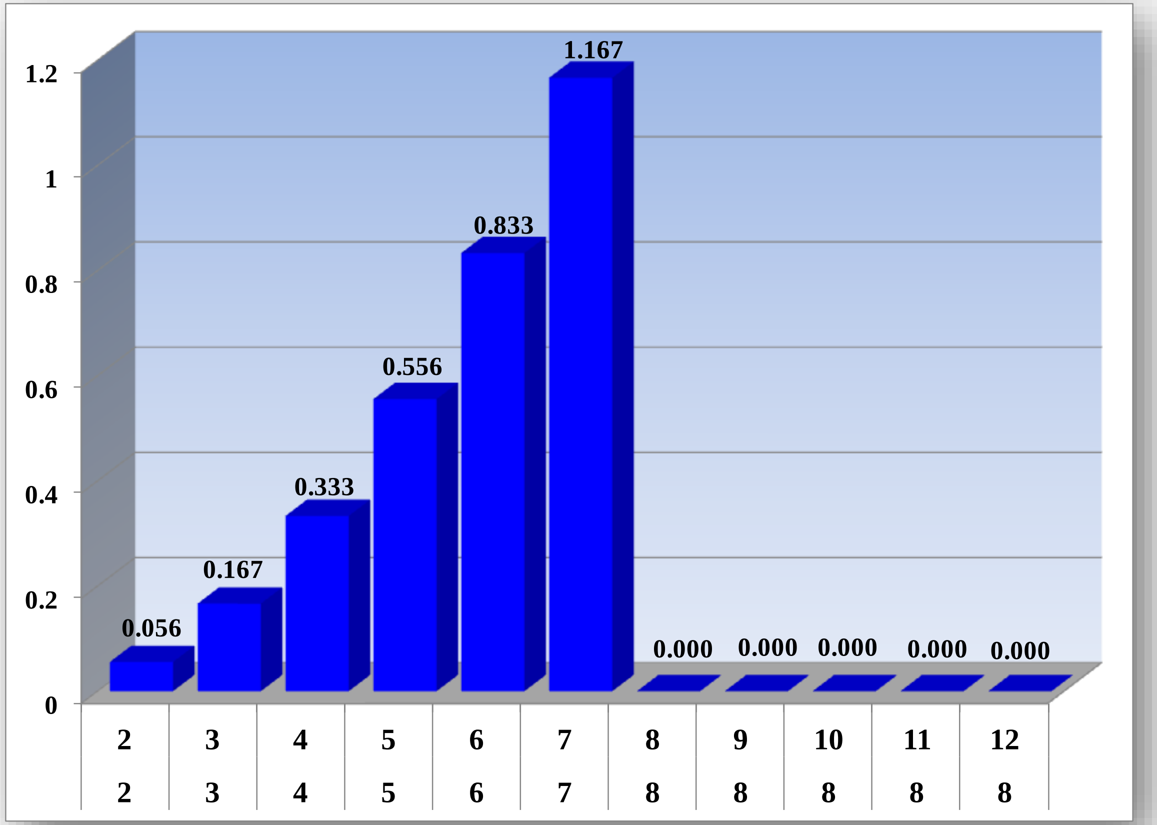
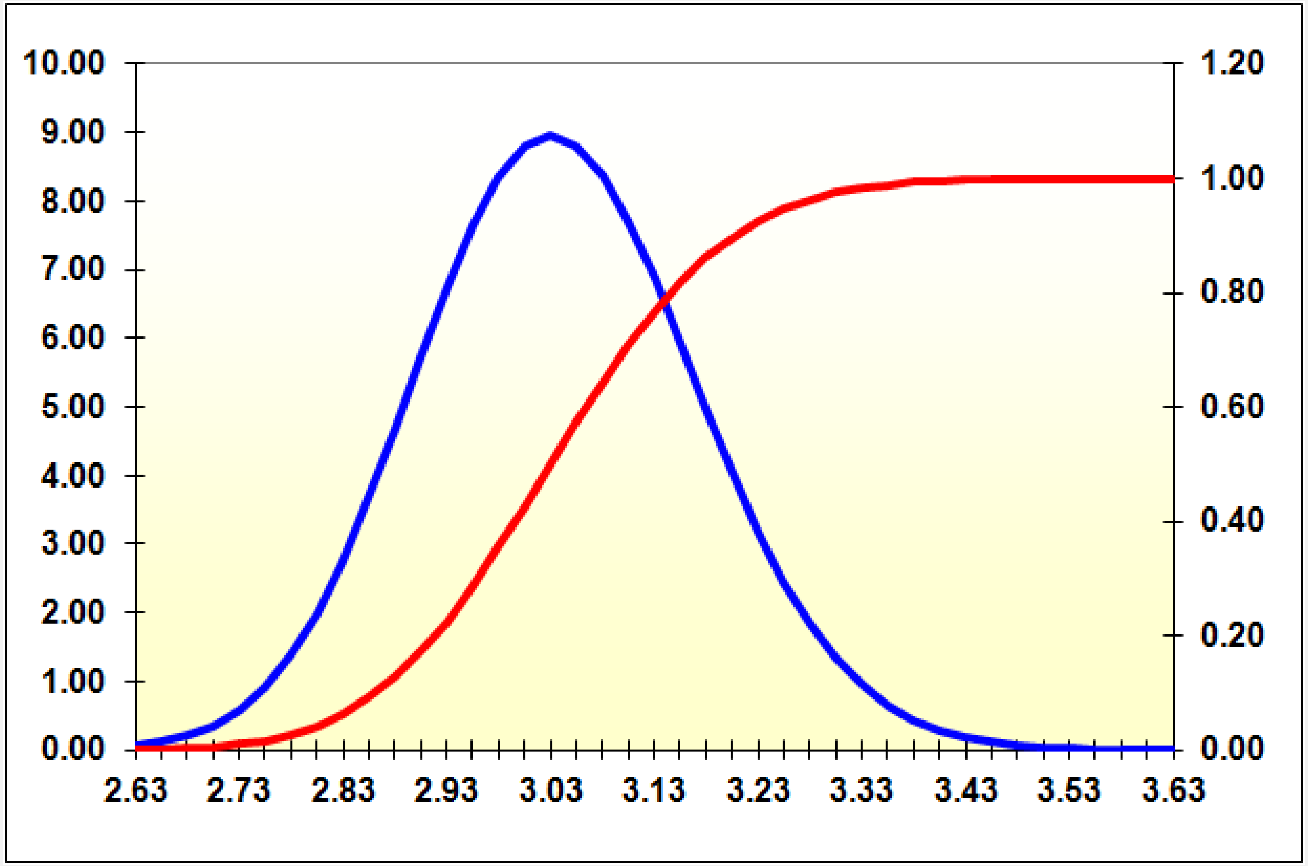
The Taboga Options Pricing Model

