Zurich University
of Applied Sciences

School of
Management and Law

26 Lines of Code to Price Single 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

26 LINES OF CODE TO PRICE SINGLE FACTOR DERIVATIVES

NORBERT HILBER

ABSTRACT. We provide short, easy to use Matlab and Python codes to solve derivative pricing problems for one underlying whose evolution is modelled by a diffusion with possibly time-dependent coefficients.

CONTENTS

1. INTRODUCTION

The pricing and hedging of financial derivatives as well as the calibration of models to market data are major challenges in the financial derivatives industry. Over the last decades, the complexity of the used models and of the payoff structure of derivatives written on different types of stochastic underlyings have constantly grown. Great efforts have been undertaken both in academia as well as in the financial industry to solve the corresponding pricing, hedging and calibration problems. Due to the mathematical complexity, most of these can not be solved analytically, but a solution has to be found by some approximation procedure. Here, we have

Date: June 3, 2019.
2010 *Mathematics Subject Classification.* 35K15, 65M06, 91G20, 91G30.
Key words and phrases. Finite differences, partial differential equations, derivative securities, interest rates.

available (borrowed mostly from other areas of science) different types of methods, for example Monte Carlo simulation, solving numerically (stochastic, ordinary, partial) differential equations, non-linear regression, particle method and neural networks, to name but a few. The findings of the research undertaken are summarised and made available to a greater audience in well written books; we just mention [Ber16, BR02, Bos14, BL02, BM06, BMP13, CT03, Cré13, DS06, Duf06, Fil09, Gla03, JL14, Hir13, Kwo08, Lab09, PS06, ZWCS13]. and we are aware that this list is subjective and far of being complete. These books describe the numerical procedure used to solve a specific pricing, hedging or calibration problem at different levels of detailing, ranging from a sketchy to an in-depth description. In most of these books, however, a code which could be directly used by the reader is not available. The goal of this note is to partially close this gap. We give short Matlab and Python codes which solve (linear) pricing problems for derivatives written on a single factor diffusion by the finite-difference-method.

The note is organised as follows. In chapter 2, we give a short description of the considered pricing problems and partial differential equations (PDEs), respectively. In the next chapter 3, we derive a second-order finite difference discretisation (in the space variable) used to approximate the solution of the pricing PDE. Here, special attention is paid to the incorporation of different types of (the important) boundary conditions. Chapter 4 is then devoted to the discretisation with respect to time, which we achieve by considering the Padé approximation of the exponential function (a topic which seems not to be widespread in the finite difference community). In chapter 5 we provide Matlab/Octave and Python codes to solve numerically the PDE of chapter 2. In the last chapter 6, we apply the routine presented in chapter 5 to four particular pricing problems. For each of these problems we again provide codes.

2. Pricing equation

Consider a stochastic process $X(t)$ modelling (for example) a stock price at time t. We assume that X solves (under a local unique martingale measure) the stochastic differential equation (SDE)

$$\mathrm{d}X(t) = \mu(X(t), t)\mathrm{d}t + \sigma(X(t), t)\mathrm{d}W(t), \quad X(0) = x_0 \, , \tag{2.1}$$

with given bivariate deterministic functions $\mu(x, t)$ and $\sigma(x, t) > 0$ and with $W(t)$ a standard Brownian motion. Now consider a non-american style financial derivative with payoff (function) g and maturity T written on X. Furthermore, let $r(t)$ be the (deterministic) continuously compounded risk free. By the general principles of derivatives theory, the value $V(x, t)$ of the derivative is given by the conditional expectation

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

To find $V(x, t)$, one can try to solve the expectation analytically which is, however, only possible for (very) few models of X. For most models, we have to apply numerical methods to get the pricing function. The prevailing method used in the financial industry is Monte Carlo simulation, since it is relatively easy to realise it on a computer. To estimate the expectation, one integrates numerically N-times (for example by using the Euler approximation) the SDE (2.1) to get approximate realisations x_T^i of $X(T)$ and then sets $V(x,0) \approx e^{-\int_0^T r(u)\mathrm{d}u} \frac{1}{N}\sum_{i=1}^N g(x_T^i)$, see e.g. [Gla03]. Other possible methods are transform methods (like the Laplace transform, the fast Fourier transform or the cos-method) and trees; in this note we focus on approximating V by solving partial differential equations (PDEs) with the finite-difference-method. An alternative method to solve PDEs is the mighty finite-element-method, which is not considered here, since its mathematical foundation is much more involved, see for example [HRSW13]. The link between the expectation (2.2) and partial differential equations is provided by the Feynman-Kac theorem. It states that, under certain conditions on r, μ, σ and V, the function V in (2.2) solves the partial differential equation (on the domain $G \subset \mathbb{R}$ of X)

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

see e.g. [HS00]. Herein, $\mathcal{A}$ is the so-called (infinitesimal) generator of the process X defined, for f being a continuous function, as

$$\mathcal{A}f(x) := \lim_{t\to 0} \frac{\mathbb{E}[f(X(t)) \mid X(0)=x] - f(x)}{t} \; .$$

The operator $\mathcal{A}$ is the generator of the semigroup $\{\mathbb{E}[f(X(t)) \mid X(0)=x] : t \geq 0\}$, see any book on Functional Analysis for the notion of a semigroup and its generator. It turns out that the generator of X in (2.1) is given by the second order differential operator

$$\mathcal{A} = \frac{1}{2}\sigma^2(x,t)\partial_{xx} + \mu(x,t)\partial_x \; . \tag{2.4}$$

We refer to (2.3) as the pricing equation.

Example 2.1. i) In the (standard) Black-Scholes model the process $X(t)$ (the stock price) follows a geometric Brownian motion, i.e.,

$$\mathrm{d}X(t) = (r-q)X(t)\mathrm{d}t + \sigma X(t)\mathrm{d}W(t), \quad X(0) = x_0 > 0 \; .$$

Here, the underling pays a constant continuous dividend yield $q \geq 0$ and $\sigma > 0$ is the constant volatility. Thus the functions in (2.1) are $\mu(x,t) = (r-q)x$ and $\sigma(x,t) = \sigma x$, $G = \mathbb{R}^+$, and the pricing equation becomes the Black-Scholes equation

$$\partial_t V + \frac{1}{2}\sigma^2 x^2 \partial_{xx} V + (r-q)x\partial_x V - rV = 0 \; .$$

ii) It turns out that the Feyman-Kac theorem also holds if X is the short rate, $X = r$, see e.g. [Fil09]. In the CIR short rate model for example there holds

$$\mathrm{d}X(t) = \kappa(m - X(t))\mathrm{d}t + \sigma\sqrt{X(t)}\mathrm{d}W(t), \quad X(0) = x_0 \geq 0 \tag{2.5}$$

for some constants $\kappa, \theta, \sigma > 0$. The price $V(x,t)$ of a zero-bond with maturity T then satisfies $V(x,T) = g(x) = 1$, $G = \mathbb{R}_0^+$ and

$$\partial_t V + \frac{1}{2}\sigma^2 x \partial_{xx} V + \kappa(m-x)\partial_x V - xV = 0 \; .$$

Note that for some path-dependent derivatives the payoff g in (2.2) does not only depend on $X(T)$, but also on other variables like $\min_{t\in[0,T]} X(t)$, $\max_{t\in[0,T]} X(t)$ or $X(t_j)$, where the t_j's are observation dates. As an example, consider the payoff of a (knock-out) double barrier call with strike K and with lower and upper barriers L, U, given by

$$\begin{aligned} &g\big(X(T), \min_{t\in[0,T]} X(t), \max_{t\in[0,T]} X(t)\big) \\ &\quad = \max\{X(T)-K,0\}1_{\{\min_{t\in[0,T]} X(t) > L\}} 1_{\{\max_{t\in[0,T]} X(t) < U\}} \; . \end{aligned}$$

It turns out that in this case the pricing equation (2.3) still holds, with $G =]L, U[$. If the payoff additionally depends on the underlying evaluated at some observation points $t_j \in [0,T]$, the pricing PDE (2.3) has to be replaced by a sequence of PDEs. Examples are discretely monitored barrier options, plain vanillas with discrete dividend payments of the underlying or some (auto-)callable structured products, compare with the last example in chapter 6.

To solve the PDE (2.3) by the finite-difference-method, we change to the time-to-maturity $t \mapsto T - t$, restrict the (typically unbounded) domain G to an open interval $]x_l, x_r[$ (which we again call G) and set (contract dependent) boundary conditions on x_l and x_r. Thus, the (truncated) pricing problem is a particular case of the problem: Find $w(x,t)$ such that

$$(2.6) \quad \left\{ \begin{array}{rcll} \partial_t w + a(x,t)\partial_{xx} w + b(x,t)\partial_x w + c(x,t) w &=& f(x,t) & \text{in} \quad G\times]0,T] \\ \partial_x^{(n_l)} w(x_l,t) &=& w_l(t) & \text{in} \quad]0,T] \\ \partial_x^{(n_r)} w(x_r,t) &=& w_r(t) & \text{in} \quad]0,T] \\ w(x,0) &=& g(x) & \text{in} \quad G \end{array} \right. .$$

Here, a, b and c are some given functions and by $\partial_x^{(n)} w(z,t)$ we denote the n-th derivative of w with respect to x evaluated at $x = z$. Thus, $n = 0$ corresponds to a Dirichlet boundary condition and $n = 1$ to a Neumann boundary condition. Note that n_l and n_r are typically different.

The switch from the pricing PDE (2.3) to the PDE (2.6) is called localisation and typically introduces a localisation error, $e(x,t) := |w(x,t) - V(x, T-t)| > 0$, $(x,t) \in]x_l, x_r[\times]0,T[$. From an analytical point of view, this is unsatisfactory, from a financial point of view however the situation is not that bad. Usually, we are interested in prices V only for a certain range of the value of the underlying x and we therefore may chose x_l and x_r such that the localisation error becomes negligible small in the range of interest. Of course, if x represents a non-negative quantity (stock-prices, variance), then we can not chose x_l to be negative. Note that for certain derivatives the boundary conditions are known. For example, for a knock-out (single) barrier option with barrier B there holds $x_l = B$, $w(x_l,t) = 0$ if it is down-and-out option and $x_r = B$, $w(x_r,t) = 0$ if it is a up-and-out option. For most pricing problems however, the boundary conditions are not known and it is tempting in such situations to solve the pricing PDE up to the boundary. However, it is not clear wether stating no conditions for $w(x,t)$ on $\{x_l\}\times]0,T[$ and/or $\{x_r\}\times]0,T[$ constitutes a well defined problem. For the case $x_l = 0$ a partial answer is given by the theory of PDEs of second order with non-negative characteristic form, see for example [OR73]. Applied to the equation (already switched to time-to-maturity)

$$\partial_t w + a(x,t)\partial_{xx} w + b(x,t)\partial_x w + c(x,t)w = 0$$

in (2.6) the theory states that there is no boundary condition needed if $a(0,t) = 0$, $\forall t$ and if

$$b(0,t) - \partial_x a(0,t) \leq 0, \quad \forall t \tag{2.7}$$

holds. For illustration, we apply this to the pricing problems in example 2.1.

