Retirement Planning and Simulation

BUSI 721: Data-Driven Finance I

Kerry Back, Rice University

Outline

- Retirement planning
    - Review
    - With growing savings
- Inflation and real returns
- Simulation
    - Risk in the long run
    - Skewness and kurtosis

RETIREMENT PLANNING

REVIEW

Main Time-Value-of-Money Formulas

$$\mathrm{PV} = \frac{\mathrm{FV}}{(1+r)^n} \quad \Leftrightarrow \quad \mathrm{FV} = \mathrm{PV} \times (1+r)^n$$

$$B_0 = P\left[\frac{1}{1+r} + \frac{1}{(1+r)^2} + \cdots + \frac{1}{(1+r)^n}\right]$$

Problem to Solve

- Hope to spend y dollars per year during n years of retirement
- Save x dollars per year during m years of working
- Expect to earn r per year on investments.
- How large does x need to be?
- Timeline:
    - years $1, \ldots, m \to$ save x
    - years $m+1, \ldots, m+n \to$ spend y.

Match PVs

- Compute PV of spending at year m (standard annuity formula)
- Discount to present - divide by $(1+r)^m$
- Match PV of savings:

$$x \times \left[\frac{1}{1+r} + \cdots + \frac{1}{(1+r)^m}\right]$$

$$= y \times \frac{1}{(1+r)^m} \left[\frac{1}{1+r} + \cdots + \frac{1}{(1+r)^n}\right]$$

Match FVs

- Or match the values at the end of year m: account balance = retirement target
- Multiply both PVs by $(1+r)^m$ to get these FVs:

$$x \times \left[(1+r)^{m-1} + \cdots + 1\right]$$

$$= y \times \left[\frac{1}{1+r} + \cdots + \frac{1}{(1+r)^n}\right]$$

EXAMPLE

In [77]:

```
import numpy_financial as npf

spending = 100000
num_spending_years = 25
num_saving_years = 30
r = 0.06

pv_spending_at_retirement = npf.pv(
    rate=r,
    nper=num_spending_years,
    pmt=-spending
)

print(f"We need to have ${pv_spending_at_retirement:,.0f} at retirement.")
```

```
We need to have $1,278,336 at retirement.
```

MATCH PVs

In [78]:

```
savings = - npf.pmt(
    pv=pv_spending_at_retirement / (1+r)**num_saving_years,
    rate=r,
    fv=0,
    nper=num_saving_years
)

print(f"We need to save ${savings:,.0f} each year.")
```

```
We need to save $16,170 each year.
```

MATCH FVs

In [79]:

```
savings = - npf.pmt(
    pv=0,
    rate=r,
    fv=pv_spending_at_retirement,
    nper=num_saving_years
)

print(f"We need to save ${savings:,.0f} each year.")
```

```
We need to save $16,170 each year.
```

GROWING SAVINGS

- Save x first year, $x(1+g)$ second year, $x(1+g)^2$ third year, etc.
- E.g., $x=20,000$, $g=0.05$, second year is $21,000$, third year is $22,050$, etc.
- FV of savings:

$$x(1+r)^{m-1}+x(1+g)(1+r)^{m-2}+\cdots+x(1+g)^{m-1}$$

$$=x\left[(1+r)^{m-1}+(1+g)(1+r)^{m-2}+\cdots+(1+g)^{m-1}\right]$$

- Solve for x:

$$x=\text{target}/\left[(1+r)^{m-1}+(1+g)(1+r)^{m-2}+\cdots+(1+g)^{m-1}\right]$$

MATCH FVs

In [80]:

```python
import numpy as np

g = 0.03
m = num_saving_years

factors = (1+g) ** np.arange(m)
factors *= (1+r) ** np.arange(m-1, -1, -1)
x = pv_spending_at_retirement / np.sum(factors)

print(f"We need to save ${x:,.0f} each year.")
```

```
We need to save $11,564 each year.
```

REAL RETURNS

- Inflation rate is % change in Consumer Price Index (CPI)
- Real rate of return is inflation-adjusted rate
- Example:
  - Item costs $\$100$ today
  - Can earn 8% on investments
  - Inflation is 3%
  - Instead of buying today, you could invest $\$100$.
  - Have $\$108$ in one year, item costs $\$103$, buy 1 and have $\$5$ left over
  - Extra $\$5$ will buy $5/103$ units. Real rate of return is $5/103$.

