Ramy Tanios
MSc. Computational Science
Department of Mathematics
ETH Zürich
Zürich 8092, Switzerland
rtanios@ethz.ch

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

C++ Implementation of The Finite Element Method In Option Pricing Under Feller–Lévy Models

261-5110-00L : **Term Paper** : Dr. Kristin Kirchner
Due Friday, January 31, 2020

Contents

Introduction **3**

Theoretical Derivations **4**
Finite Element Spatial Discretization . . . 5
Sinc Quadrature . . . 5
Time Discretization . . . 6
Iterative Methods for Solving Linear Systems . . . 6

C++ Object-Oriented Implementation **7**
Galerkin Matrices Assembly . . . 7
Sinc Quadrature . . . 9
Dirichlet Boundary Conditions . . . 11
Time Stepping . . . 11

Numerical Experiments **14**
The 1D Fractional Heat Equation - Convergence Results . . . 14
The 2D Fractional Heat Equation . . . 17
An Application to Option Pricing . . . 17
Comparison of Direct and Iterative Solvers . . . 19

Conclusion **21**

Appendix **23**
Additional Convergence Results . . . 23

Code

1 1D assembly of the mass matrix with general coefficient γ. 7
2 Element matrix provider base class. 7
3 **Q** matrix class. 9
4 **Q** matrix-vector multiplication. 10
5 Boundary edge selector. 11
6 Time stepper class. 11
7 The conjugate gradient descent. 12
8 1D uniform refinement. 14
9 Condition number computation in C++. 20

List of Figures

1 The L^2-norm of the discretization error versus the mesh size h for the Explicit Euler, Implicit Euler and the Crank–Nicolson schemes. 16
2 The L^∞-norm of the discretization error versus the mesh size h for the Explicit Euler, Implicit Euler and the Crank–Nicolson schemes. 17
3 The solution of the 2D fractional heat equation. 18
4 The price of European options under Feller–Lévy models (orange), together with the payoff (blue). For comparison: the corresponding price values in the Black-Scholes market (green). 19
5 Evolution of the condition number κ for the case of the fractional heat equation and the pricing problem for N=50 and N=70. The solid lines correspond to the CGD and the dashed lines correspond to the PCGD. 20
6 The L^2-norm of the discretization error versus the mesh size h for $\theta \in \{1/4, 3/4\}$. 23
7 The L^∞-norm of the discretization error versus the mesh size h for $\theta \in \{1/4, 3/4\}$. 23

List of Tables

1 Computational time comparison for the LU-factorization, CGD and PCGD. 19

Introduction

In mathematical finance and especially in option pricing, price processes are typically modeled as solutions to stochastic differential equations driven either by Brownian motions in order to obtain continuous-in-time price processes[1], or by Poisson processes if sample paths of the price processes are assumed to exhibit a finite number of jumps[6].

More specifically, in the case of a Brownian motion, the stock price is modeled by a large number of independent price changes under the assumption of finite variance. Dropping the latter assumption, the Generalised Central Limit Theorem[8] states that the sum of i.i.d. random variables with heavy-tailed probability distributions, after appropriate shifting and scaling, follows a Lévy-stable distribution. In addition to that, the sequence of partial sums of the i.i.d. random variables then converges to a process that has jumps and indeed not to a Brownian Motion.

Motivated by that, we model the stock price by the sum of stochastic increments that follow a limiting Lévy-stable distribution. More specifically, a Lévy price process $(X_t)_{t\geq 0}$ with $X_{t^-} = \lim\limits_{s\to t, s<t} X_s$ for càdlàg processes, solves the Lévy-driven stochastic differential equation

$$\mathrm{d}X_t = b(X_{t^-})\mathrm{d}t + a(X_{t^-})^{1/\alpha}\mathrm{d}L_t^\alpha,\ X_0 = x, \tag{1}$$

where $(L_t^\alpha)_{t\geq 0}$ denotes an $\alpha-$stable Lévy process, $\alpha \in (0, 2]$.

For option pricing under Feller–Lévy models, we apply in this project deterministic numerical methods targeting the Kolmogorov forward equation in contrast to the widely used and non-deterministic Monte Carlo methods that rely on random numbers generation.

More precisely, we use the **Feynman–Kac** theorem to state the parabolic partial differential equation that governs the option price process, where the infinitesimal generator of the Feller process gives rise to a fractional power of the Laplace operator and thus to a fractional heat equation. The fractional Laplacian can then be represented in integral form in terms of the non-fractional Laplacian[7]. We first present the Finite Element semi-discrete form (i.e., after discretization in space) with a special treatment of the fractional Laplacian, where the integral representation is approximated by a sinc quadrature[2]. Afterwards, we use the θ-scheme for the temporal discretization in order to formulate the fully discrete problem. For the implementation part, we present the most important aspects of the object-oriented C++ implementation of the Finite Element method. We also compare different direct and iterative methods for solving linear systems. Finally, we run a convergence analysis of our FEM discretization with regular mesh refinement and for different combinations of the parameters θ and the fractional power of the Laplacian, and present results corresponding to finance applications.

Theoretical Derivations

The arbitrage-free price of financial products with payoff g at time $t \in [0,T]$, is given by

$$V(t,x) = \mathbb{E}[\mathrm{e}^{-rT} g(X_T) | X_t = x],$$

where r is the risk-free interest rate. Transforming to time-to-maturity, the **Feynman–Kac** theorem states that the price $v(t,x) := V(T-t,x)$ is given by the Kolmogorov forward equation

$$\begin{cases} v_t(t,x) - \mathcal{A}v(t,x) + rv(t,x) = 0, & \text{in } \mathbb{R}_{>0} \times \mathbb{R}_{>0}, \\ v(0,x) = g(x), & \forall\ x \in \mathbb{R}_{>0}. \end{cases} \tag{2}$$

In equation (2), the differential operator $\mathcal{A}$ is the generator of the Lévy price process $(X_t)_{t\geq 0}$ that solves the Lévy-driven stochastic differential equation

$$\mathrm{d}X_t = b(X_{t^-})\mathrm{d}t + a(X_{t^-})^{1/\alpha}\mathrm{d}L_t^\alpha, \quad X_0 = x,$$

where $(L_t^\alpha)_{t\geq 0}$ denotes an $\alpha-$stable Lévy process, $\alpha \in (0,2]$, and a, b some appropriate functions. The solution of the latter stochastic differential equation is a Feller process[3]. Assuming that there is an increasing concave function $\kappa(x)$ on $[0,+\infty)$ such that[9]

$$\kappa(0)=0, \quad \int_0^\infty \kappa^{-1}(x)\mathrm{d}x = +\infty \quad \text{and} \quad |b(x)-b(y)| \leq \kappa(|x-y|) \quad \forall x,y \in \mathbb{R},$$

holds, and if there exists an increasing function $\rho(x)$ on $[0,+\infty)$ such that

$$\rho(0)=0, \quad \int_0^\infty \rho(x)^{-1}\mathrm{d}x = \infty \quad \text{and} \quad |a(x)^{1/\alpha} - a(y)^{1/\alpha}|^\alpha \leq \rho(|x-y|) \quad \forall x,y \in \mathbb{R},$$

holds, Komatsu[5] showed that, for $\alpha \in (1,2)$, (1) admits a pathwise unique solution. For $\alpha \in (0,1]$, if additionally $\rho(x)$ is concave and satisfies $|a(x)^{1/\alpha} - a(y)^{1/\alpha}| \leq \rho(|x-y|)$ for all $x, y \in \mathbb{R}$, then we obtain existence and uniqueness of the solution to (1). On the other side, one can show that $(X_t)_{t\geq 0}$ is a Feller process[3] and that

$$\mathcal{A} = b(x)\partial_x - a(x)(-\partial_{xx})^{\alpha/2}. \tag{3}$$

Combining equations (2) and (3), the parabolic pricing PDE is given by

$$\begin{cases} v_t(t,x) - b(x)\partial_x v(t,x) + a(x)(-\partial_{xx})^{\alpha/2}v(t,x) + rv(t,x) = 0, & \text{in } \mathbb{R}_{>0} \times \mathbb{R}_{\geq 0}, \\ v(0,x) = g(x), & \forall x \in \mathbb{R}_{\geq 0}. \end{cases}$$

More generally, given a reaction-diffusion linear elliptic and symmetric second-order differential operator $\mathcal{L}$ with appropriate boundary conditions, we consider after truncating the price domain to the open domain $D := (0,R)$, the homogeneous Dirichlet parabolic problem

$$\begin{cases} v_t(t,x) + \tilde{b}(x)\partial_x v(t,x) + \tilde{c}(x)v(t,x) + \tilde{a}(x)\mathcal{L}^\beta v(t,x) = 0, & \text{in } \mathbb{R}_{>0} \times\ D, \\ v(t,0) = v(t,R) = 0, & \forall t\ \in\ \mathbb{R}_{>0}, \\ v(0,x) = g(x), & \forall x \in \mathbb{R}_{\geq 0}. \end{cases} \tag{4}$$

where $R > 0$, $\tilde{a}(x) > 0\ \forall x \in D$ and $\beta \in (0,1)$.

