Computing from LaTeX : automated numerical computing from LaTeX expressions

Silvia Crafa, Fabio Marcuzzi, Marco Virgulin

Department of Mathematics, University of Padova,
Via Trieste 63, 35131 Padova, Italy. E-mail: crafa,marcuzzi@math.unipd.it

June 5, 2014

Abstract

We present CFL (Computing from LaTeX), a software tool that allows the user to automatically perform numerical computation without any programming activity. The user has just to express his computational problem in the usual mathematical language found in written books, and to typeset it using LaTeX for a well-formatted printout.

CFL is designed so that it parses a LaTeX document to recognize well-defined computational problems. An executable Python script is then automatically generated so that mathematical objects are encoded as Python data objects, and the script's logic reflects an appropriate numerical solution pattern for the recognized computational problem. The results of the numerical computations can also be automatically inserted in a resulting LaTeX document so to be displayed by means of tables and graphics.

Therefore, CFL is a numerical problem-solving environment that converts the *specification* of a mathematical problem into an appropriate *resolution pattern* that can be directly executed. The use of high-level mathematical abstractions, the familiar LaTeX typesetting and the automated code generation are key design choices that provide ease-of-use and explorative computation, with a negligible overhead.

1 Introduction

The development of efficient and expressive software capable of solving numerical problems is an important issue in order to support simulation scientists and engineers in modeling complex physical phenomena. Usually, the resulting software turns out to be a complex programming exercise since it requires the coding of various mathematical aspects. To be specific, it involves the description of the problem (equations, parameters and known data in general), its analysis (e.g. residual tests, error indicators, etc.), the numerical solution method and the validation tests (of the numerical results and the theoretical properties that should be satisfied by the problem solution or by its discretization, etc.). The state of the art shows a tension between advanced numerical software libraries (possibly optimized for given architectures) and their usability by a mathematically trained user. Indeed, the transformation of a well-defined mathematical problem into an executable computer code is a major difficulty in the context of computational maths. In this paper we propose a software tool, called CFL (Computing from LaTeX), that aims at bridging this gap by leveraging an interacting problem-solving environment to expose the benefits of modern computational libraries up to the mathematical user level.

Our key observation is that LaTeX is a significant setting where maths and computing actually meet. Indeed, LaTeX is a programming language (a document markup language) that is widely used for a well-formatted printout of a mathematical problem as it is found in written books. Encoding a mathematical object into a (set of) LaTeX expressions is a familiar practice, especially in academia. We then require the user to simply specify the numerical problem as a familiar LaTeX document, which will represent the actual input of the CFL tool. In addition, we rely on the Python programming language, whose high level syntax allows a clean definition of mathematical abstractions. As a consequence, the (LaTeX) specification of the mathematical problem and its automated translation in the (Python-coded) numerical solution are pretty similar.

The core of CFL consists of a parser, a code generator and an engine. The parser scans a LaTeX document in order to recognize well-defined mathematical problems such as functions evaluation, linear algebraic problems, initial value problems for DLTI systems and for systems of ODEs, initial/boundary value problems for PDEs, feedback control problems, etc. . As a result, the parser automatically generates an executable simulation code, written in the Python programming language, according to a predetermined set of resolution patterns. More precisely, the generated code is structured into (*i*) the definition of a set of data and (*ii*) a list of instructions defining the execution logic of the program. Data are Python objects encoding the mathematical objects into which the problem is decomposed. On the other hand, the generated instructions correspond to the computational steps of an appropriate numerical solution pattern for the given computational problem. Finally, the engine contains the class definitions for both the numerical methods used by the resolution patterns and the mathematical objects, from algebraic equations, PDEs, control equations, etc.. down to (un)known variables and functions of unknown variables.

