Finite Difference Numerical PDE Solver in Scala

Contents

1 Introduction

Partial differential equations are used in most fields to model various phenomena. In science they are used to model heat, sound, electrostatics, electrodynamics, fluids, elasticity, bacteria growth and so on. Outside of science, they are used in economics to model asset pricings (see Black-Scholes model).

Despite their widespread useage, many PDEs are not simple to solve explicitly. For those cases, one must rely numerical methods. There exists many types of numerical methods to solve PDEs, but the three most common ones are: the finite difference method, where one approximates functions by values at certain grid points and derivatives are appproximated through differences in these values; the finite element method, where functions are split into baiss functions and then the PDE is solved in integral form; finally, there is the spectral method which represents functions as a sum of basis functions.

This project implemented various forms of the finite difference method to solve different categories of PDEs.

2 Theory

2.1 Finite Differences

When using the finite difference method, one has to assume that the function is well behaved, thus we can expand it in a Taylor's series.

$$\begin{aligned} f(x_0+h) &= f(x_0) + \frac{f'(x_0)}{1!}h + \frac{f''(x_0)}{2!}h^2 + ... \\ &= f(x_0) + \frac{f'(x_0)}{1!}h + O\left(h^2\right) \end{aligned}$$

where h is the step size. From there we can write out

$$\begin{aligned} f'(x_0) &= \frac{f(x_0+h) - f(x_0)}{h} + \frac{1}{h}O\left(h^2\right) \\ f'(x_0) &\approx \frac{f(x_0+h) - f(x_0)}{h} \end{aligned}$$

This particular approximation would be called a forwards approximation because we take a step forwards in x to obtain the derivative's approximation. A backwards step would be

$$f'(x_0) \approx \frac{f(x_0) - f(x_0-h)}{h}$$

In this case, the truncation error is $\frac{f''(x_0)}{2!}h$ which we obtain because we cut off the Taylor expansion early.

Other derivative discretizations for higher order, bivariate functions are

$$\frac{\partial u}{\partial x} \approx \frac{u(x+\Delta x,t)-u(x,t)}{\Delta x}$$

$$\frac{\partial u}{\partial t} \approx \frac{u(x,t+\Delta t)-u(x,t)}{\Delta t}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta,t)}{\Delta x^2}$$

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u(x,t+\Delta t)-2u(x,t)+u(x,t+\Delta t)}{\Delta t^2}$$

$$\frac{\partial u}{\partial x \partial t} = \frac{\partial u}{\partial t \partial x} \approx \frac{u(x+\Delta x,t+\Delta t)-u(x+\Delta x,t-\Delta t)-u(x-\Delta x,t+\Delta t)+u(x-\Delta x,t-\delta t)}{4\Delta x \Delta t}$$

2.2 Generating Solutions to PDEs

A simple first order PDE looks like

$$a\frac{\partial u}{\partial x}+b\frac{\partial u}{\partial t}+cu(x,t)+f(x,t)=0$$

Where a, b and c are functions of x and t. First we split up the domain of the function into a grid. So we have x_i where $0 < i < m$ where x_0 is the minimum value on the boundary x can have and x_m is the maximum. Likewise, we have t_j with $0 < j < n$. Now at these grid points we shall write $u(x_i, t_j)$ as $u_{i,j}$ and similarly for a, b, c and f.

$$a_{i,j}\left(\frac{u_{i+1,j}-u_{i-1,j}}{2h}\right)+b_{i,j}\left(\frac{u_{i,j+1}-u_{i,j}}{k}\right)+c_{i,j}u_{i,j}+f_{i,j}=0 \tag{1}$$

where we have approximated the x derivative by a central difference. Now we can explicitly solve for $u_{i,j+1}$ as

$$u_{i,j+1}=\frac{k}{b_{i,j}}\left(a_{i,j}\left(\frac{u_{i-1,j}-u_{i+1,j}}{2h}\right)+\left(\frac{b_{i,j}}{k}-c_{i,j}\right)u_{i,j}-f_{i,j}\right)$$

As shown in 1.