Example 2.2. i) In the Black-Scholes pricing equation there holds $a(x,t) = -\frac{1}{2}\sigma^2 x^2$, $\partial_x a(x,t) = -\sigma^2 x$ and $b(x,t) = -(r-q)x$, such that the condition (2.7) is satisfied and the Black-Scholes PDE holds also at $x = 0$: it is not necessary to specify a boundary condition.

ii) Consider the pricing of a zero-bond in the CIR short-rate model. Since $a(x,t) = -\frac{1}{2}\sigma^2 x$, $\partial_x a(x,t) = -\frac{1}{2}\sigma^2$ and $b(x,t) = -\kappa(m-x)$, the condition (2.7) translates to

$$-\kappa m + \frac{1}{2}\sigma^2 \leq 0 \Rightarrow \kappa m \geq \frac{1}{2}\sigma^2 \ .$$

Hence, if $2\kappa m \geq \sigma^2$, then the pricing equation also holds at $x = 0$ and there are no conditions needed. However, it is shown in [ET11] that we need not to specify a boundary condition even if $2\kappa m < \sigma^2$.

In the next section, we develop a second order finite-difference-method to solve (2.6) numerically. Special attention is payed on the realisation of different boundary conditions, where we allow n_l, n_r to take the values $0, 1, 2$. We also consider the case where no boundary conditions are needed/given, i.e., we solve the PDE also on the boundary of G.

3. Spatial discretisation - Finite difference method

In this section, we assume for simplicity that the coefficients a, b and c in (2.6) do not depend on t. The idea of the finite-difference-method is to consider the PDE not for all $x \in G$ but only for a finite number of so-called grid points $x_i \in G$ and to replace at these grid points the partial derivatives (with respect to x) by their corresponding finite difference quotients. Let $N \in \mathbb{N}^\times$ and define the grid $\mathcal{G}_x := \{x_i \mid i = 0, \dots, N+1\}$ with

$$x_l = x_0 < x_1 < \cdots < x_N < x_{N+1} = x_r \ .$$

For simplicity, we chose an equidistant grid with $x_i = x_0 + ih$, where $h = \frac{x_r - x_l}{N+1}$ is the (constant) mesh width. By Taylorexpansion, we have

$$\partial_x w(x_i,t) = \delta_h w(x_i,t) + \mathcal{O}(h^2), \quad \partial_{xx} w(x_i,t) = \delta_h^2 w(x_i,t) + \mathcal{O}(h^2) \ , \tag{3.1}$$

where δ_h and δ_h^2 are the difference-operators

$$\delta_h f(x) := \frac{f(x+h) - f(x-h)}{2h}, \quad \delta_h^2 f(x) := \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \ .$$

Thus, by (2.6), for an arbitrary grid point x_i, $i = 1, \dots, N$, there holds

$$\partial_t w(x_i,t) + a(x_i)\delta_h^2 w(x_i,t) + b(x_i)\delta_h w(x_i,t) + c(x_i)w(x_i,t) = f(x_i,t) + \mathcal{O}(h^2) \ .$$

Setting $w_i(t) \approx w(x_i,t)$, the above becomes, for $i = 1, \dots, N$,

$$\begin{aligned} &\partial_t w_i(t) + a(x_i)\frac{w_{i-1}(t) - 2w_i(t) + w_{i+1}(t)}{h^2} \\ &\quad + b(x_i)\frac{w_{i+1}(t) - w_{i-1}(t)}{2h} + c(x_i)w_i(t) = f(x_i,t) \ . \end{aligned} \tag{3.2}$$

3.1. **Dirichlet boundary conditions.** Assuming for a brief moment Dirichlet boundary conditions, i.e., $w_0(t) = w_l(t)$, $w_{N+1}(t) = w_r(t)$, the N equations in (3.2) can simply written as

$$\left\{ \begin{array}{rcl} \mathbf{w}'(t) + \mathbf{A}\mathbf{w}(t) & = & \mathbf{f}(t) \\ \mathbf{w}(0) & = & \mathbf{g} \end{array} \right. \ . \tag{3.3}$$

Herein, we define the vectors (for the definition of the vector $\mathbf{f}(t)$ see the equation (3.5) below)

$$\mathbf{w}(t) := \begin{pmatrix} w_1(t) \\ w_2(t) \\ \vdots \\ w_N(t) \end{pmatrix}, \quad \mathbf{w}'(t) := \begin{pmatrix} \partial_t w_1(t) \\ \partial_t w_2(t) \\ \vdots \\ \partial_t w_N(t) \end{pmatrix}, \quad \mathbf{g} := \begin{pmatrix} g(x_1) \\ g(x_2) \\ \vdots \\ g(x_N) \end{pmatrix} \ . \tag{3.4}$$

The $N \times N$-matrix $\mathbf{A}$ in (3.3) is a discrete version of the differential operator $a(x_i)\partial_{xx} + b(x_i)\partial_x + c(x_i)$ and is thus the sum of three matrices

$$\mathbf{A} = \mathbf{M}_a^{(2)} + \mathbf{M}_b^{(1)} + \mathbf{M}_c^{(0)} \ .$$

For $k = 0, 1, 2$, and a continuous function y, the tridiagonal matrices $\mathbf{M}_y^{(k)}$ are defined in the following definition, compare also with [HRSW13, section 9.5].

Definition 3.1. Let y be a continuous function. Then, the entries $(m_y^{(k)})_{i,j}$ of the $N \times N$-Matrices $\mathbf{M}_y^{(k)}$ are given as

$$(m_y^{(0)})_{i,i} = y(x_i) \ ,$$

$$(m_y^{(1)})_{i,i+1} = \frac{y(x_i)}{2h}, \quad (m_y^{(1)})_{i+1,i} = -\frac{y(x_{i+1})}{2h} \ ,$$

$$(m_y^{(2)})_{i,i} = -2\frac{y(x_i)}{h^2}, \quad (m_y^{(2)})_{i,i+1} = \frac{y(x_i)}{h^2}, \quad (m_y^{(2)})_{i+1,i} = \frac{y(x_{i+1})}{h^2} \ .$$

Herein, $i = 1, \dots, N$ for the main diagonal elements and $i = 1, \dots, N-1$ for the sub-and super diagonal elements. Furthermore, $x_i = x_0 + ih$ are the equidistant grid points, and $h = (x_{N+1} - x_0)/(N+1)$. Unspecified entries are zero.

A remark to the definition 3.1 is at order. The superscript (k) in $\mathbf{M}_y^{(k)}$ indicates the order of the derivative considered (second $k = 2$, first $k = 1$ or no derivative $k = 0$) such that, for example, the matrix $\mathbf{M}_a^{(2)}$ is a discrete version of the operator $a(x_i)\partial_{xx}$.

The vector $\mathbf{f}$ in (3.3) is, in case of Dirichlet boundary conditions, given by

$$\mathbf{f}(t) = \mathbf{f}^{rhs}(t) - \left(\mathbf{M}_a^{(2),bc} + \mathbf{M}_b^{(1),bc}\right)\mathbf{w}^{bc}(t) \ , \tag{3.5}$$

with the (column) vectors of length N

$$\mathbf{f}^{rhs}(t) := \begin{pmatrix} f(x_1,t) \\ f(x_2,t) \\ \vdots \\ f(x_{N-1},t) \\ f(x_N,t) \end{pmatrix} , \quad \mathbf{w}^{bc}(t) := \begin{pmatrix} w_l(t) \\ 0 \\ \vdots \\ 0 \\ w_r(t) \end{pmatrix} . \tag{3.6}$$

The entries $(m_y^{(k),bc})_{i,j}$ of the $N \times N$-matrices $\mathbf{M}_y^{(k),bc}$ in (3.5) (belonging to Dirichlet boundary conditions, thus the additional superscript bc) are all zero, except the entries

(3.7)
$$(m_y^{(1),bc})_{1,1} = \frac{y(x_1)}{2h}, \quad (m_y^{(1),bc})_{N,N} = -\frac{y(x_N)}{2h}, \quad (m_y^{(2),bc})_{i,i} = \frac{y(x_i)}{h^2}, \quad i \in \{1, N\} \ .$$

3.2. **Non-Dirichlet boundary conditions.** If $n_{l,r} \neq 0$ in (2.6), we need to adapt the matrices appearing in (3.3). As an example, assume that we need to apply a Neumann boundary condition in (2.6) on x_r, i.e., $\partial_x w(x_r,t) = w_r(t)$. Observe that, in contrast to a Dirichlet boundary condition, the function value $w(x_r,t)$ is also unknown. As a consequence, the function $w_{N+1}(t)$ in the last of the N equations in the system (3.2) has to be found as well. It turns out however that we can eliminate this unknown function by using the Neumann condition. This can be achieved by approximating the derivative $\partial_x w(x_r,t)$ by a non-central finite difference quotient. Indeed, using a Taylorexpansion one can show

$$f'(x) = \frac{\pm 3f(x) \mp 4f(x \mp h) \pm f(x \mp 2h)}{2h} + \mathcal{O}(h^2) \tag{3.8}$$

$$f''(x) = \frac{2f(x) - 5f(x \pm h) + 4f(x \pm 2h) - f(x \pm 3h)}{h^2} + \mathcal{O}(h^2) \ . \tag{3.9}$$

Thus, we have with (3.8) left to $x_{N+1} = x_r$,

$$\partial_x w(x_r,t) = w_r(t) \Rightarrow \frac{w(x_{N+1} - 2h, t) - 4w(x_{N+1} - h, t) + 3w(x_{N+1}, t)}{2h} \approx w_r(t)$$

or

$$\frac{w_{N-1}(t) - 4w_N(t) + 3w_{N+1}(t)}{2h} = w_r(t) \ ,$$

such that

$$w_{N+1}(t) = -\frac{1}{3}w_{N-1}(t) + \frac{4}{3}w_N(t) + \frac{2}{3}hw_r(t) \; .$$

We plug in this expression into the N-th equation of (3.2) and end up with an equation which does not involve $w_{N+1}(t)$ anymore

$$\begin{aligned}\partial_t w_N(t) + a(x_N)\frac{\frac{2}{3}w_{N-1}(t) - \frac{2}{3}w_N(t) + \frac{2}{3}hw_r(t)}{h^2} \\ + b(x_N)\frac{-\frac{4}{3}w_{N-1}(t) + \frac{4}{3}w_N(t) + \frac{2}{3}hw_r(t)}{2h} + c(x_N)w_N(t) = 0 \; .\end{aligned} \tag{3.10}$$

Similarly, for a Neumann condition on the left boundary $x_l = x_0$ the first equation in (3.2) changes to

$$\begin{aligned}\partial_t w_1(t) + a(x_1)\frac{-\frac{2}{3}w_1(t) + \frac{2}{3}w_2(t) - \frac{2}{3}hw_l(t)}{h^2} \\ + b(x_1)\frac{-\frac{4}{3}w_1(t) + \frac{4}{3}w_2(t) + \frac{2}{3}hw_l(t)}{2h} + c(x_1)w_1(t) = 0 \; .\end{aligned} \tag{3.11}$$

