Zürcher Hochschule
Für Angewandte Wissenschaften

School of
Management and Law

32 Lines of Code to Price Two Factor Derivatives

Department of Banking, Finance and Insurance

Norbert Hilber

ZHAW School of Management and Law
St.-Georgen-Platz 2
P.O. Box
8401 Winterthur
Switzerland

Department of Banking, Finance and Insurance
https://www.zhaw.ch/en/sml/about-us/departments/department-of-banking-finance-insurance/

Author, Contact
Norbert Hilber
Norbert.Hilber@zhaw.ch

July 2019

32 LINES OF CODE TO PRICE TWO FACTOR DERIVATIVES

NORBERT HILBER

Abstract. We provide a simple Matlab and Python finite difference code to solve degenerate, linear parabolic partial differential equations with two space dimensions subject to a variety of boundary conditions. This code is then exemplarily applied to derivative pricing problems involving two stochastic factors. In particular, we price a call option in the Heston-Jacobi model and so-called autocallable, multi barrier convertibles in the bivariate Black-Scholes model.

Contents

1. Introduction 1
2. Pricing equation 3
2.1. Do we need boundary conditions? 5
3. Finite difference method 8
3.1. Finite difference discretisation with respect to space 8
3.2. Time stepping 9
4. Code 12
5. Examples 16
5.1. European call in the Jacobi-Heston model 16
5.2. Discretely monitored Lookback option in the CEV model 17
5.3. Autocallable Multi Barrier Reverse Convertible in the BS model 19
References 24
Appendix 25
A.1. The eigenvalues of $\mathbf{Q}$ 25
A.2. Matlab/Python code `matgen` 26
A.3. Matlab/Python code `perm_matrix` 28

1. Introduction

A large part of pricing problems in the financial derivatives industry involve multivariate stochastic processes, typically in continuous time. The reason for this is threefold. First, certain options and structured products are written on several underlyings. For example, about 85% of so-called Barrier Reverse Convertibles tradable on the exchange "SIX Structured Products" are written on at least two

Date: July 4, 2019.
2010 *Mathematics Subject Classification.* 35K15, 65M06, 91G20, 91G30.
Key words and phrases. Finite differences, partial differential equations, derivative securities.

underlyings, which are typically stocks. Second, the payoff of an exotic option depends not just on the level of the involved underlyings at maturity, but also on additional factors driven by the price history of these underlyings. For example, the value of lookback options depends on the realised maximum or minimum of the underlying over a certain time period. This renders the corresponding pricing problem multivariate, even if we consider just one underlying. Third, state-of-the-art pricing models involve - besides the underlying itself - additional stochastic factors like stochastic volatility, stochastic local volatility, path-dependent volatility and/or stochastic interest rates, see for example [Ber16]. Again, if we consider one underlying only, the corresponding pricing problem becomes typically multivariate in such models, nonetheless. Most of the mentioned problems do not admit a closed-form solution, such that we have to rely on approximation schemes. Here, the choice of the scheme heavily depends on the number d of stochastic factors. It is common knowledge that for d larger than 3 or 4, the prevailing scheme applied in the financial industry is any of the available Monte Carlo methods (MC). However, besides delivering random prices, the convergence rate of MC is typically low. Standard grid-based methods as finite differences or finite elements which rely on approximating the solution of the corresponding pricing partial differential equation (PDE) show (much) better convergence rates, but are limited to solve only low-dimensional pricing problems, since they suffer from the so-called curse of dimensionality and are thus applicable in practice for $d \leq 4$ only. The same can be said about transform methods (FFT, Cos and the like). By applying so-called sparse grid approximation methods, this upper bound may be shifted to the right, $d \leq 10$ say, see for example [HRSW13] and then references therein. Recently, deep neural networks (DNN) have proven to be a valuable alternative to both MC and grid-based methods, see e.g. [EHJ17]. However, an analysis of the approximation error and proven convergence rates for DNNs are available only for very specific (and somewhat unrealistic) pricing problems [EGJS18, GHJvW18]. The main advantage of DNN over standard grid based approximation schemes is that they do not suffer from the curse of dimensionality.

The mathematical complexity of the mentioned approximations methods and of their corresponding error analysis and approximation properties comes at different levels. Whereas MC methods typically require a sound knowledge in stochastic calculus, the finite element method can be understood and analysed only by applying concepts of functional analysis and PDE theory. The finite difference method (FDM) in contrast is (much) simpler (also in implementation). This simplicity comes at a price: certain pricing problems can not be solved well, as the convergence rate might break down due to discontinuous payoffs (even if we apply grid stretching), and convergence rates (with respect to the number of unknowns) are at most algebraic. Furthermore, the often heard argument that the domain of the pricing PDE is simply a rectangular box in $\mathbb{R}^d$ and hence an application of an approximation scheme which is able to handle more complex domains is not necessary is somewhat void. For example, we can not price continuously monitored lookback options by standard finite difference methods (unless we consider additional transformations of

the problem). Nevertheless, the FDM is a very simple and yet flexible approximation method for the majority of low-dimensional pricing problems. Despite its simplicity and flexibility, most of the scientific literature does only describe the FDM but does state code in the sense of a numerical receipt. The goal of this paper is thus to provide directly usable Matlab and Python codes which approximate the solution to (possibly degenerate) parabolic linear PDEs with time-independent coefficients in $d = 2$ space dimensions subject to variety of boundary conditions.

The paper is organised as follows. In chapter 2, we give a short description of the considered pricing problems and partial differential equations (PDEs), respectively. We further discuss on which subsets of the domain of the PDE we need to specify boundary conditions. In chapter 3 we develop a fully discrete approximation scheme to solve the PDE of chapter 2 numerically. We first use a second-order finite difference discretisation with respect to the space variables and then apply a ADI time-stepping scheme to discretise with respect to time. Here, special attention is paid to avoid solving large-banded linear systems. In the next chapter 4, we provide Matlab/Octave and Python codes which realise the developed fully discrete scheme of chapter 3. In the last chapter 5, we apply the routine presented in chapter 4 to particular pricing problems. For each of these problems we again provide codes.

2. Pricing equation

Consider a $\mathbb{R}^d$-valued stochastic process $\mathbf{X}(t) := (X_1(t), \dots, X_d(t))^\top$ solving the system of stochastic differential equations (SDE)

$$\mathrm{d}\mathbf{X}(t) = \boldsymbol{\mu}(\mathbf{X}(t), t)\mathrm{d}t + \boldsymbol{\sigma}(\mathbf{X}(t), t)\mathrm{d}\mathbf{W}(t)\ , \quad \mathbf{X}(0) = \mathbf{x}_0\ . \tag{2.1}$$

Herein, the column vector $\boldsymbol{\mu}(\mathbf{x}, t) \in \mathbb{R}^d$ and the matrix $\boldsymbol{\sigma}(\mathbf{x}, t) \in \mathbb{R}^{d\times m}$ are given by

$$\boldsymbol{\mu}(\mathbf{x}, t) := \begin{pmatrix} \mu_1(\mathbf{x}, t) \\ \mu_2(\mathbf{x}, t) \\ \vdots \\ \mu_d(\mathbf{x}, t) \end{pmatrix}, \quad \boldsymbol{\sigma}(\mathbf{x}, t) := \begin{pmatrix} \sigma_{11}(\mathbf{x}, t) & \sigma_{12}(\mathbf{x}, t) & \dots & \sigma_{1m}(\mathbf{x}, t) \\ \sigma_{21}(\mathbf{x}, t) & \sigma_{22}(\mathbf{x}, t) & \dots & \sigma_{2m}(\mathbf{x}, t) \\ & & \vdots & \\ \sigma_{d1}(\mathbf{x}, t) & \sigma_{d2}(\mathbf{x}, t) & \dots & \sigma_{dm}(\mathbf{x}, t) \end{pmatrix},$$

with functions $\mu_i, \sigma_{ij} : \mathbb{R}^d \times \mathbb{R}_0^+ \to \mathbb{R}$, $(\mathbf{x}, t) \mapsto \mu_i(\mathbf{x}, t), \sigma_{ij}(\mathbf{x}, t)$. Furthermore, the column vector $\mathbf{W}(t) = (W_1(t), \dots, W_m(t))^\top \in \mathbb{R}^m$ in (2.1) contains m independent standard Brownian motions. The process $\mathbf{X}(t)$ with dynamics (2.1) is assumed to model the time evolution of at least one financial underlying.

We consider a European style financial derivative with payoff function $\mathbb{R}^d \ni \mathbf{x} \mapsto g(\mathbf{x}) \in \mathbb{R}$ and maturity $T > 0$ written on $\mathbf{X}$. Then, if $r(t)$ denotes the (deterministic) continuously compounded risk free, the value $V(\mathbf{x}, t)$ of the derivative is

$$V(\mathbf{x}, t) = \mathbb{E}\big[e^{-\int_t^T r(u)\mathrm{d}u} g(\mathbf{X}(T)) \mid \mathbf{X}(t) = \mathbf{x}\big]\ . \tag{2.2}$$

Under certain conditions satisfied by the problem data $\boldsymbol{\mu}, \boldsymbol{\sigma}, r, g$, the Feynman-Kac theorem (see e.g. [HS00]) states that the function V equivalently solves the partial differential equation

$$\begin{cases} \partial_t V + \mathcal{A}V - r(t)V &= 0 & \text{in} & G \times [0, T[\\ V(\mathbf{x}, T) &= g(\mathbf{x}) & \text{in} & G \end{cases} \tag{2.3}$$

where $\mathcal{A}$ is the generator of the process $\mathbf{X}$ and where $G \subset \mathbb{R}^d$. If $\mathbf{X}$ is as in (2.1), then the generator is

$$\mathcal{A}f := \frac{1}{2}\mathrm{tr}[\boldsymbol{\sigma}(\mathbf{x},t)\boldsymbol{\sigma}(\mathbf{x},t)^\top D^2 f] + \boldsymbol{\mu}(\mathbf{x},t)^\top \nabla f \ . \tag{2.4}$$

Herein, $D^2 f = (\partial_{x_i x_j} f)_{1\le i,j\le d}$ and $\nabla f = (\partial_{x_1} f, \dots, \partial_{x_d} f)^\top$ denote the Hessian and the gradient, respectively, of f. Furthermore, $\mathrm{tr}[\mathbf{M}] = \sum_{i=1}^d m_{i,i}$ denotes the trace of a $d \times d$-matrix $\mathbf{M}$.

Example 2.1. i) In the multivariate Black-Scholes model, the d underlying follow a d-dimensional geometric Brownian motion, where $\mu_i(\mathbf{x},t) = (r - q_i)x_i$ ($q_i \ge 0$ denotes the continuous dividend yield of the i-th underlying) and

$$\boldsymbol{\sigma}(\mathbf{x},t) = \begin{pmatrix} x_1 & & & \\ & x_2 & & \\ & & \ddots & \\ & & & x_d \end{pmatrix} \mathbf{L} \ .$$

The $d \times d$-matrix $\mathbf{L}$ is such that $\mathbf{L}\mathbf{L}^\top = \boldsymbol{\Sigma} = (\sigma_{ij})$, the covariance-matrix of the log-returns $\ln(X_i(t)/X_i(0))$ of the underlying. Thus, the ij-entry of the matrix $\boldsymbol{\sigma}(\mathbf{x},t)\boldsymbol{\sigma}(\mathbf{x},t)^\top$ is $\sigma_{ij}x_i x_j$ and the generator $\mathcal{A}$ becomes

$$\mathcal{A} = \frac{1}{2}\sum_{i,j=1}^d \sigma_{ij} x_i x_j \partial_{x_i x_j} + \sum_{i=1}^d (r - q_i) x_i \partial_{x_i} \ .$$

For $d = 2$ in particular, we have with $x := x_1$, $y := x_2$ and $\sigma_{ii} := \sigma_i^2$ (the variance of the i-th underlying)

$$\mathcal{A} = \frac{1}{2}\sigma_1^2 x^2 \partial_{xx} + \frac{1}{2}\sigma_2^2 y^2 \partial_{xx} + \sigma_{12} xy \partial_{xy} + (r - q_1)x\partial_x + (r - q_2) y \partial_y \ . \tag{2.5}$$

The domain G is in this case $(x, y) \in G = \mathbb{R}^+ \times \mathbb{R}^+$.

ii) The Jacobi-Heston model is introduced in [AFP18] and is a generalisation of the benchmark stochastic volatility model of Heston [Hes93]. In this model, the bivariate process $\mathbf{X}(t) := (X(t), V(t))^\top$, where $X(t)$ is the underlying and $V(t)$ denotes its (stochastic) variance, solves (under an non-unique martingale measure) the system (2.1) with coefficients

$$\boldsymbol{\mu}(\mathbf{x},t) = \begin{pmatrix} (r-q)x \\ \kappa(m - v) \end{pmatrix} \ , \quad \boldsymbol{\sigma}(\mathbf{x},t) = \begin{pmatrix} \rho\sqrt{Q(v)}x & \sqrt{v - \rho^2 Q(v)}x \\ \delta\sqrt{Q(v)} & 0 \end{pmatrix} \ .$$

Herein, $\mathbf{x} = (x, v)$ and the function Q is, for a minimum and maximum level variance $0 \le v_{\min} < v_{\max}$, given as

$$Q(v) := \frac{(v - v_{\min})(v_{\max} - v)}{(\sqrt{v_{\max}} - \sqrt{v_{\min}})^2} \ge 0 \ .$$

Applying (2.4), the generator becomes

$$\mathcal{A} = \frac{1}{2}vx^2\partial_{xx} + \frac{1}{2}\delta^2 Q(v)\partial_{vv} + \rho\delta x Q(v)\partial_{xv} + (r-q)x\partial_x + \kappa(m-v)\partial_v \ . \tag{2.6}$$

The domain G is $G = \mathbb{R}^+ \times]v_{\min}, v_{\max}[$.

