A C++ Differential Equations Solver Using Object-Oriented Numeric

Jean Belanger[1]

[1]**Jean Bélanger** is an industrial physicist who has expertise in scientific computing and physical modeling. He can be contacted at **belanger@elligno.com.**

Contents

Abstract **4**

Introduction **5**

1 Motivation for the Object-Oriented Approach **7**

1.1 The Problem of Change 7

1.2 Problems with Conventional Approach 8

1.3 Advantages with Object-Oriented Techniques 9

1.4 The Need for Abstractions 10

1.5 Lay Out of This Report 10

2 The ODE Solving System **11**

2.1 Physical System 11

2.1.1 Conservation Law System 11

2.1.2 Numerical Flux Algorithm 11

2.2 Numerical Method 12

2.2.1 Grid 13

2.2.2 Spatial Discretization 13

2.2.3 Temporal Discretization 14

3 Object-Oriented Design and Analysis of Finite Difference Method **15**

3.1 Requirements 15

3.2 Key Concepts 16

3.2.1 Discretization Concept 16

3.2.2 Field Concept 17

3.2.3 Spatial Operator Concept 17

3.2.4 Time Evolution Concept 17

3.2.5 Numerical Method Concept 17

3.3 Key Classes 17

3.3.1 Explicit Integrator class 18

3.3.2 ODESolver class . . . 18
3.3.3 Physical Algorithm class . . . 18
3.4 Scenario . . . 19
3.4.1 Solve a predefined problem with a predefined method . . . 20

4 Numerical Package **21**
4.1 Finite-Difference Framework . . . 21
4.2 Implementation in C++ . . . 21
4.2.1 An Application: One-Dimensional Frictionless Dam-Break Problem . . . 22
4.2.2 Example: Main Program . . . 22
4.3 Discussion . . . 22

Conclusion And Future Work **23**

Terminology **24**

A Data Representation Model Notation **25**
A.1 Association Relationship and Multiplicity . . . 25
A.2 Relationship Diagrams . . . 25

B Brief Introduction To Scientific Computing **27**
B.1 What is Scientific Computing? . . . 27
B.2 Problems from Science and Engineering . . . 27
B.3 Grid and Field Lattice . . . 28
B.4 Time Stepping . . . 28

C Some Details About Finite Difference Method **29**

List of Figures

1.1 Wave Simulator Use Case Diagram . . . 9

3.1 Framework Class diagram . . . 18

3.2 Key classes in cooperation . . . 19

A.1 Aggregation Association . . . 26

A.2 Binary Association . . . 26

C.1 One-Dimensional Mesh . . . 30

Abstract

Over the last few years we have been migrating a small library of numerical code originally written in C to C++. In this report, we present the mathematical abstractions used and how object-oriented programming techniques are applied for scientific software design. Finally implementations details are provided including relationship between data structure. The result is tight, readable code that is easy to maintain and extend. Example with Shallow water equations is drawn from our prototype C++ based environment.

Key words: Object-Oriented, scientific computing, physical modeling, partial differential equations

Introduction

Time-dependent partial differential equations, PDEs, model many important processes in different field of science and engineering (e.g motion of particles, changes in animal populations, and the movement of resources in a financial market). In order to solve these equation on a computer, we need to develop a program that efficiently performs the necessary calculations. In this report we focuss on the System of Conservation Laws used in Computational Fluid Dynamics (CFD). Recently, we have started the migration of our CFD numerical library. Our solution has been to use object-oriented design principles and languages like C++ to write the code. All interesting programs will have to evolve, and it is important that necessary modifications can be made easily. Because we regard these aspects as essential, we have followed an object-oriented methodology in order to develop the design. Compared with traditional design methodologies, OO methodologies yields programs that are easier to change.

The building block of OOP support a step-wise software development from simple problems to more complicated problems by identifying concepts of the problem at hand. The task is to develop computational "objects" that represent fundamental abstractions of elements in a computational model. The key to this goal is the reuse in other applications of already tested components, which is a fundamental issue in OO. This result is an implementation that is manageable, extensible, and easily modified.

Then what is OO software? The most important concept is the *object*, which represents a relevant abstraction either from the real world or from the implementation domain. The conceptual similarity between objects in the software model and real-world objects is a key to explain the success of OO software construction. This is enhanced by program constructs which model relationship between objects, for instance that one object is an *aggregate* [1] of several other objects.

There are four fundamental elements contained in the concept of an object-oriented model. These are abstraction, encapsulation, modularity and hierarchy. These are abstraction, encapsulation, modularity and hierarchy.

To achieve this, OO uses three main themes: encapsulation, inheritance and polymorphism. The former, means that software is organized into objects that store both data and operations on data. By encapsulating the data and operations together isolates the classes and promotes the reuse of code. It prevents a program from being so interdependent that a small change has a massive ripple effects. Changes to a class affect only the class in question. It allows the details of the implementation of the object to be hidden, and thus easily modified. Code reuse is further enhanced by inheritance, that consists of putting abstractions in a hierarchy share behavior through attributes and operators that are common to several subclasses into superclasses, which is implemented once for all. Interdependencies between the classes are explicitly laid out in the class interfaces. Finally, polymorphism allows the same operation to behave differently in different classes and thus allows objects of one class to be used in place of those of another related class.

Object-Oriented programming is an appropriate tool in the context of numerics. The logical entities are often readily available in the form of mathematical abstractions. Thus, large parts of the design are already inherent in the underlying mathematical structures.

[1] see Terminology section for the definition

In this report, we present an Object-Oriented approach to the construction of ODE solvers. In particular, we stress the program's ability to adapt to change *i.e.* the ease with which the program can be modified and demonstrate the benefits of such an approach. First we present an overview of the content of the libraries. Thereafter, some of the classes related to finite difference programming are discussed in more detail with an emphasis on OOP. The code uses inheritance to group similar data structures and procedures into families that share some elements and that are, in appropriate contexts, interchangeable. A concrete example is given to demonstrate the principles for a simple model problem, Finally, we discuss how OOP can be utilized to structure simulators and their mutual dependencies. It is assumed that the reader is familiar with C++ and has an understanding of the basic concepts in object-oriented programming.

Chapter 1

Motivation for the Object-Oriented Approach

In this chapter, we motivate the OO approach that we use. In the first section, we give example of a traditionally designed program where small changes of the requirements may lead to large amount of changes in the code. In the second section, we briefly discuss some concepts of OO technology. In many respects, OO programming languages yield programs that have better properties than traditionally designed programs, for instance with respect to code reuse and understandability. However, to use an OO programming language does not guarantee an object-oriented program. Therefore, we will also introduce OO methodologies.

1.1 The Problem of Change

Everything evolves, and a PDE solver is no exception. In this section, we state some common reasons for this, and we strive to understand why this implies a problem for traditionally constructed PDE solvers.