From the theory of semigroups of linear operators, and given $\alpha \in (0,1)$ and a Banach space X, the negative fractional power of $\mathcal{L} : X \to X$, namely $\mathcal{L}^{-\alpha}$ is given by the integral[7]

$$\mathcal{L}^{-\alpha} := \frac{\sin(\alpha\pi)}{\pi}\int_0^\infty t^{-\alpha}(t\mathcal{I} + \mathcal{L})^{-1}\mathrm{d}t,$$

where $\mathcal{I} : X \to X$ is the identity operator. One can show that this integral converges in Bochner sense with respect to $L(X)$, where $L(X)$ is the space of linear operators on X. On the other side, $\mathcal{L}^{-\alpha}$ can also be written in terms of the analytic semigroup $T(t) : \mathbb{R}_{\geq 0} \to L(X)$ whose infinitesimal generator is given by $-\mathcal{L}$. That is using the Laplace transform $(t\mathcal{I}+\mathcal{L})^{-1} = \int_0^\infty \mathrm{e}^{-st}T(s)\mathrm{d}s$, in integral form we have

$$\mathcal{L}^{-\alpha} = \frac{1}{\Gamma(\alpha)}\int_0^\infty t^{\alpha-1}T(t)\mathrm{d}t.$$

Using the properties of the fractional power operator, we can write $\mathcal{L}^{\beta} = \mathcal{L}^{-(1-\beta)}\mathcal{L}$ where $\mathcal{L}^{-(1-\beta)}$ can be written in integral form as

$$\begin{aligned}\mathcal{L}^{-(1-\beta)} &= \frac{\sin(\pi(1-\beta))}{\pi}\int_0^\infty \lambda^{-(1-\beta)}(\lambda\mathcal{I}+\mathcal{L})^{-1}\mathrm{d}\lambda \\ &= \frac{2\sin(\pi(1-\beta))}{\pi}\int_{-\infty}^\infty \mathrm{e}^{2(1-\beta)x}(\mathcal{I}+\mathrm{e}^{2x}\mathcal{L})^{-1}\mathrm{d}x,\end{aligned}$$

where the second equality is obtained from the change of variables $\lambda = \mathrm{e}^{-2x}$. The latter integral is then approximated by numerical quadrature integration and this will be addressed in the following sections.

Finite Element Spatial Discretization

For a mesh $\mathcal{M}$ of the domain D, we define the space $\mathcal{S}_1^0(\mathcal{M})$ to be the space of all continuous piecewise linear functions on D. More formally,

$$\mathcal{S}_1^0(\mathcal{M}) := \{v : D \to \mathbb{R} \text{ such that } v \in C^0(D), \quad v \in \mathcal{P}_1(K)\ \forall K \in \mathcal{T}(\mathcal{M})\},$$

where $\mathcal{T}(\mathcal{M})$ is the set of all elements of the mesh $\mathcal{M}$. We denote by $V_{0,N}$ the finite element space discretized by the means of Lagrangian linear finite elements, such that $V_{0,N} \subset \mathcal{S}_1^0(\mathcal{M})$, where the 0 subscript in $V_{0,N}$ indicates the homogoneous Dirichlet boundary conditions. The corresponding basis of $V_{0,N}$ is $\mathcal{B} = \{\phi_1, \dots, \phi_N\}$ consisting of the hat functions, where $N = \dim(V_{0,N})$. We denote by $\boldsymbol{\mu}(t) \in \mathbb{R}^N$ the $\mathcal{B}$-basis expansion coefficients vector of the finite element solution $v_N(x,t)$ at a given time t. The discrete variational problem then reads

$$\begin{gathered}\forall t \in (0,T], \text{ find } v_N(t,x) \in V_{0,N} \text{ such that} \\ m_N(\dot{v}_N(t,x), \psi_N(x)) + \tilde{a}_N(v_N(t,x), \psi_N(x)) = 0,\ \forall \psi_N \in V_{0,N},\end{gathered} \tag{5}$$

where

$$\begin{aligned}&m_N(u,v) = \langle u, v\rangle_{L^2(D)}, \\ &\tilde{a}_N(u,v) = \langle \tilde{b}(x)\partial_x u, v\rangle_{L^2(D)} + \langle \tilde{c}(x)u, v\rangle_{L^2(D)} + \langle \tilde{a}(x)\mathcal{L}^{-(1-\beta)}\mathcal{L}u, v\rangle_{L^2(D)}.\end{aligned}$$

We introduce the matrices

$$\begin{aligned}&\mathbf{M} = [\langle \phi_j, \phi_i\rangle_{L^2(D)}]_{i,j=1}^N, \\ &\mathbf{M}^{\tilde{c}(x)} = [\langle \tilde{c}(x)\phi_j, \phi_i\rangle_{L^2(D)}]_{i,j=1}^N, \\ &\mathbf{M}^{\tilde{a}(x)} = [\langle \tilde{a}(x)\phi_j, \phi_i\rangle_{L^2(D)}]_{i,j=1}^N, \\ &\mathbf{B}^{\tilde{b}(x)} = [\langle \tilde{b}(x)\partial_x\phi_j, \phi_i\rangle_{L^2(D)}]_{i,j=1}^N,\end{aligned}$$

and $\mathbf{L}$ the FE-approximation of $\mathcal{L}$.

Sinc Quadrature

The integral representation of $\mathcal{L}^{-(1-\beta)}$ is approximated by the sinc quadrature[2]

$$\mathcal{L}^{-(1-\beta)} \approx \frac{2\sin(\pi(1-\beta))}{\pi}k\sum_{l=-K^-}^{K+} \mathrm{e}^{2(1-\beta)lk}(\mathrm{e}^{2lk}\mathcal{L}+\mathcal{I})^{-1}, \tag{6}$$

where k is the step size in the sinc quadrature, calibrated as Bonito and Pasciak[2] suggested for the case of elliptic problems, that is $k = -\log(h)$ where h is the mesh size. We then introduce the matrix $\mathbf{Q}$ as the FE-approximation of (6) given by

$$\mathbf{Q} = \frac{2\sin(\pi(1-\beta))}{\pi} k \sum_{l=-K^-}^{K+} \mathrm{e}^{2(1-\beta)lk} (\mathrm{e}^{2lk}\mathbf{L} + \mathbf{M})^{-1}. \tag{7}$$

In matrix form, the semi-discrete formulation of problem (5) reads

$$\begin{aligned} &\forall t \in (0,T], \text{ find } \boldsymbol{\mu}(t) \in \mathbb{R}^N \text{ such that} \\ &\mathbf{M}\frac{\mathrm{d}\boldsymbol{\mu}}{\mathrm{d}t}(t) + \tilde{\mathbf{A}}\boldsymbol{\mu}(t) = 0, \\ &\boldsymbol{\mu}(0) = \mathcal{P}(g(x)), \end{aligned}$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{M}^{\tilde{a}(x)}\mathbf{QL}$ with $\mathbf{A} := \mathbf{M}^{\tilde{c}(x)} + \mathbf{B}^{\tilde{b}(x)}$, and $\mathcal{P} : \{\mathbb{R}_{>0} \to \mathbb{R}_{>0}\} \to V_{0,N}$ the L^2-projection operator onto the discrete finite element space $V_{0,N}$.

