1 Numerical implementation details

This section covers detail regarding the solution algorithm and numerical implementation. Each method is presented in a separate subsection, and the final solution algorithm is presented in the last subsection, which combines each of the methods. These span points sampling, numerical integration techniques, function approximation methods and solution techniques specific to this class of problemns.

1.1 Numerical integration

Consider the basic problem with proportional transaction costs, basic risky assets and a risk-free asset and no stochastic parameters. We need to evaluate the expectation of the value function: $\mathbb{E}\left[v_{t+\Delta t}(\mathbf{x}_{t+\Delta t})\right]$. In order to compue this expectation, we need to evaluate the integral:

$$\mathbb{E}_t\left[\pi_{t+1}^{1-\gamma}v_{t+1}(x_{t+1})\right] = \int \pi_{t+1}^{1-\gamma}v_{t+1}(x_{t+1})f(R_{t+1}), dR_{t+1} \tag{1}$$

where $f(R_{t+1})$ is the probability density function of the risky asset returns. If we look at the case of stochastic parameters, would need to evaluate the conditional expectation with regard to these aswell, given some distributional assumption on the parameters. The integral can be computed using Monte-carlo methods or by using quadrature rules.

1.1.1 Gauss-Hermite quadrature

Gaussian quadrature is a numerical integration method based on approximation and interpolation theory. Gaussian quadrature can be used to approximate integrals using the following form, Judd (1998):

$$\int_a^b f(x)w(x)dx \approx \sum_{i=1}^{n} \omega_i f(x_i), \tag{2}$$

Where ω_i are quadrature weights, x_i are quadrature nodes and $w(x)$ is a weighting function. This approximation is exact when $f(x)$ is a polynomial of degree $2n-1$ or less. Then we can approximate the integral using n points x_i and n weights ω_i. There are many different Gaussian quadrature schemes, with differering intervals $[a,b]$ and weighting functions $w(x)$. We consider the use of a Gauss-Hermite quadrature rule, for a comprehensive review on Gaussian quadrature rules, see Judd (1998). Gauss-Hermite quadrature is used to approximate integrals of the form:

$$\int_{-\infty}^{\infty} f(x)e^{-x^2}dx \approx \sum_{i=1}^{n} \omega_i f(x_i) + \frac{n!\sqrt{\pi}}{2^n} \cdot \frac{f^{(2n)}(\zeta)}{(2n)!}, \tag{3}$$

Where $\zeta \in (-\infty, \infty)$. If a random variable X is normally distributed, i.e $X \sim \mathcal{N}(\mu, \sigma^2)$, then we can compute the expectation, $\mathbb{E}[f(X)]$, which is given by:

$$\mathbb{E}[f(X)] = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{\infty} f(x) e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \tag{4}$$

Using a change of variables $y = \frac{x-\mu}{\sqrt{2}\sigma}$, then we can rewrite the expectation on the form of the Gauss-Hermite quadrature rule:

$$\begin{aligned} \mathbb{E}[f(X)] &= \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{\infty} f(\sqrt{2}\sigma y + \mu) e^{-y^2} \sqrt{2}\sigma dy \qquad (5)\\ &= \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} e^{-y^2} f(\sqrt{2}\sigma y + \mu) dy \qquad (6)\\ &\approx \frac{1}{\sqrt{\pi}} \sum_{i=1}^{n} \omega_i f(\sqrt{2}\sigma x_i + \mu) \qquad (7) \end{aligned}$$

Where ω_i are the quadrature weights, x_i are the quadrature nodes over the interval $(-\infty, \infty)$.
When X is log-normal, i.e $\log X \sim \mathcal{N}(\mu, \sigma^2)$, then we can use a variable change once again: $X = e^Y$ and $Y \sim \mathcal{N}(\mu, \sigma^2)$. Then we can rewrite the expectation as:

$$\mathbb{E}[f(X)] = \mathbb{E}[f(e^Y)] \approx \pi^{-\frac{1}{2}} \sum n_{i=1} \omega_i f\left(e^{\sqrt{2}\sigma x_i + \mu}\right) \tag{8}$$

If we want to extend this framework to multiple dimensions we can use product rules as noted by Cai, Judd and Xu (2013). Consider Y which is multivariate normal, i.e $Y \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, where μ is the drift vector and Σ is the covariance matrix. Let L be a lower-triangular matrix such that $LL^\top = \Sigma$ (Cholesky factorisation). Then we have that:

$$\begin{aligned} \mathbb{E}\{f(Y)\} &= \left((2\pi)^d \det(\Sigma)\right)^{-\frac{1}{2}} \int_{\mathbb{R}^d} f(y)\, e^{-\frac{1}{2}(y-\mu)^\top \Sigma^{-1}(y-\mu)}\, dy \qquad (9)\\ &= \left((2\pi)^d \det(L)^2\right)^{-\frac{1}{2}} \int_{\mathbb{R}^d} f\left(\sqrt{2}Ly + \mu\right)\, e^{-\frac{1}{2}y^\top y}\, dy \qquad (10)\\ &\approx \pi^{-\frac{d}{2}} \sum_{i_1=1}^{n} \cdots \sum_{i_d=1}^{n} \omega_{i_1} \cdots \omega_{i_d}\, f\Bigg(\sqrt{2}L_{1,1}y_{i_1} + \mu_1,\\ &\qquad \sqrt{2}(L_{2,1}y_{i_1} + L_{2,2}y_{i_2}) + \mu_2,\ \ldots,\ \sqrt{2}\big(\sum_{j=1}^{d} L_{d,j}y_{i_j}\big) + \mu_d\Bigg) \qquad (11) \end{aligned}$$

