PDE Solvers in Tensorflow Finance

Contents

Introduction

The `experimental.pde_v2` package provides ops to compute numerical solutions of linear parabolic second order partial differential equations.

We currently support equations of the following form:

$$\frac{\partial V}{\partial t} + \sum_{i,j=1}^{N} a_{ij}(\mathbf{x},t)\frac{\partial^2}{\partial x_i \partial x_j}\left[A_{ij}(\mathbf{x},t)V\right] + \sum_{i=1}^{N} b_i(\mathbf{x},t)\frac{\partial}{\partial x_i}\left[B_i(\mathbf{x},t)V\right] + c(\mathbf{x},t)V = 0. \tag{1}$$

Given $V(\mathbf{x}, t_0)$, the solver approximates $V(\mathbf{x}, t_1)$. The solver can go both forward ($t_1 > t_0$) and backward ($t_1 < t_0$) in time.

This includes as particular cases the backward Kolmogorov equation: $A_{ij} \equiv 1, B_{ij} \equiv 1, t_1 < t_0$, and the forward Kolmogorov (Fokker-Plank) equation: $a_{ij} \equiv 1, b_{ij} \equiv 1, t_1 > t_0$.

The spatial grid (i.e. the grid of $\mathbf{x}$ vectors) can be arbitrary in one-dimensional problems ($N = 1$). In multiple dimensions the grid should be rectangular and uniform in each dimension (the spacing in each dimension can be different, however).

We support Robin boundary conditions on each edge of the spatial grid:

$$\alpha(\mathbf{x}_b, t)V(\mathbf{x}_b, t) + \beta(\mathbf{x}_b, t)\frac{\partial V}{\partial \mathbf{n}}(\mathbf{x}_b, t) = \gamma(\mathbf{x}_b, t), \tag{2}$$

where $\mathbf{x}_b$ is a point on the boundary, and $\partial V/\partial \mathbf{n}$ is the derivative with respect to the outer normal to the boundary. In particular, Dirichlet ($\alpha \equiv 1, \beta \equiv 0$) and Neumann ($\alpha \equiv 0, \beta \equiv 1$) boundary conditions are supported.

Below we describe in detail how to use the solver, and then the algorithms it uses internally.

Usage guide

Simplest case: 1D PDE with constant coefficients

Let's start with a one-dimensional PDE with constant coefficients[1]:

$$\frac{\partial V}{\partial t} - D\frac{\partial^2 V}{\partial x^2} + \mu\frac{\partial V}{\partial x} - rV = 0. \tag{3}$$

Let the domain be $x \in [-1, 1]$, the distribution at the initial time be Gaussian, and the boundary conditions - Dirichlet with zero value on both boundaries:

$$\begin{aligned} &V(x, t_0) = e^{-x^2/2\sigma^2}(2\pi\sigma)^{-1/2}, \\ &V(-1, t) = 0, \\ &V(1, t) = 0. \end{aligned} \tag{4}$$

We're seeking $V(x, t_1)$ with $t_1 > t_0$.

Let's prepare the necessary ingredients. First, the spatial grid:

```
grid = grids.uniform_grid(minimums=[-1],
                          maximums=[1],
                          sizes=[300],
                          dtype=tf.float32)
```

This grid is uniform with 300 points between (and including) $x = -1$ and $x = 1$. The `grids` module provides other types of grids, for example a log-spaced grid. The `grid` object is a list of `Tensors`. Each element in the list represents a spatial dimension. In our example `len(grid) = 1` and `grid[0].shape = (300,)`.

We can also easily make a custom grid out of a numpy array:

```
grid_np = np.array(...)
grid = [tf.constant(grid_np, dtype=tf.float32)]
```

The next ingredient is the PDE coefficients:

[1] This is a diffusion-convection-reaction equation with constant diffusion coefficient D, drift μ, and reaction rate r.

```
d = 1
mu = 2
r = 3

def second_order_coeff_fn(t, grid):
  return [[-d]]

def first_order_coeff_fn(t, grid):
  return [mu]

def zeroth_order_coeff_fn(t, grid):
  return -r
```

Note the square brackets - these are required for conformance with multidimensional case.

Next, the values at the initial time t_0:

```
variance = 0.2
xs = grid[0]
initial_value_grid = (tf.math.exp(-xs**2 / (2 * variance)) /
    tf.math.sqrt(2 * np.pi * variance)
```

And finally, define the final time, initial time, and number of time steps:

```
t_0 = 0
t_1 = 1
num_time_steps = 100
```

Just as the spatial grid, the temporal grid can be customized: we can specify number of time steps, a size of a step, or even a callable that accepts the current time and returns the size of the next time step.

We now have all the ingredients to solve the PDE:

```
result_value_grid, final_grid, end_time, steps_performed =
    fd_solvers.solve_forward(
        start_time=t_0,
        end_time=t_1,
        num_steps=num_time_steps,
        coord_grid=grid,
        values_grid=initial_value_grid,
        second_order_coeff_fn=second_order_coeff_fn,
        first_order_coeff_fn=first_order_coeff_fn,
        zeroth_order_coeff_fn=zeroth_order_coeff_fn)
```

The resulting approximation of $V(x, t_1)$ is contained in `result_value_grid`. The solver also yields the final spatial grid (because the solver may modify the grid we've provided), the end time (because it may not be exactly the one we specified, e.g. in case of custom time step callable), and the number of performed steps.

To visualize the result, we can simply convert it to a numpy array (assuming we use Tensorflow 2.x or are in the eager mode of Tensorflow 1.x), and apply the usual tools:

```
import matplotlib.pyplot as plt

xs = final_grid[0].numpy()
vs = result_value_grid.numpy()
```

```
plt.plot(xs, vs)
```

TODO: insert pictures.

Non-constant PDE coefficients: Black-Scholes equation.

