Optimal Transport: Model-Free Methods in Finance

by

Rowan Austin

Department of Mathematics
King's College London
The Strand, London WC2R 2LS
United Kingdom

Email: Rowan.Austin@kcl.ac.uk
Tel: +44 (0)78 56 35 21 31

28 August 2025

Report submitted in partial fulfilment of the requirements for the degree of MSc in Financial Mathematics in the University of London

Abstract

This paper will survey the elements of Optimal Transport theory required to implement model-agnostic methods in quantitative finance, including Kantorovich Duality, entropic calibration methods, and martingale constrained Linear Programming approaches. This will be followed by several numerical demonstrations. Specifically, we begin by calibrating a joint density capable of pricing exotics on a pair of FX rates while remaining consistent with vanilla option prices on all currency pairings. We then demonstrate how a linear programming framework can be applied to find model-independent price bounds in the spirit of work by Henry-Labordère and Hobson, and explore the interpretations of the Lagrangian dual variables as static hedging strategies. We also examine the sparsity of the maximal and minimal admissible distributions when enforcing martingale constraints, and how it relates to the concept of "Left-Curtain Couplings". Finally, we consider a higher-dimensional linear programming problem in the form of the VIX/SPX calibration problem originally solved by Julien Guyon. [1]

[1]Many thanks to Martin Forde for supervising this project.

Contents

Introduction

Effective pricing and hedging of uncertain future payoffs is the central problem of Financial Mathematics. The classical approach has long been to model the underlying assets as stochastic processes and determine a no-arbitrage price by considering the expectation of the payoff under a risk-neutral measure. The efficacy of this method hinges on the model's ability to accurately capture the behaviour of the underlying assets. Thus, since the invention of the prototypical Black-Scholes model, a substantial amount of research has focused on the invention of new, more expressive classes of models better able to capture the statistical properties of the financial data. Merton [27] incorporated jump processes to better model extreme events with heavier-tailed distributions. Stochastic volatility models such as the Heston Model [22], better accounted for the presence of the observed volatility smile, and Dupire [11] provided a framework for exact fitting of such models to the observed market smiles. More recent research has found success modelling volatility as a rough or even path-dependent process [13, 19]. Well-constructed models balance tractability against expressiveness, while remaining parsimonious enough to avoid over-fitting. These advancements have improved practitioners' ability to price and hedge exotic derivatives by providing the flexibility to fit a wide variety of vanilla option data. In complete markets, the selection and calibration of a model to this data fully specifies the equivalent martingale measure, and therefore the risk-neutral price, of an exotic payoff. However, it is well understood that since there is a very wide class of martingale measures capable of satisfying any finite calibration data, there still exists substantial "Knightian Uncertainty" arising from the possibility that we have misspecified the form of our model entirely, yielding poor performance on exotic options despite fitting.

Optimal Transport, as branch of mathematics, originated in 1781 by Gaspard Monge's paper "Mémoire sur la théorie des déblais et des remblais". It was reformulated and generalised in the 1940s by Leonid Kantorovich, and has been the subject of renewed interest in recent years due

to the development of new theoretical contributions by mathematicians such as Benamou, Brenier, and Villani[3, 32], which highlighted the potential applications for the theory in a variety of applied fields. Many researchers and practitioners of quantitative finance have embraced optimal transport as a powerful complementary approach to the classical theory. It provides a non-parametric framework through which one can not only calibrate SLV models using fully-flexible functional parameters [15], but more profoundly, work in a fully model-agnostic setting. That is to say, optimal transport provides the tools for calibrating pricing densities to the market inferred marginal distributions of the asset prices alone, whilst making no structural assumptions about the dynamics of underlying assets. Crucially, optimal transport addresses the problem of model risk by allowing us to solve for robust theoretical upper and lower bounds on the price of an exotic based only on the calibration data observed in the market [1]. This represents invaluable context for any market participant seeking to price that exotic with conventional techniques, as it quantifies the price uncertainty inherent to any model even once perfectly calibrated to the available data.

This paper presents the theoretical minimum of Martingale Optimal Transport theory required to practically implement model-free methods on observed market data. Several numerical examples will be presented, selected from the growing literature on model-free density calibration and robust pricing of exotics being developed by authors such as Beiglböck, De-Marche, Guyon, Henry-Labordère, Hobson, Guo, Loeper, Oblój, and Wang [1, 20, 18, 14, 23].

Chapter 1

Elements of Optimal Transport Theory in Finance

1.1 The Classical Formulation

1.1.1 Monge-Kantorovich Problems

In the theory of continuous optimal transport, the movement of mass is represented by transformations between two probability measure spaces which we shall denote as $(X, \mathcal{A}_X, \mu)$ and $(Y, \mathcal{A}_Y, \nu)$. Transport is typically performed subject to a cost function $c : X \times Y \to \mathbb{R} \cup \{+\infty\}$, which we would like to minimise. The main sources for the theoretical results in this section are Cédric Villani [32], and a set of lecture notes by Matthew Thorpe [31].

Definition 1.1.1 (Transportation Maps)**.** *$T : X \to Y$ is a transport map between $(X, \mathcal{A}_X, \mu)$ and $(Y, \mathcal{A}_Y, \nu)$, if*

$$\nu(B) = \mu(T^{-1}(B)) \quad \forall B \in \mathcal{A}_Y. \tag{1.1}$$

We may denote $\nu = T_{\#}\mu$ as the "push-forward" measure of μ by T, and

$$\mathcal{M}_\mu(X, Y) := \{\, T : X \to Y \;\mid\; T \text{ is } \mu\text{-measurable} \,\}.$$

Equivalently, for a random variable $X \sim \mu$, ν is the law of $T(X)$.

Definition 1.1.2 (Monge Optimal Transport Problem)**.** *Given $(X, \mathcal{A}_X, \mu)$ and $(Y, \mathcal{A}_Y, \nu)$,*

$$\inf_{T \in \mathcal{M}_\mu(X,Y)} \int_X c(x, T(x))\, d\mu(x) \tag{1.2}$$

over μ-measurable maps $T : X \to Y$ subject to $\nu = T_{\#}\mu$.

Note that under the Monge formulation, each $x \in X$ is transported by a deterministic mapping to exactly one point $T(x) \in Y$. For financial applications, this is generally far too restrictive. In particular, if we calibrated a financial model using a Monge transport framework, it would always demand that there is a deterministic relationship between the assets modelled, which is rarely a reasonable assumption. In this paper, we will work exclusively with the following generalisation of these concepts due to Kantorovich:

Definition 1.1.3 (Transportation Plans)**.** *A joint probability measure π on $X \times Y$ is a transportation plan from $(X, \mathcal{A}_X, \mu)$ to $(Y, \mathcal{A}_Y, \nu)$ if it has marginals μ and ν. We denote by $\Pi(\mu, \nu)$ the set of such admissible plans, noting that the following characterisations are equivalent:*

1. *$\pi \in \Pi(\mu, \nu)$, i.e. π has marginals μ and ν*
2. *$\pi(A \times Y) = \mu(A), \quad \pi(X \times B) = \nu(B), \quad \forall A \in \mathcal{A}_X,\ B \in \mathcal{A}_Y$.*
3. *For all bounded and measurable $\varphi : X \to \mathbb{R}$ and $\psi : Y \to \mathbb{R}$,*

$$\int_{X \times Y} \big(\varphi(x) + \psi(y)\big)\, d\pi(x, y) = \int_X \varphi(x)\, d\mu(x) + \int_Y \psi(y)\, d\nu(y).$$

Remark 1.1.4. *The set $\Pi(\mu, \nu)$ is always non-empty, since the product measure $\mu \otimes \nu \in \Pi(\mu, \nu)$.*

Definition 1.1.5 (Kantorovich Optimal Transport Problem)**.** *Given $(X, \mathcal{A}_X, \mu)$ and $(Y, \mathcal{A}_Y, \nu)$,*

$$\inf_{\pi \in \Pi(\mu, \nu)} \int_{X \times Y} c(x, y)\, d\pi(x, y). \tag{1.3}$$

Financial Interpretation A common way in which the Kantorovich problem arises in a financial context is in the calibration a joint density for two assets X and Y, whose marginal densities μ and ν have been inferred from market data (see Theorem 1.3.1). The cost function $c(x, y)$ can then be chosen as a measure of deviation from a reference model (typically relative entropy), with constraints such as market data and other expected relationships between X and Y encoded into a Lagrangian. Guo et al. [16] summarise the framework whereby model calibration can be viewed as an optimal transport problem.

Proposition 1.1.6 (Existence of 1.3)**.** *The infimum in the Kanotorovich Optimal Transport Problem is attained by some $\pi^* \in \Pi(\mu, \nu)$ under mild conditions.*

Proof. Proposition 2.1 of Villani [32] proves this result assuming only semi-continuity of the cost function and that X and Y are Polish Spaces. □

Theorem 1.1.7 (Kantorovich Duality)**.** *Let $(X, \mathcal{A}_X, \mu)$ to $(Y, \mathcal{A}_Y, \nu)$ be probability spaces where X, Y are Polish spaces. Let $c : X \times Y \to [0, +\infty]$ be a lower semi-continuous cost function. Define*

$$\Phi_c = \left\{(\varphi, \psi) \in L^1(\mu) \times L^1(\nu) : \varphi(x) + \psi(y) \leq c(x, y)\right\},$$

where the constraint holds almost everywhere in X and Y with respect to their marginal measures. Then,

$$\min_{\pi \in \Pi(\mu,\nu)} \Big(\int_{X \times Y} c(x, y)\, d\pi(x, y) \Big) = \sup_{(\varphi,\psi) \in \Phi_c} \Big(\int_X \varphi\, d\mu + \int_Y \psi\, d\nu \Big).$$

Proof. Theorem 1.3 in Villani [32], who provides both a formal proof using the minimax principle and a rigorous measure-theoretic proof. □

Financial Interpretation Kantorovich Duality is one of the features of the theory of optimal transport that makes it most amenable to financial application, due to the potential for inferring quantities of practical interest from the dual functions. Specifically, we will see in section 3.1.3 that if the primal objective of the optimisation is an asset price, the dual variables can be interpreted as a form of hedging or replication strategy by analogy with risk-neutral pricing principles. See section 1.2.1 for the explicit relation between Monge-Kantorovich and Linear Programming Problems.

1.1.2 Martingale Optimal Transport

Martingale Optimal Transport is a variant of the Kantorovich problem that enforces an additional constraint on the optimal coupling, namely that we restrict our search to

$$\Pi_M(\mu, \nu) = \{\pi \in \Pi(\mu, \nu) | X \sim \mu, Y \sim \nu \implies \mathbb{E}^\pi[Y|X] = X\}.$$

For financial interpretability, the problem is best understood in its dual form which reveals the Langrangian.

Definition 1.1.8 (Dual MOT Problem [20, 2])**.**

$$\sup_{(\varphi,\psi,h) \in \Phi_c} \left\{ \int_X \varphi(x)\, d\mu(x) + \int_Y \psi(y)\, d\nu(y) \right\} \tag{1.4}$$

where Φ_c *is the set of* (φ, ψ, h) *such that* $\varphi \in L^1(\mu), \psi \in L^1(\nu)$*, and* h *is a bounded continuous function on* X *satisfying:*

$$\varphi(x) + \psi(y) + h(x)(y - x) \leq c(x, y), \quad \forall (x, y).$$

1.2 Computational Methods

1.2.1 Linear Programming

If we consider probability distributions on finite supports such that $\mu = \sum_{i=1}^{m} \alpha_i \delta_{x_i}$,and $\nu = \sum_{j=1}^{n} \beta_j \delta_{y_j}$ where $\sum_{i=1}^{m} \alpha_i = \sum_{j=1}^{n} \beta_j = 1$, $\alpha_i \geq 0$, $\beta_j \geq 0$. Denoting $c_{ij} = c(x_i, y_j)$ and $\pi_{ij} = \pi(x_i, x_j)$, the Kantorovich problem in definition 1.1.5 is reduced to solving the following linear program over the $m \times n$ variables π_{ij}:

$$\min_{\pi_{ij}} \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} \pi_{ij} \tag{1.5}$$

$$\text{such that } \pi_{ij} \geq 0, \quad \sum_{i=1}^{m} \pi_{ij} = \beta_j, \quad \sum_{j=1}^{n} \pi_{ij} = \alpha_i \quad \forall i, j.$$

Assumption 1.2.1 (LP Approximation)**.** *We assume that Kantorovich problems, including Martingale constrained problems, can be approximated to an arbitrary degree of accuracy through discretisation and linear programming.*

This discretisation of the problem gives us a powerful tool for solving optimal transport problems computationally, by treating them as a continuous limit of linear program that can be solved very efficiently using standard algorithms. In this context, it is the Strong Duality Theorem of Linear Programming (Theorem 4.4 of [4]) that allows seamless switching between solving the primal and dual Kantorovich problems. Section 3.1 contains a detailed linear programming framework for determining the maximum and minimum admissible prices for an exotic option using martingale optimal transport.

Performance of Solvers Linear programs are typically solved with variants of either a *Simplex* algorithm or an *Interior-Point* algorithm. Chapters 4 and 9 of [4] present these two algorithms alongside a discussion of their

worst-case complexity. The Simplex algorithm, due to Dantzig (1947), provably converges in a finite number of iterations and theoretically has a worst-case complexity of $\mathcal{O}(2^N)$ where N is the number of constraints. However, this is typically only observed in extremely pathological cases and in practice, optimal transport LPs such as (1.5) converge in low-order polynomial time in the number of constraint $(m+n)$. By contrast, the convergence of the interior point method only holds asymptotically; however, its worst-case complexity is provably polynomial in $m+n$. See Appendix A.1 for example usage of `Scipy.Optimize.Linprog` and the commercial solver `Mosek`, which are both designed to select between the two methods as needed.

1.2.2 Entropic Regularisation and Sinkhorn's Algorithm

Solving Monge-Kantorovich problems directly with linear programming can be computationally demanding, and may lead to highly irregular transport plans. The standard method of addressing this is to add a regularisation term to the original problem, designed to penalise deviation from a smooth reference model (for instance the product measure). The computational advantages of this approach were first noted by Cuturi [9, 29], who noticed the problem could be solved extremely efficiently using a matrix-scaling algorithm due to Sinkhorn[30]. The approach has been rapidly adopted by a wide range of practitioners [18, 10], for its superior rate of convergence compared with LP Simplex methods [29, 9], and the smoothness of the solutions it generates. It was expanded to work in multi-dimensional martingale constrained settings by De March[26].

Definition 1.2.2 (Relative Entropy (Kullback-Liebler Divergence))**.**

$$H(\mu,\bar{\mu}) = \begin{cases} \mathbb{E}^{\mu}\left[\ln \frac{d\mu}{d\bar{\mu}}\right] = \int \ln\left(\frac{d\mu}{d\bar{\mu}}\right) d\mu & \text{if } \mu \ll \bar{\mu}, \\ +\infty & \text{otherwise.} \end{cases}$$

Definition 1.2.3 (Entropically Regularised OT Problem)**.**

$$\min_{\pi\in\Pi(\mu,\nu)} \int_{X\times Y} c(x,y)\, d\pi(x,y) + \varepsilon H(\pi, \mu\otimes\nu) \tag{1.6}$$

Derivation of the dual form (following Nutz[28]) We start by formulating a Lagrangian incorporating the marginal constraints using Lagrange

multipliers φ and ψ:

$$L(\pi, \varphi, \psi) = \int_{X\times Y} c(x,y)\, d\pi(x,y) + \varepsilon H(\pi, \mu\otimes\nu)$$
$$+\int_X \varphi(x)\left(\mu(x) - \int_Y \pi(x,y)\, d\nu(y)\right) d\mu(x) + \int_Y \psi(y)\left(\nu(y) - \int_X \pi(x,y)\, d\mu(x)\right) d\nu(y). \tag{1.7}$$

It can be shown that the functional derivative of the Lagrangian with respect to π is

$$c(x,y) + \varepsilon \log\left(\frac{d\pi}{d(\mu\otimes\nu)}\right) - \varphi(x) - \psi(y),$$

which set to zero for first-order optimality, yields the optimal Radon-Nikodym derivative

$$\frac{d\pi}{d(\mu\otimes\nu)} = e^{\frac{\varphi(x)+\psi(y)-c(x,y)}{\varepsilon}}.$$

Substituting this form back into the 1.7 cancels out π entirely, giving a Lagrangian depending only on φ and ψ:

$$L(\varphi,\psi) = \int_X \varphi(x)\, d\mu(x) + \int_Y \psi(y)\, d\nu(y) - \varepsilon \int_{X\times Y} e^{\frac{\varphi(x)+\psi(y)-c(x,y)}{\varepsilon}}\, d(\mu\otimes\nu)(x,y) + \varepsilon. \tag{1.8}$$

Definition 1.2.4 (Dual Entropic OT Problem)**.** *The dual form of problem 1.6 is given by:*

$$\max_{\varphi,\psi} L(\varphi,\psi) \tag{1.9}$$

where $(\varphi,\psi) \in L^1(\mu)\times L^1(\nu)$ are referred to as the problem's "Schrödinger potentials".

Remark 1.2.5. *Note that the entropic regularisation has produced a problem convex in φ and ψ. Cuturi's key insight was that this allows us to solve the problem efficiently with a fixed point approach involving iteratively enforcing the marginal constraints:*

$$\varphi(x) = -\varepsilon \log \int_Y e^{\frac{\psi(y)-c(x,y)}{\varepsilon}}\, d\nu(y) \quad \mu\text{-a.s.},$$

$$\psi(y) = -\varepsilon \log \int_X e^{\frac{\varphi(x)-c(x,y)}{\varepsilon}}\, d\mu(x) \quad \nu\text{-a.s.}$$

Theorem 1.2.6 (Convergence of the Sinkhorn Algorithm (§6 Nutz[28])). *Algorithm 1 converges to a fixed point as $N \to \infty$, giving an optimal transport plan of the form:*

$$d\pi(x,y) = e^{\frac{\varphi(x)+\psi(y)-c(x,y)}{\varepsilon}} d(\mu \otimes \nu)(x,y).$$

Algorithm 1: The General Sinkhorn Algorithm

Input: Measures μ, ν; cost function $c(x,y)$; regularization parameter ε

Output: Potentials φ, ψ

Initialise $\varphi_0(x) \leftarrow 0$;

for *$t = 0, 1, 2, \ldots, N$* **do**

  `// Update` ψ_t

  for *each y* **do**

    $\psi_t(y) \leftarrow -\varepsilon \log \int_X e^{\frac{\varphi_t(x)-c(x,y)}{\varepsilon}} d\mu(x)$;

  `// Update` φ_{t+1}

  for *each x* **do**

    $\varphi_{t+1}(x) \leftarrow -\varepsilon \log \int_Y e^{\frac{\psi_t(y)-c(x,y)}{\varepsilon}} d\nu(y)$;

return *Approximate Potentials φ_N and ψ_N*

Theorem 1.2.7 (Sinkhorn Rate of Convergence). *Algorithm 1 converges exponentially fast.*

Proof. A rigorous proof is presented in [8]. See section 2.2 for a numerical demonstration of the convergence rate. □

1.3 Inference of Marginal Densities from Call Options

1.3.1 Breeden-Litzenberger Formula

A Kantorovich formulation of optimal transport consists of searching for an optimal coupling of multiple marginal densities. The Breeden-Litzenberger formula [6] is the classical result which allows the inference of a terminal price density from call option prices at that maturity.

Theorem 1.3.1 (Breeden-Litzenbeger Formula)**.** *Assume the existence of an asset whose terminal price, S_T, admits a probability density under a risk-neutral measure $\mathbb{Q}$. Let $C(K,T) = e^{-rT} \cdot \mathbb{E}^{\mathbb{Q}}[(S_T - K)^+]$ be the risk-netural price of a call option at strike K, interest rate r, and maturity T. Then the probability density of S_T is given by:*

$$p_{S_T}(K) = e^{rT} \cdot \frac{\partial^2 C(K,T)}{\partial K^2}$$

Proof. $C(K,T) = e^{-rT} \int_0^\infty \max(S-K,0)p(S)dS = e^{-rT} \int_K^\infty (S-K)p(S)dS$

By the Leibniz integral rule,

$$\frac{\partial}{\partial K} C(K,T) = e^{-rT} \cdot [(K-K)p(K) - \int_K^\infty p(S)dS] = -e^{-rT}\mathbb{Q}(S_T > K).$$

For the second derivative we note that:

$$\frac{d}{dK}(-\mathbb{Q}(S_T > K)) = \frac{d}{dK}(1 - \mathbb{Q}(S_T > K)) = \frac{d}{dK}(\mathbb{Q}(S_T \leq K)).$$

This last quantity is the derivative of the cumulative distribution function, and so by the fundamental theorem of calculus, it is the density function evaluated at K. Thus as claimed:

$$\frac{\partial^2 C(K,T)}{\partial K^2} = e^{-rT} \cdot p_{S_T}(K)$$

□

For simplified presentation, we will assume in the examples to come that $r = 0$, especially as it is common to normalise price data by its forward price in order to work with a driftless process. To apply the Breeden-Litzenberger formula all we require is a closed-form expression for $C(K,T)$, for which we will use the "Black-76" variation of the standard Black-Scholes formula[5]. As we are looking to match real market prices, our closed-form function for the call price must account for the observed volatility smile in the asset. It will therefore take the form $C_{\text{BS}}(F,K,T,\hat{\sigma}(K))$ where $\hat{\sigma}(K) : [0,\infty) \to \mathbb{R}_{\geq 0}$ is a suitable parametrisation of the volatility smile.

1.3.2 SVI Parametrisations

We employ the Stochastic Volatility Inspired (SVI) parametrisation due to Gatheral [12]. Under this framework, the implied volatility, $\hat{\sigma}(K)$, is characterised by a five-parameter expression for its variance as:

$$\hat{\sigma}(K)^2 T = a + b\{\rho(k-m) + \sqrt{(k-m)^2 + \sigma^2}\}, \tag{1.10}$$

where $k = \log(K/F)$ with F being the forward rate of the asset.

Parameter Interpretation SVI is widely used in industry due to its simplicity, tractability, and the intuitive interpretation of its parameters as identified by Gatheral: a represents a baseline variance level, b controls the slope of the smile's wings, σ the smoothness of the vertex, ρ the skew, and m allows translation in the log-moneyness axis. The parametrised smile also has the desirable property of being asymptotically linear in log-moneyness as is observed in real market volatilities. A parametrisation of the form (1.10) is at risk of two types of static arbitrage: Calendar Arbitrage and Butterfly Arbitrage. Gatheral [12] states the following restrictions are sufficient to prevent both forms:

$$0 \leq b \leq \frac{4}{T(1+|\rho|)}, \quad -1 \leq \rho \leq 1, \quad a + b\sigma\sqrt{1-\rho^2} \geq 0.$$

Calibration Fitting the parameters to implied volatility data observed at strikes $K_1, \ldots, K_N$ is a straightforward application of standard optimisation packages such as `Scipy.Optimize` to solve the following residual sum of squares minimisation scheme:

$$\underset{a,b,\sigma,\rho,m \in \mathbb{R}}{\text{minimize}} \qquad \sum_{i=1}^{N} [\hat{\sigma}(K_i) - \text{MarketIV}(K_i)]^2, \tag{1.11}$$

subject to the Gatheral No-Arbitrage conditions. MarketIV(K_i) should be thought of as the Black-Scholes implied volatility corresponding to the observed mid-price of a call option with strike K_i.

In summary, a combination of Theorem 1.3.1, the Black-76 Formula, and a well calibrated SVI volatility smile, gives the marginal price density as

$$p(K) = \frac{\partial^2}{\partial K^2} C_{\text{BS}}(F, K, T, \hat{\sigma}(K)), \tag{1.12}$$

which, though a tedious application of the chain rule, results in a fully closed-form density function which is precisely in accordance with real market prices. Equipped with these densities, we are ready to solve financial Optimal Transport problems.

Chapter 2

Entropic Density Calibration for FX rates

We consider the commonly traded currency triple of EUR/USD, GBP/USD and EUR/GBP. That is: the price of one Euro in US Dollars, the price of one Pound in US Dollars, and the price of one Euro in Pounds, which we will denote by X, Y and Z, respectively. Note that X, Y and Z are assumed to be the value of the rates at a single future maturity T, and we expect that $Z = X/Y$ should hold under all circumstances. We use US Dollars as the domestic reference currency for pricing purposes. Our goal is to calibrate a model-agnostic joint density for X and Y, which we may use to price a derivative with any arbitrary exotic payoff depending on the terminal values of X and Y alone. Our strategy for doing this is to infer marginal densities from option price data using the framework in section 1.3, and solve the entropic optimal transport calibration problem using Sinkhorn's algorithm.

2.1 Inference of marginal FX Densities

To derive marginal densities for X,Y and Z, we use the data in Table 2.1 to fit three SVI parametrisations (1.10) with which to interpolate the implied volatility of each T-maturity asset at any strike K. A Python implementation of this process, leveraging `Scipy.Optimize`, can be found in appendix A.1.2. Table 2.2 shows the result of fitting parameters to the data in table 2.1, with a plot of the resulting curves shown in figure 2.1. We then apply Theorem 1.3.1 to infer terminal price densities from each of our currency rates X,Y and Z. Recall, this is done by composing the Black-Scholes formula for

the price of a call option with our SVI volatility curves, to give closed-form expressions for option prices on each rate for any value of K. Differentiating each expression twice by K gives the marginal density functions for X,Y and Z, which we shall refer to as μ_X,μ_Y,$\mu_Z : [0, \infty) \to \mathbb{R}$.

EUR/USD		GBP/USD		EUR/GBP	
Strike	Implied Vol	Strike	Implied Vol	Strike	Implied Vol
1.0681	0.0554	1.2456	0.060 55	0.843 86	0.038 09
1.0791	0.053 115	1.2595	0.058 665	0.849 69	0.037 25
1.0904	0.0516	1.2740	0.0573	0.855 85	0.037 225
1.1014	0.051 435	1.2883	0.057 185	0.862 34	0.038 25
1.1119	0.0523	1.3017	0.057 65	0.868 75	0.039 86

Table 2.1: One-Month Implied Volatility of call options observed using Bloomberg on 16th March 2024. Forward prices: EUR/USD 1.0903, GBP/USD 1.2738, EUR/GBP 0.85590.