Where d refers to the number of dimensions, n is the number of quadrature points, ω_i are the quadrature weights and y_i are the quadrature nodes. $L_{i,j}$ is the ith row and jth column of the Cholesky factorisation matrix L. det is the matrix determinant. We note that the use of product rules suffers from the curse of dimensionality, as the complexity scales exponentionally with the number of dimensions. This is because the quadrature points with the product rule, normally use a tensor product grid, which is constructed using the Cartesian product of the quadrature points in each dimension. We can use sparse grid methods to partially tackle this. One common method is the Smolyak method, Smolyak (1963). Smolyaks sparse grid method approximates multidimensional integrals, over dimesion d while limiting the amount of points used. The method is composed of the following:

1. **Univariate Quadrature Rules**: Each dimension of the integration domain is assigned a univariate quadrature rule, which provides both nodes (quadrature points) and weights for numerical integration in that dimension. The accuracy of each rule is determined by its *level*, denoted by i_d for each dimension d. The level determines the number of quadrature points in that dimension, which improves the accuracy of the quadrature rule.

2. **Approximation Level (μ)**: The accuracy of the Smolyak sparse grid is controlled by the *approximation level* μ. This parameter sets a limit on the sum of levels across all dimensions, controlling the total number of grid points. Higher values of μ result in more accurate approximations but increase computational complexity.

3. **Multi-Index and Combination of Levels**: In a d-dimensional integral, the Smolyak method uses a *multi-index* $i = (i_1, i_2, \ldots, i_d)$ to represent the level of the quadrature rule in each dimension. The multi-index specifies a unique combination of quadrature levels for each dimension, where i_d denotes the level for dimension d. To construct a sparse grid, Smolyak's method restricts the sum of these levels using the following condition:

$$d \leq i_1 + i_2 + \cdots + i_d \leq d + \mu$$

   This constraint on the sum of levels, reduces the number of tensor products. We denote the sum of multi indicies: $|i| = i_1 + i_2 + \ldots i_d$.

4. **Tensor Product of Univariate Rules**: The Smolyak grid is formed by taking the *tensor product* of univariate quadrature rules that satisfy the multi-index constraint. Each univariate quadrature rule, represented by Q_{i_d} at level i_d in dimension d, is combined across dimensions according to the set of multi-indices i. This combination is given by:

$$A(\mu, d) = \sum_{d \leq |i| \leq d+\mu} (-1)^{\mu+d-|i|} \binom{d-1}{\mu + d - 1 - |i|} \bigotimes_{d=1}^{d} Q_{i_d}$$

   where:

   - Q_{i_d} is the univariate quadrature rule at level i_d in dimension d,

- $\bigotimes$ denotes the tensor product, and
- $\binom{d-1}{\mu+d-1-|i|}$ is a combinatorial coefficient that assigns weights to each tensor product, for accurate integration up to the specified approximation level μ.

By resticting the multi indicies i with the approximation level μ, the Smolyak method reduces the number of points needed for numerical integration in higher dimensions. Tensor grid methods grows exponentially with the number of dimensions d, the Smolyak grid grows polynomially, Judd et al. (2014), hence it directly combats the curse of dimensionality. For more on this see Smolyak (1963), Judd et al. (2014) and Horneff, Maurer and Schober (2016).

1.1.2 Monte Carlo integration (MC)

Monte Carlo integration is a numerical integration method based on *sampling*, as opposed to quadrature rules which are based on interpolation.
The convergence of Monte Carlo integration is generally slower than some quadrature methods; however, its convergence rate is independent of the dimensionality of the integral, making it well-suited for high-dimensional problems. Monte Carlo integration breaks the curse of dimensionality. Monte Carlo (MC) integration is based on random sampling[1] over the domain of the integral, and then computing the sample average of the function to be integrated. Assume we wish to approximate the d-dimensional integral:

$$I = \int_\Omega f(\mathbf{x})g(\mathbf{x})d\mathbf{x} = \mathbb{E}[f(\mathbf{x})], \tag{12}$$

where $g(\mathbf{x})$ is the probability density function of the random variable $\mathbf{x}$ over its support Ω, we approximate I as:

$$Q_N = \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{X}_i), \tag{13}$$

where $\mathbf{X}_i$ are independent samples drawn from $g(\mathbf{x})$. The procedure is then:

1. Sample N points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ from $g(\mathbf{x})$.
2. Approximate the expectation $\mathbb{E}[f(\mathbf{x})]$ by the sample average:

$$I \approx Q_N = \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i).$$

The Law of Large Numbers ensures that the sample average converges to the mean as $N \to \infty$:

$$\lim_{N\to\infty} Q_N = \mathbb{E}[f(\mathbf{x})] = I.$$

[1]Strictly speaking the samples are not random, but pseudo-random, meaning that deterministic samples are used, which appear random. For more in this see Judd (1998) or Glasserman (2004)

And by the Central Limit Theorem, we have:

$$\sqrt{N}\left(Q_N - I\right) \xrightarrow{d} N\left(0, \sigma^2\right),$$

where $\sigma^2 = \mathrm{Var}[f(\mathbf{x})]$ does not depend on N or d. The standard error of Q_N is:

$$\sigma_{Q_N} = \frac{\sigma}{\sqrt{N}}.$$

The convergence rate of $1/\sqrt{N}$ is independent of the dimension.

