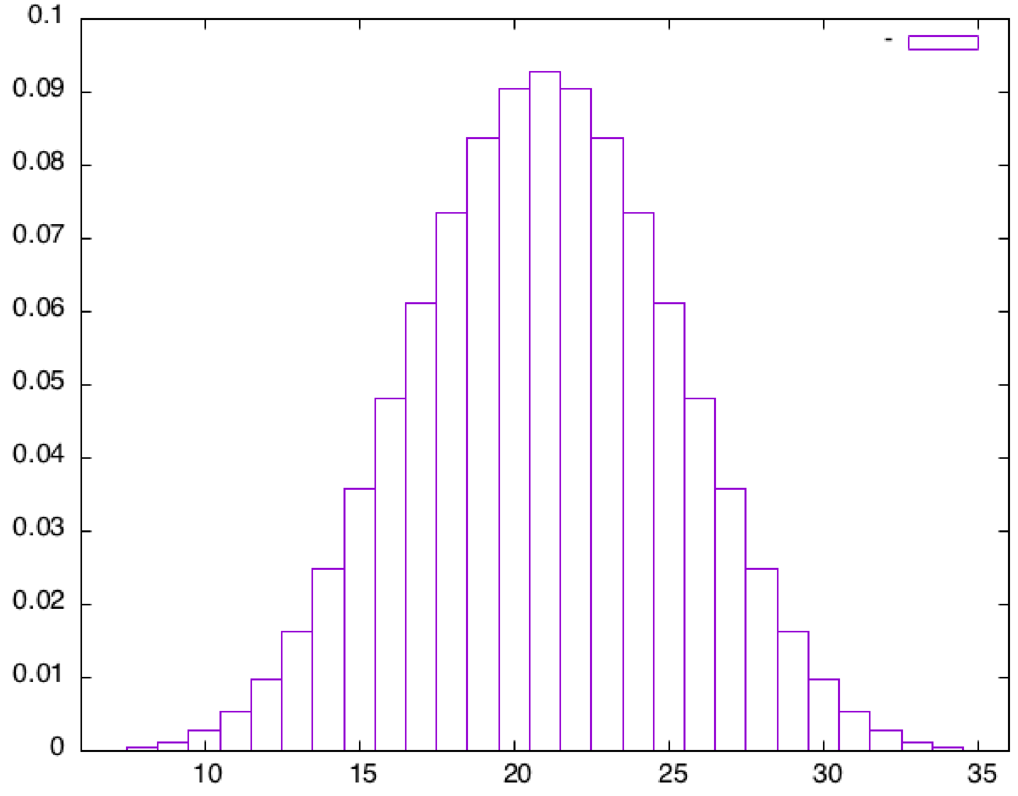
```
sub convol {
   my ($in1, $in2) = @_;
   my @out = (0) x ($#$in1 + $#$in2 + 1);
   for my $i (0 .. $#$in1) {
      for my $j (0 .. $#$in2) {
         $out[$i + $j] += $in1->[$i] * $in2->[$j];
      }
   }
   return \@out;
}
```

It is useful in many situations, such as digital audio filtering and blur and sharpen image filters.

Six Dice

Next, set up the PMF for one die and run it through the convolution a bunch of times.

```
my $d6 = [0, (1/6) x 6];
my $tot = $d6;
$tot = convol($d6, $tot) for 2 .. 6;
```



Hero System

In Hero System game mechanics, BODY damage is determined as follows: rolling a 1 does no damage, 2 to 5 does 1, and rolling 6 does 2.