The system we aim at shall be able to solve a PDE problem. We want to describe the problem in such a way that a computer can solve it (approximately, since exact mathematical solutions may not be obtained) and present the solution. We want this system to be flexible, especially with regard to the following features:

- Change of the PDE problem we solve.

  Change of the problem may be motivated by a new geometry, other mathematical models of the same physical problem, or maybe a completely new problem etc... For example, we want to test the numerical scheme (validate) with Burger's equation, simpler model which contains many difficulties (convective term),

- Change of the numerical method we use

  We may have several different reasons for wanting to change the numerical method. Maybe, the numerical method yields wrong answers for the particular problem. This includes unstable solutions, solutions that are mathematically correct but physically incorrect, or solutions that are not accurate enough. For example, in an industrial project as we add in hydraulic, it's very difficult to know at the beginning which numerical scheme shall be used. We have to test different solutions, it needs a great deal of flexibility. Another reason for changing the method is efficiency.

- We require that the system shall be easily extended. It is not enough if we can solve a predefined set of problems with a predefined set of numerical methods. A user shall be able to describe new problems, or new solution methods, and add these to the system. This is particularly useful if, for instance, the user develops a new numerical method and wants to compare this method against other methods.

To summarize, we have several motivations why a PDE solver shall be designed in such a way that it easy to modify. We proceed with an examination on why this issue is difficult, when traditional programming methodologies are used.

Strongly simplified, a typical PDE problem may be solved with the algorithm below. (For readers unfamiliar with the subject of scientific programming, we have summarized some concepts of PDE problems ad how to solve them numerically in Appendix A).

Initiate $u^{(0)}$ *according to initial conditions*
Loop from time $t = 0$ *to* T *step* dt

- *Calculate the space derivative*
- *Calculate* $u^{(t+dt)}$ *in the interior*
- *Calculate* $u^{(t+dt)}$ *at the boundaries*

end loop

Here, $u^{(t)}$ denotes the numerical solution at time t. Step 1) depend on the chosen numerical method for approximating space derivatives. Step 2) depends on the result of step 1) and the chosen numerical method for approximating time derivatives as well as the PDE problem coefficients (which may be space, time, and solution dependent). Step 3) depends on the boundary conditions from the PDE problem, and from the numerical approximation of them.

So far, the dependencies are quite simple, but the situation is in fact more complicated. For instance, if we have periodical boundary conditions along some dimension, this will influence not only step 3) but also step 1) and step 2). If we want to solve the problem more accurately, we can use a higher-order method for approximating the space derivatives. This may influence the algorithm for calculating the solution u at the boundaries.

The conclusion is therefore that programs whose design has focussed on the algorithm, as traditional design techniques does, tend to be difficult to modify.

1.2 Problems with Conventional Approach

Within the framework of conventional programming a program is usually constructed by putting the emphasis on the computational "procedure" rather than the "data structure concerned with computation". It is difficult to modify the existing codes and to extend the codes to adapt them to new uses, models, and solution procedures. It require a high degree of knowledge of the entire program is required to work on even a minor portion of the code. Furthermore the reuse of code is difficult because numerous interdependencies between the components of the design are hidden and difficult to establish. A key element in numerical simulation is the selection of numerical algorithm for the problem at hand.

The PDEs that are to be approximated can be quite complex (non-linear). The difference approximations that are used can vary from relatively simple (e.g. second-order finite difference methods) to quite complex (e.g unsplit Godunov procedures for incompressible flow), or fully fourth-order finite difference methods. Also, it is often necessary to change a model or algorithm of a program for the sake of correction or improvements during the development or execution of a numerical simulation. The net result of the data structures, advanced algorithms, and modern architectures is a PDE solver code that is an extremely complex systems.

Typical numerical element programs consist of several hundred of lines of procedural code, written in C. It is difficult to modify the existing codes and to extend the codes to adapt it to new uses, models, and solution procedures. The inflexibility is demonstrated in several ways:

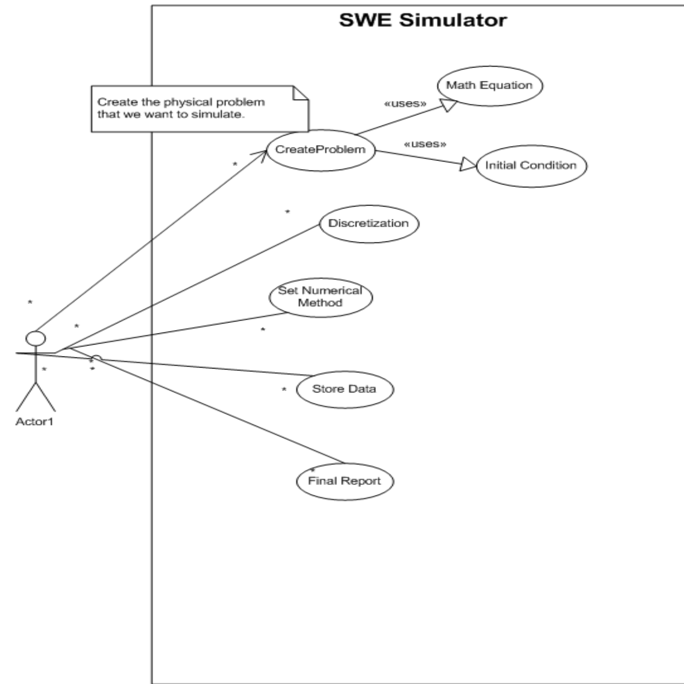

Figure 1.1: Wave Simulator Use Case Diagram

1. a high degree of knowledge of the entire program is required to work on even a minor portion of the code;
2. reuse of the code is difficult;
3. a small change in the data structures can ripple throughout the system;
4. the numerous interdependencies between the components of teh design are hidden and difficult to establish;

Industrial applications

Experience gained from our recent projects in the hydraulic industry, open-channel flow simulation (see Belanger and *al.*), validation of numerical schemes, we find very difficult to modify or adapt our program to change. Many situations we would like to experiment with new schemes, change various boundary condition, modify part of the algorithm, etc... It requires lot of flexibility. Our C program could not handle such a rapid change. The reasons are the following:

- rapid prototyping

1.3 Advantages with Object-Oriented Techniques

Object-Oriented is not only concern with the language like C++, but also with technique such as the UML process. Truly Object-oriented means the use of this process. Briefly, UML is a way (series of step to design the software). Consist of Use Case, requirements, class diagram and sequence diagram. Capture the most important features of the system. Too many people start coding before they setup what your program should do and how it's going to do it. For more details about UML, the reader is referred to the bibliography (see ...). Represent graphically the abstractions and their relationship. Enhance the code maintenance, because

UML diagram show you the global picture of the program, you don't have to know the detail of it, but how component interact and the effect of change on one could affect others. You can see it as part of the documentation. This is the main advantage of using good practice. From the author point of view,

The typical process of constructing a computational model consists of the following steps:

- Create and/or modify a discretized domain
- Create and/or modify initial conditions and/or boundary conditions
- Compute numerical approximation to the governing equations
- Visualize and/or analyze the results using a separate visualization package

This is represented in the Use Case Diagram above. This kind of diagram capture the essential of the problem.