Real Return

- Real rate of return in example is

$$5/103 = (108 - 103)/103 = 108/103 - 1 = 1.08/1.03 - 1$$

- Real rate of return in general is

$$r_{\text{real}} = \frac{1 + r_{\text{nominal}}}{1 + \text{inflation}} - 1$$

- Also called "return in constant dollars"

Retirement Planning and Inflation

- Do everything with expected real rate of return
- If savings are expected to grow, use the real growth rate

$$g_{\text{real}} = \frac{1 + g_{\text{nominal}}}{1 + \text{inflation}} - 1$$

- Then retirement spending will be in today's dollars.
- https://learn-investments.rice-business.org/borrowing-saving/inflation

SIMULATION

Independent random normals

- Annual returns are approximately normally distributed.
- And are approximately independent from one year to the next.
- Simulate random normals with np.random.normal.

In [81]:

```
mean = 0.1
stdev = 0.15
np.random.seed(0)
np.random.normal(loc=mean, scale=stdev)
```

Out[81]:

```
0.36460785189514955
```

SIMULATE MULTIPLE YEARS

In [82]:
```python
n = 10
np.random.seed(0)

rets = np.random.normal(
    loc=mean,
    scale=stdev,
    size=n
)
rets
```

Out[82]:
```
array([ 0.36460785,  0.16002358,  0.2468107 ,  0.43613398,  0.3801337
,
       -0.04659168,  0.24251326,  0.07729642,  0.08451717,  0.1615897
8])
```

Compound Returns

- How much would $1 grow to?
  - Answer is np.prod(1+rets)
- What is the total return over the n years?
  - Answer is np.prod(1+rets) - 1

In [83]:
```
print(f"$1 would grow to ${np.prod(1+rets): .3f}")
print(f"the total return is {np.prod(1+rets)-1: .1%}")
```

```
$1 would grow to $ 6.289
the total return is  528.9%
```

SIMULATE MULTIPLE YEARS MULTIPLE TIMES

In [84]:

```
num_prds = 10
num_sims = 5
np.random.seed(0)

rets = np.random.normal(
    loc=mean,
    scale=stdev,
    size=(num_prds, num_sims)
)
np.prod((1+rets), axis=0)
```

Out[84]:

```
array([1.30568504, 4.01010011, 2.92173927, 2.62512839, 4.17660966])
```

DISTRIBUTION OF THE COMPOUND RETURN

- Compounding produces positive skewness
- So, the median is below the mean
- The difference between median and mean is larger when there is more risk.

In [85]:

