Lazy Portfolio Allocation Algorithm

Andrew Brown

January 2, 2022

1 The Lazy Portfolio Allocation Algorithm

The Lazy Portfolio Allocation Algorithm, first described by Albert H. Mao [1], is a way of keeping your portfolio balanced with regular contributions or withdraws, without the need for any explicit balancing operations. It takes a list of assets (which could be funds, accounts, etc), their current values, and their target or *desired* allocation percentage, as well as an amount to contribute. The algorithm then calculates the optimal way to split up the contribution between the assets such as to minimize each asset's deviation from the desired allocation.

[1] Optimal lazy portfolio rebalancing calculator

My goal is to expand on the original algorithm description, providing derivation of the formulas, an annotated reference implementation, and examples.

1.1 Definitions

- a_n is the starting or "actual" amount in dollars in asset n
- d_n is the desired allocation of asset n as a fraction in the range $0 \dots 1$
- C is the amount in dollars to contribute, or a negative amount to withdraw
- $T = \sum a + C$ is the total amount in all assets after the contributions
- $t_n = T d_n$ is the target or "ideal" dollar amount in asset n

1.2 Fractional Deviation

The metric we want to optimize for each asset is

$$f_n = a_n / t_n \tag{1}$$

which we call the *fractional deviation* or the fraction of an asset's current value out of its target value. This value will be less than 1 for under-allocated assets, and greater than 1 for over-allocated assets.

The strategy is to first sort the assets by their fractional deviations such that the asset with the lowest deviation value—or most under-allocated—is first. Then contribute to the first asset an amount such that its fractional deviation is brought equal to that of the second asset. Then repeat for the first two assets: contribute an equal portion to assets one and two until their deviation equals the third asset, and so on.

To bring asset 1 up to asset 2's deviation, add to it

$$t_1 \frac{a_2}{t_2} - a_1$$

a_2/t_2 is asset 2's deviation, so $t_1 a_2/t_2$ is the amount of money for asset 1 which would have the same deviation as asset 2. Then subtract a_1—the amount currently in asset 1—to get the amount to *add* to asset 1.

We can re-arrange and simplify this expression using our definition for the fractional deviation (1) to get:

$$\begin{aligned} t_1 \frac{a_2}{t_2} - a_1 &= t_1 \left(\frac{a_2}{t_2} - \frac{a_1}{t_1} \right) \\ &= t_1 (f_2 - f_1) \end{aligned}$$

We now have a general expression for the amount to add to asset n to bring its deviation up to the deviation of asset m: add to it

$$t_n (f_m - f_n) \tag{2}$$

After, the value of asset n will be $t_n a_m / t_m$, which is off from its target value by the same proportion as asset m.

> **Note:** In the original description of the algorithm [1], Mao defines the fractional deviation as $a_n/t_n - 1$ instead of a_n/t_n. However, this makes no difference to the equations derived in this paper, since they use the difference between two f values.

1.3 Contributions on Each Step

Now we have the foundation to build the algorithm. We'll break things down into steps. Step 1 contributes to asset 1 until its fractional deviation reaches asset 2's. Step 2 contributes to assets 1 *and* 2 until their fractional deviations reach asset 3's. The steps repeat in this fashon until all contributions are used up.

We know from (2) that the amount to add to asset 1 to bring it up to asset 2's deviation is

$$t_1 (f_2 - f_1)$$

which is also the total amount contributed in step 1.

In step 2, we need to bring assets 1 *and* 2 up to the fractional deviation of step 3. To compute this, it's easier to compute the *total* amount to contribute to assets 1 *and* 2 for *both* steps 1 and 2, by using (2) to bring both assets up to fractional deviation f_3:

$$t_1 (f_3 - f_1) + t_2 (f_3 - f_2)$$

and subtract back out the contributions from step 1:

$$\begin{aligned} t_1(f_3 - f_1) + t_2(f_3 - f_2) - t_1(f_2 - f_1) &= t_1 f_3 - t_1 f_1 + t_2 f_3 - t_2 f_2 - t_1 f_2 + t_1 f_1 \\ &= t_1 f_3 + t_2 f_3 - t_2 f_2 - t_1 f_2 \\ &= t_1(f_3 - f_2) + t_2(f_3 - f_2) \\ &= (t_1 + t_2)(f_3 - f_2) \end{aligned}$$

And thus we have an expression for how much more to contribute in *just* step 2.