1.4 The Need for Abstractions

We refer to the primary unknown u as scalar fields. These field are defined over domains, which in the software appear in a discretized form called *grid*. Moreover, the PDE contains spatial and temporal operators as well as associated boundary and initial conditions. OOP makes it possible to programs in terms of such mathematical abstractions instead of directly manipulating primitive array structures. Of course, operations on abstract quantities must be realized as array operations in "do-loops" in the complied code to obtain maximum efficiency, but the programmer is not involved with such low-level code.

The obvious attempt to apply OOP to PDE solvers is to create ADTs for the common mathematical and numerical quantities like fields, grids, operators, boundary conditions, linear systems etc.

1.5 Lay Out of This Report

The layout of this report is as follow: in the chapter we point out the problems related with conventional programming, specifically changing the code to adapt to new requirements. We also do a short review on existing projects using OO technology. In the second chapter

Chapter 2

The ODE Solving System

2.1 Physical System

In this section we present the physical problem that we want to solve.

2.1.1 Conservation Law System

In this section we present briefly the mathematical equations that we are solving. The equations that model 1-D open-channel flow (Saint-Venant equations), in a wide rectangular channel, can be written in a conservative or divergent form along the direction of movement

$$\mathbf{U}_t + \mathbf{F}_x + \mathbf{S} = 0 \tag{2.1}$$

where the vector $\mathbf{U}$ is the state variables vector, $\mathbf{F}$ flux of the state variables, $\mathbf{S}$ represents sinks and sources of the momentum arising from the bed slope and friction losses. They can be expressed in terms of flow variables as

$$\mathbf{U} = \begin{pmatrix} h \\ hv \end{pmatrix} \qquad \mathbf{F} = \begin{pmatrix} hv \\ hv^2 + gh^2/2 \end{pmatrix} \qquad \mathbf{S} = \begin{pmatrix} 0 \\ gh(S_{0x} - S_{fx}) \end{pmatrix}$$

where g is the gravitational acceleration, h is the water depth of flow, $Q = vh$ the discharge, $S_0 = -z_x$ is the bed slope expressed in term of the spatial derivative of the bottom elevation z, the friction lost S_{fx} can be evaluated from the Manning formula given by

$$S_{fx} = \frac{N^2\, v\,|v|}{h^{4/3}}$$

where N is the Manning coefficient.

2.1.2 Numerical Flux Algorithm

In order that the flux discretization be compatible with the bed slope source term Nujic's solution comprises first a splitting of the flux terms in a convective and a pressure contributions and then perform appropriate discretization. The splitting of the momentum flux is performed as $\mathbf{F} = f_c + f_p$

$$\mathbf{F} = \begin{pmatrix} hv \\ hv^2 + gh^2/2 \end{pmatrix} = \begin{pmatrix} hv \\ hv^2 \end{pmatrix} + \begin{pmatrix} 0 \\ gh^2/2 \end{pmatrix} \tag{2.2}$$

1. **Convective Term**

   is discretized by the scheme ... The intercell flux, $f_{j+1/2}$ is evaluated by solving the approximate Riemann problem which involves only two shocks, although two rarefaction fans could be considered, separating three states; left-right, and intermediate state. For the Shallow-Water equations Toro(1992) presented a suitable HLL-type flux based on the suggested approximations of Harten. The intercell flux is chosen from

$$F^{HLL}(U^L, U^R) = \begin{cases} F^L & \text{if} \quad 0 \leq S_L \\ F^* & \text{if} \quad S_L \leq 0 \leq S_R \\ F^R & \text{if} \quad S_R \geq 0 \end{cases}$$

   For the shallow-water wave equations, the intermediate states u^* and h^* are given by,

$$\sqrt{gh^*} = \frac{1}{2}(\sqrt{gh_L} + \sqrt{gh_R}) - \frac{1}{4}(u_R - u_L) \tag{2.3}$$

$$u^* = \frac{1}{2}(u_L + u_R) + \sqrt{gh_L} - \sqrt{gh_R} \tag{2.4}$$

   and are used to estimate the shock speeds.

$$\begin{aligned} S_L &= \min\{u_L - \sqrt{gh_L}, u^* - \sqrt{gh^*}\} \\ S_R &= \min\{u_R + \sqrt{gh_R}, u^* + \sqrt{gh^*}\} \end{aligned} \tag{2.5}$$

   which are estimates of the largest and smallest propagation speed. These wave speeds are used to solve for the flux in the intermediate state

$$\mathbf{F}^* = \frac{S_R F_L - S_L F_R + S_L S_R (U_R - U_L)}{S_R - S_L} \tag{2.6}$$

   by satisfying the integral form of the conservation law over a control volume.

2. **Pressure Term**

   is treated centrally leading to the following expression for the corresponding numerical flux at the cell interface $j + 1/2$

$$\left(\frac{gh^2}{2}\right)_{j+1/2} = \frac{g}{2}\,(h_j^2 + h_{j+1}^2) \tag{2.7}$$

3. **Bed Slope Term**

   is computed normally at the centre of the corresponding cell, but with a spread of the slope discretization

$$\left(gh\frac{\partial z}{\partial x}\right)_j = \frac{g}{2}(h_{j+1} + h_{j-1})\left(\frac{z_{j+1} + z_{j-1}}{2\,\Delta x}\right) \tag{2.8}$$

   where Δx represents the grid spacing. This treatment solves the water at rest problem and improves the robustness of the computation.

2.2 Numerical Method

To solve this system of equations we need to use some approximation method called numerical method. They are many numerical method available but we concentrate on on a semi-discrete method often called "Method-of-Lines".

Here the details of the high order shock fitting algorithm are presented. A point-wise method of lines approach is used. This method simplifies the required coding, allows separate temporal and spatial discretizations, and also allows for the incorporation of the source terms. With the time discretized this way,

it is natural to calculate approximations to the space derivatives first, and thereafter the time derivatives. This technique-called method of lines (MOL)-is often useful, because it separates the algorithms in two parts, this making it possible to improve the numerical method in the part of the algorithm where it is needed the most. Also, from the perspective of the time derivative approximation, we may view the time discretization as an ODE (ordinary differential equation) instead of a PDE, and theory from this field of scientific computing may be used.

In the following sections, the computational grid will be defined, the Harten-Lax-Levy spatial discretization scheme will be outlined, and the temporally second-order Runge-Kutta scheme for time discretization will be given.