Currency Pair	a	b	σ	ρ	m
EUR/USD	0.0002	0.0024	0.0198	−0.1920	0.0024
GBP/USD	0.0003	0.0013	−0.0118	−0.4738	0.0001
EUR/GBP	0.0001	0.0018	0.0140	0.0954	−0.0021

Table 2.2: Least Squares fit of the SVI curve to data in table 2.1

The specific form of the SVI marginal density can be confirmed using a symbolic differentiation tool and the full process has been implemented as a cohesive Python Library in appendix A.1.2. This code was used to produce the plot of μ_X,μ_Y and μ_Z in figure 2.2. Note that for convenience, X,Y and Z have each been normalised by their forward rate, so that the probability density functions are centred around 1.

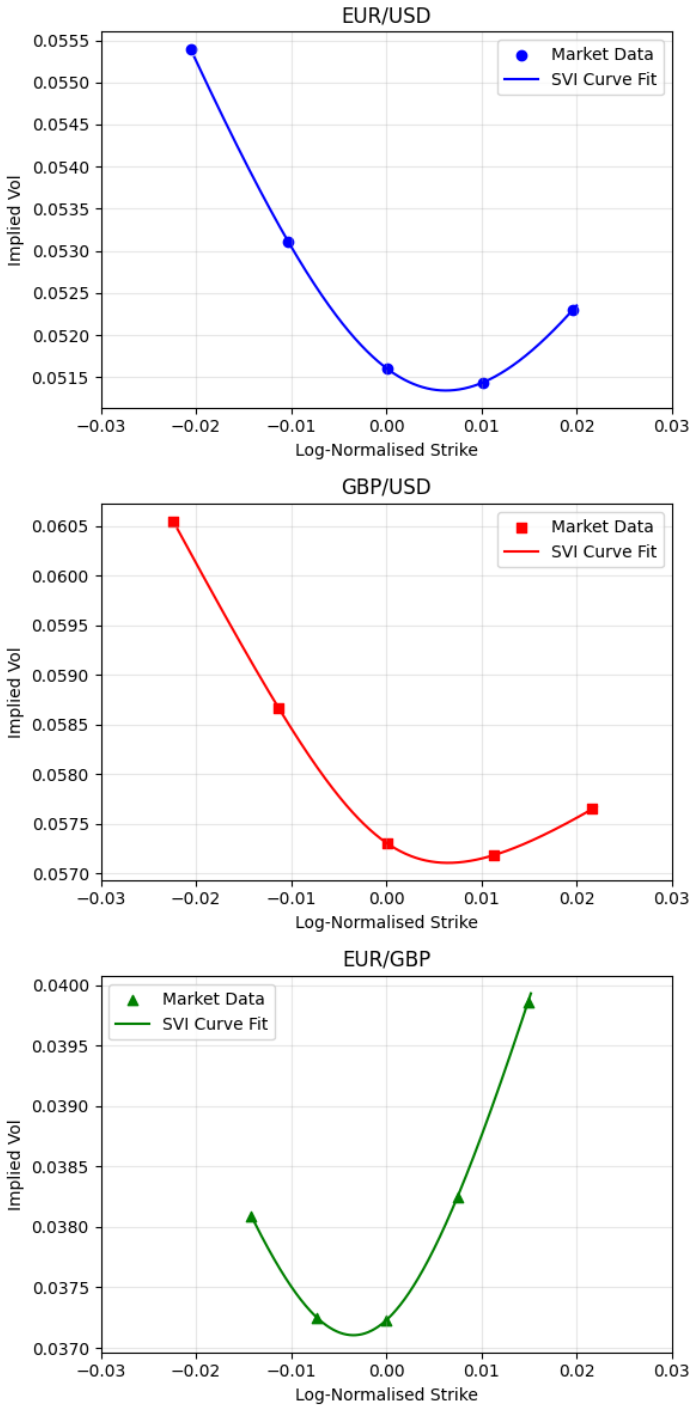

Figure 2.1: SVI smiles fit to the volatility data for call options on EUR/USD, GBP/USD and EUR/GBP (see table 2.1 for data).

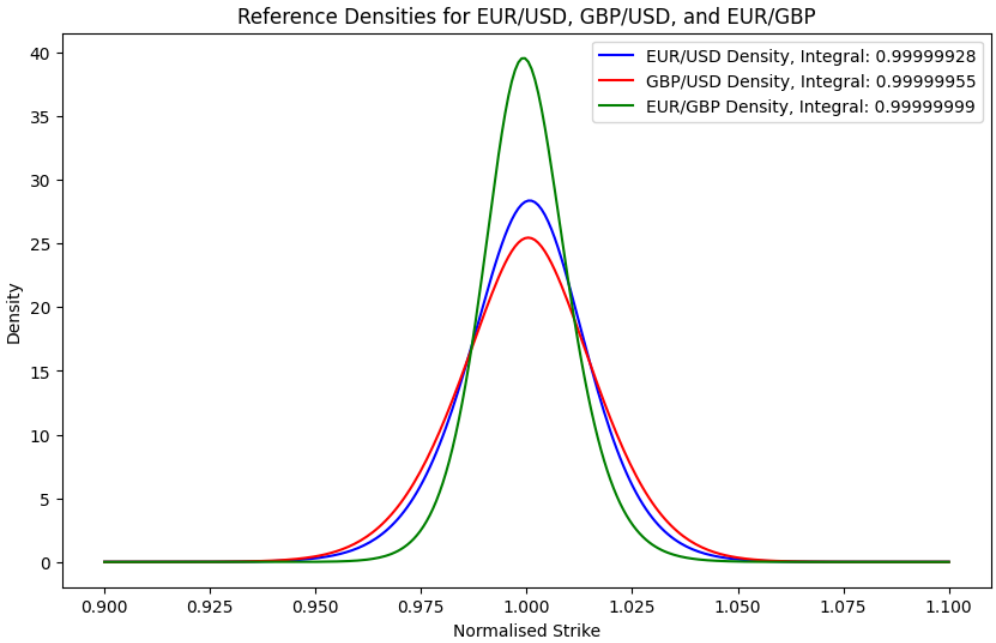

Figure 2.2: SVI density functions for X,Y and Z derived using the Breeden-Litzenberger formula on the Black-Scholes call price with the curves from figure 2.1 as the volatility input.

2.2 Calibration of the joint density

We follow section 1.2.2 in solving an EOT calibration problem subject to the marginal densities μ_X, μ_Y and μ_Z, which serve as the constraints which the desired joint density should obey. If $\Pi(\mu_X, \mu_Y)$ is the set of probability distributions on $[0,\infty) \times [0,\infty)$ with marginals μ_X and μ_Y, then we are seeking a coupling $\mu^* \in \Pi(\mu_X, \mu_Y)$ such that for all $K \geq 0$:

$$\int_{[0,\infty)\times[0,\infty)} (x - Ky)^+ \mu^*(dx, dy) = \int_{[0,\infty)} (z - K)^+ \mu_Z(dz),$$

i.e. that μ^* prices call options on $Z = X/Y$ in accordance with μ_Z. Our reference density will be $\bar{\mu}(x, y) = \mu_X(x) \cdot \mu_Y(y)$, which already conforms to the marginal constraints for X and Y. The independence between X and Y is not realistic but will be corrected as we enforce the marginal of Z. To properly scale the reference model, we will follow Guyon et al. [18] in the use of a "Gibbs factor", which can be thought of as an ansatz for the Radon-Nikodym derivative corresponding to a change of measure which appropriately calibrates the model.

Assumption 2.2.1. *There exists* $u^*, v^*, w^* \in \mathcal{C}^1([0,\infty), \mathbb{R})$ *such that*

$$\mu^*(x, y) = e^{u^*(x)+v^*(y)+yw^*\left(\frac{x}{y}\right)} \bar{\mu}(x, y) \tag{2.1}$$

Working with the ansatz 2.1 we see that the marginalisation constraints on X and Y require:

$$\mu_X(x) = e^{u(x)} \int_0^\infty e^{v(y)+yw\left(\frac{x}{y}\right)} \mu_X(x)\mu_Y(y)\, dy$$

$$\mu_Y(y) = e^{v(y)} \int_0^\infty e^{u(x)+yw\left(\frac{x}{y}\right)} \mu_X(x)\mu_Y(y)\, dx$$

These may be rearranged to:

$$u(x) = -\log \int_0^\infty e^{v(y)+yw\left(\frac{x}{y}\right)} \mu_Y(y)\, dy, \tag{2.2}$$

$$v(y) = -\log \int_0^\infty e^{u(x)+yw\left(\frac{x}{y}\right)} \mu_X(x)\, dx. \tag{2.3}$$

Note that these precisely coincide with the Schrödinger potentials of the Dual EOT Problem (see remark 1.2.5). Enforcing the Z marginal on the joint

density requires the use of a Dirac delta function to concentrate probability at the line where $K = \frac{x}{y}$.

$$\begin{aligned}
\mu_Z(K) &= \int_0^\infty \int_0^\infty y\,\delta\left(\frac{x}{y} - K\right) e^{u(x)+v(y)+yw\left(\frac{x}{y}\right)} \bar{\mu}(x,y)\,dy\,dx \\
&= \int_0^\infty \int_0^\infty \frac{x}{z}\,\delta(z-K)\,e^{u(x)+v\left(\frac{x}{z}\right)+\frac{x}{z}w(z)} \frac{x}{z^2}\,\bar{\mu}\left(x, \frac{x}{z}\right)\,dz\,dx \\
&= \int_0^\infty \frac{x}{K}\,e^{u(x)+v\left(\frac{x}{K}\right)+\frac{x}{K}w(K)} \frac{x}{K^2}\,\bar{\mu}\left(x, \frac{x}{K}\right)\,dx
\end{aligned}$$

The second equality being the result of making a change of variable, $z = \frac{x}{y}$, and the third line reflecting the Dirac function enforcing that all density is concentrated at $z = K$. Though less tractable than the previous marginals, this suggests an implicit constraint on $w(z)$ of the form:

$$\int_0^\infty e^{u(x)+v\left(\frac{x}{z}\right)+\frac{x}{z}w(z)} \frac{x^2}{z^3} \mu_X(x)\mu_Y\left(\frac{x}{z}\right)\,dx - \mu_Z(z) = 0 \tag{2.4}$$

which can be solved as a root finding exercise. Calibration now consists of numerically determining the functions u^*, v^* and w^* which can jointly fit our constraints (2.2), (2.3) and (2.4). Noting the similarity of these three equations analogous to the Schrödinger potentials in the Dual Entropic Optimal Transport problem (1.9), we make a natural extension of Algorithm 1 in section 1.2.2, to accommodate all three marginals in a discrete setting.

Algorithm 2: A Discrete Sinkhorn Algorithm for the FX Triple

Input: Discretised Grids $\{x_i\}$, $\{y_j\}$, $\{z_k\}$; target densities functions μ_X, μ_Y, μ_Z; No. iterations N

Output: u_N, v_N, w_N

Initialise $u_0(x_i) \leftarrow 0$, $v_0(y_j) \leftarrow 0$, $w_0(z_k) \leftarrow 0$ $\forall i, j, k$;

for $n = 0$ **to** $N - 1$ **do**

`// Step 1: Update` u_{n+1} `using` (2.2)

for *each* x_i **do**

$$u_{n+1}(x_i) \leftarrow -\log \int_0^\infty e^{v_n(y) + y\, w_n\left(\frac{x_i}{y}\right)} \mu_Y(y)\, dy;$$

`// Step 2: Update` v_{n+1} `using` (2.3)

for *each* y_j **do**

$$v_{n+1}(y_j) \leftarrow -\log \int_0^\infty e^{u_{n+1}(x) + y_j\, w_n\left(\frac{x}{y_j}\right)} \mu_X(x)\, dx;$$

`// Step 3: Update` w_{n+1} `using` (2.4)

for *each* z_k **do**

Perform root finding to assign $w_{n+1}(z_k)$ such that

$$\int_0^\infty e^{u_{n+1}(x) + v_{n+1}\left(\frac{x}{z_k}\right) + \frac{x}{z_k} w_{n+1}(z_k)} \frac{x^2}{z_k^3} \mu_X(x) \mu_Y\left(\frac{x}{z_k}\right) dx - \mu_Z(z_k) = 0;$$

return $\{u_N(x_i)\}$, $\{v_N(y_j)\}$, $\{w_N(z_k)\}$;

To recover a completely closed form expression for the joint density it is necessary to fit a cubic splines or similar interpolations. Our approximation of μ^* after N iterations is then given by:

$$\mu_N(x, y) = e^{u_N(x) + v_N(y) + y w_N\left(\frac{x}{y}\right)} \bar{\mu}(x, y). \tag{2.5}$$

A Python implementation of Algorithm 2 can be found in appendix A.1.3, which was used to generate figures 2.3, 2.4 and 2.5. Figure 2.3 is a semi-log plot showing that $\log(\|\mu_N - \mu_{N-10}\|_{L_2}) \to -\infty$ linearly as $N \to \infty$ with gradient of approximately $-\frac{1}{2}$. This indicates exponential convergence of the Sinkhorn algorithm, approximately as $\mathcal{O}\left(e^{-\frac{N}{2}}\right)$.

Figure 2.4 shows the evolution of the functions u, v and w as the number of iterations increases, and figure 2.5 shows the joint densities that these curves engender at each stage. Note that after only two iterations, the qualitative features of the splines and joint density are almost completely in place, in particular the dependence between X and Y is immediately established. Further iterations refine the scale of the splines while broadly retaining their

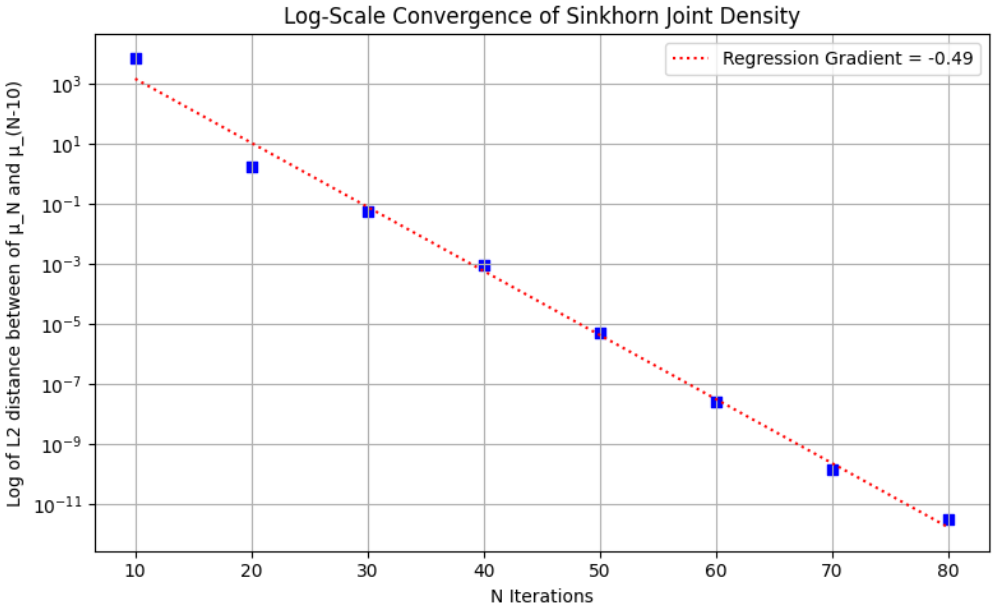

Figure 2.3: Plot of $\log(\|\mu_N - \mu_{N-10}\|_{L_2})$ against N. The linear convergence indicated by regression line suggests exponential convergence of the Sinkhorn Algorithm.

shape. The difference between ten and twenty iterations is negligible, as we would expect from the rate of convergence we have established.

2.3 Pricing from the joint density

Equipped with a joint density approximating $\mu^*(x, y)$, pricing is simply a matter of computing expectations of the payoff function under the measure associated with that density. This is nothing but two-dimensional integration, which in appendix A.1 is performed numerically via Gaussian Quadrature. For a first example, we can test that the joint density correctly prices a vanilla option on $Z = X/Y$. Figure 2.6 shows the result of pricing vanilla call Z options using equation 2.6, and the close fit to market data confirms that Sinkhorn calibration to Z option data has been successful.

$$\text{CallPrice}_Z(K, T) = \mathbb{E}^{\mu^*}[Y(\frac{X}{Y} - K)^+] = \iint_{[0,\infty)^2} (x - Ky)^+ \mu^*(x, y) \mathrm{d}x\mathrm{d}y \quad (2.6)$$

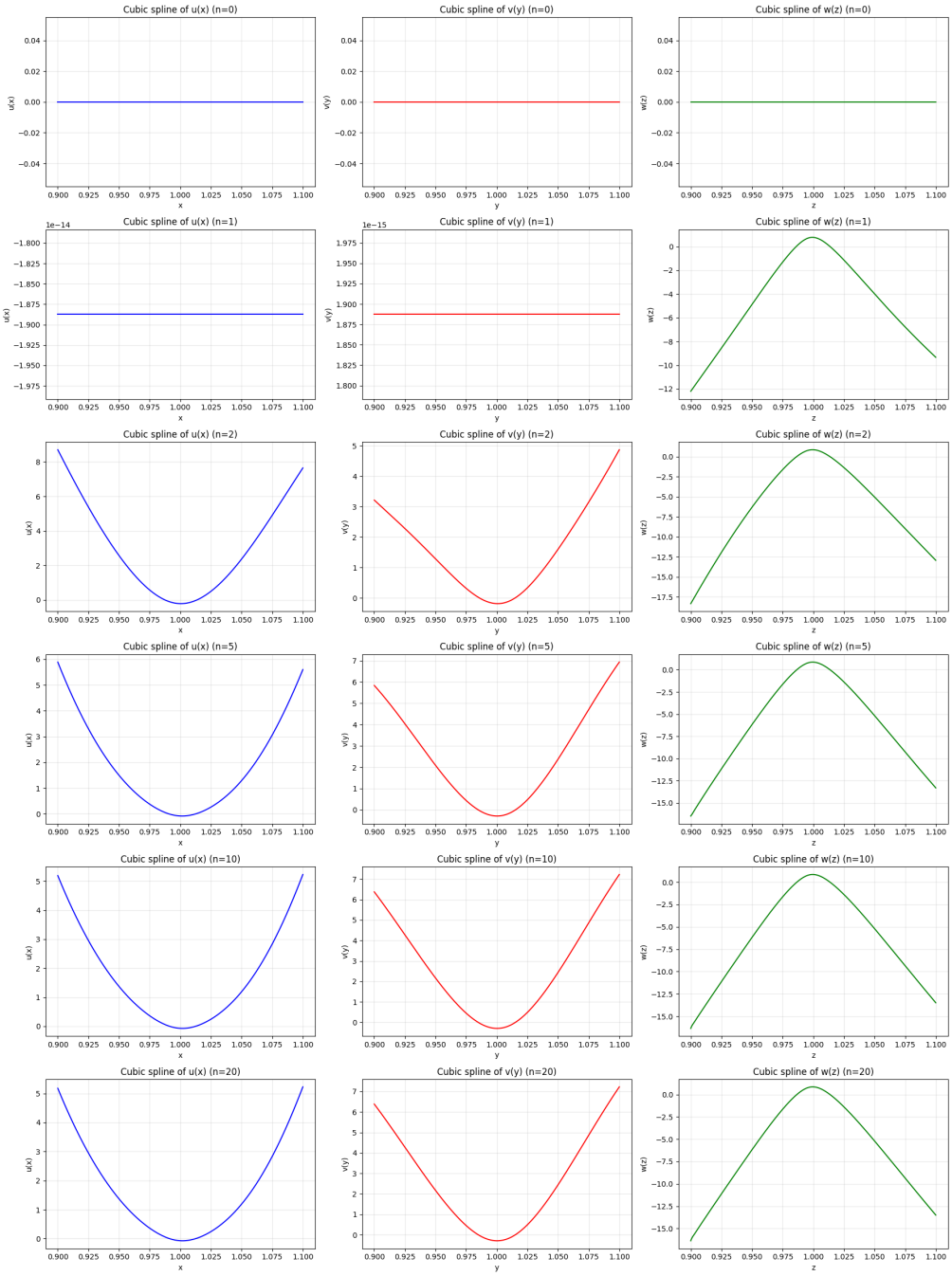

Figure 2.4: Evolution of the cubic splines of u, v and w as the number of iterations in the Sinkhorn algorithm increases.

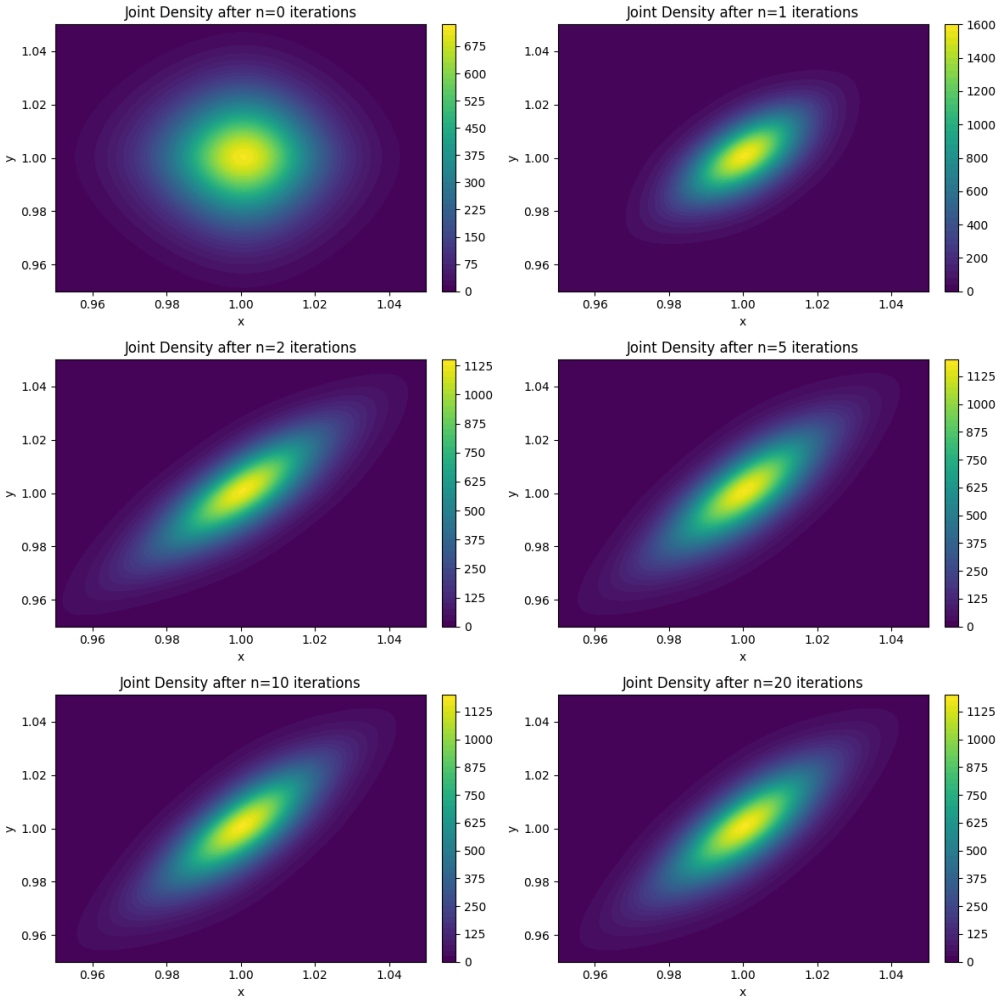

Figure 2.5: The calibrated joint densities engendered by the splines in figure 2.4. We quickly see convergence to a joint density for X and Y, calibrated to the marginals μ_X, μ_Y and μ_Z.

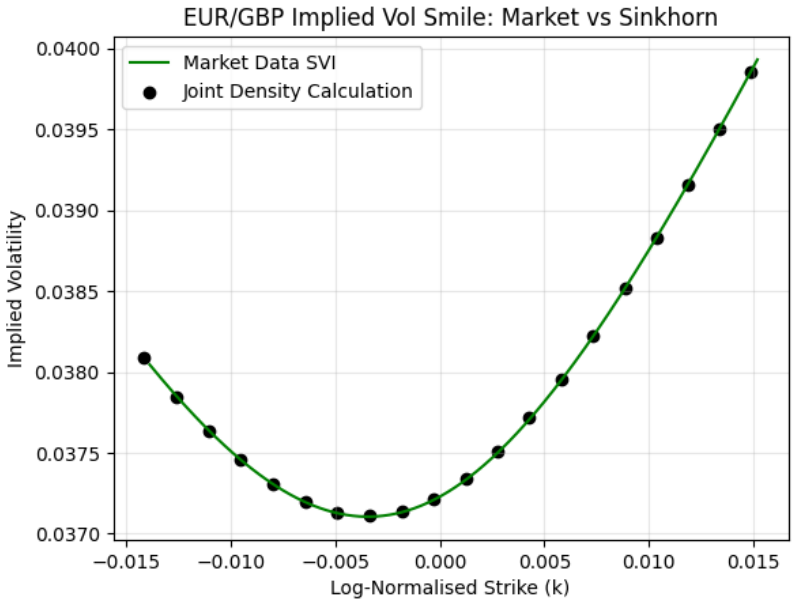

Figure 2.6: Plot showing the market observed SVI smile for Z call options, scattered with points representing prices implied by the calibrated Sinkhorn density. This is an important test to confirm that the joint density has been correctly calibrated to the options data on Z.

Given our joint density was calibrated to precisely these Z call options, we expect the result of pricing under our Sinkhorn density to agree exactly with market prices. We conclude with Table 2.3 showing the results of numerically pricing a "Basket" Call, a "Quanto" call, a "Best-Of" call and a quadratic payoff function. Our testing assures us that each of these prices is "admissible" i.e. fully consistent with the vanilla call options prices used for calibrating the joint density. However, important questions remain about the practical value of these prices. For instance, we have constructed one admissible pricing measure, however we have not quantified of the size of this class of admissible models. How sensitive are the prices associated to different payoff functions to changes in the underlying pricing measure? Does the calibration data constrain the admissible class sufficiently for practical use pricing exotics? Chapter 3 provides a numerical framework for investigating these questions.