The described European style setting can be relaxed to certain exotic derivatives, for which the payoff g depends on additional stochastic factors derived from $\mathbf{X}(t)$. See section 5 for examples. For the rest of this note, we restrict to two stochastic factors, $d = 2$, and assume for simplicity time-independent coefficients $\boldsymbol{\mu}(\mathbf{x})$, $\boldsymbol{\sigma}(\mathbf{x})$. Thus, we consider pricing problems taking the form: Find $V = V(x, y, t)$ such that

$$\left\{ \begin{array}{rcll} \partial_t V + \mathcal{A}V - r(t)V &=& 0 & \text{in } G \times [0, T[\\ V(x, y, T) &=& g(x, y) & \text{in } G \end{array} \right. , \tag{2.7}$$

where the operator $\mathcal{A}$ is given by

$$\mathcal{A} = a_1(x,y)\partial_{xx} + a_2(x,y)\partial_{yy} + a_3(x,y)\partial_{xy} + b_1(x,y)\partial_x + b_2(x,y)\partial_y \tag{2.8}$$

for some bivariate functions a_i, b_i and c, and where the domain is $G =]x_l, x_r[\times]y_l, y_r[$ for some $-\infty \le x_l < x_r \le \infty$, $-\infty \le y_l < y_r \le \infty$. To solve the PDE (2.7) by the finite-difference-method, we change to the time-to-maturity $t \mapsto T - t$ and restrict the domain G if necessary to an open finite rectangular set, i.e, $x_{l,r}$ as well as $y_{l,r}$ are real numbers. Furthermore, we need to set conditions for the unknown function on the boundary ∂G of G. In this regard, we assume that the n-th (partial) derivative ($n = 0, 1, 2$) with respect to x and/or y on ∂G vanishes, i.e., $\partial_x^{(n)} w(x_{l,r}, y, t) = 0$ and $\partial_y^{(n)} w(x, y_{l,r}, t) = 0$. Thus, the truncated version of (2.7) is a particular case of the problem: Find $w(x, y, t)$ such that

$$\left\{ \begin{array}{rcll} \partial_t w + \mathcal{A}w + c(x,y)w &=& 0 & \text{in } G\times]0,T] \\ \partial_x^{(n_{x_l})} w(x_l, y, t) &=& 0 & \text{in }]y_l, y_r[\times]0,T] \\ \partial_x^{(n_{x_r})} w(x_r, y, t) &=& 0 & \text{in }]y_l, y_r[\times]0,T] \\ \partial_y^{(n_{y_l})} w(x, y_l, t) &=& 0 & \text{in }]x_l, x_r[\times]0,T] \\ \partial_y^{(n_{y_r})} w(x, y_r, t) &=& 0 & \text{in }]x_l, x_r[\times]0,T] \\ w(x, y, 0) &=& g(x,y) & \text{in } G \end{array} \right. \tag{2.9}$$

with $\mathcal{A}$ as in (2.8). Note that for certain pricing problems, some of the boundary conditions in (2.9) might be void, i.e., the PDE holds also on subsets of the boundary ∂G of G and no conditions for w are needed on these subsets. To investigate wether a boundary condition has to be set, we employ the theory of PDEs of second order with non-negative characteristic form, see for example [OR73].

2.1. **Do we need boundary conditions?** For $n_{x_l} = n_{x_r} = n_{y_l} = n_{y_r} = 0$, the PDE (2.9) is a particular problem of the following abstract equation. For a given set $\Omega \subset \mathbb{R}^n$ with boundary $\mathcal{B} := \partial\Omega$ and given functions $f(\mathbf{z}), \omega(\mathbf{z})$ consider the problem: Find $w(\mathbf{z})$ such that

$$\left\{ \begin{array}{rcll} \mathcal{L}w &=& f & \text{in } \Omega \\ w &=& \omega & \text{in } \mathcal{B}_2 \cup \mathcal{B}_3 \end{array} \right. . \tag{2.10}$$

The sets $\mathcal{B}_2, \mathcal{B}_3 \subset \mathcal{B}$ are subsets of the boundary of Ω and are defined below. Note that on the remaining boundary $\mathcal{B} \setminus (\mathcal{B}_2 \cup \mathcal{B}_3)$ no conditions on w are set. In (2.10), the operator $\mathcal{L}$ is - for given functions q_{ij}, b_i, c - defined as

$$\mathcal{L} := \sum_{i,j=1}^{n} q_{ij}(\mathbf{z})\partial_{z_i z_j} + \sum_{j=1}^{n} b_j(\mathbf{z})\partial_{z_j} + c(\mathbf{z}) \ .$$

We collect the n^2 functions q_{ij} in the $n \times n$-matrix $\mathbf{Q} = (q_{ij})$ and the n functions b_i in the column vector $\mathbf{b} = (b_i)$. Furthermore, we assume that $\mathbf{Q}$ is symmetric. If there holds for all $\mathbf{z} \in \Omega$

$$\boldsymbol{\xi}^\top \mathbf{Q}(\mathbf{z})\boldsymbol{\xi} \geq 0 \quad \forall \boldsymbol{\xi} \in \mathbb{R}^n \ ,$$

then the equation (2.10) is called a second-order PDE with non-negative characteristic. All linear pricing problems fall into this class of PDEs, see the example 2.2 below. For which points $\mathbf{z} \in \mathcal{B}$ on the boundary $\mathcal{B}$ we need to specify a boundary condition or not depends wether the characteristic vanishes. To be more precise, for $\mathbf{z} \in \mathcal{B}$ consider its unit inward normal vector $\boldsymbol{\nu} = (\nu_1, \ldots, \nu_n)^\top$ and divide the boundary $\mathcal{B}$ into the subsets $\mathcal{B}^0$ and $\mathcal{B}_3$ as follows

$$\begin{aligned} \mathcal{B}^0 &:= \{\mathbf{z} \in \mathcal{B} \mid \boldsymbol{\nu}^\top \mathbf{Q}(\mathbf{z})\boldsymbol{\nu} = 0\} \\ \mathcal{B}_3 &:= \{\mathbf{z} \in \mathcal{B} \mid \boldsymbol{\nu}^\top \mathbf{Q}(\mathbf{z})\boldsymbol{\nu} > 0\} = \mathcal{B} \setminus \mathcal{B}^0 \ . \end{aligned}$$

Furthermore, for $\mathbf{z} \in \mathcal{B}^0$ consider the so-called Fichera function

$$\beta(\mathbf{z}) := \sum_{i=1}^{n} \Big(b_i(\mathbf{z}) - \sum_{j=1}^{n} \partial_{z_j} q_{ij}(\mathbf{z}) \Big) \nu_i \ . \tag{2.11}$$

On the subset $\mathcal{B}^0$ the characteristic of the PDE is zero and we further subdivide $\mathcal{B}^0$ into the subsets (depending on the sign of β)

$$\mathcal{B}^0 = \mathcal{B}_0 \cup \mathcal{B}_1 \cup \mathcal{B}_2$$

with

$$\begin{aligned} \mathcal{B}_0 &:= \{\mathbf{z} \in \mathcal{B}^0 \mid \beta(\mathbf{z}) = 0\} \\ \mathcal{B}_1 &:= \{\mathbf{z} \in \mathcal{B}^0 \mid \beta(\mathbf{z}) > 0\} \\ \mathcal{B}_2 &:= \{\mathbf{z} \in \mathcal{B}^0 \mid \beta(\mathbf{z}) < 0\} \ . \end{aligned}$$

Thus, we have split the boundary $\mathcal{B}$ into the four subsets $\mathcal{B} = \mathcal{B}_0 \cup \mathcal{B}_1 \cup \mathcal{B}_2 \cup \mathcal{B}_3$; we do not need to specify a boundary condition on $\mathcal{B}_0 \cup \mathcal{B}_1$.

Example 2.2. Consider the Jacobi-Heston model in example 2.1 with generator $\mathcal{A}$ as in (2.6). The truncated pricing problem $-\partial_t w + \mathcal{A}w - rw = 0$ has then to be solved on $\Omega :=]0, x_r[\times]v_{\min}, v_{\max}[\times]0, T[$. The matrix $\mathbf{Q}$ and the vector $\mathbf{b}$ read for $\mathbf{z} = (x, v, t)$

$$\mathbf{Q}(\mathbf{z}) = \begin{pmatrix} \frac{1}{2}vx^2 & \frac{1}{2}\rho\delta x Q(v) & 0 \\ \frac{1}{2}\rho\delta x Q(v) & \frac{1}{2}\delta^2 Q(v) & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad \mathbf{b}(\mathbf{z}) = \begin{pmatrix} (r-q)x \\ \kappa(m-v) \\ -1 \end{pmatrix} \ . \tag{2.12}$$

Hence, the characteristic is

$$\boldsymbol{\nu}^\top \mathbf{Q}(\mathbf{z})\boldsymbol{\nu} = \frac{1}{2}vx^2\nu_1^2 + \rho\delta x Q(v)\nu_1\nu_2 + \frac{1}{2}\delta^2 Q(v)\nu_2^2$$

from which is it possible to prove that it is indeed non-negative (the eigenvalues of $\mathbf{Q}$ are ≥ 0, compare with the appendix A.1). We study the sign of the characteristic on all six faces of the cube Ω. The five faces where the characteristic is zero (the set $\mathcal{B}^0$) are $\{0\}\times]v_{\min}, v_{\max}[\times]0,T[$, $]0,x_r[\times\{v_{\min}, v_{\max}\}\times]0,T[$ and $]0,x_r[\times]v_{\min}, v_{\max}[\times\{0,T\}$. Thus, on the remaining face (the set $\mathcal{B}_3$) we need to a specify boundary condition. In particular, on the face $\{x_r\}\times]v_{\min}, v_{\max}[\times]0,T[$ we need to specify $w(x_r, v, t) = \omega_1(v,t)$ for some function ω_1. To find the splitting $\mathcal{B}^0 = \mathcal{B}_0 \cup \mathcal{B}_1 \cup \mathcal{B}_2$, we calculate the Fichera function

$$\begin{aligned}\beta(\mathbf{z}) = \big((r-q)x - (vx + \rho\delta x Q'(v))\big)\nu_1 \\ + \big(\kappa(m-v) - (\rho\delta Q(v) + \frac{1}{2}\delta^2 Q'(v))\big)\nu_2 - \nu_3\ .\end{aligned}$$

Since $Q'(v) = (-2v + v_{\min} + v_{\max})/(\sqrt{v_{\max}} - \sqrt{v_{\min}})^2$, we have for the five faces belonging to $\mathcal{B}^0$

- the face $\{0\}\times]v_{\min}, v_{\max}[\times]0,T[$. Since $x = 0$ (zero stock price) and $\boldsymbol{\nu} = (1,0,0)^\top$ we have $\beta = 0$, such that this face belongs to $\mathcal{B}_0$ and no boundary condition has to be set.
- the face $]0,x_r[\times\{v_{\min}\}\times]0,T[$. Since $Q(v_{\min}) = 0$ and $\boldsymbol{\nu} = (0,1,0)^\top$ we have $\beta = \kappa(m - v_{\min}) - \frac{1}{2}\delta^2 Q'(v_{\min})$, such that we need no boundary condition if

$$\kappa(m - v_{\min}) \geq \frac{1}{2}\delta^2 \frac{v_{\max} - v_{\min}}{(\sqrt{v_{\max}} - \sqrt{v_{\min}})^2}\ . \tag{2.13}$$

- the face $]0,x_r[\times\{v_{\max}\}\times]0,T[$. Since $Q(v_{\max}) = 0$ and $\boldsymbol{\nu} = (0,-1,0)^\top$ we have $\beta = -\kappa(m - v_{\max}) + \frac{1}{2}\delta^2 Q'(v_{\max})$, such that we need no boundary condition if

$$\kappa(m - v_{\max}) \leq \frac{1}{2}\delta^2 \frac{v_{\min} - v_{\max}}{(\sqrt{v_{\max}} - \sqrt{v_{\min}})^2}\ . \tag{2.14}$$

- the face $]0,x_r[\times]v_{\min}, v_{\max}[\times\{0\}$. Since $\boldsymbol{\nu} = (0,0,1)^\top$ we find $\beta = -1$ such that this face belongs to $\mathcal{B}_2$ and we have to specify a boundary condition $w(x,v,0) = \omega_2(x,v)$. Because $t = 0$ (zero time-to-maturity, i.e., physical time equal to maturity) the function ω_2 is the payoff function of the option.
- the face $]0,x_r[\times]v_{\min}, v_{\max}[\times\{T\}$. Since $\boldsymbol{\nu} = (0,0,-1)^\top$ there holds $\beta = 1$ and this face belongs to $\mathcal{B}_1$. There is no boundary condition needed (we solve the PDE by some time stepping scheme up to $t = T$).

