Section 8

Date	
Property	
Weeks	4주차

1. Choosing the policy portfolio

- performance seeking portfolio vs liability hedging portfolio
  - 문제는 risk aversion coefficient "gamma" → 관측 불가능, 현실에 마땅히 대응하는 친구가 없음
- risk aversion parameter as parameter: 우리가 임의로 정하자!
  - 실무적으로는 risk-budget을 책정해 놓고, 거기서 최대PSP 포트폴리오를 구성함
  - 이 예산은 agent에 의해서 결정됨 - 연금이나, 누가 물주이냐 이런거에 따라서
- long-term vs short-term
  - "언제까지의 수익률"로 맞추냐에 따라서 값들이 달라짐
- dollar vs risk-budget
  - 돈 많이 부으면 좋은데, 무턱대고 부을 수는 없어서

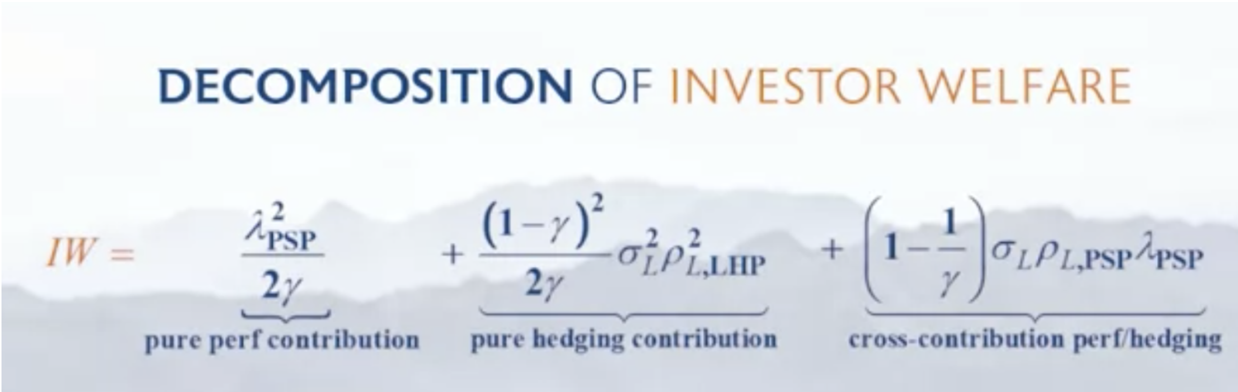
2. Lab Session-Monte Carlo simulation of coupon-bearing bonds using CIR

다른 벡터화된 인풋이 들어와도 코드가 동작할 수 있도록

시뮬레이션 했어요... 그랬다구요

3. Beyond LDI

- PSP 와 LHP를 잘 섞은 어떤 조합이 있지 않을까?
- 그것을 Investor Welfare라고 불러보자... 그러면 그 함수는 대충 이렇게 생겼을 것이다.



- 첫째 항은 PSP의 Sharpe-ratio를 나타내는 부분
- 두번째 항은 liability와 선택한 hedging수단의 의 상관관계. 이 숫자가 클수록 헷징을 잘 하고 있다는 뜻
- 세번째 항은... 따로 노는 것 같은 두 부분간의 correlation을 의미하는 부분. 두 항이 서로 같다면 상관성인