1.1.3 Quasi-Monte Carlo integration (QMC)

Quasi-Monte Carlo integration substitutes the 'random' samples in Monte Carlo integration with specific deterministic sequences such as equidistributred sequences, low-discrepancy sequences (LDS) or Lattice point rules etc. We will focus on the use of low discrepancy sequences. For a comprehesive review of sequences and rules see Judd (1998). LDS are deterministic sequences which cover the domain of the integral more evenly than random samples. Discrepancy is in this case a measure of deviation from perfect uniformity over the domain of the integral. Thus to go from MC in (13) to QMC, we replace the random samples $\mathbf{X}_i$ with LDS samples. We note that the sampling of the QMC is now dependent on the dimensionality of the integral, as opposed to MC, as the LDS samples have to be drawn with respect to the dimensionality of the integral.

We consider two different types of LDS sequences, the Halton sequence and the Sobol sequence. Both sequences are popular LDS sequences, which are used in quasi-Monte Carlo (MPT) applications, (Glasserman 2004).

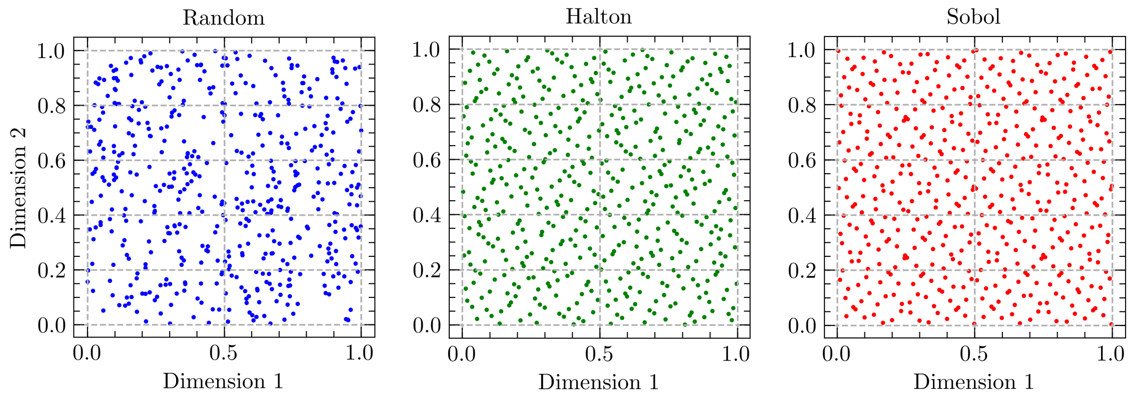
The convergence rate of MPT is:

$$\frac{(\log N)^d}{N} \tag{14}$$

Hence QMC is generally faster than MC, e.g $\frac{(\log N)^d}{N} < \frac{1}{\sqrt{N}}$ for large N and small d. We note that as dimensionality d increases, the quality of the Halton sequence decreases, as the dimensions become more correlated, Glasserman (2004). Specifically the Halton seuqnce will produce diagonal points when projected onto a 2D plane. This is displayed in figure 1.1. We therefore prefer the Sobol sequence when the dimensionality is sufficiently high, and as not to complicate matters, also use the Sobol sequence in lower dimensions, when MPT schemes are used. Figures below shows Random samples, Halton samples and Sobol samples in 2d. Second figure shows the same in 18 dimensions. Halton shows that dimension 17 and 18 are correlated.

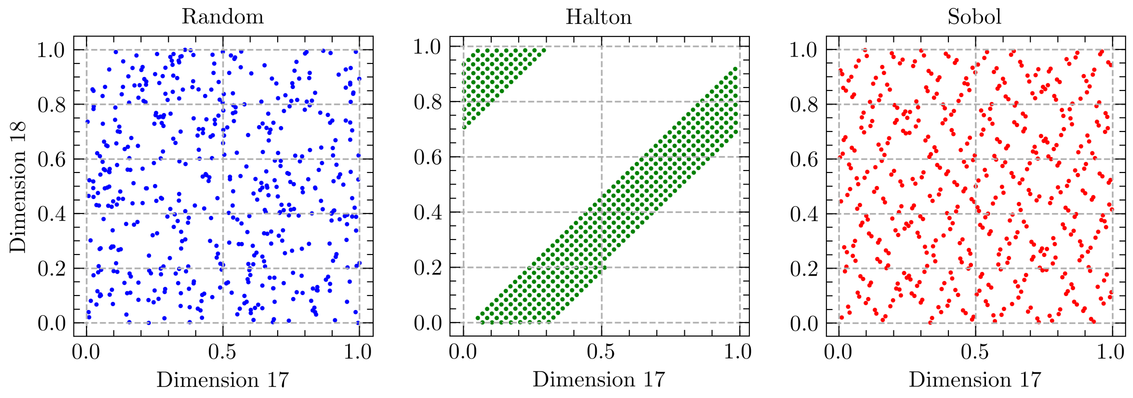
QMC is generally found to be more efficient than MC, as noted by Glasserman (2004), Judd (1998), and notably Glasserman find that dimensionality has to be quite large before

Figure 1.1: Comparison of sample generation for Monte Carlo and Quasi-Monte Carlo



Note: Each sequence was generated using $N = 500$ samples and $d = 2$ dimensions.

Figure 1.2: Comparison of sample generation for Monte Carlo and Quasi-Monte Carlo with increased dimensionality



Note: Each sequence was generated using $N = 500$ samples and $d = 18$ dimensions.

the Monte Carlo method is favorable to the quasi Monte Carlo method. Furthermore Glasserman find that while we generally might assume that N must by increase a lot when d is increased, this is not always the case in classic financial applications, as the integrals employed in these examples can often be approximated by integrals of much lower dimension. QMC therefore performs better than to be expected.