In other terms, CFL can be seen as a numerical problem-solving environment that converts the *specification* of a mathematical problem into a corresponding *resolution pattern* that can be directly executed. The gap between the mathematical definition of a numerical problem and the computation of its solution is covered by relying on high-level mathematical abstractions, which can be expressed both in LaTeX and in Python, helping both in problem recognition and in the generation of the code for the involved resolution pattern. Besides generating the executable simulation code, CFL allows to automatically insert the numerical results in an output LaTeX document, so that they can be displayed by means of tables or graphics. As a consequence, CFL can be easily used as a tool for explorative computation where parameters can be iteratively refined. As far as CFL efficiency is concerned, we observe that, apart from trivial problems, the parsing and the generation of the Python code represents a negligible overhead with respect to the computation of the numerical solution.

In the literature there exist various papers and related software projects with excellent contributions towards a better readability of mathematical software and its prototyping, but they aim at the creation of a new domain-specific language (DSL) [1] [2], which means to introduce a new programming language in the user operational framework (see Mathematica [3], Matlab [4], R [5], Maxima [6], Modelica [7], freefem++ [8], getfem++ [9], DOLFIN [10]).

Here we follow a different approach, taking into consideration two existing languages that are widely used and versatile: LaTeX and Python. We take advantage of the way they support mathematical talking and we let them interoperate by forming an execution chain that starts from the mathematical problem written in the LaTeX text and ends with the automatic execution of the numerical experiment that solves it. We did rely mainly in the creation of appropriate LaTeX definitions and on the definition of Python classes where the operators are conveniently overloaded, in such a way that the written formulation of the mathematical problem and its Python representation are as similar as possible. In practice, with Python it has been possible to write a library that remains Python but has the feeling of a DSL. Moreover, the approach adopted is applicable in general and therefore, although we will here make only a few meaningful examples of differential problems, it can be extended to different families of problems in scientific computing.

The structure of the paper is as follows: Section 2 begins with an overview of CFL and shows a few examples of mathematical problems CFL can deal with. In Section 3 we define a few requirements for a correct parsing of the LaTeX document, and the LaTeX commands needed in CFL computing. In Section 4 we explain how CFL converts mathematical problems into the Python code that implements the appropriate resolution patterns and the main design issues for their object oriented implementation. In sections 5 and 6 we give a few examples and draw conclusions and perspectives of this work.

2 An overview of CFL

CFL is a software tool that takes as input a LaTeX document containing the specification of a mathematical problem together with a few ad hoc commands dealing with the execution of the involved numerical experiments, that is to set the desired numerical methods and parameters to be used. The output is the same LaTeX document enriched with the obtained numerical results, represented by means of tables and figures.

A fundamental characteristic of CFL is that the details of the resolution methods are abstracted, not hidden. Therefore, they are within the control of the user but are independent from a specific numerical library. The aim is to generate expressive Python code that implements the required resolution pattern and can be linked to the preferred library.

2.1 CFL architecture

CFL is a software package is composed of the following modules:

1. a LaTeX package named *computable*, containing the definition of the new `computable_expression` environment which must be used to delimit the portion of a LaTeX document containing the specification of the problem (see Section 3.1). This package also contains the definitions of few handy LaTeX commands available to user to control the numerical experiments;
2. a parser of the LaTeX text (see Section 3) which extracts the mathematical problem(s) contained in the LaTeX text;
3. a Python code generator (see Section. 4) which generates the code that implements the resolution pattern associated to the recognized mathematical problem(s);
4. the Python modules containing the class definitions that implement the mathematical objects declared in the resolution patterns (see Section 4.2).

The CFL approach can be applied to a quite general family of problems. To illustrate, we present in the following a few basic families of mathematical problems that can be currently solved with CFL . However, its modular architecture has been designed taking into account the extensibility of CFL to new family of problems (see section 4.3).

2.2 Initial value problem for a system of ODEs

Initial value problems for ODEs are usually formulated in an abstract form: be $\mathbf{x} \in \mathcal{R}^n$, n is given, solve

$$\dot{\mathbf{x}} = \mathbf{f}(t,\mathbf{x}), \quad \mathbf{x}(t_0) = 0, \quad t \in [t_0, t_f] \tag{1}$$

but this is not the best expression to formulate a specific problem; the vector function $f(t,x)$ must be specified and usually it is preferred to describe each single differential equation in the system:

$$\begin{aligned} \dot{x}_1 &= f_1(t,\mathbf{x}), \quad x_1(t_0) = 0 \\ &\dots \\ \dot{x}_n &= f_n(t,\mathbf{x}), \quad x_n(t_0) = 0 \end{aligned}$$

We assume the writer express the computational problem in this second way. Therefore, CFL here considers the explicit writing of each equation of the ODE system.