$\mathbb{E}\left[\left(\frac{1}{2}(X+Y)-1\right)^+\right]$	$\mathbb{E}\left[\left(\frac{X}{Y}-1\right)^+\right]$	$\mathbb{E}\left[\left(\max(X,Y)-1\right)^+\right]$	$\mathbb{E}\left[(X-Y)^2\right]$
0.005886	0.004331	0.008431	0.000122

Table 2.3: Prices of different exotic payoffs under μ^*

Chapter 3

Robust Price Bounds using Martingale Optimal Transport

Our application of Entropic Optimal Transport via the Sinkhorn algorithm has yielded a plausible joint density representing an arbitrage-free pricing measure fully consistent with market data. However, it is important to note that, in most practical applications this density is not uniquely determined by the market data supplied and that it exists in a continuum of possible pricing measures. For practical application of a model-free approach to pricing and hedging assets, it is highly advantageous to know the maximum and minimum admissible prices of assets within that set of well-calibrated risk-neutral measures.

These can be approached by discretising the joint densities used in §2 into multi-dimensional probability mass functions (PMFs) and using techniques in linear programming (section 1.2.1) to maximise or minimise the price over all admissible PMFs subject to constraints analogous to those used in the Sinkhorn Algorithm. Henry-Labordère's [20] comprehensive treatment of Martingale Optimal Transport for robust hedging served as a strong influence on the presentation of the theory below.

3.1 Linear Programming - A Robust Pricing Framework

3.1.1 Probability Mass Function Calibration

Suppose we have a set of D random variables, $\{X_i\}_{i=1}^D$, that we would like to jointly model probabilistically to price exotic derivatives on their values. In full generality, this set may represent the value of D unique underlying assets at D unique future maturities, $\{T_i\}_{i=1}^D$, however most practical applications are either based on multiple assets evaluated at one maturity (as in the FX triangle example), or a small number of assets evaluated at more than one maturity (as in the VIX/SPX calibration example).

We model each random variable X_i as having a set of N discrete outcomes $\{x_1^{(i)}, \ldots, x_N^{(i)}\}$, so that the total space of outcomes in our model is the Cartesian product $\mathcal{D} = \{x_1^{(1)}, \ldots, x_N^{(1)}\} \times \ldots \times \{x_1^{(D)}, \ldots, x_N^{(D)}\}$, which consists of N^D tuples. Let $\Pi(\mathcal{D})$ be the set of probability mass functions defined over this grid. Each element of $\Pi(\mathcal{D})$ is characterised by its entries $\rho_{i_1,\ldots,i_D} = \mathbb{P}(X_1 = x_{i_1}^{(1)}, \ldots, X_D = x_{i_D}^{(D)})$ where $i_1, \ldots, i_D \in \{1, \ldots, N\}$. As probabilities these are subject to

$$\sum_{i_1,\ldots,i_D} \rho_{i_1,\ldots,i_D} = 1. \tag{3.1}$$

Under the assumption of zero interest rates, a PMF gives the fair price of an arbitrary payoff $c(X_1, \ldots, X_D)$ paid at final maturity ($\max_i\{T_i\}$) as

$$\mathbb{E}[c(X_1, \ldots, X_D)] = \sum_{i_1,\ldots,i_D} c(x_{i_1}^{(1)}, \ldots, x_{i_D}^{(D)})\rho_{i_1,\ldots,i_D}. \tag{3.2}$$

We can view this expectation as a kind of inner product between the tensor of payoffs with the tensor of probabilities.

Definition 3.1.1 (Inner Product (Expectation) of f with respect to $\rho \in \Pi(\mathcal{D})$)**.**

$$\langle f, \rho \rangle = \mathbb{E}[f(X_1, \ldots, X_D)] = \sum_{i_1,\ldots,i_D} f(x_{i_1}^{(1)}, \ldots, x_{i_D}^{(D)})\rho_{i_1,\ldots,i_D}$$

Definition 3.1.2 (Calibrated subsets of $\Pi(\mathcal{D})$)**.** *Calibration consists of restricting $\Pi(\mathcal{D})$ to a non-empty (feasible) subset of distributions satisfying specified constraints. Given equality constraints $g_1, \ldots, g_{M_{eq}} : \mathcal{D} \to \mathbb{R}$ with*

targets $\{t_1, \ldots, t_{M_{eq}}\}$*, and inequality constraints* $h_1, \ldots, h_{M_{in}} : \mathcal{D} \to \mathbb{R}$*, with bounds* $\{b_1, \ldots, b_{M_{in}}\}$ *we define the calibrated subset:*

$$\begin{aligned} \Pi_{\text{cal}} \;=\; \Big\{ \rho \in \mathbb{R}^{N^D}_{\geq 0} \;&: \langle g_j, \rho \rangle = t_j \qquad \forall\, j \in \{1, \ldots, M_{eq}\}, \\ &\langle h_k, \rho \rangle \leq b_k \qquad \forall\, k \in \{1, \ldots, M_{in}\}, \\ &\langle \mathbf{1}, \rho \rangle = 1 \Big\}, \end{aligned} \tag{3.3}$$

where $\mathbf{1}(\cdot) \equiv 1$ *on* $\mathcal{D}$*. The set of* ***admissible prices*** *for a payoff* $c(X_1, \ldots, X_D)$ *is then*

$$\mathcal{P}(c) = \{\langle c, \rho \rangle : \rho \in \Pi_{cal}\}.$$

We will see in the examples to come that the equality constraints can have many financial interpretations. They might involve enforcing the market-observed prices of various derivatives on the underlying assets, or alternatively the marginal distributions inferred from those options, or encoding other expected relations between modelled quantities such as martingale or dispersion properties. Inequality constraints can be used to provide controlled slackness in the model, such as enforcing only that the model prices assets in the range of an observed market bid and ask.

3.1.2 The Primal Optimisation Problem

Our stated aim of determining an upper and lower bound for admissible prices is now much more tangible.

Definition 3.1.3 (Least & Greatest admissible prices)**.**

$$\overline{p}(c) = \sup_{\rho \in \Pi_{\text{cal}}} \langle c, \rho \rangle, \quad \underline{p}(c) = \inf_{\rho \in \Pi_{\text{cal}}} \langle c, \rho \rangle. \tag{3.4}$$

Definition 3.1.4 (Primal Linear Program for price bounds)**.**

$$\begin{aligned} &\textit{Minimise / Maximise} \quad \underset{\rho \in \mathbb{R}^{N^D}}{} && c^\top \rho \\ &\textit{subject to} && A_{\text{eq}} \rho = b_{\text{eq}}, \\ & && A_{\text{ub}} \rho \leq b_{\text{ub}}, \\ & && \rho \geq 0. \end{aligned} \tag{3.5}$$

Despite the fact that the payoff function may be highly non-linear in the state variables, as long as the objective function and constraints are linear

in ρ, the problem can be put into a standard linear programming package such as `Scipy.Optimize.Linprog` with some minor adjustments. Notably, linear programming packages typically require the optimisation variable to be a vector, so we work above with flattened versions of c and ρ.

Conversion to Linprog with a `NumPy` "C-Style" Flattening Bijection

$$\phi : \{1, \dots, N\}^D \to \{1, \dots, N^D\}, \quad \phi(i_1, \dots, i_D) = 1 + \sum_{d=1}^{D} (i_d - 1) N^{D-d}. \tag{3.6}$$

Then the flattened version of the D-dimensional $N \times \cdots \times N$ tensor ρ is the vector $\vec{\rho} \in \mathbb{R}^{N^D}$, defined as $\vec{\rho}_{\phi(i_1,\dots,i_D)} = \rho_{i_1,\dots,i_D}$. The matrix A_{eq} should then be constructed such that each row is the flattened version of an equality constraint in (3.3), with the targets making up b_{eq}. A_{eq} and b_{ub} can then be constructed in the same way from the inequality constraints and bounds. Full Python implementations of this framework are discussed in §3.2-3.4, and provided in appendix A.1.

3.1.3 The Dual Optimisation Problem

We have seen in Theorem 1.1.7 that continuous formulations of optimal transport possess a form of strong duality. In the context of this chapter we need only appeal to the Strong Duality Theorem of Linear Programming (Theorem 4.4 of [4]), which states that when a linear programming problem has a finite optimal solution, its dual problem is feasible and possesses the same optimal solution.

Definition 3.1.5 (Dual Problem for Upper Bound)**.** *The dual linear program corresponding to the upper bound primal problem is given by:*

$$\begin{aligned} &\underset{\lambda \in \mathbb{R}^{M_{eq}}, \mu \in \mathbb{R}^{M_{ub}}_{\geq 0}}{\textit{Minimise}} && b_{\text{eq}}^\top \lambda + b_{\text{ub}}^\top \mu \\ &\textit{subject to} && A_{\text{eq}}^\top \lambda + A_{\text{ub}}^\top \mu \geq c, \\ &&& \mu \geq 0. \end{aligned} \tag{3.7}$$

Definition 3.1.6 (Dual Problem for Lower Bound)**.** *The dual linear program*

corresponding to the lower bound primal problem is given by:

$$\begin{aligned} \underset{\lambda \in \mathbb{R}^{M_{eq}}, \mu \in \mathbb{R}^{M_{ub}}_{\leq 0}}{\textit{Maximise}} \quad & b_{\text{eq}}^{\top}\lambda + b_{\text{ub}}^{\top}\mu \\ \textit{subject to} \quad & A_{\text{eq}}^{\top}\lambda + A_{\text{ub}}^{\top}\mu \leq c, \\ & \mu \leq 0. \end{aligned} \tag{3.8}$$

Financial Interpretation The objective function and its Lagrange multipliers λ, μ in problems (3.7) and (3.8) can have interesting financial interpretations depending on the form of the constraints chosen for A_{eq} and A_{ub}. When the constraints enforce market-observed prices (or bid-ask bounds) on vanilla derivatives (as in §3.2), we can interpret the Lagrange multipliers as the optimal weights of the cheapest portfolio of those assets which is capable of statically super/sub-hedging the payoff c. In this view, duality reflects a no-arbitrage condition by which the greatest (least) admissible price for a payoff c, is the cost of super-hedging (sub-hedging) that payoff using the calibration assets. When the constraints enforce a martingale condition on an asset evaluated at multiple maturities (as in §3.3 and §3.4), the Lagrange multipliers corresponding to those constraints can be viewed as specifying quantities of the underlying that should be held at each time step representing a dynamic hedging strategy in the spirit of Black-Scholes (for more details see the discussion in [1]).

3.2 Example 1: Price Bounds for FX exotics

We consider again the context of the FX triangle $X, Y,$ and Z in §2. We would like to apply the framework of §3.1 to attain upper and lower bounds for the price of an arbitrary payoff $c(X, Y)$, across discrete probability mass functions which satisfy equality constraints given by the call option prices in Table 2.1.

3.2.1 Applying the framework

PMF Grid We will model the joint distribution of X and Y on the square grid $\mathcal{D} = \{x_1, \ldots, x_N\} \times \{y_1, \ldots, y_N\}$. For convenience, we continue to work with the terminal price of X and Y normalised by their respective forward rates. An appropriate choice of the bounds of this grid can be made from

observations of the marginal distributions of X and Y. For example, figure 2.2 might suggest the use of $x_1 = y_1 = 0.9$ and $x_N = y_N = 1.1$.

Calibration to table 2.1 The subset of probability mass functions "calibrated" to the call option data is:

$$\left\{ \rho \in \Pi(\mathcal{D}) \;\middle|\; \begin{aligned} &\sum_{i,j} \max(x_i - K_x, 0)\, \rho_{ij} = C_X(K_x), && \forall K_x \in \mathcal{K}_X, \\ &\sum_{i,j} \max(y_j - K_y, 0)\, \rho_{ij} = C_Y(K_y), && \forall K_y \in \mathcal{K}_Y, \\ &\sum_{i,j} \max(x_i - K_z\, y_j, 0)\; \rho_{ij} = C_Z(K_z), && \forall K_z \in \mathcal{K}_Z, \\ &\sum_{i,j} x_i\, \rho_{ij} = 1, \\ &\sum_{i,j} y_j\, \rho_{ij} = 1 \end{aligned} \right\}$$

where $C_X(K_x)$, $C_Y(K_y)$, $C_Z(K_z)$ are the option prices observed at the sets of traded strikes: $\mathcal{K}_X$, $\mathcal{K}_Y$, and $\mathcal{K}_Z$, on $X, Y,$ and Z, respectively. Note that these quantities are all forward-normalised for consistency. The final two constraints simply enforce that the expected value of each marginal should be equal to one (since they are both forward-normalised). This is not strictly necessary to produce sensible PMFs, as enforcing the price of options close to at-the-money will encode similar information, however it allows the hedging portfolios in the dual problem to be more expressive by including forwards in X and Y in the available assets.

Python Implementation Full Python code implementing the optimisation problem 3.5 for the above constraints is available in appendix A.1.4. It takes any payoff function $c(X, Y)$ as input, and outputs the primal upper and lower price bounds, the duality gap with the dual bounds (expected to be zero) and Sinkhorn price using 30 iterations of Algorithm 2, which always lies between the bounds. Finally, it produces a bar chart showing the composition of the optimal super-hedging portfolio and a heat-map of how closely its payoff compares to the exotic payoff. This code was used to produce the results and figures in the following sections.

3.2.2 Interpreting Price Bounds Quality using Duality

Table 3.1 shows the result of using our linear programming approach to bound a range of derivatives (including those priced in §2.3), ordered according to the relative size of the difference between the lower and upper bounds which we have quantified as

$$\frac{\text{Upper Bound} - \text{Lower Bound}}{0.5 \times (\text{Upper Bound} + \text{Lower Bound})}.$$

The data suggests two important practical considerations, which we hope to explain by appealing to the dual variable interpretation.

Derivative Payoff	**Lower**	**Sinkhorn**	**Upper**	**Spread**
ATM Call $(X-1)^+$	0.005931	0.005944	0.005956	0.42%
ATM Put $(1-Y)^+$	0.006578	0.006601	0.006621	0.65%
Quanto Call $\left(\frac{X}{Y}-1\right)^+$	0.004256	0.004331	0.004439	4.21%
Basket Call $\left(\frac{1}{2}(X+Y)-1\right)^+$	0.004736	0.005886	0.006286	28.11%
Basket Put $\left(1-\frac{1}{2}(X+Y)\right)^+$	0.004736	0.005886	0.006286	28.11%
Best-of Call $\left(\max(X,Y)-1\right)^+$	0.006996	0.008431	0.009857	33.96%
Worst-of Call $\left(\min(X,Y)-1\right)^+$	0.002696	0.004114	0.005535	69.00%
Quadratic $(X-Y)^2$	0.000121	0.000122	0.000374	102.12%
Digital $\mathbf{1}_{\{\min(X,Y)>1\}}$	0.173835	0.393115	0.616783	112.05%
Far OTM Call $(X-1.03)^+$	0.000000	0.000172	0.000602	200.00%
Far OTM Put $(0.97-Y)^+$	0.000000	0.000321	0.000783	200.00%

Table 3.1: Bounds on the prices presented in §2.3 using code in A.1.4, produced using a square grid with $N = 50$. Sinkhorn prices after 30 iterations sit consistently within the linear programming bounds as expected. Ordered by relative "Spread" of bounds calculate as (Upper − Lower) ÷ (0.5 × (Upper + Lower))

Functional form of payoff The numerical duality gap in these examples was consistently less than 10^{-12}, matching our expectation of strong duality. In the general framework, we stressed that the payoff function can take any form linear in ρ while still maintaining strong duality, however, it is not surprising that constraining the optimisation to market prices on instruments with a similar functional form to the objective function leads to tighter bounds. This explains why we have extremely tight bounds on

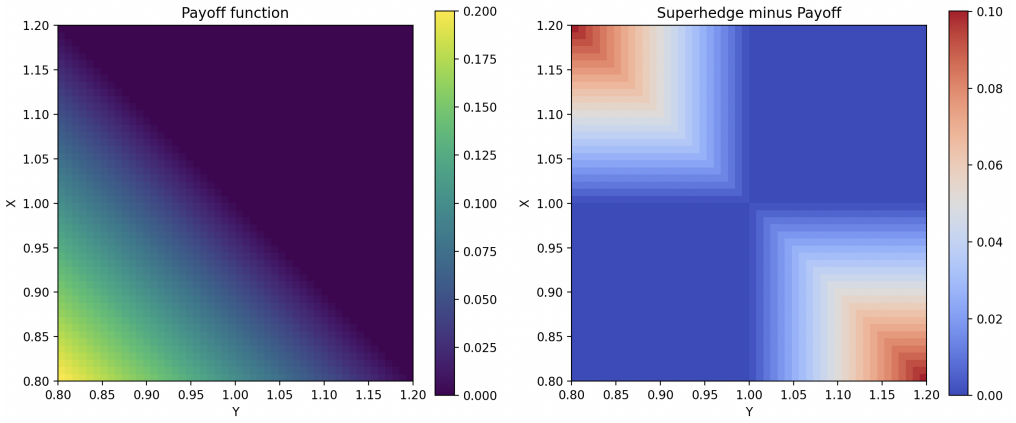

Figure 3.1: Performance of the optimal super-hedging portfolio for a put option on a basket of X and Y, which consists of half an ATM put and half an ATM call in each of X and Y.

directly observed close-to-the-money vanillas, middling bounds on exotics, such as baskets and quantos, which share some similarities in payoff profile, and very poor bounds on derivatives that have non-linear, non-convex or discontinuous payoff profiles e.g. Quadratic or digital payoffs.

Figure 3.1 shows the composition of the optimal super-hedging portfolio produced by solving the dual problem for a Basket Put, and how the super-hedge's payoff compares. The portfolio consists of approximately one ATM call on X, one ATM call on Y, and short positions in half an X forward and a half a Y forward. By put-call parity, this is equivalent to half a put and half a call in each rate. We observe that the super-hedging portfolio is able to closely replicate the payoff exactly in most regions while modestly out-performing in two areas. In the dual interpretation, it is from the cost of this out-performance in super-hedge and sub-hedge that the upper and lower price bounds arise. If a payoff could be perfectly replicated by the market instruments we have observed, we would know its price with certainty.

Figure 3.2 shows the contrasting case of a digital option paying $\mathbf{1}_{\{\min(X,Y)>1\}}$, where the discontinuous form of the payoff prohibits close replication by our observed assets. Super-hedging or sub-hedging portfolios for this payoff are a more complex mix of long and short call positions in X,Y and Z. These portfolios are relatively costly, and outperform the digital payoff significantly in some scenarios, resulting in a high spread between upper and lower bound for this asset (see Table 3.1).

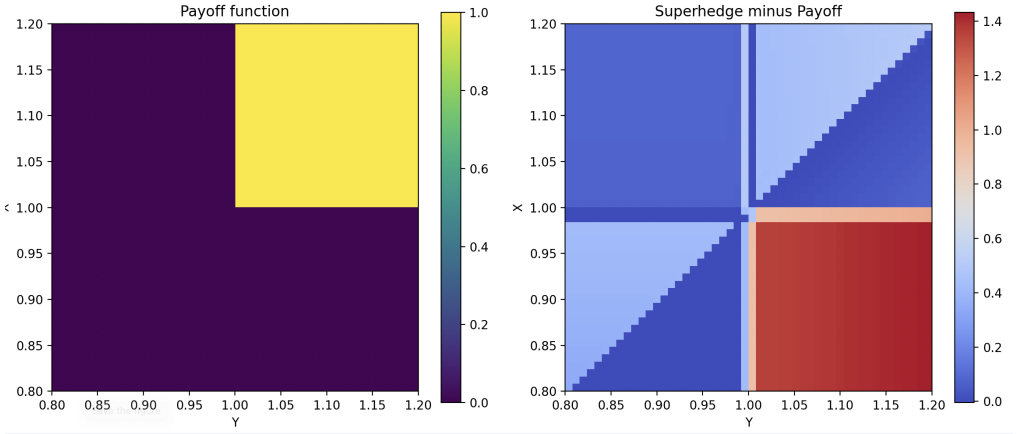

Figure 3.2: Composition and performance of the optimal super-hedging portfolio for an option paying $\mathbf{1}_{\{\min(X,Y)>1\}}$.

Out-of-Sample Payoffs Even restricting attention to payoffs with the same functional form as our constraining assets, assignment of bounds can fail when exploring scenarios far from the observed data. We note that the worst bounds attained in Table 3.1 are those on the far out-of-the-money vanilla options. $K = 1.02$ is the highest strike call option in X in the calibration set, therefore, when we attempt to bound the price of a call option even further out-of-the-money, such as the payoff $(X - 1.03)^+$, we face the challenge that any instrument in our dual portfolio capable of super-hedging this payoff will outperform it significantly, meaning our upper bound is likely very loose since it is only informed by the fact $E[(X - 1.03)^+] \leq E[(X - 1.02)^+]$. Moreover, there is no asset in our calibrating set guaranteed to sub-replicate this contract, so we have no way of setting a lower bound at all. This could be viewed as a desirable property of the approach, as it suggests we are not extrapolating prices on scenarios where we have received little information from the market. These effects could, for example, be mitigated by calibrating to the full SVI-inferred marginals of the X and Y, instead of the individual call prices, however it should be recognised this represents an extrapolation which is arguably contrary to a model-free approach.

We observe a similar effect in the right tail of the basket option price plotted against strikes in figure 3.3. One can clearly observe a point when the upper and lower bound become constant while the Sinkhorn price continues decreasing. Practitioners should question the reliability of the Sinkhorn price in this region, given the significant uncertainty expressed by a linear programming framework calibrated with the same data.

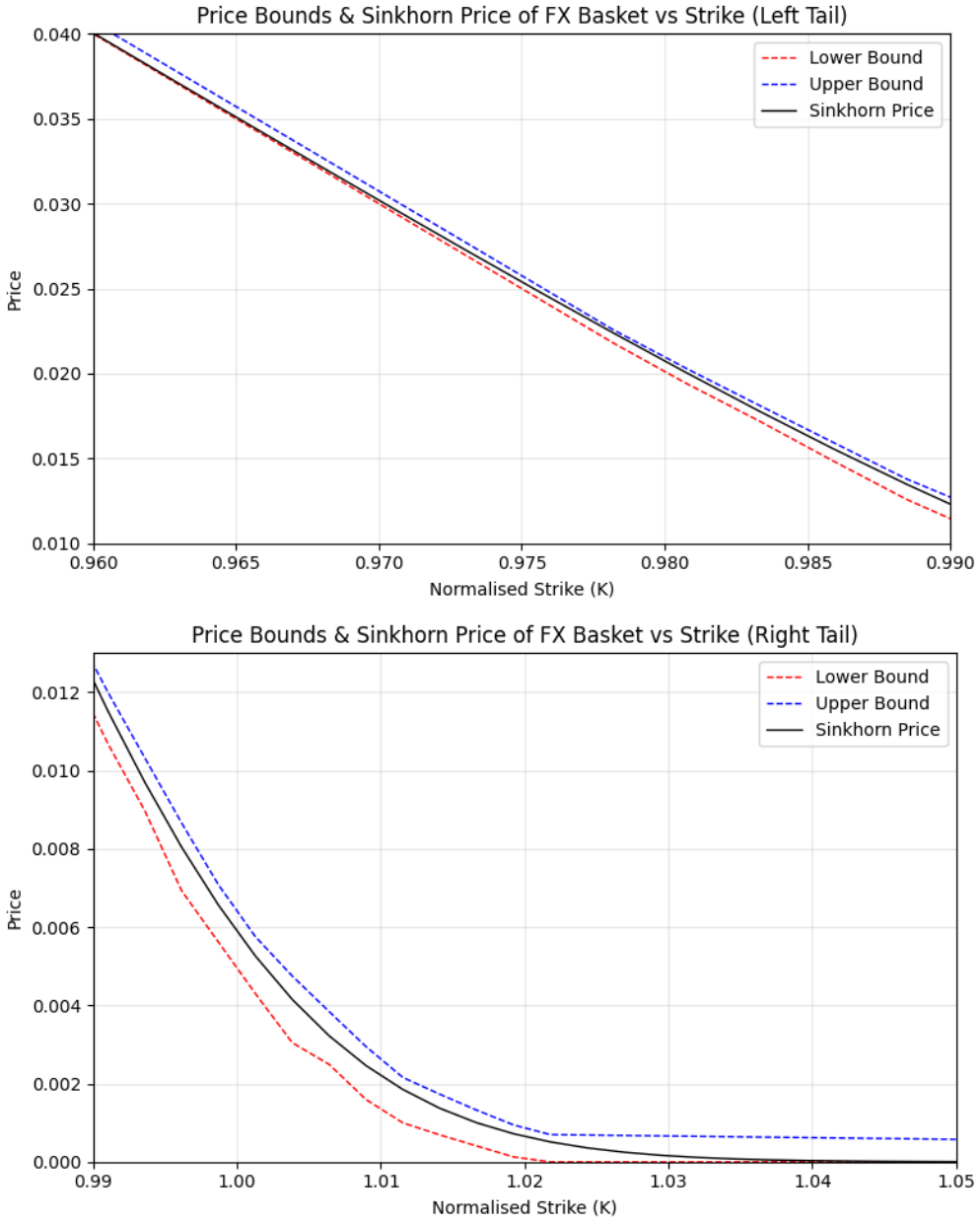

Figure 3.3: Plots of how model-free bounds of the left and right tail of the Basket Call option vary across strike.

These plots reveal another interesting qualitative feature of the Sinkhorn algorithm which is that on payoffs involving a form of strike, each tail of the price curve will tend towards an extremal solution (either the least or maximum admissible price depending on the specifics of the payoff).

3.3 Example 2: Price Bounds for forward-starting S&P 500 contracts

Thus far, we have looked at rates X and Y as two underlying assets evaluated at the same maturity T. The following example will allow for a more complete exploration of the theory of martingale optimal transport as we apply our framework to SPX evaluated at two distinct maturities. Given $0 < T_1 < T_2$, we denote $S_1 \triangleq SPX_{T_1}$ and $S_2 \triangleq SPX_{T_2}$. We use the SPX data in Table 3.2 to construct our example. We again apply framework §3.1 with call option data on constraints, but with an added martingale constraint. It should be noted that when working with realistic market data, S_t is not itself a martingale due to the effects of interest rates and dividends. The correct martingale constraint is on a driftless, discounted price process $\tilde{S}_t$, which in our discrete context is simply S_1 and S_2 normalised by the forward price at time zero for their respective maturities which we will denote by F_1 and F_2. The correct martingale constraint is then $E[\tilde{S}_2|S_1] = \tilde{S}_1$.