However we note that MPT lacks a straightforward variance estimator, a feature recovered through *randomized QMC*, which will be discussed in the next section.

1.1.4 Randomized Quasi-Monte carlo integration (RQMC)

Randomized quasi-Monte Carlo integration (RQMC) is a combination of MPT and MC integration. We consider the the QMC integral, i.e the equaiton of (13), using an LDS sequence. The point of randomized quasi-Monte Carlo (MPT) is then to introduce randomness to the sequence: $P_n = \{x_1, \ldots, x_n\}$. We will cover the most simple case, *Random shift* and *Scrambling* methods, however for a comprehensive review of randomization methods see Glasserman (2004). The most simple method of randomizing P_n is to add a *random shift* to each point in the sequence, using random numbers drawn from a uniform distribution of the same dimensionality as the sequence, wrapped to the interval of P_n. Hence if $x_i \in [0,1)^d$ then we add a random shift $u_i \bmod 1$, where $\bmod 1$ keeps the shift within the interval $[0,1)$. A major disadvantage of the random shift is that is changes the discrepancy properties of the sequence, and hence the quality of the sequence is lost. Scrambled nets is a method of randomization which can be applied to LDS sequences specifically. Scrambling works by applying a sequence of random permutations to the digits in the base-b representation of each coordinate in the LDS. Each digit is permuted based on the values of the digits that came before it. This structure retains the low-discrepancy properties while introducing a controlled level of randomness, which enables the calculation of variance for RQMC estimates. In multi-dimensional settings, this scrambling is applied independently to each coordinate of the sequence, allowing us to estimate variance across the entire space. Scrambling the Sobol sequence has been found to be particularily effective in financial applications, as noted by Hok and Kucherenko (2023). QMC is generally more efficient than MC, and RQMC increases the rate of converence of QMC and allows for the estimation of variance.

1.2 Value function approximation

This section covers the necessary function approximation methods used in the solution algorithm. We will cover the use of Gaussian process regression (GPR) and Baysian optimization, in order to maximize the value function of the dynamic portfolio allocation problem.

1.2.1 Gaussian process regressions (GPR)

A Gaussian process (GP) is a probabilistic model that defines a distribution over functions used to make predictions based on available data. It is specified by two functions: the mean function and the covariance function, also called the kernel. The mean function, $m(\mathbf{x})$, represents the expected value of the function at a given input $\mathbf{x}$, and the covariance function, $k(\mathbf{x}, \mathbf{x}')$, captures the covariance between function values at different input points $\mathbf{x}$ and $\mathbf{x}'$. In a GP, any finite set of input points $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ within the domain $\mathbb{R}^d$ results in the function values $\mathbf{f} = (f(\mathbf{x}_1), \ldots, f(\mathbf{x}_N))$ having a joint multivariate Gaussian distribution. This property enables a GP to provide a prior distribution over functions based on the defined mean and covariance.

We use GPR to estimate the value function in the dynamic portfolio allocation problem, when we are not at the terminal period, i.e., $t < T$, following Gaegauf, Scheidegger and Trojani (2023). The GP is formulated by the previously mentioned mean and covariance functions:

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')), \tag{15}$$

The covariance kernel function $k(\mathbf{x}, \mathbf{x}')$ can be any Mercer kernel, i.e., positive definite (Murphy 2023). Common kernel choices include the Radial Basis Function (RBF) kernel, the Matern kernel, and the Exponential kernel. We employ a Matern kernel, which, depending on the parameter ν, can be a generalization of the RBF kernel or the Exponential kernel. This choice follows Gaegauf, Scheidegger and Trojani (2023). The Matern kernel is given by:

$$k_{\text{Matern}}(\mathbf{x}, \mathbf{x}') = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell} \right) K_\nu \left(\frac{\sqrt{2\nu}\|\mathbf{x} - \mathbf{x}'\|_2}{\ell} \right), \tag{16}$$

where $\|\cdot\|_2$ is the Euclidean norm, Γ is the gamma function, and K_ν is the modified Bessel function. The length scale ℓ and smoothness parameter ν are both positive. As $\nu \to \infty$, the Matern kernel converges to the RBF kernel (Gonzalvez et al. 2019). Functions from this class are k-times differentiable when $\nu > k$. When $\nu = 1/2$, the Matern kernel corresponds to the Ornstein-Uhlenbeck process (Murphy 2023), which is commonly used in financial applications, such as models of interest rates (Glasserman 2004).

Consider a training dataset $\{\mathbf{X}, \mathbf{y}\}$ with N states $\mathbf{x}_i$ and observed values $\mathbf{y}$. We assume that the observations $\mathbf{y}$ are generated by an unknown function f, such that

$$y_i = f(\mathbf{x}_i) + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma_\varepsilon^2),$$

where σ_ε^2 represents the observational noise[2]. The goal is to train a GP on this dataset and then use it to predict the value function at a new state $\mathbf{x}_*$, yielding a new predicted output f_*.

The training observations $\mathbf{y}$ and the predicted noise-free function f_* have a joint Gaussian distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} k(\mathbf{X}, \mathbf{X}) + \sigma_\varepsilon^2 \mathbf{I} & k(\mathbf{X}, \mathbf{x}_*) \\ k(\mathbf{x}_*, \mathbf{X}) & k(\mathbf{x}_*, \mathbf{x}_*) \end{bmatrix} \right) \tag{17}$$

[2]The noise assumption implies that the GP model does not interpolate the data but rather fits a smooth function. This results in computational costs of $O(N)$ for the mean prediction and $O(N^2)$ for the variance prediction. For more details, see (Murphy 2023).

