A Program for Designing Programs for Program Trading using Dynamic Programming

Allison Bishop, Proof Trading

Abstract

Turning common constraints for program trading into a convenient mathematical form, defining a metric to optimize, and computing optimizing schedules in a reasonable time is a challenging problem. Ironically, this is why some aspects of "program trading" are often still handled by humans instead of computer programs. Here we outline our initial approach to designing an agency algo for program trading that uses flexible but conveniently structured cost estimates to allow us to compute optimized schedules quickly using dynamic programming. *And did we mention it involves programs?*

1 Introduction

A program trade is a basket of orders in individual symbols that are intended to be worked contemporaneously, within a set of holistic constraints that cannot be expressed as independent constraints on the individual orders. For example, suppose we have a basket of just two orders, a buy order in stock X and a sell order in stock Y. We may want to keep the orders roughly in sync, so that at any point in time we have sold about as much (in dollars) as we have bought (in dollars). This kind of constraint is called cash-neutrality. If we were to hand off these orders individually to algos that have discretion to trade them faster or slower based on things like current market conditions, historical volume curves, presence of substantial dark liquidity, etc., then there is no reason to think that our cash neutrality constraint will be satisfied. One way to deal with this is to remove essentially all discretion and dictate a fixed, common schedule for the two orders. This can be done by, say, using a TWAP algo that spreads volume evenly over time, or perhaps a VWAP algo that trys to track the market volume curves, if stock X and stock Y tend to follow a similar curve in how their market volume distributes over the trading day.

These kind of rigid "solutions" are rather unsatisfying. Giving up the ability to dynamically adjust trading behavior to market conditions and indications of available liquidity is bound to result in worse trading performance for both orders. To get around this, a next approach is to have a human trader essentially managing the cash neutrality constraint by portioning out pieces of the orders to algos that are given some discretion in trading each piece. For example, the the human trader may send a rather small buy order in X and a similarly small order in Y to algos. Perhaps the buy order finishes quickly, while the sell order proceeds slowly. The human trader may then change the parameters on the sell order to get it to go faster, or wait for it to catch up before sending another small buy order, etc. This way, the human can ensure that the overall deviation from cash neutrality stays within desired boundaries.

This solution is still unsatisfying though, for several reasons. For one, it is not scalable - as a pair of orders with one holistic constraint grows into, say, a basket of 500 orders with several holistic constraints, keeping track of this becomes increasingly unmanageable for a human trader. This kind of scalability challenge is made particularly frustrating by the fact that the human's actions and decisions here feel eminently automatable. But there is also a more subtle downside to this approach: since algos are being used just to trade smaller pieces of individual orders, they are operating without the fuller context, and hence the now per-piece performance that they presumably trying to optimize can diverge from the higher level goal of optimizing performance across the basket.

As an example, let's suppose that our first small buy order in X trades quickly, as the algo operating to buy X correctly recognized some "good" institutional liquidity on the other side and made a large trade. This might lead to better performance for the buy order in X, but the resulting pressure on the sell order in Y to catch up may cause worse overall performance in Y, potentially erasing all performance gains in X. Because the basket has holistic constraints that tie the orders together, what is best for one order individually may not be best for the basket as a whole. And to further apply this concept even more narrowly, what is best for one *piece* of one order individually may not be what is what is best

for that order as whole. A "good" trade that finishes a piece of the buy order may result in excellent performance for that piece, but may cause some information leakage that could harm subsequent pieces. This does not mean the underlying algo is bad! After all, we are giving it only a piece of an order at a time, so persumably it will behave as if that piece is the only thing that matters.

Generally speaking, when we use math and machine-learning to try to find the best set of actions to achieve a goal, any daylight that creeps in between our human-intuitive real-world goal and our formal mathematical expression of our goal can cause enormous problems. This is a phenomenon that we encounter in daily life, too. It's the difference between a teacher teaching to the test and legitimately covering the material. Or the passive-aggressive spouse who turns all of your white shirts pink in the laundry because you did technically say the laundry just had to be clean, not that the colors had to be properly separated. If algos are well-designed, then we should expect to get the best possible outcomes when we give them the most information and specify the most fully-aligned goals, rather than having them work on toy sub-problems of order scheduling that aren't the *real* problems we are actually interested in solving.

This leads us to the goal of designing an algo that can understand and operate on a full basket level. It may be tempting to start by automating what a human trader would do and build out a top layer of code that will parcel out smaller order pieces to our existing algos. This would likely give us reasonably good behavior and excellent scalability, but it would neglect the large potential gains that could come from the overall program scheduling being optimized holistically, rather than optimizing per-piece-per-order performance as a proxy for optimizing overall basket performance. So in the rest of this paper, we will present a framework that can be used to build a holistic optimization of program trading.

The main challenge of formulating and solving such a holistic problem is that mathematical optimization is a treacherous and disjointed field, where problems that are known to be efficiently solvable are rare and specially structured. This makes it easy to change just one seemingly tiny thing about the formulation and end up falling off the balance beam of efficient solutions and into a mathematical complexity deathtrap. The high dimensionality of our problem (needing to support basket sizes of hundreds of orders, for example) does not help. Nonetheless, we find a way to formulate a version of holistic problem space that can be solved by a combination of known efficient tools, such as dynamic programming, linear programming, etc. There are many potential improvements that we would like to explore beyond what is presented here, but we view this as a meaningful first step.