Let's now turn to an example of a PDE with non-constant coefficient - the Black- Scholes equation:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0. \tag{5}$$

All we need to change in the previous example is the PDE coefficient callables:

```
sigma = 1
r = 3

def second_order_coeff_fn(t, grid):
  s = grid[0]
  return [[sigma**2 * s**2 / 2]]

def first_order_coeff_fn(t, grid):
  s = grid[0]
  return [r * s]

def zeroth_order_coeff_fn(t, grid):
  return -r
```

As seen, the coordinates are extracted from the `grid` passed into the callables, and then can undergo arbitrary Tensorflow transormations. The returned tensors must be either scalars or have a shape implied by the `grid` (this is easy to do in 1D, and a bit more tricky in multidimensional case, more details are below).

The Black-Scholes equation is evolved backwards in time. Therefore use `fd_solvers.solve_backward` instead of `fd_solvers.solve_forward`, and make sure that `start_time` is greater than `end_time`.

That's it, we can now numerically solve the Black-Scholes equation.

Coefficients under the derivatives. Fokker-Planck equation.

As the next example let's consider the Fokker-Planck equation arising in the Black-Scholes model for the probability distribution of the stock prices:

$$\frac{\partial p}{\partial t} - \frac{1}{2}\sigma^2 \frac{\partial^2}{\partial S^2}\left[S^2 p\right] + \mu \frac{\partial}{\partial S}\left[Sp\right] = 0. \tag{6}$$

To specify coefficients under the derivatives, use `inner_second_order_coeff_fn` and `inner_first_order_coeff_fn` arguments. The specification is exactly the same as for `second_order_coeff_fn` and `first_order_coeff_fn`. For our example we write:

```
sigma = 1
mu = 2

def inner_second_order_coeff_fn(t, grid):
  s = grid[0]
```

```
  return [[-sigma**2 * s**2 / 2]]

def inner_first_order_coeff_fn(t, grid):
  s = grid[0]
  return [mu * s]

result_value_grid, final_grid, end_time, steps_performed =
    fd_solvers.solve(...,
        inner_second_order_coeff_fn=inner_second_order_coeff_fn,
        inner_first_order_coeff_fn=inner_first_order_coeff_fn)
```

We may specify both "outer" and "inner" coefficients, so the following is equivalent (albeit possibly less performant):

```
sigma = 1
mu = 2

def second_order_coeff_fn(t, grid):
  return [[-sigma**2 / 2]]

def first_order_coeff_fn(t, grid):
  return [mu]

def inner_second_order_coeff_fn(t, grid):
  s = grid[0]
  return [[s**2]]

def inner_first_order_coeff_fn(t, grid):
  s = grid[0]
  return [s]

result_value_grid, final_grid, end_time, steps_performed =
    fd_solvers.solve(...,
        second_order_coeff_fn=second_order_coeff_fn,
        first_order_coeff_fn=first_order_coeff_fn,
        inner_second_order_coeff_fn=inner_second_order_coeff_fn,
        inner_first_order_coeff_fn=inner_first_order_coeff_fn)
```

Batching

The solver can work with multiple PDEs in parallel. For example, let's solve Black-Scholes equation for European options with a batch of payoff values. European options imply the final condition $V(S, t_f) = (S - K)_+$, where `K` is the payoff. So with one payoff value we write:

```
payoff = 0.3
s = grid[0]
final_value_grid = tf.nn.relu(s - payoff)
```

With a batch of `b` payoff values, we need to stack the final value grids, so that `final_value_grid[i]` is the grid for `i`th payoff value. The simplest way to do this is by using `tf.meshgrid`:

```
payoffs = tf.constant([0.1, 0.3, 0.5])
s = grid[0]
```

```
payoffs, s = tf.meshgrid(payoffs, s, indexing='ij')
final_value_grid = tf.nn.relu(s - payoffs)
```

`tf.meshgrid` broadcasts the two tensors into a rectangular grid, and then we can combine them with any algebraic operations. Make sure that the batch dimension is the first one in the resulting Tensor.

There can be arbitrary number of batching dimensions, which is convenient when there are multiple parameters:

```
param1 = tf.constant(...)
param2 = tf.constant(...)
s = grid[0]
param1, param2, s = tf.meshgrid(param1, param2, s, indexing='ij')
final_value_grid = ...  # combine param1, param2 and s
```

After constructing `final_value_grid`, we pass it to `fd_solvers` as usual, and the `result_value_grid` will contain the batch of solutions.

We may also have different models for each element of the batch, for example:

```
payoffs = tf.constant([0.1, 0.3, 0.5])
sigmas = tf.constant([1.0, 1.5, 2])
rs = tf.constant([0.0, 1.0, 2.0])

s = grid[0]
payoffs, s = tf.meshgrid(payoffs, s, indexing='ij')
final_value_grid = tf.nn.relu(s - payoffs)

def second_order_coeff_fn(t, grid):
  s = grid[0]
  sigmas_mesh, s_mesh = tf.meshgrid(sigmas, s, indexing="ij")
  return [[sigmas_mesh**2 * s_mesh**2 / 2]]

def first_order_coeff_fn(t, grid):
  s = grid[0]
  rs_mesh, s_mesh = tf.meshgrid(rs, s, indexing="ij")
  return [rs_mesh * s_mesh]

def zeroth_order_coeff_fn(t, grid):
  s = grid[0]
  rs_mesh, s_mesh = tf.meshgrid(rs, s, indexing="ij")
  return -rs_mesh
```

This way we construct three PDEs: `i`-th equation has payoff `payoffs[i]` and model parameters `sigmas[i]`, `rs[i]`.

In the simplest case, the batch shapes of `final_value_grid` and PDE coefficient tensors match exactly, like in the last example. The precise requirement is as follows. The shape of value grid is composed of `batch_shape` and `grid_shape`. Both are determined from the shape of `final_value_grid`. The dimensionality `dim` of the PDE, i.e. the the rank of `grid_shape`, is inferred from the `grid` passed into the solver: `dim = len(grid)` (in all examples so far, `dim = 1`). `grid_shape` may evolve with time, so should be determined from the `grid` argument passed into `second_order_coeff_fn`, `first_order_coeff_fn` and `zeroth_order_coeff_fn`. Recall that `grid` is a List of 1D Tensors; `grid_shape` is a concatenation of shapes of these tensors. The requirement is that `second_order_coeff_fn(...)[i][j]`,

`first_order_coeff_fn(...)[i]` and `zeroth_order_coeff_fn(...)` must be tensors whose shape is broadcastable to the shape `batch_shape + grid_shape`.