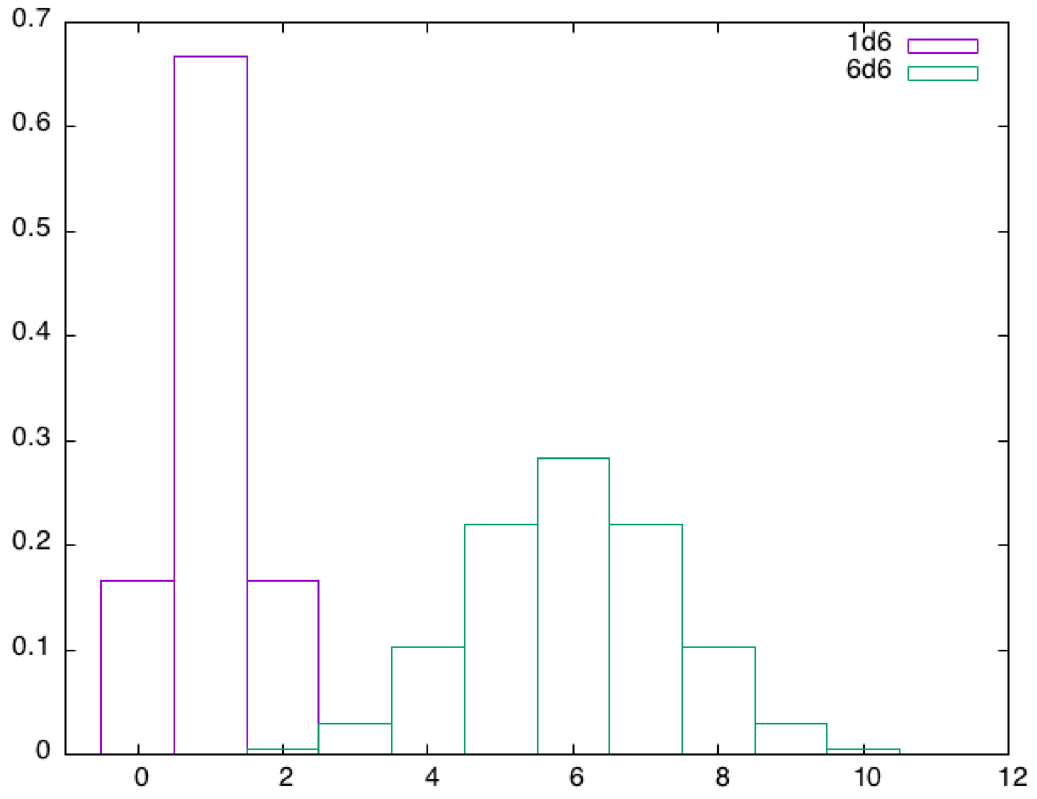
The convolution algorithm can handle this just fine, by starting off with an appropriate input.

```
my $d6 = [1/6, 4/6, 1/6];
my $tot = $d6;
$tot = convol($d6, $tot) for 2 .. 6;
```

Multiple Data Sets On One Plot

```
my $plot = Graphics::GnuplotIF->new(persist => 1,
   style => 'boxes', xrange => [-1, $#$prod],
   plot_titles => ['1d6', '6d6']);
$plot->gnuplot_plot_many([0 .. $#$d], $d,
   [0 .. $#$prod], $prod);
```

In the game, if you are trying to break down a door that has 4 DEFense, you'll need to roll above 4 on the green boxes. Otherwise, you'll just hurt your shoulder.



Shadowrun

The dark future game Shadowrun has an interesting dice-rolling mechanic.

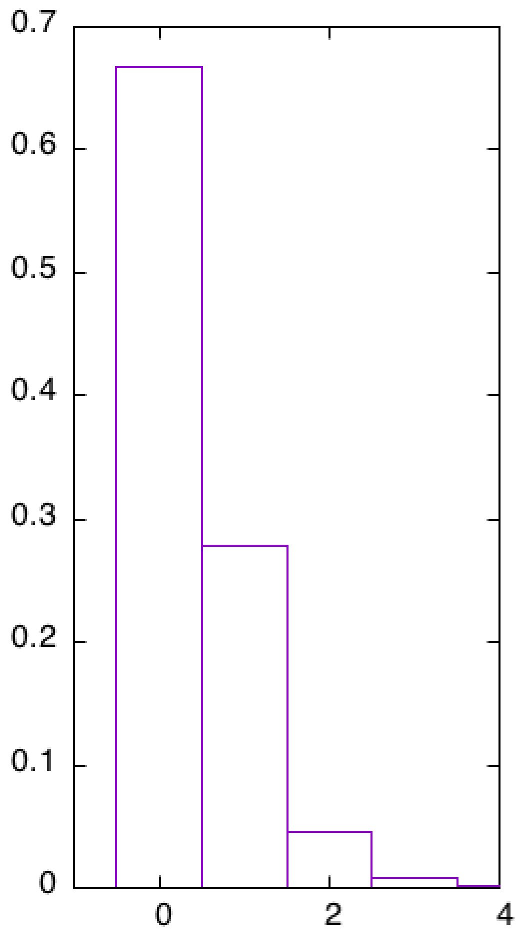
Rolling 1 to 4 scores nothing, and 5 or 6 is a “hit.” If the “rule of six” applies, a 6 means you also get to roll another die.