3.3.1 Applying the framework

PMF grid We will model the joint distribution of S_1 and S_2 on a square grid $\mathcal{D} = \{s_1^{(1)}, \ldots, s_N^{(1)}\} \times \{s_1^{(2)}, \ldots, s_N^{(2)}\}$. Though we shall proceed in absolute terms, the grid can also be constructed with forward normalisation, provided the constraints are formulated accordingly. Appropriate bounds for our grid can again be inferred from the Breeden-Litzenberger marginal distributions of S_1 and S_2. This method applied to the data in Table 3.2, shows that in both distributions virtually all probability mass is contained in an interval between 80% and 110% of their respective forward prices, so we proceed using these as the limit of our PMF grid. N can be chosen for computational feasibility, we have found $N = 100$ suitable for plotting purposes.

Strike	S_{T_1} **Implied Vol**	S_{T_2} **Implied Vol**	**Strike**	V_{T_1} **Implied Vol**
4500	0.308 340	0.256 853	0.1200	0.502 489
4550	0.295 476	0.247 760	0.1250	0.531 473
4600	0.283 464	0.239 149	0.1300	0.580 484
4650	0.271 111	0.230 164	0.1350	0.637 109
4700	0.258 451	0.221 471	0.1400	0.684 093
4750	0.245 508	0.213 216	0.1450	0.736 859
4800	0.233 694	0.204 941	0.1500	0.787 045
4850	0.221 358	0.196 783	0.1550	0.827 806
4900	0.209 098	0.188 797	0.1600	0.870 005
4950	0.197 249	0.180 978	0.1700	0.946 478
5000	0.186 334	0.173 596	0.1800	1.020 997
5050	0.174 676	0.166 311	0.1900	1.094 756
5100	0.164 666	0.159 104	0.2000	1.167 988
5150	0.154 888	0.152 338	0.2100	1.219 599
5200	0.145 824	0.145 637		
5250	0.137 304	0.139 202		
5300	0.129 232	0.132 974		
5350	0.121 549	0.127 012		
5400	0.114 418	0.121 383		
5450	0.108 020	0.116 176		
5500	0.102 798	0.111 556		
5550	0.098 878	0.108 040		
5600	0.095 900	0.105 025		
5650	0.093 499	0.102 493		
5700	0.091 856	0.100 738		
5750	0.091 887	0.099 360		
5800	0.093 406	0.098 370		

Table 3.2: Implied volatility smiles for SPX and VIX call options. SPX maturities are $T_1 = 20/251$ ($F_1 = 5489.83$) and $T_2 = 40/251$ (F_2= 5509.62). VIX maturity is $T_V = 20/251$ ($F_V = 0.1438$).

Calibration to Table 3.2 The subset of probability mass functions "calibrated" to the call option and martingale constraints is:

$$\left\{ \rho \in \Pi(\mathcal{D}) \;\middle|\; \begin{aligned} &\sum_{i,j} \max(s_i^{(1)} - K_1, 0)\, \rho_{ij} = C_1(K_1), && \forall K_1 \in \mathcal{K}_1, \\ &\sum_{i,j} \max(s_j^{(2)} - K_2, 0)\, \rho_{ij} = C_2(K_2), && \forall K_2 \in \mathcal{K}_2, \\ &\sum_{j} \left(\frac{s_j^{(2)}}{F_2} - \frac{s_i^{(1)}}{F_1} \right) \rho_{ij} = 0, && \forall i \end{aligned} \right\}$$

where $C_1(K_1)$ and $C_2(K_2)$ denote the observed market prices of European call options on S_1 and S_2, evaluated at strikes $K_1 \in \mathcal{K}_1$ and $K_2 \in \mathcal{K}_2$, respectively, with $\mathcal{K}_1$ and $\mathcal{K}_2$ the corresponding sets of traded strikes.

Python Implementation The admissible price bounds subject to these equality constraints are then found as the solution to the optimisation problem (3.5). See appendix A.1.5 for a Python implementation of the solution to this optimisation problem. The code produces price bounds, optimal hedging portfolios and plots of the distribution of S_2 conditioned on S_1 under the extremal probability mass functions. It is equally possible to constrain the LP to marginal distributions for S_1 and S_2 derived using SVI parametrisations and the Breeden-Litzenberger formula (see section 1.3). In practice using both marginals and the martingale condition can result in numerical feasibility issues, so it is sometimes necessary to allow a small amount of slackness in the martingale constraint. See Appendix A.1.7 for an example of a marginal constrained Linear Program with martingale conditions.

3.3.2 Forward-Starting Straddle Bounds

Following the example of Hobson et al. [23, 25], we begin by considering the price bounds of a straddle with payoff: $|S_2 - S_1|$. Using the code in Appendix A.1.5 the primal solution gives the range of admissible prices as approximately $[76.635, 193.07]$. This is clearly a relatively wide interval for practical purposes, indicating that the payoff is costly to super-hedge with the available traded assets. We examine some qualitative features of the extremal admissible models.

Sparsity in the extremal distributions Extracting the extremal PMF associated to the upper and lower bound price, shows that these edge cases demonstrate interesting qualitative features. Figure 3.4 shows that these PMFs are highly concentrated. They appear to show that $S_2|S_1$ follows a binomial or trinomial pattern in the extremal distributions. The structure of the straddle payoff, and the constraints we have applied, provide some intuitive justification for this. The straddle has its lowest value on the line $S_1 = S_2$, thus we seem the maximal price PMF concentrating mass as far away from this line as the option constraints will allow from this line in two cases : $S_1 > S_2$ and $S_2 > S_1$. The minimal price PMF clearly shows a preference for placing probability on the line $S_1 = S_2$, however it is not possible to place all probability on this line without violating the martingale condition, thus the line bifurcates in two places. In section 3.3.4 we make this argument more rigorous by considering an idealised version of the problem.

Interpretation of the Dual Variables The dual variables of this upper/lower bound solution represent a semi-static super/sub-hedging portfolio where the variables corresponding to traded call prices are static positions in those assets, and the variables corresponding to the martingale constraint provide a strategy specifying for any given observed value of S_{T_1} what position in the S_{T_2} forward contract is required to super/sub-hedge the payoff. Beiglböck et al. [1] show how how this structure generalises to a multi-period model with a larger number of maturities. Figure 3.5 shows the value of the martingale dual variables when solving for upper and lower bounds on the straddle payoff. This strategy has a fairly intuitive interpretation; the straddle has a high payoff when the values of S_1 and S_2 are far apart, therefore if S_1 is observed particularly high the strategy guarantees outperformance of the super-hedge by prescribing a short position in the futures of S_2, which will payoff if S_2 drifts away from S_1. On the other hand, the opposite situation where S_1 is low and S_2 is relatively high is super-hedged statically by short positions in S_1 call options and long positions in S_2 call options, as shown in figure 3.6. Figure 3.7 is a heat-map showing the extent to which the full Lagrangian for the dual formulation of the upper bound problem dominates the straddle payoff, the support points where the payoff coincides with the Lagrangian also serve to confirm the presence of duality in this linear program. We show illustrative cross-sections of this full Lagrangian in figure 3.8, to show how the Lagrangian rests on these binomial solutions. The substantial outperformance of the super-hedge explains why the spread between upper and lower bound is so high.

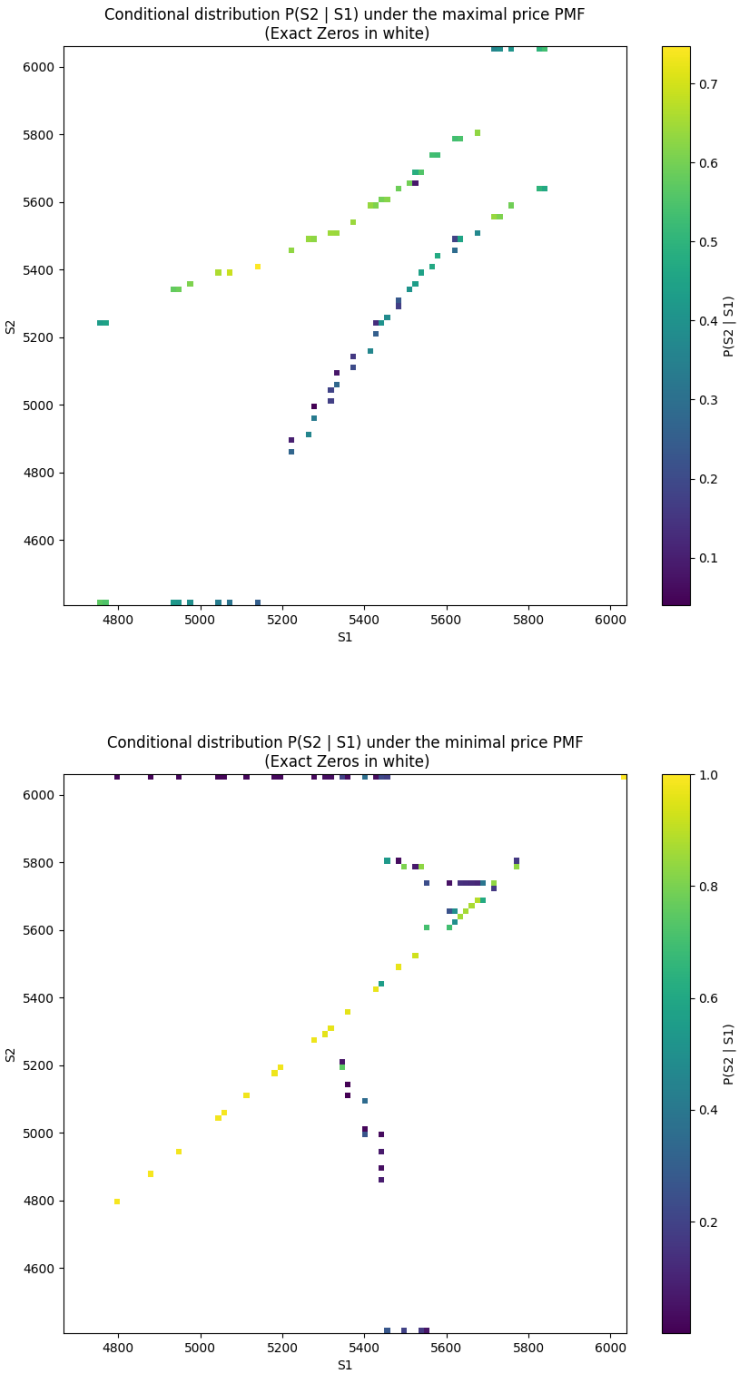

Figure 3.4: Extremal Probability Mass functions associated with the upper and lower solutions to vanilla price constrained forward straddle linear programming problem.

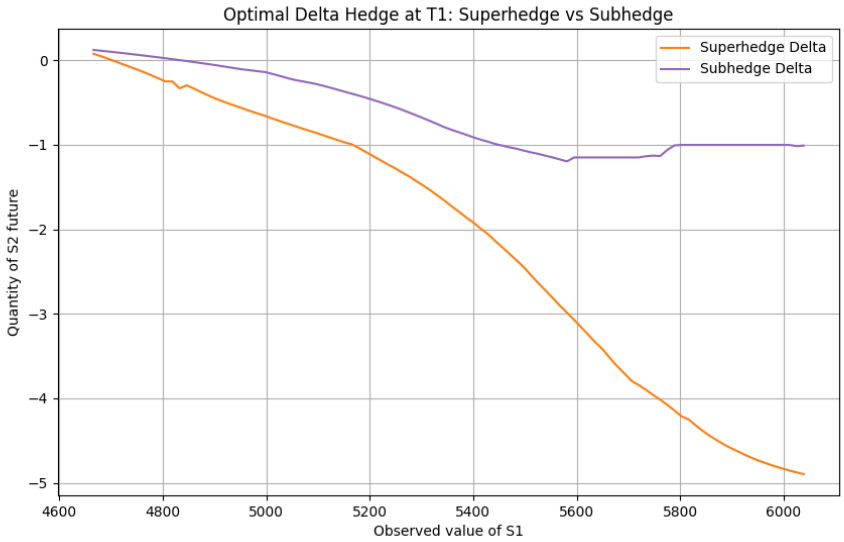

Figure 3.5: Plot of the martingale dual variables in the upper and lower bound solutions to the vanilla price constrained forward straddle linear programming problem.

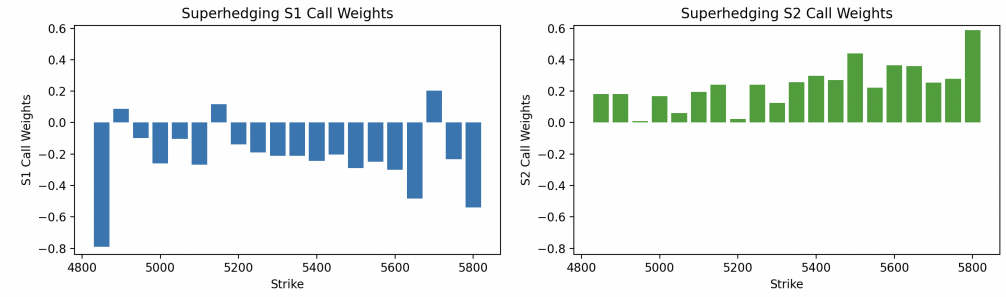

Figure 3.6: Plot of the call option dual variables in the upper bound solution to the vanilla price constrained forward straddle linear programming problem.

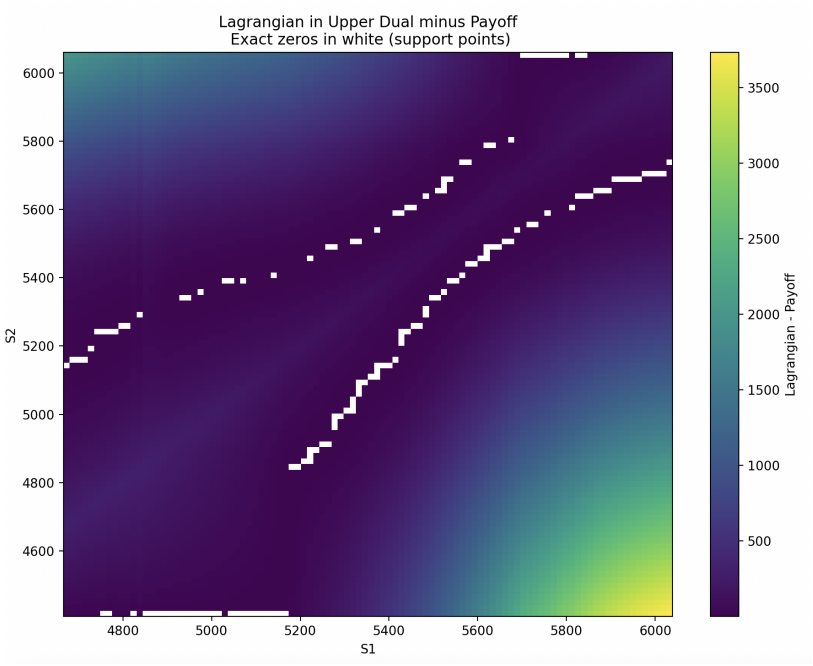

Figure 3.7: Plot of the upper dual problem's Lagrangian function for the Forward-Starting Straddle. The heat-map shows the difference between the Lagrangian function and the payoff of the Straddle with exact zeros in white to highlight the presence of strong duality.

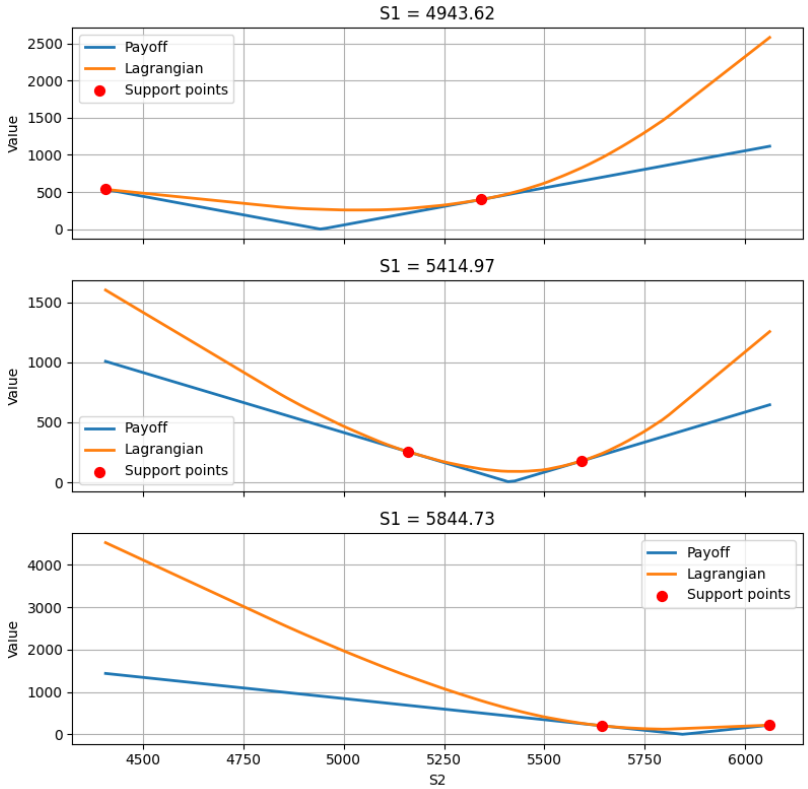

Figure 3.8: Three cross sections of the Lagrangian for the upper bound solution to the vanilla price constrained forward straddle linear programming problem.

3.3.3 Comparison with Marginally Constrained MOT

The extremal PMFs in figure 3.4 demonstrate similar qualitative behaviour to idealised versions of the price bounds problem studied by Hobson & Neuberger [25, Chapter 10], and Henry-Labordère & Touzi [21, Chapter 3]. Both papers enforce log-normal marginals to represent prices processes, and depict the sparsity of extremal coupling, with Hobson & Neuberger studying the straddle payoff $|S_2 - S_1|$, and Henry-Labordère and Touzi focusing primarily on the "Variance Swap" payoff defined as $\log^2(\frac{S_2}{S_1})$. We will demonstrate that by fitting an SVI parametrisation and Breeden-Litzenberger density to our SPX smile we can observe the same patterns in the extremal distributions as in the idealised log-normal case. Python scripts solving the price bounds problem using both the SPX inferred marginals, and the idealised log-normal marginals are available in Appendices A.1.6 and A.1.7.

Figure 3.9 shows the upper and lower extremal PMFs for the straddle, bearing similarities to both Hobson's plots and the call option calibrated linear program in figure 3.4. Figure 3.10 shows the dual functions of the optimal super-hedge and sub-hedge. Figure 3.11 shows the upper and lower extremal PMFs for the variance swap payoff, which matches the plots produced by Henry-Labordère and Touzi as expected. Figure 3.12 shows the dual functions of the optimal super-hedge and sub-hedge.

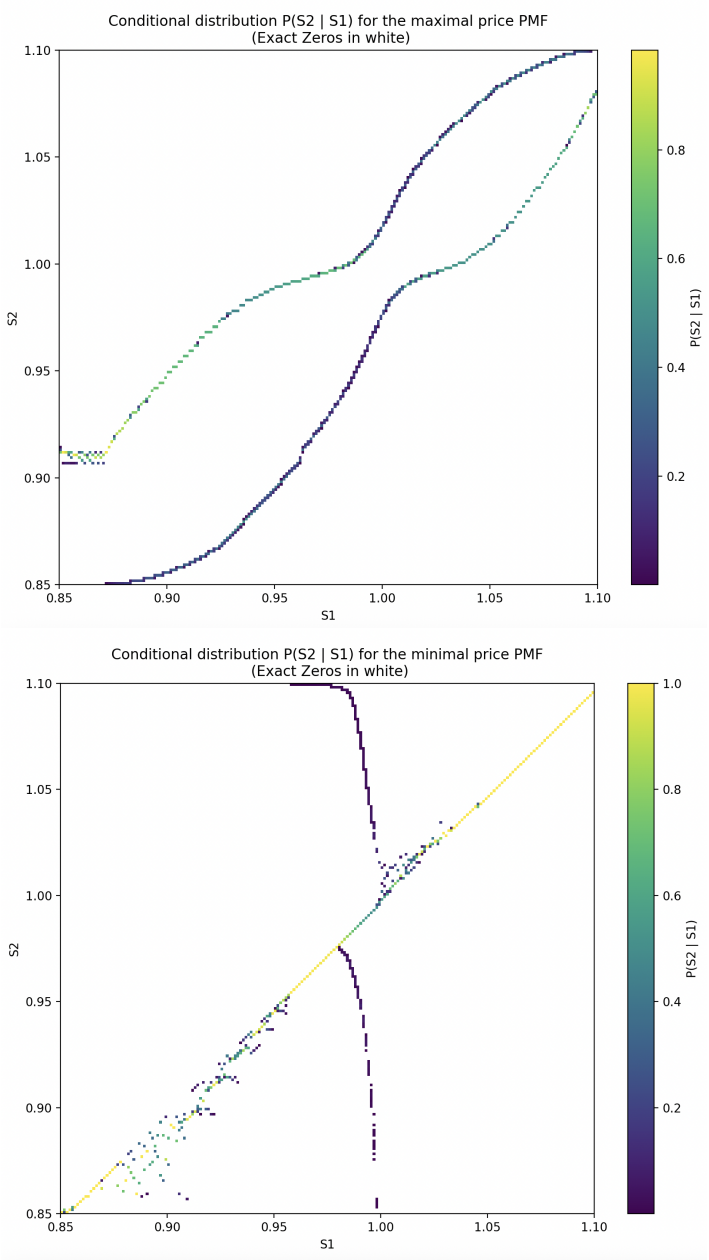

Figure 3.9: The upper and lower extremal PMFs for the straddle, found using a linear program constrained by SVI approximations of the marginal densities for S_1 and S_2.

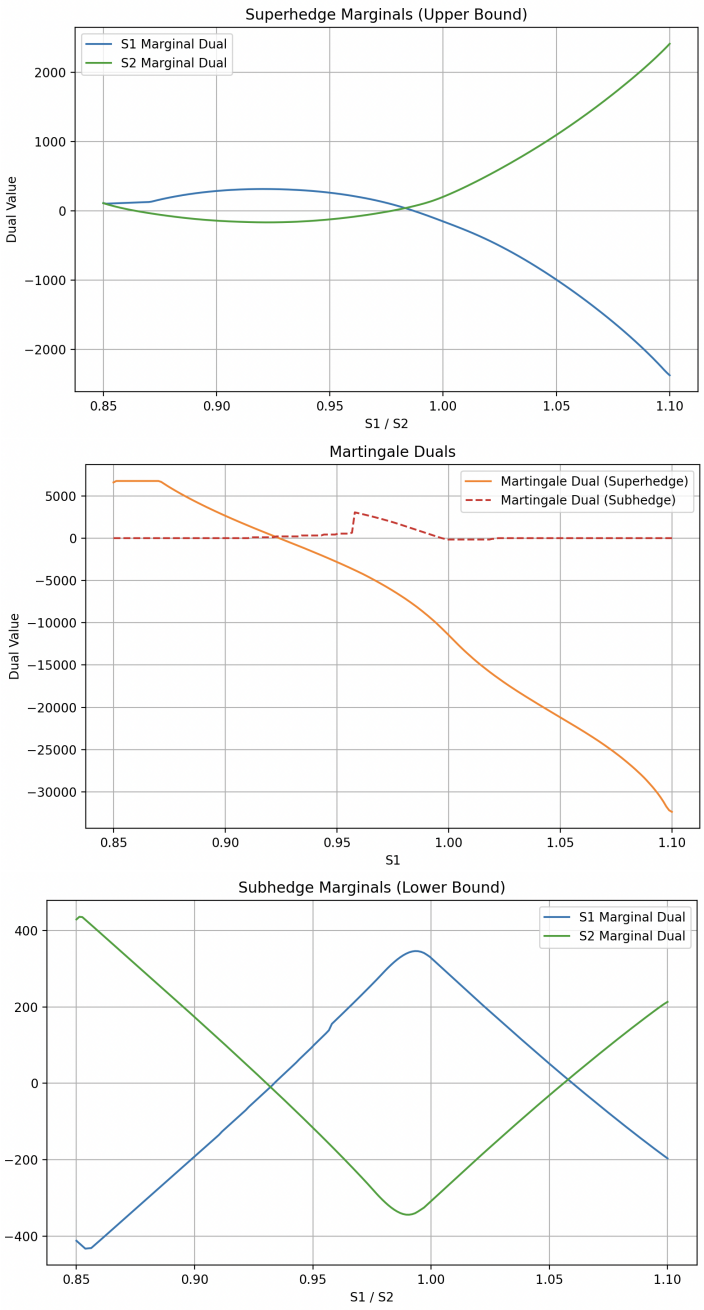

Figure 3.10: The dual functions corresponding to the upper and lower bounds of the straddle price, constrained by SVI approximations of the marginal densities for S_1 and S_2.