The format of the ODE system is quite general (see the examples of sec. 5): each equation is supposed to have associated an independent variable, say $x_i(t)$, and one or more of the following terms:

- a first order derivative multiplied by a coefficient, e.g.: $C_i \frac{dx_i(t)}{dt}$, $d_R \dot{x}_i(t)$, etc.
- (optional) one linear term for each independent variable in the system: $R_i(x,t)x_i(t)$
- a forcing term: $f_i(t)$
- (optional) a nonlinear term $N_i(x,t)$

2.3 Initial-boundary value problem for a second order PDE

Let $\Omega = [x_a, x_b] \times [y_a, y_b]$ be a rectangular domain, whose boundary is $\partial\Omega = \Gamma_D \cup \Gamma_N$, where Γ_D and Γ_N are supposed to be two sets of segments of Ω in which we want to apply Dirichlet and Neumann boundary conditions, respectively. Let us consider the well-known class of initial-boundary value problems:

$$\begin{cases} d\frac{\partial c}{\partial t} - \mu\Delta c + \mathbf{b}\cdot\nabla c + \sigma c & = & f, & in \quad [0,t_f]\times\Omega \\ c & = & c_0, & on \quad \{0\}\times\Omega \\ c & = & c_D, & on \quad (0,t_f]\times\Gamma_D \\ \mu\frac{\partial c}{\partial n} & = & c_N, & on \quad (0,t_f]\times\Gamma_N \end{cases} \tag{2}$$

where c is the solution of the convection-reaction-diffusion equation, with initial condition c_0 and boundary conditions c_D e c_N on Dirichlet and Neumann boundaries.

2.4 Feedback Control

Both model problems previously considered can be coupled with a control law. For example, if we want to control a functional F of the solution c of problem (2) with respect to a reference value F_{ref}, we define the error function:

$$e(c) = F_{ref} - F(c) \tag{3}$$

and a simple Proportional-Integrative-Derivative (PID) control law that, for example, modifies the Dirichlet boundary conditions trying to zero the error:

$$c_D = K_D\frac{de(c)}{dt} + K_P e(c) + K_I\int e(c) \tag{4}$$

3 Parsing a well-defined LaTeX document

In this section, we define the simple requirements we impose on the input LaTeX text to allow a correct parsing without restricting the expressiveness of the overall document.

1. We require the computational problem to be unambiguously defined in a clearly delimited portion of the text, leaving a total freedom of expression in the remaining part of the latex document.

2. We introduce a few novel LaTeX commands specifically for CFL . These commands transform a typesetting language like LaTeX in a programming language for the computing environment CFL , maintaining the mathematical typesetting appearance.

3.1 The *computable_expression* environment

The new LaTeX environment `computable_expression` allows to specify a mathematical problem, a numerical experiment on it and to automatically execute it with CFL . More precisely, a LaTex document can contain one or more blocks of type:

```
\begin{computable_expression}
      ...
\end{computable_expression}
```

containing free text, formulas, and specific CFL commands. Each block is elaborated separately, i.e. each `computable_expression` is assumed to be independent form the others.

We observe that the specification of the mathematical problem inside a `computable_expression` involves mathematical objects of different kinds, that can be expressed in a number of ways. First, they can be written within the standard LaTeX tags like `$ ... $`, `$$ ... $$`, `\[ ... \]` and `\begin{equation} ... \end{equation}`, as usually done by mathematicians. Notice that the text written between these tags is constrained by the LaTeX syntax, but allows to define the same mathematical object in different ways.

It is a matter of style, which can depend upon the personal feeling of the writer, or has its origin in the notation adopted by an influencing group of mathematicians or books. A different approach would be to define a set of specific CFL commands, enforcing a programming language, namely a DSL, into LaTeX .