This keeps going if you roll another 6, so there's theoretically no limit to the number of hits you can roll.

Shadowrun Dice

We can handle this by setting a `$max` number of hits and lumping everything above that together.

```
sub sr_d6 {
   my ($max) = @_;
   my @out = (0) x ($max + 1);
   my $p = 1;
   my $i = 0;
   while ($i < $max) {
      $p /= 6;
      $out[$i] += 4 * $p; # rolled 1-4
      $i++;
      $out[$i] += $p; # rolled 5
   }
   $out[$i] += $p; # $max or higher
   return \@out;
}
```



Modified Convolution

Modified to include maximum number of hits.

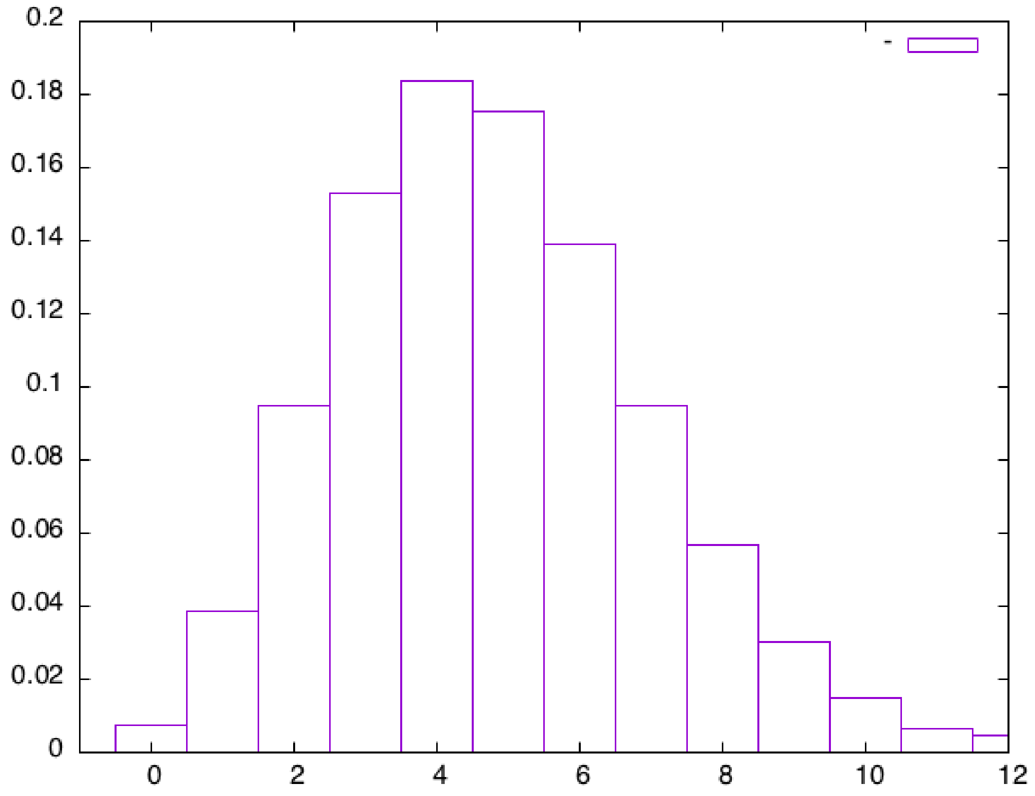
```
sub convol_max {
   my ($in1, $in2, $max) = @_;
   my @out = (0) x ($max + 1);
   for my $i (0 .. $#$in1) {
      for my $j (0 .. $#$in2) {
         my $k = $i + $j;
         $k = $max if $k > $max;
         $out[$k] += $in1->[$i] * $in2->[$j];
      }
   }
   return \@out;
}
```