4. Lab Session-Naive risk budgeting between the PSP & GHP

how to write allocators

how to write strategies

how to backtest strategies that construct portfolios that mix the PSP and GHP

▼ def

두 포트폴리오에서 return을 가져와 봅시다. 얘네 둘을 섞는데 사용하는게 allocator

star kwargs : 어떤 parameter, argument 도 받아들이게?

```
def bt_mix(r1, r2, allocator, **kwargs):
    """
    Runs a back test (simulation) of allocating between a two sets of returns
    r1 and r2 are T x N DataFrames or returns where T is the time step index and N is the number of scenarios.
    allocator is a function that takes two sets of returns and allocator specific parameters, and produces
    an allocation to the first portfolio (the rest of the money is invested in the GHP) as a T x 1 DataFrame
    Returns a T x N DataFrame of the resulting N portfolio scenarios
    """
    if not r1.shape == r2.shape:
        raise ValueError("r1 and r2 should have the same shape")
    weights = allocator(r1, r2, **kwargs)
    if not weights.shape == r1.shape:
        raise ValueError("Allocator returned weights with a different shape than the returns")
    r_mix = weights*r1 + (1-weights)*r2
    return r_mix

def fixedmix_allocator(r1, r2, w1, **kwargs):
    """
    w1 is the weights in the first portfolio.
    Produces a time series over T steps of allocations between the PSP and GHP across N scenarios
```

```
    PSP and GHP are T x N DataFrames that represent the returns of the PSP and GHP such that:
    each column is a scenario
    each row is the price for a timestep
    Returns an T x N DataFrame of PSP Weights
    """
    return pd.DataFrame(data = w1, index=r1.index, columns=r1.columns)
```

```
def terminal_values(rets):
    """
    Computes the terminal values from a set of returns supplied as a T x N DataFrame
    Return a Series of length N indexed by the columns of rets
    """
    return (rets+1).prod()

def terminal_stats(rets, floor = 0.8, cap=np.inf, name="Stats"):
    """
    Produce Summary Statistics on the terminal values per invested dollar
    across a range of N scenarios
    rets is a T x N DataFrame of returns, where T is the time-step (we assume rets is sorted by time)
    Returns a 1 column DataFrame of Summary Stats indexed by the stat name
    """
    terminal_wealth = (rets+1).prod()
    breach = terminal_wealth < floor
    reach = terminal_wealth >= cap
    p_breach = breach.mean() if breach.sum() > 0 else np.nan
    p_reach = breach.mean() if reach.sum() > 0 else np.nan
    e_short = (floor-terminal_wealth[breach]).mean() if breach.sum() > 0 else np.nan
    e_surplus = (cap-terminal_wealth[reach]).mean() if reach.sum() > 0 else np.nan
    sum_stats = pd.DataFrame.from_dict({
        "mean": terminal_wealth.mean(),
        "std" : terminal_wealth.std(),
        "p_breach": p_breach,
        "e_short":e_short,
        "p_reach": p_reach,
        "e_surplus": e_surplus
    }, orient="index", columns=[name])
    return sum_stats

def glidepath_allocator(r1, r2, start_glide=1, end_glide=0.0):
    """
    Allocates weights to r1 starting at start_glide and ends at end_glide
    by gradually moving from start_glide to end_glide over time
    """
    n_points = r1.shape[0]
    n_col = r1.shape[1]
    path = pd.Series(data=np.linspace(start_glide, end_glide, num=n_points))
    paths = pd.concat([path]*n_col, axis=1)
    paths.index = r1.index
    paths.columns = r1.columns
    return paths
```

▼ 해본다

```
In [8]: pd.concat([erk.terminal_stats(rets_bonds, name="FI"),
                   erk.terminal_stats(rets_eq, name="Eq"),
                   erk.terminal_stats(rets_7030b, name="70/30")],
                  axis=1)
```

Out[8]:

	FI	Eq	70/30
mean	1.383025	1.873734	1.722892
std	0.107982	0.898662	0.571827
p_breach	NaN	0.052000	0.006000
e_short	NaN	0.110863	0.089708
p_reach	NaN	NaN	NaN
e_surplus	NaN	NaN	NaN