We can generalize this pattern. Define TC_n as the Total Contributions made after step n (across all steps $1 \dots n$):

$$TC_n = \sum_{i=1}^{n} t_i(f_{n+1} - f_i) \tag{3}$$

This formula adds to each asset $1 \dots n$ an amount to bring the respective fractional deviations up to that of asset $n+1$. Then, to figure out the contributions for an individual step n, we compute:

$$\begin{aligned} TC_n - TC_{n-1} &= \sum_{i=1}^{n} t_i(f_{n+1} - f_i) - \sum_{i=1}^{n-1} t_i(f_n - f_i) \\ &= \sum_{i=1}^{n-1} t_i(f_{n+1} - f_i) + t_n(f_{n+1} - f_n) - \sum_{i=1}^{n-1} t_i(f_n - f_i) \\ &= \sum_{i=1}^{n-1} \left[t_i(f_{n+1} - f_i) - t_i(f_n - f_i)\right] + t_n(f_{n+1} - f_n) \\ &= \sum_{i=1}^{n-1} \left[t_i f_{n+1} - t_i f_i - t_i f_n + t_i f_i\right] + t_n(f_{n+1} - f_n) \\ &= \sum_{i=1}^{n-1} \left[t_i f_{n+1} - t_i f_n\right] + t_n(f_{n+1} - f_n) \\ &= \sum_{i=1}^{n-1} \left[t_i(f_{n+1} - f_n)\right] + t_n(f_{n+1} - f_n) \\ &= \sum_{i=1}^{n} \left[t_i\right](f_{n+1} - f_n) \end{aligned}$$

We define a new label r_n to be the "running total" of all asset target values from $1 \dots n$:

$$r_n = \sum_{i=1}^{n} t_i \tag{4}$$

and so we can write the expression for the additional contributions added in step n over step $n-1$ across all assets $1 \dots n$:

$$TC_n - TC_{n-1} = r_n(f_{n+1} - f_n) \tag{5}$$

1.4 Computing the number of steps

The algorithm should stop at a particular step s if the amount needed for the next step (TC_{s+1}) equals or exceeds the original contribution amount C. So we must find the largest s such that

$$TC_s < C$$
$$\sum_{i=1}^{s} t_i(f_{s+1} - f_i) < C$$

We could do this by iteratively trying successive values for s and computing TC_s, however since the definition for TC_n involves a summation, this isn't very efficient. We can instead use the expression for $TC_n - TC_{n-1}$ (5) that uses the running total r_n (4) to compute the next TC_n from the previous TC_{n-1}

$$TC_n = r_n(f_{n+1} - f_n) + TC_{n-1}$$
$$r_n = r_{n-1} + t_n$$

So each iteration of the algorithm we compute the new r_n and the new TC_n. If TC_n equals or exceeds C then the algorithm must stop at $n-1$.

> **Note:** The contributions C may not exactly equal TC at any particular step, in which case we'll have some money left over to distribute among the assets $1 \dots s+1$. We'll see how to compute this in the next section. To make the implementation easier, the stop condition is $TC_s < C$ instead of $TC_s \leq C$ so that our stopping step s will always leave at least the final asset for use in the calculation for the "extra" money, even if $TC_s = C$ exactly.

1.5 Computing the Contributions

Once we have found the number of steps s (the largest integer such that $TC_s < C$) then by the definition of step s, each asset $1 \dots s$ will be at fractional deviation f_{s+1}. Therefore, by (2) the amount to contribute to each asset n is given by

$$\Delta_n = \begin{cases} t_n(f_{s+1} - f_n) & \text{if } n \leq s \\ 0 & \text{if } n > s \end{cases}$$

However, this only contributes a total of TC_s dollars to assets $1 \dots s$. There is still $C - TC_s$ left to contribute. Since all assets $1 \dots s+1$ are at fractional deviation f_{s+1}, the leftover money is distributed proportionally across these $s+1$ assets.