This explains why we can't return just `-rs` from `zeroth_order_coeff_fn` in the example: the shape of `rs` is `(3,)`, which is not broadcastable to the required shape `batch_shape + grid_shape = (3, 300)`.

The boundary conditions (see below) can be also batched in a similar way. The coordinate grid and the temporal grid cannot be batched.

Boundary conditions

The solver supports boundary conditions in Robin form:

$$\alpha(\mathbf{x}_b, t)V(\mathbf{x}_b, t) + \beta(\mathbf{x}_b, t)\frac{\partial V}{\partial \mathbf{n}}(\mathbf{x}_b, t) = \gamma(\mathbf{x}_b, t), \tag{7}$$

where $\mathbf{x}_b$ is a point on the boundary, $\partial V/\partial n$ is the derivative with respect to the outer normal to the boundary. The functions α, β, γ can be arbitrary.

The boundary conditions are specified by callables returning α, β, γ as Tensors or scalars. For example, a boundary condition

$$V + 2\frac{\partial V}{\partial \mathbf{n}} = 3 \tag{8}$$

is defined as follows:

```
def boundary_cond(t, grid):
  return 1, 2, 3
```

Dirichlet and Neumann boundary conditions can be specified by setting $\alpha = 1, \beta = 0$ and $\alpha = 0, \beta = 1$ explicitly, or by using utilities found in `boundary_conditions.py`:

```
@boundary_conditions.dirichlet
def dirichlet_boundary(t, grid):
  return 1  # V = 1

@boundary_conditions.neumann
def neumann_boundary(t, grid):
  return 2  # dV/dn = 2
```

Note that the derivative is taken with respect to the outer normal to the boundary, not along the coordinate axis. So, for example, the condition $(\partial V/\partial x)_{x=x_{min}} = 2$ translates to $(\partial V/\partial \mathbf{n})_{x=x_{min}} = -2$.

The callables are passed into solver as a list of pairs, where each pair represents the lower and the upper boundary of a particular spatial dimension. In 1D this looks as follows:

```
grid = grids.uniform_grid(minimums=[-1], maximums=[1], ...)

def lower_boundary(t, grid):
  return ...  # boundary condition at x = -1

def upper_boundary(t, grid):
  return ...  # boundary condition at x = 1
```

```
result_value_grid, final_grid, end_time, steps_performed =
    fd_solvers.solve_forward(...,
        boundary_conditions=[(lower_boundary, upper_boundary)])
```

Multidimensional PDEs; Hull-Heston-White equation.

As a first example of a multi-dimensional PDE, let's consider a 2D PDE with constant coefficients[2]:

$$\frac{\partial V}{\partial t} - D_{xx}\frac{\partial^2 V}{\partial x^2} - 2D_{xy}\frac{\partial^2 V}{\partial x \partial y} - D_{yy}\frac{\partial^2 V}{\partial y^2} + \mu_x \frac{\partial V}{\partial x} + \mu_y \frac{\partial V}{\partial y} - rV = 0 \tag{9}$$

First, we create a rectangular grid:

```
x_min, x_max = -1.0, 1.0
y_min, y_max = -2.0, 2.0
x_size, y_size = 200, 300

grid = grids.uniform_grid(minimums=[y_min, x_min],
                          maximums=[y_max, x_max],
                          sizes=[y_size, x_size],
                          dtype=tf.float32)
```

The `grid` object is a list of two Tensors of shapes (300,) and (200,).

Currently, only uniform grids are supported in the multidimensional case. However, the steps of the grid can be different in each dimension.

The PDE coefficient callables look as follows:

```
d_xx, d_xy, d_yy = 1, 0.2, 0.5
mu_x, mu_y = 1, 0.5
r = 3

def second_order_coeff_fn(t, grid):
  return [[-d_yy, -d_xy], [-d_xy, -d_xx]]

def first_order_coeff_fn(t, grid):
  return [mu_y, mu_x]

def zeroth_order_coeff_fn(t, grid):
  return -r
```

The matrix returned by `second_order_coeff_fn` can be a list of lists like above, a Numpy array or a Tensorflow tensor - the requirement is that `second_order_coeff_fn(...)[i][j]` is defined for `i <= j < dim` and represents the corresponding PDE coefficient. The matrix is assumed symmetrical, and elements with `i > j` are never accessed. Therefore, they can be anything, so the following is also acceptable[3]:

```
def second_order_coeff_fn(t, grid):
  return [[-d_yy, -d_xy], [None, -d_yy]]
```

[2] This is a 2D diffusion-convection-reaction equation with anisotropic diffusion (i.e. D is now a 2x2 matrix).

[3] `None` can also be used to indicate that certain PDE terms are absent. For example, when $\mu_x = 0$, we can return `[mu_y, None]` from `first_order_coeff_fn`. If both $\mu_x, \mu_y = 0$, we can return `[None, None]` or simply pass `first_order_coeff_fn=None` into the solver.

The initial values grid can be constructed, as usual, with the help of `tf.meshgrid`. For example,

$$V(x, y, t_0) = (2\pi\sigma)^{-1} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{10}$$

translates into[4]

```
sigma = 0.1
ys, xs = grid
y_mesh, x_mesh = tf.meshgrid(ys, xs, indexing="ij")
initial_value_grid = (tf.math.exp(-(x_mesh**2 + y_mesh**2) / (2 * sigma))
    / (2 * np.pi * sigma))
```

Finally, call `fd_solvers.solve_forward` or `fd_solvers.solve_backward` as usual:

```
t_0 = 0
t_1 = 1
num_time_steps = 100

result_value_grid, final_grid, end_time, steps_performed =
    fd_solvers.solve_forward(
        start_time=t_0,
        end_time=t_1,
        num_steps=num_time_steps,
        coord_grid=grid,
        values_grid=initial_value_grid,
        second_order_coeff_fn=second_order_coeff_fn,
        first_order_coeff_fn=first_order_coeff_fn,
        zeroth_order_coeff_fn=zeroth_order_coeff_fn)
```