Time Discretization

For the time discretization, we adopt the method of lines with the θ-scheme applied to the following ODE in $\boldsymbol{\mu}$

$$\frac{\mathrm{d}\boldsymbol{\mu}}{\mathrm{d}t}(t) = -\mathbf{M}^{-1}\tilde{\mathbf{A}}\boldsymbol{\mu}(t). \tag{8}$$

To this extent, we define the discrete evolution operator $\Psi_\theta^\tau : \mathbb{R}^N \to \mathbb{R}^N$ such that $\Psi_\theta^\tau \boldsymbol{\mu}(t) = \boldsymbol{\mu}(t+\tau)$. For the $\theta-$scheme with time step τ, the discrete evolution operator applied to the ODE $\mathbf{y}_t(t) = \mathbf{f}(\mathbf{y}(t))$ is given by

$$\Psi_\theta^\tau \mathbf{y}(t) = \mathbf{y}(\mathbf{t}) + \tau[\theta\mathbf{f}(\mathbf{y}(\mathbf{t}+\tau)) + (1-\theta)\mathbf{f}(\mathbf{y}(\mathbf{t}+\tau))]. \tag{9}$$

Applying scheme (9) to equation (8), we obtain the fully discrete model

$$[\mathbf{M} + \tau\theta\tilde{\mathbf{A}}]\boldsymbol{\mu}^m = [\mathbf{M} - \tau(1-\theta)\tilde{\mathbf{A}}]\boldsymbol{\mu}^{m-1}, \ \forall m = 1, \ldots, M. \tag{10}$$

Iterative Methods for Solving Linear Systems

Solving the linear system in equation (7) was firstly done using a direct LU-factorization applied to the system

$$(\mathrm{e}^{2lk}\mathbf{L} + \mathbf{M})\mathbf{x} = \mathbf{b}. \tag{11}$$

Two iterative methods were then implemented to solve system (11), namely the conjugate gradient descent(CGD) and the preconditioned conjugate gradient descent(PCGD). For the PCGD, we apply diagonal preconditioning, that is we apply the CGD algorithm to the system

$$\mathbf{D}^{-1}(\mathrm{e}^{2lk}\mathbf{L} + \mathbf{M})\mathbf{x} = \mathbf{D}^{-1}\mathbf{b}, \quad \mathbf{D} = \mathrm{diag}(\mathrm{e}^{2lk}\mathbf{L} + \mathbf{M}).$$

On the other side, for the time stepping in equation (10), inverting the matrix $[\mathbf{M} + \tau\theta\tilde{\mathbf{A}}]$ in equation (10) requires the inversion the matrix $\mathbf{Q}$. However, and since $\mathbf{Q}$ is not known explicitly and can only be accessed by matrix vector multiplication, solving the linear system (10) requires an iterative solver. To that extent, we apply the CGD algorithm to the system (10) at each time step, with an appropriate tolerance value as a stopping criteria for the algorithm.

C++ Object-Oriented Implementation

The C++ object-oriented implementation of the finite element method is based on the LehrFEM++ library for the 2D problem, and relies on a self-developed *Eigen*-based code for the 1D case. In this section, we discuss the most important aspects of the implementation, such as the Galerkin matrices assembly, treatment of the Dirichlet boundary conditions, sinc quadrature and the time stepping.

Galerkin Matrices Assembly

The Galerkin matrices assembly is implemented in *space_discr.h*. For the 1D problem, the assembly is done directly inside the function in (1), returning a vector of triplets for the non-zero entries. An example is the following assembly of the mass matrix $\mathbf{M}^{\gamma(x)}$ with a general coefficient $\gamma(x)$, using the local trapezoidal rule.

```cpp
using TripletForm = std::vector<Triplet>;
template <typename FUNCTOR>
TripletForm getMassMatrixWeighted_TripletForm(const Eigen::VectorXd &
    mesh, FUNCTOR gamma) {
  TripletForm triplets;
  unsigned M = mesh.size() - 1;
  triplets.reserve(M + 1);

  /* Diagonal Entries*/
  double diagEntry = 0.5 * gamma(mesh(0)) * (mesh(1)-mesh(0));
  triplets.push_back(Triplet(0,0,diagEntry));

  diagEntry = 0.5 * gamma(mesh(M)) * (mesh(M) - mesh(M-1));
  triplets.push_back(Triplet(M,M,diagEntry));

  for (int i=1; i<M; i++){
    double dx_right = mesh(i+1) - mesh(i);
    double dx_left = mesh(i) - mesh(i-1);
    diagEntry = 0.5 * gamma(mesh(i)) * (dx_right+dx_left);
    triplets.push_back(Triplet(i,i,diagEntry));
  }
  return triplets;
}
```

Code 1: 1D assembly of the mass matrix with general coefficient γ.

However, transforming the matrix of type `std::vector<Triplet>` to an actual sparse matrix of type `Eigen::SparseMatrix<double>` is mandatory and is done in the main program, by calling the function `setFromTriplet`.

In contrast, the assembly of the Galerkin matrices for the 2D problem relies on a distributing scheme. That is the element matrix is computed for each element of the mesh, and its entries are distributed to the full Galerkin matrix entries by the so called a local-global mapping. Motivated by this distributing scheme, the code snippet (2) provides a base class for an `ElementMatrixProvider` object for the element matrix calculation. Note that in a code, one inherits from that base class a derived class where the coefficient function `MAP coeff_mapping_` in the corresponding bilinear form is passed to the constructor.

```cpp
  typedef Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>
    elemMatrix;

  /**
```

```
 * @base_class ElementMatrixProvider
 *
 * @brief Computing the element matrix corresponding to a cell
 *
 * @tparam MAP the type of the corresponding coefficient functor
 *
 */
template<class MAP>
class ElementMatrixProvider{
  public:
    ElementMatrixProvider() {}
    bool isActive(const lf::mesh::Entity& ){ return true; }
    virtual const elemMatrix Eval(const lf::mesh::Entity&) = 0;
    virtual ~ElementMatrixProvider(){}
  protected:
    lf::quad::QuadRule myQuadRule_Tria_;
    lf::quad::QuadRule myQuadRule_Quad_;
    MAP coeff_mapping_;
    std::array<lf::uscalfe::PrecomputedScalarReferenceFiniteElement<
 double>, 5> fe_precomp_;
};
```

Code 2: Element matrix provider base class.

The element matrix provider class stores 4 important member variables that are `myQuadRule_Tria_`, `myQuadRule_Quad_`, `coeff_mapping_` and `fe_precomp_`. The first two defines the quadrature rules to be used for the local integrations, for both triangular and quadrilateral elements. These definitions are done in the body of the constructor where the quadrature nodes and weights for each reference element are implemented. The object `coeff_mapping_` is the corresponding coefficient of the bilinear form. On the other side, the object `fe_precom_p_` stores precomputed values of the reference shape functions and their gradients, at the quadrature nodes on the reference elements.
The main method `virtual const elemMatrix Eval(const lf::mesh::Entity&)` takes as input a `const lf::mesh::Entity&` cell type and returns the element matrix corresponding to that cell using parametric finite elements. For example the element cross matrix with a general vector valued coefficient $\beta(x)$ is given by

$$\begin{aligned}
\Big[\int_K \beta(x)^\top \nabla\phi_i(x)\phi_j(x)\Big]_{i,j=1}^m \mathrm{d}x &= \Big[\int_{\hat{K}} \beta(\Phi_K(\hat{x}))^\top \nabla\phi_i(\Phi_K(\hat{x}))\phi_j(\Phi_K(\hat{x}))|D\Phi_K(\hat{x})|\mathrm{d}\hat{x}\Big]_{i,j=1}^m \\
&= \Big[\int_{\hat{K}} \beta(\Phi_K(\hat{x}))^\top D\Phi_K^{-\top}(\hat{x})\nabla\hat{\phi}_i(\hat{x})\hat{\phi}_j(\hat{x})|D\Phi_K(\hat{x})|\mathrm{d}\hat{x}\Big]_{i,j=1}^m \\
&= \Big[\sum_{l=1}^q \omega_l \beta(\Phi_K(\hat{\zeta}_l))^\top D\Phi_K^{-\top}(\hat{\zeta}_l)\nabla\hat{\phi}_i(\hat{\zeta}_l)\hat{\phi}_j(\hat{\zeta}_l)|D\Phi_K(\hat{\zeta}_l)|\Big]_{i,j=1}^m \\
&= \sum_{l=1}^q \omega_l \beta(\Phi_K(\hat{\zeta}_l))^\top |D\Phi_K(\hat{\zeta}_l)| \Big[D\Phi_K^{-\top}(\hat{\zeta}_l)\nabla\hat{\phi}_i(\hat{\zeta}_l)\hat{\phi}_j(\hat{\zeta}_l)\Big]_{i,j=1}^m,
\end{aligned}$$

where $m \in \{3,4\}$, K is an element, $\hat{K}$ the corresponding reference element, $\Phi_K : \hat{K} \to K$ linear in the case of triangles and bilinear in the case of quadrilaterals, $(\omega_l)_{l=1}^q$ are the quadrature weights and $(\zeta_l)_{l=1}^q$ are the quadrature nodes on $\hat{K}$. The element mass matrix with a scalar valued coefficient $\gamma(x)$ is given by