We do this by finding some new fractional deviation to increase assets $1 \dots s+1$ to such that it uses all our contributions exactly. We can do this by solving this equation.

$$C = \sum_{i=1}^{s+1} [t_i(f_{s+1} + X - f_i)]$$

In other words, we increase each asset's deviation to some amount over f_{s+1} but $\leq f_{s+2}$ since then we would have increased the step s instead.

Now we can solve for X:

$$\begin{aligned}
C &= \sum_{i=1}^{s+1} [t_i(f_{s+1} + X - f_i)] \\
C &= \sum_{i=1}^{s+1} [t_i(f_{s+1} - f_i)] + \sum_{i=1}^{s+1} [t_i X] \\
C &= \sum_{i=1}^{s} [t_i(f_{s+1} - f_i)] + t_{s+1}(f_{s+1} - f_{s+1}) + X r_{s+1} \\
C &= TC_s + X r_{s+1} \\
C - TC_s &= X r_{s+1} \\
\frac{C - TC_s}{r_{s+1}} &= X
\end{aligned}$$

So after we find our final step s, we calculate the final fractional deviation value f_f as

$$f_f = f_{s+1} + \frac{C - TC_s}{r_{s+1}} \tag{6}$$

and then calculate the final delta values as

$$\Delta_n = \begin{cases} t_n(f_f - f_n) & \text{if } n \leq s+1 \\ 0 & \text{if } n > s+1 \end{cases} \tag{7}$$

2 Implementation

Presented here is a complete and commented implementation of the Lazy Allocation algorithm. The code handles a few edge cases not yet described, and will be explained in the following sections.

[1]:
```
from fractions import Fraction
from dataclasses import dataclass
from typing import List

# This dataclass will hold information about each asset. All values are stored
# using the Python Fraction class, which stores rational numbers as two
# arbitrary precision integers in order to avoid arithmetic rounding errors.
@dataclass
class Asset:
    # Actual (current) amount in dollars
    a: Fraction

    # Desired allocation, as a fraction between 0 and 1
    # (all asset allocations should sum to 1)
    d: Fraction
```

```
    # Computed target amount in dollars
    t: Fraction = None

    # Computed fractional deviation
    f: Fraction = None

    # Computed amount to add to this asset
    delta: Fraction = None



def lazy_alloc(assets: List[Asset],
               C: Fraction) -> List[Fraction]:
    """The Lazy Asset Allocation Algorithm

    :param assets: A list of Asset objects with the 'actual' and 'allocation'
        values filled in

    :param C: The amount in dollars to contribute

    :returns: The list of 'delta' values, or amount to contribute
        to each asset

    """
    # Compute the 'target' and 'f' values for each asset.
    # Also annotate each asset with its index in the list, so we remember
    # the original asset ordering
    # Values we use are converted into Fraction types if not already.
    T = sum(Fraction(asset.a) for asset in assets) + Fraction(C)
    for asset_index, asset in enumerate(assets):
        # To allow for a desired allocation of 0%, we have to set a minimum
        # target value. A value of exactly 0 for the target would cause a zero
        # division error when calculating the fractional deviation.
        asset.t = T * Fraction(asset.d) or Fraction("0.001")
        asset.f = Fraction(asset.a) / asset.t
        asset.i = asset_index
    C = Fraction(C)

    # Now we can order the assets by their fractional deviation, reversing the
    # direction if we're withdrawing.
    assets.sort(key=lambda asset: asset.f)
    if C < 0:
        assets.reverse()

    # Each loop iteration computes these values for the current step.
    # Note that Python lists are 0-indexed, and we start at step 0.
    # Each loop iteration computes the TC for the current step, but if it
```

```
    # exceeds C (or is the last step), then we "back up" and use prev_TC -- the
    # previous step's TC -- to calculate the final fractional deviation.
    step = 0
    r = 0
    prev_TC = 0

    while True:
        # Update this step's fractional deviation and the running total values
        this_f = assets[step].f
        r += assets[step].t

        # First exit condition: if this is the last asset, then exit the loop.
        # If we exit here, it indicates we have more money than required to
        # bring all assets up to the last asset's fractional deviation. The
        # remaining money (C - prev_TC) will be distributed to all assets.
        if step + 1 == len(assets):
            break

        # Calculate the total contributions for the current step, so we can
        # compare it against the contributions.
        next_f = assets[step + 1].f
        TC = prev_TC + r * (next_f - this_f)

        # Second exit condition: TC for the current step exceeds the
        # contributions. We've found the maximum step as described in the
        # equations, which is `step-1`.
        # We compare the absolute values since the signs are inverted when
        # withdrawing.
        if abs(TC) >= abs(C):
            break

        # Increment the loop variables for the next loop iteration
        step += 1
        prev_TC = TC

    # We've exited the loop. The variable `step` has gone one *past* the final
    # step as described in the equations. We therefore use prev_TC to compute
    # the final fractional deviation.
    f_f = this_f + (C - prev_TC) / r

    for asset_index, asset in enumerate(assets):
        # Again since our actual final step is `step-1`, we update all
        # assets <=step (instead of `step+1` as given in the equations)
        if asset_index <= step:
            # Values are rounded to the nearest cent. This could cause the total
            # of all delta values to not sum to the contributions. If this is
            # a problem, take out the round() call for exact answers.
```

```
            asset.delta = round(asset.t * (f_f - asset.f), 2)
        else:
            asset.delta = Fraction(0)

    # Re-order the assets to their original ordering
    assets.sort(key=lambda asset: asset.i)

    # And return the delta values
    return [asset.delta for asset in assets]
```