Before jumping to describing our formulation of basket-level optimization, we'll spend a little time re-conceptualizing our existing impact minimization framework for single order trading. This is because our single-order framework and its efficient dynamic programming solution have several features that position it well for expansion to program over baskets, but it was not initially formulated to accommodate enough uncertainty to cover all of the kinds of actions that we might want a PT algo to take. In particular, we made the simplistic assumption in our single-order framework (see https://prooftrading.com/docs/main-algo.pdf) that short-term scheduling decisions could be assumed to complete. In other words, if our impact-minimizing scheduler decided to schedule 400 shares in the next 10 minutes, then we would assume that these shares had fully traded by the end of the 10 minutes. This was a reasonable assumption in context, as our impact minimizing scheduler was designed to operate alongside but separately from an opportunistic liquidity seeker. So actions that have more uncertain outcomes are performed by the liquidity seeking component of the algo rather than the impact-minimizing scheduler component, mitigating the need to model greater uncertainty within our impact minimizing framework.

In a program trading context, however, we cannot let a liquidity seeking component act independently of the holistic constraints, so we need to unify our model of algo actions, including actions with less certain outcomes. It turns out that this is relatively easy to do! In the single-order context, we can instead model the future state 10 minutes out (say) as a random variable, and account for this level of uncertainty within our dynamic programming computation of our expected impact. This is an expansion of our impact-minimizing framework for single orders that is likely to be useful on its own - allowing us to model a larger space of potential algo actions under one impact model, potentially removing the need for structuring the algo as having two independent components. In the next section, we'll present a description of how this works for the single order case. After that, we'll demonstrate how we can expand all of this to program trading over baskets without sacrificing the ability to compute things efficiently.

2 A Starting Point - Single-Order Impact Minimization with Probabilistic Actions/Effects

As a warm-up, let's think about how we might try to find a schedule that minimizes expected trading cost for a given order, where we need to complete V units of volume over N time buckets. Defining expected trading cost requires us to form a model of how prices evolve through the period of our trading, and how our actions influence this evolution.

modeling price evolution We'll model prices as a sequence of random variables, where P_0 denotes the arrival price, and P_i denotes the price at the end of bucket i. We let Δ_i denote the random variable that represents the logarithm of the price change over the course of time bucket i. In other words, we have:

$$P_i = P_{i-1} \cdot e^{\Delta_i} = P_0 \cdot e^{\sum_{j \leq i} \Delta_j}.$$

For simplicity for now, we'll assume that any shares we trade within bucket i will be traded at price P_i. A more realistic assumption would be that our trades within bucket i achieve some average price that may be between P_{i-1} and P_i (or maybe not!), but let's ignore this nuance temporarily as there will be more than enough other sources of complication to wrap our heads around. Under these assumptions, the cost for an order that trades T_i shares within bucket i can be expressed as:

$$\sum_i T_i P_i = P_0 \cdot \sum_i T_i e^{\sum_{j \leq i} \Delta_j}.$$

Keeping track of relevant state From the perspective of an algo deciding what actions to take within a given time bucket, there are a few things we want to keep track of. These include i and N, the indices of the current and last time bucket respectively. Together, these tell us how many time buckets we have left to trade in. It is useful to keep these as separate variables instead of collapsing them into $N - i$ (the number of buckets left), since knowing specifically when we are in the trading day will effect distributions of market behavior. We will also keep track of V_i, which is the amount of volume we have remaining to trade, as of the beginning of time bucket i.

We will additionally define a more flexible state variable S_i to contain whatever else we know at the beginning of time bucket i that we feel is relevant to determining the distribution of Δ_i. In particular, this can contain information about our own trading in bucket $i - 1$, as accounting for this allows us to model reversion. This S_i can also contain features of intra-day trading so far that may be predictive of volume or volatility distributions in the current time bucket, as well as symbol characteristics like ADV and historical volatility.

Because it is conceptually flexible, there is a natural inclination to throw the kitchen sink into S_i and make it a vector of every feature we can think of that might be meaningful. As we will see later, this would be a mistake. There are two key pressures that will force us to keep S_i rather minimalistic. One is that high dimensional state vectors will explode the computational space, making our optimization problems much harder to solve or approximate in a reasonable amount of computation time. The second is that modeling distributions of Δ_i as complex functions of many variables will quickly lead to overfitting, making our models highly non-robust and non-generalizing. For these reasons, we will keep a tight level of discipline over our implementation of S_i, including only features that we have clear evidence are meaningful and non-redundant.

Putting this together, we can think of the inputs to our decision-making process at the beginning of bucket i as being the set of values (N, i, V_i, S_i).

Modeling our actions We can think of a potential action a that our algo may take as a function from the current values of (N, i, V_i, S_i) to a distribution over values $(N, i+1, V_{i+1}, S_{i+1})$ and Δ_i. This high level "action" does not specify trading logic at the level of what venues/order types to use, when to reprice child orders, when to post vs. cross the spread, etc. But it does allow us to model the aggregate effect of such choices as a distribution over relevant variables at the next checkpoint in time. Notably, the fact that we are modeling outcomes as distributions allows us to potentially capture various non-deterministic phenomena that may drive our actions or their effects. For example, we may randomize our own trading behavior within some limits in order to avoid detectable patterns, or we may consider the probability that the symbol price drifts in such a way that makes our order unmarketable. Thus, an "action" here can be more nuanced than a scheduling decision like "trade 1000 shares in the next time bucket." Non-determinism in our modeling of V_{i+1} in particular makes great nuance possible.