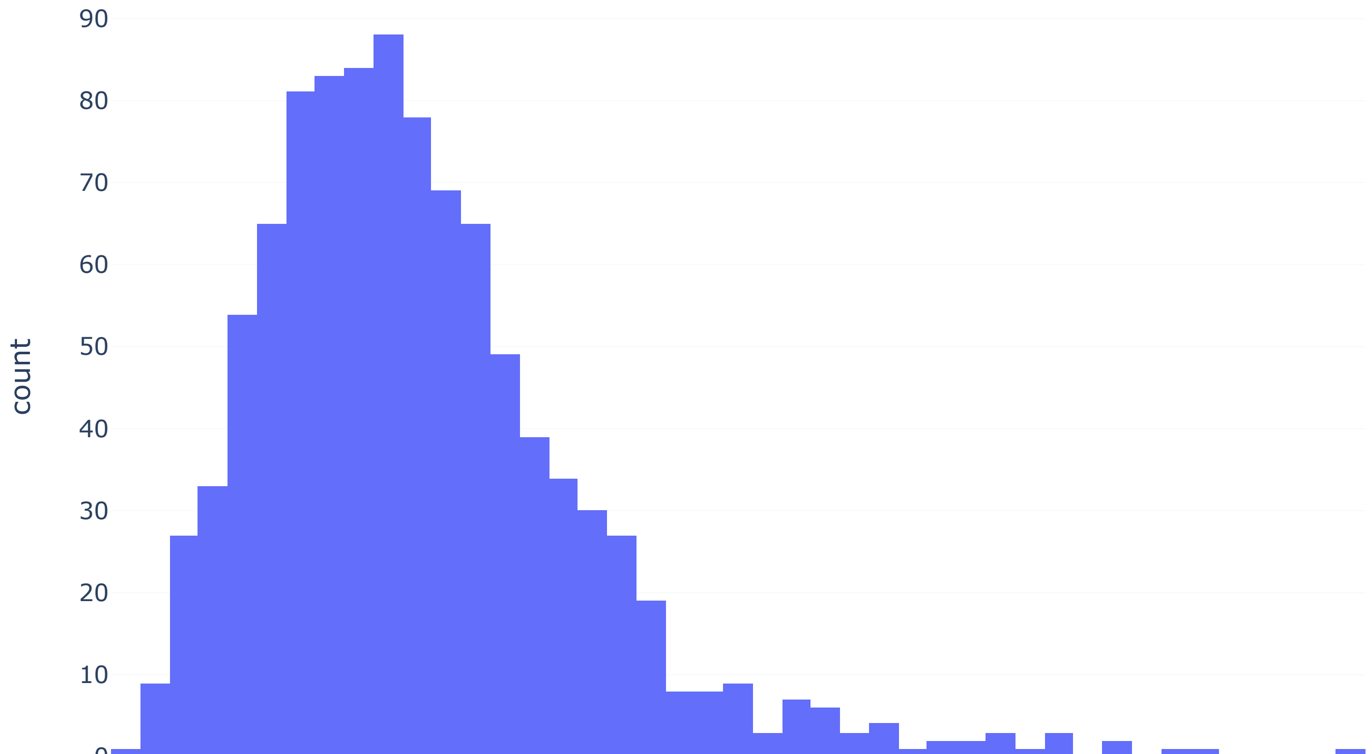
```python
num_sims = 1000
np.random.seed(0)

rets = np.random.normal(
    loc=mean,
    scale=stdev,
    size=(num_prds, num_sims)
)
compound_rets = np.prod((1+rets), axis=0) - 1

import pandas as pd
pd.Series(compound_rets).describe()
```

Out[85]:

```
count    1000.000000
mean        1.525291
std         1.123597
min        -0.464922
25%         0.737312
50%         1.330335
75%         2.061266
max         7.906556
dtype: float64
```

In [87]:

```
import plotly.express as px

fig = px.histogram(compound_rets)
fig.update_layout(
    showlegend=False,
    template="plotly_white"
)
fig.show()
```



RISK IN THE LONG RUN

- Law of large numbers does not eliminate risk (uncertainty) in the long run
- Law of large numbers applies to average of gambles, not the sum
    - So it applies to the average return, not the cumulative return
    - Theorem: a random walk walks everywhere!
- However, if the game is in your favor (the house at a casino or the stock market) and you play a long time, it is very unlikely you will end with less than you start.

https://learn-investments.rice-business.org/risk/long-run

MOMENTS OF DISTRIBUTIONS

- Non-central moments
  - first $= \text{mean} = E[x]$
  - second $= E[x^2]$
  - third $= E[x^3]$
  - fourth $= E[x^4]$
- Central moments
  - second $= \text{variance} = E[(x - \text{mean})^2]$
  - third $= E[(x - \text{mean})^3]$
  - fourth $= E[(x - \text{mean})^4]$