By introducing Neumann boundary conditions, the equations satisfied by the functions $w_i(t)$, $i = 2, \ldots, N-1$ in (3.2) do not change. Thus, assuming that we apply a Neumann condition on both boundaries x_l and x_r, the N unknown functions $w_1(t), \ldots, w_N(t)$ are again solutions of the system (3.3), but the matrix $\mathbf{A}$ as well as the vector $\mathbf{f}(t)$ change to

$$\begin{aligned}\mathbf{A} &= {}^{n}_{n}\mathbf{M}_a^{(2)} + {}^{n}_{n}\mathbf{M}_b^{(1)} + \mathbf{M}_c^{(0)} \\ \mathbf{f}(t) &= \mathbf{f}^{rhs}(t) - \left({}^{n}_{n}\mathbf{M}_a^{(2),bc} + {}^{n}_{n}\mathbf{M}_b^{(1),bc}\right)\mathbf{w}^{bc}(t) \; .\end{aligned} \tag{3.12}$$

In ${}^{\cdot}_{\cdot}\mathbf{M}_y^{(k)}$ we use the left subscript to indicate the boundary condition on the left boundary x_l and the left superscript to indicate the boundary condition on the right boundary x_r. Thus, for example, ${}^{n}_{s}\mathbf{M}_a^{(2)}$ means that we consider the operator $a(x_i)\partial_{xx}$ together with the condition $\partial_{xx}w(x_l,t) = w_l(t)$ on the left boundary x_l (s stands for second derivative) and a Neumann condition on the right boundary (n stands for Neumann). If there is a Dirichlet boundary condition, we use equivalently no left sub-or superscript (as in definition 3.1) or the letter d (for Dirichlet). For example, the operator $b(x_i)\partial_x$ with a Dirichlet condition both on x_l and x_r can be denoted by ${}^{d}_{d}\mathbf{M}_b^{(1)}$ or by $\mathbf{M}_b^{(1)}$.

The Neumann matrices ${}^{n}_{n}\mathbf{M}_y^{(k)}$ only differ from the Dirichlet matrices $\mathbf{M}_y^{(k)}$ in the first and last row. For example, by definition 3.1, the first three entries of the first row of the matrix $\mathbf{M}_y^{(2)}$ are $\frac{y_1}{h^2}(-2, 1, 0)$, whereas the same entries of the matrix ${}^{\cdot}_{n}\mathbf{M}_y^{(2)}$ are given by $\frac{y_1}{h^2}(-\frac{2}{3}, \frac{2}{3}, 0)$. We denote by ${}_{j}\mathbf{m}^{(k)}$ the (vector of) the first three elements of the first row of the matrix ${}^{\cdot}_{j}\mathbf{M}_y^{(k)}$ with $j \in \{d, n, s\}$ and by ${}^{j}\mathbf{m}^{(k)}$ the (vector of) the last three elements of the last row of the matrix ${}^{j}_{\cdot}\mathbf{M}_y^{(k)}$. Thus, for example, there holds ${}^{n}\mathbf{m}^{(1)} = \frac{y_N}{2h}(0, -\frac{4}{3}, \frac{4}{3})$ by (3.10) or ${}_{n}\mathbf{m}^{(2)} = \frac{y_1}{h^2}(-\frac{2}{3}, \frac{2}{3}, 0)$ by (3.11). Table 1 summarises the possible boundary conditions and their resulting first and last rows of the matrices ${}^{\cdot}_{\cdot}\mathbf{M}_y^{(k)}$, respectively. Note that the case $j = d$ is already covered in the definition 3.1.

matrix	row	Dirichlet $j = d$	Neumann $j = n$	second der. $j = s$
${}^{\cdot}_{\cdot}\mathbf{M}_y^{(1)}$	first, ${}_j\mathbf{m}^{(1)} =$	$\frac{y_1}{2h}(0,\ 1,\ 0)$	$\frac{y_1}{2h}(-\frac{4}{3},\ \frac{4}{3},\ 0)$	$\frac{y_1}{2h}(-\frac{5}{2},\ 3,\ -\frac{1}{2})$
	last, ${}^j\mathbf{m}^{(1)} =$	$\frac{y_N}{2h}(0,\ -1\ 0)$	$\frac{y_N}{2h}(0,\ -\frac{4}{3},\ \frac{4}{3})$	$\frac{y_N}{2h}(\frac{1}{2},\ -3,\ \frac{5}{2})$
${}^{\cdot}_{\cdot}\mathbf{M}_y^{(2)}$	first, ${}_j\mathbf{m}^{(2)} =$	$\frac{y_1}{h^2}(-2,\ 1,\ 0)$	$\frac{y_1}{h^2}(-\frac{2}{3},\ \frac{2}{3},\ 0)$	$\frac{y_1}{h^2}(\frac{1}{2},\ -1,\ \frac{1}{2})$
	last, ${}^j\mathbf{m}^{(2)} =$	$\frac{y_N}{h^2}(0,\ 1,\ -2)$	$\frac{y_N}{h^2}(0,\ \frac{2}{3},\ -\frac{2}{3})$	$\frac{y_N}{h^2}(\frac{1}{2},\ -1,\ \frac{1}{2})$

Table 1. For non-Dirichlet boundary conditions, the first and the last row of the matrices $\mathbf{M}_y^{(k)}$ in definition 3.1 have to be adapted.

Similarly, we have to adapt the entries $(m_y^{(k),bc})_{i,i}$ for $i = 1$ and $i = N$ of the matrices in (3.7), compare with table 2. We realise the different types of boundary

entry	i	Dirichlet $j = d$	Neumann $j = n$	second der. $j = s$
$\left({}^{\cdot}_{\cdot}m_y^{(1),bc}\right)_{i,i}$	1	$-\frac{y_1}{2h}$	$\frac{y_1}{3}$	$-\frac{y_1}{4}h$
	N	$\frac{y_N}{2h}$	$\frac{y_N}{3}$	$\frac{y_N}{4}h$
$\left({}^{\cdot}_{\cdot}m_y^{(2),bc}\right)_{i,i}$	1	$\frac{y_1}{h^2}$	$-\frac{2y_1}{3h}$	$\frac{y_1}{2}$
	N	$\frac{y_N}{h^2}$	$\frac{2y_N}{3h}$	$\frac{y_N}{2}$

Table 2. For non-Dirichlet boundary conditions, the entries $\left(\mathbf{M}_y^{(k),bc}\right)_{i,i}$ in (3.7) have to be adapted.

conditions in one routine in a such a way that we are able to combine arbitrary (different) conditions. To do so, we define the auxiliary functions

$$h^d(x) := \frac{1}{2}x^2 - \frac{3}{2}x + 1, \quad h^n(x) := -x^2 + 2x, \quad h^s(x) := \frac{1}{2}x^2 - \frac{1}{2}x\ .$$

The variables $n_l, n_r \in \{0, 1, 2\}$ in (2.6) indicate the order of the derivative of the function $w(x,t)$ at the left and right boundary of the interval $]x_l, x_r[$. The auxiliary functions have the property that, for $x = n_l$ or $x = n_r$, there holds

$$h^{\cdot}(x) = 0 \text{ or } 1\ .$$

Using the functions $h^{\cdot}$, the first three entries of the first row of the matrices ${}^{\cdot}_{\cdot}\mathbf{M}_y^{(k)}$ are for some boundary condition on x_l just given by

$$\sum_{j\in\{d,n,s\}} h^j(n_l)\, {}_j\mathbf{m}^{(k)} \ .$$

Analogously, there holds for the last three entries of the last row of ${}^{\cdot}_{\cdot}\mathbf{M}_y^{(k)}$

$$\sum_{j\in\{d,n,s\}} h^j(n_r)\, {}^j\mathbf{m}^{(k)} \ .$$

Thus, we just need to specify n_l and n_r and the routine then overwrites the first and last row of the Dirichlet matrices in (3.1) by the corresponding sums.

Now we consider the case where no boundary conditions are set. Since there is no additional information on the function $w(x,t)$ at $x \in \{x_l, x_r\}$, we have to solve the PDE up to the boundary, i.e. we have to let the index i in (3.2) run from 0 to $N+1$ and the number of unknowns becomes $N+2$. Now consider exemplarily the left boundary, $i = 0$. As before, we need non-centered finite differences based on grid points to the right of x_l. By (3.8) and (3.9), we have

$$\begin{aligned}\partial_t w_0(t) + a(x_0)&\frac{2w_0(t)-5w_1(t)+4w_2(t)-w_3(t)}{h^2} \\ +b(x_0)&\frac{-3w_0(t)+4w_1(t)-w_2(t)}{2h} + c(x_l)w_0(t) \quad = \quad 0\ .\end{aligned}$$

As similar equation can derived at the right boundary ($i = N+1$). The equations in (3.2) for $i = 1,\dots,N$ do not change such that we end up again with the system (3.3) for the $N+2$ functions $\mathbf{w}(t) = (w_0(t),\dots,w_{N+1}(t))^\top$, where the $(N+2)\times(N+2)$-matrix $\mathbf{A}$ is given by

$$\mathbf{A} = {}^i_i\mathbf{M}_a^{(2)} + {}^i_i\mathbf{M}_b^{(1)} + {}^i_i\mathbf{M}_c^{(0)} \ . \tag{3.13}$$

Herein, we use the following definition, compare with definition 3.1. The left sub/-superscript i stands for “intrinsic”.

Definition 3.2. Let y be a continuous function. Then, the $(N+2)\times(N+2)$-matrices ${}^i_i\mathbf{M}_y^{(k)}$ are given as

$${}^i_i\mathbf{M}_y^{(0)} := \begin{pmatrix} y(x_0) & & & & \\ & y(x_1) & & & \\ & & \ddots & & \\ & & & y(x_N) & \\ & & & & y(x_{N+1}) \end{pmatrix},$$

$$
{}_i^i\mathbf{M}_y^{(1)} := \frac{1}{2h}\begin{pmatrix}
-3y(x_0) & 4y(x_0) & -y(x_0) & & & & \\
-y(x_1) & 0 & y(x_1) & & & & \\
& -y(x_2) & 0 & y(x_2) & & & \\
& & \ddots & \ddots & & & \\
& & & -y(x_N) & 0 & y(x_N) \\
& & & y(x_{N+1}) & -4y(x_{N+1}) & 3y(x_{N+1})
\end{pmatrix},
$$

$$
{}_i^i\mathbf{M}_y^{(2)} := \frac{1}{h^2}\begin{pmatrix}
2y(x_0) & -5y(x_0) & 4y(x_0) & -y(x_0) & & & \\
y(x_1) & -2y(x_1) & y(x_1) & & & & \\
& y(x_2) & -2y(x_2) & y(x_2) & & & \\
& & \ddots & \ddots & & & \\
& & & y(x_N) & -2y(x_N) & y(x_N) \\
& & -y(x_{N+1}) & 4y(x_{N+1}) & -5y(x_{N+1}) & 2y(x_{N+1})
\end{pmatrix}.
$$

Herein, $x_i = x_0 + ih$ are the $N+2$ equidistant grid points, and $h = (x_{N+1} - x_0)/(N+1)$.