... with applications

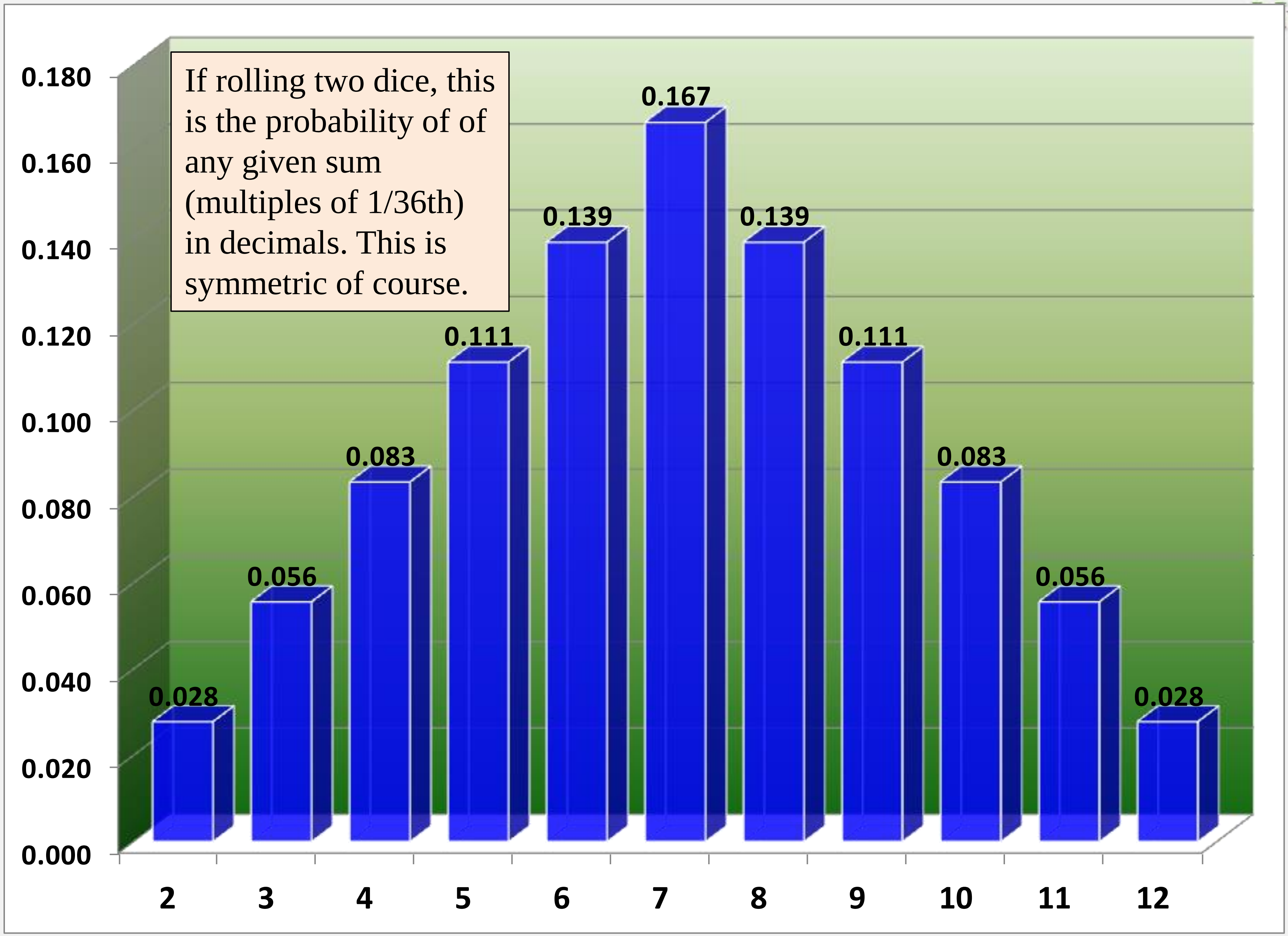


Remember with our assumptions we imply a log-normal distribution

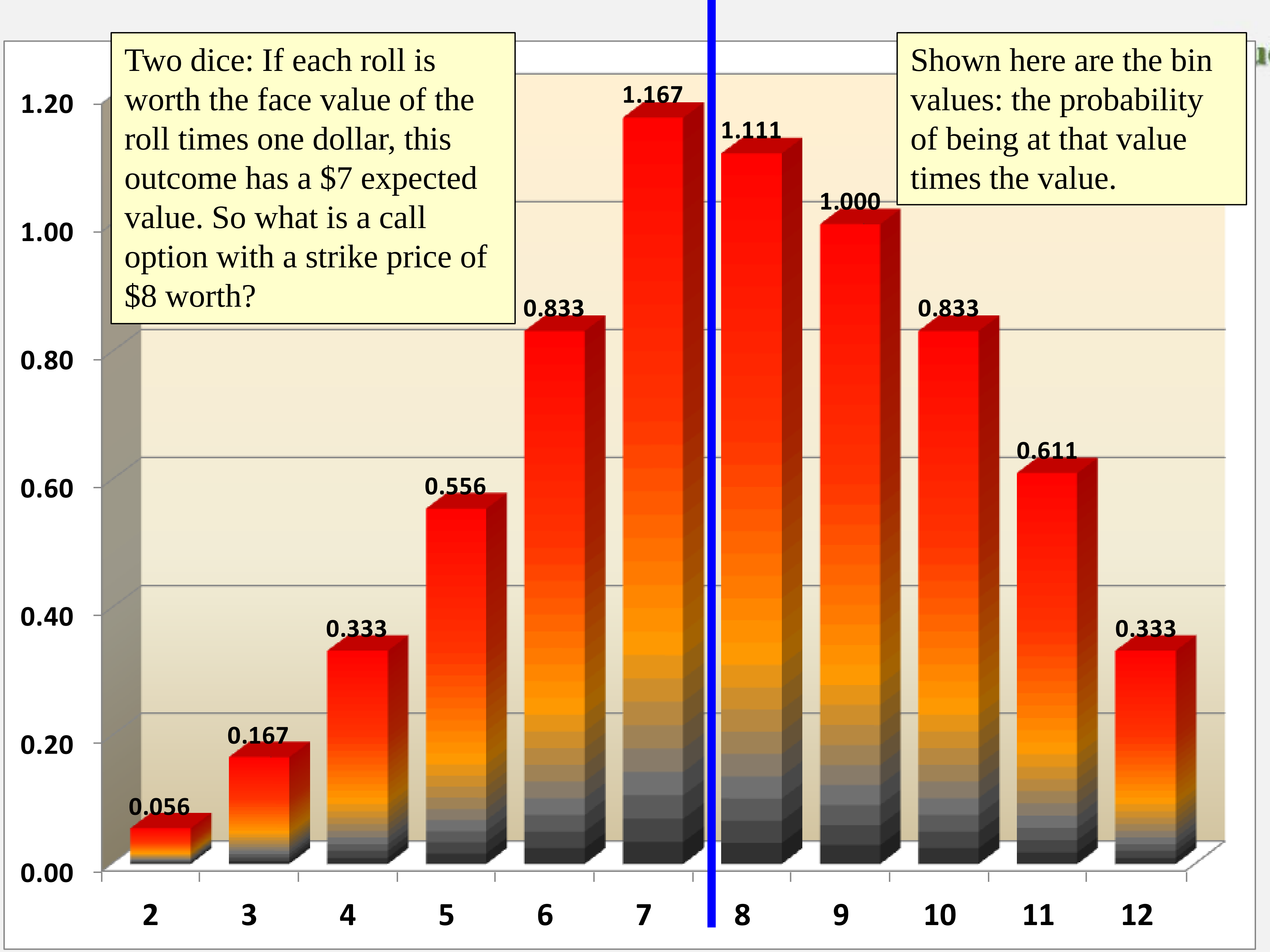


... which we can show as a true log-linear distribution as shown on the left, or as a Gaussian distribution with a log-linear abscissa.