Here i have assumed a zero mean function[3], and the kernel function is the Matern kernel. The posterior distribution of the predicted value function f_* given the training data is then a multivariate normal (Murphy 2023), with mean:

$$\tilde{\mu}(\mathbf{x}) = k(\mathbf{x}_*, \mathbf{X})[k(\mathbf{X}, \mathbf{X}) + \sigma_\varepsilon^2 \mathbf{I}]^{-1}\mathbf{y}, \tag{18}$$

And covariance:

$$\tilde{k}(\mathbf{x}_*, \mathbf{x}_*^{'}) = k(\mathbf{x}_*, \mathbf{x}_*^{'}) - k(\mathbf{x}_*, \mathbf{X})[k(\mathbf{X}, \mathbf{X}) + \sigma_\varepsilon^2 \mathbf{I}]^{-1} k(\mathbf{X}, \mathbf{x}_*^{'}) \tag{19}$$

Therefore in order to predict the value function at a new state $\mathbf{x}_*$, we need to compute the mean and covariance. This step is computationally burdensome as we have to compute the four covariance matrices in the joint distribution (17). Afterwards we can compute predictions using the mean function (18) and the covariance function (19) can be used to compute error bands on our predictions.

As noted, training and predicting with a GP is computationally expensive. I will therefore introduce the methods employed to reduce the computational burden of the GP.

We use automatic relevance detection (ARD) which is a modification to the Matern kernel to use a length scale for each dimension, ℓ_i. Dimensions with low impact has a high length scale, and are effectively ignored. Note that this is not the same as Lasso, as these coefficients are not set to 0. We use Structured Kernel Interpolation for Products (SKIP) to reduce the computational burden of computen the matrices in the joint distribution (17).

Use LancZos Variance Estimate (Love) and SKIP to reduce computational burden.

Here is the documentation in my package `https://docs.gpytorch.ai/en/stable/examples/02_Scalable_Exact_GPs/index.html`

And here is a paper on the subject `https://arxiv.org/pdf/1803.06058`.

1.3 Approximating the No trade region

Since i now have introduced methods to approximate the next-period value function v_{t+1}, and methods for evaluating the expectation $\mathbb{E}[\cdot]$ over known distributions, we can now approximate the no-trade-region (NTR) using a dynamic programming (DP) scheme. In order to do this some assummptions regarding the unknown NTR are formed, these are drawn directly from (Gaegauf, Scheidegger and Trojani 2023)

Assumption 1. *The NTR is a D-dimensional convex polytope.*

[3]Zero mean ... XXXX

A polytope is a generalization of a polyhedron (polytope in 2D), which is a geometric object with flat sides and straight edges. The convex polytope is a polytope which bounds a convex set, and can therefore be defined by a convex hull. Hence, any linear combination of points in the NTR or on the boundary of the NTR is also in the NTR. In other words, the NTR is a closed convex set. **Wikipedia reference her?**.

Assumption 2. *The NTR has 2^D vertices.*

This assumption is regarding the shape of the NTR. Note that if the actual NTR has less than 2^D vertices, the approximation will be close to the actual shape, as the approximated vertices will be on top of each other. However if the NTR as more then 2^D vertices, then the approximation will be a simplification of the actual shape. The existing litterature finds that the NTR is a D-dimensional parallelogram, this is formally shown with uncorrelated assets by (Liu 2004), and with correlated assets the same is found by (Cai, Judd and Xu 2013; Dybvig and Pezzo 2020). Hence i believe this sampling scheme to be sufficient, for the case of proportional transaction costs.

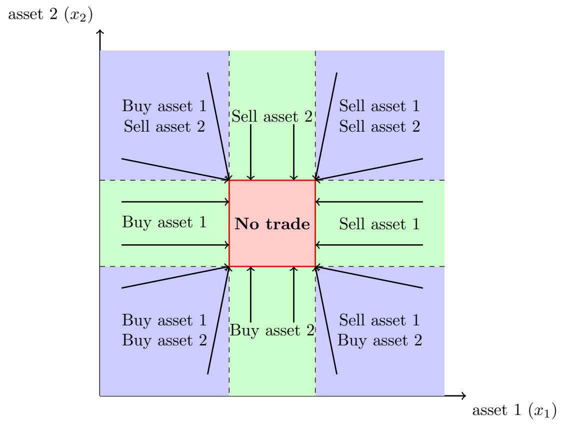
(Dybvig and Pezzo 2020) find that the NTR is a circle or ellipse when there are only fixed costs, and when there are asset specifc costs the NTR is a hexagon in the 2D case, as one vertice is added per asset. This would suggest other sampling schemes for these cases. For the circular case i would need to sample evenly around the circle (sphere / hypersphere), this problem is well known in mathematics and computer graphics and many methods for this exists, among others lattice point methods. For more on this see for example (*Distributing points on the sphere* n.d.) or (Bono, Nicoletti and Ricci-Tersenghi 2024). For the hexagon case, i could add more midpoints between the vertices of the existing sampling scheme.

With these two assumptions in place, a strategy for approximating the NTR can be formed with few initial points. Given assumption 2 we can approximate the NTR by using 2^D points, which are the vertices of the NTR, and by assumption 1 we can approximate the NTR by using the convex hull of these points, i.e connecting the vertices by straight lines to form the outer hull.