Thus, no matter which boundary condition applies (Dirichlet, Neumann, second derivative, none), we end with a system $\mathbf{w}'(t) + \mathbf{A}\mathbf{w}(t) = \mathbf{f}(t)$ of ordinary first order differential equations. In Table 3 we summarise our findings.

typ	p	matrix $\mathbf{A} \in \mathbb{R}^{p\times p}$	vector $\mathbf{f}(t) = \mathbf{f}^{rhs}(t) -$
Dirichlet	N	$\mathbf{M}_a^{(2)} + \mathbf{M}_b^{(1)} + \mathbf{M}_c^{(0)}$	$\big(\mathbf{M}_a^{(2),bc} + \mathbf{M}_b^{(1),bc}\big)\mathbf{w}^{bc}(t)$
Neumann	N	${}_n^n\mathbf{M}_a^{(2)} + {}_n^n\mathbf{M}_b^{(1)} + \mathbf{M}_c^{(0)}$	$\big({}_n^n\mathbf{M}_a^{(2),bc} + {}_n^n\mathbf{M}_b^{(1),bc}\big)\mathbf{w}^{bc}(t)$
second derivative	N	${}_s^s\mathbf{M}_a^{(2)} + {}_s^s\mathbf{M}_b^{(1)} + \mathbf{M}_c^{(0)}$	$\big({}_s^s\mathbf{M}_a^{(2),bc} + {}_s^s\mathbf{M}_b^{(1),bc}\big)\mathbf{w}^{bc}(t)$
intrinsic	$N+2$	${}_i^i\mathbf{M}_a^{(2)} + {}_i^i\mathbf{M}_b^{(1)} + {}_i^i\mathbf{M}_c^{(0)}$	$\mathbf{0}$

TABLE 3. Boundary conditions. p is the number of unknowns.

Note that for homogeneous boundary conditions, i.e. $w_l(t) = w_r(t) = 0$, there holds $\mathbf{f}(t) = \mathbf{f}^{rhs}(t)$, independently of the type of boundary condition. Also note that we can arbitrarily combine the considered boundary conditions. For example, we might have no condition on the left boundary and a Neumann condition on the right boundary. The matrix $\mathbf{A}$ becomes then $\mathbf{A} = {}_i^n\mathbf{M}_a^{(2)} + {}_i^n\mathbf{M}_b^{(1)} + {}_i\mathbf{M}_c^{(0)}$ and is a $(N+1)\times(N+1)$-matrix, since there are $N+1$ unknowns $w_0(t), \ldots, w_N(t)$.

4. Time stepping

As we have seen in the previous section, the finite difference discretisation (with respect to x) of the pricing PDE (2.6) leads to the system (3.3) of ordinary differential equations satisfied by the p unknown functions $w_i(t)$. These functions approximate at the grid points x_i the value (2.2) of the financial derivative under consideration, $w_i(t) \approx V(x_i, T-t)$. The system (3.3) is a special case of the problem

$$\begin{cases} \mathbf{w}'(t) + \mathbf{A}\mathbf{w}(t) &= \mathbf{f}(t) \\ \mathbf{w}(t_j) &= \mathbf{w}_j \end{cases}$$

where $t_j \geq 0$ is arbitrary. The solution of the system at some time $t_{j+1} = t_j + k$ is given by

$$\mathbf{w}(t_{j+1}) = e^{-k\mathbf{A}}\mathbf{w}_j + k\int_0^1 e^{-k\mathbf{A}(1-s)}\mathbf{f}(t_j + ks)\mathrm{d}s \ . \tag{4.1}$$

Typically $t_j = 0$, $t_{j+1} = T$, $\mathbf{w}_j = \mathbf{w}_0 = \mathbf{g}$, $\mathbf{f} = \mathbf{0}$ and the i-th component of the vector $e^{-\mathbf{A}T}\mathbf{g}$ contains the price of the derivative at inception $t = 0$ for the value x_i of the underlying. However, calculating the matrix exponential of $-k\mathbf{A}$ is feasible (in Matlab and/or Python) only for a moderate size of $\mathbf{A}$. This fact leads to the necessity of approximating the solution of the system (3.3) rather than calculating it analytically. We achieve such an approximation by looking at the so-called Padé approximation of e^{-x}. The following definition will help us to do so.

Definition 4.1. The Padé approximation of function $f : \mathbb{R} \to \mathbb{R}$ admitting a power series around $x = 0$, i.e., $f(x) = \sum_{j=0}^{\infty} c_j x^j$ for some sequence (c_j) is a rational function of the form

$$R_{\ell,m}(x) := \frac{p_\ell(x)}{q_m(x)} := \frac{a_0 + a_1 x + \cdots + a_\ell x^\ell}{1 + b_1 x + \cdots + b_m x^m} \tag{4.2}$$

such that

$$q_m(x)f(x) - p_\ell(x) = \mathcal{O}(x^{\ell+m+1}), \quad \text{as } x \to 0 \ .$$

If this holds, we say that the order of the Padé approximation is $\ell + m$.

For $f(x) = e^{-x}$ it is possible to show (see for example [Tho06]) that the polynomials p_ℓ and q_m are given by

(4.3)

$$p_{\ell,m}(x) = \sum_{j=0}^{\ell} \frac{(\ell+m-j)!\ell!}{(\ell+m)!(\ell-j)!j!}(-x)^j, \quad q_{\ell,m}(x) = \sum_{j=0}^{m} \frac{(\ell+m-j)!m!}{(\ell+m)!(m-j)!j!}x^j \ .$$

Thus, the solution of (4.1) is, for $\mathbf{f} = 0$, approximated by

$$\mathbf{w}(t_{j+1}) \approx R_{\ell,m}(k\mathbf{A})\mathbf{w}_j$$

which is with $\mathbf{w}_{j+1} \approx \mathbf{w}(t_{j+1})$ equivalent to

$$q_{\ell,m}(k\mathbf{A})\mathbf{w}_{j+1} = p_{\ell,m}(k\mathbf{A})\mathbf{w}_j \ .$$

If $\mathbf{f} \neq 0$, we need to approximate the integral

$$k\int_0^1 e^{-k\mathbf{A}(1-s)}\mathbf{f}(t_j + ks)\mathrm{d}s$$

by the finite sum

$$k\sum_{i=1}^{p} P_i(k\mathbf{A})\mathbf{f}(t_j + ks_i) \ .$$

Here, the functions P_i are again rational and the numbers $s_i \in [0,1]$, $i = 1, \ldots, p$, are quadrature points. If the denominator of all the P_i's are equal to the denominator $q_{\ell,m}$ of the rational function $R_{\ell,m}$, then the solution of (4.1) is approximated by

$$q_{\ell,m}(k\mathbf{A})\mathbf{w}_{j+1} = p_{\ell,m}(k\mathbf{A})\mathbf{w}_j + k\sum_{i=1}^{p} p^i_{\ell,m}(k\mathbf{A})\mathbf{f}(t_j + ks_i) \ , \tag{4.4}$$

where $p^i_{\ell,m}$ is the numerator of the rational function P_i. Equation (4.4) defines a fully discrete scheme of order $\mathcal{O}(h^2 + k^{\ell+m})$ to solve (2.6) as follows. Set $\mathbf{w}_0 = \mathbf{g}$, chose $M \in \mathbb{N}^*$ and let $k = \frac{T}{M}$ be the time step as well as $t_j = jk$. Then, apply the recursion (4.4) for $j = 0, 1, \ldots, M-1$ to eventually end up with $\mathbf{w}_M$. An element $w_{i,M}$ of $\mathbf{w}_M$ is an approximation to the function w in (2.6) evaluated at (x_i, T), which corresponds to - up to a possible localisation error - the value of the derivative under consideration, i.e., $w_{i,M} \approx w(x_i, T) \approx V(x_i, 0)$. In order for the time stepping error being (at least) of the same order as of the finite-difference-method (applied to x), we must have $\ell + m = 2$.

Example 4.1. In (4.3) let $\ell = m = 1$. Thus

$$p_{1,1}(x) = 1 - \frac{1}{2}x, \quad q_{1,1}(x) = 1 + \frac{1}{2}x \ .$$

We apply the procedure described in [Tho06][Lemma 9.2] to find the P_i's belonging to the rational function $R_{1,1}$. It turns out that $p = 1$, $s_1 = \frac{1}{2}$ as well as $P_1(x) = \frac{1}{q_{1,1}(x)}$. Thus

$$\big(\mathbf{I} + \frac{1}{2}k\mathbf{A}\big)\mathbf{w}_{j+1} = \big(\mathbf{I} - \frac{1}{2}k\mathbf{A}\big)\mathbf{w}_j + k\mathbf{f}(t_j + k/2) \ , \qquad j = 0, 1, \ldots, M-1 \tag{4.5}$$

is a second-order approximation to (3.3). This scheme is called the Crank-Nicolson scheme.

For $\ell = 0$ and $m = 1$ we find $p_{0,1}(x) = 1$, $q_{0,1}(x) = 1 + x$ and obtain the implicit Euler scheme

$$\big(\mathbf{I} + k\mathbf{A}\big)\mathbf{w}_{j+1} = \mathbf{w}_j + k\mathbf{f}(t_j + k) \ ;$$

for $\ell = 1$ and $m = 0$ we have $p_{1,0}(x) = 1 - x$, $q_{0,1}(x) = 1$, which leads to the explicit Euler scheme

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

Both Euler schemes have order $\ell + m = 1$ (with respect to t) and are therefore not interesting to be combined with the second-order approximations (3.1) with respect to x. However, a Fourier analysis shows - see for example [GC06] - that the Crank-Nicolson scheme in example 4.1 is not able to dampen un-smooth payoffs appearing, for instance, in binary options (in this regard, compare also with the pricing problem in section 6.4 for another example). This, in general, reduces the order of the Crank-Nicolson scheme for such contracts typically from two to one. The implicit Euler scheme, on the other hand, is known to dampen such shocks, but converges only with first order. It was then the idea of Rannacher [Ran84] to combine these schemes to a scheme which is, even when applied to pricing problems with un-smooth g, of second order. This scheme is known as the Rannacher scheme. The idea is to apply first $R > 0$ (R is even) implicit Euler steps of step size $k/2$ and then to use the Crank-Nicolson scheme (with step size k) for the remaining $M - R/2$ steps. Thus, instead of (4.5) we do the following. Set again $\mathbf{w}_0 = \mathbf{g}$. Then, first do

$$\big(\mathbf{I} + \frac{k}{2}\mathbf{A}\big)\mathbf{w}_{(j+1)/2} = \mathbf{w}_{j/2} + \frac{k}{2}\mathbf{f}(t_{j/2} + k/2), \quad j = 0, 1, \dots, R-1 \tag{4.6}$$

(where we denote by $\mathbf{w}_{j/2}$ the approximation to $w(\cdot, t_{j/2})$, with $t_{j/2} = jk/2$), then do

$$\big(\mathbf{I} + \frac{k}{2}\mathbf{A}\big)\mathbf{w}_{j+1} = \big(\mathbf{I} - \frac{k}{2}\mathbf{A}\big)\mathbf{w}_j + k\mathbf{f}(t_j + k/2)\ , \qquad j = R/2, \dots, M-1\ . \tag{4.7}$$

Typically, $R = 2$ initial implicit Euler steps with halved time step are sufficient.