Expressing Our Cost Function We can now think of our high level scheduling logic for an algo as a function that takes in inputs of the form (N, i, v_i, s_i) and outputs an action. Here, v_i and s_i denote concrete values in the range of our random variables V_i and S_i. Our goal in choosing this function is to minimize our expected trading costs. For a given scheduling logic, we can express our expected trading costs starting from values (N, i, v_i, s_i) as:

$$C(N, i, v_i, s_i) := \mathbb{E}\left[\sum_{k=i}^{N} T_k e^{\sum_{i \le j \le k} \Delta_j}\right].$$

So our goal is to choose a high level scheduling logic that minimizes this, subject to ensuring completion of trading volume (i.e. $V_{N+1} = 0$). To help us do this, we start by noting a recursive property of this expression. We'll derive this in a few steps, starting with a consequence of the linearity of expectation:

$$\mathbb{E}\left[\sum_{k=i}^{N} T_k e^{\sum_{i \le j \le k} \Delta_j}\right] = \mathbb{E}\left[T_i e^{\Delta_i}\right] + \mathbb{E}\left[e^{\Delta_i} \sum_{k=i+1}^{N} T_k e^{\sum_{i+1 \le j \le k} \Delta_j}\right]$$

In the second expectation here, we will iterate over the possible values δ_i for Δ_i, the possible values s_{i+1} for S_{i+1}, and the possible values v_{i+1} for V_{i+1}:

$$\mathbb{E}\left[e^{\Delta_i} \sum_{k=i+1}^{N} T_k e^{\sum_{i+1 \le j \le k} \Delta_j}\right] = \sum_{\delta_i, s_{i+1}, v_{i+1}} \mathbb{P}[\delta_i, s_{i+1}, v_{i+1}] \cdot e^{\delta_i} \cdot \mathbb{E}\left[\sum_{k=i+1}^{N} T_k e^{\sum_{i+1 \le j \le k} \Delta_j} | s_{i+1}, v_{i+1}\right].$$

This holds because we are assuming that all dependence between Δ_i and further values Δ_j, T_k is captured by conditioning on s_{i+1} and v_{i+1}. From here we can see the recursive property:

$$\mathbb{E}\left[e^{\Delta_i} \sum_{k=i+1}^{N} T_k e^{\sum_{i+1 \le j \le k} \Delta_j}\right] = \sum_{\delta_i, s_{i+1}, v_{i+1}} \mathbb{P}[\delta_i, s_{i+1}, v_{i+1}] \cdot e^{\delta_i} \cdot C(N, i+1, v_{i+1}, s_{i+1}).$$

By the definition of conditional probability, we can rewrite this as:

$$= \sum_{s_{i+1}, v_{i+1}} \mathbb{P}[s_{i+1}, v_{i+1}] \left(\sum_{\delta_i} \mathbb{P}[\delta_i | s_{i+1}, v_{i+1}] \cdot e^{\delta_i}\right) \cdot C(N, i+1, v_{i+1}, s_{i+1})$$

$$= \sum_{s_{i+1}, v_{i+1}} \mathbb{P}[s_{i+1}, v_{i+1}] \mathbb{E}\left[e^{\Delta_i} | s_{i+1}, v_{i+1}\right] \cdot C(N, i+1, v_{i+1}, s_{i+1}).$$

Putting this all back together, we have:

$$C(N, i, v_i, s_i) = \mathbb{E}\left[T_i e^{\Delta_i}\right] + \sum_{s_{i+1}, v_{i+1}} \mathbb{P}[s_{i+1}, v_{i+1}] \mathbb{E}\left[e^{\Delta_i} | s_{i+1}, v_{i+1}\right] \cdot C(N, i+1, v_{i+1}, s_{i+1}).$$

This holds also if we replace the cost function C for a particular scheduling logic with the cost function $\tilde{C}$ for the minimizing scheduling logic, where the minimization is over the set $A(i, N, v_i, s_i)$ of actions a that we can take in bucket i, given the current values of i,N,v_i,s_i:

$$\tilde{C}(N, i, v_i, s_i) = \min_{a \in A(i, N, v_i, s_i)} \left(\mathbb{E}[T_i e^{\Delta_i} | a] + \sum_{s_{i+1}, v_{i+1}} \mathbb{P}[s_{i+1}, v_{i+1} | a] \mathbb{E}\left[e^{\Delta_i} | s_{i+1}, v_{i+1}, a\right] \cdot \tilde{C}(N, i+1, v_{i+1}, s_{i+1})\right).$$

If we are willing to keep our state space and our action set small enough, this formulation allows us to efficiently compute $\tilde{C}$ and to solve for the minimizing action a using dynamic programming, as we can compute $\tilde{C}(N, i+1, v_{i+1}, s_{i+1})$ for all possible values v_{i+1} and s_{i+1} before computing $\tilde{C}(N, i, v_i, s_i)$. We can also incorporate hard constraints on our trading behavior by, say, setting $\tilde{C}$ to be infinite when the input violates our constraints, and only minimizing over an allowed set of actions. This formulation can also seamlessly incorporate improvements in our modeling, by adding new relevant features to state variable S_i, for example.

