Google Summer of Code
Numerically Solving PDEs in Scala

Timothy Spurdle

Abstract

Although multiple methods exist to numerically solve PDEs. Depending on the boundary conditions, not all of them will always work. The method which can be adjusted to work in the most cases is the finite difference method. Although it is an extremely simple method, many subtlties arrise when deciding on the step size and when calculating errors.

1 Introduction

Partial differential equations are used to model many of our world phenomenons. From the heat equation to Laplace's equation, all types of PDEs exist and many fields of science use them to model specific situations. Often, one can not actually find a real solution for a PDE, normally because of the complex nature of it. Despite that, numerical solutions can almost always be used.

1.1 Basic Interface

I would like to implement some of these methods as well as error propagation code. The first thing I will be working on will be a parser, so that one can write PDE's in the more natural way. Scala offers a good library for parsing strings into tokens. I'm thinking of having the PDE written as something like

```
val eq = PDE("a u_{xx} + b u_{tx} + c u_{tt} + d u_t+ e u_x + f u = g(x, t)")
```

Where a, b, c, d, e, f and g are functions of x and t.

This follows LaTeX's convention, so potentially one could even adjoin my project to a LaTeX/Scala PDE solver/generator. Unless I can think of a more natural way to express PDEs, I'll keep it at this.

The above function the gets converted to something like:

```
class LinearPDE2 (a: (Double, Double) => Double,
                  b: (Double, Double) => Double,
                  c: (Double, Double) => Double,
                  d: (Double, Double) => Double,
                  e: (Double, Double) => Double,
                  f: (Double, Double) => Double,
                  g: (Double, Double) => Double)
```

At which point to generate a solution, one calls either

```
val (u, delta) = eq.generate_solution(boundary)
```

to generate a function u which is the solution and δ the error on the approximation. Although u is a function, the solution is stored as a matrix of solution points. When one calls say $u(x, y)$ unless x and y are points at exact step sizes, a weighted average of nearby points is returned. To call the function, one needs to define boundary conditions. In cartesian coordinates, that would be at least 3 functions which define an enclosing area. I will start by doing it for rectangular boundaries and then move on to other shapes. A rectangular boundary is defined by

```
class Boundary (
    val b1: ((Double, Double) => Double, (Double, Double), (Int, Double)),
    val b2: ((Double, Double) => Double, (Double, Double), (Int, Double)),
    val b3:  ((Double, Double) => Double, (Double, Double), (Int, Double)),
    val b4:  ((Double, Double) => Double, (Double, Double), (Int, Double))
    )
```

For example,

```
((x: Double, t: Double) => x*t, (-1, 1), (1, 3))
```

would represent

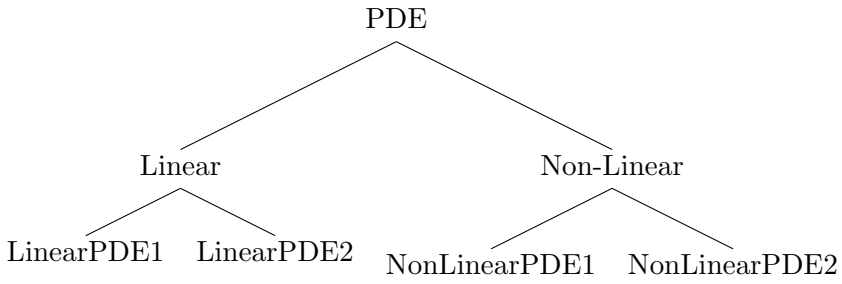
$$u(x,3) = g(x) = x * 3 \ for \ -1 \leq x \leq 1 \tag{1}$$

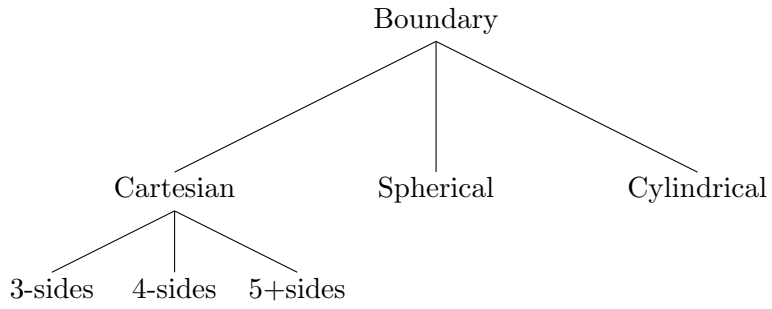
So the first input is a function. The second is the interval from which it is valid and the third is to represent what variable is held constant and at what point. That notation is confusing, but I have not thought of any other solution for the while. Once the boundary is created, assertions are made to ensure that the connecting points match.

When calling the function which generates a solution, derivatives of the boundary are computed to ensure that the boundary is viable for the PDE.