12 Shadowrun Dice

```
my $max = 12;
my $sr1 = sr_d6($max);
my $sr2 = convol_max($sr1, $sr1, $max);   # 1+1=2 dice
my $sr4 = convol_max($sr2, $sr2, $max);   # 2+2=4 dice
my $sr8 = convol_max($sr4, $sr4, $max);   # 4+4=8 dice
my $sr12 = convol_max($sr8, $sr4, $max); # 8+4=12 dice
```

This produces another bell-curveish result, but it's a bit skewed.

Use of this “doubling” method reduces the number of `convol` calls from 11 to 4.



Multiple Convolution

The doubling method is nifty, but how do you come up with the sequence of convolutions?

There's a simple bit-bashing algorithm for that.

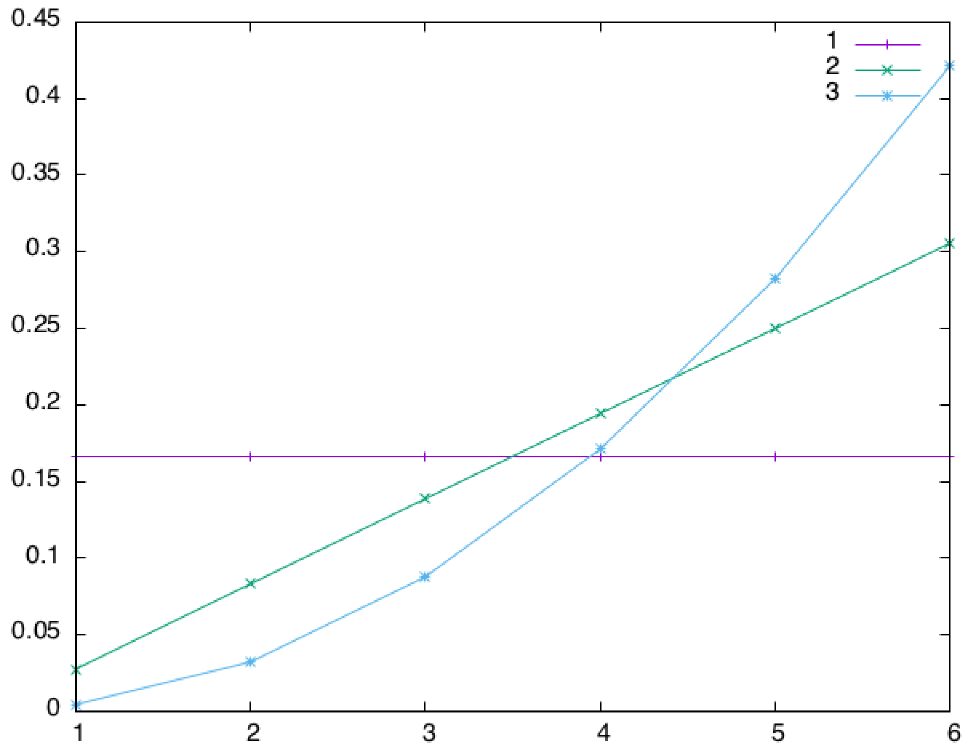
```
sub multi_convol_max {
   my ($x, $n, $max) = @_;
   my $y = $n & 1 ? $x : [ 1 ];
   while ($n >>= 1) {
      $x = convol_max($x, $x, $max);
      $y = convol_max($x, $y, $max) if $n & 1;
   }
   return $y;
}
```

This technique is also used to calculate exponentials in cryptographic algorithms such as RSA.