2.2.1 Grid

Written in vector notation, Eqs. (??) take on the form

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial}{\partial x}\mathbf{f}(\mathbf{u}) = \mathbf{s}(\mathbf{u}), \qquad x \in (0, L) \tag{2.9}$$

Here the $\mathbf{u}$ is used to denote the set of conserved dependent variables,

$$\mathbf{u} = \Big(A, Q, H\Big)^T$$

Stricly speaking, H is not conserved, but evolves due to the source term. The vector $\mathbf{f}$ is a set of fluxes of each conserved quantity, and $\mathbf{s}$ is a source. A uniform Cartesian grid is used to discretize the domain $\Omega = \{(x,t) \in \mathbf{R}^2 \,|\, 0 < x < 1,\, 0 < t < T\}$ with N equally spaced nodes. Next, one allows the semi-discretization $\mathbf{u}(x,t) \to \mathbf{u}_i(t)$ and the full discretization $\mathbf{u}_i(t) \to \mathbf{u}_i^n$, where the semi-discretized and fully discretized numerical solution vectors are denoted by $\mathbf{u}_i$ and $\mathbf{u}_i^n$, respectively. Here, i is the spatial node number corresponding to the location $x_i = x_{min} + i\,\Delta x$, where $\Delta x = x_{max}/N$, and n is the time level corresponding to $t_n = \sum_{m=1}^{n} \Delta t_m$, where Δt_m is the time step for each integration step.

We would like to solve by the upwind finite difference scheme for the domain The numerical algorithm then takes the following form (compute u_i^k from the explicit scheme)

$$u_i^k = u_i^{k-1} - \frac{\Delta t}{\Delta x}\Big(f(u_i^{k-1}) - f(u_{i-1}^{k-1})\Big)$$

for $i = 2, \ldots, n$ and $k = 1, \ldots$

2.2.2 Spatial Discretization

Following spatial discretization, Eq. (??) can be approximated as as system of Ordinary Differential Equations in t:

$$\frac{d\mathbf{u}_i}{dt} = \mathbf{L}(\mathbf{u}_i), \tag{2.10}$$

where the operator $\mathbf{L}$ is a discrete approximation to the continuous convection and source operators of Eq. (??):

$$\mathbf{L}(\mathbf{u}_i) \approx \Big(\frac{\partial}{\partial x}\mathbf{f}(\mathbf{u}) - \mathbf{s}(\mathbf{u})\Big)\Big|_{x=x_i}$$

Next, the definition of $\mathbf{L}$ for various spatial nodes are defined.

1. Nodes $i > N$

   For nodes for which $i > N$, which are necessary for calculation of some fluxes, a zero gradient condition is enforced. The backward characteristic emanating from this boundary was calculated.

2. Nodes $2 \leq i \leq N-1$

   Nodes at the interior of the domain. For $\mathbf{L}(\mathbf{u}_i)$ the HLL scheme with a local Riemann solver is used. This scheme is conservative flux difference ?? method, which has been shown to be stable, and yields the proper .... The operator $\mathbf{L}(\mathbf{u}_i)$ is given by:

$$\mathbf{L}(\mathbf{u}_i) = \frac{\hat{\mathbf{f}}_{j+1/2} - \hat{\mathbf{f}}_{j-1/2}}{\Delta\, x} + \mathbf{s}(\mathbf{u}_i)$$

   where $\hat{\mathbf{f}}_{j+1/2}$ and $\hat{\mathbf{f}}_{j-1/2}$ are numerical approximations to the flux function, $\mathbf{f}(\mathbf{u})$, and $\mathbf{s}(\mathbf{u})_i$ is a simple evaluation of the source terms at node x_i.

   In particular, for the local Lax-Friedrichs scheme, on takes

$$\hat{\mathbf{f}}_{j+1/2} = \hat{\mathbf{f}}^{+}_{j+1/2} + \hat{\mathbf{f}}^{-}_{j+1/2} \tag{2.11}$$

   where for each component, $f^{\pm}$, of the vector $\hat{\mathbf{f}}^{\pm}$ one has

$$\hat{\mathbf{f}}^{+}_{j+1/2} = \Xi(f^{+}_{i+1}, f^{+}_{i}, f^{+}_{i-1}) \tag{2.12}$$

$$\hat{\mathbf{f}}^{-}_{j+1/2} = \Xi(f^{-}_{i+1}, f^{-}_{i}, f^{-}_{i-1}) \tag{2.13}$$

   and

$$f^{+}_{i} = \frac{1}{2}\left(f(\mathbf{u}_i) + \alpha\, \mathbf{u}_i\right) \tag{2.14}$$

$$f^{-}_{i} = \frac{1}{2}\left(f(\mathbf{u}_i) - \alpha\, \mathbf{u}_i\right) \tag{2.15}$$

   where

$$\alpha = \max(|f'(\mathbf{u}_i)|, |f'(\mathbf{u}_{i+1})|) \tag{2.16}$$

   The interpolating function $\Xi()$ is defined next.

2.2.3 Temporal Discretization

With the discretization operator $\mathbf{L}$ now defined, Eqs. (2.10) could be solved by a wide variety of standard numerical techniques, explicit or implicit, which have been developed over the years for large systems of ordinary differential equations. Here, an explicit two-stage Runge-Kutta scheme with second order temporal accuracy is chosen. Given a solution $(\mathbf{u}^n_i)$ at t_{n+1} is constructed in the following manner

$$\bar{\mathbf{u}}^1_i = \mathbf{u}^n_i \tag{2.17}$$

$$\bar{\mathbf{u}}^j_i = \mathbf{u}^n_i + \Delta\, t_n \sum_{k=1}^{j-1} a_{jk} \tag{2.18}$$

In this problem, for which the effect of the source term has been resolved, it is convection which dictates the time step restriction. All computations performed here have $0.8 < \text{CFL} < 1.5$, where CFL represents the traditional Courant-Friedrichs-Lewy number. The high order of the Runge-Kutta method enables CFL to be slightly greater than unity while maintaining numerical stability. The results were verified to be insensitive to small changes in CFL.

Chapter 3

Object-Oriented Design and Analysis of Finite Difference Method

The design process is often recognized as the most important phase of OOP. Object-Oriented analysis require to determine the concepts or abstractions of the problem at hand. This section is intended to give a short overview over the numerical concept used, as well as over some of the most basic classes needed for it's realization.

Problem Statement To solve a physical problem that can be numerically simulated.

The overall of the software project is to provide a rapid prototyping environment for validating numerical scheme on the Dam-Break problem for the One-Dimensional Shallow-Water equations.

We are solving a physical problem which is represented by mathematical equations, initial and boundary conditions (initial value problem).