But let's backtrack some to explore some logic …

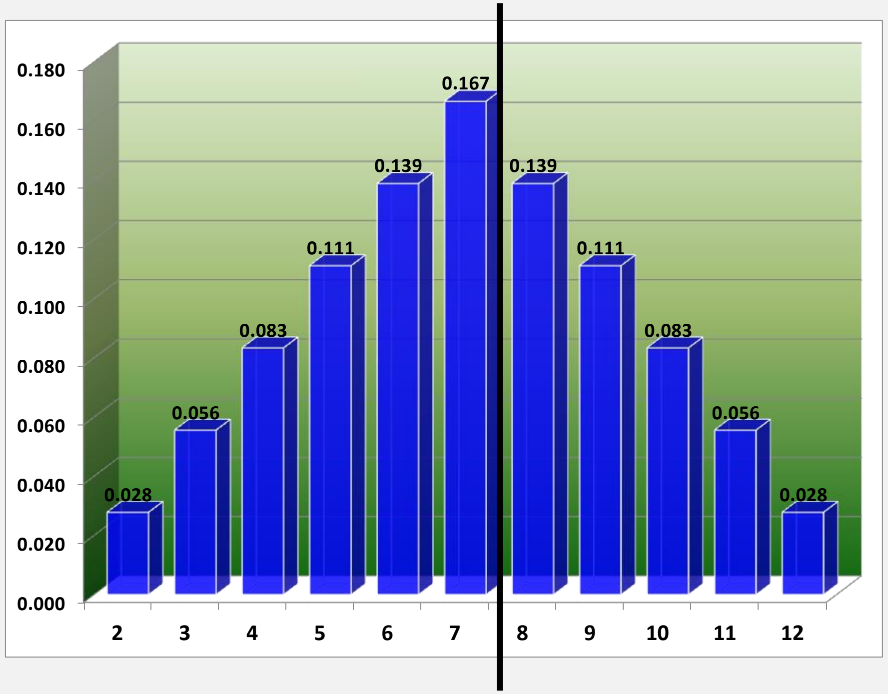


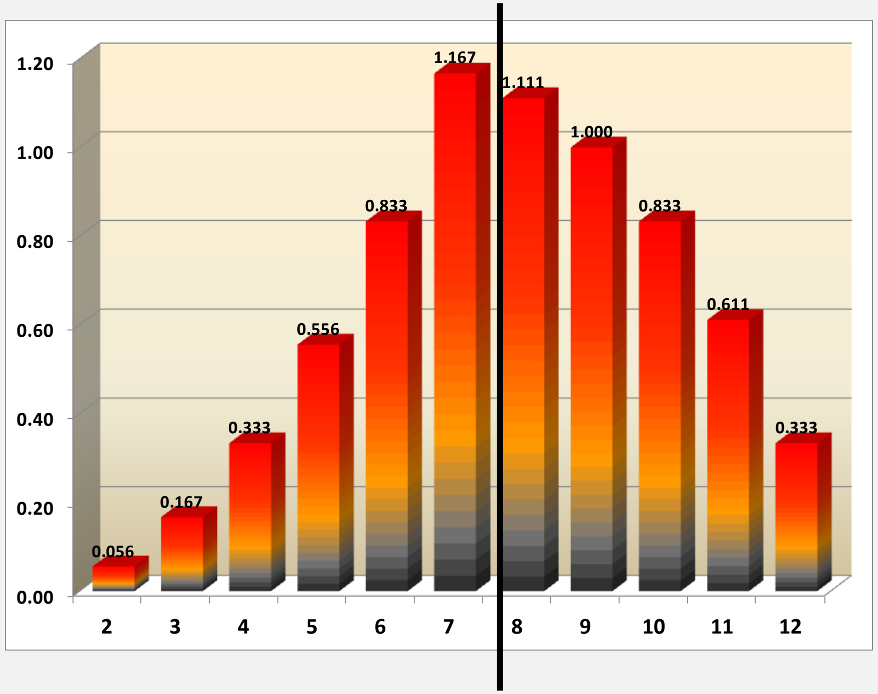
Mudd Finance


Two dice: If each roll is worth the face value of the roll times one dollar, this outcome has a $7 expected value. So what is a call option with a strike price of $8 worth?
Shown here are the bin values: the probability of being at that value times the value.
1.20
1.00
0.80
0.60
0.40
0.20
0.00
0.056
0.167
0.333
0.556
0.833
1.167
1.111
1.000
0.833
0.611
0.333
2
3
4
5
6
7
8
9
10
11
12

