PDE Solver Mid-Term Report

1 Week by Week Summary

This is an approximate week by week summary of the work which has been done thus far. Since this summary was written after 5 weeks of work based on memory, some past details may be innacurate.

June 17th:
- Began the project by implementing a solver for elliptical 2nd order bivariate pdes. The pdes we're initially stored as a series of functions, similarly for the boundaries.
- This usage was very unatural, so I was advised to change it. The code written during this week used later on with almost no modificaitons.

June 24th:
- Most of this week was spent designing the interface and structure of the solver.
- I considered making a parser to convert strings to pde objects, but a DSL seemed more effective.
- Instead of storing a pde as a set of functions, it is now a set of `Expr`'s (Expressions). `Expr`'s are abstract syntax trees representing algebraic expressions.
- This also allowed for the distinction between function variables (`FunctionVariable`) and other variables (`NonFunctionVariable`).
- The syntax for a pde would be something like:

  ```
  val laplace = dd(u, x, x) + dd(u, t, t) === 0
  ```

  where `u` has already been defined as a function variable, and `x` and `t` have already been set as non-function variables. `dd` and `d` are classes used to represent derivatives. `===` is an `Expr`method which takes in as input another `Expr`, in this case `Const(0)`, and spits out a pde.
- The code for most expression and variable code was all written this week.

July 1st:
- Implemented methods to evaluate expression. The method takes as input a map from non-function variables and evaluates the syntax tree as needed.
- Redesigned and wrote the code for boundary functions. Initially a boundary was also defined through some awkward combination of functions and tuples which depended heavily on the order in which the functions we're passed in.

  A rectangular boundary is now composed of multiple `BFunction`'s (Boundary Functions). These would look like

  ```
  new BFunction(fixVar(x, 0), From(0, 10), condition(u, 0))
  ```

in math this would look like

$$u(0,t) = 0 \qquad\qquad 0 < t < 10$$

- Implemented the code to split up an expression into it's derivative terms and then convert it to a pde.

July 8th:

- Started implementing a data type for values with error associated with them. I decided to read up more about similar implementations, but I forgot about them while working on other parts of the code.
- Reimplemented the code for solving elliptical equations. Most of it was exactly the same as done in the first week. For now what's returned is a tuple of 2 2D arrays of doubles. One for the solution and one for the error associated with each point.
- Tested the solver with Laplace's equation with fairly simple boundary conditions. Solving for Laplace's equation went well with no notable problems.
- The boundary assertions had to be loosened for approximately equals due to truncation differences between doubles.

July 15th:

- For clarity sake, the constructor for a pde no longer takes in multiple expressions. The constructor was changed to accept a `Map[Function, Expr]`. A `Function` is either a `FunctionVariable`, `d`, `dd` or `noOrder`. Where `noOrder` is a `Function` object with the sole purpose of identifying that part of the pde. In

  $$\alpha u_{xx} + \beta u_{tt} + \gamma u_{xt} + \zeta u_x + \nu u_t + \xi u + f(x,t) = 0$$

  $f(x,t)$ would be `noOrder`.
- Implemented a solver for general bivariate pdes. Only the elliptical case has been covered and various code surrounding validity intervals are still in the works. If the function only has valid constant coefficients then it works right now.

July 22th:

- Implemented a solver for first order PDEs. Tested with a fairly simple example, but it works fine. The error code still has to be worked in since it is just a rework of Laplace's error right now.
- Continued work on the error value data type and the solution. The class should work such that.

```
val point56 = solution(5,6)
// point56 = 1.2 +/- 0.4
val noErrorAdd = point56 + 2.0
// 3.2 +/- 0.4
solution(-1000, -1000)
// throws out of bounds exception
```

  So the solution is stored in a data type where one can retrieve the value of all points within bounds. This value is returned in a new data type (`errorVal`). `errorVal` interacts with `Doubles` by while keeping the same error. `errorVal` interacts with others of the same type by adding the errors appropriately.

July 29th
- Wrote this report.
- Started a cleanup of the code. Lots of the ideas we're added on-the-fly to test an idea. Unfortunately the ideas which were kept were left in the code in their experimental forms.

2 TODO

More Testing: Most of the examples I've tested on we're fairly simple ones. Some more complicated examples should probably be tested on. This would also include writting a propper unit testing section instead of just printing to the command line.

`errorVal`: This data type still has to be properly implemented. As well as the container for the solution.

Errors: The

Speed Testing: I haven't really tested much in terms of speed. I've messed arround with the step sizes and max iterations, but that was mostly so that I could get an idea of what's happening in a reasonable time.

Comments: Comments should probably be sprinkled here and there to explain some of the code.