3.1 Requirements

You determine exactly what you want your program to do before you begin developing it. If you don't have a good grasp of the requirements, your initial software will have only the most basic features and won't be sufficient to support your simulation. You could find yourself adding various capabilities to the simulator (program) during the course of developing the simulator. Why not get it right from the start? Even if the requirements have been discussed earlier, we state them explicitly in this section.

1. Concept of field lattice shall be available

   For example, a scalar field that holds the state variable (A,Q) and H, defined at every node of the mesh.

2. It shall be possible to change component of the flux algorithm

   Reconstruction procedure of the state variable. We might want some interpolation algorithm, or even to switch to another reconstruction method.

3. It shall be possible to change the flux algorithm;

   We might want to test different techniques: shock-capturing, ENO extrapolation for the convective flux.

4. It shall be possible to set the finite difference operator to evaluate derivative;

Evaluation of the right-hand side terms. We consider the conservative form of the equation, the flux derivative can be evaluated by an upwind scheme, the pressure term by finite difference scheme (central) and the source term point-wise.

5. It shall be possible to change the time integrator;

   Single time stepping. Stability problem, we may have to go to a more stable stepping, for example the two-stage Runge-Kutta.

6. It shall be possible to change the discretization;

7. It shall be possible to set ON/OFF the reconstruction procedure;

8. It shall be possible to set initial condition for the simulation;

   Test the dam-Break problem with different water level and compare ,,,

9. It shall be possible to turn source terms ON/OFF;

   In this example, we present the frictionless case without pressure. For comparison purpose, we may want to test by switching between different configuration.

3.2 Key Concepts

Here we present the mathematical abstractions for the problem at hand and show we can implement ... We consider that these abstraction are the most important of our problem.

3.2.1 Discretization Concept

Numerical approximation consist of transformation from continuum to discrete space. This concept is at the heart of our model (problem that we modelize). This the heart of the problem that we want to approximate. The discretized equations is constructed over each element. Every numerical method consist of this transformation from a continuum space to a discretized space, leading to a set of algebraic equations to solve. The discretization concept is what characterize the ... to be defined

Global discretization domain is defined as a set of all nodes and all elements (Ω = All Nodes, $\sum \Omega^e$)

The main step is the choice of the *discretization method* of the mathematical formulation and involves two components, the *space discretization* and the *equation discretization*. The space discretization consists of setting up a mesh or a grid by which the continuum of space is replaced by a finite number of points where the numerical values of the variables will have to be determined. It is intuitively obvious that the accuracy of a numerical approximation will be directly dependent on the size of the mesh, that is, the better the discretized space approaches the continuum, the better the approximation of the numerical scheme. In other words, the error of the numerical simulation has to to tend to zero when the mesh size tends to zero, and the rapidity of this variation will be characterized by the *order* of the numerical discretization of the equations.

Once the mesh has been defined the equations can be discretized, leading to the transformation of the differential or integral equations to discrete algebraic operations involving the values of the unknowns at the mesh points. The basis of all numerical methods consists of this transformation of the physical equations into an algebraic, linear or non-linear, systems of equations.

The discretization concept can be characterized by three features:

- The domain of the problem is represented by a collection of simple domains, called *cell*;

- Over the cell, the physical process is approximated by functions of desired type (polynomials or otherwise), and an algebraic equations relating physical quantities at selective points, called *nodes*, of the the element are developed,
- the discretized equations is constructed over each cell;

3.2.2 Field Concept

The grid represents the node of the discretization, but an abstraction which represents the physical entities actually represented in the problem is also needed. The grid function serves as such an abstraction, which holds one or more values in every point of the grid (sometimes, this concept is called a field).

3.2.3 Spatial Operator Concept

The right-hand side of the equation. This abstraction is in direct relation with the spatial discretization since this operator is responsible to evaluate the spatial term of the equation. It can be the derivative of the flux, source terms.

Express the spatial discretization, explicit or implicit (see appendix for more details). Numerical representation or algebraic form of the mathematical equations. In the explicit case, we have to solve an algebraic relation relating current time from previous time ($u_i^{n+1} = u_i^n - A(u_i - u_{i-1})$). In the implicit case, we have to solve a linear or a non-linear system ($Ax = b$).

3.2.4 Time Evolution Concept

The grids are used to discretize in space. The normal way to discretize in time is by viewing the grid functions as representing the physical entities on certain discrete time levels. The overall purpose of our program will be to calculate the grid function on the next time level, given the grid function on the current time level and maybe some previous levels.

3.2.5 Numerical Method Concept

It a numerical representation of the mathematical equation based on discretization. This abstraction ... to be completed

3.3 Key Classes

By examining the application domain (key concept which are described in the previous section), we identify some of the key classes of our system.

To begin with, we realize that we need a **PDEProblem** class for the representation of our actual problem that is to be solved. The **PDEProblem** knows the governing equations, the boundary conditions and the initial condition (since the problem we are interested in *are* initial boundary value problems).

Since conservative and non-conservative PDE problems are represented in different ways, the inheritors **Conservative Problem** and **Non Conservative Problem** are introduced. Concrete problems will be heirs to either of these classes.This implies that the system will be extendible with respect to the problems that can be handled.

The **PDEProblem** class shown in figure ???, where class that represent the initial condition and the boundary conditions also are shown (the notation is explained in Appendix B). We could have included the

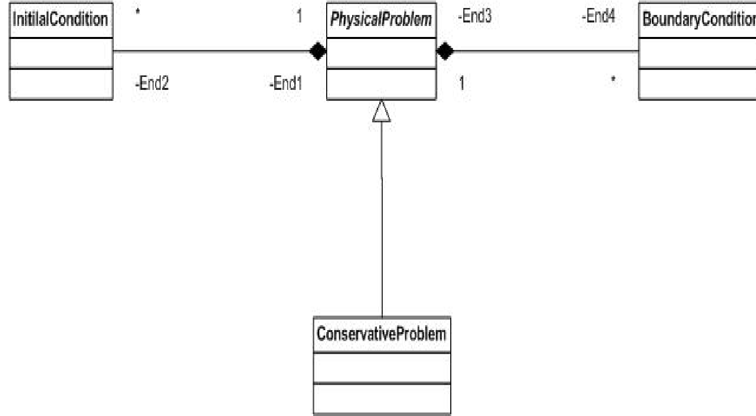

Figure 3.1: Framework Class diagram

Grid as a part of the **PDEProblem** aggregate, but we choose not to do so, emphasizing that the grid belongs to a lower level of abstraction.

Next, we focus on numerical method. Since we aim at a flexible framework, the notion of separating the handling of space and time derivatives is appropriate. We introduce the **SpatialOperator** and **Explicit-Integrator** which are responsible for taking care of spatial and time integration. The numerical method of separating spatial and time is often called MOL(Method-of-Lines). Advantage get high-order accuracy for spatial.