Note that we derived the Rannacher scheme (4.6)–(4.7) under the assumption that the coefficients a, b and c in (2.6) do not depend on t. However, it turns out that the scheme also applies in the case the coefficients are time-dependent. Since the (constant) matrix $\mathbf{A}$ in (3.3) becomes in this case the matrix-valued function $t \mapsto \mathbf{A}(t) := \mathbf{M}_a^{(2)}(t) + \mathbf{M}_b^{(1)}(t) + \mathbf{M}_c^{(0)}(t)$, we just need to replace the matrix $\mathbf{A}$ by $\mathbf{A}(t_{j/2} + k/2)$ in (4.6) and by $\mathbf{A}(t_j + k/2)$ in (4.7). Hence, we are able to price derivatives for example also in the important so-called local volatility model, where $\sigma(x,t)$ in (2.1) is given by $\sigma(x,t) = \sigma_{\mathrm{LV}}(x,t)x$ and the coefficient a in the PDE (2.6) becomes $a(x,t) = -\frac{1}{2}\sigma_{\mathrm{LV}}^2(x, T-t)x^2$. The construction of the local volatility function σ_{LV} from option market data is a different story and cannot be addressed here. We refer to [GJ14], for example.

5. Code

The considerations of the previous sections lead to the following pseudo-code to find the approximative value of a 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 (in this case, the functions w_l and w_r are ignored). The input parameters to the routine are the problem dependent "parameters" given by the coefficients $a(x,t)$, $b(x,t)$ and $c(x,t)$ of the PDE, the right hand side function $f(x,t)$, the initial condition $g(x)$, the domain $G =]x_l, x_r[$, and boundary conditions $w_l(t)$, $w_r(t)$ (together with n_l and n_r). Furthermore, we have to specify the discretisation parameters N (number of (inner) grid points), M (number of time steps) and R (number of initial implicit Euler time steps). The routine then returns the vector of grid points x_i as well as the functions values $w(x_i, T)$ of $w(x,t)$ at these grid points at time $t = T$.

If at least one of the coefficients a, b or c are time-dependent, we need to re-calculate the matrices $\mathbf{A}(t_j)$ and $\mathbf{M}^{bc}(t_j)$ in every time step. This is time consuming and can not be avoided. If none of the coefficients depend on t, the matrix-valued function $t \mapsto \mathbf{A}(t)$ is constant and can be evaluated once outside the loop of the time stepping. The time stepping is realised in the subroutine `rannacher`; the parameter `td` indicates wether the coefficients depend on t (`td` $\neq 0$) or not (`td` $= 0$).

In Matlab/Octave, the pseudo code may thus take the following form.

```
Define the functions $a(x,t)$, $b(x,t)$, $c(x,t)$, $f(x,t)$ and $g(x)$. Define $T > 0$.
Define $x_l, x_r \in \mathbb{R}$, $n_l, n_r \in \{0,1,2,3\}$, and $w_l(t)$, $w_r(t)$.
Chose $N, M, R \in \mathbb{N}^\times$, $R$ even. Set $k := T/M$.
Get the matrix-valued function $t \mapsto \mathbf{A}(t) := \mathbf{M}_a^{(2)}(t) + \mathbf{M}_b^{(1)}(t) + \mathbf{M}_c^{(0)}(t)$.
Get the matrix-valued function $t \mapsto \mathbf{M}^{bc}(t) := \mathbf{M}_a^{(2),bc}(t) + \mathbf{M}_b^{(1),bc}(t)$.
Set $\mathbf{w}_0 := \mathbf{g}$.
For $j = 0, 1, \ldots, R-1$,
    Set $t_j := (j+1)k/2$. Set $\mathbf{f}_j := \mathbf{f}^{rhs}(t_j) - \mathbf{M}^{bc}(t_j)\mathbf{w}^{bc}(t_j)$.
    Solve the system
    $\left(\mathbf{I} + \frac{k}{2}\mathbf{A}(t_j)\right)\mathbf{w}_{(j+1)/2} = \mathbf{w}_{j/2} + \frac{k}{2}\mathbf{f}_j$
For $j = R/2, \ldots, M-1$,
    Set $t_j = (j+1/2)k$. Set $\mathbf{f}_j = \mathbf{f}^{rhs}(t_j) - \mathbf{M}^{bc}(t_j)\mathbf{w}^{bc}(t_j)$.
    Solve the system
    $\left(\mathbf{I} + \frac{k}{2}\mathbf{A}(t_j)\right)\mathbf{w}_{j+1} = \left(\mathbf{I} - \frac{k}{2}\mathbf{A}(t_j)\right)\mathbf{w}_j + k\mathbf{f}_j$
Output the vector $\mathbf{x}$ of grid points and the function values $\mathbf{w}_M$
at these grid points.
```

TABLE 4. Description of the fully discrete scheme to solve the PDE (2.6)

```matlab
function [x,w] = pricing_1d(a,b,c,T,xl,wl,nl,xr,wr,nr,g,f,N,M,R,td)

% define grid, mesh width, time step, boundary conditions
h = (xr-xl)/(N+1); x = (xl:h:xr)'; BC = [nl nr];
if nl < 3; x(1) = []; end, if nr < 3; x(end)= []; end

% define matrix functions t -> A(t), t -> Mbc(t)
[M2,M1,M0,M2bc,M1bc] = matgen_t({{'M2',a},{'M1',b},{'M0',c}},BC,xl,xr,N);
A = @(t)M2(t)+M1(t)+M0(t); I = speye(length(x)); Mbc = @(t)M2bc(t)+M1bc(t);

% the vector valued function t -> wbc(t), time stepping
wbcl = sparse(length(I),1); wbcr = wbcl;
wbcl(1) = 1; wbcr(end) = 1; wbc = @(t)wl(t)*wbcl+wr(t)*wbcr;
w = rannacher(I,A,g(x),@(t)f(x,t)-Mbc(t)*wbc(t),T/M,M,R,td);

function w = rannacher(I,A,w,f,k,M,R,td)
if td == 0
    A = A(0); for j = 1:R, tj = j*k/2; w = (I+k/2*A)\(w+k/2*f(tj)); end
    B = I + k/2*A; C = I - k/2*A;
    for j = R/2+1:M, tj = (j-1+0.5)*k;  w = B\(C*w+k*f(tj)); end
else
    for j = 1:R, tj = j*k/2; w = (I+k/2*A(tj))\(w+k/2*f(tj)); end
    for j = R/2+1:M
        tj = (j-1+0.5)*k; Aj = A(tj); w = (I+k/2*Aj)\((I-k/2*Aj)*w+k*f(tj));
    end
```

```
end
```

The corresponding Python code looks almost the same.

```
import numpy as np; from scipy import sparse;
from scipy.sparse.linalg import spsolve; from matgen_t import matgen_t

def rannacher(I,A,w,f,k,M,R,td):
    if td == 0:
        A = A(0); B = I + k/2*A; C = I - k/2*A;
        for j in range(0,R): tj = (j+1)*k/2; w = spsolve(I+k/2*A,w+k/2*f(tj))
        for j in range(int(R/2),M): w = spsolve(B,C*w+k*f(tj));
    else:
        for j in range(0,R):
            tj = (j+1)*k/2; B = I+k/2*A(tj); w = spsolve(B,w+k/2*f(tj));
        for j in range(int(R/2),M):
           tj = (j+0.5)*k; Aj = A(tj); B = I+k/2*Aj; C = I-k/2*Aj;
           w = spsolve(B,C*w+k*f(tj));
    return w

def pricing_1d(a,b,c,T,xl,wl,nl,xr,wr,nr,g,f,N,M,R,td):
    # define grid, mesh width, time step
    h = (xr-xl)/(N+1); x = np.arange(xl,xr+h,h);
    x = x[1:] if nl<3 else x; x = x[:-1] if nr<3 else x

    # define matrix functions t -> A(t), t -> Mbc(t)
    I = sparse.eye(N+(nr==3)+(nl==3));
    Mat = matgen_t([["M2",a],["M1",b],["M0",c]],[nl,nr],xl,xr,N);
    A = lambda t:Mat[0](t)+Mat[1](t)+Mat[2](t); Mbc = lambda t: Mat[3](t)+Mat
    [4](t)

    # the vector valued function t -> wbc(t), time stepping
    wbcl = np.zeros(len(x)); wbcr = np.zeros(len(x))
    wbcl[0] = 1; wbcr[-1] = 1; wbc = lambda t:wl(t)*wbcl+wr(t)*wbcr;

    w = rannacher(I,A,g(x),lambda t:f(x,t)-Mbc(t)*wbc(t),T/M,int(M),R,td);
    return x, w
```

The subroutine `matgen_t` outputs for any of the discussed boundary conditions the matrix-valued functions $t \mapsto \mathbf{M}_y^{(k)}(t)$ and $t \mapsto \mathbf{M}_y^{(k),bc}(t)$.

6. Examples

In this section, we give examples which fit into the framework of the pricing problem (2.6). For each of these examples, we provide (uncommented) Matlab and Python codes which solve the corresponding pricing problems. Whereas both Matlab and Python yield the same prices, the computation time in Matlab is typically shorter than in Python, especially if the coefficients a, b, c of the PDE do not depend on t.

6.1. **Double knock-out call in the CEV model.**

We consider a double barrier (knock-out) call option with lower and upper barriers L and U, respectively, and strike K in the CEV model. Hence, the domain G in (2.3) is equal to $G =]L, U[$, and, since the option becomes worthless when the underlying hits L or U, the boundary conditions in (2.6) are $w_l(t) = w_r(t) = 0$ (homogeneous Dirichlet conditions). The functions μ and σ in (2.1) are in the CEV model $\mu(x,t) = (r-q)x$ and $\sigma(x,t) = \delta x^\beta$, where $\delta > 0$ and $\beta \in \mathbb{R}$ are model parameters to be found by calibration of the model to market data. Hence, the coefficients a, b, c and f in (2.6) are $a(x,t) = -\frac{1}{2}\delta^2 x^{2\beta}$, $b(x,t) = -(r-q)x$, $c(x,t) = r$ and $f \equiv 0$.