Maximum of Several Dice

```
use List::Util qw( max );
sub highest {
   my ($d1, $d2) = @_;
   my @out = (0) x (max($#$d1, $#$d2) + 1);
   for my $i (0 .. $#$d1) {
      for my $j (0 .. $#$d2) {
         $out[max($i, $j)] += $d1->[$i] * $d2->[$j];
      }
   }
   return \@out;
}
```

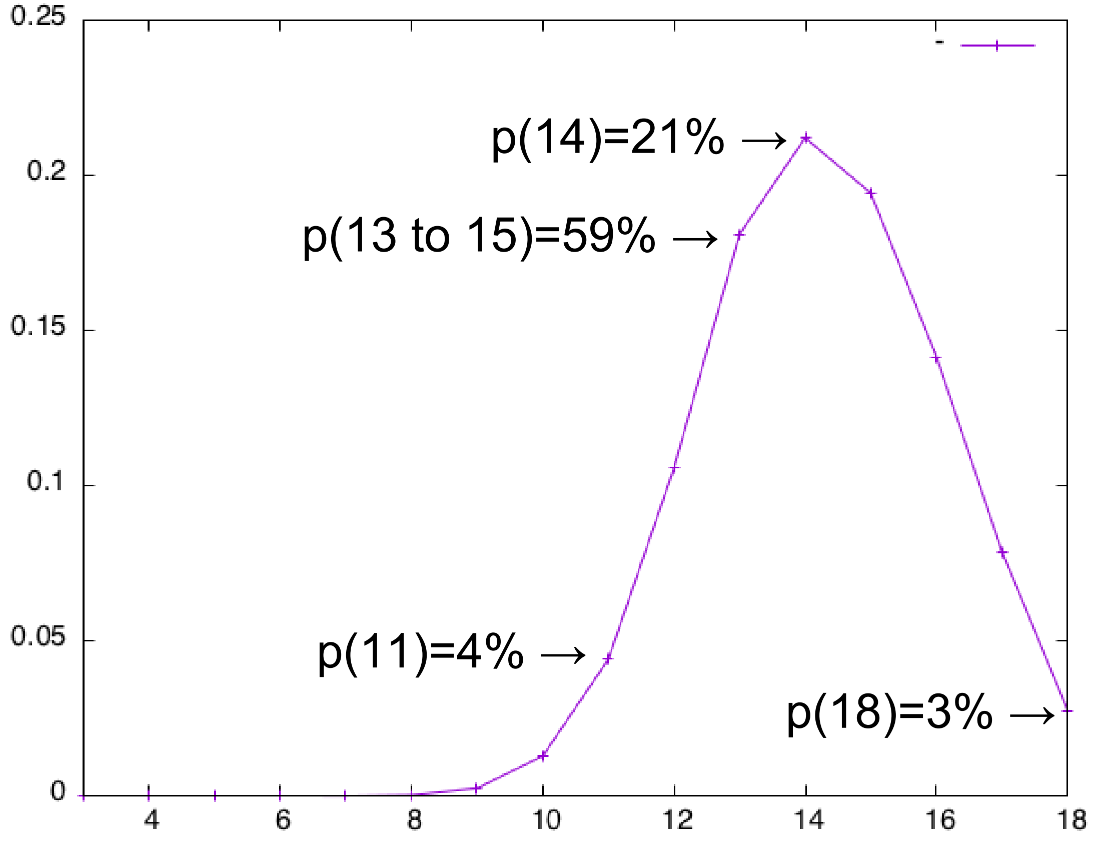
Here's another version of convolution, modified to find the maximum. Shown: 1, 2, and 3 dice, plotted with `style=>'linespoints'`



D&D Ability Scores

And the input to `highest` can be any PMF.

In old-school D&D you roll 3d6 for each of 6 ability scores. What is the highest score likely to be?