$$\begin{aligned}\left[\int_K \gamma(x)\phi_i(x)\phi_j(x)\right]_{i,j=1}^m \mathrm{d}x &= \left[\int_{\hat{K}} \gamma(\Phi_K(\hat{x}))\phi_i(\Phi_K(\hat{x}))\phi_j(\Phi_K(\hat{x}))|D\Phi_K(\hat{x})|\mathrm{d}\hat{x}\right]_{i,j=1}^m \\ &= \left[\int_{\hat{K}} \gamma(\Phi_K(\hat{x}))\hat{\phi}_i(\hat{x})\hat{\phi}_j(\hat{x})|D\Phi_K(\hat{x})|\mathrm{d}\hat{x}\right]_{i,j=1}^m \\ &= \left[\sum_{l=1}^q \omega_l\gamma(\Phi_K(\hat{\zeta}_l))\hat{\phi}_i(\hat{\zeta}_l)\hat{\phi}_j(\hat{\zeta}_l)|D\Phi_K(\hat{\zeta}_l)|\right]_{i,j=1}^m \\ &= \sum_{l=1}^q \omega_l\gamma(\Phi_K(\hat{\zeta}_l))|D\Phi_K(\hat{\zeta}_l)|\left[\hat{\phi}_i(\hat{\zeta}_l)\hat{\phi}_j(\hat{\zeta}_l)\right]_{i,j=1}^m,\end{aligned}$$

and the stiffness matrix with a matrix valued coefficient $\alpha(x)$ is given by

$$\begin{aligned}\left[\int_K \alpha(x)\nabla\phi_i(x)\nabla\phi_j(x)\right]_{i,j=1}^m \mathrm{d}x &= \left[\int_{\hat{K}} \alpha(\Phi_K(\hat{x}))\nabla\phi_i(\Phi_K(\hat{x}))\phi_j(\Phi_K(\hat{x}))|D\Phi_K(\hat{x})|\mathrm{d}\hat{x}\right]_{i,j=1}^m \\ &= \left[\int_{\hat{K}} \alpha(\Phi_K(\hat{x}))D\Phi_K^{-\top}(\hat{x})\nabla\hat{\phi}_i(\hat{x})D\Phi_K^{-\top}(\hat{x})\nabla\hat{\phi}_j(\hat{x})|D\Phi_K(\hat{x})|\mathrm{d}\hat{x}\right]_{i,j=1}^m \\ &= \left[\sum_{l=1}^q \omega_l\alpha(\Phi_K(\hat{\zeta}_l))D\Phi_K^{-\top}(\hat{\zeta}_l)\nabla\hat{\phi}_i(\hat{\zeta}_l)D\Phi_K^{-\top}(\hat{x})\nabla\hat{\phi}_j(\hat{\zeta}_l)|D\Phi_K(\hat{\zeta}_l)|\right]_{i,j=1}^m \\ &= \sum_{l=1}^q \omega_l\alpha(\Phi_K(\hat{\zeta}_l))|D\Phi_K(\hat{\zeta}_l)|\left[D\Phi_K^{-\top}(\hat{\zeta}_l)\nabla\hat{\phi}_i(\hat{\zeta}_l)D\Phi_K^{-\top}(\hat{x})\hat{\phi}_j(\hat{\zeta}_l)\right]_{i,j=1}^m.\end{aligned}$$

Finally, the assembly is done in the main program using the function `lf::assemble::AssembleMatrixLocally` that implements the previously discussed distributing scheme.

Sinc Quadrature

Code 3 defined in *space_discr.h* implements a `SincMatrix` class that represents the $\mathbf{Q}$ matrix object (7), where the matrix-vector $*$ operator is overloaded to implement equation (7). Note that $\mathbf{Q}$ is not stored explicitly, and hence is accessed by matrix-vector operations. The operator $*$ takes a vector $\mathbf{f} \in \mathbb{R}^N$ and returns $\mathbf{Qf}$ as seen in line (15) of code (3).

```
1  /**
2   * @class SincMatrix
3   *
4   * @brief Representing the sinc matrix Q, without storing its elements
      explicitly.
5   *        Accessing the matrix Q through matrix-vector product by
      overloading the *            operator.
6   *        Given f \in \mathbb{R}^n, the * method returns Q*f.
7   *
8   */
9  class SincMatrix{
10   public:
11     template<typename T>
12     SincMatrix(T&&); /// Move and copy constructors
13     template<typename T>
14     SincMatrix(double, T&& L, T&& M);
15     Eigen::VectorXd operator* (Eigen::VectorXd& f) const;
16   private:
17     double k_;
18     SparseMatrix L_, M_;
19     template<typename SincMatType, bool is_theta_zero>
20     friend class thetaSchemeTimeStepper;
```

```cpp
};
```

Code 3: **Q** matrix class.

For solving the linear system in (7), three methods were implemented and the computational time for each was then reported. More precisely, the first method was to apply the LU-factorization and to solve the system using the sparse-LU solver of type `Eigen::SparseLU<Eigen::SparseMatrix<double>>`. For the iterative conjugate gradient descent, we make use of the solver of type `Eigen::ConjugateGradient< SparseMatrix<double>, Lower|Upper, _Preconditioner>` where the `_Preconditioner` template parameter is set to `Eigen::IdentityPreconditioner` for the CGD and to `Eigen::DiagonalPreconditioner` for the PCGD. For the CGD, we applied the algorithm to solve

$$(\mathrm{e}^{2lk}\mathbf{L} + \mathbf{M})\mathbf{x} = \mathbf{b},$$

as for the PCGD, we applied diagonal preconditioning, that is we applied the CGD algorithm to the system

$$\mathbf{D}^{-1}(\mathrm{e}^{2lk}\mathbf{L} + \mathbf{M})\mathbf{x} = \mathbf{D}^{-1}\mathbf{b},\ \ \mathbf{D} = \mathrm{diag}(\mathrm{e}^{2lk}\mathbf{L} + \mathbf{M}).$$

Code (4) shows the implementation of the member method in line (15) of code (3). Avoiding the explicit for loop, we make use of the standard library function `std::for_each()` to perform the sum in equation (7).

```cpp
/**
 * @member operator*
 *
 * @brief Performing the matrix-vector product Q*f
 *
 * @param f the vector f
 * @return the product Q*f
 *
 */
Eigen::VectorXd SincMatrix::operator* (Eigen::VectorXd& f) const{
  if (M_.cols() != f.size())
    throw std::length_error("Matrix-vector sizes do not match!");

  Eigen::VectorXd res = Eigen::VectorXd::Zero(f.size());

  double Cb = M_PI/(2*std::sin(M_PI*beta));

  Eigen::ConjugateGradient<Eigen::SparseMatrix<double>, Eigen::Lower|
   Eigen::Upper,
      Eigen::IdentityPreconditioner> PCGD_Solver;
  PCGD_Solver.setMaxIterations(1000);
  PCGD_Solver.setTolerance(1e-10);

  Eigen::SparseLU<Eigen::SparseMatrix<double>> SparseLU_Solver;

  std::for_each(K_.data(), K_.data()+K_.size(), [&](double arg){
       PCGD_Solver.compute(M_+std::exp(2*arg*k_)*L_);
       res = res + (k_/Cb)*std::exp(2*beta*arg*k_)*PCGD_Solver.solve(f)
   ; });

  return res;
}
```

Code 4: **Q** matrix-vector multiplication.

Dirichlet Boundary Conditions

In the 1D problem, the Galerkin matrices are triadiognal, and the ordering of the nodes makes it easy to impose the 0 Dirichlet boundary conditions on the Galerkin matrices, by changing the entries corresponding to the first and the last nodes. That is the corresponding rows and columns entries are set to 0, except the diagonal entry is set to 1, according to the following transformation

$$\begin{pmatrix} a_1 & b_1 & & & & \\ c_1 & a_2 & b_2 & & & \\ & c_2 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{N-2} & a_{N-1} & b_{N-1} \\ & & & & c_{N-1} & a_N \end{pmatrix} \Longrightarrow \begin{pmatrix} 1 & 0 & & & & \\ 0 & a_2 & b_2 & & & \\ & c_2 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{N-2} & a_{N-1} & 0 \\ & & & & 0 & 1 \end{pmatrix}$$

In fact, that is what the function `imposeZeroDirichletBoundaryConditions`, implemented in *space_discr.h* does.

```cpp
/* Enforce dirichlet BC, 0 on the boundaries of the domain */
  lf::mesh::utils::CodimMeshDataSet<bool>
        bd_flags = lf::mesh::utils::flagEntitiesOnBoundary(mesh_p,2);
  std::function<bool(unsigned int dof_idx)> bd_selector_d=
      [&] (unsigned int dof_idx){
      return bd_flags(dofh.Entity(dof_idx));
  };
```

Code 5: Boundary edge selector.