One way to visualize the result is by creating a heatmap:

```
plt.imshow(result_value_grid.numpy(),
           extent=[x_min, x_max, y_min, y_max],
           cmap='hot')
plt.show()
```

The boundary of the domain is rectangular, and each side can has its boundary condition (by default - Dirichlet with zero value). For example, let's "heat up" the right boundary and set zero flux across y-boundaries:

```
@boundary_conditions.dirichlet
def boundary_x_max(t, grid):
  return 1.0

@boundary_conditions.dirichlet
def boundary_x_min(t, grid):
  return 0.0
```

[4] With multiplicatively separable functions, we can avoid creating `x_mesh, y_mesh` with the help of `tf.tensordot`:

```
sigma = 0.1
ys, xs = grid
initial_value_grid = (tf.tensordot(tf.math.exp(-ys**2 / (2 * sigma)),
                                   tf.math.exp(-xs**2 / (2 * sigma)),
                                   [[], []]) / (2 * np.pi * sigma))
```

```
@boundary_conditions.neumann
def boundary_y_max(t, grid):
  return 0.0

@boundary_conditions.neumann
def boundary_y_min(t, grid):
  return 0.0

result_value_grid, final_grid, end_time, steps_performed =
    fd_solvers.solve_forward(...,
         boundary_conditions=[(boundary_y_min, boundary_y_max),
                              (boundary_x_min, boundary_x_max)])
```

The boundary conditions can be inhomogeneous. E.g. a "heat source" near $x = x_{max}, y = 0$ may look like this:

```
sigma = 0.2

@boundary_conditions.dirichlet
def boundary_x_max(t, grid):
  ys = grid[0]
  return tf.exp(-ys**2 / (2 * sigma**2))
```

The general requirement for the shape of the boundary condition tensors is that they must be either scalars or tensors of shape `batch_shape + grid_shape'`, where `grid_shape'` is `grid_shape` excluding the axis orthogonal to the boundary. For example, in 3D the value grid shape is `batch_shape + (z_size, y_size, x_size)`, and the boundary tensors on the planes $y = y_{min}$ and $y = y_{max}$ should be either scalars or have shape `batch_shape + (z_size, x_size)`.

As a more elaborate example, let's translate into code the Heston-Hull-White PDE, as defined in [1]:

$$\begin{aligned}&\frac{\partial\varphi}{\partial t} + \frac{1}{2}s^2 v\frac{\partial^2\varphi}{\partial s^2} + \frac{1}{2}\sigma_1^2 v\frac{\partial^2\varphi}{\partial v^2} + \frac{1}{2}\sigma_2^2 v\frac{\partial^2\varphi}{\partial r^2} + \\ &\rho_{12}\sigma_1 sv\frac{\partial^2\varphi}{\partial s\partial v} + \rho_{13}\sigma_2 s\sqrt{v}\frac{\partial^2\varphi}{\partial s\partial r} + \rho_{23}\sigma_1\sigma_2\sqrt{v}\frac{\partial^2\varphi}{\partial v\partial r} + \\ &rs\frac{\partial\varphi}{ds} + \kappa(\eta - v)\frac{\partial\varphi}{dv} + a(bt - r)\frac{\partial\varphi}{dr} - ru = 0\end{aligned} \tag{11}$$

with boundary conditions

$$\begin{aligned}&\varphi(0, v, r, t) = 0, \\ &\frac{\partial\varphi}{\partial s}(s_{max}, v, r, t) = 1, \\ &\varphi(s, 0, r, t) = 0, \\ &\varphi(s, v_{max}, r, t) = s, \\ &\frac{\partial\varphi}{\partial r}(s, v, r_{min}, t) = 0, \\ &\frac{\partial\varphi}{\partial r}(s, v, r_{max}, t) = 0.\end{aligned} \tag{12}$$

```
def second_order_coeff_fn(t, grid):
  s, v, r = tf.meshgrid(grid[0], grid[1], grid[2], indexing='ij')
  coeff_ss = s**2 * v / 2
  coeff_vv = sigma1**2 * v / 2
  coeff_rr = sigma2**2 * v / 2
  coeff_sv = rho12 * sigma1 * s * v / 2
  coeff_sr = rho13 * sigma2 * s * tf.sqrt(v) / 2
  coeff_vr = rho23 * sigma1 * sigma2 * tf.sqrt(v) / 2
  return [[coeff_ss, coeff_sv, coeff_sr],
          [None, coeff_vv, coeff_vr],
          [None, None, coeff_rr]]

def first_order_coeff_fn(t, grid):
  # Note how we avoid high-dimensional meshgrids and use tf.expand_dims to
  # make the shapes broadcastable to (s_size, v_size, r_size).
  s, v, r = grid
  s_mesh, r_mesh = tf.meshgrid(s, r, indexing='ij')
  coeff_s = r_mesh * s_mesh  # shape is (s_size, r_size)
  coeff_s = tf.expand_dims(coeff_s, axis=1)  # shape is (s_size, 1, r_size)

  coeff_v = kappa * (eta - v)  # shape is (v_size,)
  coeff_v = tf.expand_dims(coeff_v, axis=-1)  # shape is (v_size, 1)

  coeff_r = a * (b * t - r)  # shape is (r_size,)
  return [coeff_s, coeff_v, coeff_r]

def zeroth_order_coeff_fn(t, grid):
  return -r

@boundary_conditions.dirichlet
def boundary_s_min(t, grid):
  return 0.0

@boundary_conditions.neumann
def boundary_s_max(t, grid):
  return 1.0

@boundary_conditions.dirichlet
def boundary_v_min(t, grid):
  return 0.0

@boundary_conditions.dirichlet
def boundary_v_max(t, grid):
  s_mesh, r_mesh = tf.meshgrid(grid[0], grid[2], indexing='ij')
  return s_mesh  # shape is (s_size, r_size)

@boundary_conditions.neumann
def boundary_r_min(t, grid):
  return 0.0

@boundary_conditions.neumann
```

```
def boundary_r_max(t, grid):
  return 0.0
```

Customizing the time marching scheme

The solver allows specifying a time marching scheme. Time marching schemes define how a single time step is performed (after spatial discretization of a PDE), and they differ in numerical stability, accuracy, and performance.