- Standardized moments
  - third $= \text{skewness} = E[(x - \text{mean})^3] / \text{stdev}^3$
  - fourth $= \text{kurtosis} = E[(x - \text{mean})^4] / \text{stdev}^4$

EXAMPLE 1: NORMAL

In [88]:

```
# central moments

np.random.seed(0)
x = np.random.normal(size=100000)
mean = np.mean(x)
variance = np.mean(
    (x-mean)**2
)
third = np.mean(
    (x-mean)**3
)
fourth = np.mean(
    (x-mean)**4
)
print(f"mean={mean:.2f}, variance={variance:.2f}, third={third:.2f}, fourth={
```

```
mean=0.00, variance=0.99, third=-0.01, fourth=3.00
```

In [89]:

```python
# standardized central moments

stdev = np.sqrt(variance)

skewness = third / stdev**3

kurtosis = fourth / stdev**4

print(f"stdev={stdev:.2f}, skewness={skewness:.2f}, kurtosis={kurtosis:.2f}")
```

```
stdev=1.00, skewness=-0.01, kurtosis=3.03
```

Theorem

For any normal distribution, skewness = 0 and kurtosis = 3.

Adjusted Kurtosis and Leptokurtosis

- The common definition of kurtosis is

$$\frac{E[(x - \text{mean})^4]}{\text{stdev}^3} - 3$$

- With this definition, the kurtosis of a normal distribution is 0.
- Positive kurtosis means unadjusted kurtosis > 3. This is often called "excess kurtosis."
- Distributions with positive kurtosis are called leptokurtic. Or fat tailed.

EXAMPLE 2: LOGNORMAL

In [90]:

```python
# central moments

y = np.exp(x)
mean = np.mean(y)
variance = np.mean(
    (y-mean)**2
)
third = np.mean(
    (y-mean)**3
)
fourth = np.mean(
    (y-mean)**4
)
print(f"mean={mean:.2f}, variance={variance:.2f}, third={third:.2f}, fourth={
```

```
mean=1.65, variance=4.60, third=55.13, fourth=1429.58
```