According to a little Python program rollem.py, the value of the distribution above 8 is \$3.89. If we split the gamble, that is what the top half is worth. But with a call option with a strike price of 8, we have to pay \$8 for the right to accept any value above \$8.

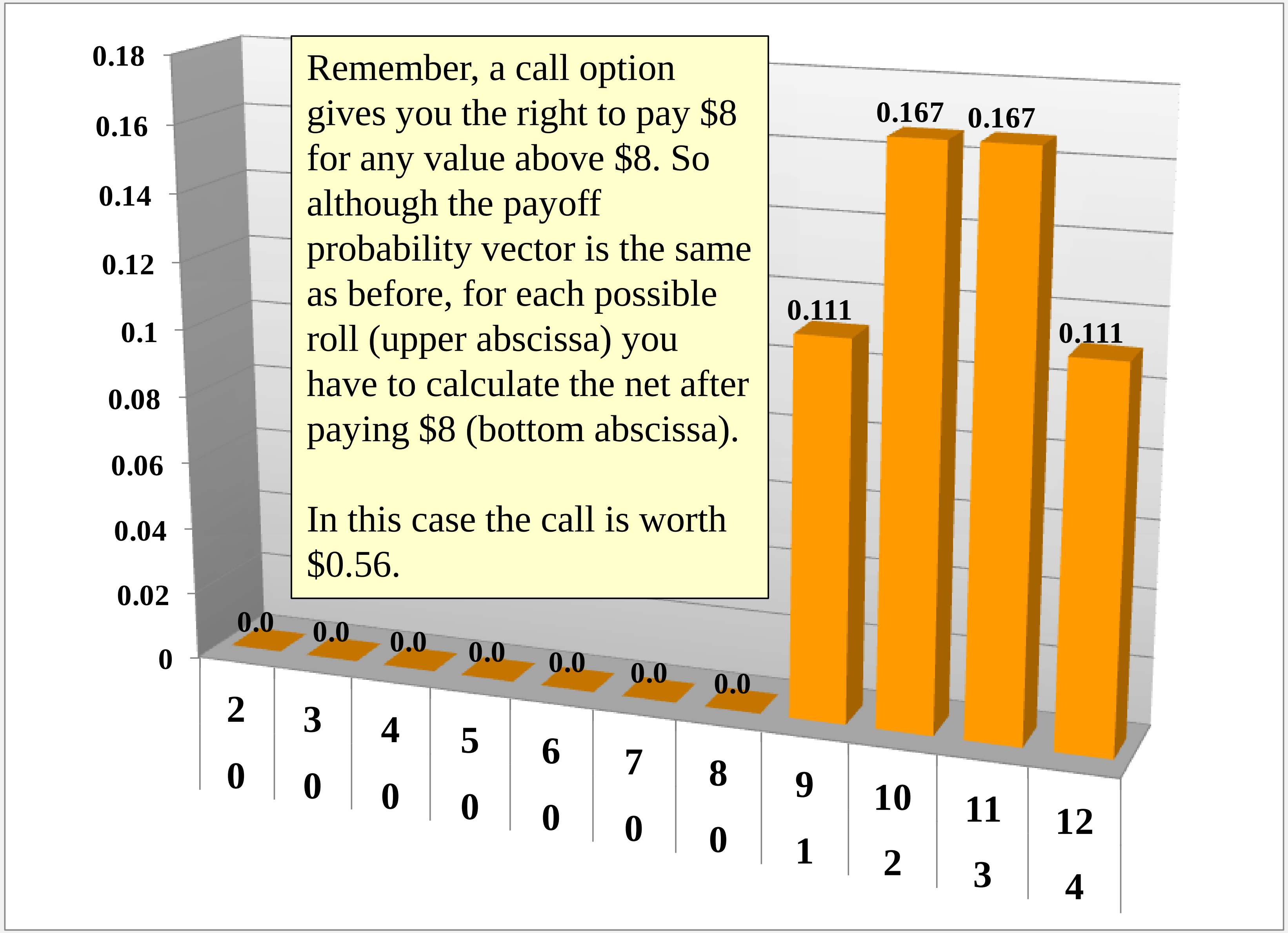
```
Total value of bet: 7.0
Bottom to 7:
[ 0.05555556  0.16666667  0.33333333  0.55555556  0.83333333  1.16666667]
Value of 2 to 7: 3.11111111111
8 to 12:
[ 1.11111111  1.          0.83333333  0.61111111  0.33333333]
Sum of 8 to 12 probs: 0.416666666667
Value of 8 to 12: 3.88888888889
```





But what will an $8 call be worth ??