To use, for example, the explicit scheme, which is less accurate and stable but much faster than the default (Crank-Nicolson) scheme, we write:

```
result_value_grid, final_grid, end_time, steps_performed =
    fd_solvers.solve_forward(...
        one_step_fn=steppers.explicit.explicit_step)
```

Currently the following schemes are supported for 1D:

1. Explicit,
2. Implicit,
3. Crank-Nicolson,
4. Weighted explicit-implicit,
5. Extrapolation scheme, [3]
6. Crank-Nicolson with oscillation damping (see discussion below).

For multidimensional problems we currently have:

1. Douglas ADI [1, 4].

By default, the Crank-Nicolson with oscillation damping is used for 1D problems and Douglas ADI with $\theta = 0.5$ - for multidimensional problems.

See below a detailed discussions of these schemes.

Implementation

The bird's eye view of a finite-difference PDE solving algorithm is

```
for time_step in time_steps:          (1)
   do_spatial_discretization()        (2)
   apply_time_marching_scheme()       (3)
```

The spatial discretization converts a PDE for $V(\mathbf{x}, t)$ into a linear system of equations for $\mathbf{V}(t)$, which is a vector consisting of $V(\mathbf{x}_i, t)$, where $\mathbf{x}_i$ are grid points. The resulting system of equations has the form

$$\frac{d\mathbf{V}}{dt} = \hat{L}(t)\mathbf{V} + \mathbf{b}(t), \tag{13}$$

where $\mathbf{b}$ is a vector, and $\hat{L}$ is a sparse matrix describing the influence of adjacent grid points on each other. $\hat{L}(t)$ and $\mathbf{b}(t)$ are defined by both PDE coefficients and the boundary conditions, evaluated at time t.

The job of a time marching scheme is to approximately solve Eq.(13) for $\mathbf{V}(t + \delta t)$ given $\mathbf{V}(t)$ (δt may be positive or negative).

All three lines in the above pseudocode represent the routines that are mostly independent from each other. Therefore, the solver is modularized accordingly:

- `fd_solvers.py` implement looping over time steps, i.e. line (1) in the pseudocode. The routines accept (and provide reasonable defaults for) `one_step_fn` argument, which is a callable performing both (2) and (3).
- `parabolic_equation_stepper.py` and `multidim_parabolic_equation_stepper.py` perform space discretization for one-dimensional and multidimensional parabolic PDEs, respectively. They accept a `time_marching_scheme` callable for performing (3).
- `crank_nicolson.py`, `explicit.py`, `douglas_adi.py` and other modules implement specific time marching schemes. Each of these modules has a `<scheme_name>_scheme` function for the scheme itself, and a `<scheme_name>_step` function. The latter is a convenience function which combines the given scheme and the appropriate space discretizer; it can be passed directly to `fd_solvers`.

This setup is flexible, allowing for easy customization of every component. For example, it's straightforward to implement a new time marching scheme and plug it in.

Below are the implementation notes for space discretizers and the time marching schemes.

Spatial discretization

The spatial derivatives are approximated to the second order of accuracy, and taking into account that the grid may be non-uniform. Consider first the 1D case, and denote x_0 the given point in the grid, and Δ_+, Δ_- the distances to adjacent grid points. The Taylor's expansion of a function $f(x)$ with smooth second derivative gives

$$f(x_0 + \Delta_+) = f(x_0) + f'(x_0)\Delta_+ + \frac{1}{2}f''(x_0)(\Delta_+)^2 + \mathcal{O}((\Delta_+)^3), \tag{14}$$

$$f(x_0 - \Delta_-) = f(x_0) - f'(x_0)\Delta_- + \frac{1}{2}f''(x_0)(\Delta_-)^2 + \mathcal{O}((\Delta_-)^3). \tag{15}$$

Solving this system of equations for $f'(x_0)$ and $f''(x_0)$ yields

$$f'(x_0) \approx C_+ f(x_0 + \Delta_+) + C_- f(x_0 - \Delta_-) - (C_+ + C_-)f(x_0), \tag{16}$$

$$C_\pm = \pm\frac{\Delta_\mp}{(\Delta_+ + \Delta_-)\Delta_\pm}. \tag{17}$$

and

$$f''(x_0) \approx D_+ f(x_0 + \Delta_+) + D_- f(x_0 - \Delta_-) - (D_+ + D_-)f(x_0), \tag{18}$$

$$D_\pm = \frac{2}{(\Delta_+ + \Delta_-)\Delta_\pm}. \tag{19}$$

Thus, space discretization a 1D homogeneous parabolic PDE yields Eq.(13) with $\mathbf{b} = 0$, and a tridiagonal matrix $\hat{L}$.

In multidimensional case the discretization is done similarly, but we additionally need to take care of mixed second derivatives. Since only uniform grids are currently supported in multidimensional case, we apply the usual approximation of mixed derivatives:

$$\frac{\partial^2 f}{\partial x \partial y}(x_0, y_0) \approx \frac{f(x_+, y_+) - f(x_-, y_+) - f(x_+, y_-) + f(x_-, y_-)}{4\Delta_x \Delta_y}, \tag{20}$$

where $x_\pm = x_0 \pm \Delta_x, y_\pm = y_0 \pm \Delta_y$ are adjacent grid points.

Considering that $\mathbf{V}$ in Eq.(13) is a vector, i.e. a flattened multidimensional value grid, the matrix $\hat{L}$ is now banded, with contributions coming from adjacent points in every dimension.