3.3.1 Explicit Integrator class

It consists of a recursive time integrator that advances a set of Data Objects over a time step as well as components that evaluate and assemble the Right Hand Side (RHS), one patch at the time. This integration scheme is only conditionally; that is, only very small time steps lead to a stable solution. An estimation for the critical time step is made by (Numerical Reynold number)

$$dt_{max} = dL/c$$

where c is the maximal wave propagation speed in the medium and dL is the smallest element length of the model.

3.3.2 ODESolver class

In C++ it would be natural to have a class where the input data, like u, u_prev, dx, dt, etc ... are class members and where the user defined functions are virtual functions. Moreover, we include a *scan* function for reading the input data and allocating dynamic arrays (u and u_prev). A specific set of user functions can then be implemented in a subclass for a particular application.

3.3.3 Physical Algorithm class

which is responsible for evaluating the different terms in the equation. At this level we need lot of flexibility. For example, as Nujic pointed out, we split the flux in two parts: convective and pressure terms. These two terms are evaluated separately. For the convective we usually use shock capturing-technic, particularly Riemann Approximate solver. We introduce the class **PhysicalAlgorithm** provide a set methods for implementing .... Inheritance from a well-designed base class makes it possible to write a suite of solvers with a common interface.

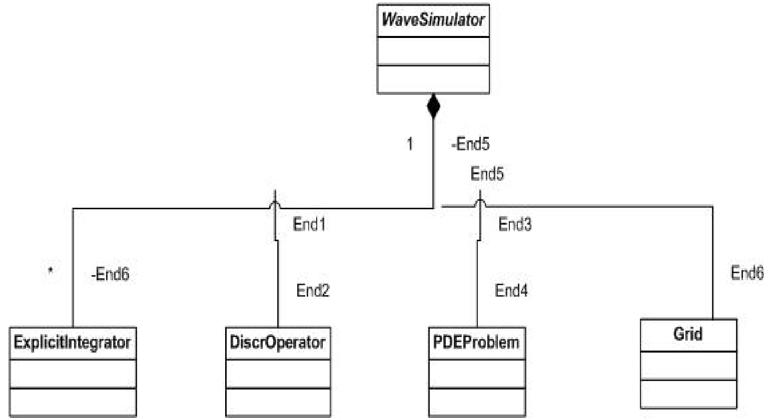

Figure 3.2: Key classes in cooperation

Flux algorithm is an abstraction that represent the Nujic (1995) algorithm for treating the physical flux numerically. The flux is split as pressure term and the convective term. The former we use standard finite difference for the derivative, the latter, responsible for the shock, a shock-capturing technique is used. Belongs to the Riemann problem (Approximate Riemann Solver).

This concept express the way we evaluate the numerical flux at the interface. Sometimes we might want to use a reconstruction procedure to obtain a high-order of the state variable at the interface (actually we perform an interpolation) combined with a flux limiters method (gradient is limited). Reconstruction at the (cell) interface. We refer the reader at the appendix for more about the Finite Volume Method. Briefly the reconstruction process consist of interpolating the state variable or the flux variable at different order.

3.4 Scenario

Having found some immediate key classes from the problem domain, we look through some scenarios in order to find more class to validate the object model. We consider the following scenarios:

1. Solve the frictionless Dam-Break problem with a predefined algorithm (Nujic algorithm), using a Roe solver for the computation of the cell interface flux, a two-stage Runge-Kutta of second order is used for the time-stepping. A reconstruction of type MUSCL for the state variables at the interface is also used to get second-order for the spatial order;

2. Solve frictionless Dam-Break problem, change some component of the algorithm, (HLL solver for the cell interface flux), the overall second order is achieved by a recoonstruction of type MUSCL of the state variables and a two-stage Runge-Kutta;

3. Solve the frictionless Dam-Break problem, use the HLL algorithm for the cell interface flux computation, but no reconstruction procedure is performed for the state variable (first-order). For the time stepping a Euler method of first order is used (first order);

We found that a control class for connecting the objects involved in the scenarios will make their use easier. This class, **WaveSimulator**, will also be responsible for storing intermediate results. The **WaveSimulator** is shown in figure ?, together with its associations to its parts. We choose to model the association as an aggregate, since we regard the **WaveSimulator** as one entity. We observe that a **GridFunction** is mentioned. As explained in section Appendix ?, this is an important concept of the problem domain. However, as with the **Gird**, for the purpose of this report we regard the **GridFunction** as belonging to a lower level of abstraction.

3.4.1 Solve a predefined problem with a predefined method

The first scenario shows the interplay between the components of the system. In order to solve a specific problem with a specific method we basically follow the algorithm sketched in Section ??, but the important difference is the separation of concerns between different parts of the process. Here we examine one iteration of the loop.

1. The **Explicit Integrator** order the **Spatial Operator** to calculate the right hand side of the equations $u_t =$ right hand side
2. The **SpatialOperator** accomplishes this ...

Operations that are discovered through this scenario include

- *ExplicitIntegrator::step()*
  Advances the solution one time step(the whole scenario)
- *SpatialOperaor::applyTo()*
  Computes the right hand side of the equations
- *FluxAlgorithm::calculFF()*
  Computes the numerical flux according to the given algorithm
- *BoundaryCondition::applyBC()*
  Apply the boundary conditions
- *FiniteDifferenceModel::advance()*
  Advances the solution one time step in a specific region
- *ODESolver::solve()*
  Solve the problem for given initial/boundary condition

Below we show the sequence diagram (we express these steps in a sequence diagram).

Chapter 4

Numerical Package

4.1 Scope

For illustration of the methodology, we consider the development of a fluid flow simulator based on the incompressible Shallow-water equations. These equations express the balance between the different forces applied to a mass of fluid on a sloped bed (topography).

4.2 Finite-Difference Framework

The finite-difference framework contains basic building blocks for the numerical solution of a generic differential equation of the form (see equation ??)

$$w_{t|j} + \mathcal{L}(w) = 0 \tag{4.1}$$

where $w = (\ldots, w_j, w_{j+1}, \ldots)$ is the vector of unknowns and $\mathcal{L}$ is the discretization operator *i.e* ... Writing the equation in the above form allows us to implement separately the discretization of the differential operator $\mathcal{L}$ and the time scheme for the evolution of the solution. Once the differential operator $\mathcal{L}$ has been discretized, a number of choices are available for discretizing the time derivative at the left-hand side of the equation. In this framework, such choice is encapsulated in so-called **Evolver**, given $\mathcal{L}$ and the solution w^n at time t^n, yield the solution to the solution u^{n+1} at the next time step. A number of evolvers are currently provided in the library which implement well-known schemes. The problem can be discretized at time t^{n+1} by

$$\mathcal{T}(w, \Delta t) = 0 \tag{4.2}$$

where v^{n+1} is the approximation given by the scheme of the solution w at time t^{n+1}, Δt is the time step and $\mathcal{T}$ is an "abstract" time discretization operator.