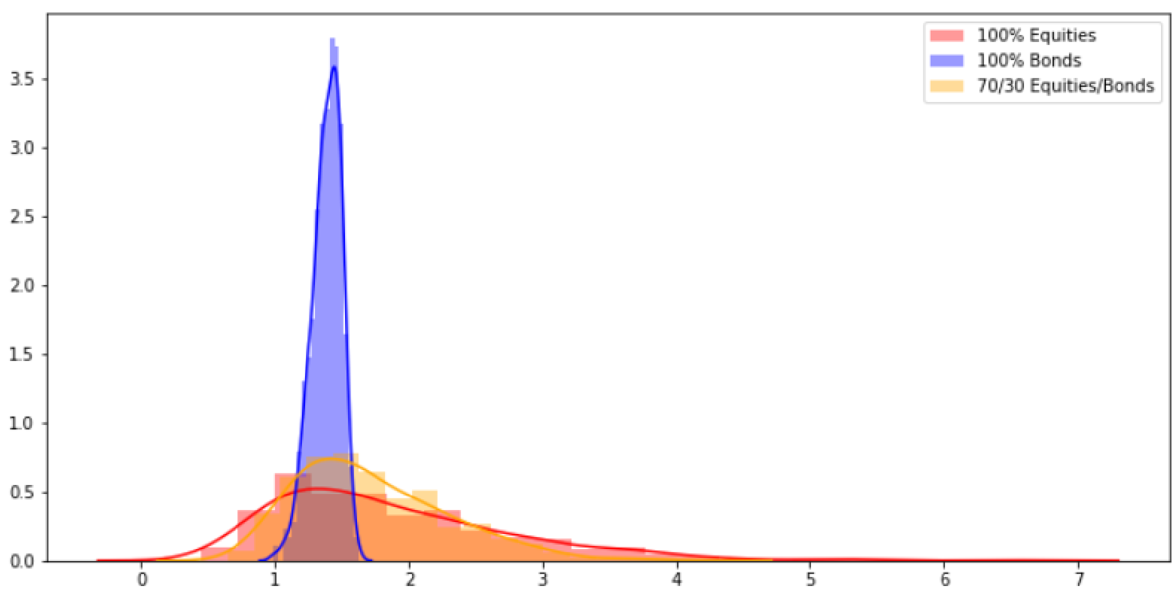
breach = terminal_wealth < floor

p_breach = breach.mean()

e_short = (floor-terminal_wealth[breach]).mean()

e_surplus = (cap-terminal_wealth[reach]).mean()

```
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))
sns.distplot(erk.terminal_values(rets_eq), color="red", label="100% Equities")
sns.distplot(erk.terminal_values(rets_bonds), color="blue", label="100% Bonds")
sns.distplot(erk.terminal_values(rets_7030b), color="orange", label="70/30 Equities/Bonds")
plt.legend();
```



100% 주식 : 너무 위험 100% 채권 : 너무 보수적 7030 : 너무 고정적

→ 시간에 따라 가중치를 변화시키자

예를들면 처음에 주식 80 채권 20에서 시작해서 10년 뒤엔 주식 20 채권 80으로..

```
def glidepath_allocator(r1, r2, start_glide=1, end_glide=0.0):
    """
    Allocates weights to r1 starting at start_glide and ends at end_glide
    by gradually moving from start_glide to end_glide over time
    """
    n_points = r1.shape[0]
    n_col = r1.shape[1]
    path = pd.Series(data=np.linspace(start_glide, end_glide, num=n_points))
    paths = pd.concat([path]*n_col, axis=1)
    paths.index = r1.index
    paths.columns = r1.columns
    return paths
```

In [11]:

```
rets_g8020 = erk.bt_mix(rets_eq, rets_bonds, allocator=erk.glidepath_allocator, start_glide=.8, end_glide=.2)
pd.concat([erk.terminal_stats(rets_bonds, name="FI"),
           erk.terminal_stats(rets_eq, name="Eq"),
           erk.terminal_stats(rets_7030b, name="70/30"),
           erk.terminal_stats(rets_g8020, name="Glide 80 to 20")],
          axis=1)
```

Out[11]:

	FI	Eq	70/30	Glide 80 to 20
mean	1.383025	1.873734	1.722892	1.633690
std	0.107982	0.898662	0.571827	0.427764
p_breach	NaN	0.052000	0.006000	0.002000
e_short	NaN	0.110863	0.089708	0.035501
p_reach	NaN	NaN	NaN	NaN
e_surplus	NaN	NaN	NaN	NaN

5. Liability-friendly equity portfolios

6. Lab Session-Dynamic risk budgeting between PSP & LHP

전에 했던 CPPI랑 비슷합니다

앞에 했던 거랑도 비슷한데 좀더 현실적인 allocator들을 생각해 봅시다.