So far, this matches pretty closely what we did to design the impact minimizing scheduler for our Proof algo (see our whitepaper at https://prooftrading.com/docs/main-algo.pdf), but with one subtle difference. In the prior whitepaper, we assumed that V_{i+1} was a deterministic outcome conditioned on our choice of action. This assumption simplified the formulation a bit, but we've since realized that it is not needed. The dynamic programming computation of $\tilde{C}$ here can seamlessly handle V_{i+1} as a random variable, as long as we have a model for its joint distribution with S_{i+1} as a function of our actions.

3 Program Trading Optimization as an Expansion of this Framework

3.1 Naive and inefficient approaches

So what happens if we try to apply this cost minimization framework to a basket of orders that are constrained holistically? The first thing we might consider is just independently minimizing the expected cost of each order. However, this will likely lead to a set of schedules that violate basket-level constraints. It also ignores any possibility of modeling cross-symbol effects, like how our trading in one symbol may be expected to have some impact on another, if there is some general correlation between the prices of the two symbols.

We could simply impose basket-level constraints on top of individually impact-minimizing schedules, but this is a bit unsatisfying, as it breaks any meaningful guarantees of the minimizations. Given that we would probably have to intervene substantially to enforce basket-level constraints, the individual schedules would drift considerably away from our original minimizations, leaving us with no real basis to think they are still close to optimal, or even particularly good.

A tempting alternative is to simply expand our random variables of remaining volume V_i and current state S_i into vectors of dimension d, where d is the number of orders in the basket. Each coordinate would keep track of these values for a single order, and our probability distributions would be modeled as joint distributions over the d-dimensional space. Similarly, our actions would be modeled as function from d-dimensional inputs to distributions over d-dimensional inputs. This would be an extremely expressive modeling approach, allowing us to enforce basket constraints by declaring which state and volume vectors are permissible and allowing us to potentially model cross symbol effects. But... the dynamic programming approach to compute $\tilde{C}$ that we outlined above would explode to require exponential memory and computation time in terms of the dimension d. This would only scale to very *very* small baskets as a result, like pairs trading or "baskets" of say at most 5 orders.

3.2 An Efficient Approach

If we want a holistic basket optimization that can be efficiently computed, we do have other choices. One approach is to start with a fixed set of deterministic schedules for the basket orders that would satisfying all desired constraints, and then introduce slack variables to represent allowable levels of deviation from these schedules. We can then allocate total slack smartly across orders, allowing deviation strategically for orders whose expected costs will benefit the most from the additional flexibility.

To explore this approach, let's first develop some helpful notation for keeping track of a basket of orders. We'll let $\mathcal{B}$ denote our basket of orders, and an individual order will be denoted by $b \in \mathcal{B}$. Our variables V_i^b, S_i^b will now be superscripted by b to indicate that they represent the state of order b at the beginning of time bucket i.

A fixed, deterministic schedule for order b from time period i onward can be represented by a sequence of proscribed values $(v_i^b, \ldots, v_N^b)$ for $V_i^b, \ldots, V_N^b$, or equivalently by a sequence of proscribed values $(t_i^b, \ldots, t_N^b)$ for $T_i^b, \ldots, T_N^b$, where $t_j^b = v_j^b - v_{j+1}^b$.

Given a set of fixed schedules $\{v_j^b\}_{b\in\mathcal{B}}$, we can ask: if these schedules were perfectly adhered to and prices stayed constant, do they satisfy all of the constraints on the basket? In full generality, we can think of a constraint like "approximate cash neutrality" as a function from inputs like $\{v_j^b\}_{b\in\mathcal{B}}$ and current prices $\{P_b\}_{b\in\mathcal{B}}$ to a binary output that indicates whether the constraint is satisfied. Such a function could be expressed as follows. We let $\mathcal{B}_{buy}$ represent the subset of buy orders in our basket, and $\mathcal{B}_{sell}$ represent the subset of sell orders. At the beginning of each time bucket j, the cash value of our executed buy orders since the beginning of time bucket i can be written as:

$$\sum_{b\in\mathcal{B}_{buy}} P_b(v_i^b - v_j^b).$$

Similarly, the cash value of our executed sell orders can be written as:

$$\sum_{b\in\mathcal{B}_{sell}} P_b(v_i^b - v_j^b).$$

Thus, an approximate constraint on cash neutrality can be expressed as an inequality constraint on difference between these two values that most hold for every j. However - there is an important detail here. Looking at the difference between these values for each value of j will only ensure that the constraint

holds exactly at the beginning of each time bucket, but not necessarily in between. If we don't want to divide time into finer buckets and micro-manage how the algo will trade each t_i^b units in bucket i for each order b, then we may need to cover even worse case scenarios, like all of the buy volume happening in the beginning of bucket while all of the sell volume happens near the end. In this case, we would consider indices j and $j+1$. We could require, for example, that the cash value of the sell orders at the beginning of bucket j and the cash value of the buy orders at the end of bucket $j+1$ are also within our desired tolerance around cash neutrality.

Finding an initialization point The first step in our basket scheduling framework will be to find a set of fixed schedules $\{\overline{v}_j^b\} \in \mathcal{B}$ that would satisfy all of our constraints. Of course, if our constraints are contradictory somehow, this won't be possible. Similarly, if our constraints are arbitrarily complex functions, finding a single set of satisfying fixed schedules may not be computationally feasible. But these kind of problems are not likely to be a concern in practice, where the space of communicable constraints will be tightly specified and impossible parameter settings can be detected and rejected.