Mudd Finance

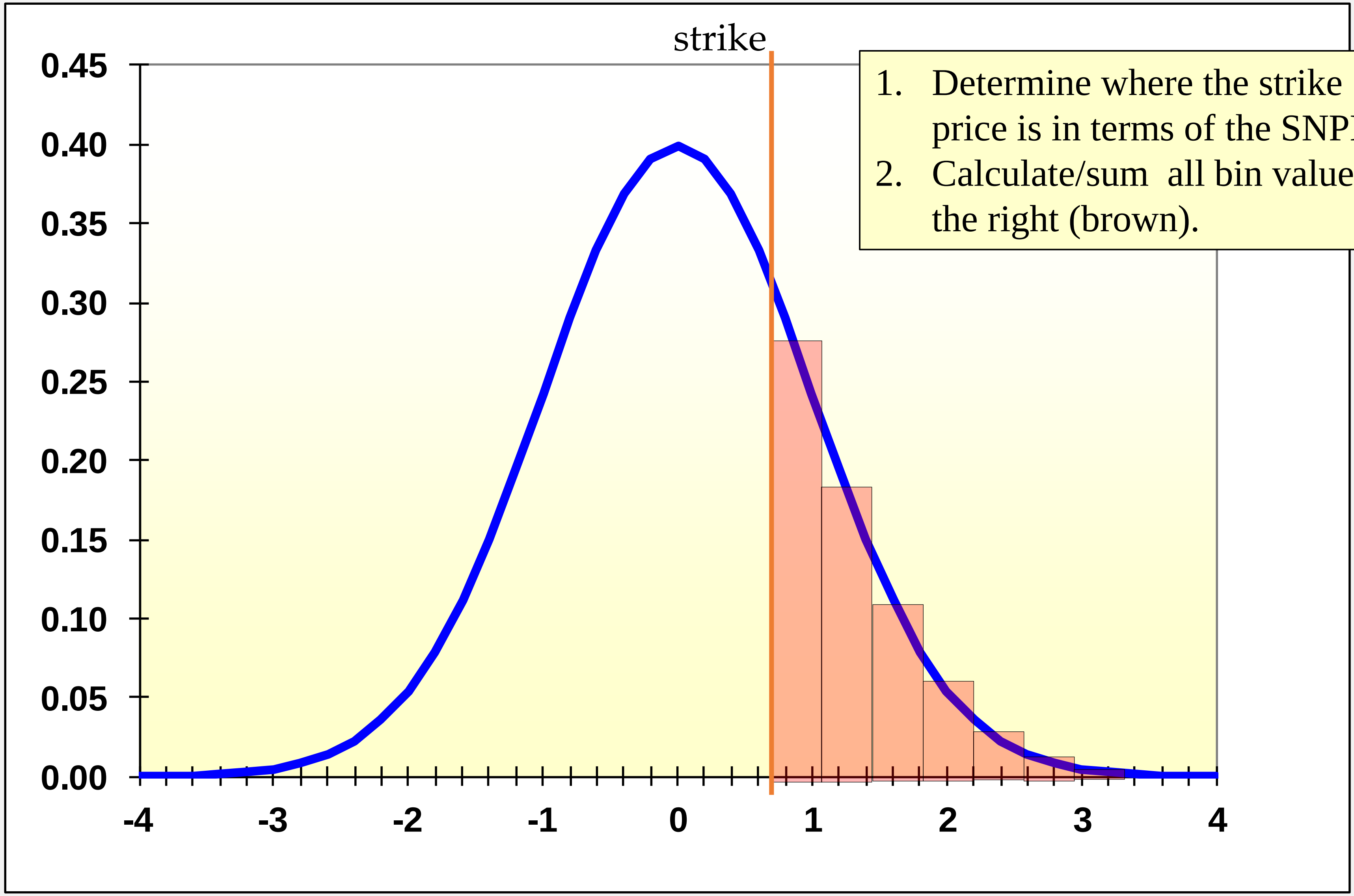


Remember, a call option gives you the right to pay $8 for any value above $8. So although the payoff probability vector is the same as before, for each possible roll (upper abscissa) you have to calculate the net after paying $8 (bottom abscissa).

In this case the call is worth $0.56.