Boundary conditions

The most general form of boundary conditions we support is the Robin boundary conditions, see Eq. (2). When discretizing Eq. (2), we approximate the derivative to the second order of accuracy, just like we did with the PDE terms. The central approximation, Eq. (18), is however not applicable, because it uses the values at adjacent points on both sides. Instead, we express the boundary derivative via the two closest points on one side. Consider the lower boundary x_0, and the two closest grid points $x_1 = x_0 + \Delta_0$, and $x_2 = x_1 + \Delta_1$:

$$f(x_1) \approx f(x_0) + f'(x_0)\Delta_0 + \frac{1}{2}f''(x_0)\Delta_0^2, \tag{21}$$

$$f(x_2) \approx f(x_0) + f'(x_0)(\Delta_0 + \Delta_1) + \frac{1}{2}f''(x_0)(\Delta_0 + \Delta_1)^2 \tag{22}$$

Eliminating $f''(x_0)$ from this system, we obtain

$$f'(x_0) \approx \frac{(\Delta_0 + \Delta_1)^2 f(x_1) - \Delta_0^2 f(x_2) - \Delta_0(2\Delta_0 + \Delta_1)f(x_0)}{\Delta_0\Delta_1(\Delta_0 + \Delta_1)} \tag{23}$$

(see e.g. [2], §2.3; only uniform grids are considered there though).

Similarly, we express the derivative at the upper boundary x_n via the two closest points on the left of it: $x_{n-1} = x_n - \Delta_{n-1}$ and $x_{n-2} = x_{n-1} - \Delta_{n-2}$. The expression differs essentially only in the sign:

$$f'(x_n) \approx -\frac{(\Delta_{n-1} + \Delta_{n-2})^2 f(x_{n-1}) - \Delta_{n-1}^2 f(x_{n-2}) - \Delta_{n-1}(2\Delta_{n-1} + \Delta_{n-2})f(x_n)}{\Delta_{n-1}\Delta_{n-2}(\Delta_{n-1} + \Delta_{n-2})} \tag{24}$$

Substituting these two equations into Eq. (2) (taking into account that $\partial V/\partial \mathbf{n} = -\partial V/\partial x$ on the lower boundary, and $\partial V/\partial \mathbf{n} = \partial V/\partial x$ on the upper boundary), we get the following discretized versions of it:

$$V_0 = \xi_1 V_1 + \xi_2 V_2 + \eta, \tag{25}$$

$$V_n = \bar{\xi}_1 V_{n-1} + \bar{\xi}_2 V_{n-2} + \bar{\eta}, \tag{26}$$

where

$$\xi_1 = \beta(\Delta_0 + \Delta_1)^2/\kappa, \tag{27}$$

$$\xi_2 = -\beta\Delta_0^2/\kappa, \tag{28}$$

$$\eta = \gamma\Delta_0\Delta_1(\Delta_0 + \Delta_1)/\kappa, \tag{29}$$

$$\kappa = \alpha\Delta_0\Delta_1(\Delta_0 + \Delta_1) + \beta\Delta_1(2\Delta_0 + \Delta_1). \tag{30}$$

The expressions for $\bar{\xi}_1, \bar{\xi}_2$ and $\bar{\eta}$ are exactly the same, except $\Delta_{1,2}$ is replaced by $\Delta_{n-1,n-2}$, and of course α, β and γ come from the upper boundary condition.

The evolution of the values on the inner part of the grid, $V_0, \dots V_{n-1}$ is defined by a tridiagonal matrix, as discussed in the previous section:

$$\frac{dV_i}{dt} = L_{i,i-1}V_{i-1} + L_{i,i}V_i + L_{i,i+1}V_{i+1}, \qquad i = 1 \dots N-1 \tag{31}$$

Substituting Eqs. (25), (26), we obtain for the inner part $\mathbf{V}_{inner} = [V_1, ...V_{n-1}]^T$:

$$\frac{d\mathbf{V}_{inner}}{dt} = \hat{\hat{L}}\mathbf{V}_{inner} + \mathbf{b}, \tag{32}$$

where

$$\tilde{L}_{11} = L_{11} + \xi_1 L_{01}, \qquad \tilde{L}_{12} = L_{12} + \xi_2 L_{01}, \tag{33}$$

$$\tilde{L}_{n-1,n-1} = L_{n-1,n-1} + \bar{\xi}_1 L_{n,n-1}, \qquad \tilde{L}_{n-1,n-2} = L_{n-1,n-2} + \bar{\xi}_2 L_{n,n-1}, \tag{34}$$

$$\tilde{L}_{ij} = L_{ij} \qquad i = 2 \dots n-2, \tag{35}$$

$$b_1 = L_{01}\eta, \qquad b_{n-1} = L_{n,n-1}\bar{\eta}, \tag{36}$$

$$b_i = 0 \qquad i = 2 \dots n-2. \tag{37}$$

Note that in case of Dirichlet conditions ($\alpha = 1, \beta = 0$) this simplifies greatly: $\xi_{1,2} = 0$ (so there are no corrections to $\hat{L}$), and $\eta = \gamma$.

Thus to take into account the boundary conditions we

- apply the corrections to the time evolution equation given by Eqs. (33)-(37),
- apply the chosen time marching scheme to find the "inner" part of the values vector $\mathbf{V}_{inner}(t_{i+1})$ from Eq. (32) given $\mathbf{V}_{inner}(t_i)$,
- and finally, find V_0 and V_n from Eqs. (25), (26) at time $t + \delta t$, and append them to $\mathbf{V}_{inner}(t_{i+1})$ to get the resulting vector $\mathbf{V}(t_{i+1})$.

This approach generalizes straightforwardly to the multidimensional case: $\mathbf{V}_{inner}$ would be the value grid with all the boundaries "trimmed", and Eqs. (33)-(37) would similarly apply to the items next to the boundary.

Time marching schemes

Time marching schemes are algorithms for numerically solving the equation

$$\frac{d\mathbf{V}}{dt} = \hat{L}(t)\mathbf{V} + \mathbf{b}(t), \tag{38}$$

for $\mathbf{V}(t + \delta t)$ given $\mathbf{V}(t)$, with a small δt.

Note that in case of constant parameters $\hat{L}$ and $\mathbf{b}$ this equation has an exact solution involving the matrix exponent of $\hat{L}$. Calculating the matrix exponent is however infeasible in practice, primarily due to memory constraints. The matrix $\hat{L}$ is never explicitly constructed in the first place. Recall that in 1D the matrix $\hat{L}$ is tridiagonal, and in multidimensional case it is banded. Only the non-zero diagonals are constructed, and the time marching schemes make use of the sparseness of $\hat{L}$.