We will not spend time here on discussing *how* to find an initial set of constraint-satisfying fixed schedules, as good methods for doing this will likely depend on the kind of constraints supported and client preferences for default ways to trade. For example, one could start by estimating volume curves and define VWAP-like schedules as a default, then make some modifications if needed to stay within constraints. Suffice it to say - there are probably several answers to how one can efficiently and reasonably find an initial set of schedules for a basket that would satisfy all of the constraints. For notational convenience, we will refer to our initial fixed schedule for order b by $\overline{v}^b := \{\overline{v}_j^b\}$.

We might wonder - why not find the *optimal* set of fixed schedules that satisfy our constraints? In other words, if we have a way of estimating the expected cost C of a fixed schedule for each order b, why not spend the time to determine the choice of fixed schedules that minimizes the total expected cost over the basket among all choices of fixed schedules that satisfy the basket constraints? Well, we could. And this is one reasonable way to produce an initial set of fixed schedules if the constraints and cost formulation in combination allow an efficient solution to this. But - even an "optimal" set of fixed schedules may ultimately perform much worse than schedules that are dynamic and probabilistic. For example, a fixed schedule will never allow one to opportunistically take block liquidity when it is available but not currently scheduled for that time bucket. For reasons like this, we choose to focus on introducing flexibility into our schedules as a means of reducing expected cost, rather than primarily optimizing among fixed schedules.

Adding slack variables For each order b in our basket, let's introduce a variable γ_b to represent how far order b may deviate from its initial fixed schedule. This means that at the beginning of each time bucket j, the value for V_j^b needs to be within $\pm\gamma_b$ of the fixed value $\overline{v}_j^b$. We note that this tolerance still applies for the beginning of the last time bucket $j = N$, but V_{N+1} is still required to be precisely 0 (trading must complete). Our overall slack variables $\{\gamma_b\}_{b\in\mathcal{B}}$ will ultimately have to be chosen so that the basket constraints remain satisfied as long as all individual orders stay within $\pm\gamma_b$ of their initial fixed schedules.

Having this kind of tolerance γ_b around the target volume for each time bucket can given an algo space to perform more speculative actions, like resting a dark order opportunistically that may not be filled, or choosing to get ahead of schedule if there seems to be above-average liquidity. Consequently, we would expect that the expected cost of order completion goes down as flexibility goes up (to a point).

Let's see how we might incorporate the variable γ_b into our expected cost function for trading order b. In time bucket i, the set of allowable algo actions will now be a function of i,N,s_i^b,v_i^b,γ_b, and $\overline{v}_{i+1}^b$. This is because we want to ensure that we only take actions that result in distributions over V_{i+1}^b that have support contained in $\overline{v}_{i+1}^b \pm \gamma_b$, so that we will stay within our tolerance around the fixed schedule.

We can now define a version of our minimum expected cost $\tilde{C}$ that minimizes only over actions that preserve the property of staying within tolerance, and it will still have a recursive property:

$$\begin{aligned}\tilde{C}(N,i,v_i^b,s_i^b,\gamma_b,\overline{v}^b) \quad = \quad & \min_{a\in A(N,i,v_i^b,s_i^b,\gamma_b,\overline{v}_{i+1}^b)} (\mathbb{E}[T_i^b e^{\Delta_i^b}|a] + \\ & \sum_{s_{i+1}^b, v_{i+1}^b} \mathbb{P}[s_{i+1}^b, v_{i+1}^b|a] \cdot \mathbb{E}[e^{\Delta_i^b}|s_{i+1}^b, v_{i+1}^b, a] \cdot \tilde{C}(N,i+1,v_{i+1}^b,s_{i+1}^b,\gamma_b,\overline{v}^b)).\end{aligned}$$

Now, for a fixed value of γ_b, we can compute this quantity via dynamic programming as before. Crucially, whether an action satisfies the condition of keeping within the tolerance γ_b of the fixed scheduled

$\overline{v}_b$ is something we can efficiently check as we combine our pre-computations of $\tilde{C}(N, i+1, v_{i+1}^b, s_{i+1}^b, \gamma_b, \overline{v}^b$ to compute $\tilde{C}(N, i, v_i^b, s_i^b, \gamma_b, \overline{v}^b)$.

The question now becomes: what constraints do our values of $\{\gamma_b\}_{b \in \mathcal{B}}$ need to satisfy, how can we choose them wisely to try to minimize our total cost? More formally, we want to minimize

$$\sum_{b \in \mathcal{B}} \tilde{C}(N, i, v_i^b, s_i^b, \gamma_b, \overline{v}^b),$$

subject to some constraints.

The form of the constraints on γ_b values is likely to be simple. We need constraints like $\gamma_b \geq 0$, and also $\sum_{b \in \mathcal{B}} \gamma_b \leq \Gamma$ for some constant Γ that represents how far away the fixed schedules are from violating some basket constraint. Let's assume for now that these are the only constraints on γ_b values: they are non-negative values whose total sum is bounded.

How to best distribute the total slack Γ across the orders in the basket depends on how the various cost functions $\tilde{C}$ behave as functions of γ_b. If $\tilde{C}$ functions behave arbitrarily, finding an optimal or even good setting of the γ_b values may be a very difficult problem. But if $\tilde{C}$ behaves in more particular ways, we can often find a efficient solution to choosing the best γ_b values within the constraints.