I can leverage the following intution from (Gaegauf, Scheidegger and Trojani 2023), and from **??**: For any point outside the NTR, the optimal policy is to trade towards the boundary of the NTR. Since each point on the boundary of the NTR is optimal, the optimal trading route minimizes the distance, and hence the optimal trading route is a straight line to the boundary of the NTR. If the points ahre chosen correctly, the optimal trading route will be to a vertex of the NTR. This is seen in the figure below:

If one considers the example in figure 1.3, i can effectively approximate the NTR, by sampling a point in each of the blue regions, and then solving the optimization problem to find the vertices. When the NTR is unknown, sampling from the blue regions seem difficult at a first glance. However, i can sample the vertices of each simplex that covers

Figure 1.3: Illustration of the no-trade region (NTR) and the optimal policies outside this.



This is a schematic NTR. Blue regions are regions where optimal policy $\boldsymbol{\delta}$ is to adjust both asset allocations. Green regions are regions where the optimal policy is to hold in one asset and adjust the other. This figure is a recreation of Figure 1. in Gaegauf, Scheidegger and Trojani (2023).

the feasible space, and the midpoints between these. This sampling scheme leads to the following points in the 2-dimensional case:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0.5 & 0.5 \end{bmatrix}$$

Extensions of this sampling scheme to higher dimensions is trivial. This should effectively cover the feasible space, and allow for approximating the NTR. Note that this sampling scheme only covers NTR with no borrowing, and no short-selling as noted in (Gaegauf, Scheidegger and Trojani 2023). If borrowing and short-selling were introduced, we would have to set some bounds on the borrowing and short-selling, and then sample from these bounds. Effectively creating a square (cube / hypercube), around the feasible space, and then sample the vertices of this space.

Having approximated the NTR, we can now use this in the solution algorithm. There are two main ways which the NTR approximation can be leveraged in order to lessen the computational burden of the solution algorithm. these will be covered below.

1.3.1 Strategic point sampling

After having approximated the NTR i need to efficiently approximate the value function in the time step related to the NTR. This is done by sampling points over the entire feasible space, and then solving for the optimal trade route for each point. In order to ensure that the approximation of the value function is of high quality, and that this value function can effectively be used for any point in the state space, we need to ensure that the points are sampled in a strategic manner. This means i need points of a few different types: I need points inside the NTR, and around the NTR in any direction, and various distances to the NTR. This leads to three types of points i need to sample: *Points inside the NTR*, *points near the kinks of the NTR* and *points in the general state space, outside the NTR*. An easily implemntable solution is to use a naive grid sampling method, such as uniform draws over the feasible state space, or to use a grid-method which evenly covers the feasible state space. However, a simple naive grid method for sampling points over the state-space has a few drawbacks which i need to tackle.

First of all, a naive grid method, such as uniform draws, will not cover the NTR efficiently, especially for small NTRs. I would need a large amount of grid points to be sure that there are multiple points inside the NTR. A pure random grid would likewise need a large amount of points, in order to cover the NTR efficiently, especially in each direction around the NTR. Both of these methods, and a schematic NTR are shown in appendix **??**.
I therefore instead follow the method of (Gaegauf, Scheidegger and Trojani 2023), and sample points in a strategic manner. This scheme consists of the three point types mentioned earlier, with a sampling method for each of these points. Having approximated the NTR, i can effectivly sample point in this manner:

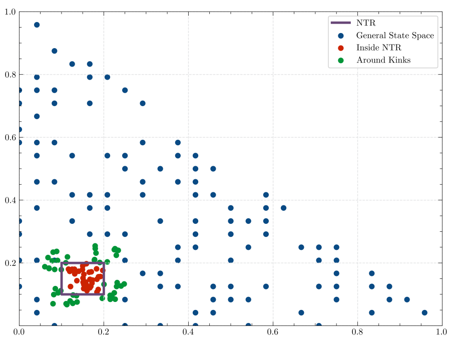
1. Sample points outside the NTR in the general state space using a uniform grid. I then remove all points inside the NTR, and sample random grid points until i have enough points.

2. Sample points inside the NTR. For this i consider random draws, as the placement inside the NTR is not of high importance. I just need enough to approximate the value function.

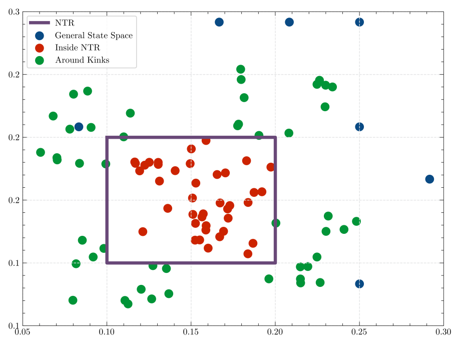
3. Sample points around the NTR kinks. For this i consider each NTR vertice. I then interpolate between adjacent vertices slightly, and extend these outward with random noise draws.

The resulting points are plotted in figure 1.4a, and a zoom in on the NTR kinks are shown in figure 1.4b. (Gaegauf, Scheidegger and Trojani 2023) find that especially increased sampling around the kinks, leads to a better approximation of the value function, and $N > 100$ points leads to sufficient approximations, as most of the approximation error is due to the kinks of the NTR. This choice of sampling scheme furthermore reduces the strain by curse of dimensionality, as grid sampling schemes would increase the number of points exponentially with the number of dimensions. Note that while this scheme still increases the number of points needed with the dimensionality, the oversampling of grid points especially reduce the number of points needed in higher dimensions.

Figure 1.4: The designed sampling strategy for state space coverage.