The conditions (2.13) and (2.14) can be summarised as

$$\delta^2 \frac{v_{\max} - v_{\min}}{(\sqrt{v_{\max}} - \sqrt{v_{\min}})^2} \leq 2\kappa \min\{m - v_{\min}, v_{\max} - m\}\ ; \tag{2.15}$$

this is the condition such that the variance process $V(t)$ stays in the interval $]v_{\min}, v_{\max}[$ for all $t \geq 0$, compare with [AFP18, Theorem 2.1]. If (2.15) is violated, we need to specify a boundary condition on the faces $]0,x_r[\times\{v_{\min}, v_{\max}\}\times]0,T[$.

In the next section, we develop a second order finite-difference-method to solve (2.9) numerically under the assumption that the coefficient functions a_i, b_i and c factorise with respect to the coordinates.

3. Finite difference method

We first discretise the PDE (2.9) in space; the resulting linear system of ODEs (with respect to time) is then approximatively solved by a time-marching scheme.

3.1. **Finite difference discretisation with respect to space.** As mentioned above, we now assume that the functions a_i, b_i and c in (2.8) can be written as products

$$a_i(x,y) = a_i^x(x)a_i^y(y), \quad b_i(x,y) = b_i^x(x)b_i^y(y), \quad c(x,y) = c^x(x)c^y(y) \tag{3.1}$$

for univariate functions a_i^x, a_i^y, b_i^x, b_i^y and c^x, c^y. The idea of the finite-difference-method is to consider the PDE not for all $(x,y) \in G$ but only for a finite number of (equidistant) grid points (with $N_x, N_y \in \mathbb{N}^\times$ and $h_x = (x_r - x_l)/(N_x+1)$, $h_y = (y_r - y_l)/(N_y+1)$)

$$(x_i, y_j) \in \mathcal{G} := \{(x_l + ih_x, y_l + jh_y) \mid i = 0, \ldots, N_x+1, j = 0, \ldots, N_y+1\} \subset \overline{G}$$

and to replace at these grid points the partial derivatives by their corresponding finite difference quotients. To do so, we introduce the following difference-operators

$$\begin{aligned}
\delta_h^x f(x,y) &:= \frac{f(x+h,y) - f(x-h,y)}{2h} \\
\delta_k^y f(x,y) &:= \frac{f(x,y+k) - f(x,y-k)}{2k} \\
\delta_h^{2,x} f(x,y) &:= \frac{f(x-h,y) - 2f(x,y) + f(x+h,y)}{h^2} \\
\delta_k^{2,y} f(x,y) &:= \frac{f(x,y-k) - 2f(x,y) + f(x,y+k)}{k^2} \\
\delta_{h,k}^{x,y} f(x,y) &:= \frac{f(x-h,y-k) - f(x+h,y-k) - f(x-h,y+k) + f(x+h,y+k)}{4hk}
\end{aligned}$$

The PDE in (2.9) thus becomes with (3.1) at grid points (x_i, y_j), $i = 1, \ldots, N_x$, $j = 1, \ldots, N_y$,

$$\begin{aligned}
&\partial_t w(x_i,y_j,t) + a_1^x(x_i)a_1^y(y_j)\delta_{h_x}^{2,x} w(x_i,y_j,t) + a_2^x(x_i)a_2^y(y_j)\delta_{h_y}^{2,y} w(x_i,y_j,t) \\
&\quad + a_3^x(x_i)a_3^y(y_j)\delta_{h_x,h_y}^{x,y} w(x_i,y_j,t) + b_1^x(x_i)b_1^y(y_j)\delta_{h_x}^{x} w(x_i,y_j,t) \\
&\quad + b_2^x(x_i)b_2^y(y_j)\delta_{h_y}^{y} w(x_i,y_j,t) + c^x(x_i)c^y(y_j)w(x_i,y_j,t) = 0 + \mathcal{O}(h_x^2) + \mathcal{O}(h_y^2)\,.
\end{aligned}$$

By replacing $w(x_i,y_j,t)$ with $w_{i,j}(t)$, the above is equivalent to the linear system of $N := N_x N_y$ ODEs

$$\left\{ \begin{array}{rcl} \mathbf{w}'(t) + \mathbf{A}\mathbf{w}(t) &=& \mathbf{0} \\ \mathbf{w}(0) &=& \mathbf{g} \end{array} \right. , \tag{3.2}$$

where the ℓ-th component of the (column) vector $\mathbf{w}(t)$ is given by

$$w_\ell(t) = w_{(j-1)N_x+i}(t) = w_{i,j}(t)\ , \quad i = 1, \dots, N_x\ , \quad j = 1, \dots, N_y\ .$$

Similarly, the ℓ-th component of $\mathbf{w}'(t)$ is $\partial_t w_\ell(t) = \partial_t w_{(j-1)N_x+i}(t) = \partial_t w_{i,j}(t)$ and the ℓ-th component of $\mathbf{g}$ is $g_\ell = g_{(j-1)N_x+i} = g(x_i, y_j)$. Furthermore, if we set Dirichlet conditions on ∂G, i.e., $n_{x_l} = n_{x_r} = n_{y_l} = n_{y_r} = 0$ in (2.9), then the $N \times N$-matrix $\mathbf{A}$ in (3.2) is a sum of Kronecker products

$$\begin{aligned} \mathbf{A} \ := \ & \mathbf{M}^{(0)}_{a_1^y} \otimes \mathbf{M}^{(2)}_{a_1^x} + \mathbf{M}^{(2)}_{a_2^y} \otimes \mathbf{M}^{(0)}_{a_2^x} + \mathbf{M}^{(1)}_{a_3^y} \otimes \mathbf{M}^{(1)}_{a_3^x} \\ & + \mathbf{M}^{(0)}_{b_1^y} \otimes \mathbf{M}^{(1)}_{b_1^x} + \mathbf{M}^{(1)}_{b_2^y} \otimes \mathbf{M}^{(0)}_{b_2^x} + \mathbf{M}^{(0)}_{c^y} \otimes \mathbf{M}^{(0)}_{c^x}\ . \end{aligned} \tag{3.3}$$

For a function f and $k \in \{0, 1, 2\}$, the matrices $\mathbf{M}^{(k)}_f$ are defined in [Hil19]. For non-Dirichlet boundary conditions, we have to use non-centered finite-difference-quotients and the matrices $\mathbf{M}^{(k)}_f$ have to be changed to matrices ${}^r_l\mathbf{M}^{(k)}_f$. Here, the lower prescript $l \in \{n, s, i\}$ indicates the typ of the boundary condition on the left boundary of the corresponding interval and the upper prescript $r \in \{n, s, i\}$ specifies the condition on the right boundary of the interval. The letter n stands for a Neumann condition, the letter s means that the second derivative is specified, and i indicates that we solve the PDE also on the boundary (no boundary condition is needed in this case). The matrices ${}^r_l\mathbf{M}^{(k)}_f$ are defined in [Hil19].

Example 3.1. Consider the pricing equation for a plain vanilla option in the Jacobi-Heston model. We have seen in example 2.2 that if the condition (2.15) is violated, we need to specify conditions on the faces $]0, x_r[\times\{v_{\min}, v_{\max}\}\times]0, T[$. Suppose we set a homogeneous second derivative on these faces, i.e., $\partial_{vv} w(x, v_{\min}, t) = \partial_{vv} w(x, v_{\max}, t) = 0$. If we additionally set a homogenous Neumann condition on the set $\mathcal{B}_3 = \{x_r\}\times]v_{\min}, v_{\max}[\times]0, T[$, then, according to the generator $\mathcal{A}$ in (2.6), the matrix $\mathbf{A}$ in (3.3) becomes

$$\begin{aligned} \mathbf{A} \ := \ & \mathbf{M}^{(0)}_v \otimes {}_i\mathbf{M}^{(2)}_{-\frac{1}{2}x^2} + {}^s_s\mathbf{M}^{(2)}_{-\frac{1}{2}\delta^2 Q(v)} \otimes {}_i\mathbf{M}^{(0)}_1 + {}^s_s\mathbf{M}^{(1)}_{-\rho\delta Q(v)} \otimes {}_i\mathbf{M}^{(1)}_x \\ & + \mathbf{M}^{(0)}_1 \otimes {}_i\mathbf{M}^{(1)}_{-(r-q)x} + {}^s_s\mathbf{M}^{(1)}_{-\kappa(m-v)} \otimes {}_i\mathbf{M}^{(0)}_1 + \mathbf{M}^{(0)}_1 \otimes {}_i\mathbf{M}^{(0)}_r\ . \end{aligned}$$

3.2. **Time stepping.** The finite difference discretisation (with respect to space) of the pricing PDE (2.3) leads to the system (3.2) of ordinary differential equations satisfied by the $w_\ell(t)$. These functions approximate at the grid points (x_i, y_j) the value of the option under consideration, $w_\ell(t) \approx V(x_i, y_j, T - t)$. The system (3.2) has the closed form solution $\mathbf{w}(T) = e^{-\mathbf{A}T}\mathbf{g}$. Typically, the matrix $\mathbf{A}$ has a large dimension such that calculating the matrix exponential is infeasible. Thus, it is computationally (much) better to approximate $\mathbf{w}(T)$ by some time-stepping scheme. Applying the so-called Padé approximation of order 2 to e^{-x} (see e.g. [Tho06]), we end up with the Crank-Nicolson time stepping scheme as follows. Chose $M \in \mathbb{N}^\times$, let $t_j = jk$, $j = 0, 1, \dots, M$ with $k = \frac{T}{M}$ the time step. Set $\mathbf{w}_0 = \mathbf{g}$ and let $\mathbf{w}_j$ be an approximation to $\mathbf{w}(t_j)$. For $\theta = \frac{1}{2}$ and for $j = 0, 1, \dots, M - 1$ successively solve

the systems

$$\big(\mathbf{I} + k\theta\mathbf{A}\big)\mathbf{w}_{j+1} = \big(\mathbf{I} - k(1-\theta)\mathbf{A}\big)\mathbf{w}_j \ . \tag{3.4}$$

The ℓ-th component $w_{\ell,M}$ of the vector $\mathbf{w}_M$ is then an approximation to $V(x_i, y_j, 0)$. We will not apply (3.4), since the matrix $\mathbf{A}$ has a large bandwidth such that solving the corresponding M linear systems is slow.

A square matrix $\mathbf{M}$ has upper bandwidth q if $m_{i,j} = 0$ for $j > i + q$ and lower bandwidth p if $m_{i,j} = 0$ for $i > j + p$. The matrix $\mathbf{A}$ in (3.4) is a sum of Kronecker products, whence its bandwidths are $\mathcal{O}(N_x)$ (recall that N_x denotes the number of inner grid points in the interval $]x_l, x_r[$) with constants that depend on the boundary conditions. Indeed, let $\alpha := 1_{\{n_{x_l}=3\}} + 1_{\{n_{x_r}=3\}} \in \{0,1,2\}$ and define, for $n \in \{0,1,2,3\}$ the functions

$$n \mapsto \beta(n) := 1 + n - 1_{\{n>0\}} \in \{1,2,3\}, \quad n \mapsto \gamma(n) := 2 \cdot 1_{\{n\geq 2\}} + 1_{\{n<2\}} \in \{1,2\} \ .$$

Then, the matrix $\mathbf{A}$ has lower (p) and upper (q) bandwidth

$$\begin{aligned} p &\leq \max\{\beta(n_{y_r})(N_x+\alpha), \gamma(n_{y_r})(N_x+\alpha) + \gamma(n_{x_r})\} \\ q &\leq \max\{\beta(n_{y_l})(N_x+\alpha), \gamma(n_{y_l})(N_x+\alpha) + \gamma(n_{x_l})\} \end{aligned}$$

and so does the matrix $\mathbf{I} + k\theta\mathbf{A}$ (unless $\theta = 0$, and the scheme becomes explicit in this case). Clearly, solving a (sparse) linear system becomes the faster the smaller the bandwidths are. For example, if $\mathbf{M}$ is $n \times n$-matrix with $p, q \ll n$, then solving $\mathbf{Mx} = \mathbf{f}$ via LU factorization without pivoting requires about $2n(pq + p + q)$ flops, see [GVL13]. The idea of any so-called Alternating Direction Implicit (ADI) time-stepping scheme is to avoid solving systems with large bandwidths. To do so, we split the matrix $\mathbf{A}$ into the sum $\mathbf{A} = \mathbf{A}_0 + \mathbf{A}_1 + \mathbf{A}_2$ with

$$\begin{aligned} \mathbf{A}_0 &:= \mathbf{M}^{(1)}_{a_3^y} \otimes \mathbf{M}^{(1)}_{a_3^x} \\ \mathbf{A}_1 &:= \mathbf{M}^{(0)}_{a_1^y} \otimes \mathbf{M}^{(2)}_{a_1^x} + \mathbf{M}^{(0)}_{b_1^y} \otimes \mathbf{M}^{(1)}_{b_1^x} + \frac{1}{2}\mathbf{M}^{(0)}_{c^y} \otimes \mathbf{M}^{(0)}_{c^x} \\ \mathbf{A}_2 &:= \mathbf{M}^{(2)}_{a_2^y} \otimes \mathbf{M}^{(0)}_{a_2^x} + \mathbf{M}^{(1)}_{b_2^y} \otimes \mathbf{M}^{(0)}_{b_2^x} + \frac{1}{2}\mathbf{M}^{(0)}_{c^y} \otimes \mathbf{M}^{(0)}_{c^x} \ . \end{aligned} \tag{3.5}$$

We now observe that the matrix $\mathbf{A}_1$ has bandwidths which are independent of N_x and are given by $p \leq \beta(n_{x_r})$ and $q \leq \beta(n_{x_l})$. Furthermore, the matrix $\mathbf{A}_2$ can be made small-banded by some suitable permutation, see below and figure 1. Only the matrix $\mathbf{A}_0$ is not and can not be made small-banded, such that any ADI scheme treats this term explicit.