1. floor allocator "쿠션이 있으면 PSP에 더 많은 할당을 준다."

```
def floor_allocator(psp_r, ghp_r, floor, zc_prices, m=3):
    """
    Allocate between PSP and GHP with the goal to provide exposure to the upside
    of the PSP without going violating the floor.
    Uses a CPPI-style dynamic risk budgeting algorithm by investing a multiple
    of the cushion in the PSP
    Returns a DataFrame with the same shape as the psp/ghp representing the weights in the PSP
    """
    if zc_prices.shape != psp_r.shape:
```

```
        raise ValueError("PSP and ZC Prices must have the same shape")
    n_steps, n_scenarios = psp_r.shape
    account_value = np.repeat(1, n_scenarios)
    floor_value = np.repeat(1, n_scenarios)
    w_history = pd.DataFrame(index=psp_r.index, columns=psp_r.columns)
    for step in range(n_steps):
        floor_value = floor*zc_prices.iloc[step] ## PV of Floor assuming today's rates and flat YC
        cushion = (account_value - floor_value)/account_value
        psp_w = (m*cushion).clip(0, 1) # same as applying min and max
        ghp_w = 1-psp_w
        psp_alloc = account_value*psp_w
        ghp_alloc = account_value*ghp_w
        # recompute the new account value at the end of this step
        account_value = psp_alloc*(1+psp_r.iloc[step]) + ghp_alloc*(1+ghp_r.iloc[step])
        w_history.iloc[step] = psp_w
    return w_history
```

In [5]:

```
rets_floor75m5 = erk.bt_mix(rets_eq, rets_zc, allocator=erk.floor_allocator, zc_prices=zc_prices[1:], floor=.75, m=5)
rets_floor75m10 = erk.bt_mix(rets_eq, rets_zc, allocator=erk.floor_allocator, zc_prices=zc_prices[1:], floor=.75, m=10)
pd.concat([erk.terminal_stats(rets_zc, name="ZC", floor=0.75),
           erk.terminal_stats(rets_eq, name="Eq", floor=0.75),
           erk.terminal_stats(rets_7030b, name="70/30", floor=0.75),
           erk.terminal_stats(rets_floor75, name="Floor75", floor=0.75),
           erk.terminal_stats(rets_floor75m1, name="Floor75m1", floor=0.75),
           erk.terminal_stats(rets_floor75m5, name="Floor75m5", floor=0.75),
           erk.terminal_stats(rets_floor75m10, name="Floor75m10", floor=0.75)
          ],
          axis=1).round(2)
```

Out[5]:

	ZC	Eq	70/30	Floor75	Floor75m1	Floor75m5	Floor75m10
mean	1.34	1.95	1.74	1.92	1.61	1.93	1.93
std	0.00	0.95	0.58	0.96	0.42	0.96	0.96
p_breach	NaN	0.04	0.01	NaN	NaN	0.00	0.02
e_short	NaN	0.12	0.08	NaN	NaN	0.00	0.00
p_reach	NaN	NaN	NaN	NaN	NaN	NaN	NaN
e_surplus	NaN	NaN	NaN	NaN	NaN	NaN	NaN

2. drawdown_allocator "고정된 floor (x) "

```
def drawdown_allocator(psp_r, ghp_r, maxdd, m=3):
    """
    Allocate between PSP and GHP with the goal to provide exposure to the upside
    of the PSP without going violating the floor.
    Uses a CPPI-style dynamic risk budgeting algorithm by investing a multiple
    of the cushion in the PSP
    Returns a DataFrame with the same shape as the psp/ghp representing the weights in the PSP
    """
    n_steps, n_scenarios = psp_r.shape
    account_value = np.repeat(1, n_scenarios)
    floor_value = np.repeat(1, n_scenarios)
    ### For MaxDD
    peak_value = np.repeat(1, n_scenarios)
    w_history = pd.DataFrame(index=psp_r.index, columns=psp_r.columns)
    for step in range(n_steps):
        ### For MaxDD
        floor_value = (1-maxdd)*peak_value ### Floor is based on Prev Peak
        cushion = (account_value - floor_value)/account_value
        psp_w = (m*cushion).clip(0, 1) # same as applying min and max
        ghp_w = 1-psp_w
        psp_alloc = account_value*psp_w
        ghp_alloc = account_value*ghp_w
        # recompute the new account value at the end of this step
```

```
        account_value = psp_alloc*(1+psp_r.iloc[step]) + ghp_alloc*(1+ghp_r.iloc[step])
        ### For MaxDD
        peak_value = np.maximum(peak_value, account_value) ### For MaxDD
        w_history.iloc[step] = psp_w
    return w_history
```