The function `doubleknockoutcall` realises this in Matlab/Octave

```
function V = doubleknockoutcall(x0,beta,delta,r,q,U,L,K,T)

a = @(x,t)-0.5*delta^2*x.^(2*beta); b = @(x,t)-(r-q)*x;
c = @(x,t)r*x.^0; g = @(x)max(x-K,0); N = 2^11-1; M = ceil(0.1*N);
[x,w] = pricing_1d(a,b,c,T,L,@(t)0,0,U,@(t)0,0,g,@(x,t)0,N,M,2,0);
V = interp1(x,w,x0,'spline');
```

respectively in Python

```
import numpy as np; from scipy.interpolate import interp1d
from pricing_1d import pricing_1d

def doubleknockoutcall(x0,beta,delta,r,q,U,L,K,T):

    a = lambda x,t:-0.5*delta**2*x**(2*beta); b = lambda x,t: -(r-q)*x;
    c = lambda x,t:r*(x)**0; f = lambda x,t:0;
    g = lambda x: np.maximum(x-K,0); N = 2**11-1; M = np.ceil(0.1*N);
    x,w = pricing_1d(a,b,c,0.5,L,lambda t:0,0,U,lambda t:0,0,g,f,N,M,2,0);
    return interp1d(x,w,kind='cubic')(x0)
```

Now consider the particular example of a stock with initial price $x_0 = 100$ which pays no dividend, $q = 0$. The CEV model parameters are chosen to be $\beta = -3$ and $\delta = 0.25x_0^{1-\beta}$. The parameters of the option are set to $L = 90$, $U = 120$, $T = 0.5$. The risk free is $r = 0.1$. We apply the function `doubleknockoutcall` to strikes $K \in \{95, 100, 105\}$ and obtain the corresponding option prices $V \in \{3.80882413, 2.50592423, 1.36967823\}$. In [DL01], the authors state the values $V \in \{3.8088, 2.5059, 1.3696\}$, which they obtain by inverting Laplace transforms.

We remark that Matlab finds the option values about 3 times faster than Python.

6.2. **Fixed-strike asian call in the BS model.** Consider a (discrete) set of $J+1$ observation points $\mathcal{T} := \{t_0, t_1, \ldots, t_J\}$ with $t_0 = 0$ and $t_J = T$. Denote by $A(T)$ the discrete average of the underlying $X(t)$ over $\mathcal{T}$, i.e.,

$$A(T) := \frac{1}{J+1}\sum_{j=0}^{J} X(t_j) .$$

The payoff of a so-called fixed strike (discretely monitored) asian call option with strike K and maturity T is then

$$g\big(A(T)\big) = \max\{A(T) - K, 0\} .$$

If we assume that the underlying X follows a geometric Brownian motion, then one can show (see [Věc02]) that the price $V(x_0,0)$ of this option at inception can be written as the product

$$V(x_0,0) = x_0 v\Big(h(T) - e^{-rT}\frac{K}{x_0}, T\Big) , \tag{6.1}$$

where the function $v = v(y,t)$ is solution of the PDE

$$\begin{cases} \partial_t v + a(y,t)\partial_{yy} v &= 0 & \text{in} & \mathbb{R}\times]0,T] \\ v(y,0) &= g(y) & \text{in} & \mathbb{R} \end{cases} , \tag{6.2}$$

and where the coefficient $a(y,t)$ as well as the initial condition g are given by

$$a(y,t) := -\frac{1}{2}\sigma^2\big(y - e^{-q(T-t)}h(t)\big)^2, \quad g(y) = \max\{y,0\} .$$

The function $t \mapsto h(t)$ appearing in $a(y,t)$ is given by

$$h(t) := \frac{1}{J+1}e^{-qt}\sum_{j=\omega(t)}^{J} e^{(-r+q)(T-t_j)} ,$$

with the step-function

$$\omega(t) := j \quad \text{if } t_j \le T-t < t_{j+1} .$$

The PDE (6.2) needs to be localised to a finite interval $]y_l, y_r[$ and boundary conditions have to be set. We thus consider the particular problem of (2.6), which reads as follows: Find $w(y,t)$ such that

$$\begin{cases} \partial_t w + a(y,t)\partial_{yy} w &= 0 & \text{in} &]y_l,y_r[\times]0,T] \\ w(y_l,t) &= 0 & \text{in} &]0,T] \\ \partial_y w(y_r,t) &= 1 & \text{in} &]0,T] \\ w(y,0) &= g(y) & \text{in} &]y_l,y_r[\end{cases} . \tag{6.3}$$

The boundary conditions as well as the values of boundaries

$$y_l = -\frac{1}{J+1}\sum_{j=0}^{n} e^{(r-q+\sigma^2/2)t_j + 4\sigma\sqrt{t_j}}, \quad y_r = h(T)$$

are derived in [Yu12]. The function `asiancall` is a realisation of (6.3) in Matlab/Octave

```matlab
function V = asiancall(x0,sigma,r,q,T,Tau,K,N,M)

tj = [eps;Tau]; omega = @(t)find(tj+t-T>0,1,'first')-1; J = length(tj)-1;
d = exp((-r+q)*(T-tj)); h = @(t)exp(-q*t)*sum(d(omega(t)+1:J+1))/(J+1);
a = @(x,t)-1/2*sigma^2*(x-exp(-q*(T-t)).*h(t)).^2; b = @(x,t)0*x; c = @(x,t)
    0*x;
g = @(x)max(0,x); yr = h(T); wr = @(t)1; wl = @(t)0;
yl = -exp(-r*T)*sum(exp((r-q+sigma^2/2)*tj+4*sigma*sqrt(tj)))/(J+1);
[y,w] = pricing_1d(a,b,c,T,yl,wl,0,yr,wr,1,g,@(x,t)0,N,M,2,1);
V = x0.*interp1(y,w,h(T)-exp(-r*T)*K./x0);
```

and in Python

```python
import numpy as np; from scipy.interpolate import interp1d
from pricing_1d import pricing_1d

def asiancall(x0,sigma,r,q,T,Tau,K,N,M):

    tj = np.hstack((0,Tau)); omega = lambda t: (tj+t-T>0).nonzero()[0][0];
    J = len(tj)-1; d = np.exp((-r+q)*(T-tj));
    h = lambda t: np.exp(-q*t)*np.sum(d[omega(t)-1*(t==T):J+2])/(J+1);
    a = lambda x,t: -0.5*sigma**2*(x-np.exp(-q*(T-t))*h(t))**2; b = lambda x,
    t:0*x;
    c = lambda x,t:0*x; g = lambda x:np.maximum(x,0); yr = h(T); wr = lambda
    t:1;
    yl = -np.exp(-r*T)*np.sum(np.exp((r-q+sigma**2/2)*tj+4*sigma*np.sqrt(tj))
    )/(J+1);
    y,w = pricing_1d(a,b,c,T,yl,lambda t:0,0,yr,wr,1,g,lambda t,x:0,N,M,2,1);
    return x0*interp1d(y,w,kind='linear')(h(T)-np.exp(-r*T)*K/x0)
```

Here, `Tau` is a vector containing the observation points $\mathcal{T} = \{t_1, t_2, \ldots, t_J\}$ (without $t_0 = 0$). We now consider the particular example where $t_j = \frac{j}{250}T$, $j = 0, 1, \ldots, 250$, $T = 1$, and $K \in \{90, 100, 110\}$[1]. For the values $x_0 = 100$, $\sigma = 0.17801$, $r = 0.0367$ and $q = 0$ we find by applying the function `asiancall` with $N = 2^{11} - 1$ (inner) grid points and $M = J = 250$ time steps the option values $V(x_0, 0) \doteq \{11.940566, 4.952142, 1.413360\}$. In [LX12], the authors state the values $V(x_0, 0) = \{11.940563, 4.952157, 1.414467\}$, which they obtain by applying a transform method. Matlab/Octave is about 30% faster than Python.

6.3. **Receiver-swaption in the CIR++ model.**

We consider the so-called CIR++ model, where the short rate $X(t)$ is given by $X(t) = Y(t) + \varphi(t)$. Here, $Y(t)$ is a stochastic process satisfying the CIR SDE (2.5), and $\varphi(t)$ is a deterministic function (called shift) chosen such that the zero-bond curve at $t = 0$ implied by the model X coincides with the zero-bond curve of the market. It turns out then that φ must be equal to the difference of the forward curve of the market and the forward curve implied by the model Y, i.e., $\varphi(t) = f^{\mathrm{M}}(0, t) - f^{\mathrm{Y}}(0, t)$, see for example [BM06].

We price a swaption with strike K and maturity $T = T_0$ written on a receiver-swap with tenor $\mathcal{T} = \{T_1, T_2, \ldots, T_J\}$, where $T < T_1$. For a description of these financial products, see again [BM06]. The value $V(x, t)$ of the swaption solves the PDE

$$\left\{ \begin{array}{rcll} \partial_t V + \mathcal{A}(t)V - xV & = & 0 & \text{in} \quad G \times [0, T[\\ V(x, T) & = & g(V_{\mathrm{rS}}(T)) & \text{in} \quad G \end{array} \right.$$

with $g(x) = \max\{x, 0\}$. Herein, $\mathcal{A}(t)$ is the (time-dependent) generator of the process X

$$\mathcal{A}(t) = \frac{1}{2}\sigma^2 \big(x - \varphi(t)\big)\partial_{xx} + \big(\kappa(m - x + \varphi(t)) + \varphi'(t)\big)\partial_x$$

[1]We have to solve the PDE (6.2) only once and obtain the option values for an arbitrary number of (different) strikes by (6.1) and interpolation.

and $V_{\mathrm{rS}}(T)$ is the value at $t = T$ of the corresponding receiver swap, which is (we assume that the nominal is equal to 1) given by

$$\begin{aligned} V_{\mathrm{rS}}(T) &= P(T,T_J) - P(T,T_0) + K\sum_{j=1}^{J}(T_j - T_{j-1})P(T,T_j) \\ &= K\sum_{j=1}^{J-1}(T_j - T_{j-1})P(T,T_j) + \big(1 + K(T_J - T_{J-1})\big)P(T,T_J) - 1 \;. \qquad (6.4) \end{aligned}$$

Herein, every zero-bond with price $P(T,T_j) := V_j(x,t)$ solves the PDE

$$\left\{\begin{array}{rcll} \partial_t V_j + \mathcal{A}(t)V_j - xV_j &=& 0 & \text{in } G \times [T, T_j[\\ V_j(x,T_j) &=& 1 & \text{in } G \end{array}\right. .$$

Hence, to find the value of the swaption with the proposed algorithm, we switch to time-to-maturity and restrict the domain G to $]x_l, x_r[$. We then solve first for the J zero-bonds, conduct the summation (6.4) to get the payoff $g(V_{\mathrm{rS}}(T))$ and finally solve the PDE for the swaption. The pricing equation of the j-th zero-bond becomes a particular example of the PDE (2.6), with source-term $f \equiv 0$, maturity $T_j - T_0$ and coefficients

$$\begin{aligned} a(x,t) &= -\frac{1}{2}\sigma^2\big(x - \varphi(T_j - t)\big) \;, \\ b(x,t) &= -\big(\kappa(m - x + \varphi(T_j - t)) + \varphi'(T_j - t)\big) \;, \\ c(x,t) &= x \;. \end{aligned}$$

Furthermore, the pricing equation of the swaption itself is again of the form (2.6), with maturity T_0 and coefficients as above, where T_j has to be replaced by T_0. In all of the $J + 1$ PDEs, we let $x_r = 2$ and chose no boundary condition on x_l and a homogenous Neumann condition on x_r. It remains to specify x_l. As the function $a(x,t)$ must be non-positive, we need $x \geq \varphi(T_j - t)$, $t \in [0, T_j - T]$. If we assume that φ is strictly increasing on $[T, T_j]$, then the minimum of $\varphi(T_j - t)$ is attained at $t = T_j - T$, such that we let $x_l = \varphi(T)$.