Classical ADI schemes are the Douglas scheme, the (modified) Craig-Sneyd scheme and the scheme proposed by Hundsdorfer and Verwer (HV), which we apply in the present situation. For a description and a stability analysis of the mentioned schemes see [I'HM13]. The HV scheme reads as follows: For $\mathbf{w}_0 = \mathbf{g}$ and $j = 0, \ldots, M-1$

do

$$
\left\{\begin{array}{rcll}
\mathbf{y}_0 &=& \big(\mathbf{I}-k\mathbf{A}\big)\mathbf{w}_j & \\
\big(\mathbf{I}+k\theta\mathbf{A}_i\big)\mathbf{y}_i &=& \mathbf{y}_{i-1}+k\theta\mathbf{A}_i\mathbf{w}_j, & i=1,2\\
\mathbf{z}_0 &=& \mathbf{y}_0-\frac{1}{2}k\mathbf{A}(\mathbf{y}_2-\mathbf{w}_j) & \\
\big(\mathbf{I}+k\theta\mathbf{A}_i\big)\mathbf{z}_i &=& \mathbf{z}_{i-1}+k\theta\mathbf{A}_i\mathbf{y}_2, & i=1,2\\
\mathbf{w}_{j+1} &=& \mathbf{z}_2 &
\end{array}\right. \tag{3.6}
$$

Note that the two systems involving the matrix $\mathbf{B}_1 := \mathbf{I}+k\theta\mathbf{A}_1$ are small-banded and their solution can be therefore calculated efficiently. The matrix $\mathbf{B}_2 := \mathbf{I}+k\theta\mathbf{A}_2$, however, is still large-banded with $p \le \beta(n_{y_r})(N_x+\alpha)$ and $q \le \beta(n_{y_l})(N_x+\alpha)$. As an example, consider the Jacobi-Heston model from example 3.1. Herein, we have $n_{x_l}=3$, $n_{x_r}=1$ (such that $\alpha=1$) and $n_{y_l}=n_{y_r}=2$ and the matrix

$$
\mathbf{A}_2 = \mathbf{M}^{(2)}_{-\frac{1}{2}\delta^2 Q(v)} \otimes \mathbf{M}^{(0)}_1 + \mathbf{M}^{(1)}_{-\kappa(m-v)} \otimes \mathbf{M}^{(0)}_1 + \frac{1}{2}\mathbf{M}^{(0)}_1 \otimes \mathbf{M}^{(0)}_r
$$

has $p=q=\beta(2)(N_x+1)=2(N_x+1)$, compare also with figure 1. A typical summand of $\mathbf{B}_2$ (respectively $\mathbf{A}_2$) is of the form $\mathbf{Y}\otimes\mathbf{X}$, where $\mathbf{Y}$ denotes a matrix in the y-coordinate direction and $\mathbf{X}$ denotes a matrix in the x-coordinate direction. Whereas the matrix $\mathbf{Y}\otimes\mathbf{X}$ is not small-banded, the matrix $\mathbf{X}\otimes\mathbf{Y}$ is, and we try therefore to solve a system involving the matrix $\mathbf{X}\otimes\mathbf{Y}$ instead of $\mathbf{Y}\otimes\mathbf{X}$. This is possible since for matrices $\mathbf{X}\in\mathbb{R}^{n\times n}$ and $\mathbf{Y}\in\mathbb{R}^{m\times m}$ there exists a perfect shuffle matrix $\mathbf{P}^{nm\times nm}$ with $\mathbf{P}^\top\mathbf{P}=\mathbf{P}\mathbf{P}^\top=\mathbf{I}$ and such that

$$
\mathbf{Y}\otimes\mathbf{X} = \mathbf{P}(\mathbf{X}\otimes\mathbf{Y})\mathbf{P}^\top .
$$

Using these properties of $\mathbf{P}$, it is easy to see that solving $(\mathbf{Y}\otimes\mathbf{X})\mathbf{w}=\mathbf{f}$ for $\mathbf{w}$ is equivalent to solving $(\mathbf{X}\otimes\mathbf{Y})\widetilde{\mathbf{w}}=\mathbf{P}^\top\mathbf{f}$ for $\widetilde{\mathbf{w}}$ and then setting $\mathbf{w}=\mathbf{P}\widetilde{\mathbf{w}}$. In the HV scheme (3.6), the matrix $\mathbf{A}_2$ from (3.5) need thus to be replaced by the matrix

$$
\widetilde{\mathbf{A}}_2 = \mathbf{M}^{(0)}_{a_2^x} \otimes \mathbf{M}^{(2)}_{a_2^y} + \mathbf{M}^{(0)}_{b_2^x} \otimes \mathbf{M}^{(1)}_{b_2^y} + \frac{1}{2}\mathbf{M}^{(0)}_{c^x} \otimes \mathbf{M}^{(0)}_{c^y} . \tag{3.7}
$$

For example, the matrix $\widetilde{\mathbf{A}}_2$ corresponding to the Heston-Jacobi model has $p=q=2$. For the definition and calculation of the matrix $\mathbf{P}$ we refer, for instance, to [HS81].

Hence, in each time step of (3.6), we do not solve one large-banded system as in (3.4), but four small-banded systems (two involving $\mathbf{A}_1$, two involving $\widetilde{\mathbf{A}}_2$). The latter is by factors faster than the former.

Whereas Matlab's `mldivide` (the backslash operator) recognises the structure of the sparse matrix and uses then a (sparse) banded solver, Python does not seem to provide any solver for sparse banded systems. As a consequence, we use Python's `solve_banded`, which however requires the extraction of all the non-zero k-th diagonals of the matrices $\mathbf{A}_1$ and $\widetilde{\mathbf{A}}_2$, compare with the Python code below.

We denote by $w_{\ell,M}$ the ℓ-th component of the vector $\mathbf{w}_M$. If the payoff function g is continuous, the fully discrete scheme (3.6) yields approximations $w_{\ell,M}$ to the solution $w(x_i,y_j,T)$, $\ell=(j-1)N_x+i$, of the PDE (2.9) satisfying

$$
e_{h,k} := |w_{\ell,M}-w(x_i,y_j,T)| = \mathcal{O}(h_x^2)+\mathcal{O}(h_y^2)+\mathcal{O}(k^2) .
$$

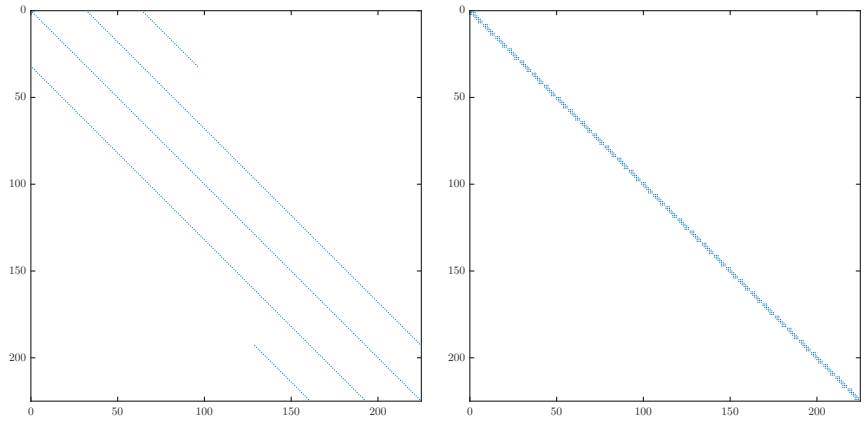

FIGURE 1. Sparsity pattern of the matrices $\mathbf{A}_2$ (left) and $\widetilde{\mathbf{A}}_2 = \mathbf{P}^\top \mathbf{A}_2 \mathbf{P}$ (right) for the Jacobi-Heston model. For better visibility, we use $N_x = 31$ inner grid points in $]0, x_r[$ and $N_y = 7$ inner grid points in $]v_{\min}, v_{\max}[$. Due to the chosen boundary conditions, these matrices have dimension $(N_x+1)N_y \times (N_x+1)N_y = 224 \times 224$; only 672 entries are non zero. For the matrix $\mathbf{A}_2$ the bandwidths are $p = q = 2(N_x + 1) = 64$; for the matrix $\widetilde{\mathbf{A}}_2$ there holds $p = q = 2$. In the HV scheme (3.6), we avoid therefore solving linear systems involving $\mathbf{A}_2$.

In terms of the total number of unknowns $N = N_x N_y$ there holds $e_{h,k} = \mathcal{O}(N^{-1})$ (if we let the number of time steps $M = \mathcal{O}(\sqrt{N})$). More generally, the above described fully discrete scheme of order two applied to derivatives pricing problems involving d stochastic factor yields convergence rates at most

$$e_{h,k} = \mathcal{O}(N^{-\frac{2}{d}}) \, .$$

The curse of dimensionality is clearly visible.

4. CODE

The considerations of the previous sections lead to the following pseudo-code to find the approximative value of a two-factor derivative. Note/recall that $n = 0$ indicates a Dirichlet boundary condition, $n = 1$ a Neumann boundary condition, $n = 2$ the second derivative and $n = 3$ no boundary condition. The input parameters to the routine are the problem dependent "parameters" given by the coefficients $a_i^x(x), a_i^y(y)$, $i = 1, 2, 3$, $b_i^x(x), b_i^y(y)$, $i = 1, 2$ and $c^x(x)c^y(y)$ of the PDE, the initial condition $g(x, y)$, the domain $G =]x_l, x_r[\times]y_l, y_r[$, and the type of boundary conditions (via $n_{x_l}, n_{x_r} n_{y_l}, n_{y_r}$). Furthermore, we have to specify the discretisation parameters N_x, N_y (number of (inner) grid points in $]x_l, x_r[$ and $]y_l, y_r[$, respectively), M (number of time steps) and $\theta \in [0, 1]$ in the HV scheme (3.6). The perfect shuffle matrix $\mathbf{P}$ is also taken as an input. The routine called `pricing_2d` then returns the

array of all grid points $(x_i, y_j) \in \mathcal{G}$ as well as the functions values $w(x_i, y_j, T)$ of $w(x, y, t)$ at these grid points at time $t = T$.

Define the functions a_i^x, a_i^y, b_i^x, b_i^y, c^x, c^y, and g. Define $T > 0$.
Define $x_l, x_r, y_l, y_r \in \mathbb{R}$, $n_{x_l}, n_{x_r}, n_{y_l}, n_{y_r} \in \{0, 1, 2, 3\}$.
Chose $N_x, N_y, M \in \mathbb{N}^{\times}$. Chose $\theta \in [0, 1]$. Set $k := T/M$.
Get the matrices $\mathbf{A}_i$, $i = 0, 1, 2$, in (3.5), and the matrix $\widetilde{\mathbf{A}}_2$ in (3.7),
    boundary conditions already incorporated.
Define the matrices $\mathbf{A} := \mathbf{A}_0 + \mathbf{A}_1 + \mathbf{A}_2$, $\mathbf{B} := \mathbf{I} - k\mathbf{A}$.
Define the matrices $\mathbf{B}_0 := k/2\mathbf{A}$, $\mathbf{B}_1 := \mathbf{I} + \theta k\mathbf{A}_1$, $\mathbf{B}_2 := \mathbf{I} + \theta k\widetilde{\mathbf{A}}_2$.
Define the matrices $\mathbf{C}_i := k\theta\mathbf{A}_i$, $i = 1, 2$. Set $\mathbf{w}_0 := \mathbf{g}$.

For $j = 0, \ldots, M - 1$,
    Set $\mathbf{y}_0 := \mathbf{B}\mathbf{w}_j$. Solve the system $\mathbf{B}_1\mathbf{y}_1 = \mathbf{y}_0 + \mathbf{C}_1\mathbf{w}_j$.
    Solve the system $\mathbf{B}_2\widetilde{\mathbf{y}}_2 = \mathbf{P}^{\top}(\mathbf{y}_1 + \mathbf{C}_1\mathbf{w}_j)$. Set $\mathbf{y}_2 := \mathbf{P}\widetilde{\mathbf{y}}_2$.
    Set $\mathbf{z}_0 := \mathbf{y}_0 + \mathbf{B}_0(\mathbf{w}_j - \mathbf{y}_2)$. Solve the system $\mathbf{B}_1\mathbf{z}_1 = \mathbf{z}_0 + \mathbf{C}_1\mathbf{y}_2$.
    Solve the system $\mathbf{B}_2\widetilde{\mathbf{z}}_2 = \mathbf{P}^{\top}(\mathbf{z}_1 + \mathbf{C}_2\mathbf{y}_2)$. Set $\mathbf{w}_{j+1} := \mathbf{P}\widetilde{\mathbf{z}}_2$.

Output the array $\mathcal{G}$ of grid points and the function values $\mathbf{w}_M$
at these grid points.