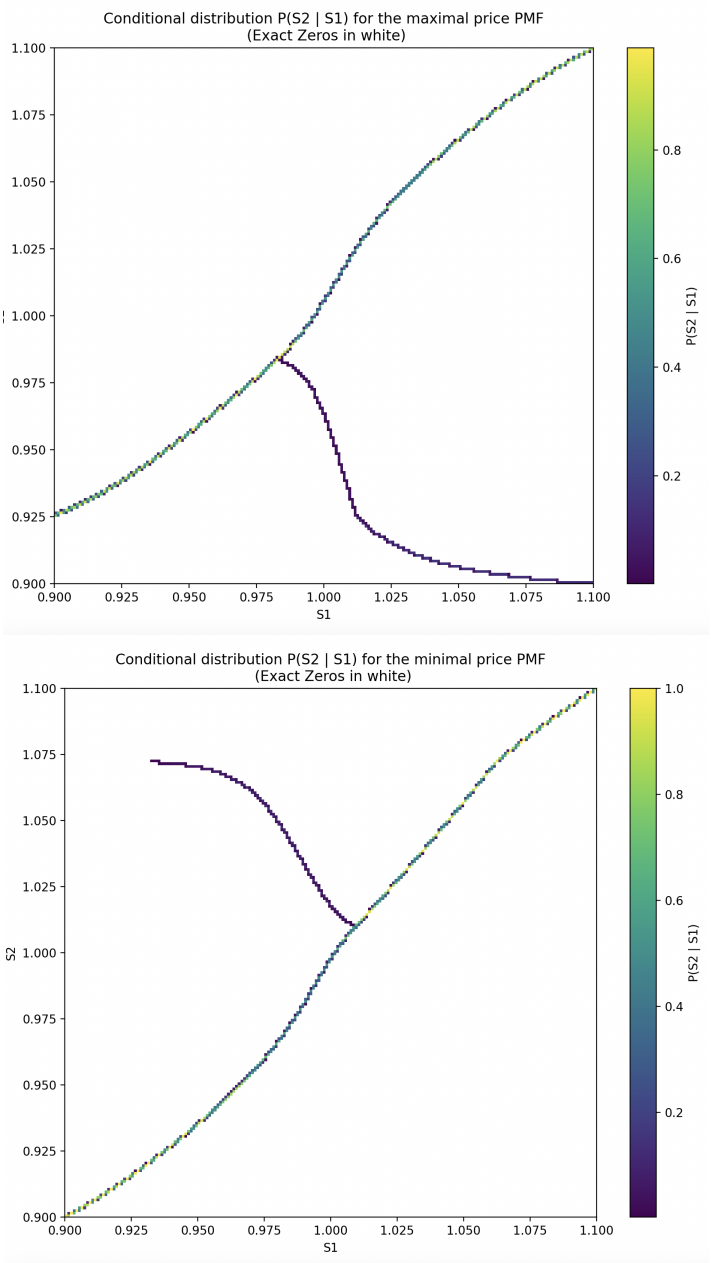

Figure 3.11: The upper and lower extremal PMFs for the variance swap payoff, found using a linear program constrained by SVI approximations of the marginal densities for S_1 and S_2.

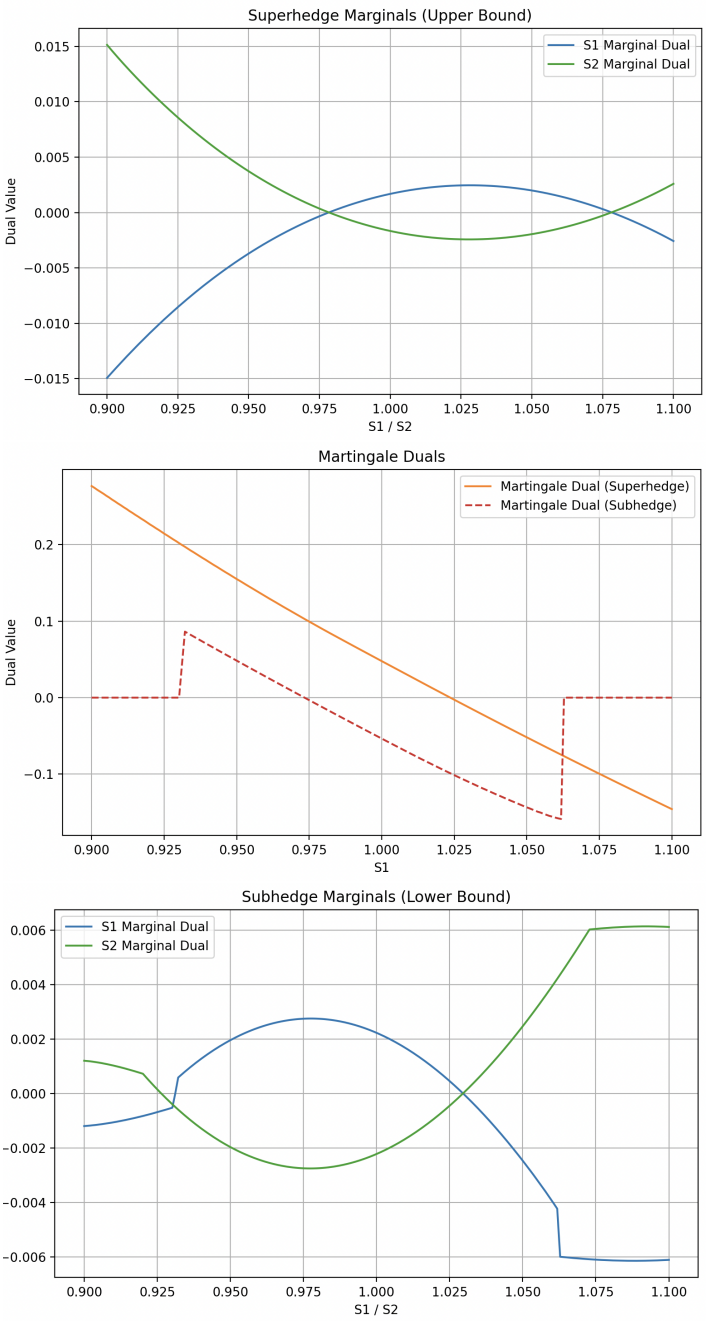

Figure 3.12: The dual functions corresponding to the upper and lower bounds of the variance swap price, constrained by SVI approximations of the marginal densities for S_1 and S_2.

3.3.4 Left-Curtain Couplings

The sparseness observed in the extremal densities of Martingale Optimal Transport problems can be explained rigorously in many cases by a concept called the Left-Curtain Coupling (LCC), whose existence and uniqueness was proved by Beiglbock and Juillet [2]. A formal construction was developed by Henry-Labordère and Touzi[21], who described the left-curtain coupling as a martingale constrained variant of a well known result in optimal transport called Brenier's Theorem [7] which posits the existence of unique, deterministic, optimal couplings for Kantorovich OT problems satisfying specific conditions on the cost function (see §2.1.5 of [20] for a detailed statement). In the context of the Martingale optimal transport problem (Section 1.1.2) we present below the key results from these papers, to show how a binomial extremal model can arise for payoffs satisfying certain conditions.

Definition 3.3.1 (Convex Order [2])**.** *Two measures μ and ν are in convex order, if they have finite expectation satisfying:*

$$\int \varphi(x)d\mu(x) \leq \int \varphi(x)d\nu(x)$$

for every convex function $\varphi : \mathbb{R} \to \mathbb{R}$. We denote this by $\mu \preceq \nu$.

Definition 3.3.2 (Left-Monotonicity [21])**.** *A coupling $\pi \in \Pi_M(\mu, \nu)$ is called **left-monotone** if there exists a Borel set $\Gamma \subset \mathbb{R} \times \mathbb{R}$ with $\pi(\Gamma) = 1$, such that for all $(x, y_1), (x, y_2), (x', y') \in \Gamma$ with $x < x'$, it must hold that $y' \notin (y_1, y_2)$.*

Theorem 3.3.3 (Existence and Uniqueness of LCC [2])**.** *If $\mu \preceq \nu$, then there exists a unique $\pi \in \Pi_M(\mu, \nu)$ that is left-monotone, which we call the "left-curtain coupling".*

Proposition 3.3.4 (Binomial Structure of LCC [2])**.** *If $\mu \preceq \nu$ and μ is continuous, then the left-curtain coupling π_{lc} is supported entirely on the graphs of two functions $T_d, T_u : \mathbb{R} \to \mathbb{R}$ satisfying:*

1. $\forall x,\ T_d(x) \leq x \leq T_u(x)$,
2. $\forall x < x',\ T_u(x) < T_u(x')$ *and* $T_d(x') \notin (T_d(x), T_u(x))$.

Proposition 3.3.5 (Optimality Condition for the LCC [21])**.** *If $\mu \preceq \nu$ and a cost function $c : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ such that $\partial_{xxy} c(x, y) > 0 \quad \forall (x, y)$, then the left-curtain coupling is the unique minimiser of the MOT problem:*

$$\inf_{\pi \in \Pi_M(\mu,\nu)} \mathbb{E}^{\pi}[c(X, Y)], \qquad \text{where } X \sim \mu, Y \sim \nu$$

The sufficient condition for optimality that $\partial_{xxy}c(x,y) > 0$ is known as the martingale Spence-Mirrlees condition. Specifically, proposition 3.3.5 explains the emergence of the binomial behaviour in the extremal distributions of the variance swap, since $\partial_{xxy}\log^2(\frac{y}{x}) = \frac{2}{x^2y} > 0$ for $x > 0$ and $y > 0$. This explains rigorously the bifurcation we observe in figure 3.11. Another example of a payoff matching the constraint would be the cubic payoff $(y-x)^3$.

3.3.5 Hobson-Klimmek-Neuberger Solutions

Henry-Labordère and Touzi remark, that the the straddle payoff $|y-x|$ does not satisfy the optimality conditions required in the LCC construction. The binomial structure of the upper extremal price distribution for this straddle payoff, was explicitly derived by Hobson and Neuberger [25], before the proposition of the left-curtain approach was invented. Hobson and Klimmek [23] later demonstrated that the lower extremal price distribution follows a trinomial distribution. Note that these rigorous results agree with the behaviour observed in figures 3.4 and 3.9. Below we provide a simplified version of Hobson and Neuberger's argument in chapter 5 of [25], which is adapted to prove the existence of the binomial structure of upper extremal distribution without explicitly deriving an expression for the probabilities.

Proof Sketch: Sparsity of the Upper Extremal Coupling for $|Y-X|$

The Martingale Optimal Transport problem for the upper price bound of the straddle is given by:

$$\sup_{\pi\in\Pi_M(\mu,\nu)} \mathbb{E}^\pi[|Y-X|]$$

where $\Pi_M(\mu,\nu)$ denotes the set of martingale couplings with marginals. The corresponding dual problem is:

$$\inf_{(\varphi,\psi,h)\in\Phi_c}\left\{\int\varphi(x)d\mu(x)+\int\psi(y)d\nu(y)\right\},$$

over the set triples (φ,ψ,h) where $\varphi\in L^1(\mu)$, $\psi\in L^1(\nu)$, and h is a bounded continuous function such that:

$$\varphi(x)+\psi(y)+h(x)(y-x)\geq|y-x|,\quad\forall(x,y).$$

Consider $L(x,y) = |y-x|-\psi(y)-\varphi(x)-h(x)(y-x) \leq 0$. Due to Kantorovich duality, we expect that an optimal super-hedge attains the

equality on the support of the optimal coupling π^*, i.e. we expect $L(x, y) = 0$. Hobson and Neuberger's key insight is that for a fixed x, $y \mapsto L(x, y)$ is a strictly convex mapping due its term in $|y - x|$. It is a fundamental geometric property of strictly convex functions from $\mathbb{R} \to \mathbb{R}$ that the preimage of any single point in their co-domain contains at most two unique point[1]. As a consequence, the optimal coupling's support on a given value of x can consist of two points at most i.e. the conditional distribution at that point has a binomial structure. □

3.4 Example 3: Joint calibration to VIX and SPX smiles

Our final example seeks to add a third dimension to the martingale optimal transport problem by considering also the CBOE VIX. The VIX is an index of volatility whose value is entirely determined by the observed market value of SPX options, but on which options have traded since 2006. VIX derivatives are typically used for hedging or speculating on volatility and market sentiment, and the demand for these instruments impacts the price of the SPX options that are needed to replicate them, which in turn feeds back into the calculation of the VIX. This close and complex relationship between the two markets, makes a jointly calibrated model extremely valuable for setting robust prices. A particular concern was that institutions trading in both SPX and VIX instruments could open themselves to arbitrage without understanding the joint dynamics at play. The SPX-VIX calibration problem proved intractable for many years, until its solution by Julien Guyon [18], who after a series of investigations into the difficulties of joint calibration using very flexible flexible rough volatility models with jumps [17], decided to eschew a model-based approach entirely. Guyon's solution involves construction of an appropriately constrained Sinkhorn scheme, which converged to an admissible price demonstrating definitively that a no-arbitrage pricing model exists. Below we show how the complementary linear programming problem approximates upper and lower bounds on volatility derivatives, making clear exactly where the "no-arbitrage region" for these assets lies. To this end, we will consider three-dimensional probability mass functions on the triple

[1] To see why this is true: imagine a function mapping three or more points on an x-axis to a single point on a y-axis. If convexity is enforced this function must be a flat straight line, therefore a strictly convex function can only map two points at most to the same location.

(S_1, V, S_2), where $S_1 \triangleq S_{T_1}$,$V \triangleq \text{VIX}_{T_1}$ and $S_2 \triangleq S_{T_2}$.

Definition 3.4.1 (VIX Contract)**.** *The theoretical value of the VIX index at time t is*

$$VIX_t = \sqrt{\frac{2}{\tau}\mathbb{E}\left(\log \frac{S_{t+\Delta}}{S_t}\middle|\mathcal{F}_t\right)}. \tag{3.9}$$

Typically $\tau = \frac{1}{12}$ corresponding to one month of the trading year. In our two step model we use $\tau = T_2 - T_1$.

Remark 3.4.2. *For simplicity of exposition we have made the assumption that our modelled VIX asset is evaluates at T_1. In reality the monthly maturities of VIX contracts and SPX contracts differ by a couple of days. Remark 10 of [18] explains in detail how this can be dealt with in full detail.*

Definition 3.4.3 (The VIX-SPX Dispersion Constraint)**.** *The definition of the VIX requires that any coupling of VIX to SPX should enforce the following constraint:*

$$\mathbb{E}\left[-\frac{2}{\tau}\log\left(\frac{\tilde{S}_2}{\tilde{S}_1}\right)\middle|S_1, V\right] = V^2 \tag{3.10}$$

Definition 3.4.4 (The VIX-SPX Martingale Constraint)**.** *Likewise, the appropriate Martingale constraint is now:*

$$\mathbb{E}\left[\tilde{S}_2\middle|S_1, V\right] = \tilde{S}_1 \tag{3.11}$$

3.4.1 Applying the framework

PMF grid We will model the triple (S_1, V, S_2) on cubic grid $\mathcal{D} = \{s_1^{(1)}, \ldots, s_N^{(1)}\} \times \{v_1, \ldots, v_N\} \times \{s_1^{(2)}, \ldots, s_N^{(2)}\}$. We shall proceed in absolute terms, while making sure that constraints (3.11) and (3.10) are enforced with the appropriate forward normalisation. We employ the same data and bounds for S_1 and S_2 as in the previous model. A Breeden-Litzenberger density for the VIX data in Table 3.2 suggests that the marginal VIX density is widely spread around its forward price a grid of 50-150% of the forward price, was required for feasibility of the linear program. Working in three dimensions, may require a reduction in N, for computational tractability demanding on available resources. At the cost of increased runtime we have proceed once again with $N = 100$, though informative results may be observed in with lower values of N.

Calibration to Table 3.2 The set of calibrated probability mass functions ρ on the discrete 3D grid $\mathcal{D} = \{(s_i^{(1)}, s_j^{(2)}, v_k)\}$ is defined as those that correctly price all traded vanilla call options, while remaining consistent with martingale and dispersion constraints:

$$\left\{ \rho \in \Pi(\mathcal{D}) \;\middle|\; \begin{array}{ll} \sum_{i,j,k} \max(s_i^{(1)} - K_1, 0)\, \rho_{ijk} = C_1(K_1), & \forall K_1 \in \mathcal{K}_1, \\ \sum_{i,j,k} \max(s_j^{(2)} - K_2, 0)\, \rho_{ijk} = C_2(K_2), & \forall K_2 \in \mathcal{K}_2, \\ \sum_{i,j,k} \max(v_k - K_V, 0)\, \rho_{ijk} = C_V(K_V), & \forall K_V \in \mathcal{K}_V, \\ \sum_{j} \left(\frac{s_j^{(2)}}{\mathrm{F}_{T_2}} - \frac{s_i^{(1)}}{\mathrm{F}_{T_1}} \right) \rho_{ijk} = 0, & \forall i, k, \\ \sum_{j} \left(-\frac{2}{\tau} \log \left(\frac{s_j^{(2)}}{s_i^{(1)}} \frac{\mathrm{F}_{T_1}}{\mathrm{F}_{T_2}} \right) - v_k^2 \right) \rho_{ijk} = 0, & \forall i, k \end{array} \right\}$$

where $C_1(K_1)$, $C_2(K_2)$, and $C_V(K_V)$ denote the observed market prices of European call options on S_1, S_2, and V_1 respectively. Their respective sets of traded strikes are noted $\mathcal{K}_1, \mathcal{K}_2, \mathcal{K}_V$.

Python Implementation The admissible price bounds subject to these equality constraints are then found as the solution to the optimisation problem (3.5). See Appendix A.1.8 for a Python implementation of the solution to this optimisation problem. The code produces price bounds, and visualisations of the extremal probability mass functions. Given the higher dimensionality of this problem it is extremely important to use a spare matrix format to build the constraints before passing them to a solver. Each constraint consists of a (mostly-sparse) three-dimensional tensor that when flattened and multiplied by the flattened variable ρ enforces that single constraint.

3.4.2 VIX-SPX Extremal Pricing Distributions

For comparison with the calibrations in (S_1, S_2), we look again at the PMF associated to the greatest admissible price for the straddle $|S_2 - S_1|$. Extracting the upper extremal PMF, figure 3.13 shows that the distribution remains sparse in three dimensions, with integration over the V axis revealing the familiar binomial relationship between S_1 and S_2 (figure 3.14). The code in

Appendix A.1.8 gives the range of admissible prices as $[78.184, 186.56]$ which is not a significant improvement from the interval given by the 2D density. However, the mere feasibility of this optimisation is proof that no-arbitrage models exist that can account for all the traded options prices, which is itself valuable information. For example, we can use our linear program to produce an upper and lower smile for vanilla calls on the VIX options that shows the size of the "arbitrage-free" region at that strike. In other words, we can plot price bounds on a VIX call option at any near-the-money strike which are jointly consistent with the observed SPX and VIX data. The result of this procedure is shown in figure 3.15. We can observe that these bounds are relatively tight and would allow a trader to confidently price VIX options without opening themselves to arbitrage in the SPX market.

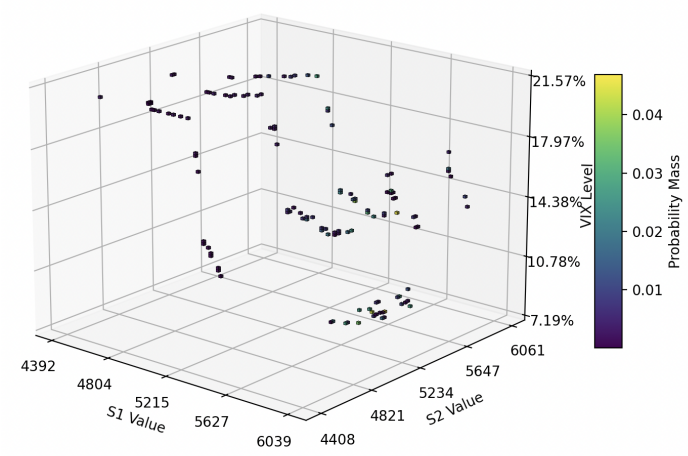

Figure 3.13: Three dimensional sparse PMF associated to the upper price bound of the forward straddle constrained to VIX and SPX option prices.

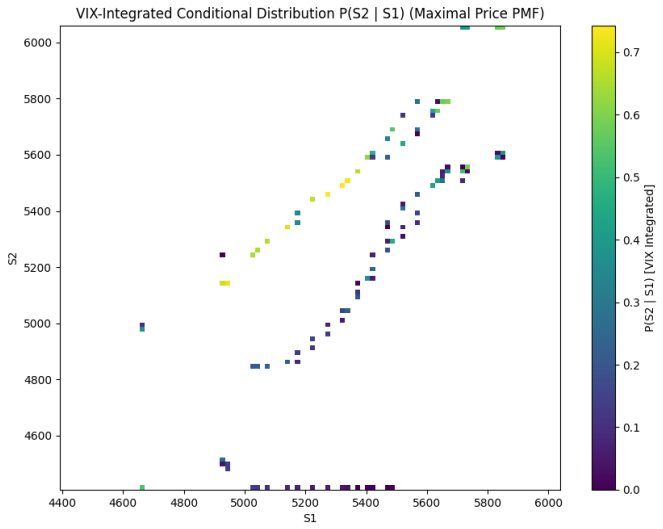

Figure 3.14: The result of integrating in the VIX dimension to recover the binomial solution for S_1 and S_2.

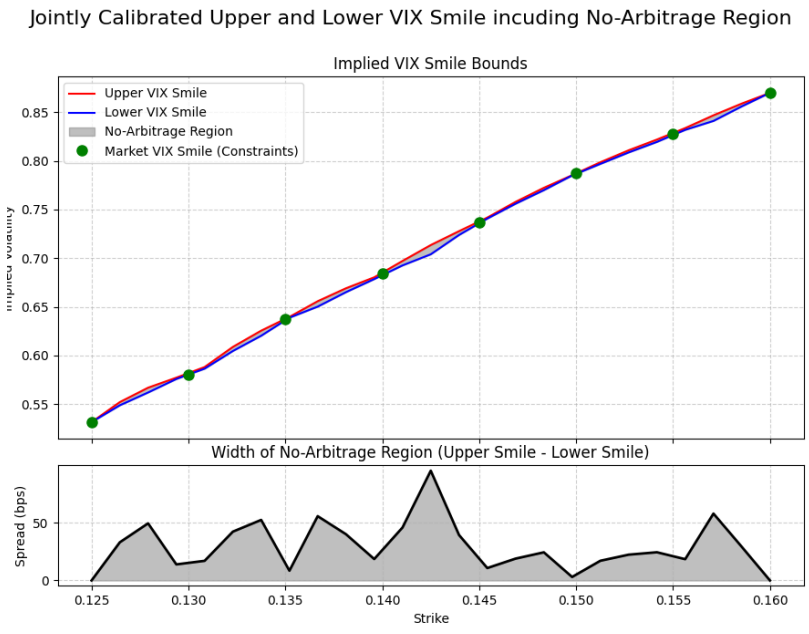

Figure 3.15: The upper and lower VIX smile jointly constrained to VIX and SPX call option prices.

Conclusion

This paper has demonstrated on two real financial datasets the utility of entropic and linear programming approaches to solving optimal transport problems, and made explicit how the results of these problems should inform risk management when trading in exotics on those underlying assets. Specifically, on the FX data we successfully applied Sinkhorn's algorithm to find a pricing density capable of fitting all observed vanilla prices. Applying linear programming showed that solution exists within a continuum of admissible densities which we were able to directly quantify.

Using the SPX and VIX data we not only found price bounds for the Straddle and Variance Swap payoffs, but successfully showed that the sparse solutions for MOT on log-normal marginals predicted by the Left-Curtain Coupling framework also occur when constraining prices to real market data. In the course of these computations we have seen that for practical purposes, constraining the linear program to preserve observed option prices alone is just as effective as enforcing marginals, preserving much of the underlying structure of the resultant densities, with less risk of extrapolation away from observed data points. The lower number of constraints makes the computation more efficient and scalable, and, in practice, provides far more actionable information about hedging portfolios. Following the example of Guyon, we had success using non-parametric approaches to locate the class of no-arbitrage couplings for the notoriously difficult VIX-SPX calibration, allowing us to draw the upper and lower admissible VIX smiles.

The numerical investigations we have conducted have many aspects which can be expanded upon. Our implementation of the Sinkhorn algorithm could be improved by considering more realistic reference models, such as the Gaussian copula. Notably, the martingale constrained linear programming problems we have tackled are only two-step models whereas the framework can in theory accommodate any number of time steps. Calibration to a larger number maturities on the SPX index for example would provide a natural

context for attempting to effectively bound prices on path-dependent payoffs such as American Puts [24], and also to investigate the sparsity of the extremal solutions that these higher-dimensional optimisations. It would also enable us to assess the rate at which the curse of dimensionality affect the linear programs for the problems. Another avenue of exploration would be consider how the linear program for exotic price bounds can be adapted to better reflect real market conditions, such as bid-ask spreads, transaction costs, finite liquidity, or enforcing that only whole numbers of contracts can be contained in the hedging portfolio. Equally, it would be valuable to see how price bounds for exotic options respond to the inclusion of publicly traded assets other than vanilla options in the dual portfolio. Finally, it is still an open question how a trader should make optimal decisions faced with signals from a combination of traditional models and model-free analyses.

The strength of the Optimal Transport approach to finance does not lie in its ability to consistently provide tight pricing bounds on exotic options. To the contrary, we have seen many examples of exotic options which have extremely wide price bounds despite calibration to the full volatility surface of the underlying assets. Our linear programming framework starkly shows that deciding on a point-wise price will almost always require making assumptions on the underlying dynamics of the assets. The true strength of the model-free approach is precisely that it acts to highlight exactly which modelling assumptions are not supported by the market data, allowing informed decision making, and that it provides explicit strategies for managing the risks associated with model selection. Moreover, as Guyon's work on the VIX-SPX calibration problem shows, whenever traditional models appear resistant to calibration practitioners should always consider whether a non-parametric approach could provide the flexibility they are missing.

Bibliography

[1] Mathias Beiglböck, Pierre Henry-Labordère, and Friedrich Penkner. Model-independent bounds for option prices: A mass transport approach, 2013.

[2] Mathias Beiglböck and Nicolas Juillet. On a problem of optimal transport under marginal martingale constraints. *The Annals of Probability*, 44(1), 2016.

[3] Jean-David Benamou and Yann Brenier. A computational fluid mechanics solution to the monge–kantorovich mass transfer problem. *Numerische Mathematik*, 84(3):375–393, 2000.

[4] D. Bertsimas and J.N. Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1997.

[5] Fischer Black. The pricing of commodity contracts. *Journal of Financial Economics*, 3(1):167–179, 1976.

[6] Douglas T. Breeden and Robert H. Litzenberger. Prices of state-contingent claims implicit in option prices. *Journal of Business*, 51(4):621–651, October 1978.

[7] Yann Brenier. Décomposition polaire et réarrangement monotone des champs de vecteurs. *Comptes Rendus de l'Académie des Sciences. Série I, Mathématique*, 305(19):805–808, 1987.

[8] Guillaume Carlier. On the linear convergence of the multimarginal sinkhorn algorithm. 32:786–794, 05 2022.

[9] Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transportation distances. *arXiv preprint arXiv:1306.0895*, 2013.

[10] Hadrien De March and Pierre Henry-Labordere. Building arbitrage-free implied volatility: Sinkhorn's algorithm and variants. January 31 2019.