Calculating the value of an OTM call option (brute force):

Mudd Finance



1. Determine where the strike price is in terms of the SNPDF.
2. Calculate/sum all bin values to the right (brown).

standard deviations (or values)

Introducing finutil.py and otranche (and others)

Mudd Finance

```
#
# csnd integrates a standard normal distribution up to some sigma.
#
def csnd(point):
    return (1.0 + math.erf(point/math.sqrt(2.0)))/2.0
#
# cnd integrates a Gaussian distribution up to some value.
#
def cnd(center,point,stdev):
    return (1.0 + math.erf((point - center)/(stdev*math.sqrt(2.0))))/2.0
#
```

```
# An elementary price expected-mean-value adjustment multiplier for log
# distributed prices. The mean of a log-distributed pdf is adjusted by minus
# one-half variance.
#
def lnmeanshift(sigma):
    return 1.0*math.exp(-1.0*(sigma*sigma/2))
```

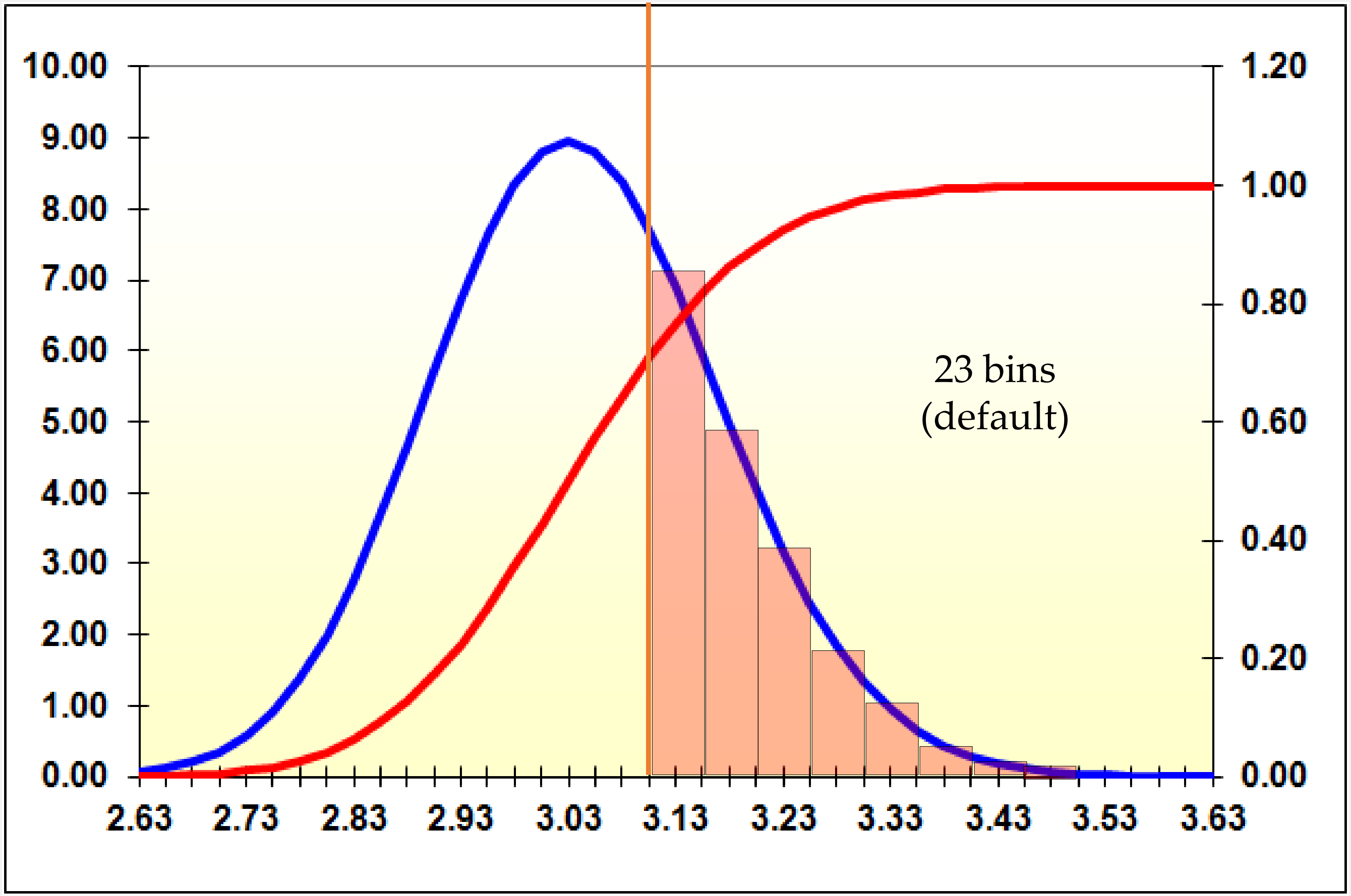
(cont)

```
#
# This is an option tranche value calculator function that assumes you have a
# stock price and strike price, adjusted externally (for example, drift is
# adjusted with drift above). This will calculate the strike-price adjusted
# tranche from either -5 sigma to the strike or from the strike to +5 sigma.
# Sigma used here is duration sigma, adjusted outside using durvol above.
# Main program must set call to true if a call, false if a put.
#
```

```
def otranche(stock,strike,dursigma,call):
    sspread = (math.log(strike/stock))/dursigma
    if call:
        binborder = np.linspace(sspread, 5.00, num=24, dtype=float)
    else:
        binborder = np.linspace(-5.0, sspread, num=24, dtype=float)
    size = len(binborder)
    binedgeprob = np.zeros(size)
#
    for i in range(0,size):
        binedgeprob[i] = csnd(binborder[i])
    size = size - 1
    binprob = np.zeros(size)
    binmidprice = np.zeros(size)
    binvalue = np.zeros(size)
#
    for i in range(0,size):
        binprob[i] = binedgeprob[i+1] - binedgeprob[i]
        binmidprice[i] = ((stock*math.exp(((binborder[i+1]+binborder[i])/2.0)
        *dursigma))*lnmeanshift(dursigma)) - strike
        binvalue[i] = binmidprice[i]*binprob[i]
#
    if call:
        optionprice = np.sum(binvalue[0:(i+1)])
    else:
        optionprice = (np.sum(binvalue[0:(i+1)]))*-1.0
#
    return optionprice
#
```

This is the key right here … this is calculating the value of each bin, just like in the dice problem. Note it is adjusting the center by half-sigma.