TABLE 1. Description of the fully discrete scheme to solve the PDE (2.9)

In Matlab/Octave, the pseudo code may thus take the following form. The subroutine `matgen` outputs for any of the discussed boundary conditions the matrices ${}_l^r\mathbf{M}_f^{(k)}$, see the appendix A.2.

```
function [x,y,w] = pricing_2d(a,b,c,T,g,G,BC,N,M,P,theta)

% some aux vars
xl = G(1); xr = G(2); yl = G(3); yr = G(4);
hx = (xr-xl)/(N(1)+1); hy = (yr-yl)/(N(2)+1); k = T/M;
nxl = BC(1); nxr = BC(2); x = (xl-(nxl==3)*hx:hx:xr+(nxr==3)*hx)';
nyl = BC(3); nyr = BC(4); y = (yl-(nyl==3)*hy:hy:yr+(nyr==3)*hy)';

% the matrices
[M2a1x,M1a3x,M1b1x,M0a2x,M0b2x,M0cx] = matgen({{'M2',a{1}},{'M1',a{5}},...
    {'M1',b{1}},{'M0',a{3}},{'M0',b{3}},{'M0',c{1}}},BC(1:2),xl,xr,N(1));
[M2a2y,M1a3y,M1b2y,M0a1y,M0b1y,M0cy] = matgen({{'M2',a{4}},{'M1',a{6}},...
    {'M1',b{4}},{'M0',a{2}},{'M0',b{2}},{'M0',c{2}}},BC(3:4),yl,yr,N(2));

A1 = kron(M0a1y,M2a1x)+kron(M0b1y,M1b1x)+1/2*kron(M0cy,M0cx);
A2 = kron(M2a2y,M0a2x)+kron(M1b2y,M0b2x)+1/2*kron(M0cy,M0cx);
A2t = kron(M0a2x,M2a2y)+kron(M0b2x,M1b2y)+1/2*kron(M0cx,M0cy);
A0 = kron(M1a3y,M1a3x);

A = A0+A1+A2; I = speye(length(A)); B = I-k*A; B0 = 0.5*k*A; PT = P';
B1 = I+theta*k*A1; B2 = I+theta*k*A2t; C1 = k*theta*A1; C2 = k*theta*A2;

% initial condition
[x,y] = ndgrid(x(2:N(1)+1+(nxl==3)+(nxr==3)),y(2:N(2)+1+(nyl==3)+(nyr==3)));
w = g(x,y); w = w(:);

% HV time-stepping
for j = 1:M
    y0 = B*w; y1 = B1\(y0+C1*w); y2 = B2\(PT*(y1+C2*w)); y2 = P*y2;
    z0 = y0+B0*(w-y2); z1 = B1\(z0+C1*y2); w = B2\(PT*(z1+C2*y2)); w = P*w;
end
w = reshape(w,N(1)+(nxl==3)+(nxr==3),N(2)+(nyl==3)+(nyr==3));
```

The corresponding Python code is (significantly) longer, since `solve_banded` requires not the matrices $\mathbf{A}_1$ and $\widetilde{\mathbf{A}}_2$ directly, but all their non-zero diagonals. The extraction of those is provided by the sub-function `get_diagonals`.

```
import numpy as np; from matgen import matgen;
from scipy import sparse; from scipy.linalg import solve_banded

def get_diagonals(B,nl,nr):
    bp1 = np.concatenate((np.asarray([0]),np.diag(B,1))); b0 = np.diag(B,0)
    bm1 = np.concatenate((np.diag(B,-1),np.asarray([0])))
    M = np.vstack((bp1,b0,bm1))

    if nl>1: # upper
        for j in range(1,nl):
            b = np.concatenate((np.zeros(j+1),np.diag(B,j+1))); M = np.vstack
    ((b,M))
```

```
12
13      if nr>1: # lower
14          for j in range(1,nr):
15              b = np.concatenate((np.diag(B,-j-1),np.zeros(j+1))); M = np.
        vstack((M,b))
16
17      return M
18
19  def pricing_2d(a,b,c,T,g,G,BC,N,M,P,theta):
20
21      # some aux vars
22      xl = G[0]; xr = G[1]; yl = G[2]; yr = G[3]
23      hx = (xr-xl)/(N[0]+1); hy = (yr-yl)/(N[1]+1); k = T/M
24      nxl = BC[0]; nxr = BC[1]; x = np.arange(xl-(nxl==3)*hx,xr+hx+(nxr==3)*hx,
        hx)
25      nyl = BC[2]; nyr = BC[3]; y = np.arange(yl-(nyl==3)*hy,yr+hy+(nyr==3)*hy,
        hy)
26      beta = lambda n:1+n-(n>0);
27
28      # the matrices
29      Matx = matgen([["M2",a[0]],["M1",a[4]],["M1",b[0]],["M0",a[2]],
30                     ["M0",b[2]],["M0",c[0]]],[nxl,nxr],xl,xr,N[0])
31      Maty = matgen([["M2",a[3]],["M1",a[5]],["M1",b[3]],["M0",a[1]],
32                     ["M0",b[1]],["M0",c[1]]],[nyl,nyr],yl,yr,N[1])
33
34      A1 = (sparse.kron(Matx[0],Maty[3])+sparse.kron(Matx[2],Maty[4])+
35            0.5*sparse.kron(Matx[5],Maty[5]))
36      A1t = (sparse.kron(Maty[3],Matx[0])+sparse.kron(Maty[4],Matx[2])+
37            0.5*sparse.kron(Maty[5],Matx[5]))
38      A2 = (sparse.kron(Matx[3],Maty[0])+sparse.kron(Matx[4],Maty[2])+
39            0.5*sparse.kron(Matx[5],Maty[5]))
40      A0 = sparse.kron(Matx[1],Maty[1])
41
42      I = sparse.eye((N[0]+(nxr==3)+(nxl==3))*(N[1]+(nyr==3)+(nyl==3)))
43      A = A0+A1+A2; B1 = I+k*theta*A1t; B2 = I+k*theta*A2; B0 = 0.5*k*A
44      B = I-k*A; C1 = k*theta*A1; C2 = k*theta*A2; PT = P.T;
45      B1 = get_diagonals(B1.A,nxl,nxr); B2 = get_diagonals(B2.A,nyl,nyr)
46
47      # initial condition
48      x = np.linspace(xl+(1-(nxl==3))*hx,xr-(1-(nxr==3))*hx,N[0]+(nxl==3)+(nxr
        ==3))
49      y = np.linspace(yl+(1-(nyl==3))*hy,yr-(1-(nyr==3))*hy,N[1]+(nyl==3)+(nyr
        ==3))
50      y,x = np.meshgrid(y,x); w = g(x,y); w = w.flatten('C')
51
52      #HV scheme
53      for j in range(0,M):
54          y0 = B*w; y1 = solve_banded((beta(nxr),beta(nxl)),B1,PT*(y0+C1*w));
55          y1 = P*y1; y2 = solve_banded((beta(nyr),beta(nyl)),B2,y1+C2*w);
56          z0 = y0+B0*(w-y2); z1 = solve_banded((beta(nxr),beta(nxl)),B1,PT*(z0+
        C1*y2));
57          z1 = P*z1; w = solve_banded((beta(nyr),beta(nyl)),B2,z1+C2*y2)
```

```
58
59      w = np.reshape(w,(N[0]+(nxl==3)+(nxr==3),N[1]+(nyl==3)+(nyr==3)))
60
61      return x,y,w
```

5. Examples

In this section, we give examples which fit into framework of the pricing problem (2.3). For each of these, we provide (uncommented) Matlab and Python codes. Whereas both languages yield the same prices, Matlab is typically two to three times faster than Python.

5.1. **European call in the Jacobi-Heston model.** We consider a European call option with strike K and maturity T in the Jacobi-Heston model, compare with example 2.1. We solve the PDE on the domain $[0, x_r[\times]v_{\min}, v_{\max}[$, hence we set no boundary condition on the set $\{0\}\times]v_{\min}, v_{\max}[$. We assume that the condition (2.15) is not satisfied such that we need to specify boundary conditions. We chose homogeneous second derivatives on $[0, x_r[\times\{v_{\min}, v_{\max}\}$, i.e., $n_{y_l} = n_{y_r} = 2$ in (2.9). Additionally, we consider a homogeneous Neumann condition on $\{x_r\}\times]v_{\min}, v_{\max}[$, i.e., $n_{x_r} = 1$.

The function `calljacobiheston` realises this in Matlab/Octave (note that $n_{x_l} = 3$; no boundary condition)

```
function V = calljacobiheston(x0,y0,K,T,r,q,param)

delta = param(1);  kappa = param(2); m = param(3);
vmin = param(4); vmax = param(5); rho = param(6);

L = [10,5]; N = 2.^L-1; M = ceil(0.05*max(N));
Q = @(v)(v-vmin).*(vmax-v)/(sqrt(vmax)-sqrt(vmin))^2;
a = {@(x)-0.5*x.^2,@(y)y,@(x)-0.5*delta^2*x.^0,@(y)Q(y),@(x)-rho*delta*x,@(y)
    Q(y)};
b = {@(x)-(r-q)*x,@(y)y.^0,@(x)x.^0,@(y)-kappa*(m-y)}; c = {@(x)r*x.^0,@(y)y
    .^0};
g = @(x,y)max(x-K,0).*y.^0; G = [0 4*K vmin vmax];

BC = [3 1 2 2]; P = perm_matrix(N(2)+(BC(3)==3)+(BC(4)==3),...
    N(1)+(BC(1)==3)+(BC(2)==3));
[x,y,w] = pricing_2d(a,b,c,T,g,G,BC,N,M,P,1);

V = interpn(x,y,w,x0,y0);
```

respectively in Python

```
import numpy as np; from pricing_2d import pricing_2d
from scipy.interpolate import interpn; from perm_matrix import perm_matrix

def calljacobiheston(x0,y0,K,T,r,q,param):

    delta = param[0];  kappa = param[1]; m = param[2];
    vmin = param[3]; vmax = param[4]; rho = param[5];

```

```
    L = np.asarray([10,5]); N = 2**L-1; M = np.int(np.ceil(0.05*max(N)));
    Q = lambda v: (v-vmin)*(vmax-v)/(np.sqrt(vmax)-np.sqrt(vmin))**2;

    a = [lambda x:-0.5*x**2,lambda y:y,lambda x:-0.5*delta**2*x**0,
         lambda y:Q(y),lambda x:-rho*delta*x,lambda y:Q(y)];
    b = [lambda x:-(r-q)*x,lambda y:y**0,lambda x:x**0,lambda y:-kappa*(m-y)
    ];
    c = [lambda x:r*x**0,lambda y:y**0];
    g = lambda x,y: np.maximum(x-K,0)*y**0; G = [0,4*K,vmin,vmax];

    BC = [3,1,2,2];
    P = perm_matrix(N[0]+(BC[0]==3)+(BC[1]==3),N[1]+(BC[2]==3)+(BC[3]==3))

    [x,y,w] = pricing_2d(a,b,c,T,g,G,BC,N,M,P,1)

    return interpn((x.T[0],y[0]),w,(x0,y0))
```

Now consider the particular example of a call option with strike $K \in \{e^{-0.1}, e^0, e^{0.1}\}$ and maturity $T = 1/12$ written on a stock with initial price $x_0 = 1$ which pays no dividend, $q = 0$. The model parameters are chosen to be

$$\{\delta, \kappa, m, v_{\min}, v_{\max}, \rho\} = \{1, 0.5, 0.04, 10^{-4}, 0.08, -0.5\} .$$

Note that for this choice of the parameter values the condition (2.15) is not satisfied. The risk free is $r = 0$. The function `calljacobiheston` then gives call prices $V \in \{0.0969003, 0.0221449, 0.0008354\}$. In [AFP18], the authors state the corresponding (model) implied volatilities $\sigma^{\text{impl}} \in \{22.75\%, 19.23\%, 19.25\%\}$, which lead to option prices $V \in \{0.0969001, 0.0221433, 0.0008347\}$ via the Black-Scholes formula. We remark that Matlab finds the option values about 3 times faster than Python.