(a) The sampling strategy for the 2-dimensional case



(b) A zoom in on the NTR for the sampling strategy

Note: Sample consists of $N = 200$ points, with 122 points in the general state space (55%), 40 points inside the NTR (20%) and 48 points around the NTR kinks (25%).

1.3.2 Utilising the NTR approximation for $\boldsymbol{\delta}$ bounds

Having constructed an efficient sampling strategy, i can further leverage the NTR approximation to find bounds on the optimal policy $\boldsymbol{\delta}$, for the optimization step for each of these points. For this consider the schematic NTR in figure 1.3. At each point outside the NTR, the optimal policy is to trade towards the boundary of the NTR. This can either mean trading towards a vertice of the NTR or one of the faces. For the blue regions, trading towards a vertice is optimal, and this means that the optimal policy is to reallocate in both risky assets.

In this case, we can set bounds on the optimal policy $\boldsymbol{\delta}$, by considering the euclidian distance to the NTR. Hence if i know beforehand, that for asset 1 we need to sell (lower-right blue region), then i can set bounds on δ_1^+ to 0 and effectively remove this from the optimization problem. I can likewise do this the other way around for the asset 2, which i need to buy more of, and set bounds on δ_2^+ to 0.

For the green regions in the figure, the optimal policy is to trade towards a face of the NTR, and this means that the optimal policy is to reallocate in one risky asset and hold the other. I can therefore set bounds on the optimal policy $\boldsymbol{\delta}_i$ to 0 for the asset which is to be held, and only consider reallocation in the second asset.

This method of setting bounds on the optimal policy $\boldsymbol{\delta}$ is a way to reduce the computational burden of the optimization problem, and to ensure that the optimization problem is well defined. Furthermore, by knowing that the optimal policy reduces the euclidian distance to the NTR, i can effectively remove policies which would suggest buying and selling the ith asset.

1.3.3 Multiple Gaussian Process Regressions

The final ingredient in the algorithm is the use of multiple GPRs. Since i now can effectively sample points, and have information on their placement relative to the NTR, i can leverage this, and estimate two seperate value functions, one inside the NTR and one outside the NTR. This strategy effectively deals with the kinks of the NTR, as this otherwise would pose a problem for any smooth function approximations. I construct one GP for the points inside the NTR, and one for the points outside the NTR, and when i then evaluate the value function at a point $v_{t+1}(\mathbf{x}_{t+1})$, i select the appropriate GP to evaluate the value function.

This is done after having optimized over the N points from the sampling strategy, i construct two datasets:

$$\mathbf{X}_{t,\text{inside}} = \{\mathbf{x}_{t,i}, \hat{v}_{t,i} \mid \mathbf{x}_{t,i} \in \hat{\boldsymbol{\Omega}}_t\} \tag{20}$$

$$\mathbf{X}_{t,\text{outside}} = \{\mathbf{x}_{t,i}, \hat{v}_{t,i} \mid \mathbf{x}_{t,i} \notin \hat{\boldsymbol{\Omega}}_t\} \tag{21}$$

Then each GP is fit over the dataset, which consists of asset allocations and the corresponding value function output. In the next period, $t-1$ (since we iterate backwards), i can then evalute the next period value functtion $v_{t+1}(\mathbf{x_{t+1}})$, by selecting the appropriate GP, and using the predictive mean from (18):

$$\tilde{\mu}(\mathbf{x}_{t+1}) = k(\mathbf{x}_{t+1}, \mathbf{X}_{t+1})[k(\mathbf{X}_{t+1}, \mathbf{X}_{t+1}) + \sigma_\varepsilon^2\mathbf{I}]^{-1}\hat{\mathbf{v}}_{t+1}, \tag{22}$$

1.4 Final solution algorithms

Now that each component regarding the solution algorithm has been covered, i can now presents the solution algorithms for the dynamic portfolio allocation problem, in pseudo code. Starting at the second to last period, which is the last period where the investment decision is not trivial, the algorithm is as follows: Sample 2^D points to approximate the NTR. Then Approximate the NTR by solving the optimization problem for these points.

Sample N points in a strategic manner, as described in **??**. For each $x_{i,t} \in X_t$ with $\{X_t\}_{i=1}^N$, solve the optimization problem to find the optimal policy $\boldsymbol{\delta}_i$.

Construct the datasets $\mathbf{X}_{t,\text{inside}}$ and $\mathbf{X}_{t,\text{outside}}$ and fit two GPRs to the datasets $\mathbf{X}_{t,\text{inside}}$ and $\mathbf{X}_{t,\text{outside}}$. The code can be split into two parts, algorithim (A) and algorithm (B). Algorithm A covers approximatin the NTR and algorithm B covers the entire DP scheme. These are drawn from the framework which has been covered, above, created by (Gaegauf, Scheidegger and Trojani 2023).