We need to specify the deterministic shift $\varphi(t) = f^{\mathrm{M}}(0,t) - f^{\mathrm{Y}}(0,t)$ appearing in the pricing equations. Since the CIR model is affine, the forward curve of Y is known to be

$$f^{\mathrm{Y}}(0,t) = \partial_T A(0,t) + \partial_T B(0,t)y_0 \;,$$

where ∂_T means taking the derivative with respect to the second argument and where the functions $A(t,T)$ and $B(t,T)$ are given by

$$A(t,T) := -\frac{2\kappa m}{\sigma^2}\ln\frac{2de^{(\kappa+d)(T-t)/2}}{Z}, \quad B(t,T) := 2\frac{e^{d(T-t)} - 1}{Z} \;,$$

with $Z := (\kappa + d)(e^{d(T-t)} - 1) + 2d$ and $d := \sqrt{\kappa + 2\sigma^2}$, compare for example with [BM06]. The forward curve $f^{\mathrm{M}}(0,t)$ of the market is given by $f^{\mathrm{M}}(0,t) = -\partial_T \ln P^{\mathrm{M}}(0,t)$, such that we need the zero-bond curve $t \mapsto P^{\mathrm{M}}(0,t)$ of the market. This mapping can be obtained by market prices P_i^{M} of zero-bonds with maturity t_i and some interpolation procedure. We simplify things by assuming that $P^{\mathrm{M}}(0,t)$ is given as $P^{\mathrm{M}}(0,t) = e^{-\alpha t+\beta(1-e^{-\gamma t})}$ for positive constants α, β, γ, compare with figure 1. Note that since also $\varphi'(t)$ appears in the coefficients a and b of the pricing equations, we will also need to calculate the second derivatives $\partial_{TT}A(t,T)$, $\partial_{TT}B(t,T)$ and $\partial_T f^{\mathrm{M}}(0,T) = -\partial_{TT}\ln P^{\mathrm{M}}(0,T)$.

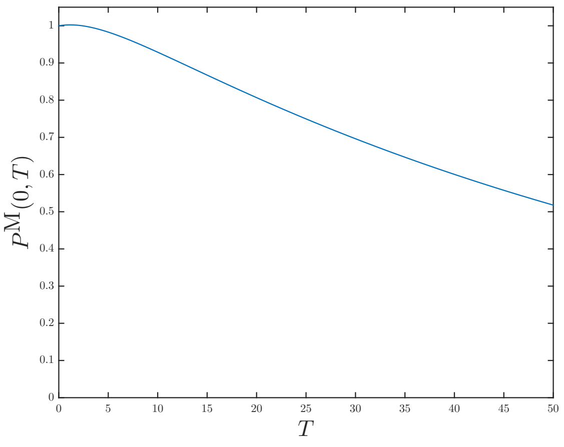

FIGURE 1. The hypothetical zero-bond curve $T \mapsto P^{\mathrm{M}}(0,T) = e^{-\alpha T+\beta(1-e^{-\gamma T})}$ of the market. We take the parameter values $\alpha = 0.014806$, $\beta = 0.082234$, $\gamma = 0.235463$.

The function `receiverswaption` returns the value of the swaption. Here, the input parameter `Tau` is a vector corresponding to $\mathcal{T}$, `fM` and `fM_t` are the forward curve of the market $f^{\mathrm{M}}(0,T)$ and its derivative $\partial_T f^{\mathrm{M}}(0,T)$, respectively. They need to be entered as functions (via `@` in Matlab, via `lambda` in Python).

```
function V = receiverswaption(y0,kappa,m,sigma,K,T,Tau,fM,fM_t)

d = sqrt(kappa^2+2*sigma^2); Z = @(x,y)(kappa+d)*(exp(d*(y-x))-1)+2*d;
A_T = @(x,y)kappa*m*(d^2-kappa^2)/sigma^2*(exp(d*(y-x))-1)./Z(x,y);
A_TT = @(x,y)2*d^2*kappa*m*(d^2-kappa^2)/sigma^2*exp(d*(y-x))./Z(x,y).^2;
B_T = @(x,y)4*d^2*exp(d*(y-x))./Z(x,y).^2;
B_TT = @(x,y)-4*d^3*exp(d*(y-x)).*((kappa+d)*(exp(d*(y-x))+1)-2*d)./Z(x,y)
    .^3;
fY = @(x)A_T(0,x)+B_T(0,x)*y0; fY_t = @(x)A_TT(0,x)+B_TT(0,x)*y0;
phi = @(x)fM(x)-fY(x); phi_t = @(x)fM_t(x)-fY_t(x);

g = @(x)1; wl = @(t)0; wr = @(t)0; xl = phi(T); c = @(x,t)x; Tau = [T;Tau];
z = K*diff(Tau); z(end) = 1+z(end); N = 2^10-1; M = ceil(0.05*N); V = -1;

for j = 1:length(Tau)-1
    a = @(x,t)-0.5*sigma^2*(x-phi(Tau(j+1)-t));
    b = @(x,t)-kappa*(m-x+phi(Tau(j+1)-t))-phi_t(Tau(j+1)-t);
    [x,w] = pricing_1d(a,b,c,Tau(j+1)-Tau(1),xl,wl,3,2,wr,1,g,@(x,t)0,N,M
    ,2,1);
    V = V+z(j)*w;
end
```

```

a = @(x,t)-0.5*sigma^2*(x-phi(Tau(1)-t)); g = @(x)max(V,0); M = ceil(0.15*N);
b = @(x,t)-kappa*(m-x+phi(Tau(1)-t))-phi_t(Tau(1)-t);
[x,w] = pricing_1d(a,b,c,Tau(1),xl,wl,3,2,wr,1,g,@(x,t)0,N,M,2,1);
V = interp1(x,w,y0+phi(0));
```

```
import numpy as np; from scipy.interpolate import interp1d
from pricing_1d import pricing_1d

def receiverswaption(y0,kappa,m,sigma,K,T,Tau,fM,fM_t):

    d = np.sqrt(kappa**2+2*sigma**2); Z = lambda x,y:(kappa+d)*(np.exp(d*(y-x
    ))-1)+2*d;
    A_T = lambda x,y:kappa*m*(d**2-kappa**2)/sigma**2*(np.exp(d*(y-x))-1)/Z(x
    ,y);
    A_TT = lambda x,y:2*d**2*kappa*m*(d**2-kappa**2)/sigma**2*np.exp(d*(y-x))
    /Z(x,y)**2;
    B_T = lambda x,y:4*d**2*np.exp(d*(y-x))/Z(x,y)**2;
    B_TT = lambda x,y:-4*d**3*np.exp(d*(y-x))*((kappa+d)*(np.exp(d*(y-x))+1)
    -2*d)/Z(x,y)**3;
    fx = lambda x:A_T(0,x)+B_T(0,x)*y0; fx_t = lambda x:A_TT(0,x)+B_TT(0,x)*
    y0;
    phi = lambda x:fM(x)-fx(x); phi_t = lambda x:fM_t(x)-fx_t(x); g = lambda
    x:x**0;

    c = lambda x,t:x; wl = lambda t:0; wr = lambda t:0; xl = phi(T);
    Tau = np.hstack((T,Tau)); tau = np.diff(Tau); z = K*tau; z[-1] = 1+z[-1];
    N = 2**10-1; M = np.ceil(0.05*N); V = -1;

    for j in range(0,len(Tau)-1):
        a = lambda x,t:-0.5*sigma**2*(x-phi(Tau[j+1]-t));
        b = lambda x,t:-kappa*(m-x+phi(Tau[j+1]-t))-phi_t(Tau[j+1]-t);
        x,w = pricing_1d(a,b,c,Tau[j+1]-Tau[0],xl,wl,3,2,wr,1,g,lambda x,t:0,
    N,M,2,1);
        V = V+z[j]*w;

    a = lambda x,t:-0.5*sigma**2*(x-phi(Tau[0]-t)); g = lambda x:np.maximum(V
    ,0);
    b = lambda x,t: -kappa*(m-x+phi(Tau[0]-t))-phi_t(Tau[0]-t); M = np.ceil
    (0.15*N);
    x,w = pricing_1d(a,b,c,Tau[0],xl,wl,3,2,wr,1,g,lambda x,t:0,N,M,2,1);
    return interp1d(x,w,kind='linear')(y0+phi(0))
```

For the particular case $y_0 = 0.03$, $K = 0.02$, $T = T_0 = 2$, $\mathcal{T} = \{3,4,5,6,7\}$ and $\kappa = 2$, $m = 0.02$, $\sigma = 0.1$, `receiverswaption` returns $V \doteq 0.06034864$. In the CIR++ model, the value of the swaption can be obtained analytically (via Jamshidian's trick, see [BM06]); the exact value is $V \doteq 0.06034871$. Again, Matlab is about 30% faster than Python.

6.4. **Express certificate in the BS model.** Express certificates are structured products belonging to the group of so-called yield enhancement products. A typical specification of the redemption of such a product is as follows. If there was no early redemption, then the holder of the product receives (at the redemption date)

$$1_{\{X(T)\leq B\}}\frac{X(T)}{X(0)}N + 1_{\{B<X(T)\}}N \ ,$$

where $X(t)$ denotes the level of the underlying at t, $B < X(0)$ is some barrier and N is the denomination. If an early redemption occurs at a (pre-defined) autocall observation date t_j^A, the product expires immediately and the holder receives (at the corresponding early redemption date t_j^R) the denomination plus an early redemption coupon amount $N + c_j$. Typically, there are $J > 1$ autocall observation dates $\mathcal{T}^A := \{t_1^A, t_2^A, \ldots, t_J^A\}$ and the last of these dates is equal to the maturity of the product, $T = t_J^A$. Furthermore, the j-th redemption date satisfies $t_j^R > t_j^A$, and the time span $\delta_j := t_j^R - t_j^A$ between an autocall observation date and its corresponding early redemption date (typically a few days) has taken into account when it comes to the pricing of the product (compare with the payoff functions g_j below).