Let's start with perhaps the easiest example: suppose that each $\tilde{C}(N, i, v_i^b, s_i^b, \gamma_b, \overline{v}^b)$ behaves as a linear function of γ_b when all of the other inputs are fixed. In other words, the contribution to cost from order b will be of the form $c_b + \gamma_b c_b'$ for some constants c_b and c_b'. Presumably c_b' is negative, as having more slack should reduce expected cost.

In this case, the quantity we are trying to minimize is:

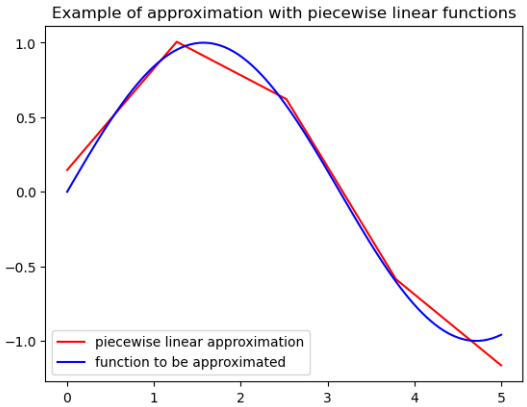
$$\sum_{b \in \mathcal{B}} c_b + \gamma_b c_b',$$

and this will be minimized by allocating the full budget of total slack to whatever order has the most negative coefficient, i.e. $\gamma_{b^*} = \Gamma$ for the b^* where $|c_b'|$ is maximized, and $\gamma_b = 0$ for all other orders.

While simple and appealing, this is clearly unrealistic nonsense. The idea that you can keep adding slack for a single order and costs will keep decreasing proportionally must break down at some point, as infinite slack cannot correspond to infinitely negative cost! But... it could make sense to treat costs as linear in a very limited range of values for γ_b. We could essentially add new constraints of the form $\gamma_b \leq \ell$ for some small constant ℓ, and then approximate $\tilde{C}$ by a linear function of γ_b values within this small range. The optimal solution within these constraints would then be to max out $\gamma_b = \ell$ for the order b with the most negative slope, then max out the second most negative slope, etc., until running out of the total slack budget Γ. We should point out here that this is a special case of a linear program, but the more general methods of linear program solving are not needed here because our constraint set is sufficiently simple that we can just derive the optimal solution immediately.

This approach may still not be satisfying though, as the bound of ℓ that makes linear cost approximation reasonable may be lower than what we might want to consider as a ceiling for each γ_b. Thus, we could be missing potential gains from allocating even higher slack values to some orders. So let's consider what kind of non-linear functions $\tilde{C}$ might still be manageable.

We could imagine instead that each $\tilde{C}$ is piecewise linear, for example. This alone is not assuming much, since piecewise linear functions can be used to approximate any continuous function (where the approximation can be made arbitrarily tight by using more and more linear pieces):

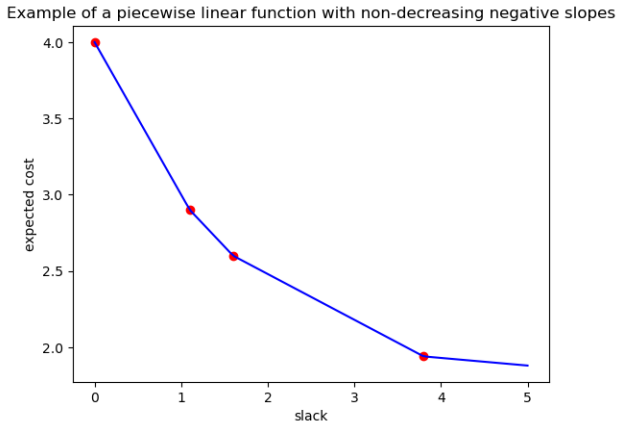


For the case of $\tilde{C}$ in particular, we might further suppose that the slopes of the linear pieces are non-decreasing, corresponding to a regime of diminishing returns. (Note that the slopes will be negative, so non-decreasing here means that their absolute values are non-increasing. In other words, the slope are getting less steep as they go on.) This is certainly true at some point, as there is some optimal set of choices for an order b that requires some particular amount of slack γ_b from the default schedule, and beyond that further deviation is not helpful in reducing the value of $\tilde{C}$.

If we can reasonably approximate each $\tilde{C}$ with a piecewise linear function with non-decreasing negative slopes, then we can again solve efficiently for $\{\gamma_b\}_{b\in\mathcal{B}}$ values that minimize the total expected costs over the basket $\mathcal{B}$. To see how, let's express the piecewise linear approximation of the expected cost function $\tilde{C}$ for order b as an initial constant term c_b and a sequence of slopes $c'_{b,j}$, where j indexes the linear pieces and the slopes satisfy $c'_{b,j} \leq 0$ and $c'_{b,j} \leq c'_{b,j+1}$ for all j. In other words, our pieces break up the domain of γ_b into ranges $[r_{b,j}, r_{b,j+1})$, and for $\gamma_b \in [r_{b,j}, r_{b,j+1}]$, we have:

$$\tilde{C}(\gamma_b) \approx \left(c_b + \sum_{j'<j} c'_{b,j'} r_{j'} \right) + c'_{b,j} \gamma_b.$$

Here is an illustration of what a piecewise linear function with non-decreasing slopes looks like, with the red dots marking the endpoints of the linear pieces:



Now let's think about how we can allocate our total Γ of slack to minimize the sum of all the $\tilde{C}$ values in such a case. The intuitive idea is that we will allocate slack in a greedy fashion, always looking to add slack where the slope is most steeply negative in order to reduce total cost as much as possible. We might generally worry that behaving greedily could cause us to miss out on potential gains deeper into the γ_b ranges, but this is where the non-decreasing assumption on slopes kicks in to ensure optimality.