[11] B. Dupire. *Pricing and hedging with smiles: proceedings of AFFI Conference, La Boule, June 1993*. 1993.

[12] Jim Gatheral. A parsimonious arbitrage-free implied volatility parameterization with application to the valuation of volatility derivatives. 2004.

[13] Jim Gatheral, Thibault Jaisson, and Mathieu Rosenbaum. Volatility is rough, 2014.

[14] Ivan Guo, Grégoire Loeper, Jan Obłój, and Shiyi Wang. Joint modelling and calibration of spx and vix by optimal transport. 2020.

[15] Ivan Guo, Gregoire Loeper, Jan Obloj, and Shiyi Wang. Optimal transport for model calibration, 2021.

[16] Ivan Guo, Gregoire Loeper, Jan Obloj, and Shiyi Wang. Optimal transport for model calibration, 2021.

[17] Julien Guyon. On the joint calibration of SPX and VIX options. Presentation at QuantMinds 2018, May 16 2018.

[18] Julien Guyon. The joint S&P 500/VIX smile calibration puzzle solved. *Risk*, April 2020.

[19] Julien Guyon and Jordan Lekeufack. Volatility is (mostly) path-dependent. *Quantitative Finance*, 23(9):1221–1258, 2023.

[20] Pierre Henry-Labordère. *Model-free Hedging: A Martingale Optimal Transport Viewpoint*. Chapman and Hall/CRC, 2017.

[21] Pierre Henry-Labordere and Nizar Touzi. An explicit martingale version of brenier's theorem, 2024.

[22] Steven L. Heston. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *The Review of Financial Studies*, 6(2):327–343, 1993.

[23] David Hobson and Martin Klimmek. Robust price bounds for the forward starting straddle, 2013.

[24] David Hobson and Dominykas Norgilas. Robust bounds for the american put, 2018.

[25] David E. Hobson and Anthony Neuberger. Robust bounds for forward start options. *Mathematical Finance*, 22(1):31–56, 2012.

[26] Hadrien De March. Entropic approximation for multi-dimensional martingale optimal transport, 2018.

[27] Robert C. Merton. Option pricing when underlying stock returns are discontinuous. *Journal of Financial Economics*, 3(1):125–144, 1976.

[28] Marcel Nutz. Introduction to entropic optimal transport, 2022. Lecture Notes.

[29] Gabriel Peyre and Marco Cuturi. Computational optimal transport. *Foundations and Trends in Machine Learning*, 11(5-6):355–607, 2019.

[30] Richard Sinkhorn. A relationship between arbitrary positive matrices and doubly stochastic matrices. *The Annals of Mathematical Statistics*, 35(2):876–879, 1964.

[31] Matthew Thorpe. Introduction to optimal transport. Lecture Notes, University of Cambridge, Part III Maths, 2018-2019, 2019.

[32] Cédric Villani. *Topics in Optimal Transportation*, volume 58 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 2003.

Appendix A

Python Implementation with Testing

As we remark in Section 2.3, the most important test that should be performed on the output of a non-parametric optimisation problem is that the resultant coupling correctly prices the assets in the constraining sets, and that other constraints such as martingale conditions are satisfied. It is also important to check that any densities are positive everywhere they are defined and that marginal densities sum to one.

A.1 Code

A.1.1 Black-Scholes Formula

```
"""Code for implementing conversion between implied vol
   and price"""
import numpy as np
from scipy.stats import norm
from scipy.optimize import root_scalar

"""LIBRARY FUNCTIONS"""
def black_scholes_call(F, K, T, sigma):
    """Black-Scholes price for a call option given
       forward F, strike K, expiry T, and implied vol
       sigma."""
```

```
    """F is forward price; K is the raw strike; T is
       maturity; sigma is the volatility"""
    d1 = (np.log(F / K) + 0.5 * sigma ** 2 * T) / (sigma
        * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    price = F * norm.cdf(d1) - K * norm.cdf(d2)
    return price

def implied_vol(F, K, T, price):
    """Computes the Black-Scholes implied volatility
       given the call price"""
    def obj(sigma):
        return black_scholes_call(F, K, T, sigma) -
           price
    result = root_scalar(obj, bracket=[-5, 5.0], method=
       'brentq')
    return result.root if result.converged else np.nan

"""LIBRARY TESTS"""
#Test that takes implied vol data calculates price, then
    inverts
def test_black_scholes_inversion():
    T=1/12
    X_strikes = np.array
       ([1.0681,1.0791,1.0904,1.1014,1.1119])
    X_market_vols = np.array
       ([0.0554,0.053115,0.0516,0.051435,0.0523])
    X_forw=1.0903

    Y_strikes = np.array
       ([1.2456,1.2595,1.274,1.2883,1.3017])
    Y_market_vols = np.array
       ([0.06055,0.058665,0.0573,0.057185,0.05765])
    Y_forw=1.2738

    Z_strikes = np.array
       ([0.84386,0.84969,0.85585,0.86234,0.86875])
    Z_market_vols = np.array
       ([0.03809,0.03725,0.037225,0.03825,0.03986])
    Z_forw=0.8559
    # EUR/USD
    for i in range(len(X_strikes)):
```

```
        price = black_scholes_call(X_forw, X_strikes[i],
            T, X_market_vols[i])
        iv = implied_vol(X_forw, X_strikes[i], T, price)
        assert abs(iv - X_market_vols[i]) < 1e-8, "EUR/
           USD failed"
    # GBP/USD
    for i in range(len(Y_strikes)):
        price = black_scholes_call(Y_forw, Y_strikes[i],
            T, Y_market_vols[i])
        iv = implied_vol(Y_forw, Y_strikes[i], T, price)
        assert abs(iv - Y_market_vols[i]) < 1e-8, "GBP/
           USD failed"
    # EUR/GBP
    for i in range(len(Z_strikes)):
        price = black_scholes_call(Z_forw, Z_strikes[i],
            T, Z_market_vols[i])
        iv = implied_vol(Z_forw, Z_strikes[i], T, price)
        assert abs(iv - Z_market_vols[i]) < 1e-8, "EUR/
           GBP failed"
```

A.1.2 SVI Parametrisation

```
"""Code for handling implied vol data, fitting SVI
   smiles, transforming the smiles into probability
   densities"""
import numpy as np
from scipy.optimize import minimize
import math

def svi_var(k_lognrm,params):
    """
    Implements Gatheral's SVI parameterisation of the
       volatility smile

    Inputs are:
    k: array of strikes (LOG-NORMALISED)
    and following params
    +++++++
    a: base volatility
    b: slope in wings
    sigma: curvature of minimum
    rho: skew
```

```
18      m: location of minimum volatility
19      """
20      a, b, sigma, rho, m = params[0], params[1], params
           [2], params[3], params[4]
21
22      #No arbitrage conditions on the parameters
23      rho=min(max(rho,-1.0),1.0)
24      b=min(b,2/(1+np.abs(rho)))
25      a=max(a,-b*sigma*np.sqrt(1-rho**2))
26
27      x=k_lognrm
28      return a + b * (rho * (x - m) + np.sqrt((x - m)**2 +
            sigma**2))
29
30  def svi_vol(k_lognrm,params,T):
31      """SVI parameterisation of the implied volatility
32      Ensure that strikes are log-normalised!
33      """
34      return np.sqrt(svi_var(k_lognrm,params)/T)
35
36  def svi_rss(k_lognrm,params,obs_v,T):
37      """A sum of squares loss function used to fit the
           SVI parameters in svi_fit"""
38      return np.sum((svi_vol(k_lognrm,params,T)-obs_v)**2)
39
40  def svi_fit(k_lognrm,obs_vars,bestp,T):
41      """fits an svi smile to implied volatility data"""
42      params_init = bestp  #best guess [a, b, sigma, rho,
           m]
43      def objective(params):
44          return svi_rss(k_lognrm,params,obs_vars,T)
45
46      result = minimize(fun=objective, x0=params_init,
           method='Nelder-Mead',
47          tol=1E-14,
48          options={"maxiter":2000}
49      )
50      return result
51
52  def SVIDensity(K_norm,params):
53      """Closed form expression for the density function
           associated to an SVI smile,
54          derived using Breeden-Litzenberger Rule
```

```
        NB: Strike input is to be normalised by the
           forward price but NOT in log-space
        """
    a=params[0];b=params[1];sigma=params[2];rho=params
       [3];m=params[4];
    k=np.log(K_norm)
    V=a + b*(rho*(k - m) + np.sqrt((k - m)**2 + sigma
       **2))
    V1=b*(rho + (k-m)/np.sqrt((m - k)**2 + sigma**2))
    V2=b*sigma**2/(m**2 - 2*m*k + k**2 + sigma**2)**1.5
    tmp=-np.exp(-(4*k**2 + V**2)/(8*V))
    tmp2=-4*k**2*V1**2 +4*V*V1*(4*k +V1) +  V**2*(V1**2
       -8*(2 + V2))
    tmp3=16*K_norm**1.5*np.sqrt(2*math.pi)*V**2.5
    return tmp*tmp2/tmp3
```

A.1.3 Sinkhorn Algorithm

```
"""Code for implementing the Sinkhorn Algorithm for
   optimal transport"""

from scipy.interpolate import CubicSpline
import numpy as np
from scipy.optimize import bisect
import src.svi_lib as svi_lib
"""Sinkhorn Spline Calibration function"""
def sinkhorn(sinkgrid,sinkweights,mu_X,mu_Y,mu_Z,
   iterations):
    """Input: Accepts a Gaussian Quadrature grid, the
       three density functions for calibration
        and the desired number of iterations
       OUTPUT: The three spline functions needed for the
           gibbs ansatz for FX density
        """
    u_x= np.zeros(len(sinkgrid))
    v_y= np.zeros(len(sinkgrid))
    w_z= np.zeros(len(sinkgrid))
    x_points=sinkgrid
    y_points=sinkgrid
    z_points = np.linspace(np.min(x_points)/np.max(
       y_points), np.max(x_points)/np.min(y_points), len
       (sinkgrid))
```

```
    for it in range(iterations):
        w_spline= CubicSpline(z_points, w_z)
        #u update
        for i in range(u_x.shape[0]):
            yw_z = y_points*w_spline(x_points[i]/
               y_points)
            intx=np.exp(np.clip(v_y+yw_z,-700,700))*mu_Y
               (y_points)
            u_x[i] = -np.log(np.sum(intx * sinkweights))
        #v update
        for j in range(v_y.shape[0]):
            yw_z = y_points[j]*w_spline(x_points/
               y_points[j])
            inty=np.exp(np.clip(u_x+yw_z,-700,700))*mu_X
               (x_points)
            v_y[j] = -np.log(np.sum(inty * sinkweights))
        #w update using bisection for w
        v_spline = CubicSpline(y_points, v_y)
        for k in range(len(z_points)):
            zk = z_points[k]
            def root_func(wz):
                x_over_z = x_points / zk
                exp_term = u_x + v_spline(x_over_z) +
                   x_over_z * wz
                exp_term_clipped = np.clip(exp_term,
                   -700, 700)
                intz = np.exp(exp_term_clipped) * (
                   x_points**2 / zk**3) * mu_X(x_points)
                    * mu_Y(x_points/zk)
                return np.sum(intz * sinkweights) - mu_Z
                   (zk)
            w_z[k] = bisect(root_func, -5000, 100,rtol=1
               e-4)
    return u_x, v_y, w_z

"""outputs sinkhorn PMF directly from market data"""
def sink_density(Xmesh, Ymesh,T,X_strikes_lognrm,
   X_market_vols,Y_strikes_lognrm,Y_market_vols,
   Z_strikes_lognrm,Z_market_vols):
    Xopt = svi_lib.svi_fit(X_strikes_lognrm,
       X_market_vols,[0.0001, 0.002, 0.0014, 0, 0.005],T
       )
```

```
    Yopt = svi_lib.svi_fit(Y_strikes_lognrm,
       Y_market_vols,[0.0001, 0.002, 0.0014, 0, 0.005],T
       )
    Zopt = svi_lib.svi_fit(Z_strikes_lognrm,
       Z_market_vols,[0.00001, 0.0005, 0.001, 0, 0.0],T)

    def mu_X(xvar):
        return svi_lib.SVIDensity(xvar,Xopt.x)
    def mu_Y(yvar):
        return svi_lib.SVIDensity(yvar,Yopt.x)
    def mu_Z(zvar):
        return svi_lib.SVIDensity(zvar,Zopt.x)
    def opt_mu(x,y, u_spline, v_spline, w_spline):
        return np.exp(u_spline(x)+v_spline(y)+y*w_spline
           (x/y))*mu_X(x)*mu_Y(y)

    #Quadrature for use in sinkhorn calibration
    sinkPoints = 400
    [u_s, w_s] = np.polynomial.legendre.leggauss(
       sinkPoints)
    a_s = 0.8
    b_s = 1.2
    sinkgrid = 0.5*(b_s-a_s)*u_s + 0.5*(a_s+b_s)
    sinkweights = 0.5*(b_s-a_s)*w_s
    z_points = np.linspace(np.min(sinkgrid)/np.max(
       sinkgrid), np.max(sinkgrid)/np.min(sinkgrid), len
       (sinkgrid))

    opt_u, opt_v, opt_w = sinkhorn(sinkgrid,sinkweights,
       mu_X,mu_Y,mu_Z,30)
    u_spline = CubicSpline(sinkgrid, opt_u)
    v_spline = CubicSpline(sinkgrid, opt_v)
    w_spline = CubicSpline(z_points, opt_w)
    return opt_mu(Xmesh, Ymesh, u_spline, v_spline,
       w_spline)
```

A.1.4 FX Price Bounds Linear Programming

```
"""Demonstration of LP bounds for the FX calibration
   problem"""
import src.ot_lib as ot_lib
import src.black_lib as bs_lib
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import linprog
from tabulate import tabulate

"""CALIBRATION DATA"""
T=1/12 #Maturity for FX data
X_strikes = np.array([1.0681,1.0791,1.0904,1.1014,1.1119]) #EUR/USD strikes
X_market_vols = np.array([0.0554,0.053115,0.0516,0.051435,0.0523]) #EUR/USD implied vol
X_forw=1.0903 #EUR/USD forward rate
X_norm=X_strikes/X_forw
X_strikes_lognrm = np.log(X_norm)
X_market_prices=np.zeros(len(X_strikes))

Y_strikes = np.array([1.2456,1.2595,1.274,1.2883,1.3017]) #GBP/USD strikes
Y_market_vols = np.array([0.06055,0.058665,0.0573,0.057185,0.05765]) #GBP/USD implied vol
Y_forw=1.2738 #GBP/USD forward rate
Y_norm=Y_strikes/Y_forw
Y_strikes_lognrm = np.log(Y_norm)
Y_market_prices=np.zeros(len(Y_strikes))

Z_strikes = np.array([0.84386,0.84969,0.85585,0.86234,0.86875]) #EUR/GBP strikes
Z_market_vols = np.array([0.03809,0.03725,0.037225,0.03825,0.03986]) #EUR/GBP implied vol
Z_forw=0.8559 #EUR/GBP forward rate
Z_norm=Z_strikes/Z_forw
Z_strikes_lognrm = np.log(Z_norm)
Z_market_prices=np.zeros(len(Z_strikes))

for i in range(len(X_strikes)): #Calculation of Market Prices from vol
```

```
    X_market_prices[i]=bs_lib.black_scholes_call(X_forw,
        X_strikes[i], T, X_market_vols[i])
    Y_market_prices[i]=bs_lib.black_scholes_call(Y_forw,
        Y_strikes[i], T, Y_market_vols[i])
    Z_market_prices[i]=bs_lib.black_scholes_call(Z_forw,
        Z_strikes[i], T, Z_market_vols[i])

"""PMF grid for LP"""
grid_size = 50
x_vals = np.linspace(0.8, 1.2, grid_size)
y_vals = np.linspace(0.8, 1.2, grid_size)
X, Y = np.meshgrid(x_vals, y_vals, indexing='ij')

"""CONSTRAINT MATRICES for Primal LP"""
#Normalisation constraint of PMF
A_norm=np.ones((1,grid_size**2))
b_norm=np.array([1.0])

#Flattened version of X call option price constraints
x_price = X_market_prices / X_forw
A_X = np.zeros((len(X_norm), grid_size**2))
for k, K_x in enumerate(X_norm):
    for i in range(grid_size):
        for j in range(grid_size):
            A_X[k, i * grid_size + j] = max(x_vals[i] -
               K_x, 0)

#Flattened version of Y call option price constraints
y_price = Y_market_prices / Y_forw
A_Y = np.zeros((len(Y_norm), grid_size**2))
for k, K_y in enumerate(Y_norm):
    for i in range(grid_size):
        for j in range(grid_size):
            A_Y[k, i * grid_size + j] = max(y_vals[j] -
               K_y, 0)

#Flattened version of Y call option price constraints
z_price = Z_market_prices / Z_forw
A_Z = np.zeros((len(Z_norm), grid_size**2))
for k, K_z in enumerate(Z_norm):
    for i in range(grid_size):
        for j in range(grid_size):
```

```
            A_Z[k, i * grid_size + j] = max(x_vals[i] -
                K_z * y_vals[j], 0)

#Flattened constraint that E[X]=1, E[Y]=1 (forward
    normalisation of X&Y)
row_fX = np.zeros(grid_size**2)
row_fY = np.zeros(grid_size**2)
for i in range(grid_size):
    for j in range(grid_size):
        idx = i * grid_size + j
        row_fX[idx] = x_vals[i]
        row_fY[idx] = y_vals[j]
A_forwards = []
b_forwards = []
#E[X]=1
A_forwards.append(row_fX)
b_forwards.append(1.0)
#E[Y]=1
A_forwards.append(row_fY)
b_forwards.append(1.0)

A_forwards = np.vstack(A_forwards)
b_forwards = np.array(b_forwards)

"""Stacked A_eq for primal problem"""
A_eq = np.vstack([A_X, A_Y, A_Z, A_norm, A_forwards])
b_eq = np.concatenate([x_price, y_price, z_price, b_norm
    , b_forwards])
bounds = [(0, None)] * (grid_size**2)

"""GENERIC PAYOFF FUNCTION - EDIT AS NEEDED"""
def payoff_gen(X,Y,K=None):
    #return (X-Y)**2 #Quadratic payoff
    #return np.maximum(np.maximum(X,Y)-K,0) #Best-Of
        Call payoff
    #return np.maximum(X/Y - K,0)  #Quanto call payoff
    #return np.maximum(0.5*(X+Y)-K,0) #basket option
        payoff
    #((X > K) & (Y > K)).astype(float) #Binary option
    return np.maximum(X/Y - K,0)

K=1
payoff_grid = payoff_gen(X, Y, K)
```

```
payoff_flat = payoff_grid.flatten()

"""Primal Problem Solver"""
#primal upper
res_min = linprog(c=payoff_flat, A_eq=A_eq, b_eq=b_eq,
    bounds=bounds, method="highs")
min_price = res_min.fun if res_min.success else np.nan

#primal lower
res_max = linprog(c=-payoff_flat, A_eq=A_eq, b_eq=b_eq,
    bounds=bounds, method="highs")
max_price = -res_max.fun if res_max.success else np.nan

"""Sinkhorn density (for comparison against LP bounds)
    """
#Quadrature for sinkhorn pricing
quadPoints = 400
[u, w] = np.polynomial.legendre.leggauss(quadPoints)
a = 0.9
b = 1.1
quadgrid = 0.5*(b-a)*u + 0.5*(a+b)
quadweights = 0.5*(b-a)*w
Xsink, Ysink = np.meshgrid(quadgrid, quadgrid)
#sinkhorn pricing density
sinkhorn_density = ot_lib.sink_density(Xsink,Ysink,T,
    X_strikes_lognrm,X_market_vols,Y_strikes_lognrm,
    Y_market_vols,Z_strikes_lognrm,Z_market_vols)
#sinkhorn price for comparison
weights_2d = quadweights[:, np.newaxis] * quadweights[np
    .newaxis, :]
sinkhorn_price = np.sum(payoff_gen(Xsink, Ysink,K) *
    sinkhorn_density * weights_2d)

"""Dual Problem Solver"""
# upper dual
res_dual_super = linprog(
    c=b_eq,
    A_ub=-A_eq.T,
    b_ub=-payoff_flat,
    bounds=[(None, None)] * A_eq.shape[0],
    method='highs'
)
hedge_price_super = res_dual_super.fun
```

```
weights_super = res_dual_super.x
#lower dual
res_dual_sub = linprog(
    c=-b_eq,
    A_ub=A_eq.T,
    b_ub=payoff_flat,
    bounds=[(None, None)] * A_eq.shape[0],
    method='highs'
)
hedge_price_sub = -res_dual_sub.fun
weights_sub = res_dual_sub.x

# duality gap
upper_duality_gap = max_price - hedge_price_super
lower_duality_gap = min_price-hedge_price_sub

"""OUTPUT 1 - The Price Bounds"""
table = [
    ["Upper Bound", max_price],
    ["Sinkhorn Price", sinkhorn_price],
    ["Lower Bound", min_price],
    ["Upper Duality Gap", upper_duality_gap],
    ["Lower Duality Gap", lower_duality_gap]
]
print(tabulate(table, headers=["Quantity", "Value"],
   floatfmt=".12f"))

"""OUTPUT 2 - The Superhedging Portfolio Composition"""
n_X = len(X_norm)
n_Y = len(Y_norm)
n_Z = len(Z_norm)

weights_X_super = weights_super[:n_X]
weights_Y_super = weights_super[n_X:n_X + n_Y]
weights_Z_super = weights_super[n_X + n_Y:n_X + n_Y +
   n_Z]
weight_fX_super = weights_super[-2]
weight_fY_super = weights_super[-1]

fig, axs = plt.subplots(1, 3, figsize=(15, 4), sharey=
   True)

#to scale y axis
```

```
max_weight = max(
    np.max(np.abs(weights_X_super)),
    np.max(np.abs(weights_Y_super)),
    np.max(np.abs(weights_Z_super))
)
# Plot X vanilla calls
axs[0].bar(np.arange(len(weights_X_super)),
   weights_X_super / max_weight, width=0.1)
axs[0].set_title(f"X vanilla calls\nForward weight: {
   weight_fX_super:.4f}")
axs[0].set_xticks(np.arange(len(X_norm)))
axs[0].set_xticklabels([f"{k:.3f}" for k in X_norm],
   rotation=45)
axs[0].set_xlabel("Strike")
axs[0].set_ylabel("Normalised Weight")

# Plot Y vanilla calls
axs[1].bar(np.arange(len(weights_Y_super)),
   weights_Y_super / max_weight, width=0.1, color='
   orange')
axs[1].set_title(f"Y vanilla calls\nForward weight: {
   weight_fY_super:.4f}")
axs[1].set_xticks(np.arange(len(Y_norm)))
axs[1].set_xticklabels([f"{k:.3f}" for k in Y_norm],
   rotation=45)
axs[1].set_xlabel("Strike")

# Plot Z vanilla calls
axs[2].bar(np.arange(len(weights_Z_super)),
   weights_Z_super / max_weight, width=0.1, color='green
   ')
axs[2].set_title("Z vanilla calls")
axs[2].set_xticks(np.arange(len(Z_norm)))
axs[2].set_xticklabels([f"{k:.3f}" for k in Z_norm],
   rotation=45)
axs[2].set_xlabel("Strike")

plt.tight_layout()
plt.show(block=True)

"""OUTPUT 3 - Heatmap of Performance of the Superhedge
   """
#payoff of the superhedge
```

```
superhedge_grid = np.zeros_like(payoff_grid)
for idx, K_x in enumerate(X_norm):
    superhedge_grid += weights_X_super[idx] * np.maximum
       (X - K_x, 0)

for idx, K_y in enumerate(Y_norm):
    superhedge_grid += weights_Y_super[idx] * np.maximum
       (Y - K_y, 0)

for idx, K_z in enumerate(Z_norm):
    superhedge_grid += weights_Z_super[idx] * np.maximum
       (X - K_z * Y, 0)

superhedge_grid += weight_fX_super * (X - 1.0)
superhedge_grid += weight_fY_super * (Y - 1.0)
diff_super = superhedge_grid - payoff_grid

fig, axs = plt.subplots(1, 2, figsize=(12, 5))
im0 = axs[0].imshow(payoff_grid, extent=[Y.min(), Y.max
   (), X.min(), X.max()],
                    origin='lower', aspect='equal', cmap
                       ='viridis')
axs[0].set_title("Payoff function")
axs[0].set_xlabel("Y")
axs[0].set_ylabel("X")
fig.colorbar(im0, ax=axs[0])
im1 = axs[1].imshow(diff_super, extent=[Y.min(), Y.max()
   , X.min(), X.max()],
                    origin='lower', aspect='equal', cmap
                       ='coolwarm')
axs[1].set_title("Superhedge minus Payoff")
axs[1].set_xlabel("Y")
axs[1].set_ylabel("X")
fig.colorbar(im1, ax=axs[1])
plt.tight_layout()
plt.show()
```