2.1 Withdraws

The basic algorithm works just fine in reverse with the following modifications:

1. When withdrawing, we want to take money from the asset that is most *over*-allocated first, or has the *highest* fractional deviation. Therefore, we reverse the ordering of the assets after sorting by the fractional deviation.
2. When TC and C are compared, we must compare the absolute values of each, since the signs are reversed when withdrawing.

2.2 0% Allocations

Sometimes it's useful to have an asset with 0% desired allocation, meaning no money will ever be contributed to it, and any withdraws should come from it first. This presents a problem, since the asset will have a target value

$$\begin{aligned} t_n &= Td_n \\ &= 0 \end{aligned}$$

and a fractional deviation of

$$\begin{aligned} f_n &= \frac{a_n}{t_n} \\ &= \frac{a_n}{0} \end{aligned}$$

To work around this, if the target value t_n is calculated to be 0, then the code sets it to 0.001. The algorithm then works just the same, but the extra tenth of a penny is rounded away at the end.

2.3 Rounding

This implementation is careful to avoid operations on floating point or decimal numbers, and therefore avoids implicit rounding and imprecise representation errors inherint in floating point

math. At the end, the delta values are explicitly rounded to two decimal places, partly to account for 0% allocations, but also just because currency values are typically only useful to two decimal places. The downside is that the total of all delta values may not exactly sum to the given contribution; it may be off by 0.01. For example, if contributing $100 to three assets, the values would be 33.33, 33.33, and 33.33, totaling 99.99.

If this is not desired, the rounding may be removed. Then the answer would be 100/3 for the above example.

When porting the code to an environment which lacks a comparible exact arithmetic library, the algorithm will still work, but rounding errors may cause slightly less precise results.

3 Examples

This section will go over some examples to illustrate the algorithm. First we'll define a function to print an array of Fraction values displayed as decimals rounded to two places:

[2]:
```
def print_fraction_list(l):
    print(", ".join(
        "${:.2f}".format(float(i))
        for i in l
    ))
```

Now we can try out a few examples. Here's a simple example where two assets start with different values, but we want them to have equal allocation: 50% and 50%.