1.2 Inheritance

I have yet to find any place where inheritance could save a large amount of code, but for expansion purposes, some basic inheritance models will look like:





1.3 Lightweight Modular Staging

Scala's Lightweight Modular Staging(LMS) would make the code look something like:

```
trait Equation extends Base{
    abstract class Expr[T]
    case class Funct[T](fn: Function2[Double, Double, Double]) extends Expr[T]
    case class fnName[T](x: String) extends Expr[T]
    case class Const[T](x: T) extends Expr[T]
    case class Sym[T](n: Double) extends Expr[T]
```

```
    abstract class Def[T]

    def findOrCreateDefinition[T](rhs: Def[T]): Sym[T]
    implicitdeftoExp[T](d: Def[T]): Exp[T] = findOrCreateDefinition(d)
}

trait LinearPDE2 extends Base {
    object LinearPDE2 {
        def apply[T](a: Rep[Function2[Double, Double, Double]],
                     b: Rep[Function2[Double, Double, Double]],
                     c: Rep[Function2[Double, Double, Double]],
                     d: Rep[Function2[Double, Double, Double]],
                     e: Rep[Function2[Double, Double, Double]],
                     f: Rep[Function2[Double, Double, Double]],
                     g: Rep[Function2[Double, Double, Double]]) = PDE_new(a, b, c, d, e, f, g)
    }

    def PDE_new(a: Rep[Function2[Double, Double, Double]],
                b: Rep[Function2[Double, Double, Double]],
                c: Rep[Function2[Double, Double, Double]],
                d: Rep[Function2[Double, Double, Double]],
                e: Rep[Function2[Double, Double, Double]],
                f: Rep[Function2[Double, Double, Double]],
                g: Rep[Function2[Double, Double, Double]]): Rep[LinearPDE2]
}

trait LinearPDE2OpsExp extends BasePDE {
    case class PDENew(a: Expr[Double],
                      b: Expr[Double],
                      c: Expr[Double],
                      d: Expr[Double],
                      e: Expr[Double],
                      f: Expr[Double],
                      g: Expr[Double])
      extends Def[LinearPDE2]

    def pde_new(a: Expr[Double],
                      b: Expr[Double],
                      c: Expr[Double],
                      d: Expr[Double],
                      e: Expr[Double],
                      f: Expr[Double],
                      g: Expr[Double]): Expr[PDE] = PDENew(a, b, c, d, e, f, g)

}
```

Where the above code is for 2nd order linear PDEs specifically. After that one will need to translate methods into nodes representing a PDE. Something like

```
trait LinearPDE2 extends Base {
```

```
    object LinearPDE2 {
        def apply[T](a: Rep[Function2[Double, Double, Double] ],
                     b: Rep[Function2[Double, Double, Double] ],
                     c: Rep[Function2[Double, Double, Double] ],
                     d: Rep[Function2[Double, Double, Double] ],
                     e: Rep[Function2[Double, Double, Double] ],
                     f: Rep[Function2[Double, Double, Double] ],
                     g: Rep[Function2[Double, Double, Double] ])
                       = PDE_new(a, b, c, d, e, f, g)
    }
    def generate_solution (V: Rep[Boundary]) = PDE_solve(V)

    def PDE_new(a: Rep[Function2[Double, Double, Double] ],
                b: Rep[Function2[Double, Double, Double] ],
                c: Rep[Function2[Double, Double, Double] ],
                d: Rep[Function2[Double, Double, Double] ],
                e: Rep[Function2[Double, Double, Double] ],
                f: Rep[Function2[Double, Double, Double] ],
                g: Rep[Function2[Double, Double, Double] ]): Rep[LinearPDE2]
    def PDE_solve(V: Rep[Boundary]): Function2[Double, Double, Double]
}

trait LinearPDE2OpsExp extends BasePDE {
    case class PDENew(a: Expr[Double],
                      b: Expr[Double],
                      c: Expr[Double],
                      d: Expr[Double],
                      e: Expr[Double],
                      f: Expr[Double],
                      g: Expr[Double])
      extends Def[LinearPDE2]

    def pde_new(a: Expr[Double],
                b: Expr[Double],
                c: Expr[Double],
                d: Expr[Double],
                e: Expr[Double],
                f: Expr[Double],
                g: Expr[Double]): Expr[PDE] = PDENew(a, b, c, d, e, f, g)
}
```

2 Theory

This section is incomplete and will be build up gradually.

2.1 List of sections used from Leveque

1. All of section 1
2. .2-.9
3. .3

6. .1-.4

7. Most of 7

2.2 Fourier Analysis

This method only applies to linear PDEs, but it has the advantage of being relatively simple and producing exact solutions up to rounding errors.

Page 150[158] of Leveque's notes.

3 Schedule/Milestones

July 8^{st}: Completed the LMS structure for Linear 1st and 2nd order PDEs.

July 22^{th}: Solutions for 1^{st} and 2^{nd} order PDEs can now be generated. Code will first be written for linear PDEs using Fourier Analysis, then for finite difference method for elliptical equations and finally method of lines.

Writeup of documentation for the GSoCs midterm evaluation will be done before the 15th.

Aug 5^{th}: Added code in traits LinearPDE* and LinearPDE*OpsExp to take into account error generation.

Aug 19^{th}: Adjust computation methods for the stiffness of the equation.

Sept 2^{nd}: Finalize documentation and code for GSoC final evaluations.