An early redemption occurs if the underlying at the autocall observation date t_j^A is above its so-called autocall trigger level $B < L_j \leq X(0)$, i.e., if $X(t_j^A) > L_j$. Now assume that the product was not early redeemed at t_{J-1}^A. Then, since it could be early redeemed at $t_J^A = T$, the redemption of the product at $t_J^R > T$ is

$$g(X(T)) = 1_{\{X(T)\leq B\}}\frac{N}{X(0)}X(T) + 1_{\{B<X(T)\leq L_J\}}N + 1_{\{X(T)>L_J\}}(N + c_J) \ . \tag{6.5}$$

To price this product, we thus need to solve a sequence of J PDEs: For $j = J, J-1, \ldots, 1$ find $V_j(x,t)$ such that

$$\left\{\begin{array}{rcll} \partial_t V_j + \mathcal{A}V_j - rV_j &=& 0 & \text{in} \quad G\times]t_{j-1}^A, t_j^A] \\ V_j(s, t_j^A) &=& g_j & \text{in} \quad G \end{array}\right. \ , \tag{6.6}$$

where $G = \mathbb{R}^+$. Herein, we start with the payoff function

$$g_J(x) = \Big(1_{\{x\leq B\}}\frac{N}{X(0)}x + 1_{\{B<x\leq L_J\}}N + 1_{\{x>L_J\}}(N + c_J)\Big)e^{-r\delta_J} \ ,$$

compare with (6.5), and the remaining payoff functions depend wether the underlying at t_j^A is above or below the autocall trigger level L_j, i.e., for $j = J-1, J-2, \ldots, 1$

$$g_j(x) = (N + c_j)1_{\{x>L_j\}}e^{-r\delta_j} + V_{j+1}(x, t_j^A)1_{\{x\leq L_j\}} \ .$$

The value V of the product at inception is then given by $V_1(x_0, 0)$.

We now consider the particular example of such a product (with ISIN CH0401668162) on the EURO STOXX 50, where $t = 0$ corresponds to 15/02/18 with $X(0) = x_0 = 3389.63$ and where $J = 6$, $N = 1000$,

$$L_j = (1.04 - 0.04j)X(0) \ ,$$

$B = 0.55X(0)$ and $c_j = 55j$, $j = 1, \ldots, J$. Furthermore, the autocall observation dates and early redemption dates are $t_1^A = 15/02/19$, $t_2^A = 17/02/20$, $t_3^A = 15/02/21$, $t_4^A = 15/02/22$, $t_5^A = 15/02/23$, $t_6^A = 15/02/24$ as well as $t_1^R = 20/02/19$, $t_2^R =$

20/02/20, $t_3^R = 19/02/21$, $t_4^R = 18/02/22$, $t_5^R = 20/02/23$, $t_6^R = 22/02/24$. To measure the time spans $\tau_j := t_j^A - t_{j-1}^A$ and δ_j, we need to agree on any of the available day-count conventions. Here, we apply the so-called 30/360 European rule, which leads to the sets (all values need to be divided by 360 to get year fractions)

$$\begin{aligned}\mathcal{T}^A &= \{t_1^A, \dots, t_J^A\} = \{360, 722, 1080, 1440, 1800, 2160\}\ ,\\ \mathcal{T}^R &= \{t_1^R \dots, t_J^R\} = \{365, 725, 1083, 1443, 1805, 2167\}\ .\end{aligned}$$

To model the time evolution of the underlying, we use the Black-Scholes model, such that the generator $\mathcal{A}$ in the sequence (6.6) is as in example 2.1, i.e., $\mathcal{A} = \frac{1}{2}\sigma^2 x^2 \partial_{xx} + (r-q)x\partial_x$. To solve the sequence of PDEs (6.6), we switch to time-to-maturity $v_j(x,t) := V_j(x, t_j^A - t)$, $t \in]t_{j-1}^A, t_j^A]$ and need to restrict to a bounded domain $[0, x_r[$, where we apply no boundary condition on $x = 0$ and a homogeneous Neumann condition on $x = x_r$. Thus, each of the (localised) PDEs in (6.6) is of the form (2.6),

$$\begin{cases} \partial_t w_j - \mathcal{A} w_j + r w_j &= 0 \quad \text{in} \quad [0, x_r[\times]0, \tau_j] \\ \partial_x w_j(x_r, t) &= 0 \quad \text{in} \quad]0, \tau_j] \\ w_j(x, 0) &= g_j \quad \text{in} \quad [0, x_r[\end{cases} .$$

Here, j runs from J to 1, $\tau_j = t_j^A - t_{j-1}^A$ with $t_0 := 0$, and

$$g_j(x) = (N + c_j)1_{\{x > L_j\}} e^{-r\delta_j} + w_{j+1}(x, \tau_{j+1}) 1_{\{x \le L_j\}}, \quad j = J-1, J-2, \dots, 1\ .$$

The value of the express certificate at $t = 0$ is then $V_1(x_0, 0) = v_1(x_0, t_1^A) \approx w_1(x_0, t_1^A)$. The function `expresscertificate` is a realisation of the above sequence. Here, the inputs `L,C,TauR,TauA` are vectors of length J involving L_j, c_j, t_j^R and t_j^A, respectively.

```
function V = expresscertificate(x0,sigma,r,q,Nom,B,L,C,TauR,TauA)

D = exp(-r*(TauR-TauA)); Tau = [0,TauA]; tau = diff(Tau); xr = 4*x0;
a = @(x,t)-1/2*sigma^2*x.^2; b = @(x,t)-(r-q)*x; c = @(x,t)r*x.^0;
wl = @(t)0; wr = wl; N = 2^11-1; M = ceil(0.05*N);
g = @(x)(Nom*x./x0.*(x<=B)+Nom*(B<x).*(x<=L(end))+(x>L(end))*(Nom+C(end)))*D(
    end);

for j = length(tau):-1:1
    [x,w] = pricing_1d(a,b,c,tau(j),0,wl,3,xr,wr,1,g,@(x,t)0,N,M,2,0);
     g = @(x)(Nom+C(j-1))*(x>L(j-1))*D(j-1)+w.*(x<=L(j-1));
end
V = interp1(x,w,x0);
```

```
import numpy as np; from scipy.interpolate import interp1d
from pricing_1d import pricing_1d

def expresscertificate(x0,sigma,r,q,Nom,B,L,C,TauR,TauA):

    D = np.exp(-r*(TauR-TauA)); Tau = np.hstack((0,TauA)); tau = np.diff(Tau)
    ;
    a = lambda x,t:-1/2*sigma**2*x**2; b = lambda x,t:-(r-q)*x; c = lambda x,
    t:r*x**0;
    xr = 4*x0; wl = lambda t:0; wr = lambda t:0; N = 2**11-1; M = np.ceil
    (0.05*N);
    g = lambda x: (Nom*x/x0*(x<=B)+Nom*(B<x)*(x<=L[-1])+(x>L[-1])*(Nom+C[-1])
    )*D[-1];

    for j in range(len(tau),0,-1):
        [x,w] = pricing_1d(a,b,c,tau[j-1],0,wl,3,xr,wr,1,g,lambda x,t:0,N,M
    ,2,0);
        g = lambda x:(Nom+C[j-1])*(x>L[j-1])*D[j-1]+w*(x<=L[j-1]);

    return interp1d(x,w,kind='linear')(x0)
```

At $t = 0$, we take from Bloomberg the values $\sigma = 0.173$, $q = 0.0336$ and $r = 0.00685$. The function `expresscertificate` then returns $V_1(x_0, 0) \doteq 973.66$, whereas on the term sheet of the product the issuer states the value 981.50. Matlab is about 3 times faster than Python.

References

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

[BL02] S. I. Boyarchenko and S.Z. Levendorskii, *Non-Gaussian Merton-Black-Scholes Theory*, Advanced Series on Statistical Science & Applied Probability, vol. 9, World Scientific, 2002.

[BM06] D. Brigo and F. Mercurio, *Interest Rate Models - Theory and Practice*, Springer, 2006.

[BMP13] D. Brigo, M. Morini, and A. Pallavacini, *Counterparty Credit Risk, Collateral and Funding*, Wiley Finance Series, Wiley, 2013.

[Bos14] S. Bossu, *Advanced Equity Derivatives*, Wiley Finance Series, Wiley, 2014.

[BR02] T. R. Bielecki and M Rutkowski, *Credit Risk: Modeling, Valuation and Hedging*, Springer, 2002.

[Cré13] St. Crépey, *Financial Modeling*, Springer, 2013.

[CT03] R. Cont and P. Tankov, *Financial Modelling With Jump Processes*, Financial Mathematics Series, Chapman&Hall/CRC, 2003.

[DL01] D. Davydov and V. Linetsky, *Pricing and Hedging Path-Dependent Options Under the CEV Process*, Management Science **47** (2001), no. 7, 949–965.

[DS06] F. Delbaen and W. Schachermayer, *The Mathematics of Arbitrage*, Springer, 2006.

[Duf06] D. Duffy, *Finite difference methods in financial engineering*, Wiley Finance Serie, Wiley, 2006.

[ET11] E. Ekström and J. Tysk, *Boundary conditions for the single-factor term structure equation*, The Annals of Applied Probability **21** (2011), no. 1, 332–350.

[Fil09] D. Filipović, *Term-Structure Models*, Springer, 2009.

[GC06] M. B. Giles and R. Carter, *Convergence analysis of Crank-Nicolson and Rannacher time-marching*, Journal of Computational Finance **9** (2006), no. 4, 89–112.

[GJ14] J. Gatheral and A. Jacquier, *Arbitrage-free SVI volatility surfaces*, Quantitative Finance **14** (2014), no. 1, 59–71.

[Gla03] P. Glasserman, *Monte Carlo Methods in Financial Engineering*, Applications of Mathematics, vol. 53, Springer, 2003.

[Hir13] A. Hirsa, *Computational Methods in Finance*, Financial Mathematics Series, Chapman&Hall/CRC, 2013.

[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.

[JL14] Guyon J. and P. H. Labordére, *Nonlinear Option Pricing*, Financial Mathematics Series, Chapman&Hall/CRC, 2014.

[Kwo08] Y. K. Kwok, *Mathematical Models of Financial Derivatives*, Second Edition, Springer, 2008.

[Lab09] P. H. Labordére, *Analysis, Geometry, and Modeling in Finance*, Financial Mathematics Series, Chapman&Hall/CRC, 2009.

[LX12] S. Levendorskii and J. Xie, *Pricing of Discretely Sampled Asian Options Under Lévy Processes*, 2012. Available at SSRN: http://papers.ssrn.com/abstract=2088214.

[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)

[PS06] G. Peskir and A. Shiryaev, *Optimal Stopping and Free-Boundary Problems*, Birkhäuser, 2006.

[Ran84] R. Rannacher, *Finite element solution of diffusion problems with irregular data*, Numerische Mathematik **43** (1984), no. 2, 309–327.

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

[Věc02] J. Věcěr, *Unified Asian Pricing*, Risk **15** (2002), no. 6, 113–116.

[Yu12] B. Yu, *Two Efficient Parameterized Boundaries for Věcěr's Asian Option Pricing PDE*, Acta Mathematicae Applicatae Sinica **28** (2012), no. 4, 643–652.

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

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