The Structure of the Finite Difference Template Library

Earl Waylon Flinn

November 16, 2007

1 Introduction

The Finite Difference Template Library (FDTL) is created for the purposes of quickly solving partial differential equations (currently only elliptic ones on rectangular grids) using the finite difference method. It is implemented in c++ using both the object oriented (OO) paradigm and generic programming techniques. The project was initially begun to solve a specific type of equation found in low temperature quantum mechanics (a Gross-Pitaevskii equation, a form of Schrödinger's equation), but was written in such a way as to allow the solution of a wide range of partial differential equations. It was written to enable those with only an introductory knowledge of the finite difference method (and not necessarily any knowledge of solution methods) to expeditiously create efficient software for solving an equation. It is meant, therefore, to be fast in the sense that the time required for theoretical preparation, development and execution are all small.

This document is meant to describe the structure of the project in both the OO and generic senses and the way in which these complementary structures relate. It is meant to do so in a way that will allow both users and implementors to gain an understanding adequate for thier tasks.

Throughout this document I employ terminology which is standard in the object oriented and generic disciplines (such as object, inheritance, concept and refinement). In addition to these I also use the term 'category' to refer a set of related concepts.[1]

1.1 Descritization

We use the following conventions throughout this document regarding application of the finite difference method. We assume the form of the finite differenced equation is as follows: [2]

$$a_{i,j}u_{i+1,j} + b_{i,j}u_{i-1,j} + c_{i,j}u_{i,j+1}d_{i,j}u_{i,j-1} + e_{i,j}u_{i,j} = f_{i,j} \tag{1}$$

[1] See Appendix A for further discussion of my use of the term 'category'.

[2] Assuming standard difference operators, most 2D second order equations can be written in this form.

where the subscripts denote the grid point at which the value of the function or coefficient is taken. Coefficents may be constant or (more likely) have a functional dependance on the grid point.

Descritization of variables is done using the following relations

$$x = x_0 + i\Delta x \tag{2}$$

$$y = y_0 + j\Delta y \tag{3}$$

where x_0 and y_0 are the lower bounds on the x and y boundaries, respectively, and Δx and Δy are the grid spacings in x and y, respectively.

2 Structure Overview

Both the object structure and concept structure of this library can be seen as having three basic parts. These are: *problem*, *solver*, and *goal*. Below is a description of each and of the relations between them.

2.1 Descriptions

Problem A *problem* represents a finite-differenced partial differential equation (PDE) *and* a tentative solution to it.

Solver A *solver* represents a method of solving a correctly and fully defined *problem*.

Goal A *goal* is a predicate, or test, which may be used to determine whether the tentative solution contained in a *problem* is satisfactory.

2.2 Relations

From the above we can see that the *problem* is more object-like and the *solver* and *goal* are more algorithm-like (though they are all treated like objects by c++). It follows then that the later two are likely to be things which are *applied to* the former. The following descriptions cover this relationship in more detail.

Solver → Problem ← Goal A *solver* is applied to a fully defined *problem* until that *problem*'s solution satisfies the *goal*.

Goal → Problem A *goal* tests a *problem* (and through it, it's tentative solution).

Solver → Goal → Problem A *solver* uses a *goal* to determine when a *problem* is solved.

3 Object Stucture

4 Concepts

Throughout this section (and the next) I use the ← symbol to show refinement, and the ◁ symbol to show that a type is a model of a concept.

Below are syntactic descriptions of each of the major concepts.

A ← B B is a refinement of A

B ◁ b b is a model of B

4.1 Solution

4.1.1 Static Solution

Member	Description
int I()	returns the number of grid points in the first coord.
int J()	returns the number of grid points in the second coord.
double dx()	returns the grid spacing in the first coord.
double dy()	returns the grid spacing in the second coord.
double x0()	returns the minimum in the range for the first coord.
double y0()	returns the minimum in the range for the second coord.
double at(int i, int j)	returns the value at the grid point (i,j)

4.1.2 Static Solution ← Mutable Solution

Member	Description
double& u(int i, int j)	returns a reference to the value at the grid point (i,j)

4.1.3 Mutable Solution ← Problem

Member	Description
double a(int i, int j)	Returns the the value of the first coefficient (a) at the grid point (i,j)
double b(int i, int j)	Returns the the value of the second coefficient (b) at the grid point (i,j)
double c(int i, int j)	Returns the the value of the third coefficient (c) at the grid point (i,j)
double d(int i, int j)	Returns the the value of the fourth coefficient (d) at the grid point (i,j)
double e(int i, int j)	Returns the the value of the fifth coefficient (e) at the grid point (i,j)
double f(int i, int j)	Returns the the value of the source term (f) at the grid point (i,j)

4.2 Solver

4.2.1 Unary Function ← Solver

Member	Description
int operator()(const Problem& p)	Solve the finite differenced Problem p
int solve(const Problem& p)	Solve the finite differenced Problem p

4.3 Goal

4.3.1 Unary Function ← Goal

This is a purely semantic refinement.

4.4 Prolongation Operator

4.4.1 Binary Function ← Prolongation Operator

This is a purely semantic refinement.

4.5 Restriction Operator

4.5.1 Binary Function ← Restriction Operator

This is a purely semantic refinement.

4.6 Integrator

4.6.1 Unary Function ← Integrator

This is a purely semantic refinement.

4.7 Transform

4.7.1 Static Solution ← Transform

This is a purely semantic refinement.

5 Models

Solver ◁ gauss_seidel
Solver ◁ successive_overrelaxation
Solver ◁ multigrid

Goal ◁ residual_norm
Goal ◁ solution_norm

Problem ◁ laplace
Problem ◁ simple_harmonic_oscillator

Problem ◁ gross_pitaevskii

Restriction Operator ◁ half_weighting
Prolongation Operator ◁ bilinear_interpolation

Transform ◁ gross_pitaevskii_energy

6 Appendix A. Categories

In addition to the standard terminology from the OO and generic schools I use the term *category* to refer to a related set of concepts. More formally, a *category* is any (disjoint) set of refinement hierarchies which link semantically related concepts. A category naturally contains all the concepts in the hierarchies which make it up. I also sometimes use the name of the category to describe one of it's constituents. This may seem a bit confusing at first but is actually quite a natural (and common) linguistic device.

As an example consider the term 'Iterator' as used in the Silicon Graphics Inc. (SGI) documentation of the Standard Template Library (STL). 'Iterator', as used there, does not refer to an actual concept in the STL, it instead refers to a set of six related concepts. It is consistent with both the usage in the SGI documentation and the usage in this document to create from these six concepts, and thier refinement hierarchy, a *category* and to then name this category *Iterator*. This category would then include all concepts in the refinement hierarchy rooted at *Trivial Iterator* and *Output Iterator* and extending to the concept *Random Access Iterator*, which is not further refined in the STL. I believe that doing so clarifies the role of the term 'Iterator' and makes it's subsequent use more natural.

This example also serves to illustrate the two major uses of a category's name. 'Iterator' might be used, on the one hand, to refer to the category itself, and thus to the entire set of concepts it contains, as in: "All the concepts in *Iterator* except *Output Iterator* support dereferencing." It might also be used to refer to any of the individual concepts in this category, as in: "Despite this fact *Output Iterator* is still an *Iterator*."