However in the 2D case, the nodes are not ordered as in the 1D case and can be indexed arbitrarly. That's why we need a function that tells us whether a given node is a boundary node. That is what the function `bd_selector_d` in code (5) does. In fact, the latter function takes an index and returns a boolean indicating whether a node associated with that index is a boundary node. Finally, the boundary selector is passed to the function `void dropMatrixRowsColumns(SELECTOR &&selectvals, lf::assemble::COOMatrix<SCALAR> &A)`, along with the Galerkin matrix in triplet format, and performs the previously discussed transformation of the corresponding rows and columns.

Time Stepping

The code snippet 6 defined in *time_stepper.h* provides the θ-scheme time stepper class. This class stores the Galerkin matrices as private members, and that are required for the calculations of the solution at each time step.

```cpp
/**
 * @class thetaSchemeTimeStepper
 *
 * @brief Performing the \theta scheme time stepping.
 *
 * @tparam SincMatType the type of sinc matrix
 * @tparam is_beta_zero a boolean to distinguish between
 *          fractional and non-fractional cases
 *
 */
template<typename SincMatType, bool is_beta_zero>
class thetaSchemeTimeStepper{
  public:
    thetaSchemeTimeStepper() = delete;

    /// Move and copy constructors
    thetaSchemeTimeStepper(SincMatType&& Q, SparseMatrix&& A,
  double dt, double h, double N_dofs);
    thetaSchemeTimeStepper(const SincMatType& Q, const SparseMatrix& A,
  double dt, double h, double N_dofs);

    /// \Psi^{\tau} (u(t)) = u(t+\tau)
    template<bool it_is = is_beta_zero>
      typename std::enable_if<it_is,Eigen::VectorXd>::type
      discreteEvolutionOperator(Eigen::VectorXd&) const;
    template<bool it_is = is_beta_zero>
      typename std::enable_if<!it_is,Eigen::VectorXd>::type
      discreteEvolutionOperator(Eigen::VectorXd&) const;

    virtual ~thetaSchemeTimeStepper() = default;
  private:
    double dt_, h_; int N_dofs_;
    SparseMatrix A_; /// The stiffness matrix w/o MQL
    SparseMatrix L_; /// FEM approx. of \mathcal{L}
    SparseMatrix M_; /// The mass matrix
    SincMatType Q_; /// FEM approx. of \mathcal{L}^{-beta = -(1-\beta)}
    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_; /// for non-
   fractional case
};
```

Code 6: Time stepper class.

The main member method `Eigen::VectorXd discreteEvolutionOperator_cgd(Eigen::VectorXd&) const;` takes the solution vector $\boldsymbol{\mu}$ at time t and returns it at time $t+\tau$ according to equation (9). Note that this method is overloaded, and its implementation depends on the value of β. If $\beta \neq 1$, the method implements the CGD algorithm applied to equation (10). Otherwise, the system in (10) is solved using a sparse-LU decomposition.
The following code (7) provides the conjugate gradient descent algorithm implementation.

```
/**
 * @member discreteEvolutionOperator
 *
 * @overload
 *
 * Conjugate gradient descent is implemented for fractional case
 *
 * @throws std::overflow_error if residuals in the CGD diverge
 *
 */
template<typename SincMatType, bool is_beta_zero>
template<bool it_is> typename std::enable_if<!it_is,Eigen::VectorXd>::
   type
thetaSchemeTimeStepper<SincMatType,
        is_beta_zero>::discreteEvolutionOperator(
      Eigen::VectorXd& u_prev) const{
  /// Conjugate gradient descent algorithm for the non-fractional case
  size_t n = 0;
  Eigen::VectorXd u_next = Eigen::VectorXd::Zero(u_prev.size());
  Eigen::VectorXd tmp = L_ * u_prev;
  Eigen::VectorXd rhs_vec = M_*u_prev -
        dt_*(1-theta) * (A_*u_prev + M_*(Q_*tmp) );
  Eigen::VectorXd tmpp = L_ * u_next;
  Eigen::VectorXd old_resid = rhs_vec - (M_*u_next + theta*dt_*
            (A_*u_next+M_*(Q_*tmpp) ) );
  if (old_resid.norm() < tol)
    return u_next;
  else{
    Eigen::VectorXd p = old_resid;
    Eigen::VectorXd new_resid = old_resid;
    while (new_resid.norm() > tol){
      if (new_resid.norm() > 1e3)
  throw std::overflow_error("The residual is diverging in CGD!");
      if (n > maxIterations)
  throw std::overflow_error("Maximum CGD iterations reached!");
      old_resid.swap(new_resid);
      tmp = L_ * p;
      Eigen::VectorXd Stiff_times_p = M_*p +
    theta * dt_ * (A_*p + M_*(Q_*tmp));
      double alpha = old_resid.dot(old_resid) /
      old_resid.dot(Stiff_times_p);
      u_next += alpha * p;
      new_resid = old_resid - alpha * Stiff_times_p;
      double beta_tmp = new_resid.dot(new_resid) / old_resid.dot(
   old_resid);
      p = new_resid + beta_tmp * p;
      n++;
    }
    return u_next;
  }
}
```

Code 7: The conjugate gradient descent.

Numerical Experiments

In this section, we present convergence analysis results of the Finite Element Method applied to the one-dimensional heat equation. We then compare the computational time of the direct LU-factorization and the iterative CGD and PCGD methods. Afterwards, we present some results of our numerical methods applied to the two-dimensional heat equation. Finally, we present a finance application, namely European options pricing results.

The 1D Fractional Heat Equation - Convergence Results

Consider the 1D fractional heat equation on the unit interval D=(0, 1),

$$\begin{aligned} \partial_t u(t,x) + (-\partial_{xx})^\beta u(t,x) &= 0, \quad (t,x) \in (0,T] \times D, \\ u(0,x) &= x(1-x), \quad x \in D, \\ u(t,0) = u(t,1) &= 0, \quad t \in [0,T]. \end{aligned} \tag{12}$$

For $T = 1$, we apply the finite element method to (12) with different combinations of the parameters β and θ. Namely we let $\beta \in \{1/4, 1/2, 3/4, 1\}$, and $\theta \in \{0, 1/4, 1/2, 3/4, 1\}$ for the θ-scheme time stepping. The discrete equation (10) will then satisfy $\mathbf{A} = 0, \tilde{a} \equiv 1$ and $\tilde{\mathbf{A}} = \mathbf{MQL}$.

For the error analysis, in (8) we implement a C++ class `meshUniformRefinement` that describes the mesh hierarchy used to conduct the convergence analysis.

```
class meshUniformRefinement{
  public:
    meshUniformRefinement(size_t N_0, size_t numOfLevels){
    h_.resize(numOfLevels);
  N_dofs_.resize(numOfLevels);
  meshes_.resize(numOfLevels);
  std::vector<double> L(numOfLevels);
  std::iota(L.begin(), L.end(), 0); /// Refinement levels
  /// mesh sizes: 1/N0 * 2^{-L}
  std::transform(L.begin(), L.end(), h_.begin(), [&](double level){
      return 1.0/N_0 * std::pow(2,-level);} );
        /// Total number of nodes: 1+N0*2^{L}
  std::transform(L.begin(), L.end(), N_dofs_.begin(), [&](double level)
   {
      return 1 + N_0 * std::pow(2,level);} );
  std::transform(N_dofs_.begin(), N_dofs_.end(), meshes_.begin(), [&](
   size_t Ndofs){
          return Eigen::VectorXd::LinSpaced(Ndofs, 0, 1); } );
      }

      /// Returns a pointer to the mesh at a given level
      std::shared_ptr<Eigen::VectorXd> getMesh(size_t level){
    return std::make_shared<Eigen::VectorXd>(meshes_.at(level));
  }
    std::vector<double> h_;
    std::vector<size_t> N_dofs_;

  private:
    std::vector<Eigen::VectorXd> meshes_;
};
```

Code 8: 1D uniform refinement.