Part 3: Spectral Methods

Perl Data Language

PDL is an array-slinging number-crunching tool, similar to Matlab or GNU Octave. Check out the PDL tutorial, it'll walk you through some simple image processing.

```
use PDL;
my $a = sequence(10);
print $a, "\n";
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
print $a + 1, "\n";
```

```
[1 2 3 4 5 6 7 8 9 10]
```

```
print $a ** 2, "\n";
```

```
[0 1 4 9 16 25 36 49 64 81]
```

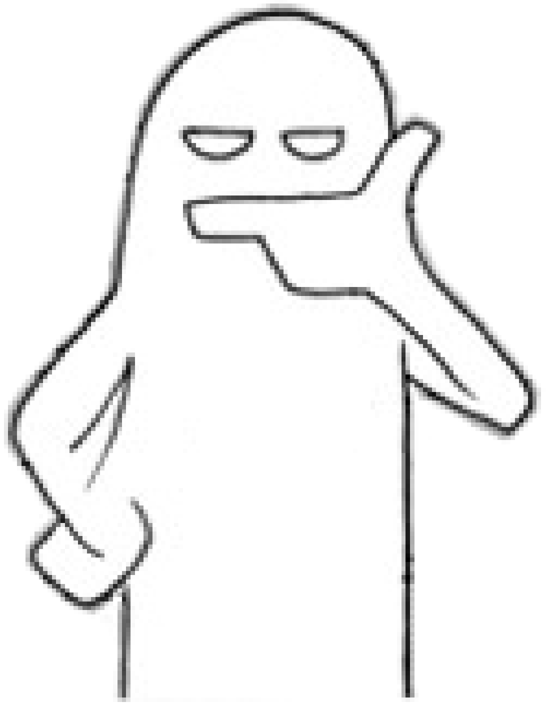
PDL Dice

Set up the d6 distribution, with extra room for the totals from rolling 10 dice.

```
use PDL::NiceSlice;
my $d = zeros(64);
$d(1:6) .= 1/6;
print $d, "\n";
```

```
[0 0.16666667 0.16666667 0.16666667
0.16666667 0.16666667 0.16666667 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

PDL uses `.=` to copy array contents instead of just copying an array reference, similar to using `@$x = @$y` instead of `$x = $y`

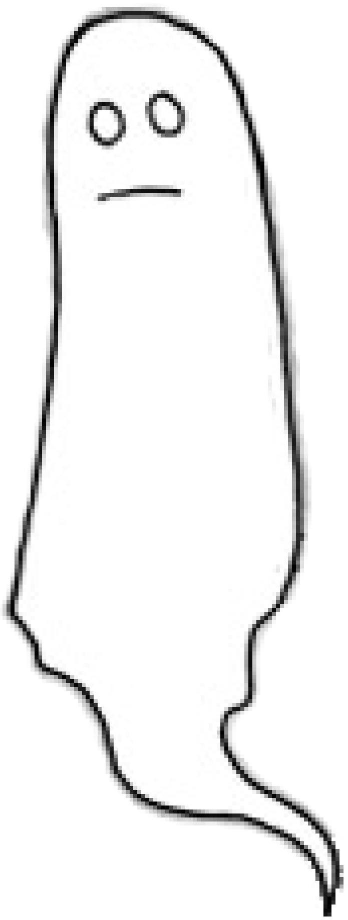


Just Smile and Nod