A.1.5 SPX Price Bounds Linear Programming

```
"""Implementation of price bounds with a martingale
   constrained asset (using option price constraints)"""
import src.black_lib as bs_lib
```

```
import numpy as np
import mosek.fusion as mf
from scipy.sparse import coo_matrix
import matplotlib.pyplot as plt
import numpy.ma as ma

T1=20/251
S1_forw=5489.83
S1_strikes=np.linspace(4500,5800,27)
S1_market_vol=np.array([0.30833965626324,
   0.295476337981542, 0.283463677875459,
   0.271111343370089,
                        0.258451404309255,
                           0.245507644865208,
                           0.233694271876973,
                           0.221358004433688,
                        0.209097873714426,
                           0.197249412151294,
                           0.186334324110301,
                           0.174675684455109,
                        0.164666282229566,
                           0.15488794443707,
                           0.145823620475546,
                           0.137304198281544,
                        0.129231575271196,
                           0.121548584918863,
                           0.114417742487412,
                           0.108019587373161,
                        0.102797547251224,
                           0.0988783071539835,
                           0.0959001836718852,
                           0.0934990753558453,
                         0.0918556359282457,
                            0.0918871975138243,
                            0.0934062296063838])

T2=40/251
S2_forw=5509.62
S2_strikes=np.linspace(4500,5800,27)
S2_market_vol=np.array([0.256853, 0.247760, 0.239149,
   0.230164,
                        0.221471, 0.213216, 0.204941,
                           0.196783,
```

```
                          0.188797, 0.180978, 0.173596,
                             0.166311,
                          0.159104, 0.152338, 0.145637,
                             0.139202,
                          0.132974, 0.127012, 0.121383,
                             0.116176,
                          0.111556, 0.108040, 0.105025,
                             0.102493,
                          0.100738, 0.099360, 0.098370])
"""Choosing a subset for calibration"""
S1_strikes = S1_strikes[7:]
S1_market_vol = S1_market_vol[7:]
S2_strikes = S2_strikes[7:]
S2_market_vol = S2_market_vol[7:]

#Call price calculation
C1 = np.array([bs_lib.black_scholes_call(S1_forw, K, T1,
    vol) for K,vol in zip(S1_strikes, S1_market_vol)])
C2 = np.array([bs_lib.black_scholes_call(S2_forw, K, T2,
    vol) for K,vol in zip(S2_strikes, S2_market_vol)])

#grid for the PMFs
grid_size = 100
s1_vals = np.linspace(0.85*S1_forw, 1.1*S1_forw,
   grid_size)
s2_vals = np.linspace(0.8*S2_forw, 1.1*S2_forw,
   grid_size)
nvar = grid_size**2
#Sparse matrix construction arrays
rows, cols, data = [], [], []
b_eq = []
rowcount = 0

# S1 call constraints
for K1, C in zip(S1_strikes, C1):
    for i, s1 in enumerate(s1_vals):
        payoff_s1 = max(s1 - K1, 0)
        if payoff_s1 > 0:
            for j in range(grid_size):
                idx = i*grid_size + j
                rows.append(rowcount)
                cols.append(idx)
                data.append(payoff_s1)
```

```
    b_eq.append(C)
    rowcount += 1

# S2 call constraints
for K2, C in zip(S2_strikes, C2):
    for i in range(grid_size):
        for j, s2 in enumerate(s2_vals):
            payoff_s2 = max(s2 - K2, 0)
            if payoff_s2 > 0:
                idx = i*grid_size + j
                rows.append(rowcount)
                cols.append(idx)
                data.append(payoff_s2)
    b_eq.append(C)
    rowcount += 1

# Martingale constraints
for i, s1 in enumerate(s1_vals):
    for j, s2 in enumerate(s2_vals):
        idx = i*grid_size + j
        rows.append(rowcount)
        cols.append(idx)
        #Forward ratio accounts for the discounting in
           the E value
        data.append(s2 - s1*(S2_forw / S1_forw))
    b_eq.append(0.0)
    rowcount += 1

# Normalisation constraint for PMF
for idx in range(nvar):
    rows.append(rowcount)
    cols.append(idx)
    data.append(1.0)
b_eq.append(1.0)
rowcount += 1
# Sparse constraint matrix put together for Mosek solver
A_sparse = coo_matrix((data, (rows, cols)), shape=(
   rowcount, nvar))
b_eq = np.array(b_eq)
A_mosek = mf.Matrix.sparse(
    A_sparse.shape[0],
    A_sparse.shape[1],
    A_sparse.row,
```

```
    A_sparse.col,
    A_sparse.data
)
#payoff grid for objective function (here a straddle)
S1_grid, S2_grid = np.meshgrid(s1_vals, s2_vals,
   indexing="ij")
payoff_flat = np.abs(S2_grid-S1_grid).flatten()#(np.log(
   S2_grid / S1_grid) ** 2).flatten()

#Minimise over admissible prices (dual and primal
   simultaneously with interior point method)
M_min = mf.Model("min_price")
x_min = M_min.variable("x", nvar, mf.Domain.greaterThan
   (0.0))
M_min.constraint("eqs", mf.Expr.mul(A_mosek, x_min), mf.
   Domain.equalsTo(b_eq))
M_min.objective("obj", mf.ObjectiveSense.Minimize, mf.
   Expr.dot(payoff_flat, x_min))
M_min.solve()
min_price = M_min.primalObjValue()

#Max over admissible prices
M_max = mf.Model("max_price")
x_max = M_max.variable("x", nvar, mf.Domain.greaterThan
   (0.0))
M_max.constraint("eqs", mf.Expr.mul(A_mosek, x_max), mf.
   Domain.equalsTo(b_eq))
M_max.objective("obj", mf.ObjectiveSense.Maximize, mf.
   Expr.dot(payoff_flat, x_max))
M_max.solve()
max_price = M_max.primalObjValue()

"""OUTPUT 1 - Price bounds"""
print("Min price:", min_price)
print("Max price:", max_price)

#extremal probability distributions
pmf_opt = np.array(x_max.level()).reshape((grid_size,
   grid_size))
pmf_min = np.array(x_min.level()).reshape((grid_size,
   grid_size))
pmf_opt = np.clip(pmf_opt, 0, None)
pmf_min = np.clip(pmf_min, 0, None)
```

```
133
134 #calculation of conditional probabilities
135 tol = 1e-12 #(avoids division by near zero vals)
136 row_sums_opt = pmf_opt.sum(axis=1, keepdims=True)
137 row_sums_min = pmf_min.sum(axis=1, keepdims=True)
138 pmf_opt_cond = np.divide(pmf_opt, row_sums_opt, out=np.
    zeros_like(pmf_opt), where=row_sums_opt > tol)
139 pmf_min_cond = np.divide(pmf_min, row_sums_min, out=np.
    zeros_like(pmf_min), where=row_sums_min > tol)
140
141 #dual variables for the upper bound
142 dual_vars = M_max.getConstraint("eqs").dual().flatten()
143 #duals for martingale constraints in maximal model
144 s1_call_duals = dual_vars[:len(C1)]
145 s2_call_duals = dual_vars[len(C1):len(C1) + len(C2)]
146 mart_duals = dual_vars[len(C1)+len(C2):len(C1)+len(C2)+
    grid_size]
147 norm_dual = dual_vars[-1]
148 #dual variables for the lower bound
149 dual_vars_min = M_min.getConstraint("eqs").dual().
    flatten()
150 #duals for the martingale constraints in minimal model
151 mart_duals_min = dual_vars_min[len(C1)+len(C2):len(C1)+
    len(C2)+grid_size]
152
153 """OUTPUT 2 - Heatmaps of the conditional probabilities
    showing sparseness"""
154 #heatmap of maximal solution
155 plt.figure(figsize=(10, 8))
156 masked = ma.masked_where(pmf_opt_cond.T <=1e-12,
    pmf_opt_cond.T)
157 cmap = plt.cm.viridis.copy()
158 cmap.set_bad(color='white')
159 plt.imshow(masked,aspect='auto',origin='lower',extent=[
    s1_vals[0], s1_vals[-1], s2_vals[0], s2_vals[-1]],
    cmap=cmap)
160 plt.colorbar(label="P(S2 | S1)")
161 plt.xlabel("S1")
162 plt.ylabel("S2")
163 plt.title("Conditional distribution P(S2 | S1) for the
    maximal price PMF\n (Exact Zeros in white)")
164 plt.show()
165
```

```
# Heatmap for the minimal solution
plt.figure(figsize=(10, 8))
masked_min = ma.masked_where(pmf_min_cond.T <= 1e-12,
    pmf_min_cond.T)
cmap_min = plt.cm.viridis.copy()
cmap_min.set_bad(color='white')  # Masked zero values
plt.imshow(masked_min,aspect='auto',origin='lower',
    extent=[s1_vals[0], s1_vals[-1], s2_vals[0], s2_vals
    [-1]],cmap=cmap_min)
plt.colorbar(label="P(S2 | S1)")
plt.xlabel("S1")
plt.ylabel("S2")
plt.title("Conditional distribution P(S2 | S1) for the
    minimal price PMF\n (Exact Zeros in white)")
plt.show()

"""OUTPUT 3 - Dual variables in the call options"""
fig1, axs1 = plt.subplots(2, 2, figsize=(12, 8), sharex=
    False)

# Top row: Superhedge
# S1 call weights (superhedge)
axs1[0, 0].bar(S1_strikes, dual_vars[:len(C1)], width=(
    S1_strikes[1]-S1_strikes[0])*0.8, color='tab:blue')
axs1[0, 0].set_xlabel("Strike")
axs1[0, 0].set_ylabel("S1 Call Weights")
axs1[0, 0].set_title("Superhedging S1 Call Weights")

# S2 call weights (superhedge)
axs1[0, 1].bar(S2_strikes, dual_vars[len(C1):len(C1)+len
    (C2)], width=(S2_strikes[1]-S2_strikes[0])*0.8, color
    ='tab:green')
axs1[0, 1].set_xlabel("Strike")
axs1[0, 1].set_ylabel("S2 Call Weights")
axs1[0, 1].set_title("Superhedging S2 Call Weights")

# Bottom row: Subhedge
# S1 call weights (subhedge)
axs1[1, 0].bar(S1_strikes, dual_vars_min[:len(C1)],
    width=(S1_strikes[1]-S1_strikes[0])*0.8, color='tab:
    blue')
axs1[1, 0].set_xlabel("Strike")
axs1[1, 0].set_ylabel("S1 Call Weights")
```

```
199 axs1[1, 0].set_title("Subhedging S1 Call Weights")
200
201 # S2 call weights (subhedge)
202 axs1[1, 1].bar(S2_strikes, dual_vars_min[len(C1):len(C1)
    +len(C2)], width=(S2_strikes[1]-S2_strikes[0])*0.8,
    color='tab:green')
203 axs1[1, 1].set_xlabel("Strike")
204 axs1[1, 1].set_ylabel("S2 Call Weights")
205 axs1[1, 1].set_title("Subhedging S2 Call Weights")
206
207 # Consistent y scaling for each row
208 for row in [0, 1]:
209     ymin = min(axs1[row, 0].get_ylim()[0], axs1[row, 1].
        get_ylim()[0])
210     ymax = max(axs1[row, 0].get_ylim()[1], axs1[row, 1].
        get_ylim()[1])
211     axs1[row, 0].set_ylim(ymin, ymax)
212     axs1[row, 1].set_ylim(ymin, ymax)
213
214 fig1.suptitle("Call Option Weights: Superhedging (Top)
    vs Subhedging (Bottom)", fontsize=16)
215 fig1.tight_layout(rect=[0, 0.03, 1, 0.97])
216 plt.show()
217
218 """OUTPUT 4 - Dual Variables for the Delta Hedging
    Strategy"""
219 plt.figure(figsize=(10, 6))
220 plt.plot(s1_vals, mart_duals, '-', color='tab:orange',
    label='Superhedge Delta')
221 plt.plot(s1_vals, mart_duals_min, '-', color='tab:purple
    ', label='Subhedge Delta')
222 plt.xlabel("Observed value of S1")
223 plt.ylabel("Quantity of S2 future")
224 plt.title("Optimal Delta Hedge at T1: Superhedge vs
    Subhedge")
225 plt.legend()
226 plt.grid(True)
227 plt.show()
228
229 """OUTPUT 5 - Plot of the full Lagrangian for upper
    bound showing duality via support points"""
230 # First compute Lagrangian for all S1, S2 pairs
231 lagrangian_3d = np.zeros((len(s1_vals), len(s2_vals)))
```

```
232
233 for i, s1 in enumerate(s1_vals):
234     # Constant part from S1 calls
235     s1_constant = 0
236     for k, K in enumerate(S1_strikes):
237         s1_constant += s1_call_duals[k] * max(s1 - K, 0)
238
239     # S2 call payoffs
240     s2_payoffs = np.zeros_like(s2_vals)
241     for k, K in enumerate(S2_strikes):
242         s2_payoffs += s2_call_duals[k] * np.maximum(
            s2_vals - K, 0)
243
244     # Forward payoff
245     forward_payoff = mart_duals[i] * (s2_vals - s1 * (
        S2_forw / S1_forw))
246
247     # Total Lagrangian for this S1
248     lagrangian_3d[i, :] = s1_constant + s2_payoffs +
        forward_payoff + norm_dual
249
250 payoff_3d = payoff_flat.reshape((len(s1_vals), len(
    s2_vals)))
251 difference_upper = lagrangian_3d - payoff_3d
252
253 # heatmap
254 significant_diff_upper = np.where(np.abs(
    difference_upper) >= 1e-6, difference_upper, np.nan)
255 original_cmap = plt.cm.viridis.copy()
256 original_cmap.set_bad(color='white')
257 plt.figure(figsize=(12, 8))
258 im = plt.imshow(significant_diff_upper.T, aspect='auto',
    origin='lower',
259                 extent=[s1_vals[0], s1_vals[-1], s2_vals
                        [0], s2_vals[-1]],
260                 cmap=original_cmap)
261 plt.colorbar(im, label='Lagrangian - Payoff')
262 plt.xlabel('S1')
263 plt.ylabel('S2')
264 plt.title('Lagrangian in Upper Dual minus Payoff\n Exact
    zeros in white (support points)')
265 plt.show()
266
```

```
"""OUTPUT 6 - Cross-sections of the Lagrangian for Superhedging"""
# select out some cross sections from the heatmap
s1_indices = [20,54,85] #Good for VolSwap payoff -> [29,65,85]
fig, axes = plt.subplots(len(s1_indices), 1, figsize=(10, 8), sharex=True)
for idx, ax in zip(s1_indices, axes):
    s1 = s1_vals[idx]
    lagrangian = lagrangian_3d[idx, :]
    payoff = payoff_3d[idx, :]
    ax.plot(s2_vals, payoff, label="Payoff", linewidth=2)
    ax.plot(s2_vals, lagrangian, label="Lagrangian", linewidth=2)
    cond = pmf_opt[idx, :]
    s2_support = s2_vals[cond > 1e-8]
    ax.scatter(s2_support, payoff[cond > 1e-8], color='red', s=50, zorder=5, label="Support points")
    ax.set_title(f"S1 = {s1:.2f}")
    ax.set_ylabel("Value")
    ax.legend()
    ax.grid(True)
plt.xlabel("S2")
plt.tight_layout()
plt.show()
```

A.1.6 SPX Marginals Linear Programming

```
"""Implementation of price bounds with a martingale constrained asset (using full SVI marginals)"""
import src.svi_lib as svi_lib
import numpy as np
import mosek.fusion as mf
from scipy.sparse import coo_matrix
import matplotlib.pyplot as plt
import numpy.ma as ma
T1 = 20/251
S1_forw = 5489.83
S1_strikes = np.linspace(4500,5800,27)
S1_market_vol=np.array([0.30833965626324, 0.295476337981542, 0.283463677875459,
```

```
   0.271111343370089,
                          0.258451404309255,
                             0.245507644865208,
                             0.233694271876973,
                             0.221358004433688,
                          0.209097873714426,
                             0.197249412151294,
                             0.186334324110301,
                             0.174675684455109,
                          0.164666282229566,
                             0.15488794443707,
                             0.145823620475546,
                             0.137304198281544,
                          0.129231575271196,
                             0.121548584918863,
                             0.114417742487412,
                             0.108019587373161,
                          0.102797547251224,
                             0.0988783071539835,
                             0.0959001836718852,
                             0.0934990753558453,
                           0.0918556359282457,
                              0.0918871975138243,
                              0.0934062296063838])

T2 = 40/251
S2_forw = 5509.62
S2_strikes = np.linspace(4500,5800,27)
S2_market_vol=np.array([0.256853, 0.247760, 0.239149,
   0.230164,
                          0.221471, 0.213216, 0.204941,
                             0.196783,
                          0.188797, 0.180978, 0.173596,
                             0.166311,
                          0.159104, 0.152338, 0.145637,
                             0.139202,
                          0.132974, 0.127012, 0.121383,
                             0.116176,
                          0.111556, 0.108040, 0.105025,
                             0.102493,
                          0.100738, 0.099360, 0.098370])

S1_strikes = S1_strikes[5:22]
```

```
S1_market_vol = S1_market_vol[5:22]
S2_strikes = S2_strikes[5:22]
S2_market_vol = S2_market_vol[5:22]
S1_norm= S1_strikes/S1_forw
S2_norm= S2_strikes/S2_forw

# Fit SVI and build marginal densities
s1opt = svi_lib.svi_fit(np.log(S1_norm), S1_market_vol,
    [0.01]*5, T1)
s2opt = svi_lib.svi_fit(np.log(S2_norm), S2_market_vol,
    [0.01]*5, T2)

def mu_s1(s):
    return svi_lib.SVIDensity(s,s1opt.x)

def mu_s2(s):
    return svi_lib.SVIDensity(s,s2opt.x)

# Discretisation grid in normalised space
grid_size = 200
s1_vals = np.linspace(0.9, 1.1, grid_size)
s2_vals = np.linspace(0.9, 1.1, grid_size)

ds1 = s1_vals[1] - s1_vals[0]
ds2 = s2_vals[1] - s2_vals[0]

nvar = grid_size**2
tol = 0.00001

# Discretised marginals
mu1_disc = np.array([mu_s1(s) for s in s1_vals]) * ds1
mu2_disc = np.array([mu_s2(s) for s in s2_vals]) * ds2
mu1_disc /= mu1_disc.sum()
mu2_disc /= mu2_disc.sum()
mu1_disc = np.clip(mu1_disc, 0, None)
mu2_disc = np.clip(mu2_disc, 0, None)

# Constraints
rows, cols, data = [], [], []
b_eq = []
rowcount = 0

# S1 marginal constraints
```

```
for i, prob in enumerate(mu1_disc):
    for j in range(grid_size):
        idx = i * grid_size + j
        rows.append(rowcount)
        cols.append(idx)
        data.append(1.0)
    b_eq.append(prob)
    rowcount += 1

# S2 marginal constraints
for j, prob in enumerate(mu2_disc):
    for i in range(grid_size):
        idx = i * grid_size + j
        rows.append(rowcount)
        cols.append(idx)
        data.append(1.0)
    b_eq.append(prob)
    rowcount += 1

# Martingale constraints
for i, s1 in enumerate(s1_vals):
    for j, s2 in enumerate(s2_vals):
        idx = i * grid_size + j
        rows.append(rowcount)
        cols.append(idx)
        data.append(s2 - s1)
    b_eq.append(0.0)
    rowcount += 1

A_sparse = coo_matrix((data, (rows, cols)), shape=(
   rowcount, nvar))
b_eq = np.array(b_eq)
b_lower = b_eq - tol
b_upper = b_eq + tol

A_mosek = mf.Matrix.sparse(
    A_sparse.shape[0],
    A_sparse.shape[1],
    A_sparse.row,
    A_sparse.col,
    A_sparse.data
)
```

```
S1_grid, S2_grid = np.meshgrid(s1_vals, s2_vals,
    indexing="ij")
payoff_flat =(np.log((S2_grid/S1_grid)*(S1_forw/S2_forw)
    )**2).flatten()# np.abs(S2_grid*S2_forw-S1_grid*
    S1_forw).flatten()#

# Minimise price
M_min = mf.Model("min_price")
x_min = M_min.variable("x", nvar, mf.Domain.greaterThan
    (0.0))
M_min.constraint("eqs", mf.Expr.mul(A_mosek, x_min), mf.
    Domain.inRange(b_lower, b_upper))
M_min.objective("obj", mf.ObjectiveSense.Minimize, mf.
    Expr.dot(payoff_flat, x_min))
M_min.solve()
min_price = M_min.primalObjValue()

# Maximise price
M_max = mf.Model("max_price")
x_max = M_max.variable("x", nvar, mf.Domain.greaterThan
    (0.0))
M_max.constraint("eqs", mf.Expr.mul(A_mosek, x_max), mf.
    Domain.inRange(b_lower, b_upper))
M_max.objective("obj", mf.ObjectiveSense.Maximize, mf.
    Expr.dot(payoff_flat, x_max))
M_max.solve()
max_price = M_max.primalObjValue()

"""OUTPUT 1 - Price bounds"""
print("Min price:", min_price)
print("Max price:", max_price)
print("Size of Bounded Interval: ", (max_price-min_price
    ))
print("Relative percentage: ", (max_price-min_price)
    /(0.5*(max_price+min_price)))

# Extract PMFs
pmf_opt = np.array(x_max.level()).reshape((grid_size,
    grid_size))
pmf_min = np.array(x_min.level()).reshape((grid_size,
    grid_size))
pmf_opt = np.clip(pmf_opt, 0, None)
pmf_min = np.clip(pmf_min, 0, None)
```

```
143
144 tol_cond = 1e-12
145 row_sums_opt = pmf_opt.sum(axis=1, keepdims=True)
146 row_sums_min = pmf_min.sum(axis=1, keepdims=True)
147 pmf_opt_cond = np.divide(pmf_opt, row_sums_opt, out=np.
    zeros_like(pmf_opt), where=row_sums_opt > tol_cond)
148 pmf_min_cond = np.divide(pmf_min, row_sums_min, out=np.
    zeros_like(pmf_min), where=row_sums_min > tol_cond)
149
150 """OUTPUT 2 - Heatmaps of the conditional probabilities
    showing sparseness"""
151 plt.figure(figsize=(10, 8))
152 masked = ma.masked_where(pmf_opt_cond.T <= 1e-12,
    pmf_opt_cond.T)
153 cmap = plt.cm.viridis.copy()
154 cmap.set_bad(color='white')
155 plt.imshow(masked, aspect='auto', origin='lower', extent
    =[s1_vals[0], s1_vals[-1], s2_vals[0], s2_vals[-1]],
    cmap=cmap)
156 plt.colorbar(label="P(S2 | S1)")
157 plt.xlabel("S1")
158 plt.ylabel("S2")
159 plt.title("Conditional distribution P(S2 | S1) for the
    maximal price PMF\n (Exact Zeros in white)")
160 plt.show()
161
162 plt.figure(figsize=(10, 8))
163 masked_min = ma.masked_where(pmf_min_cond.T <= 1e-12,
    pmf_min_cond.T)
164 cmap_min = plt.cm.viridis.copy()
165 cmap_min.set_bad(color='white')
166 plt.imshow(masked_min, aspect='auto', origin='lower',
    extent=[s1_vals[0], s1_vals[-1], s2_vals[0], s2_vals
    [-1]], cmap=cmap_min)
167 plt.colorbar(label="P(S2 | S1)")
168 plt.xlabel("S1")
169 plt.ylabel("S2")
170 plt.title(f"Conditional distribution P(S2 | S1) for the
    minimal price PMF\n (Exact Zeros in white)")
171 plt.show()
172
173 """OUTPUT 3 & 4 - Dual variables for the constraints (
    Superhedge & Subhedge)"""
```

```
dual_vars = M_max.getConstraint("eqs").dual().flatten()
dual_vars_min = M_min.getConstraint("eqs").dual().
    flatten()

# S1 marginal duals, S2 marginal duals, martingale duals
s1_duals = dual_vars[:grid_size]
s2_duals = dual_vars[grid_size:2*grid_size]
mart_duals = dual_vars[2*grid_size:]
s1_duals_min = dual_vars_min[:grid_size]
s2_duals_min = dual_vars_min[grid_size:2*grid_size]
mart_duals_min = dual_vars_min[2*grid_size:]

# Superhedge
plt.figure(figsize=(8, 5))
plt.plot(s1_vals, s1_duals, label="S1 Marginal Dual",
    color="tab:blue")
plt.plot(s2_vals, s2_duals, label="S2 Marginal Dual",
    color="tab:green")
plt.title("Superhedge Marginals (Upper Bound)")
plt.xlabel("S1 / S2")
plt.ylabel("Dual Value")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Subhedge
plt.figure(figsize=(8, 5))
plt.plot(s1_vals, s1_duals_min, label="S1 Marginal Dual"
    , color="tab:blue")
plt.plot(s2_vals, s2_duals_min, label="S2 Marginal Dual"
    , color="tab:green")
plt.title("Subhedge Marginals (Lower Bound)")
plt.xlabel("S1 / S2")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Martingale duals
plt.figure(figsize=(8, 5))
plt.plot(s1_vals, mart_duals, label="Martingale Dual (
    Superhedge)", color="tab:orange")
```

```
plt.plot(s1_vals, mart_duals_min, label="Martingale Dual (Subhedge)", color="tab:red", linestyle="--")
plt.title("Martingale Duals")
plt.xlabel("S1")
plt.ylabel("Dual Value")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

"""OUTPUT 5 - Plot of the full Lagrangian for upper bound showing duality via support points"""
lagrangian_3d = np.zeros((len(s1_vals), len(s2_vals)))
for i, s1 in enumerate(s1_vals):
    s1_constant = s1_duals[i]
    s2_contribution = s2_duals
    martingale_contribution = mart_duals[i] * (s2_vals - s1)
    lagrangian_3d[i, :] = s1_constant + s2_contribution + martingale_contribution

payoff_3d = payoff_flat.reshape((len(s1_vals), len(s2_vals)))
difference_upper = lagrangian_3d - payoff_3d

significant_diff_upper = np.where(np.abs(difference_upper) >= 1e-6, difference_upper, np.nan)
original_cmap = plt.cm.viridis.copy()
original_cmap.set_bad(color='white')
plt.figure(figsize=(12, 8))
im = plt.imshow(significant_diff_upper.T, aspect='auto', origin='lower',
                extent=[s1_vals[0], s1_vals[-1], s2_vals[0], s2_vals[-1]],
                cmap=original_cmap)
plt.colorbar(im, label='Lagrangian - Payoff')
plt.xlabel('S1')
plt.ylabel('S2')
plt.title('Lagrangian in Upper Dual minus Payoff\n Exact zeros in white (support points)')
plt.show()

```

```
"""OUTPUT 6 - Cross-sections of the Lagrangian for
   Superhedging"""
s1_indices = [int(grid_size*0.3), int(grid_size*0.5),
   int(grid_size*0.7)]
fig, axes = plt.subplots(len(s1_indices), 1, figsize
   =(10, 8), sharex=True)
for idx, ax in zip(s1_indices, axes):
    s1 = s1_vals[idx]
    lagrangian = lagrangian_3d[idx, :]
    payoff = payoff_3d[idx, :]
    cond = pmf_opt[idx, :]
    s2_support = s2_vals[cond > 1e-8]
    ax.plot(s2_vals, payoff, label="Payoff", linewidth
       =2)
    ax.plot(s2_vals, lagrangian, label="Lagrangian",
       linewidth=2)
    ax.scatter(s2_support, payoff[cond > 1e-8], color='
       red', s=50, zorder=5, label="Support points")
    ax.set_title(f"S1 = {s1:.2f}")
    ax.set_ylabel("Value")
    ax.legend()
    ax.grid(True)
plt.xlabel("S2")
plt.tight_layout()
plt.show()
```