We preferred to adopt the first approach as much as possible, that is to rely on the usual LaTeX writing apart from situations where an ad hoc command simplifies the job. In fact, in the same spirit new commands/macros are frequently used by LaTeX writers. We think that this is a good compromise between simplicity of the parser and of the mathematical text. It stays between two extrema: from one side, a mathematical problem written with a lot of pre-built commands demands a simple parser. From the other side, a mathematical problem written in a too abstract notation may demand a parser very complicated or even impossible to be realized.

Finally, remind that in the past valuable attempts to define a unique notation for mathematicians (e.g. Householder notation in linear algebra) has gone only a little bit faraway. Therefore, CFL takes into account flexibility and contemplates the possibility to encounter different writing styles for the same object. Nevertheless, CFL documentation enforces a few templates for the more common mathematical objects. For instance, we introduce the `\setfuncase{...}{...}{...}` command to simplify the definition of piecewise functions.

Technically speaking, the CFL parser uses a widely used text pattern recognition tool, called *regular expressions*, or *regexp*: *"A regular expression specifies a set of strings that matches it"*. The intrinsic variability in different writing styles of the various mathematical objects has required quite an effort in fine tuning the regexps. Indeed, the mathematical text does not contain specific delimiters for each mathematical object; for example, rounded brackets can be used in a variety of objects with a different meaning and purpose. Instead, it has been much easier to recognize the mathematical objects coded as ad-hoc commands, like `\setfuncase{...}{...}{...}`.

The flexibility offered by CFL in the writing style has obviously a fundamental limit: in order to be represented in a computer program, a mathematical object must be fully defined, leaving no implicit meaning. The parser must recognize various kinds of mathematical objects, e.g. operators, variables and functions. A full mathematical definition of such objects (at the text level) actually contains all the necessary information, allowing the translation of the LaTeX text into an executable computer program. Any incomplete or ambiguous definition of the input mathematical problem leads to an error in the CFL execution.

In the following we illustrate how CFL recognizes a few mathematical objects involved in the specification of the problems introduced in Section 2. For a detailed description we refer to [13].

3.2 LaTeX commands and mathematical objects

In all the problems considered in Section 2, i.e., (1), (2) and (4), there are several combinations of mathematical objects (terms and expressions) that satisfy, but a few variants, a precise notation and syntax. These combinations cover a variety of situations of practical interest, in applications.

Let us see a LaTeX example of a mathematical problem taken from the class (1), that is ready for CFL computing. It is a system of three ordinary differential equations (ODEs), the well known Lorenz system. At the beginning there is the definition of the ODEs coefficients, which are in this case constant, and of the three independent variables, which will be the solution of the initial-value problem:

```
Let $p = 10$, $r = 64$, $b = 8/3$.
Let $x_1(t)$, $x_2(t)$ and $x_3(t)$ be the convection
intensity, the maximum temperature difference and the
stratification change respectively.
```

Then, the differential equations are written, as usually done:

```
The system equations are:
$$\dot{x_1}(t) = p x_2(t) - p x_1(t)$$
$$\dot{x_2}(t) = r x_1(t) - x_2(t) - x_1(t) x_3(t)$$
$$\dot{x_3}(t) = x_1(t) x_2(t) - b x_3(t)$$
```

Now, something must be written about which discretization method in time to adopt, a piece of information required by CFL to solve numerically the differential problem and usually documented also in pure LaTeX documents. To this aim the user must use the `\setTimeDiscrMethod` command. Then, the time interval in which the problem is defined must be also specified:

```
The discretization in time is made with the
\setTimeDiscrMethod{Implicit Euler} method. Simulation
duration is set being $t_0=0$,$T = 2$ with $dt = 0.01$s.
```

The initial conditions must be specified to solve an initial value problem:

```
The system initial conditions are the following:
$x_1(0) = 1$, $x_2(0) = 2$ and $x_3(0) = 3$.
```

and this is enough for CFL to understand the problem to be solved, and actually solve it. The results can be retrieved in a graphical form with a command like `\createFigure`:

```
Results are shown in Figure
\createFigure{x_1(t)}{x_2(t);x_3(t):'r--'}.
```

4 Automatic generation and execution of the solution program

We designed the code generator so that the generated computer code

- implements an appropriate resolution pattern and contains all the information of a quite complex mathematical problem (as those in Section 2), and
- allows a good readability of mathematical equations in terms of a restricted number of code instructions.