```
use PDL::FFT;
realfft $d;
print $d, "\n";
```

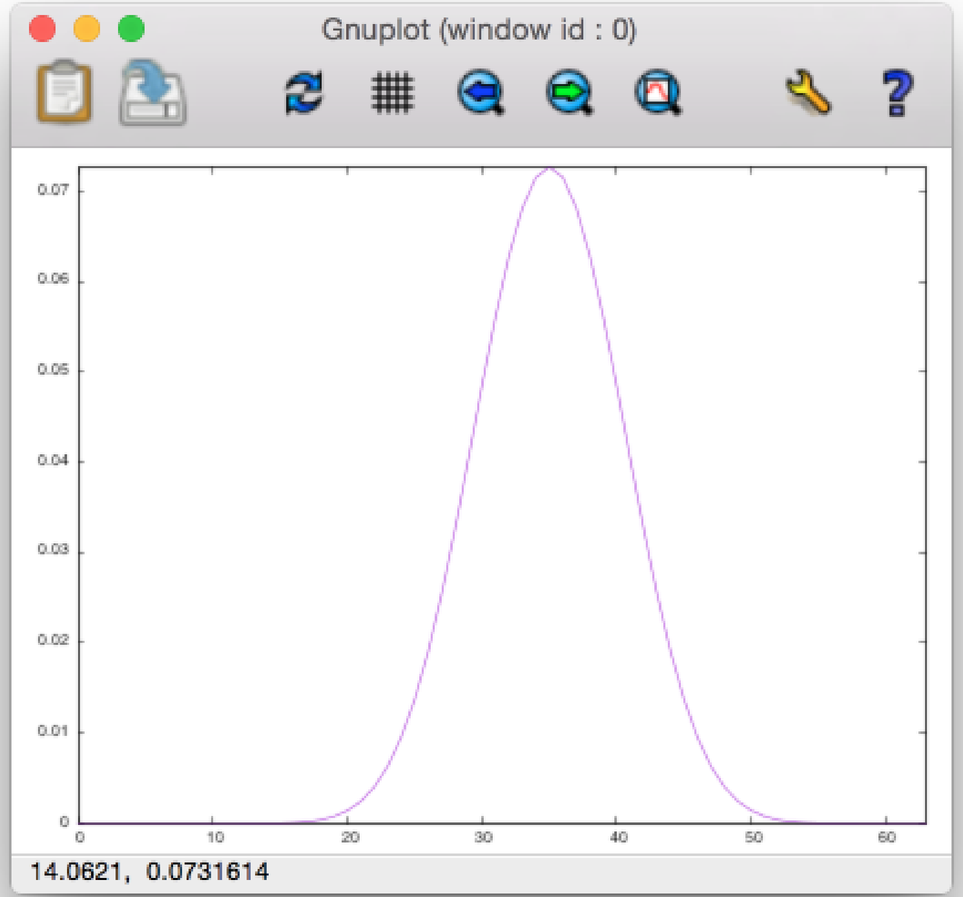
```
[1 0.92836346 0.73024915 0.45140225 0.15397992
-0.10016175 -0.26545148 -0.32328101 -0.2845178...
```

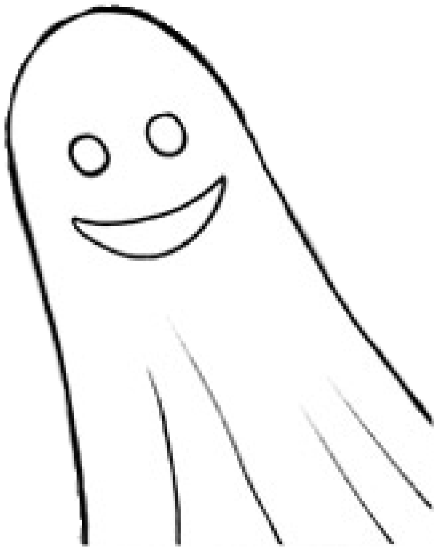
```
# calculate $d ** 10 for 10 dice
my $x = $d(0:31); # real part
my $y = $d(32:63); # imaginary part
my $r = ($x*$x + $y*$y) ** (10 / 2);
my $t = $y->atan2($x,0) * 10;
$x .= $r * cos($t); # modifies $d
$y .= $r * sin($t);

realifft $d;
```



PDL Plot

```
use PDL::Graphics::Simple;
line $d;
```





```
# my $plot = pgswin(out => 'fft.png');
# $plot->line($d);
```

So What?

Why go through all this?

- Brute force with n dice is $O(6^n)$ - too slow
- "Ordinary" convolution is $O(n^2)$
- FFT-based convolution is $O(n \log n)$

So if you ever need to roll a few hundred dice at once...

OK, never mind.

But FFTs are still cool.

That's It

The real purpose has been to introduce the convolution.

It can be a bit tricky to wrap your head around.

But it's dead simple to code.

And it's pretty fast.

And it's pretty flexible to handle variations in your particular programming problem.