Many other methods exist to solve PDEs, but they all involve similar ideas. That is, discretize the derivatives in the PDE so that one may approximate it by nearby grid-points. For further reading refer to Finite Difference Methods for Differential Equations by Randall J. LeVeque[1].

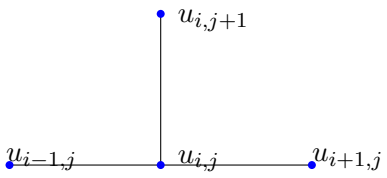

Figure 1: Explicit method for first order PDEs.

3 Examples

3.1 Useage

To use the solver, first define the variables of the function

```
import pde.model.expression._
val x = Variable("x")
val t = Variable("t")
val u = FuntionVariable("u", x, t)
```

Then define the pde and boundary conditions

```
val pde = d(u, t) := 4*dd(u, x, x)
val boundary = Boundary(
(u(x, 0) := Sin(Pi*x/L), from(0 to 10)),
(u(0, t) := 0, from(0 to 2*L)),
(u(L, t) := 0, from(0 to 2*L))
)
```

and finally solve it using

```
val solution = Solver.solve(pde, boundary)
val point11 = solution(1, 1)
```

optionally giving arguments for *xstep* and *tstep*.

3.2 First Order PDEs

Sample Code

```
object FirstOrderTest extends App {
  val x = Variable("x")
  val t = Variable("t")
  val u = new FunctionVariable("u", x, t)
```

```
  val testPDE = t*d(u, t) + x*d(u, x)  - u := 0
  val boundary = Boundary(
    (u(x, 0) := 2*x, from(0 to 10)),
    (u(0, t) := 3*t, from(0 to 10)),
    (u(10, t):= 20+3*t, from(0 to 10))
  )
  val boundaryR = Boundary(
    (u(x, 0) := 2*x, from(0 to 10)),
    (u(0, t) := 3*t, from(0 to 10)),
    (u(10, t):= 20+3*t, from(0 to 10)),
    (u(x, 10):= 2*x+30, from(0 to 10))
  )
  val testPDE2 = d(u, t) + d(u, x) - u := -x*t
  val boundary2 = Boundary(
    (u(x, 0) := x+2, from(0 to 10)),
    (u(0, t) := t+2, from(0 to 10)),
    (u(10, t):= 10+2+t+10*t, from(0 to 10)),
    (u(x, 10) := 10+2+x+4*x, from(0 to 10))
  )
  def time[A](f: => A) = {
    val s = System.nanoTime
    val ret = f
    println("time: "+(System.nanoTime-s)/1e6+"ms")
    ret
  }

  val realSolution = (x: Double, t: Double) => 2*x+3*t;
  val realSolution2 = (x: Double, t: Double) => x*t+x+t+2;
  val solution1 = {
    time {Solver.solve(testPDE, boundary, xstep = 0.1, tstep = 0.1)}
  }

  val solution2 = {
    time {Solver.solve(testPDE2, boundary2, xstep = 0.1, tstep = 0.1)}
  }

}
```