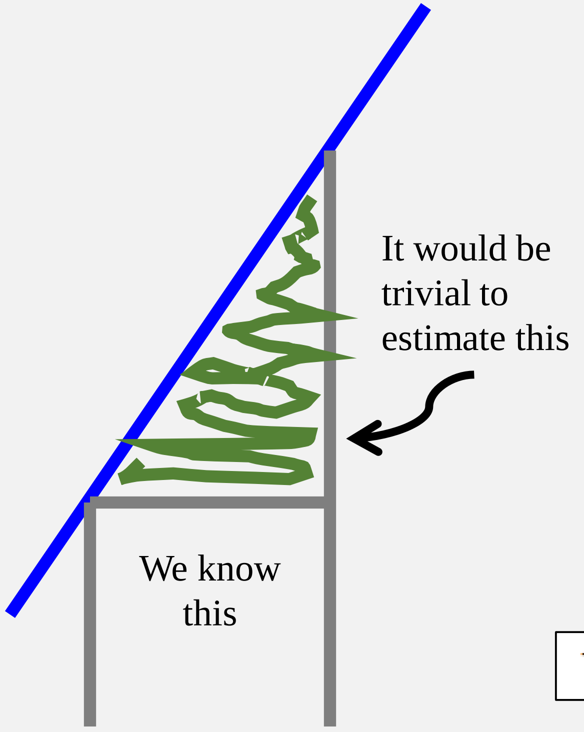
What otranche does ...

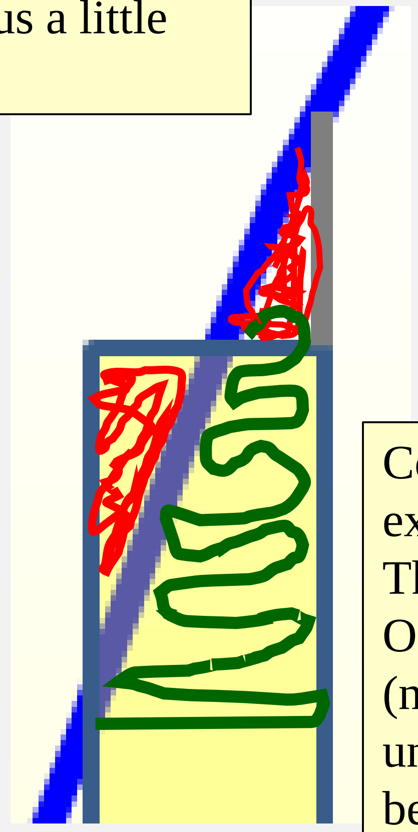


Of course we have this issue (red), which is going to give us a little bias ...

I know we can estimate the green with a Fourier process, but I doubt that will be our solution ...

Are any of you familiar with these tricks of integration??





Conclusion of March 5, 2017 (after some experimentation):
This is simply not going to be an issue. Once the bin count gets up to, say, 23 (num=24 in binborder) the error drops to under a penny. Ideal binorder seems to be:

```
binborder = np.linspace(sspread, 5.00, num=24, dtype=float)
```

Using otranche and Divide and Conquer to solve implied volatility

```python
#
# oidv calculates implied daily and duration volatility for a call or a put
# using divide and conquer (the default for most models). Also see oidvnm.
# This uses an iterative process that uses otranche (above) to calculate the
# sigma, here an implied sigma, from the existing option value (ovalue).
# The call variable is True for a call, False for a put. The convergence is
# within the while loop. This function returns a tuple of two values, daily
# IDV and duration IDV.
#
def oidv(stock,strike,ovalue,days,call):
    precision = float(1e-4)
    low = 0.0
    high = 1.0
    daysigma = float((high+low)/2)
    dursigma = daysigma*durvol(days)
    tempop = otranche(stock,strike,dursigma,call)
    while tempop<=(ovalue-precision) or tempop>=(ovalue+precision):
        if tempop >= (ovalue+precision):
            high = daysigma
        else:
            low = daysigma
        daysigma = float((high+low)/2)
        dursigma = daysigma*durvol(days)
        tempop = otranche(stock,strike,dursigma,call)
    # End of Loop!
    return [daysigma,dursigma]
#
```

Except for high, low, and precision parameters, Divide and Conquer is within the while loop. It always converges within 8 loops, whereas Newton's Method takes only 3 (Newton's Method is also in finutil as oidvnm).

Sample: Using these otranche, oidv, and other finutil utilities for the Taboga Put IDV model

Mudd Finance

Top

```python
stosym = "TSLA"
stp = float(264.35)
strike = float(260.00)
putBid = float(8.95)
putAsk = float(9.10)
days = float(3)
alpha = float(0.000)
# Calculating strike value (PEG)
spread = putAsk - putBid
pprice = putBid + ((0.6)*spread)
call = False
print ()
print ("Stock symbol: ", stosym)
print ("Stock price: ", "%.2f" % stp)
print ("Put strike price: ", "%.3f" % strike)
print ("Put Bid: ", "%.3f" % putBid)
print ("Put Ask: ", "%.3f" % putAsk)
print ("Put price: ", "%.3f" % pprice)
#
```