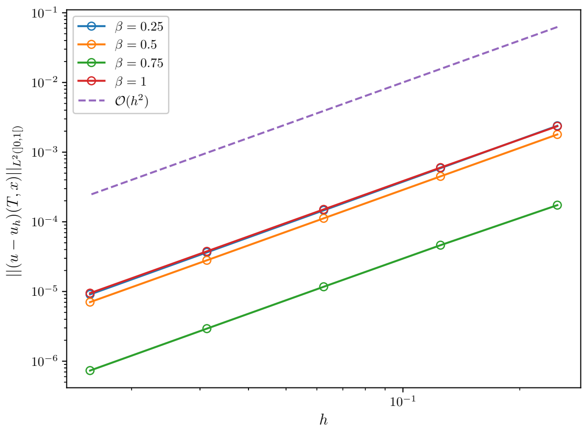
More precisely, given `N_0` and `numOfLevels` which defines the number of elements on the coarsest mesh and the number of levels in the mesh hierarchy, respectively, the constructor `meshUniformRefinement (size_t N_0, size_t numOfLevels)` constucts the meshes at all levels by uniform refinement, where the number of nodes at each level is given by $1 + N_0 2^L$, and the mesh size is given by $1/N_0 2^{-L}$ where $L \in \{0, \ldots, \texttt{numOfLevels}\}$ is the corresponding mesh level. In fact, the mesh size and the number of nodes at a certain level can be accessed through the member variables `h_` and `N_dofs_`. Also, the meshes at all the levels are stored in the private member variable `meshes_`. Finally, the member method `getMesh` takes a level and returns a pointer to the mesh corresponding to that level.
For the 1D fractional heat equation (12), we set $N_0 = 4$ and we use 6 levels. The quantities of interest studied are the L_2-norm and the L_∞-norm of the discretization error. More precisely, we are interested in the convergence of $||(u - u_h)(T, \cdot)||_{L^2(D)}$ and $||(u - u_h)(T, \cdot)||_{L^\infty(D)}$. Note that the L^2-norm of the discretization error at the final time is given by

$$\sqrt{\int_0^1 |u - u_h|^2(T, x)\mathrm{d}x} \approx \sqrt{\sum_{j=1}^{N} \frac{|u - u_h|^2(T, x_{j-1}) + |u - u_h|^2(T, x_j)}{2} \Delta x_j},$$

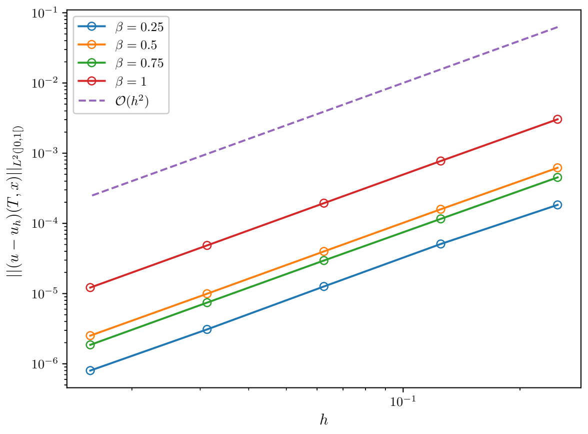
where we have used the trapezoidal rule. In addition to that, the exact solution to (12) is given by

$$u(t, x) = \sum_{j=1}^{\infty} \Big[2\mathrm{e}^{-(j\pi)^\beta t} \int_0^1 u(0, y) \sin(j\pi y)\mathrm{d}y \, \sin(j\pi x) \Big].$$

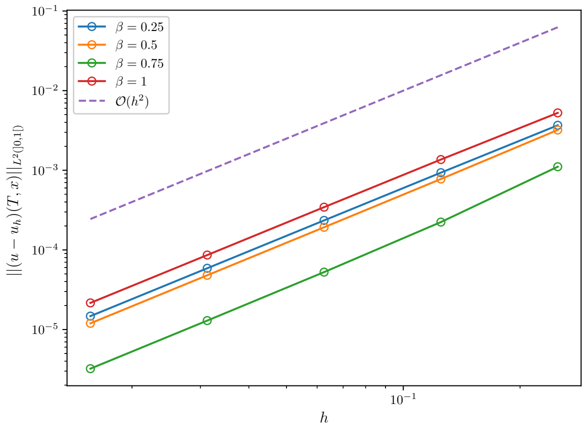
In order to calculate the reference solution u, the series is truncated to a finite sum.
We start by presenting the convergence results for the well known numerical integration schemes such as the Forward Euler, Implicit Euler and the Crank–Nicolson schemes. The case of $\beta = 1$ corresponds to the non-fractional heat equation.



(a) $\theta = 0$, Explicit Euler.



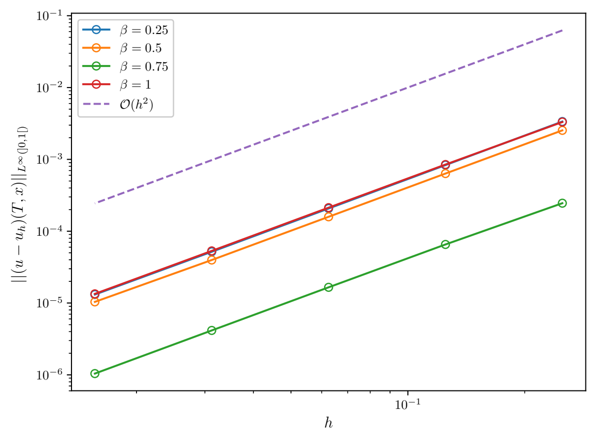
(b) $\theta = 1/2$, Crank–Nicolson.



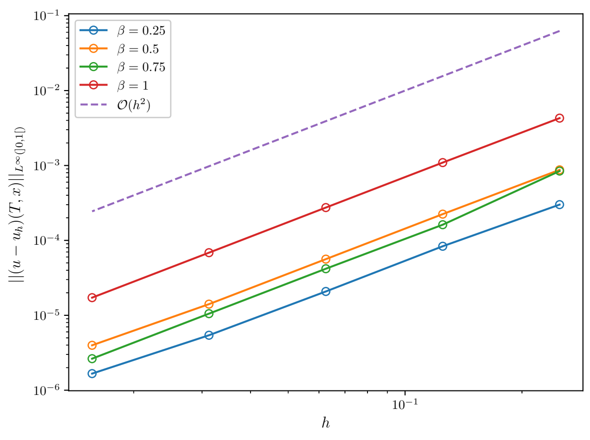
(c) $\theta = 1$, Implicit Euler.

Figure 1: The L^2-norm of the discretization error versus the mesh size h for the Explicit Euler, Implicit Euler and the Crank–Nicolson schemes.

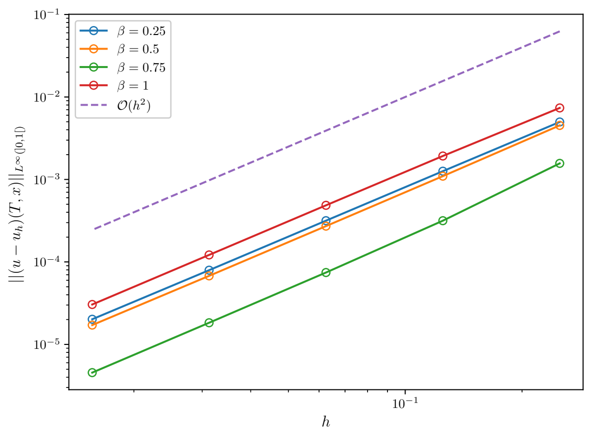
For linear finite elements, one can prove a second order convergence of the L^2 discretization error[4], in the non-fractional case. Indeed, the above plots exhibit quadratic L^2 convergence for all choices of β, in particular also for the non-fractional $\beta \in (0, 1)$. Moreover, we observe $\mathcal{O}(h^2)$ convergence of the discretization error in the L^∞-norm. Discretization error convergence results for other values of the parameter θ can be found in the appendix.
As a side note, in each iteration solving a linear system of equations is required and that provides a computational challenge because it makes the code slower. One possible solution is to parallelize the method `std::for_each()` over several cores, where each core performs a summation of (7) over a smaller number of steps, and that will speed-up the program.



(a) $\theta = 0$, Explicit Euler.



(b) $\theta = 1/2$, Crank–Nicolson.



(c) $\theta = 1$, Implicit Euler.

Figure 2: The L^∞-norm of the discretization error versus the mesh size h for the Explicit Euler, Implicit Euler and the Crank–Nicolson schemes.

The 2D Fractional Heat Equation

Consider the 2D fractional heat equation on the unit square $D = (0,1) \times (0,1)$,

$$\begin{aligned} \partial_t v(t,\mathbf{x}) + (-\Delta)^{1/2} v(t,\mathbf{x}) &= 0, \quad && (t,\mathbf{x}) \in (0,T] \times D, \\ v(0,\mathbf{x}) &= \mathbf{x}_1(1-\mathbf{x}_1)\mathbf{x}_2(1-\mathbf{x}_2), \quad && \mathbf{x} \in D, \\ v(t,\mathbf{x}) &= 0, \quad && (t,\mathbf{x}) \in (0,T] \times \partial D. \end{aligned} \tag{13}$$

For $T = 0.1$, figure (3) shows the solution $v(T,\mathbf{x})$.