For brevity, the print and import statements were ommitted.
The output:

```
1.0 * u_t + 1.0 * u_x + (-(1.0)) * u + -((-(x) * t)) = 0
time: 89664.145544ms
Solution to (t) * u_t + (x) * u_x + (-(1.0)) * u = 0
Generated Point (0.1, 0.1): [0.4499999999999983,0.5500000000000022]  (1.453004383478182E-15)(abs)
0.5
Real: 0.5
Generated Point (0.5, 0.5): [2.449999999999945,2.550000000000058]  (4.4134289052031474E-14)(abs)
2.5000000000000004
Real: 2.5
Generated Point (1, 1): [4.949999999999765,5.050000000000244]  (1.8882525167485055E-13)(abs)
5.000000000000001
Real: 5.0
Generated Point(5, 5): [24.94999999999385,25.05000000000611]  (4.7692331633346055E-12)(abs)
25.000000000000004
Real: 25.0
Generated Point(7.5, 7.5): [37.44999999998569,37.55000000001416]  (1.1210878933689465E-11)(abs)
37.500000000000014
Real: 37.5
Generated Point(9.9, 9.9): [47.01999999997645,51.98000000002320]  (1.8086037561786404E-11)(abs)
49.500000000000014
Real: 49.5
-------------------------------------------------------------------------------------
time: 17245.412616ms
Solution to 1.0 * u_t + 1.0 * u_x + (-(1.0)) * u + -((-(x) * t)) = 0
Generated Point (0.1, 0.1): [2.187999999999995,2.232000000000006]  (4.355657724309832E-15)(abs)
2.21
Real: 2.21
Generated Point (0.5, 0.5): [3.219999999999946,3.280000000000056]  (4.535128333101159E-14)(abs)
3.25
Real: 3.25
Generated Point (1, 1): [4.959999999999803,5.040000000000203]  (1.6680218275411806E-13)(abs)
4.999999999999997
Real: 5.0
Generated Point(5, 5): [36.87999999996662,37.12000000003413]  (2.810412682693202E-11)(abs)
36.999999999999304
Real: 37.0
Generated Point(7.5, 7.5): [73.07999999956860,73.42000000044115]  (3.621061309333247E-10)(abs)
73.24999999999027
Real: 73.25
Generated Point(9.9, 9.9): [116.8399999951858,122.7800000049254]  (4.024409208844229E-9)(abs)
119.80999999988663
Real: 119.81000000000002
```

Where the first line, the one with "Generated Point" is the point in interval notation, and the line right under it is the central value. The third line, "Real", is the exact value.

3.3 Laplace's Equation

Code:

```
object SecondOrderTest extends App {
  val x = new Variable("x")
  val t = Variable("t")
  val u = new FunctionVariable("u", x, t)
  val laplace = dd(u, x, x) + dd(u, t, t) := Const(0)
  val boundary = new RectBoundary(
    (u(x, 0) := 0, from(0 to 20)),
    (u(x, 10):=Sin(Pi * x/20)*5, from(0 to 20)),
    (u(0, t) :=0, from(0 to 10)),
    (u(20, t):=0, from(0 to 10))
  )
  val solution= Solver.solve(laplace, boundary, xstep = 0.2, tstep = 0.4)

  val realLSolution = (x: Double, t: Double) =>
  (5/scala.math.sinh(Pi/2)) * scala.math.sin(Pi * x / 20) * scala.math.sinh(Pi*t/20)
}
```

Output

```
Running SecondOrderTest
Generated Point (4, 2): [0.4053465601093503,0.4101618942637927]  (4.201161737557659E-14)(abs)
0.4077542271865715
Real: 0.40783644615089115
Generated Point(18, 9): [1.159954366264431,1.165624827362555]  (3.525639168648441E-14)(abs)
1.1627895968134934
Real: 1.2984714123775343
Generated Point(10, 5): [1.851338280392296,1.861537303840700]  (1.543027440455468E-13)(abs)
1.8564377921164983
Real: 1.887349271785328
```

4 Implementaiton

4.1 Modeling PDEs

The domain specific language to represent PDEs has `Expression` as the super-class for everything. In `Expression`, there are methods defined for $+, -, *$ and / when interacting with another `Expression`. These output a new node, `Add, Sub, Mul, Div`, according ot the operation. One type of `Expression` which is treated differently is `Function`'s, which can be either `Function Variables, d` or `dd`. `d` and `dd` being there to represent derivatives.

For `Expression`'s there is a method `:=` to convert split up the expression into functions which multiply the derivative and then with those it outputs a PDE.

4.2 Tracking Errors

The truncation of floating-point values and finite precision errors on the calculations are all taken care off by Eva Darulova's `SmartFloat` class[2]. As for the errors related to the truncation of the Taylor expansion, that is done separately and the error is added after.