For the present example, it means that when the Runge-Kutta scheme is used for time-stepping, boundary condition can be applied at each stage, although the time stepping object does not know about boundary condition. Thus we have succeed in separating concern. Due to this, the Runge-Kutta component is re-usable.

4.2.1 Libraries Package

1. **SfxBase**
2. **CfdCore**

4.3 Implementation in C++

An important design choice is the choice of implementation language. In practice, it was decided early in the project to use C++. Since the analysis uses inheritance and polymorphism intensively, we want OO language, and when efficiency issues are considered, C++ seems to be the most promising alternative. Below we present an example of use of our developing environment (prototyping).

4.3.1 An Application: One-Dimensional Frictionless Dam-Break Problem

On the left boundary $x = 0$ the value is 1 for $t \geq 0$ and the initial condition $t = 0$ has zero as value for $x > 0$. The initial condition reads $u(x, 0) = g(x)$. Let u_i^k be the numerical approximation to $u((i-1)\Delta x, k\ \Delta t)$, where Δx and Δt are the space and time step, respectively, and $i = 1, \ldots, n$ and $k = 1, \ldots, T/\Delta t$.

4.3.2 Example: Main Program

We illustrate how a typical main program is coded in this framework, in order to solve a concrete problem. We choose to solve the shallow-water equation in a conservative form and use a classical Runge-Kutta in time (second-order).

```
int main()
{
   // create a grid of 100 nodes with extent
   Handle<Swe::GridLattice> pGrid( new Swe::GridLattice( 100, 0., 1.));
   // create scalar fields
   Swe::Handle<Swe::FieldLattice> pAfield;
   pAfield.rebind( new Swe::FieldLattice( pGrid, std::string("A")));
   Swe::Handle<Swe::FieldLattice> pQfield;
   pQfield.rebind( new Swe::FieldLattice( pGrid, std::string("Q")));
}
```

FieldLattice a very simple finite difference field class represents the scalar field u, that is, $u(\cdot, t)$. Roughly speaking, it contains a specification of a grid and an array of the point values of the field

Handle We control deallocation of the shared data structure by use of **Handle** which is a smart pointer, template class that offers reference counting and automatically delete X when there are no more references to this object. For field with grids, this is important since several fields may share the same grid. This grid cannot be deleted before all fields have finished their use.

We note that the program consists of three parts:

1. Create objects
2. Combine objects
3. Solve

This style of programming differs remarkably from programs developed with traditional methodologies. Once again, we stress that since the algorithm is decomposed in subalgotihms with different objects are responsible for, it is very easy to change parts of the algorithm. For instance, if we change the one line ... TO BE COMPLETED

4.4 Discussion

Conclusion And Future Work

In this report we studied the design of scientific code by an Object-Oriented Numerics approach. We have demonstrated that it is possible to write a general interface class to a Numerical package. User defined algorithms for a particular application can be implemented in a subclass of the interface class. Hence, different applications have short codes because they can reuse the interface and associated data structures that are required by Numerical package. Moreover, the framework eliminates numerous code writing tasks by allowing the developer to focus on the physical aspects of the simulation and the type of numerical algorithm being implemented.

The application of Object-Oriented Numerics has several advantages. The primary advantage is that it encourages to abstract out the essential immutable qualities which becomes the building block of the API. These objects provide an enforceable interface by which it is possible to extend the application. Once specified, the object interfaces are frozen. The internal class details that are needed to provide the desired interface are invisible to the rest of the program and therefore, those implementation details may be modified without affecting other code. Thus, the design forms a stable base that can be extended with minimum effort to suit new task.

A particular attractive feature is the ability to flexible combine different numerical algorithms and data structures without disturbing the principal mathematical steps in the calling code without uninteresting implementation details.

It was not possible in our procedural code, ... manipulate array data structure through function calls.

What we have gained?

Terminology

In this section we give the basic terminology used in Object-Oriented community.

- **Abstract class** *A class A is called* **Abstract class** *if it is only used as a superclass for other classes.*
- **Aggregation**
- **Attribute**
- **Class**
- **Component**
- **Inheritance**
- **Interface**
- **Polymorphism**

Appendix A

Data Representation Model Notation

Logical Relationships Critical items of information to be captured in the DRM diagrams are the logical relationships between the classes of data. There are three kinds of relationships between classes of data: association, inheritance, and aggregation. Each type of relationship is represented by different notation.

A.1 Association Relationship and Multiplicity

A direct line between two classes denotes the weakness of relationship: association. The following notation indicates that every object of class A is associated with exactly one object in class B, and that every object in class B is associated with exactly one object in class A.

Numeral are used at either end (or both ends) of the association relationship to convey multiplicity of each class. In the above notation, the numerical "1" means "exactly one".

The "0..1" notation is used to denote "zero or more". The following notation indicates that every object in Class A is associated with zero or one object in class B, and that every object in class B is associated with exactly one object in class A.

A.2 Relationship Diagrams

Our diagrams are based on Unified Modeling Language (UML) notation. The UML notation follows the object-oriented methodology for organizing data. The abstraction presented in this section are now presented in the UML language called class diagram. Central to the UML notation is the concept of a class data. A class is an abstract, user-defined description of a type of data. It identifies the attributes of the data and the operations that can be performed on instances (*i.e.* objects) of the data. This modeling technique allows description of as system using the same terminology as the corresponding real-world objects and their associated characteristics. We refer the reader to the appendix for a brief discussion of the UML notation.

Entity relationship diagrams are commonly used to show relationships among classes in the design. A simple Entity Relationship Diagram (ERM) is shown in figure ?. Classes are shown as boxes around the class name. The links between the boxes establish the relationship between classes. Links are labeled with the type of association that exists between classes.

Figure A.1: Aggregation Association

Figure A.2: Binary Association

Appendix B

Brief Introduction To Scientific Computing

The intention of this appendix is to survey the application domain. Since the report is interdisciplinary, some readers may be unfamiliar with scientific computing.

B.1 What is Scientific Computing?

Scientific computing is the original application area of computers and remains one of the most important. The use of physical simulation arising from a wide variety of applications has exploded in the last two decades or so. Applications range from meteorology to plasma physics, environmental protection, nuclear energy, and many other fields (including, increasingly advanced virtual reality applications). All these physical phenomena are best described by mathematical equations, unfortunately, the solution of most equations cannot be expressed as simple, algebraic formulas or do not have analytical solution. We need algorithms that provide techniques for finding approximate solutions to mathematical problems. In this way mathematical models or equations that describe physical behavior can be solved. These algorithms are implemented on computers as programs. In summary, scientific computing is inter-disciplinary field which connects mathematics, physics and computing science. A subfield of scientific computing is numerical analysis, defined as ”the study of algorithms for the problem of continuous mathematics”.