5.2. **Discretely monitored Lookback option in the CEV model.** For $t \geq 0$, let $\mathcal{S}_t$ be a subset of $\mathbb{R}_0^+$ with $\mathcal{S}_t \subset \mathcal{S}_s$ for $t < s$. Let $X(t)$, $t \geq 0$, be a stochastic process with continuous paths. We define the maximum and minimum process of X as follows

$$X_{\max}(t) := \max_{\tau \in \mathcal{S}_t} X(\tau), \quad X_{\min}(t) := \min_{\tau \in \mathcal{S}_t} X(\tau) ,$$

For continuous monitoring, we have $\mathcal{S}_t = [0, t]$, for discrete monitoring, there holds $\mathcal{S}_t = \{t_j \in \mathcal{T} \mid t_j \leq t\}$ for some (finite) set $\mathcal{T} := \{t_0, t_1, t_2, \ldots, t_J\}$ of observation dates t_j with $t_0 = 0$ and $t_j < t_{j+1}$. The payoff of a fixed strike lookback option with strike K and maturity T is

$$\max\{X_{\max}(T) - K, 0\}$$

if it is a call and

$$\max\{K - X_{\min}(T), 0\}$$

if it is a put. The value of a (call) option depends on the vector process $\mathbf{X}(t) = (X(t), X_{\max}(t))^\top$;

$$V(x, m, t) = \mathbb{E}^{\mathbb{Q}}[e^{-r(T-t)} g(X_{\max}(T)) \mid (X(t), X_{\max}(t)) = (x, m)] .$$

If the monitoring is discrete, then V is the solution of the following sequence of $J+1$ PDEs. Let $t_0 = 0$ and $t_{J+1} = T$. For $j = J+1, J, \ldots, 1$ successively solve

$$\left\{ \begin{array}{rcll} \partial_t V_j + \mathcal{A} V_j - r V_j & = & 0 & \text{in } G \times [t_{j-1}, t_j[\\ V_j(x,m,t_j) & = & V_{j+1}(x, f(x,m), t_j) & \text{in } G \end{array} \right. \tag{5.1}$$

with $V_{J+2}(x, f(x,m), t_{J+1}) = g(s,m)$ and domain $G = \mathbb{R}^+ \times \mathbb{R}^+$, see for example [ZWCS13]. The price of the lookback option at inception is then $V_1(x_0, x_0, 0)$. Note carefully the different role played by the variable m. For a put, m denotes the running minimum of X whereas for a call it denotes the maximum of X. Hence, the function f appearing in the continuitiy condition $V_j(x,m,t_j) = V_{j+1}(x, f(x,m), t_j)$ at t_j is $f(x,m) = \min\{x,m\}$ for a put and $f(x,m) = \max\{x,m\}$ for a call. The operator $\mathcal{A}$ is the infinitesimal generator of $X(t)$; if we assume that the dynamics of X follow the CEV model, there holds $\mathcal{A} = \frac{1}{2}\delta^2 x^{2\beta}\partial_{xx} + (r-q)x\partial_x$, where $\delta > 0$ and $\beta \in \mathbb{R}$ are model parameters. To solve each PDE in (5.1) with `pricing_2d`, we switch to time-to-maturity and restrict to the domain $]0, x_r[\times]0, m_r[$; on all four faces of this domain we chose a homogeneous second derivative. We take $N_x = N_y = 2^{10} - 1$ (inner) grid points in each coordinate direction and $M = \lceil 2/JN_x \rceil$ time steps with $\theta = 1$ in the HV scheme. Note that since $\mathcal{A}$ operates in the x-coordinate direction only, the matrix $\mathbf{A}_0$ in the splitting (3.5) is the zero-matrix and $\mathbf{A}_2$ is diagonal. Hence, an application of the HV scheme is not necessary in this particular case and it would be sufficient to apply the Crank-Nicolson time stepping (3.4).

The function `lookbackcev` below gives the price of a fixed strike lookback put. `Tau` is array containing the observation dates $\mathcal{T}$.

```
function V = lookbackcev(x0,beta,delta,r,q,T,K,Tau)

J = length(Tau); Tau = [0,Tau]; tau = diff(Tau); G = [0 2 0 2]*x0;
a = {@(x)-0.5*delta^2*x.^(2*beta),@(y)y.^0,@(x)0*x,@(y)0*y,@(x)0*x,@(y)0*y};
b = {@(x)-(r-q)*x,@(y)y.^0,@(x)0*x,@(y)0*y}; c = {@(x)r*x.^0,@(y)y.^0};
g = @(x,y)max(K-y,0); BC = [2 2 2 2]; f = @(x,y)min(x,y);

L = [9,9]; N =2.^L-1; M = ceil(1/J*max(N));
P = perm_matrix(N(2)+(BC(3)==3)+(BC(4)==3),N(1)+(BC(1)==3)+(BC(2)==3));

[x,y,w] = pricing_2d(a,b,c,T-Tau(end),g,G,BC,N,M,P,1);
for j = 1:J
    g = @(x,y)interpn(x,y,w,x,f(x,y));
    [x,y,w] = pricing_2d(a,b,c,tau(end+1-j),g,G,BC,N,M,P,1);
end

V = interpn(x,y,w,x0,x0);
```

and in Python

```
import numpy as np; from pricing_2d import pricing_2d
from scipy.interpolate import interpn; from perm_matrix import perm_matrix

def lookbackcev(x0,beta,delta,r,q,T,K,Tau):

    J = len(Tau); Tau = np.hstack((0,Tau)); tau = np.diff(Tau);
```

```
    G = np.asarray([0,2,0,2])*x0;
    a = [lambda x:-0.5*delta**2*x**(2*beta),lambda y:y**0,lambda x:0*x,lambda
     y:0*y,lambda x:0*x,lambda y:0*y];
    b = [lambda x:-(r-q)*x,lambda y:y**0,lambda x:0*x,lambda y:0*y];
    c = [lambda x:r*x**0,lambda y:y**0];
    g = lambda x,y:np.maximum(K-y,0); BC = [2,2,2,2];
    f = lambda x,y: np.minimum(x,y);

    L = np.asarray([10,10]); N = 2**L-1; M = np.int(np.ceil(1/J*max(N)));
    P = perm_matrix(N[0]+(BC[0]==3)+(BC[1]==3),N[1]+(BC[2]==3)+(BC[3]==3)).
    tocsr();

    [x,y,w] = pricing_2d(a,b,c,T-Tau[-1],g,G,BC,N,M,P,1);
    for j in range(1,J+1):
        g = lambda x,y: interpn((x.T[0],y[0]),w,(x,f(x,y)))
        [x,y,w] = pricing_2d(a,b,c,tau[J-j],g,G,BC,N,M,P,1);

    return interpn((x.T[0],y[0]),w,(x0,x0))
```

We now consider the particular case of J equidistant observation dates $t_j = \frac{j}{J}T$ and a put with strike $K = 105$ and maturity $T = 0.5$. The underlying has value $s_0 = 100$ and pays no dividend, $q = 0$. The CEV parameters are set to $\beta = 0.5$ and $\delta = 0.25/(s_0^{\beta-1})$, the risk free is $r = 0.1$. For these values and the number J of observation dates $J \in \{52, 104, 252, 504, 1008\}$ the function `lookbackcev` finds the option prices $V(x_0, x_0, 0) \doteq \{14.5402, 14.8829, 15.1860, 15.3480, 15.4602\}$. In [SMF14], the authors state the values $V(x_0, x_0, 0) = \{14.5430, 14.8864, 15.1910, 15.3542, 15.4709\}$. We do not compare computation times for this example, since the interpolation required by the update $V_j(x, m, t_j) = V_{j+1}(x, f(x, m), t_j)$ is slow in Python.

5.3. **Autocallable Multi Barrier Reverse Convertible in the BS model.** Multi Barrier Reverse Convertibles (MBRC) are structures products written on $d > 1$ underlyings. There are different versions of MBRCs available on the market (ordinary, callable, autocallable), in this example we focus on autocallable MBRCs. The specification of the redemption of such a MRBC is as follows. If there was no early redemption, then the holder of the product (with nominal N) receives (at the redemption date) the cash flow of the corresponding ordinary MRBC

$$g = C(J) + N - N \max\left\{1 - \min_i \frac{X_i(T)}{K_i}, 0\right\} 1_{\{\exists i \in \{1,2,\dots,d\} | \min_{t\in]0,T]} X_i(t) \le B_i\}}\ , \tag{5.2}$$

where $\mathbf{X}(t) = (X_1(t), \dots, X_d(t))^\top$ denotes the price of all involved underlyings $X_i(t)$ at time t, K_i is the strike price of the i-th underlying (typically, $K_i = X_i(0)$), and $B_i < X_i(0)$ is the barrier of the i-the underlying, $i = 1, \dots, d$. The indicator function $1_{\{\cdot\}}$ returns the value 1 if any of the underlyings hits its barrier (from above) during the period $]0, T]$, and zero else. In (5.2), $C(J)$ denotes the value at time T of the coupon payments (with size C) the investor receives at the coupon payment dates

$t_j^C \in \mathcal{T}^C := \{t_1^C, t_2^C, \ldots, t_J^C\}$, i.e.,

$$C(J) := C \sum_{j=1}^{J} e^{r(T-t_j^C)} \,. \tag{5.3}$$

If an early redemption occurs at a (pre-defined) autocall observation date t_h^A, the product expires immediately and the holder receives (at the next corresponding coupon payment date t_j^C) the denomination plus the coupon amount, $N + C$. Typically, there are $J \geq H > 1$ autocall observation dates $\mathcal{T}^A := \{t_1^A, t_2^A, \ldots, t_H^A\}$ and the last of these dates is equal to the maturity of the product, $T = t_H^A$. Furthermore, there are a few days between the observation dates t_h^A and the following coupon payment date t_{J-H+h}^C and the time spans $\delta_h := t_h^R - t_{J-H+h}^A$ have to be taken into account when it comes to the pricing of the product. An early redemption occurs if all underlyings at the autocall observation date t_h^A are at or above their early redemption level ℓ_i, i.e., if $X_i(t_h^A) \geq \ell_i$, $\forall i$. Usually, $\ell_i = X_i(0) = K_i$. We collect the early redemption levels in the vector $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_d)$.

In the following, assume $\ell_i = K_i$, $\forall i$. To understand the pricing of the product via PDEs, assume for a brief moment that there is only one observation date t_1^A before maturity, i.e., $H = 2$. If at $t_2^A = T$ all the underlyings are at or above the early redemption level, the investor receives $C + N$. Since the investor did also receive all previous coupon payments (at dates t_j^C), the time T-value of all the received cash flows is $C(J) + N$. Note that since $\ell_i = K_i$, $\min_i X_i(T)/K_i \geq 1$ in this case and the cash flow $C(J) + N$ is equal to g in (5.2). If at t_2^A at least one of the underylings is below the early redemption level, the investor receives at t_J^C the cash flow g in (5.2). Hence, no matter which value $X_i(T)$ the underlyings may take at maturity, the payoff function is equal to $g^2 := g$ in (5.2). Thus, in the time interval $t \in [t_1^A, T[$, we solve the PDE $\partial_t V^2 + \mathcal{A}V^2 - rV^2 = 0$ with the terminal condition $V^2(\mathbf{x}, T) = g^2(\mathbf{x})e^{-r\delta_2}$ for the unknown value function $V^2(\mathbf{x}, t)$. To apply the pricing function `pricing_2d`, we need to split the payoff g^2. Since

$$1_{\{\exists i \in \{1,2,\ldots,d\} | \min_{t\in]0,T]} X_i(t) \leq B_i\}} = 1 - 1_{\{\forall i \in \{1,2,\ldots,d\} | \min_{t\in]0,T]} X_i(t) > B_i\}}$$

we can write $g^2 = g_1^2 - (g_2^2 - \widetilde{g}_3^2)$, where

$$\begin{aligned} g_1^2 &:= C(J) + N \\ g_2^2 &:= N \max\left\{1 - \min_i \frac{X_i(T)}{K_i}, 0\right\} \\ \widetilde{g}_3^2 &:= g_2^2 1_{\{\forall i \in \{1,2,\ldots,d\} | \min_{t\in]0,T]} X_i(t) > B_i\}} \,. \end{aligned}$$

The options with payoffs g_1^2 and g_2^2 are of European style and the corresponding pricing equations have to be solved on the domain $G_i = \mathbb{R}^+ \times \ldots \times \mathbb{R}^+$, $i = 1, 2$. The option with payoff $\widetilde{g}_3^2$ is a (multivariate version of a) down-and-out barrier option; it becomes worthless if at least one of the underlyings hits its barrier from above ($X_i(0) > B_i$). Whence, the corresponding pricing PDE (with terminal condition g_2^2)

has to be solved on the domain

$$G_3 :=]B_1, \infty[\times]B_2, \infty[\times \ldots, \times]B_d, \infty[$$

subject to zero Dirichlet boundary conditions. Since the operator $\partial_t + \mathcal{A} - r\mathrm{Id}$ is linear, we obtain the value of the product at time t_1^A by solving the PDEs (note that $g_3^2 = g_2^2$)

$$\left\{ \begin{array}{rcll} \partial_t V_i^2 + \mathcal{A} V_i^2 - r V_i^2 &=& 0 & \text{in } G_i \times [t_1^A, T[\\ V_i^2(\mathbf{x}, T) &=& g_i^2(\mathbf{x}) e^{-r\delta_2} & \text{in } G_i \end{array} \right. .$$

To find the value of the option at inception, we need to solve the same PDEs once more, but with different "payoff" functions. Indeed, suppose that at t_1^A all underlyings are at or above the early redemption level. In this case, the product expires and the investor receives (at the next coupon payment date t_{J-1}^C) the cash flow $N+C$; since she also received all the previous coupons, the payoff of the product (at t_1^A) is $(N+C(J-1))e^{-r\delta_1}$, with $C(j)$ as in (5.3). If at least underlying is below the early redemption level, the autocallable MBRC corresponds to an ordinary MBRC. Thus, to find the value $V = V^1(\mathbf{x}, 0) = V_1^1(\mathbf{x}, 0) - V_2^1(\mathbf{x}, 0) + V_3^1(\mathbf{x}, 0)$ of the product we solve the PDEs

$$\left\{ \begin{array}{rcll} \partial_t V_i^1 + \mathcal{A} V_i^1 - r V_i^1 &=& 0 & \text{in } G_i \times [0, t_1^A[\\ V_i^1(\mathbf{x}, t_1^A) &=& g_i^1(\mathbf{x}) & \text{in } G_i \end{array} \right.$$

subject to the terminal conditions

$$\begin{array}{rcl} g_1^1(\mathbf{x}) &:=& (C(J-1)+N)e^{-r\delta_1} 1_{\{\mathbf{x} \geq \boldsymbol{\ell}\}} + V_1^2(\mathbf{x}, t_1^A)(1 - 1_{\{\mathbf{x} \geq \boldsymbol{\ell}\}}) \\ g_2^1(\mathbf{x}) &:=& V_2^2(\mathbf{x}, t_1^A)(1 - 1_{\{\mathbf{x} \geq \boldsymbol{\ell}\}}) \\ g_3^1(\mathbf{x}) &:=& V_3^2(\mathbf{x}, t_1^A)(1 - 1_{\{\mathbf{x} \geq \boldsymbol{\ell}\}}) \, . \end{array}$$