Algorithm 1. Approximate the t-th period NTR in the discrete-time finite-horizon portfolio choice model with proportional transaction costs.

```
Input  : $t+1$ period's value function approximation $V_{t+1}$.
Result : Set of approximated NTR vertices: $\{\hat{\boldsymbol{\omega}}_{i,t}\}_{i=1}^{N}$; Approximated NTR: $\hat{\boldsymbol{\Omega}}_t$.
Sample the set of $N = 2^D$ points $\tilde{\mathbf{X}}_t = \{\tilde{\mathbf{x}}_{t,i}\}_{i=1}^{N}$ using section strategy from Section 1.3.
for $\tilde{\mathbf{x}}_{i,t} \in \tilde{\mathbf{X}}_t$ :
    Obtain policy $\hat{\boldsymbol{\delta}}_{i,t}$ for $\tilde{\mathbf{x}}_{i,t}$ by solving the optimization problem using $V_{t+1}$ as the next
      period's value function. (Terminal value function in $t = T-1$)
    Compute the approximate NTR vertices $\hat{\boldsymbol{\omega}}_{i,t} = \tilde{\mathbf{x}}_{i,t} + \hat{\boldsymbol{\delta}}_{i,t}$.
end
Compute the NTR approximation: $\hat{\boldsymbol{\Omega}}_t = \{\lambda\hat{\boldsymbol{\omega}}_t \mid \lambda \in (0,1)^N, \sum_{i=1}^{N} \lambda_i = 1\}$.
```

Algorithm 2. Complete Dynamic programming scheme with Gaussian process regressions and the NTR approximation.

```
Input  : Terminal value function $v_T$; time horizon $T$; sample size $N$.
Result : Set of GP approximations of the value functions $\{v_{t-1}\}_{t=0}^{T-1}$; set of approximated
         NTRs $\{\hat{\boldsymbol{\Omega}}_t\}_{t=0}^{T-1}$, obtained policies $\{\{\boldsymbol{\delta}\}_{i=1}^{N+2^d}\}_0^{T-1}$.
Set $\mathcal{V}_T = v_T$.
for $t \in [T, \dots, 1]$ :
    Approximate NTR $\hat{\boldsymbol{\Omega}}_{t-1}$ (Alg. 1) using $\mathcal{V}_T$ as the next period's value function.
    Sample $N$ points $\mathbf{X}_{t-1} = \{\mathbf{x}_{t-1,i}\}_{i=1}^{N}$ using the constructed sampling scheme.
    for $\mathbf{x}_{i,t-1} \in \mathbf{X}_{t-1}$ :
        Obtain value $\hat{v}_{i,t-1}$ and policy $\{\hat{\delta}_{i,t-1}, \hat{c}_{i,t-1}\}$ for $\mathbf{x}_{i,t-1}$ by solving the optimization
          problem using $\mathcal{V}_t$ as the next period's value function.
    end
    Define the training sets:

        $\mathcal{D}_{\text{in},t-1} = \{(\mathbf{x}_{i,t-1}, \hat{v}_{i,t-1}) \mid \mathbf{x}_{i,t-1} \in \hat{\boldsymbol{\Omega}}_{t-1}\},$
        $\mathcal{D}_{\text{out},t-1} = \{(\mathbf{x}_{i,t-1}, \hat{v}_{i,t-1}) \mid \mathbf{x}_{i,t-1} \notin \hat{\boldsymbol{\Omega}}_{t-1}\}.$

      Given $\mathcal{D}_{\text{in},t-1}$ and $\mathcal{D}_{\text{out},t-1}$, approximate $v_{t-1}$ for inside and outside of the NTR
      $\{G_{\text{in},t-1}, G_{\text{out},t-1}\}$ (using the respective datasets) with GPs.
    Set $v_{t-1} = \{G_{\text{in},t-1}, G_{\text{out},t-1}\}$.
end
```

1.5 Computational stack and implementation

The solution algorithm is implemented in Python, and takes advantage of a simple but powerfull computational stack. following (Gaegauf, Scheidegger and Trojani 2023). The economic identities and dynamic where written using the `PyTorch` package, which is a machine learning library implemented in Python. This package has an auto-differentiation feature, which allows for easily implmentable gradients for the constrainted optimization scheme. Furthermore this package is also directly linked with the `GPyTorch` package, which is a Gaussian process library implemented using PyTorch. The GPRs were implemented using the `GPyTorch` package, which is a Gaussian process library implemented

using PyTorch. This package has multiple speedups for GPRs, such as the LancZos Variance Estimate (LOVE) and the SKIP method, which reduces the computational burden of the GPRs. Furthermore the predictive mean can be computed using black-box matrix-matrix multiplication, which is a speedup for the predictive mean computation, skipping cholesky decompositions for large matrices.
The constrained optimizer i use is the `Cyipopt` package, which is a Python wrapper for the `Ipopt` package, which is a non-linear optimization package. This package is used to solve the optimization problem for each point in the state space, and is used to find the optimal policy $\boldsymbol{\delta}$ for each point.
The gaussian quadrature grid-points where implemented with the `Tasmanian` package, which is a sparse grid package. This was taken from (Schober, Valentin and Pflüger 2022), who used this package to implement sparse adaptive grids.
Finally i implemented parallelization at two points in the code. Whenever we run the optimization scheme for a point in the state space, we can run these in parallel, as they are independent operations, as long as i do this within the same timepoint t.

1.5.1 Optimization details

When solving the optimization problem, i use a tolerance of 10^{-7}, and 1000 iterations. When approximating the NTR, i solve for each point 8 times, and select the optimal solution among these. Furthermore i multiply the starting point with a decaying factor, in the number of starts, in order to add small variance at each iteration. This is because non-linear optimization problems can be sensitive to the initial starting points.

The initial starting point is chosen within the feasible space at random, when there is no approximated NTR. The random draws are chosen to be feasible given the constraints of the problem. When i later have approximated the NTR, i use the shortest distance towards the NTR as initial guess, and multiply with a decaying factor over the number of starts. For these points i solve the optimization problem 3 times. This is because when i can leverage the knowledge of the NTR, the optimization problem is easier.

For points inside the NTR i likewise guess no trading, knowing this to be optimal a-priori. Small jitter is added to this when using multiple solutions.