B.2 Problems from Science and Engineering

This section is only meant to review differential equations. A differential equation is a relationship between a variable and its derivative and defines an evolution of continuously varying quantities. The very simplest differential equations (first-order ordinary differential equations) tells you how fast one thing changes when other is changed. Many physical phenomena are described by this kind of equations. Ordinary differential equations (ODE) we describe in this article solve what is known as the “initial-value problem,” which is to find a function $u(t)$ that both satisfies the differential equation and passes through a given initial value of u.

There a multiple physical models of real life problems that can be formulated as a system of conservation laws, in which case u represents a set of variables involved in some fundamental principle of physics. For a flow for example, the system of conservation laws express the conservation of mass, momentum and energy, so the conserved quantities are the density, the momentum per unit mass and the total energy per unit mass. These conservative equations express simply that the change of some quantities inside the volume this physical change of amount in the volume Ω must stem from inward or outward flow through the boundary

of the volume $\partial\Omega$. We have thereby neglected the possibility of sinks or source inside the volume.

We obtain the conservative equation, a first order partial differential equation

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0$$

B.3 Grid and Field Lattice

We discretize space in order to solve a continuous problem numerically. The continuous domain is discretized into points that constitute a grid. A grid may be structured or non-structured. The advantage with non-structures grids is that they are easy to construct on irregular geometries. Our library used structured grids and FD (finite difference) methods. FD methods are based on the definition of the derivative and approximates derivatives at one point with values from neighboring points. Thus, FD methods can use the structure of the grid in order to rapidly find the neighbors.

B.4 Time Stepping

The time integration of the equation can, in principle, be performed using implicit or explicit integration schemes. There are many ways to "step through time". You can for instance, leap through time using *event scheduling* or you can employ small time increments using it time slicing. Using the implicit method, the solution has to be calculated iteratively at every discrete time step. On the other hand, the explicit time integration can be performed without iteration and without solving a system of linear algebraic equations. This integration scheme is only conditionally stable; that is, only very small steps lead to a stable solution. An estimation for the critical time step is made by (Numerical Reynold number: dimensionless number)

$$dt_{\mathrm{max}} = dL/c$$

where c is the maximal wave propagation speed in the medium and dL is the smallest element length of the model.

When advancing the solution in time, explicit or implicit time stepping may be used. Explicit time stepping expresses the solution on the new time level uniquely as a function of data from previous time levels, and is therefore straightforward to implement. The disadvantage is that in order to get stable solutions, very small time steps need to be used. Implicit methods on the other hand express the solution on the new time level as depending on itself as well as the previous time levels. This leads to a linear system of equations, which has to be solved in each step. Thus more work is required in each step, but on the other hand longer time steps are allowed. We concentrate our effort on explicit methods because ...

Appendix C

Some Details About Finite Difference Method

Once we have a set of ODEs describing the physical phenomenon we're trying to model, we have to solve the equations. Unfortunately, the solutions of most differential equations cannot be expressed as simple, algebraic formulas or do not have analytical solutions. So we must numerical integrate them.

To numerically integrate a ODE, we must discretize the continuous equation so that a computer can operate on data in discrete steps. Therefore, algorithms for solving differential equations must increment the time (or equivalent independent variable) in a series of small steps. In other words, the continuous variable t is replaced by a sequence of discrete values t_0, t_1, t_2, and so on. The difference between any two subsequent values of t is the step size Δt. In general, the smaller step size, the more accurate the numerical solution. For each t_i, there is a corresponding value of the dependent variable u_i. In the initial value problem, you are given an initial point (u_0, t_0).

We discretize the domain Ω into partitions with spacing Δx in space and Δt in time. We continue to use the variable u, but from now on, we now think of it as a discrete variable. Superscript indices denote time (t), and subscripts denote space (x), so we have

$$u_i^j = u(i\Delta x, j\Delta t).$$

We solve for the variables at discrete points in space and time. Hence, the derivatives are written as functions of neighboring points. In the present we use a semi-discrete method (also known as the *method of line*) to separate time and space discretization. Because all the quantities are known at time t_n, the scheme is said to be *explicit*.

One approach widely used to approximate the time derivative is the finite difference method. This method involves replacing derivative of () with finite difference quotient forward in time

$$\frac{\partial u}{\partial t} = \frac{(u^{n+1} - u^n)}{\Delta t}$$

, leading to the *Euler recursion formula*. This approximation for $u_t(t)$ is said to be first order in Δt. From a Taylor series expansion of $u(t + \Delta t)$ around point t, we obtain

$$\frac{u(t + \Delta t) - u(t)}{\Delta t} = u_t(t) + \mathcal{O}(\Delta t) \tag{C.1}$$

indicating that the truncation error $\mathcal{O}(\Delta t)$ goes to zero like the first power in Δt. The power of Δt with which this error tends to zero is called *the order of the difference approximation.*

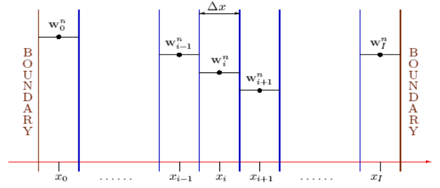

Figure C.1: One-Dimensional Mesh

Spatial discretization is done the same way by approximating the spatial derivative by a Taylor serie expansion. In conservative method, the precision is obtained through the cell interface variables which are approximated by interpolation function. The independent variables are evaluated at nodal position (along the grid) at each time step. The differential system is integrated over each control volume to produce the discrete equivalent of the conservation law

$$U_i^{n+1} = U_i^n - \lambda \left[F_i^* - F_{i-1}^* \right] \tag{C.2}$$

this is an upwind discretization of the flux derivative. Actually the F_i^* represent the interface flux at the interface $x_{i+1/2}$ which can be expressed by $F_{i+1/2}$. This integration technique forms the basis of what is known as the finite volume method. The specific difference between various finite volume schemes are the way in which they approximate the interface convective flux $F_{i+1/2} = F(U(x_{i+1/2}))$.

Bibliography

[1] Ahlander K. *An Object-Oriented Approach To Construct PDE Solvers* Uppsala University, Sweden (1996)

[2] Belanger J. *Validating Shock-Capturing Scheme ...* Technical Report ???, Elligno Inc. (2007)

[3] Belanger J. and *al. Real-Case ...* Technical Report ???, Elligno Inc. (2000)

[4] Hirsch C. *Numerical computation of external and internal flows: vol. 1 Fundamental of numerical discretization* Wiley and Sons, (1990)

[5] Hirsch C. *Numerical computation of external and internal flows: vol. 2 Computation methods for inviscid and viscous flows* Wiley and Sons (1990)

[6] Leveque R. J. *Numerical Methods for Conservation Laws* Lectures in Mathematics Birkhauser (1992)