In all cases, when using the finite difference method, the truncation error on a point is related to a higher order derivative. That derivative can also be discretized. From there using the already computed solutions points, we add the error to the solution. Going back to the example in the introduction 1.

$$a_{i,j}\left(\frac{u_{i+1,j}-u_{i-1,j}}{2h}\right)+b_{i,j}\left(\frac{u_{i,j+1}-u_{i,j}}{k}\right)+c_{i,j}u_{i,j}+f_{i,j}=0$$

The term

$$a_{i,j}\left(\frac{u_{i+1,j}-u_{i-1,j}}{2h}\right)$$

has a main truncation error of $\frac{1}{6}h^2u'''(x)$, which we approximate by

$$u_{xxx}(x,t)\approx\frac{u_{i+2,j}-2u_{i+1,j}+2u_{i-1,j}-u_{i-2,j}}{h^3}$$

similarly for the second term we compute the error related to the truncation and add it to the error on the solution.

One thing to note is that the truncation error on the truncation error is not computed. For the third derivative computed above, it has error $O\left(h^4\right)$ which is simply ignored. This is because, for one, we could keep computing errors to no end, but mostly because it becomes very insignificant. The error on u_x was $O\left(h^2\right)$, so if we multiply that by the error on u_{xxx}, we get an $O\left(h^6\right)$ error which is next to nothing.

The Taylor series truncation errors are all computed once all solution points have been generated. For one, the error depends on the surrounding points, so we need their values to compute it, but the main reason is to avoid having the error values propagate along the computations prematurely.

The last place where an additional piece of error may surface is when accessing the data. Since the solution was only computed at specific gridpoints, if the code requests a data point off the grid, yet still inside the defined boundary, a weighted average of the surrounding points is computed and the appropriate error is added to it.

$$\bar{x} = \frac{\sum_{i=1}^{n}(w_i x_i)}{\sum_{i=1}^{n} w_i} \qquad \sigma_{\bar{x}}^2 = \frac{1}{\sum_{i=1}^{n} w_i}$$

In this case the weights are the inverse distance from the closest grid-point. The closer the grid-point is to the desired value, the heavier it weights in the average.

5 Experimental Results

As far as the results go, for the examples that have been tested it seems to work well, but there is admitedly many more PDEs it could be tested on. In the examples above, we can see that for first order PDEs, the results are within the error interval of the analytic solution. For the Laplace's equation and other second order equations, the truncation error may be underestimated, since the error computed is more of an order of magnitude error than an absolute one.

5.1 Speed

Various tests we're done regarding how big the grid can be before it becomes unfeasible to use the solver. Without increasing Java's default heap space, the maximum size grid possible is in the whereabouts of 1250 by 1250.

In terms of speed, there are equivalent solvers for first order solutions using `Double`s instead of `SmartFloat`s. The speed increase when computing using normal floating-points is about 10 times faster.

6 Further Work

Due to time restraints, not all PDEs can actually be solved using this solver. The ones which do work are: first order equations, Laplace's equation, Poisson's equation and the heat equation whith scalars. The work is there for general heat equations, but the computation time was horrendous. For the heat equation, one must solve an $n \times n$ tridiagonal matrix at each time step. Though this can be done in linear time, when the matrix is of size 1000×1000 it is incredibly slow. With scalars, the matrix does not have to be regenerated at each time step, only the result column has to be computed.

For general 2nd order equations, one would have to solve a slightly more general matrix, which has not yet been implemented yet. In addition to this, other things which would have to be implemented are solving using Newman boundary conditions, where one specifies $\frac{\partial u}{\partial x}$ at some points. In terms of boundaries, more general boundaries may also want to be specified by the user. As it is now, the boundaries are very rigid in their definition. Spherical coordinates could also be usefull for radial Laplace's equation.

References

[1] Randal J. Leveque, *Finite Difference Methods for Ordinary and Partial Differential Equations, Steady State and Time Dependent Problems* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, July 2007.

[2] Eva Darulova, Viktor Kuncak, *Trustworthy Numerical Computation in Scala*, School of Computer and Communication Sciences (I&C) - Swiss Federal Institute of Technology (EPFL), Switzerland, 2001.