A.1.7 Log-normal Marginals Linear Programming

```
"""Implementation of price bounds with a martingale
   constrained lognormal marginals"""
import numpy as np
from scipy.stats import lognorm
from scipy.sparse import coo_matrix
import mosek.fusion as mf
import matplotlib.pyplot as plt
import numpy.ma as ma

# Synthetic lognormal parameters
S0 = 5500
T1 = 20 / 251
T2 = 40 / 251
sigma = 0.2
```

```
14
15 grid_size = 400
16 s1_vals = np.linspace(0.8 * S0, 1.2 * S0, grid_size)
17 s2_vals = np.linspace(0.8 * S0, 1.2 * S0, grid_size)
18 nvar = grid_size ** 2
19
20 # Allows some slackness in fitting the martingale
      condition
21 tol = 0.00001
22
23 # Marginal density masses for S1 and S2 (lognormal)
24 ds1 = s1_vals[1] - s1_vals[0]
25 ds2 = s2_vals[1] - s2_vals[0]
26 s1_pdf_vals = lognorm.pdf(s1_vals, s=sigma * np.sqrt(T1)
      , scale=S0)
27 s2_pdf_vals = lognorm.pdf(s2_vals, s=sigma * np.sqrt(T2)
      , scale=S0)
28 s1_pmf = s1_pdf_vals * ds1
29 s2_pmf = s2_pdf_vals * ds2
30 s1_pmf /= s1_pmf.sum()
31 s2_pmf /= s2_pmf.sum()
32
33 # Constraints
34 rows, cols, data = [], [], []
35 b_eq = []
36 rowcount = 0
37
38 # S1 marginal constraints
39 for i, prob in enumerate(s1_pmf):
40     for j in range(grid_size):
41         idx = i * grid_size + j
42         rows.append(rowcount)
43         cols.append(idx)
44         data.append(1.0)
45     b_eq.append(prob)
46     rowcount += 1
47
48 # S2 marginal constraints
49 for j, prob in enumerate(s2_pmf):
50     for i in range(grid_size):
51         idx = i * grid_size + j
52         rows.append(rowcount)
53         cols.append(idx)
```

```
        data.append(1.0)
    b_eq.append(prob)
    rowcount += 1

# Martingale constraints
for i, s1 in enumerate(s1_vals):
    for j, s2 in enumerate(s2_vals):
        idx = i * grid_size + j
        rows.append(rowcount)
        cols.append(idx)
        data.append(s2 - s1)
    b_eq.append(0.0)
    rowcount += 1

A_sparse = coo_matrix((data, (rows, cols)), shape=(
   rowcount, nvar))
b_eq = np.array(b_eq)

b_lower = b_eq - tol
b_upper = b_eq + tol

A_mosek = mf.Matrix.sparse(
    A_sparse.shape[0],
    A_sparse.shape[1],
    A_sparse.row,
    A_sparse.col,
    A_sparse.data
)

# Objective: log-contract payoff (variance swap)
S1_grid, S2_grid = np.meshgrid(s1_vals, s2_vals,
   indexing="ij")
payoff_flat =(np.log(S2_grid / S1_grid) ** 2).flatten()#
   (np.abs(S2_grid - S1_grid)).flatten()#

M_min = mf.Model("min_price")
x_min = M_min.variable("x", nvar, mf.Domain.greaterThan
   (0.0))
M_min.constraint("eqs", mf.Expr.mul(A_mosek, x_min), mf.
   Domain.inRange(b_lower, b_upper))
M_min.objective("obj", mf.ObjectiveSense.Minimize, mf.
   Expr.dot(payoff_flat, x_min))
M_min.solve()
```

```
min_price = M_min.primalObjValue()

M_max = mf.Model("max_price")
x_max = M_max.variable("x", nvar, mf.Domain.greaterThan(0.0))
M_max.constraint("eqs", mf.Expr.mul(A_mosek, x_max), mf.Domain.inRange(b_lower, b_upper))
M_max.objective("obj", mf.ObjectiveSense.Maximize, mf.Expr.dot(payoff_flat, x_max))
M_max.solve()
max_price = M_max.primalObjValue()

"""OUTPUT 1 - Price bounds"""
print("Min price:", min_price)
print("Max price:", max_price)

# Extract PMFs
pmf_opt = np.array(x_max.level()).reshape((grid_size, grid_size))
pmf_min = np.array(x_min.level()).reshape((grid_size, grid_size))
pmf_opt = np.clip(pmf_opt, 0, None)
pmf_min = np.clip(pmf_min, 0, None)

# Conditional probabilities
tol = 1e-12
row_sums_opt = pmf_opt.sum(axis=1, keepdims=True)
row_sums_min = pmf_min.sum(axis=1, keepdims=True)
pmf_opt_cond = np.divide(pmf_opt, row_sums_opt, out=np.zeros_like(pmf_opt), where=row_sums_opt > tol)
pmf_min_cond = np.divide(pmf_min, row_sums_min, out=np.zeros_like(pmf_min), where=row_sums_min > tol)

"""OUTPUT 2 - Heatmaps of the conditional probabilities showing sparseness"""
plt.figure(figsize=(10, 8))
masked = ma.masked_where(pmf_opt_cond.T <= 1e-12, pmf_opt_cond.T)
cmap = plt.cm.viridis.copy()
cmap.set_bad(color='white')
plt.imshow(masked, aspect='auto', origin='lower', extent=[s1_vals[0], s1_vals[-1], s2_vals[0], s2_vals[-1]], cmap=cmap)
```

```
plt.colorbar(label="P(S2 | S1)")
plt.xlabel("S1")
plt.ylabel("S2")
plt.title("Conditional distribution P(S2 | S1) for the maximal price PMF\n (Exact Zeros in white)")
plt.show()

plt.figure(figsize=(10, 8))
masked_min = ma.masked_where(pmf_min_cond.T <= 1e-12, pmf_min_cond.T)
cmap_min = plt.cm.viridis.copy()
cmap_min.set_bad(color='white')
plt.imshow(masked_min, aspect='auto', origin='lower', extent=[s1_vals[0], s1_vals[-1], s2_vals[0], s2_vals[-1]], cmap=cmap_min)
plt.colorbar(label="P(S2 | S1)")
plt.xlabel("S1")
plt.ylabel("S2")
plt.title(f"Conditional distribution P(S2 | S1) for the minimal price PMF\n (Exact Zeros in white, Sigma = {sigma})")
plt.show()

"""OUTPUT 3 & 4 - Dual variables for the constraints (Superhedge & Subhedge)"""
dual_vars = M_max.getConstraint("eqs").dual().flatten()
dual_vars_min = M_min.getConstraint("eqs").dual().flatten()

s1_duals = dual_vars[:grid_size]
s2_duals = dual_vars[grid_size:2*grid_size]
mart_duals = dual_vars[2*grid_size:]
s1_duals_min = dual_vars_min[:grid_size]
s2_duals_min = dual_vars_min[grid_size:2*grid_size]
mart_duals_min = dual_vars_min[2*grid_size:]

# Superhedge
plt.figure(figsize=(8, 5))
plt.plot(s1_vals, s1_duals, label="S1 Marginal Dual", color="tab:blue")
plt.plot(s2_vals, s2_duals, label="S2 Marginal Dual", color="tab:green")
plt.title("Superhedge Marginals (Upper Bound)")
```

```
plt.xlabel("S1 / S2")
plt.ylabel("Dual Value")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Subhedge
plt.figure(figsize=(8, 5))
plt.plot(s1_vals, s1_duals_min, label="S1 Marginal Dual", color="tab:blue")
plt.plot(s2_vals, s2_duals_min, label="S2 Marginal Dual", color="tab:green")
plt.title("Subhedge Marginals (Lower Bound)")
plt.xlabel("S1 / S2")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Martingale
plt.figure(figsize=(8, 5))
plt.plot(s1_vals, mart_duals, label="Martingale Dual (Superhedge)", color="tab:orange")
plt.plot(s1_vals, mart_duals_min, label="Martingale Dual (Subhedge)", color="tab:red", linestyle="--")
plt.title("Martingale Duals")
plt.xlabel("S1")
plt.ylabel("Dual Value")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

"""OUTPUT 5 - Plot of the full Lagrangian for upper bound showing duality via support points"""
lagrangian_3d = np.zeros((len(s1_vals), len(s2_vals)))
for i, s1 in enumerate(s1_vals):
    s1_constant = s1_duals[i]
    s2_contribution = s2_duals
    martingale_contribution = mart_duals[i] * (s2_vals - s1)
```

```
    lagrangian_3d[i, :] = s1_constant + s2_contribution
        + martingale_contribution

payoff_3d = payoff_flat.reshape((len(s1_vals), len(
    s2_vals)))
difference_upper = lagrangian_3d - payoff_3d

significant_diff_upper = np.where(np.abs(
    difference_upper) >= 1e-6, difference_upper, np.nan)
original_cmap = plt.cm.viridis.copy()
original_cmap.set_bad(color='white')
plt.figure(figsize=(12, 8))
im = plt.imshow(significant_diff_upper.T, aspect='auto',
    origin='lower',
                extent=[s1_vals[0], s1_vals[-1], s2_vals
                    [0], s2_vals[-1]],
                cmap=original_cmap)
plt.colorbar(im, label='Lagrangian - Payoff')
plt.xlabel('S1')
plt.ylabel('S2')
plt.title('Lagrangian in Upper Dual minus Payoff\n Exact
    zeros in white (support points)')
plt.show()

"""OUTPUT 6 - Cross-sections of the Lagrangian for
    Superhedging"""
s1_indices = [int(grid_size*0.3), int(grid_size*0.5),
    int(grid_size*0.7)]
fig, axes = plt.subplots(len(s1_indices), 1, figsize
    =(10, 8), sharex=True)
for idx, ax in zip(s1_indices, axes):
    s1 = s1_vals[idx]
    lagrangian = lagrangian_3d[idx, :]
    payoff = payoff_3d[idx, :]
    cond = pmf_opt[idx, :]
    s2_support = s2_vals[cond > 1e-8]
    ax.plot(s2_vals, payoff, label="Payoff", linewidth
        =2)
    ax.plot(s2_vals, lagrangian, label="Lagrangian",
        linewidth=2)
    ax.scatter(s2_support, payoff[cond > 1e-8], color='
        red', s=50, zorder=5, label="Support points")
    ax.set_title(f"S1 = {s1:.2f}")
```

```
    ax.set_ylabel("Value")
    ax.legend()
    ax.grid(True)
plt.xlabel("S2")
plt.tight_layout()
plt.show()
```

A.1.8 SPX/VIX Constrained Linear Programming

```
"""LP for price bounds jointly calibrated to SPX and
   VIX vanilla options"""
import src.black_lib as bs_lib
import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
import matplotlib.cm as cm
import mosek.fusion as mf
from matplotlib.colors import Normalize

#CBOE on 21st June 2024
T1=20/251
S1_forw=5489.83
S1_strikes=np.linspace(4500,5800,27)
S1_norm=S1_strikes/S1_forw
S1_market_vol=np.array([0.30833965626324,
   0.295476337981542, 0.283463677875459,
   0.271111343370089, 0.258451404309255,
   0.245507644865208, 0.233694271876973,
   0.221358004433688, 0.209097873714426,
   0.197249412151294, 0.186334324110301,
   0.174675684455109, 0.164666282229566,
   0.15488794443707, 0.145823620475546,
   0.137304198281544, 0.129231575271196,
   0.121548584918863, 0.114417742487412,
   0.108019587373161, 0.102797547251224,
   0.0988783071539835, 0.0959001836718852,
   0.0934990753558453, 0.0918556359282457,
   0.0918871975138243, 0.0934062296063838])

T2=40/251
S2_forw=5509.62
S2_strikes=np.linspace(4500,5800,27)
```

```
S2_norm=S2_strikes/S2_forw
S2_market_vol=np.array([0.256853, 0.247760, 0.239149,
   0.230164, 0.221471, 0.213216, 0.204941, 0.196783,
   0.188797, 0.180978, 0.173596, 0.166311, 0.159104,
   0.152338, 0.145637, 0.139202, 0.132974, 0.127012,
   0.121383, 0.116176, 0.111556, 0.108040, 0.105025,
   0.102493, 0.100738, 0.099360, 0.098370])

T_V=20/251
V_forw=0.1437955868051
V_strikes=np.array([0.1200,  0.1250,    0.1300,
   0.1350,    0.1400,    0.1450,    0.1500,    0.1550,
      0.1600,  0.17, 0.18, 0.19, 0.2, 0.21])
V_norm=V_strikes/V_forw
V_market_vol=np.array([0.502489195, 0.531472613,
   0.580483519, 0.63710948, 0.68409345, 0.736859132,
   0.787044999, 0.827805687, 0.870005448, 0.94647844,
   1.020997407, 1.09475647, 1.167988237, 1.219598942])

tau = T2 - T1

S1_strikes=S1_strikes[7:]
S1_market_vol=S1_market_vol[7:]
S2_strikes=S2_strikes[7:]
S2_market_vol=S2_market_vol[7:]
V_strikes=V_strikes[1:9]
V_market_vol=V_market_vol[1:9]

"""Market Prices of those options"""
C1 = np.array([bs_lib.black_scholes_call(S1_forw, K, T1,
    vol) for K,vol in zip(S1_strikes, S1_market_vol)])
C2 = np.array([bs_lib.black_scholes_call(S2_forw, K, T2,
    vol) for K,vol in zip(S2_strikes, S2_market_vol)])
CV= np.array([bs_lib.black_scholes_call(V_forw, K, T_V,
   vol) for K,vol in zip(V_strikes, V_market_vol)])

"""L func for dispersion"""
def L(r): return -2.0 / tau * np.log(r)

"""pmf grid"""
grid_size =50
s1_vals = np.linspace(0.8*S1_forw, 1.1*S1_forw,
   grid_size)
```

```
s2_vals = np.linspace(0.8*S2_forw, 1.1*S2_forw,
   grid_size)
v_vals = np.linspace(0.5*V_forw, 1.5*V_forw, grid_size)
S1, S2, V = np.meshgrid(s1_vals, s2_vals, v_vals,
   indexing='ij')
num_vars = grid_size ** 3

"""payoff for use in objective function"""
payoff_grid =  np.abs(S2-S1)
payoff_flat = payoff_grid.flatten()

"""Constraints built up and put into sparse matrix
   notation"""
# SPX T1 calls
A_S1 = sparse.lil_matrix((len(S1_strikes), num_vars))
for k, K1 in enumerate(S1_strikes):
    for i in range(grid_size):
        for j in range(grid_size):
            for l in range(grid_size):
                idx = i * grid_size * grid_size + j *
                   grid_size + l
                payoff = s1_vals[i] - K1
                if payoff > 0:
                    A_S1[k, idx] = payoff
A_S1 = A_S1.tocsr()

# SPX T2 calls
A_S2 = sparse.lil_matrix((len(S2_strikes), num_vars))
for k, K2 in enumerate(S2_strikes):
    for i in range(grid_size):
        for j in range(grid_size):
            for l in range(grid_size):
                idx = i * grid_size * grid_size + j *
                   grid_size + l
                payoff = s2_vals[j] - K2
                if payoff > 0:
                    A_S2[k, idx] = payoff
A_S2 = A_S2.tocsr()

# VIX T1 calls
A_V = sparse.lil_matrix((len(V_strikes), num_vars))
for k, Kv in enumerate(V_strikes):
    for i in range(grid_size):
```

```
        for j in range(grid_size):
            for l in range(grid_size):
                idx = i * grid_size * grid_size + j *
                    grid_size + l
                payoff = v_vals[l] - Kv
                if payoff > 0:
                    A_V[k, idx] = payoff
A_V = A_V.tocsr()

# Martingale constraint
rows, cols, data = [], [], []
for i in range(grid_size):
    for l in range(grid_size):
        s1 = s1_vals[i]
        for j in range(grid_size):
            idx = i * grid_size * grid_size + j *
                grid_size + l
            rows.append(i * grid_size + l)
            cols.append(idx)
            data.append(s2_vals[j]/S2_forw - s1/S1_forw)
A_mart = sparse.csr_matrix((data, (rows, cols)), shape=(
    grid_size*grid_size, num_vars))
b_mart = np.zeros(grid_size * grid_size)

# Dispersion constraint
rows, cols, data = [], [], []
for i in range(grid_size):
    for l in range(grid_size):
        s1 = s1_vals[i]
        v = v_vals[l]
        for j in range(grid_size):
            idx = i * grid_size * grid_size + j *
                grid_size + l
            r = max(s2_vals[j] / s1, 1e-8)
            rows.append(i * grid_size + l)
            cols.append(idx)
            data.append(L(r*(S1_forw/S2_forw)) - v**2)
A_disp = sparse.csr_matrix((data, (rows, cols)), shape=(
    grid_size*grid_size, num_vars))
b_disp = np.zeros(grid_size * grid_size)

# Normalisation
A_norm = sparse.csr_matrix(np.ones((1, num_vars)))
```

```
b_norm = np.array([1.0])

"""Constraints stacked into sparse matrix"""
A_eq = sparse.vstack([A_S1, A_S2, A_V, A_norm, A_mart,
    A_disp]).tocsr()
b_eq = np.concatenate([C1, C2, CV, b_norm, b_mart,
    b_disp])
bounds = [(0, None)] * num_vars

"""Change constraints to Mosek fusion matrix"""
A_mosek = mf.Matrix.sparse(
    A_eq.shape[0],
    A_eq.shape[1],
    A_eq.tocoo().row,
    A_eq.tocoo().col,
    A_eq.tocoo().data
)

"""Lower bounds optimisation problem"""
M_min = mf.Model("min_price")
x = M_min.variable("x", num_vars, mf.Domain.greaterThan
    (0.0))
M_min.constraint("eqs", mf.Expr.mul(A_mosek, x), mf.
    Domain.equalsTo(b_eq))
M_min.objective("obj", mf.ObjectiveSense.Minimize, mf.
    Expr.dot(payoff_flat, x))
M_min.solve()
min_price = M_min.primalObjValue()
print("Min price:", min_price)

"""Lower bounds optimisation problem"""
M_max = mf.Model("max_price")
x_max = M_max.variable("x", num_vars, mf.Domain.
    greaterThan(0.0))
M_max.constraint("eqs", mf.Expr.mul(A_mosek, x_max), mf.
    Domain.equalsTo(b_eq))
M_max.objective("obj", mf.ObjectiveSense.Maximize, mf.
    Expr.dot(payoff_flat, x_max))
M_max.solve()
max_price = M_max.primalObjValue()
print("Max price:", max_price)

```

```
pmf_min = x.level().reshape(grid_size, grid_size,
    grid_size)
pmf_max = x_max.level().reshape(grid_size, grid_size,
    grid_size)

# Tests on constraint violations
martingale_error = np.zeros((grid_size, grid_size))
for i in range(grid_size):
    for l in range(grid_size):
        martingale_error[i, l] = np.sum((s2_vals -
            s1_vals[i]*(S2_forw/S1_forw)) * pmf_min[i, :,
             l])

dispersion_error = np.zeros((grid_size, grid_size))
for i in range(grid_size):
    for l in range(grid_size):
        s1 = s1_vals[i]
        v = v_vals[l]
        dispersion_error[i, l] = np.sum((L(np.maximum((
            s2_vals/S2_forw) / (s1/S1_forw), 1e-8)) - v
            **2) * pmf_min[i, :, l])

assert np.max(np.abs(dispersion_error)) < 1e-8, "
    Dispersion constraint violated"
assert np.max(np.abs(martingale_error)) < 1e-8, "
    Martingale constraint violated"

"""OUTPUT 1 - marginal PMF for (S1, S2) by summing over
    the VIX axis (axis=2)"""
pmf_s1_s2_marginal = pmf_max.sum(axis=2)
s1_marginal_sums = pmf_s1_s2_marginal.sum(axis=1,
    keepdims=True)
pmf_s2_cond_on_s1 = np.divide(pmf_s1_s2_marginal,
    s1_marginal_sums,
                              out=np.zeros_like(
                                  pmf_s1_s2_marginal),
                              where=s1_marginal_sums > 1
                                  e-12)
plt.figure(figsize=(10, 8))
masked = np.ma.masked_where(pmf_s2_cond_on_s1.T <= 1e
    -12, pmf_s2_cond_on_s1.T)
cmap = plt.cm.viridis.copy()
cmap.set_bad(color='white')
```

```
plt.imshow(masked, aspect='auto', origin='lower',
           extent=[s1_vals[0], s1_vals[-1], s2_vals[0],
              s2_vals[-1]],
           cmap=cmap)
plt.colorbar(label="P(S2 | S1) [VIX Integrated]")
plt.xlabel("S1")
plt.ylabel("S2")
plt.title("VIX-Integrated Conditional Distribution P(S2
   | S1) (Maximal Price PMF)")
plt.show()

"""OUTPUT 2: Point cloud of extremal solution"""
pmf_to_plot = pmf_max
tol = 1e-8
voxel_grid = pmf_to_plot > tol
colors = np.zeros(voxel_grid.shape + (4,))
probs = pmf_to_plot[voxel_grid]
norm = Normalize(vmin=probs.min(), vmax=probs.max())
cmap = plt.cm.viridis
colors[voxel_grid] = cmap(norm(probs))
fig = plt.figure(figsize=(12, 10))
ax = fig.add_subplot(projection='3d')
ax.voxels(voxel_grid, facecolors=colors, edgecolor='k',
   linewidth=0.2, alpha=0.7)
ax.set_xticks(np.linspace(0, grid_size, num=5))
ax.set_xticklabels([f"{val:.0f}" for val in np.linspace(
   s1_vals[0], s1_vals[-1], num=5)])
ax.set_yticks(np.linspace(0, grid_size, num=5))
ax.set_yticklabels([f"{val:.0f}" for val in np.linspace(
   s2_vals[0], s2_vals[-1], num=5)])
ax.set_zticks(np.linspace(0, grid_size, num=5))
ax.set_zticklabels([f"{val:.2%}" for val in np.linspace(
   v_vals[0], v_vals[-1], num=5)])
ax.set_xlabel('S1 Value')
ax.set_ylabel('S2 Value')
ax.set_zlabel('VIX Level')
ax.set_title('Maximal PMF for the Straddle')
mappable = cm.ScalarMappable(cmap=cmap, norm=norm)
cbar = fig.colorbar(mappable, shrink=0.5, aspect=10, ax=
   ax)
cbar.set_label('Probability Mass')
plt.show()

```

```
224
225 """OUTPUT 3 - The upper/lower VIX smile consistent with
       SPX and VIX options"""
226 vix_strikes_fine = np.linspace(V_strikes[0], V_strikes
       [-1], 25)
227 prices_min_smile = []
228 prices_max_smile = []
229
230 # Loop through each strike to plot upper/lower smile
231 for K in vix_strikes_fine:
232     vix_call_payoff_grid = np.maximum(V - K, 0)
233     vix_call_payoff_flat = vix_call_payoff_grid.flatten
          ()
234
235     M_min_smile = mf.Model("min_vix_call")
236     x_min_smile = M_min_smile.variable("x", num_vars, mf
          .Domain.greaterThan(0.0))
237     M_min_smile.constraint("eqs", mf.Expr.mul(A_mosek,
          x_min_smile), mf.Domain.equalsTo(b_eq))
238     M_min_smile.objective("obj", mf.ObjectiveSense.
          Minimize, mf.Expr.dot(vix_call_payoff_flat,
          x_min_smile))
239     M_min_smile.solve()
240     prices_min_smile.append(M_min_smile.primalObjValue()
          )
241
242     M_max_smile = mf.Model("max_vix_call")
243     x_max_smile = M_max_smile.variable("x", num_vars, mf
          .Domain.greaterThan(0.0))
244     M_max_smile.constraint("eqs", mf.Expr.mul(A_mosek,
          x_max_smile), mf.Domain.equalsTo(b_eq))
245     M_max_smile.objective("obj", mf.ObjectiveSense.
          Maximize, mf.Expr.dot(vix_call_payoff_flat,
          x_max_smile))
246     M_max_smile.solve()
247     prices_max_smile.append(M_max_smile.primalObjValue()
          )
248
249 prices_min_smile = np.array(prices_min_smile)
250 prices_max_smile = np.array(prices_max_smile)
251
252 iv_min = np.array([bs_lib.implied_vol(V_forw, K, T_V, p)
       for p, K in zip(prices_min_smile, vix_strikes_fine)
```

```
    ])
iv_max = np.array([bs_lib.implied_vol(V_forw, K, T_V, p)
    for p, K in zip(prices_max_smile, vix_strikes_fine)
    ])
fig, axs = plt.subplots(2, 1, figsize=(12, 10), sharex=
    True, gridspec_kw={'height_ratios': [3, 1]})
fig.suptitle('Jointly Calibrated Upper and Lower VIX
    Smile incuding No-Arbitrage Region', fontsize=16)
axs[0].plot(vix_strikes_fine, iv_max, color='red', lw
    =1.5, label='Upper VIX Smile')
axs[0].plot(vix_strikes_fine, iv_min, color='blue', lw
    =1.5, label='Lower VIX Smile')
axs[0].fill_between(vix_strikes_fine, iv_min, iv_max,
    color='gray', alpha=0.5, label='No-Arbitrage Region')
axs[0].plot(V_strikes, V_market_vol, 'go', markersize=8,
    label='Market VIX Smile (Constraints)')
axs[0].set_ylabel('Implied Volatility')
axs[0].legend()
axs[0].grid(True, linestyle='--', alpha=0.6)
axs[0].set_title('Implied VIX Smile Bounds')
iv_spread_bps = (iv_max - iv_min) * 10000
axs[1].plot(vix_strikes_fine, iv_spread_bps, color='
    black', lw=2)
axs[1].fill_between(vix_strikes_fine, iv_spread_bps,
    color='gray', alpha=0.5)
axs[1].set_xlabel('Strike')
axs[1].set_ylabel('Spread (bps)')
axs[1].grid(True, linestyle='--', alpha=0.6)
axs[1].set_title('Width of No-Arbitrage Region (Upper
    Smile - Lower Smile)')
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```