Mid

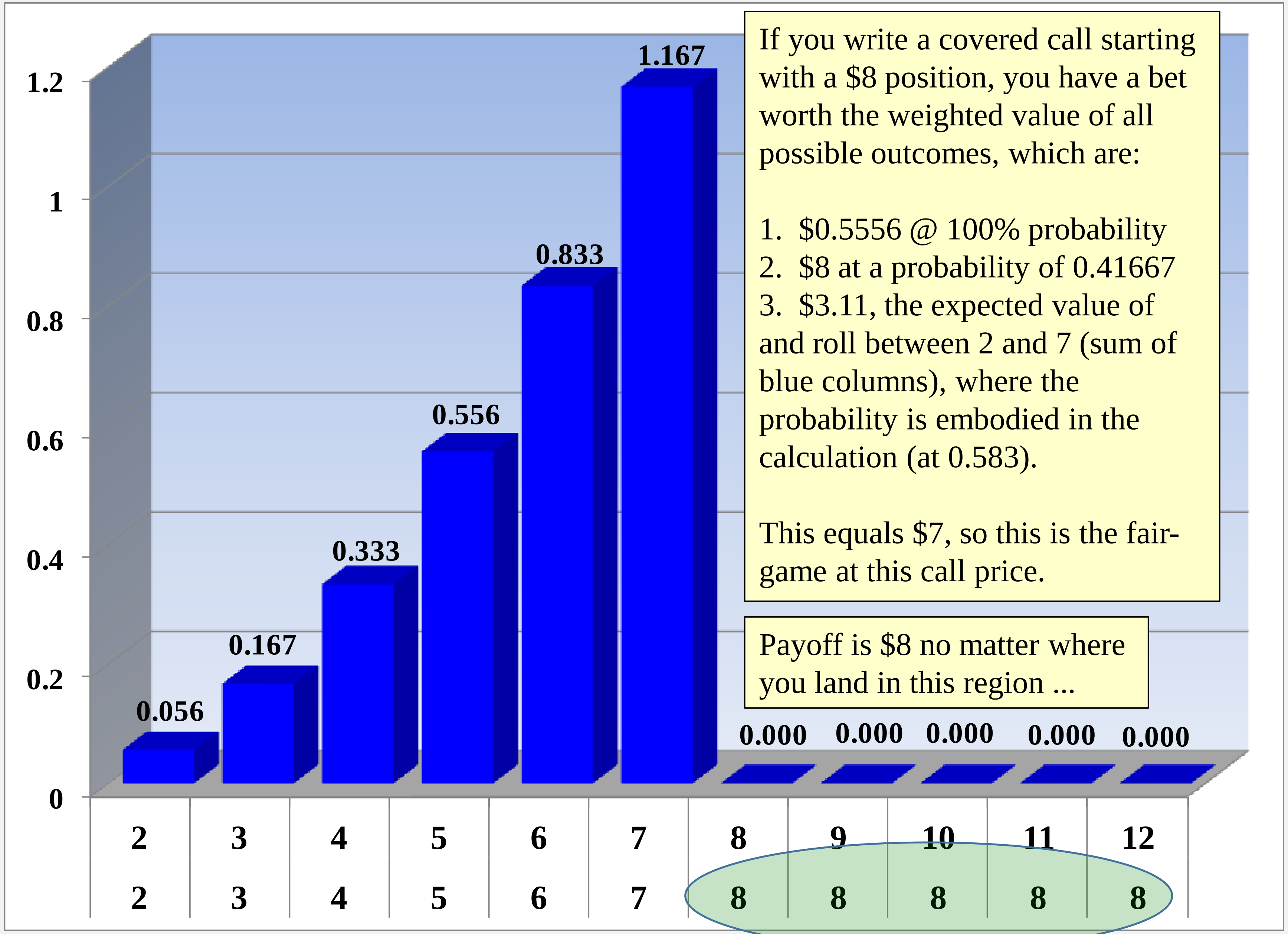
```python
# Calculate the drift if relevant
#
stp = stp*fu.drift(alpha,days)
#
idvout = fu.oidv(stp,strike,pprice,days,call)
daysigma = idvout[0]
dursigma = idvout[1]
#
print ("Days to maturity: ", "%.1f" % days)
print ("Drift rate: ", "%.5f" % alpha)
print ("Drift price: ", "%.2f" % stp)
print ("Implied duration volatility: ", "%.4f" % dursigma)
print ("Implied daily volatility: ", "%.4f" % daysigma)
#
# Calculate one day time decay using our new-found volatility
#
if days >= 2.0:
    timedecay = fu.tdecay(stp,strike,daysigma,pprice,days,call)
    print ("One day time decay: ", "%.3f" % timedecay)
else:
    print ("No time decay with one day remaining.")
```

Bottom

```python
xdelta = 0.01
pct = xdelta*100
stpx = stp*math.exp(-1.0*xdelta)
xupprice = fu.otranche(stpx,strike,dursigma,call)
ddeltad = (pprice - xupprice)/(stp - stpx)*(-1.0)
print ("Delta-adjusted price at minus", "%.1f" % pct, "percent: ", "%.3f"
       % stpx)
print ("Put option price at this new value: ", "%.3f" % xupprice)
print ("Put option gain at this new value: ", "%.3f" % (xupprice - pprice))
print ("Positive discrete delta at", "%.1f" % pct, "percent: ", "%.3f"
       % ddeltad)
stpx = stp*math.exp(xdelta)
xupprice = fu.otranche(stpx,strike,dursigma,call)
ddeltau = (xupprice-pprice)/(stpx-stp)
print ("Delta-adjusted price at plus", "%.1f" % pct, "percent: ", "%.3f"
       % stpx)
print ("Put option price at this new value: ", "%.3f" % xupprice)
print ("Put option loss at this new value: ", "%.3f" % (xupprice - pprice))
print ("Negative discrete delta at", "%.1f" % pct, "percent: ", "%.3f"
       % ddeltau)
```

As can be seen, this is largely an empty shell with a lot of output commands (print). This takes advantage of the fact that Python can be made to be very modular. Also these pricing models easily allow drift, dividends, add-ons and can calculate all of the Greeks using sensitivity analysis. They still have the classical model at the core but have moved far past the classical model.

The breakdown for the call writer (relevant to Aruba) ...



Here we are going to make a very important point ...

The expected value of the gamble determined by this model is the fair value of this call.

If the call sells for more than this value, the option writer has the advantage (is the house).

If it sells for less, the option buyer has the advantage (is the house).