More formally, we will order all of the initial slopes $\{c'_{b,0}\}_{b\in\mathcal{B}}$ from smallest (most negative) to largest (least negative). We also initialize all of our γ_b variables to 0.

We will then start allocating slack to max out the domain of the most negative slope. Let's say that order b^* has the most negative initial slope, and this slope persists from $0 = r_{b^*,0}$ to $r_{b^*,1}$. If $r_{b^*,1}$ is $\geq \Gamma$, then we set $\gamma_{b^*} = \Gamma$, leaving all other slack variables at 0. In this case, we are done.

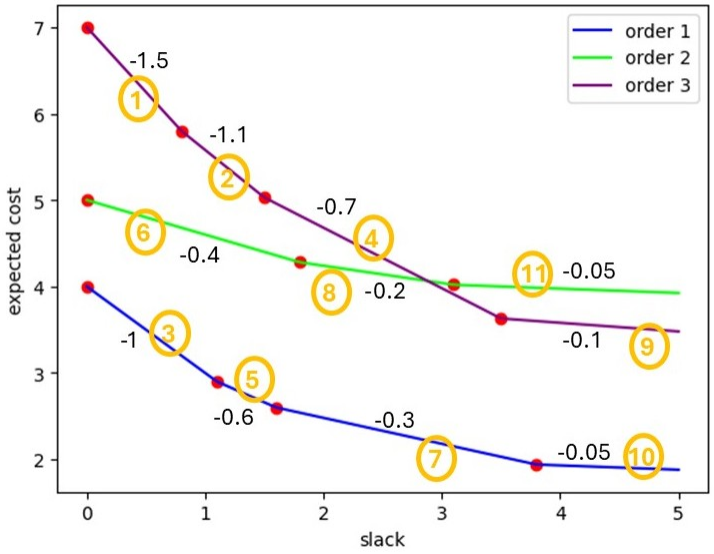
Otherwise, we have $r_{b^*,1} < \Gamma$. In this case, we update $\gamma_{b^*} = \gamma_{b^*} + r_{b^*,1}$ and proceed as follows. We update our remaining slack budget to $\Gamma - r_{b^*,1}$, we remove $c'_{b^*,0}$ from our collection of slopes under current consideration, and we add $c'_{b^*,1}$ instead. We then sort again the set of $|\mathcal{B}|$ active slopes from most negative to least negative.

Now we can repeat: until we run out of total slack to allocate, we'll keep maxing out the domain of the most negative slope, then replacing it in the set of active slopes with its successor.

To see why this yields an optimal reduction in cost for our fixed slack budget Γ, we can compare this to a clear lower bound on the total cost that any allocation of slack Γ can achieve. Imagine that we simply order *all* of the slopes $\{c'_{b,j}\}$ over all b and all j from most negative to least negative. If there are ties, we break them based on taking lower values of j first. If the j's are the same but the b's are different, we can break ties arbitrarily. A lower bound on the total cost can then be derived by imagining that we reduce the cost by the most negative $c'_{b,j}$ times its domain size $r_{b,j+1} - r_{b,j}$ followed by the next most negative slope times its domain size, etc., until the total domain sizes used add up to Γ. This is the most bang for our buck, so to speak, that we could hope to get in terms of overall cost reduction. We might worry, though, that this lower bound is not actually achievable, since the linear pieces of each

$\tilde{C}$ function for each order b have to be taken in order: we cannot get the benefit of a slope in piece j unless we also use our slack budget to get through all of the pieces before j for the same order b. This is where our assumption of slopes that never get more negative comes in. Since $c'_{b,j} \leq c'_{b,j+1}$ for all j, all the slopes of all the previous pieces would have been chosen *before* this $c'_{b,j}$ in our ordering. This means that the lower bound for total cost is achievable, and the iterative process we have outlined above does achieve it.

We can visualize this process as follows:



Here we have illustrated a toy example with three orders, where each expected cost is represented as a piecewise linear function with negative slopes whose magnitudes are non-increasing. Here we have labeled the slope for each piece in black, and used orange circled numbers to indicate the order in which these pieces would be chosen to contribute to our total slack budget, until we run out. Notice that the ordering satisfies two properties: 1) it matches the global ordering from most negative slope to least negative slope, where ties can be broken arbitrarily, and 2) each piece is only chosen *after* its preceding pieces within the same order. These two properties ensure that our greedy selection procedure is both optimal and valid.

Putting it all together Overall, we now have a framework that can produce high-level scheduling decisions for trading an incoming basket by following these steps:

1. Parse the set of individual orders in the basket as well as the basket-level constraints.
2. Determine a initial set of schedules that would satisfy the constraints, as well as a total amount of slack Γ that can be tolerated around this initialization.
3. For each basket order b, approximate $\tilde{C}$ as a function of γ_b with a piecewise linear function, where all pieces have non-positive slopes, and these slopes are non-decreasing (which means non-increasing in absolute value).
4. Compute slack values γ_b for the individual orders in the basket that minimized the total expected cost, subject to staying within the total slack budget Γ.
5. For each order b, use the specified value of γ_b and dynamic programming to determine the next action that is expected to produce the minimum expected cost while staying within slack γ_b of the fixed initialized schedule.