Explicit Scheme

The simplest scheme is the explicit scheme:

$$\frac{\mathbf{V}_1 - \mathbf{V}_0}{\delta t} = \hat{L}_0\mathbf{V}_0 + \mathbf{b}_0. \tag{39}$$

Here and below we use the notation $X_\alpha = X(t + \alpha\delta t)$ where X is any function of time and α is a number between 0 and 1.

From there we obtain

$$\mathbf{V}_1 = (1 + \delta t\hat{L}_0)\mathbf{V}_0 + \delta t\mathbf{b}_0. \tag{40}$$

The calculation boils down to multiplying a tridiagonal matrix by a vector. Tensorflow has the tf.linalg.tridiagonal_matmul Op, which can efficiently parallelize the multiplication on the GPU.

Unfortunately, the explicit scheme is applicable only with small time steps. For a well-defined PDE, the matrix $\delta t\hat{L}$ has negative eigenvalues (This statement includes PDEs solved both in forward and backward directions, i.e. δt can be both positive and negative). When an eigenvalue of $1+\delta t\hat{L}_0$ becomes less than -1, the contribution to $\mathbf{V}$ of the corresponding eigenvector grows exponentially with time. This does not happen to the exact solution, so the error of the approximate solution grows exponentially, i.e. the method is unstable. The stability requirements are $\delta t \ll \Delta x^2/D$ and $\delta t \ll \Delta x/\mu$, where D and μ are the typical values of second and first order coefficients of a PDE. This can constrain δt to unacceptably small values.

Implicit Scheme

The implicit scheme approximates the right hand side with its value at $t+\delta t$:

$$\frac{\mathbf{V}_1-\mathbf{V}_0}{\delta t}=\hat{L}_1\mathbf{V}_1+\mathbf{b}_1, \tag{41}$$

yielding

$$\mathbf{V}_1=(1-\delta t\hat{L}_0)^{-1}(\mathbf{V}_0+\delta t\mathbf{b}_1). \tag{42}$$

This takes care of the stability problem, because the eigenvalues of $(1-\delta t\hat{L}_0)^{-1}$ are between 0 and 1. However, we now need to solve a tridiagonal system of equation.

Tensorflow has the tf.linalg.tridiagonal_solve Op for this purpose. On GPU, it uses gtsv routines from Nvidia's CUDA Toolkit. These routines employ Parallel Cyclic Reduction algorithm (see e.g. [5]) which enables certain level of parallelization.

Weighted Implicit-Explicit Scheme and Crank-Nicolson Scheme

Weighted Implicit-Explicit Scheme is a combination of implicit and explicit schemes:

$$\frac{\mathbf{V}_1-\mathbf{V}_0}{\delta t}=\theta(\hat{L}_0\mathbf{V}_0+\mathbf{b}_0)+(1-\theta)(\hat{L}_1\mathbf{V}_1+\mathbf{b}_1), \tag{43}$$

where θ is a number between 0 and 1.

This yields

$$\mathbf{V}_1=(1-(1-\theta)\delta t\hat{L}_0)^{-1}[(1+\theta\delta_t\hat{L}_1)\mathbf{V}_0+\theta\delta t\mathbf{b}_0+(1-\theta)\delta t\mathbf{b}_1]. \tag{44}$$

The special case of $\theta=1/2$ is the Crank-Nicolson (CN) scheme:

$$\mathbf{V}_1=(1-\delta t\hat{L}_0/2)^{-1}[(1+\Delta_t\hat{L}_1/2)\mathbf{V}_0+\delta t(\mathbf{b}_0+\mathbf{b}_1)/2]. \tag{45}$$

CN scheme is stable (the eigenvalues are between -1 and 1), and, unlike schemes with any other value of θ, is second-order accurate in δt.

One can verify (by comparing the Taylor expansions) that replacing $\hat{L}_{0,1},\mathbf{b}_{0,1}$ with $\hat{L}_{1/2},\mathbf{b}_{1/2}$ in the right-hand side of Eq.(13) retains the second order accuracy, while saving some computation. Thus the final expression is

$$\mathbf{V}_1=(1-\delta t\hat{L}_{1/2}/2)^{-1}[(1+\delta t\hat{L}_{1/2}/2)\mathbf{V}_0+\delta t\mathbf{b}_{1/2}]. \tag{46}$$

Oscillations in Crank-Nicolson Scheme and Extrapolation Scheme

In financial applications, the initial or final data is often not smooth. For example, the call option payoff has a discontinuity in the first derivative at the strike. This can cause a severe drop in the accuracy of the Crank-Nicolson scheme: it results in oscillations around the points of discontinuity.

The reason of this effect is Crank-Nicolson scheme poorly approximating the evolution of high-wavenumber components of the initial/final condition function.

For the sake of discussion, assume $\delta t > 0$, $\mathbf{b}(t) = 0$ and $\hat{L}(t) = -\hat{A}$ (note that $\hat{A}$ has positive eigenvalues in this case). The exact solution of Eq. (38) is then

$$\mathbf{V}(t + \delta t) = \exp(-\hat{A}\delta t)\mathbf{V}(t). \tag{47}$$

Various time marching schemes can be viewed as approximations to the exponent. Crank-Nicolson scheme corresponds to the approximation

$$\exp(-y) \approx \frac{1 - y/2}{1 + y/2}. \tag{48}$$

Here $y = \lambda\delta t$ and λ is an eigenvalue of $\hat{A}$. This approximation, while being second order accurate for $y \ll 1$, is clearly inaccurate for $y \gtrsim 1$, and because it is negative there, the oscillations arise.

There are a few approaches to this problem found in the literature.