Herein, the indicator function $1_{\{\mathbf{a} \geq \mathbf{b}\}}$ has - for vectors $\mathbf{a} := (a_1, \ldots, a_d)$ and $\mathbf{b} := (b_1, \ldots, b_d)$ - to be understood component wise, i.e.,

$$1_{\{\mathbf{a} \geq \mathbf{b}\}} := \prod_{i=1}^{d} 1_{\{a_i \geq b_i\}} \, .$$

For a general number of early redemption dates, we thus need to solve a sequence of $3H$ PDEs as follows. For $h = H, H-1, \ldots, 1$ successively solve $(i = 1, 2, 3)$

$$\left\{ \begin{array}{rcll} \partial_t V_i^h + \mathcal{A} V_i^h - r V_i^h &=& 0 & \text{in } G_i \times [t_{h-1}^A, t_h^A[\\ V_i^h(\mathbf{x}, t_{h-1}^A) &=& g_i^h(\mathbf{x}) & \text{in } G_i \end{array} \right. \tag{5.4}$$

where, for $h = H$,

$$g_1^H(\mathbf{x}) = (C(J)+N)e^{-r\delta_H}, \quad g_2^H(\mathbf{x}) = g_3^H(\mathbf{x}) = N \max\left\{1 - \min_i \frac{x_i}{K_i}, 0\right\} e^{-r\delta_H} \, ,$$

and for $h = H-1, H-2, \ldots, 1$,

$$\begin{array}{rcl} g_1^h(\mathbf{x}) &:=& (C(J-H+h)+N)e^{-r\delta_h} 1_{\{\mathbf{x} \geq \boldsymbol{\ell}\}} + V_1^{h+1}(\mathbf{x}, t_{h-1}^A)(1 - 1_{\{\mathbf{x} \geq \boldsymbol{\ell}\}}) \\ g_2^h(\mathbf{x}) &:=& V_2^{h+1}(\mathbf{x}, t_{h-1}^A)(1 - 1_{\{\mathbf{x} \geq \boldsymbol{\ell}\}}) \\ g_3^h(\mathbf{x}) &:=& V_3^{h+1}(\mathbf{x}, t_{h-1}^A)(1 - 1_{\{\mathbf{x} \geq \boldsymbol{\ell}\}}) \, . \end{array}$$

The value of an autocallable MBRC at inception is then $V = \sum_{i=1}^{3}(-1)^{i+1}V_i^1(\mathbf{x}_0, 0)$.

Now we consider two underlyings from which we assume that the joint dynamics $\mathbf{X}(t) := (X(t), Y(t))^\top$ follow a two-dimensional geometric Brownian motion. Hence, the operator $\mathcal{A}$ in (5.4) is given by (2.5). We switch to time-to-maturity and restrict the PDEs to bounded domains. In particular, we restrict G_1 and G_2 to $[0, x_r[\times[0, y_r[$ and solve the corresponding equations up to the boundaries $\{0\}\times]0, y_r[$ and $]0, x_r[\times\{0\}$. In the equations for V_i^h we chose a homogeneous Neumann (Dirichlet) condition on $]0, x_r[\times\{y_r\}$ and on $\{x_r\}\times]0, y_r[$ if $i = 1$ ($i = 2$). Furthermore, we set homogeneous Dirichlet conditions on the whole boundary of G_3. The function `mbrc_autocall_bs` (see below) is a realisation of the sequence (5.4). Herein, `x0`, `q`, `K`, `B` and `l` are all vectors/arrays of length $d = 2$ containing the stock prices (x_0, y_0) at $t = 0$, the continuous dividend yields (q_1, q_2) of the stocks, the strikes (K_1, K_2), the barriers (B_1, B_2) and the early redemption levels (ℓ_1, ℓ_2). Furthermore, `TauC` and `TauA` are vectors/arrays containing (in increasing order) the coupon payment dates t_j^C and the early redemption dates t_h^A, respectively. Finally, `S` is (a point-estimator of) the covariance matrix

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix}$$

of (the log-returns of) the stocks. The function `mbrc_autocall_bs` becomes in Matlab

```
function V = mbrc_autocall_bs(x0,S,r,q,K,B,l,Nom,C,TauC,TauA)

H = length(TauA); c = C/Nom;
U = exp(-r*TauC); U = cumsum(U); R = U.*exp(r*TauC);

D = exp(-r*(TauC(end+1-H:end)-TauA)); R = R(end+1-H:end).*D;
Tau = [0;TauA]; tau = diff(Tau);
L = [9 9]; N = 2.^L-1; M = ceil(0.1*max((N))); theta = 1;
a = {@(x)-0.5*S(1,1)*x.^2,@(y)y.^0,@(x)x.^0,@(y)-0.5*S(2,2)*y.^2,...
    @(x)-S(1,2)*x,@(y)y};
b = {@(x)-(r-q(1))*x,@(y)y.^0,@(x)x.^0,@(y)-(r-q(2))*y};
cf = {@(x)r*x.^0,@(y)y.^0};

g1 = @(x,y)x.^0.*y.^0*(D(end)+c*R(end));
g2 = @(x,y)max(1-min(x/K(1),y/K(2)),0)*D(end); g3 = g2;

BC1 = [3 1 3 1]; G1 = [0 4*x0(1) 0 4*x0(2)];
P1 = perm_matrix(N(2)+(BC1(3)==3)+(BC1(4)==3),N(1)+(BC1(1)==3)+(BC1(2)==3));
BC2 = [3 0 3 0]; BC3 = [0 0 0 0]; G2 = [B(1) 5*B(1) B(2) 5*B(2)];
P2 = perm_matrix(N(2)+(BC3(3)==3)+(BC3(4)==3),N(1)+(BC3(1)==3)+(BC3(2)==3));

tic
for h = 1:H
    [x1,y1,w1] = pricing_2d(a,b,cf,tau(end+1-h),g1,G1,BC1,N,M,P1,theta);
    g1 = @(x,y)(x1>=l(1)).*(y1>=l(2))*(D(end-h)+c*R(end-h))+w1.*(1-(x1>=l(1))
    .*(y1>=l(2)));
    [x2,y2,w2] = pricing_2d(a,b,cf,tau(end+1-h),g2,G1,BC2,N,M,P1,theta);
    g2 = @(x,y)w2.*(1-(x2>=l(1)).*(y2>=l(2)));
```

```
    [x3,y3,w3] = pricing_2d(a,b,cf,tau(end+1-h),g3,G2,BC3,N,M,P2,theta);
    g3 = @(x,y)w3.*(1-(x3>=l(1)).*(y3>=l(2)));
end
toc
V = Nom*interpn(x1,y1,w1,x0(1),x0(2))-Nom*(interpn(x2,y2,w2,x0(1),x0(2))-...
    interpn(x3,y3,w3,x0(1),x0(2)));
```

and in Python, respectively.

```
import numpy as np; from pricing_2d import pricing_2d
from perm_matrix import perm_matrix; from scipy.interpolate import interpn
import timeit

def mbrc_autocall_bs(x0,S,r,q,K,B,l,Nom,C,TauC,TauA):

    H = len(TauA); c = C/Nom;
    U = np.exp(-r*TauC); U = np.cumsum(U); R = U*np.exp(r*TauC);

    D = np.exp(-r*(TauC[H-2:]-TauA)); R = R[H-2:]*D;
    Tau = np.hstack((0,TauA)); tau = np.diff(Tau);
    L = np.asarray([9,9]); N = 2**L-1; M = np.int(np.ceil(0.1*max(N))); theta
     = 1;
    a = [lambda x:-0.5*S[0,0]*x**2,lambda y:y**0,lambda x:x**0,lambda y:-0.5*
    S[1,1]*y**2,lambda x:-S[0,1]*x,lambda y:y];
    b = [lambda x:-(r-q[0])*x,lambda y:y**0,lambda x:x**0,lambda y:-(r-q[1])*
    y];
    cf = [lambda x:r*x**0,lambda y:y**0];

    g1 = lambda x,y: x**0*y**0*(D[-1]+c*R[-1]);
    g2 = lambda x,y: np.maximum(1-np.minimum(x/K[0],y/K[1]),0)*D[-1];
    g3 = lambda x,y: np.maximum(1-np.minimum(x/K[0],y/K[1]),0)*D[-1];

    tic = timeit.default_timer();
    BC1 = [3,1,3,1]; G1 = [0,4*x0[0],0,4*x0[1]];
    P1 = perm_matrix(N[0]+(BC1[0]==3)+(BC1[1]==3),N[1]+(BC1[2]==3)+(BC1
    [3]==3));
    BC2 = [3,0,3,0]; BC3 = [0,0,0,0]; G2 = [B[0],5*B[0],B[1],5*B[1]];
    P2 = perm_matrix(N[0]+(BC3[0]==3)+(BC3[1]==3),N[1]+(BC3[2]==3)+(BC3
    [3]==3));


    for h in range(0,H):
        [x1,y1,w1] = pricing_2d(a,b,cf,tau[H-1-h],g1,G1,BC1,N,M,P1,theta);
        g1 = lambda x,y: (x1>=l[0])*(y1>=l[1])*(D[H-1-h]+c*R[H-1-h])+w1*(1-(
    x1>=l[0])*(y1>=l[1]));
        [x2,y2,w2] = pricing_2d(a,b,cf,tau[H-1-h],g2,G1,BC2,N,M,P1,theta);
        g2 = lambda x,y: w2*(1-(x2>=l[0])*(y2>=l[1]));
        [x3,y3,w3] = pricing_2d(a,b,cf,tau[H-1-h],g3,G2,BC3,N,M,P2,theta);
        g3 = lambda x,y: w3*(1-(x3>=l[0])*(y3>=l[1]));

    toc = timeit.default_timer(); display(toc-tic)

```

```
38      V = Nom*interpn((x1.T[0],y1[0]),w1,x0)-Nom*(interpn((x2.T[0],y2[0]),w2,x0
        )
39      -interpn((x3.T[0],y3[0]),w3,x0));
40      return V
```

We now consider the particular example of such a product (with ISIN CH0434743727) on AMS and Logitech SA where $t = 0$ corresponds to 10/10/18 with $\mathbf{X}(0) = \mathbf{x}_0 = (x_0, y_0) = (47.54, 39.13)$ and where $N = 1000$, $J = 8$, $H = 5$, $\ell_1 = K_1 = x_0$, $\ell_2 = K_2 = y_0$, $(B_1, B_2) = (26.147, 21.5215)$ and $C = 25$. Furthermore, the coupon payment dates and the early redemption dates are $t_1^C = 17/01/19$, $t_2^C = 17/04/19$, $t_3^C = 17/07/19$, $t_4^C = 17/10/19$, $t_5^C = 17/01/20$, $t_6^C = 17/04/20$, $t_7^C = 17/07/20$, $t_8^C = 17/10/20$ as well as $t_1^A = 10/10/19$, $t_2^A = 10/01/20$, $t_3^A = 08/04/20$, $t_4^A = 10/07/20$, $t_5^A = 12/10/20$. To measure time spans, we apply the so-called 30/360 European rule, which leads to the sets (all values need to be divided by 360)

$$\begin{aligned} \mathcal{T}^C &= \{t_1^C, \ldots, t_J^C\} = \{97, 187, 277, 367, 457, 547, 637, 729\} \\ \mathcal{T}^A &= \{t_1^A \ldots, t_H^A\} = \{360, 450, 539, 630, 722\} \, . \end{aligned}$$

At $t = 0$, we take from Bloomberg the values $q_1 = \ln(1.01362)$, $q_2 = \ln(1.01867)$ and $r = -0.00473$. To get the covariance matrix $\mathbf{\Sigma}$, we use a time series of prices of the underlyings up to $t = 0$ and estimate $\sigma_1 = 0.391$, $\sigma_2 = 0.207$ and $\rho = 0.503$. The function `mbrc_autocall_bs` then returns $V(\mathbf{x}_0, 0) \doteq 988.37$. Matlab is about 1.75 times faster than Python.

References

[AFP18] D. Ackerer, D. Filipović, and S. Pulido, *The Jacobi stochastic volatility model*, Finance and Stochastics **22** (2018), no. 3, 667–700.

[Ber16] L. Bergomi, *Stochastic Volatility Modeling*, Financial Mathematics Series, Chapman&Hall/CRC, 2016.

[EGJS18] D. Elbrächter, Ph. Grohs, A. Jentzen, and Ch. Schwab, *DNN Expression Rate Analysis of High-dimensional PDEs: Application to Option Pricing*, Technical Report 2018-33, Seminar for Applied Mathematics, ETH Zürich, Switzerland, 2018.

[EHJ17] W. E, J. Han, and A. Jentzen, *Deep Learning-Based Numerical Methods for High-Dimensional Parabolic Partial Differential Equations and Backward Stochastic Differential Equations*, Communications in Mathematics and Statistics **5** (2017), no. 4, 349–380.

[GHJvW18] Ph. Grohs, F. Hornung, A. Jentzen, and Ph. von Wurstemberger, *A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of Black-Scholes partial differential equations*, Technical Report 2018-32, Seminar for Applied Mathematics, ETH Zürich, Switzerland, 2018.

[GVL13] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed., Johns Hopkins Series in the Mathematical Sciences, The Johns Hopkins University Press, 2013.

[Hes93] S. Heston, *A closed-form Solution for Options with Stochastic Volatility, with Applications to Bond and Currency Options*, The Review of Financial Studies **6** (1993), 327–343.