An Application to Option Pricing

We present numerical results corresponding to the pricing of European call and put options with continuous payoff funtions, and digital options with discontinuous payoffs. In the following, we denote by $(S_t)_{t\geq 0}$ the price process of the underlying asset, K the strike price and T the option maturity. The payoff of the option is denoted by $g(s)$, where it is equal to $(s-K)^+$ for call options, $(K-s)^+$ for put options and $\mathbb{1}_{\{s>K\}}$ for digital options. For $\beta = 1/2$ (i.e, $\alpha = 1$), the price process follows

$$\mathrm{d}S_t = rS_{t^-}\mathrm{d}t + \sigma\sqrt{S_{t^-}}\mathrm{d}L_t^1, \quad S_0 = s_0, \tag{14}$$

where $(L_t^1)_{t\geq 0}$ is an 1-stable Lévy process, $r > 0$ the risk-free interest rate and $\sigma > 0$ the volatility. In this case, existence and uniqueness of solutions to (14) follow immediately from the condi-

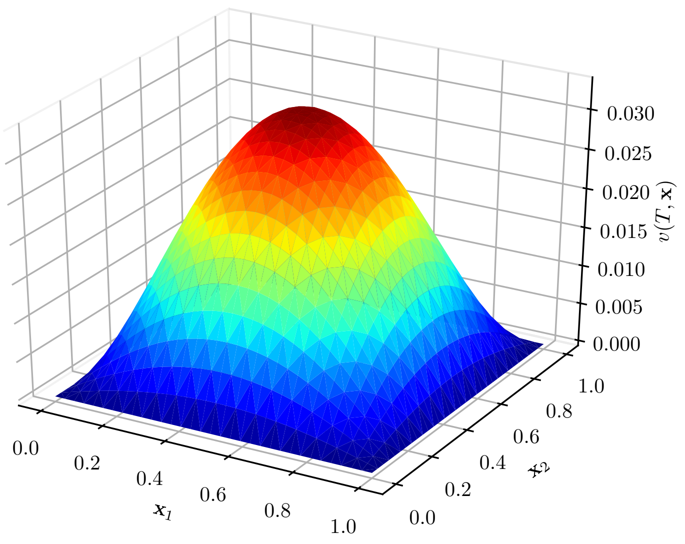


Figure 3: The solution of the 2D fractional heat equation.

tions discussed previously, with $\kappa(x) = rx$ and $\rho(x) = \sigma\sqrt{x}$. Indeed, $\kappa(x)$ is increasing and concave, $\kappa(0) = 0, \int_0^\infty rx^{-1}\mathrm{d}x = \infty$ and $|rx - ry| = r|x-y| \leq \kappa(|x-y|)\ \forall x, y \in \mathbb{R}$. In addition to that, we have that $\rho(x)$ is increasing and concave, $\rho(0) = 0$ and $\int_0^\infty(\sigma\sqrt{x})^{-1}\mathrm{d}x = \infty$. Finally, $|\sigma\sqrt{x} - \sigma\sqrt{y}| = \sigma|\sqrt{x} - \sqrt{y}| \leq \sigma\sqrt{|x-y|} = \rho(|x-y|)\ \forall x, y \in \mathbb{R}$. Indeed and assuming wlog. that $x \geq y > 0$, squaring both sides of the latter inequality leads to

$$x - 2\sqrt{xy} + y \leq x - y \implies 2\sqrt{xy} \geq 2y,$$

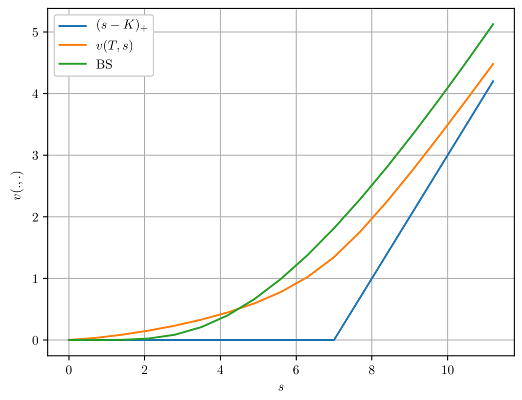
which follows from the assumption that $x \geq y$. The infinitesimal generator $\mathcal{A}^1$ of the process S is

$$\mathcal{A}^1 = rs\partial_s - \sigma\sqrt{s}(-\partial_{ss})^{1/2}.$$

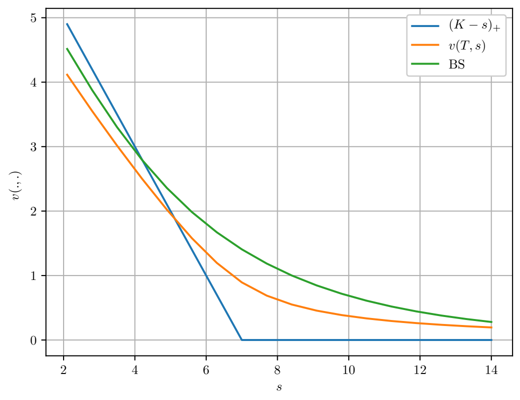
The pricing parabolic PDE is then given by

$$\begin{aligned} v_t(t,s) + \overbrace{\sigma\sqrt{s}(-\partial_{ss})^{1/2}v(t,s) - rs\partial_s v(t,s)}^{-\mathcal{A}^1 v(t,s)} + rv(t,s) &= 0, && (t,s) \in (0,T] \times (0,R), \\ v(0,s) &= g(s), && s \in [0,R], \\ v_{\text{Call}}(t,0) = 0,\ v_{\text{Digital}}(t,0) = 0,\ v_{\text{Put}}(t,R) &= 0, && t \in (0,T]. \end{aligned} \tag{15}$$

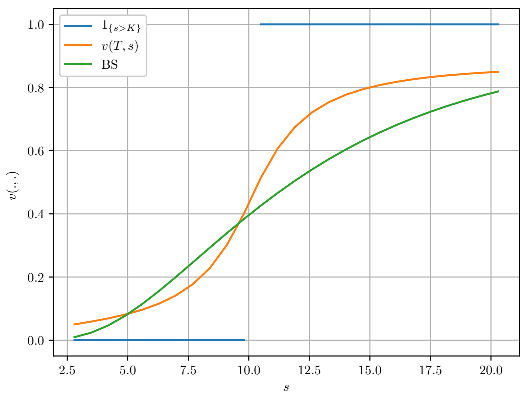
Call and digital options are approximated by down-and-out barrier options with barrier $H = 0$, so that we impose homogeneous Dirichlet boundary conditions at $s = 0$. On the other side, put options are approximated by up-and-out barrier options with barrier $H = R$, and so we impose homogeneous Dirichlet boundary conditions at $s = R$. In that case, the Poincaré inequality still holds since each of {R} and {0} has positive counting measure, and hence the whole theory of existence and uniqueness of weak solutions holds.
In the following, we fix $r = 0.06, \sigma = 0.6$ and $T = 1$.



(a) $K = 7$, European call option.



(b) $K = 7$, European put option.



(c) $K = 10$, European digital option.

Figure 4: The price of European options under Feller–Lévy models (orange), together with the payoff (blue). For comparison: the corresponding price values in the Black-Scholes market (green).

Comparison of Direct and Iterative Solvers

In this section we present a comparison in terms of computational time of the direct LU-factorization and the two iterative solvers, CGD and PCGD, all applied in case of the fractional heat equation (12) and the pricing problem (15), for different number of mesh nodes N. Table (1) summarizes the results, where the time is in seconds (s). For the fractional heat equation, the CGD always outperforms the

N	50	70	100	150
SparseLU	43.1	125.8	387.9	>1000
CGD	10.2	34.3	134.1	675
PGCD	10.9	36.1	143.5	700.5

(a) The fractional heat equation.

N	50	70	80	100
SparseLU	73.4	186.06	288.7	602.2
CGD	32.2	106.2	180.2	466.5
PGCD	28.01	87.53	147.7	365.2

(b) The pricing problem.

Table 1: Computational time comparison for the LU-factorization, CGD and PCGD.

direct LU solver, however the PCGD performs slightly worse than the CGD, but better than the direct LU solver. As for the pricing problem, the CGD also outperforms the direct LU solver and the same goes for the PCGD, but we notice that the preconditioning improves the computational time as opposed to the case of the fractional heat equation. This can be given an intuitive explanation using the following plot.

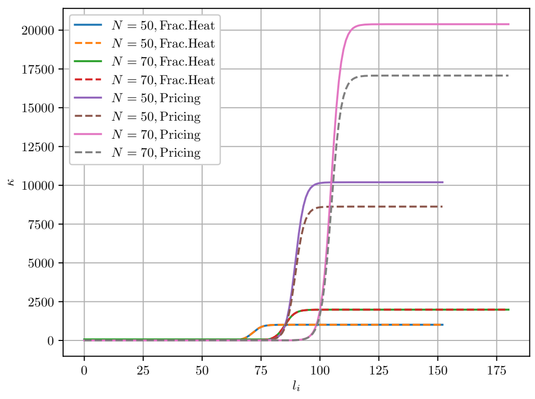


Figure 5: Evolution of the condition number κ for the case of the fractional heat equation and the pricing problem for N=50 and N=70. The solid lines correspond to the CGD and the dashed lines correspond to the PCGD.