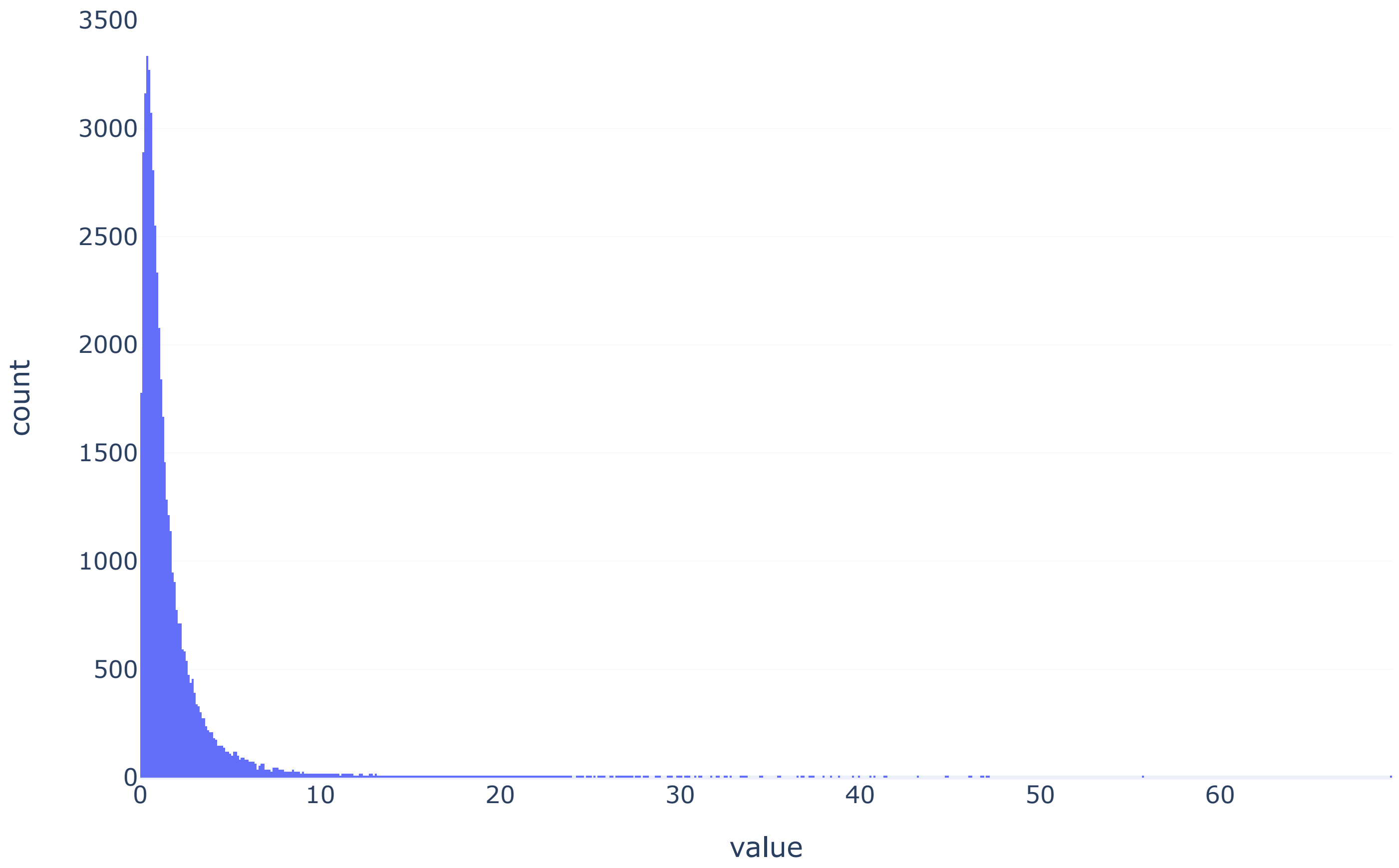
In [91]:

```
# standardized central moments

stdev = np.sqrt(variance)

skewness = third / stdev**3

kurtosis = fourth / stdev**4

print(f"stdev={stdev:.2f}, skewness={skewness:.2f}, kurtosis={kurtosis:.2f}")
```

```
stdev=2.15, skewness=5.58, kurtosis=67.44
```

In [92]:

```
fig = px.histogram(y)
fig.update_layout(
    showlegend=False,
    template="plotly_white"
)
fig.show()
```