[Hil19] N. Hilber, *26 lines of code to price single factor derivates*, Working Paper (2019), 1–26.

[HRSW13] N. Hilber, O. Reichmann, Ch. Schwab, and Ch. Winter, *Computational Methods for Quantitative Finance*, Springer, 2013.

[HS00] D. Heath and M. Schweizer, *Martingales versus PDEs in Finance: An Equivalence Result with Examples*, Journal of Applied Probability **37** (2000), no. 4, 947–957.

[HS81] H. Henderson and S. Searle, *The vec-permutation matrix, the vec operator and Kronecker products: A review*, Linear and Multilinear Algebra **9** (1981), no. 4, 271–288.

[I'HM13] K.J. In 't Hout and C. Mishra, *Stability of ADI schemes for multidimensional diffusion equations with mixed derivative terms*, Applied Numerical Mathematics **74** (2013), 83–94.

[OR73] O. Oleĭnik and E. Radkevič, *Second Order Equations With Nonnegative Characteristic Form*, Plenum Press, New York, 1973. Translated from the Russian by Paul C. Fife. MR0457908 (56 #16112)

[SMF14] D. Sesana, D. Marazzina, and G. Fusai, *Pricing exotic derivatives exploiting stucture*, European Journal of Operational Research **236** (2014), no. 1, 369–381.

[Tho06] V. Thomée, *Galerkin Finite Element Methods for Parabolic Problems*, 2nd ed., Springer Series in Computational Mathematics, vol. 25, Springer, 2006.

[ZWCS13] Y.-I. Zhu, X. Wu, I.-L. Chern, and Z.-z. Sun, *Derivative Securities and Difference Methods*, Springer, 2013.

Appendix

A.1. **The eigenvalues of Q.** We show that the eigenvalues of the matrix $\mathbf{Q}$ defined in (2.12) are non-negative, for all $(x, v) \in \mathbb{R}_0^+ \times [v_{\min}, v_{\max}]$. The characteristic polynomial is

$$p_{\mathbf{Q}}(\lambda) = -\lambda\left[\left(\frac{1}{2}vx^2 - \lambda\right)\left(\frac{1}{2}\delta^2 Q(v) - \lambda\right) - \frac{1}{4}\rho^2\delta^2 x^2 Q^2(v)\right] .$$

Clearly, $\lambda = 0$ is an eigenvalue. The other eigenvalues are solutions of the quadratic equation $\lambda^2 + b\lambda + c = 0$ with $b := -\frac{1}{2}(vx^2 + \delta^2 Q(v))$ and $c := \frac{1}{4}(\delta^2 x^2 v Q(v) - \delta^2\rho^2 x^2 Q^2(v))$. The corresponding discriminat can be simplified to

$$D := \frac{1}{4}\left[\left(vx^2 - \delta^2 Q(v)\right)^2 + 4\delta^2\rho^2 x^2 Q^2(v)\right] \geq 0 .$$

Whence, the remaining eigenvalues are $\lambda_{1,2} = \frac{1}{2}(-b \pm \sqrt{D})$. Since $x \geq 0$ and $Q(v) \geq 0$, we have that $\lambda_1 = \frac{1}{2}(-b + \sqrt{D}) \geq 0$ and it remains to show that $\lambda_2 = \frac{1}{2}(-b - \sqrt{D}) \geq 0$. The inequality $\lambda_2 \geq 0$ is equivalent to the inequality

$$vx^2 + \delta^2 Q(v) \geq \sqrt{\left(vx^2 - \delta^2 Q(v)\right)^2 + 4\delta^2\rho^2 x^2 Q^2(v)} ;$$

since the quantities to the left and to the right of this inequality are ≥ 0, it is furthermore equivalent to

$$\left(vx^2 + \delta^2 Q(v)\right)^2 \geq \left(vx^2 - \delta^2 Q(v)\right)^2 + 4\delta^2\rho^2 x^2 Q^2(v)$$

and - after simplification - to

$$vQ(v) \geq \rho^2 Q^2(v) . \tag{A.5}$$

For $v \in \{v_{\min}, v_{\max}\}$, $Q(v) = 0$ and the inequality (A.5) obviously holds. If $v \in]v_{\min}, v_{\max}[$, $Q(v) > 0$ and (A.5) further reduces to $v \geq \rho^2 Q(v)$. From [AFP18] we know that $v \geq Q(v)$, and since $\rho^2 \in [0, 1]$, the inequality $v \geq \rho^2 Q(v)$ therefore holds. Thus, $\lambda_2 \geq 0$ and we are done.

A.2. **Matlab/Python code `matgen`.** The function `matgen` outputs matrices ${}^{r}_{l}\mathbf{M}^{(k)}_{f}$.

```
% matgen(list,BC,xl,xr,N) returns NxN-matrices M_f^(k), k = 0,1,2, for
% given boundary conditions defined in BC.
%
% Example. [Mkj means the jth matrix M_f^(k)]
%
% [M21,M22,M11,M01] = matgen(list,BC,xl,xr,N)
%
% with the list
%
% list = {{'M2',@(x)x.^2},{'M2',@(x)x},{'M1',@(x)x},{'M0',@(x)x.^0}}
%
% and boundary conditions BC = [1 2] (0 = Dirichlet, 1 = Neumann,
% 2 = second derivative, 3 intrinsic) returns the finite-difference-quotients
% matrices M21 and M22 belonging to x^2*u''(x) and x*u''(x), respectively,
% the matrix M11 belonging to x*u'(x) and the matrix M01 corresponding to u(x)
% over the interval G = [xl,xr]. On the left boundary a Neumann condition
% is given, on the right boundary the second derivative is specified.

function varargout = matgen(list,BC,xl,xr,N)

% aux vars
h = (xr-xl)/(N+1); x = (xl-h:h:xr+h)'; nl = BC(1); nr = BC(2);
hd = @(x)0.5*x.^2-1.5*x+1; hn = @(x)-x.^2+2*x; hs = @(x)0.5*(x.^2-x);

% the matrices M_f^(k)
for k = 1:length(list)
    str = list(k); str = str{:};
    typ = str(1); typ = typ{:}; f = str(2); f = f{:};
    if strcmp(typ,'M2')
        M = 1/h^2*spdiags([f(x(3:N+4)),-2*f(x(2:N+3)),f(x(1:N+2))],...
            -1:1,N+2,N+2);
        if nl == 3, M(1,1:4) = f(xl)/h^2*[2 -5 4 -1];
        else
            M(1,:) = []; M(:,1) = [];
            M(1,1:3) = f(x(3))/(h^2)*(hd(nl)*[-2 1 0]+...
                hn(nl)*[-2/3 2/3 0]+hs(nl)*[1/2 -1 1/2]);
        end
        if nr == 3, M(end,end-3:end) = f(xr)/h^2*[-1 4 -5 2];
        else
            M(end,:) = []; M(:,end) = [];
            M(end,end-2:end) = f(x(N+2))/(h^2)*(hd(nr)*[0 1 -2]+...
                hn(nr)*[0 2/3 -2/3]+hs(nr)*[1/2 -1 1/2]);
        end
    elseif strcmp(typ,'M1')
        M = 1/(2*h)*spdiags([-f(x(3:N+4)),zeros(N+2,1),f(x(1:N+2))],...
            -1:1,N+2,N+2);
        if nl == 3, M(1,1:3) = f(xl)/(2*h)*[-3 4 -1];
        else
            M(1,:) = []; M(:,1) = [];
```

```
            M(1,1:3) = f(x(3))/(2*h)*(hd(nl)*[0 1 0]+...
                hn(nl)*[-4/3 4/3 0]+hs(nl)*[-5/2 3 -1/2]);
        end
        if nr == 3, M(end,end-2:end) = f(xr)/(2*h)*[1 -4 3];
        else
            M(end,:) = []; M(:,end)= [];
            M(end,end-2:end) = f(x(N+2))/(2*h)*(hd(nr)*[0 -1 0]+...
                hn(nr)*[0 -4/3 4/3]+hs(nr)*[1/2 -3 5/2]);
        end
    elseif strcmp(typ,'M0')
        M = spdiags(f(x(2:N+3)),0,N+2,N+2);
        if nl == 3; else M(1,:) = []; M(:,1) = []; end
        if nr == 3; else M(end,:) = []; M(:,end) = []; end
    else
    end
    varargout(k) = {M};
end
```

```
import numpy as np; from scipy.sparse import spdiags

def matgen(matlist,BC,xl,xr,N):
    '''matgen(list,BC,xl,xr,N) returns NxN-matrices M_f^(k), k = 0,1,2, for
    given boundary conditions defined in BC.

    Example. [Mkj means the jth matrix M_f^(k)]

    Mat = matgen(matlist,BC,xl,xr,N)

    with the list

    matlist = [["M2",lambda x: x**2],["M2",lambda x: x],["M1",lambda x: x],["
    M0",lambda x: 1]]

    and boundary conditions BC = [1,2] (0 = Dirichlet, 1 = Neumann,
    2 = second derivative, 3 intrinsic) returns the finite-difference-
    quotients
    matrices M21 and M22 belonging to x^2*u''(x) and x*u''(x), respectively,
    the matrix M11 belonging to x*u'(x) and the matrix M01 corresponding to u
    (x)
    over the interval G = [xl,xr]. On the left boundary a Neumann condition
    is given, on the right boundary the second derivative is specified.'''

    # aux vars
    h = (xr-xl)/(N+1); x = np.arange(xl-h,xr+2*h,h);
    nl = BC[0]; nr = BC[1];
    hd = lambda x: 0.5*x**2-1.5*x+1; hn = lambda x:-x**2+2*x;
    hs = lambda x: 0.5*(x**2-x)

    U = [None]*len(matlist)*2; count = 0

    # the matrices M_f^(k)
    for j in range(0,len(matlist)):
```

```
        count =count+1; v = matlist[j]; f = v[1]
        if v[0]=="M2":
            U1 = 1/(h**2)*spdiags([f(x[2:N+4]),-2*f(x[1:N+3]),f(x[0:N+2])],
                    [-1,0,1],N+2,N+2).tolil()
            if nl==3:
                U1[0,0:4] = f(xl)/h**2*np.array([2,-5,4,1])
            else:
                U1 = U1[1:,:]; U1 = U1[:,1:]
                U1[0,0:3] = f(x[2])/h**2*(hd(nl)*np.array([-2,1,0])+
                  hn(nl)*np.array([-2/3,2/3,0])+hs(nl)*np.array([1/2,-1,1/2])
    )
            if nr==3:
                U1[-1,-4:] = f(xr)/h**2*np.array([-1,4,-5,2])
            else:
                U1 = U1[:-1,:]; U1 = U1[:,:-1]
                U1[-1,-3:] = f(x[N+1])/h**2*(hd(nr)*np.array([0,1,-2])+
                  hn(nr)*np.array([0,2/3,-2/3])+hs(nr)*np.array([1/2,-1,1/2])
    )

            U[j] = U1.todia()
        elif v[0]=="M1":
            U1 = 1/(2*h)*spdiags([-f(x[2:N+4]),np.zeros(N+2),f(x[0:N+2])],
                    [-1,0,1],N+2,N+2).tolil()
            if nl==3:
                U1[0,0:3] = f(xl)/(2*h)*np.array([-3,4,-1])
            else:
                U1 = U1[1:,:]; U1 = U1[:,1:]
                U1[0,0:3] = f(x[2])/(2*h)*(hd(nl)*np.array([0,1,0])+
                  hn(nl)*np.array([-4/3,4/3,0])+hs(nl)*np.array
    ([-5/2,3,-1/2]))
            if nr==3:
                U1[-1,-3:] = f(xr)/(2*h)*np.array([1,-4,3])
            else:
                U1 = U1[:-1,:]; U1 = U1[:,:-1]
                U1[-1,-3:] = f(x[N+1])/(2*h)*(hd(nr)*np.array([0,-1,0])+
                  hn(nr)*np.array([0,-4/3,4/3])+hs(nr)*np.array([1/2,-3,5/2])
    )

            U[j] = U1.todia()

        else:
            U1 = spdiags(f(x[1:N+3]),[0],N+2,N+2).tolil();
            if nl<3: U1 = U1[1:,:]; U1 = U1[:,1:]
            if nr<3: U1 = U1[:-1,:]; U1 = U1[:,:-1]

            U[j] = U1.todia()

    return U
```

A.3. **Matlab/Python code `perm_matrix`.** We provide the function which generates the perfect shuffle matrix $\mathbf{P}^{nm\times nm}$.

```
function P = perm_matrix(n,m)
Im = speye(m); In = speye(n); P = sparse(m*n,m*n);
for j = 1:n, P((j-1)*m+1:j*m,:) = kron(Im,In(j,:)); end
```

```
import numpy as np; from scipy import sparse; from scipy.sparse import eye;
from scipy.sparse import vstack

def perm_matrix(n,m):
    Im = eye(m,m); x = np.zeros(n); x[0] = 1.0; P = sparse.kron(Im,x)
    for j in range(1,n):
        x = np.zeros(n); x[j] = 1.0; P = vstack([P,sparse.kron(Im,x)])

    return P
```

School of Management and Law, ZHAW, CH-8400 Winterthur, Switzerland

Zürcher Hochschule
für angewandte Wissenschaften

**School of
Management and Law**

St.-Georgen-Platz 2
Postfach
8401 Winterthur
Schweiz

www.zhaw.ch/sml