The condition number of a matrix $\mathbf{X}$, $\kappa(\mathbf{X})$ is defined by

$$\kappa(\mathbf{X}) = \frac{\sigma_{\max}}{\sigma_{\min}},$$

where σ is a singular value of the matrix $\mathbf{X}$. The solid lines in the plot represent the condition number $\kappa(\mathrm{e}^{2lk}\mathbf{L}+\mathbf{M})$ for all l in the sum in equation (7), where the iterative solver is applied. On the other side, the dashed lines correspond to the condition number after diagonal preconditioning, that is $\kappa(\mathbf{D}^{-1}(\mathrm{e}^{2lk}\mathbf{L}+\mathbf{M}))$ where $\mathbf{D} = \mathrm{diag}(\mathrm{e}^{2lk}\mathbf{L}+\mathbf{M})$. The following code shows how to calculate the condition number of a matrix in C++ Eigen.

```cpp
double conditionNumber(Eigen::MatrixXd& A){
  Eigen::SVDJacobi<Eigen::MatrixXd> svd(A);
  Eigen::VectorXd singularValues = svd.singularValues();
  return singularValues(0) / singularValues(singularValues.size()-1);
}
```

Code 9: Condition number computation in C++.

Clearly, we do not get any improvement in the condition number after preconditioning in the case of the fractional heat equation, and that is why the PCGD performs worse in that case. More precisely, we do not get faster convergence of the iterative solver while we are adding the computation overhead of the preconditioning. In contrast, we can see from the plot that the condition number decreases after preconditioning in the pricing problem, and that leads to an improvement in the computational time, gaining a faster convergence that is dominating the computation overhead of the preconditioning, and that explain the corresponding results in table (1).

Conclusion

In this project, we have implemented the finite element method for option pricing under Feller–Lévy models, where the fractional Laplacian operator shows up in the parabolic partial differential equation that governs the price process. The finite element approximation of the latter operator has been established using the sinc quadrature. In the finite element solver, solving a linear system at each step has been done using the conjugate gradient descent iterative method, and that is due to the fact that the finite element matrix approximation matrix of the fractional Laplacian is not known explicitly and hence cannot be inverted. However, the sinc matrix approximation of the fractional Laplacian raised a computational challenge, namely the computationally expensive sum in (7) and future work may be done by parallelizing the latter sum to speed up the program.
After highlighting the C++ object-oriented implementation of the finite element method, we showed for the case of the fractional 1D heat equation an empirical second order convergence of the discretization error in the L^2 and L^∞ norms that was in accordance with the theoretical rates of convergence presented for the non-fractional case in[4]. In addition to that, the linear system showing up in the equation of the sinc matrix has been solved using three different direct and iterative methods, namely the LU-factorization technique, and the conjugate gradient descent with and without preconditioning. We have shown that iterative methods speed up the program especially when the number of mesh nodes is large enough, however the preconditioning seems to be of importance depending on the nature of the problem.

References

[1] Black, F., and Scholes, M. The pricing of options and corporate liabilities. *The Journal of Political Economy 81*, 3, 637654,.

[2] Bonito, A., and Pasciak, J. Numerical approximation of fractional powers of elliptic operators. *Mathematics of Computation 84* (2015).

[3] Böttcher, B., Schilling, R., and Wang, J. *Lévy Mattres III, Lévy-Type Processes: Construction, Approximation and Sample Path Properties.* Springer, 2013.

[4] Hilber, N., Reichmann, O., Schwab, C., and Winter, C. *Computational Methods for Quantitative Finance.* Springer Finance, Springer Heidelberg New York Dordrecht London, 2013.

[5] Komatsu, T. On the pathwise uniqueness of solutions of one-dimensional stochastic differential equations of jump type. *Proc. Japan Acad. Ser. A Math. Sci. 58*, 8, 353–356,.

[6] Merton, R. Option pricing when underlying stock returns are discontinuous. *Journal of Financial Economics 46*, 3, 125144,.

[7] Pazi, A. *Semigroups of Linear Operators and Applications to Partial Differential Equations.* Springer, 1983.

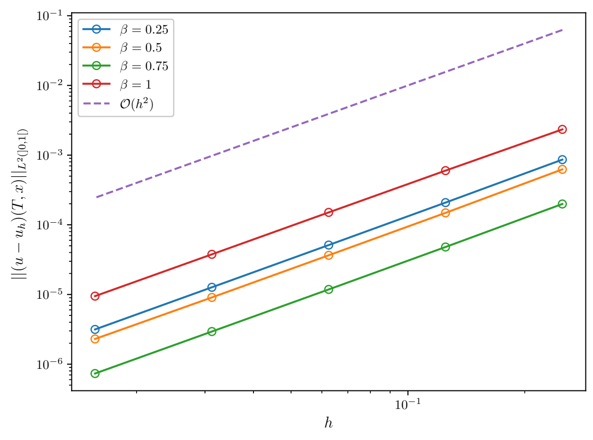
[8] Samorodnitsky, G., and Taqqu, M. *Stable Non-Gaussian Random Processes.* Chapman Hall, 1st edition, 1994.

[9] Yamada, T., and Watanabe, S. On the uniqueness of solutions of stochastic differential equations. *J. Math. Kyoto Univ*, 11, 15–167,.

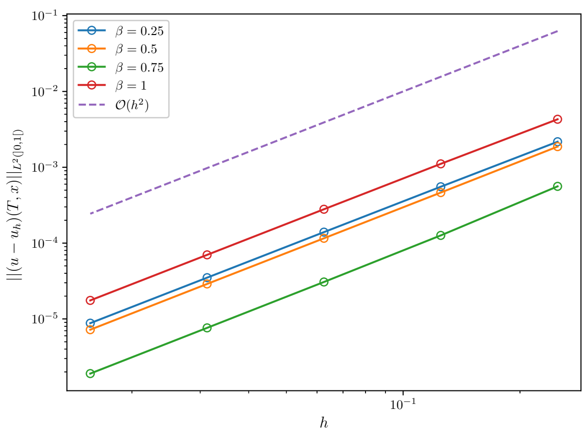
Appendix

Additional Convergence Results

The following plots show the second order convergence of the discretization error in the L^2 and L^∞ norms for $\theta \in \{1/4, 3/4\}$.

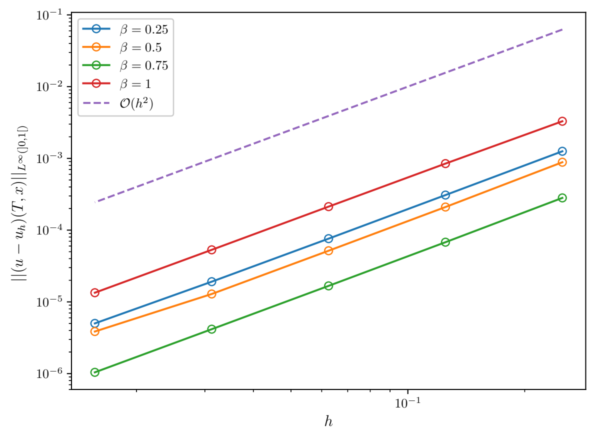


(a) $\theta = 1/4$.

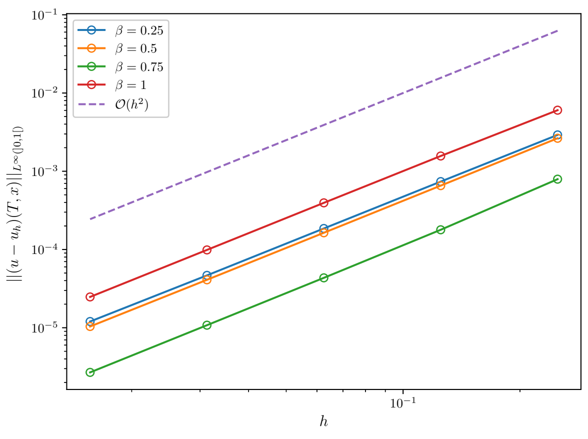


(b) $\theta = 3/4$.

Figure 6: The L^2-norm of the discretization error versus the mesh size h for $\theta \in \{1/4, 3/4\}$.



(a) $\theta = 1/4$.



(b) $\theta = 3/4$.

Figure 7: The L^∞-norm of the discretization error versus the mesh size h for $\theta \in \{1/4, 3/4\}$.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Option Pricing Under Feller—Levy Models

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Tanios	Ramy

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 30 January 2020	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.