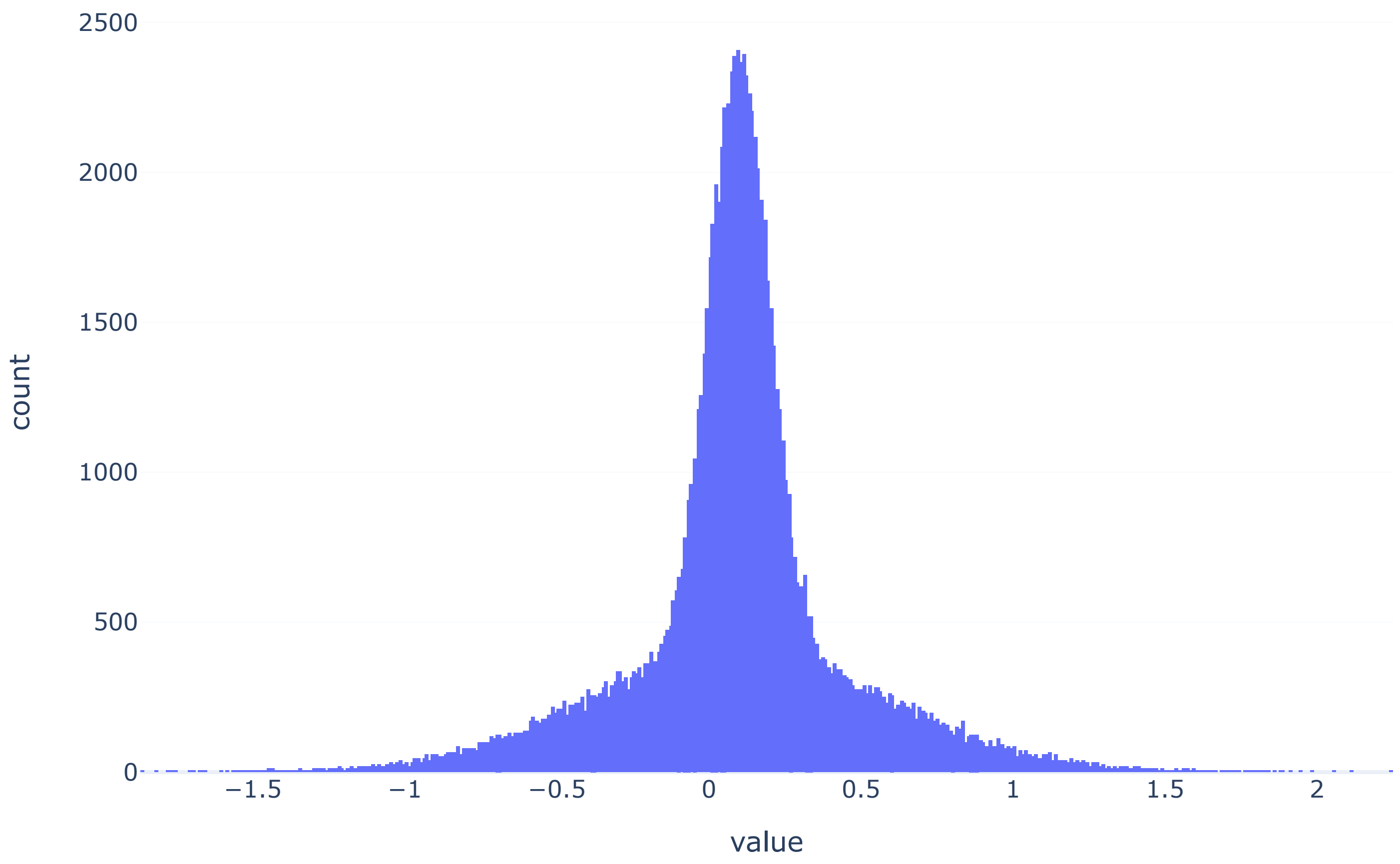
EXAMPLE 3: MIXTURE

In [93]:

```
np.random.seed(0)
x1 = np.random.normal(
    loc=0.1,
    scale=0.1,
    size=100000
)
x2 = np.random.normal(
    loc=0.1,
    scale=0.5,
    size=100000
)
z = np.random.randint(2, size=100000)
y = np.where(z, x1, x2)
```

In [94]:

```
fig = px.histogram(y)
fig.update_layout(showlegend=False, template="plotly_white"
)
fig.show()
```



In [95]:

```
# central moments

mean = np.mean(y)
variance = np.mean(
    (y-mean)**2
)
third = np.mean(
    (y-mean)**3
)
fourth = np.mean(
    (y-mean)**4
)
print(f"mean={mean:.2f}, variance={variance:.2f}, third={third:.2f}, fourth={
```

```
mean=0.10, variance=0.13, third=0.00, fourth=0.09
```

In [96]:

```
# standardized central moments

stdev = np.sqrt(variance)

skewness = third / stdev**3

kurtosis = fourth / stdev**4

print(f"stdev={stdev:.2f}, skewness={skewness:.2f}, kurtosis={kurtosis:.2f}")
```

```
stdev=0.36, skewness=0.01, kurtosis=5.56
```