[3]:
```
print_fraction_list(lazy_alloc(
    [Asset(100, 0.5), Asset(200, 0.5)],
    100
))
```

```
$100.00, $0.00
```

The result tells us that if we have $100.00 to contribute, add it all to the first asset, and none to the second.

This example is the same as before, but this time we contribute $110.00, more than enough to balance the portfolio.

[4]:
```
print_fraction_list(lazy_alloc(
    [Asset(100, 0.5), Asset(200, 0.5)],
    110,
))
```

```
$105.00, $5.00
```

$100.00 is added to the first asset as before, and the remaining $10.00 is distributed evenly among the two assets.

In this example, if not enough contributions are given, it's all added to the most under-allocated asset.

[5]:
```
print_fraction_list(lazy_alloc(
    [Asset(100, 0.5), Asset(200, 0.5)],
    50
))
```

```
$50.00, $0.00
```

Here we show what happens when different allocations are desired. Both assets start with the same amount, but the desired allocations are 75% and 25%.

[6]:
```
print_fraction_list(lazy_alloc(
    [Asset(100, 0.75), Asset(100, 0.25)],
    200
))
```

```
$200.00, $0.00
```

Note how all $200 are contributed to the first asset, which would bring both asset amounts to $300 and $100, balancing the portfolio to the desired 75% 25%.

Same starting conditions and desired allocations, but with more money contributed. Any additional money is distributed proportionally between them, as show below:

[7]:
```
print_fraction_list(lazy_alloc(
    [Asset(100, 0.75), Asset(100, 0.25)],
    300,
))
```

```
$275.00, $25.00
```

The first $200 goes to the first asset as before, but the additional $100 is distributed proportionally among the two assets in order to keep them balanced at the desired 75% 25% allocations.

Let's dig into one final example with many assets of different sizes and desired allocations.

[8]:
```
print_fraction_list(lazy_alloc(
    [Asset(9000, 0.55), Asset(4000, 0.30), Asset(3500, 0.15)],
    2500
))
```

```
$1029.41, $1470.59, $0.00
```

In this example, the assets start out at these respective allocations: 55%, 24%, 21%, and we have $2,500 to contribute among them. Asset 1 is already at its target allocation, while asset 2 is under-allocated and asset 3 is over-allocated.

The algorithm tells us to add $1,029.41 to asset 1, bringing it to $10,029.41, and add $1,470.59 to asset 2, bringing it to $5,470.59. Nothing is added to asset 3.

This brings the respective asset allocations to 53%, 29%, and 18%. Note how asset 1's allocation is now off by 2%, worse than before, because it was necessary to bring the other assets closer to their targets allocations.

3.1 Withdraws

Here are some examples looking at withdrawing from the portfolio. As mentioned before, withdraws are specified with a negative contribution amount.

This example shows withdraws from two evenly allocated assets. The withdraws are taken equaly from both.

[9]:
```
print_fraction_list(lazy_alloc(
    [Asset(100, 0.5), Asset(100, 0.5)],
    -100
))
```

```
$-50.00, $-50.00
```

If one asset is over-allocated, then withdraws are taken from it first.

[10]:
```
print_fraction_list(lazy_alloc(
    [Asset(150, 0.5), Asset(100, 0.5)],
    -100
))
```

```
$-75.00, $-25.00
```

And like before, it works with many assets of different desired allocations

[11]:
```
print_fraction_list(lazy_alloc(
    [Asset(9000, 0.55), Asset(4000, 0.30), Asset(3500, 0.15)],
    -2500
))
```

```
$-1142.86, $0.00, $-1357.14
```

Let's dig into that example a bit more. The initial allocations for the three assets were 55%, 24%, and 21% respectively. Asset 1 ended up with $9000 - 1142.86 = 7857.14$ while asset 3 ended up with $3500 - 1357.14 = 2142.86$. This brings the relative allocation of the assets to 56%, 29%, and 15% respectively. Asset 2 was brought within 1% of its desired allocation without having to touch it at all!