1) Replacing CN scheme with schemes based on Richardson extrapolation, i.e. taking a linear combination of results of making time steps of different sizes, with coefficients chosen to achieve desired accuracy. The simplest scheme corresponds to the following approximation [3]:

$$\exp(-y) \approx \frac{2}{(1 + y/2)^2} - \frac{1}{1 + y}. \tag{49}$$

   This means we are making two implicit half-steps, one implicit full step, and combine the results. This approximation is second order accurate for small y, just like CN, but is a more reasonable approximation of the exponent for large y. The cost of this improvement is having to solve three tridiagonal systems per step, compared to one system and one tridiagonal multiplication in CN scheme. The implementation of this scheme can be found in `extrapolation.py`. Higher orders of accuracy can be also achieved in this fashion. For example, combining two CN half-steps and CN full step with coefficients 4/3 and -1/3 yields a third-order accurate approximation [6].

2) Subdiving the first time step into multiple smaller steps [7]. The idea is that after the first step the irregularities in the initial conditions are already smoothened to a large enough degree that CN scheme can be applied to all further steps. But this smoothening itself needs to be modelled more accurately. By reducing δt we can ensure $y \lesssim 1$, for all components of initial/final condition function.

3) Subdividing the first step into two fully implicit half-steps ("Rannacher marching"), [8]. This is more computationally efficient than the two other approaches. However, the fully implicit steps, while indeed damping the oscillations, cause significant errors for low-wavenumber components, because of its first-order accuracy on small wavenumbers.

The optimal approach seems to be a combination of all three ideas: apply the extrapolation scheme to the first time step, or a few first steps, then proceed with CN scheme. This is mentioned in the end of [8]. Such approach adds minimal computational burden: only a finite number of extra tridiagonal systems to solve. It ensures damping of oscillations, i.e. a reasonable treatment of high-wavenumber components, while maintaining second-order accuracy of low-wavenumber components.

The high-wavenumber components are dampened by approximately $(\delta t \max_i(\lambda_i))^{-n_e}$, where n_e is number of extrapolation steps taken. One can approximate the maximum eigenvalue, and tune n_e to achieve desired level of oscillation damping.

The implementation of this can be found in `oscillation_damped_crank_nicolson.py`.

Multiple spatial dimensions, ADI

In case of multiple spatial dimensions, the matrix $\hat{L}$ is banded, making it much more difficult to do the matrix inversion in the Crank-Nicolson scheme.

The common workaround is the ADI (alternating direction explicit) method. The time step is split into several substeps, and each substep treats only one dimension explicitly. There is a number of time marching schemes based on this idea. An overview of them can be found, for example, on pages 6-9 of [1]. Here we summarize the simplest of them, the Douglas scheme, in our notations:

$$\mathbf{Y}_0 = (1 + \hat{L}_0 \delta t)\mathbf{V}_0 + \delta t \mathbf{b}_0, \tag{50}$$

$$\mathbf{Y}_j = \mathbf{Y}_{j-1} + \theta \delta t(\hat{L}_1^{(j)}\mathbf{Y}_j - \hat{L}_0^{(j)}\mathbf{V}_0 + \mathbf{b}_1^{(j)} - \mathbf{b}_0^{(j)}), \qquad j = 1 \dots dim, \tag{51}$$

$$\mathbf{V}_1 = \mathbf{Y}_{dim}. \tag{52}$$

Here $\hat{L}^{(j)}$ contains contributions to $\hat{L}$ coming from derivatives with respect to j-th dimension. Each $\hat{L}^{(j)}$ thus have only three nonzero diagonals. The contribution of the zeroth-order term is split evenly between $\hat{L}^{(j)}$. The contributions of mixed second-order terms are not included in any $\hat{L}^{(j)}$, and thus take part only in the first substep, which lacks any matrix inversion. $\mathbf{b}^{(j)}$ are contributions from boundary conditions along the j-th axis (boundary conditions are assumed to be the only source of $\mathbf{b}$). θ is a positive number.

The scheme is second-order accurate if there are no mixed derivative terms and $\theta = 1/2$, and first-order accurate otherwise. It is stable if $\theta \geq 1/2$.

The implementation can be found in `douglas_adi.py`. The main computational burden is solving large batches of tridiagonal systems plus transposing the value grid Tensor on each substep. To understand why the latter is necessary, recall that in implementation, $\mathbf{V}$ is not a vector, but a Tensor of shape `batch_shape + grid_shape`. When making the substep which treats j-th dimension implicitly, we should treat all the other dimensions as batch dimensions.

For example, consider a 3D case where the shape is `batch_shape + (z_size, y_size, x_size)`. When making the substep with respect that treats `y` implicitly, we transpose the value grid to the shape `batch_shape + (z_size, x_size, y_size)`, and send it to `tf.linalg.tridiagonal_solve`. From the point of view of the latter, the batch shape is `batch_shape + (z_size, x_size)`, exactly as we need. After that we transpose the result back to the original shape.

References

[1] Tinne Haentjens, Karek J. in't Hout. ADI finite difference schemes for the Heston-Hull-White PDE

[2] S. Mazumder. Numerical Methods for Partial Differential Equations. ISBN 9780128498941. 2015.

[3] D. Lawson, J & Ll Morris, J. The Extrapolation of First Order Methods for Parabolic Partial Differential Equations. I. 1978 SIAM Journal on Numerical Analysis. 15. 1212-1224.

[4] Douglas Jr., Jim (1962), "Alternating direction methods for three space variables", Numerische Mathematik, 4 (1): 41-63.

[5] D. Göddeke, R. Strzodka, "Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid", IEEE Transactions on Parallel and Distributed System, vol. 22, pp. 22-32, Jan. 2011.

[6] A. R. Gourlay and J. Ll. Morris. The Extrapolation of First Order Methods for Parabolic Partial Differential Equations, II. SIAM Journal on Numerical Analysis Vol. 17, No. 5 (Oct., 1980), pp. 641-655.

[7] D.Britz, O.Østerby, J.Strutwolf. Damping of Crank–Nicolson error oscillations. Computational Biology and Chemistry, Vol. 27, Iss. 3, July 2003, Pages 253-263.

[8] Giles, Michael & Carter, Rebecca. (2005). Convergence analysis of Crank-Nicolson and Rannacher time-marching. J. Comput. Finance. 9.