Now, as a time bucket completes, we'll have new information for an order b about the new state variable and the update volume left to complete. This means that we'll want to decide the next action by a fresh computation of $\tilde{C}$ for this order from this new state (still with the same γ_b parameter) in order to determine the next minimizing action. One thing to note here is that there is no inherent need to synchronize our definition of time buckets across the basket orders, or to have all of our time buckets

represent the same fixed length of time. Bucket lengths could be randomized individually for each order b, and the framework works in this form since each $\tilde{C}$ calculation is performed on an individual order with respect to its specified time buckets.

4 Extensions of this framework (some immediate and some speculative)

There is obviously a lot that this framework leaves pending. How should we define the state variables? (In other words, what features are worth keeping track of and what features are not?) How should we choose our initialization schedules? How should we handle external and unpredictable constraints like individual order limit prices, POV bands, etc.? These considerations will be the subject of future blog posts/whitepapers that we write as we continue to flesh out a fuller picture of our day one PT algo. But for now, it's worth noting some things about this current framework that it would be nice to expand upon.

More flexibility/dynamism in the slack allocations First, it would be nice if we could have more flexibility in our γ_b settings than sticking with a constant slack value for each order throughout the lifetime of the basket. One option we have is to periodically re-evaluate our basket scheduling by treating what we have left as a new basket, with updated constraints that reflect the slack between where we currently stand and what our overall constraints are. The 5 steps above can then be repeated fresh to produce new γ_b values and new resulting scheduling decisions. One limitation of this approach is it does require a synchronous re-evaluation - meaning that all orders will be simultaneously affected and we should be careful not to cause a suspicious correlation of market activity across the basket as a result.

We can try a more localized dynamic approach, periodically re-visiting our estimation of the value of slack as it pertains to particular orders. By re-approximating the cost function $\tilde{C}$ for an order b as a function of γ_b based on updated state and remaining volume, we might find that the cost reduction slope achieved for our allocated slack going forward is meaningfully more or meaningfully less than we anticipated for some orders. If we can find some orders that would really benefit from increased slack and some orders that would not be as harmed by having less slack, we can move some slack to where it is more needed. This can be done for potentially small subset of orders a time, not requiring coordination or causing correlation across large swathes of the basket at once.

Another approach would be to try to treat each γ_b as a function of time rather than a constant, and solve for an optimal set of function allocations of slack from a richer class than mere constants. Some versions of this are immediately doable with the same techniques presented above. In particular, we could set a total budget $\Gamma(t)$ as a function of time, and allocation fractions of it as $\gamma_b := \alpha_b \Gamma(t)$ for non-negative constants α_b summing to ≤ 1. As above, we could reasonably model $\tilde{C}$ as a non-increasing function of α_b and choose α_b's greedily to get an optimal allocation under these constraints. Perhaps more elaborate families of γ functions could also be supported with slight modifications of these techniques.

It should also be noted that we may need to accommodate non-zero minimums for the slack variables γ_b to bake in known variances for even our most precisely regimented trading tactics. This is easily managed by initializing each γ_b to its minimal value and then allocating the remaining slack budget optimally as detailed above.

Relaxing assumptions on the form of $\tilde{C}$ Here we dealt with the rather easy but unrealistic case of each $\tilde{C}$ being linear throughout a desired range, as well as the more realistic but still somewhat limiting case of piecewise linear functions whose negative slopes are not getting steeper as the slack parameter increases. It would be nice to discover further possible structures for $\tilde{C}$ that would yield efficiently computable solutions. Could arbitrary piecewise linear functions for $\tilde{C}$ be accommodated, for example? We think this may be possible, but we did not spend much time investigating it yet.

Accommodating expansions of the state space If we begin to discover even a rather modest number of meaningful variables that we would like to include in our state vectors to increase the accuracy of our cost projections, than the dynamic programming over all possible state values will quickly get out of hand. Accommodating this will require an evolution of approach to computing $\tilde{C}$. One possible start to this could be attempting to model the effect of state variables in a closed form, smooth fashion, rather than breaking out our formula for $\tilde{C}$ as a discrete sum over discrete state value possibilities.

Searching over differing initializations One thing we can easily do is to consider different initial schedules. Since our slack variables γ_b represent deviation from a fixed set of initial schedules, the ultimate minimizing value for the sum of $\tilde{C}$ functions over the basket that we obtain is a function of our initialization, and can potentially be improved by choosing a different initialization. If we have a way of producing multiple initialization possibilities, we can run our computation for many such possibilities and choose the one that results in the overall minimum expected cost.

Cross-symbol features As described above, our framework currently treats the total cost as a sum of individual cost functions $\tilde{C}$ for each order, unifying these only at the layer of a total slack budget and not modeling other forms of explicitly shared state between symbols. If we think that how we are trading one symbol may be substantively affecting market behavior in another symbol, then we might want instead to include some state variables that are explicitly shared across two or more symbols in our total computation of expected cost. Currently, we aren't yet sure of a good way to do this in full generality while maintaining efficient computability. The ability to perform our $\tilde{C}$ computations across orders in a parallel way is a nice feature that we would potentially lose as we try to extend our framework in this direction, and doing dynamic programming across the full joint state space of all of the basket's orders at once seems infeasible. However, grouping orders into small, bounded units - say pairs or triples that are most correlated to each other - would be a way to optimize jointly within each unit efficiently while still parallelizing over the units to avoid an exponential computation time.