For this reason we used the Python language, which enjoys operator overloading, is an easily extensible object-oriented language, possesses an API for the integration of C/C++ code and a proper set of modules for numerical computing. Python has attracted a lot of attention in the scientific computing community in the last ten years, and there are now available a lot of wrappers to well reputed C/C++ numerical libraries.

Therefore, besides the code generator, the CFL engine comprises a set of Python modules that deliver the classes for the formalization of problems with ordinary and partial differential equations, and methods to solve these problems with a variety of numerical methods. Let us see an example. The problem (2) with $d = 0$, i.e. in the stationary case, is translated into the instructions shown in Table 1:

```
- mu * laplacian(c) + (b) * grad(c) + sigma * c == f in Omega_h
c == c_D in Gamma_D(Omega_h)
c == c_N in Gamma_N(Omega_h)
```

Table 1: Python code for a boundary value problem. Note that we use the operator ”==”, that in Python is the operator of comparison between the left and the right side.

We observe that this generated code, together with the class definitions provided by the CFL engine, can be interpreted by the standard Python interpreter. Indeed, the Python classes defined in CFL carefully rely on well suited identifiers and on the overloading of the operators so that the three instructions of Table 1 are regular Python instructions. Therefore, they are parsed, interpreted and executed by the standard Python runtime. Moreover, these instructions still represent the problem expressed in an abstract form. Then, the execution of these instructions actually produces a collection of objects which are capable of solving the problem they express. These objects are instances of classes contained in the CFL engine, see subsection 4.2 for a brief description; for the details, see the on-line documentation [13].

Finally, we point out here that while mathematical objects can be written with freedom of expression, the expressivity of the CFL generated code is more restricted: different styles of writing a mathematical expression (e.g. different orderings of terms) can lead to the same code (e.g. with the same ordering of the terms). This helps the design of the CFL engine in the construction of the overloading scheme adopted for the operators and allows to think at the problem independently from the user's writing style.

In section 4.1 we briefly describe the Python implementation of the mathematical objects recognized by CFL , and the Python code generation of the resolution pattern for the mathematical problems of section 2; then we describe the CFL engine operation for these problems, see section 4.2. Finally, we outline the extension process of CFL to new families of problems (sec. 4.3).

4.1 Mathematical objects and Mathematical problems

4.1.1 Mathematical objects

Let us see the Python implementation of various mathematical objects defined in the families of problem here considered. They can be grouped as:

- known variables, that can be used without prior solving a mathematical problem. They are represented in two different ways, according to the fact that they contain numerical data, like:

  - known constants, independent variables (typically time, space, etc.), time-series or discrete-time signals, and piecewise constant/linear functions; they are declared as Python variables of type scalar/vector/matrix of `double`;

  or possess an analytical expression, like:

  - known functions of independent variables; they are defined with the standard Python functions construct "def" and return a string containing the analytical definition of the function that, when evaluated (by means of the `eval` construct), returns a value of type scalar/vector/matrix of `double`.

- unknown variables, that must be related to a precise mathematical problem and their values are known only after its solution. They are unknown constants and unknown functions of independent variables (e.g. ODEs solutions), and can be represented in two different ways, depending on we want to generate code directly for a specific solver library or for a general CFL solver:

  - for a specific solver library they are represented by Python variables of type scalar/vector/matrix of `double` (generally NumPy [12] arrays. Indeed, we assume that the library has a Python wrapper that uses NumPy arrays for its API.
  - for a general CFL solver they must be specific objects for that family of problems, e.g. of type `PdeSolution` or `OdeSolution` for the problems of sec. 2.

- Known functions of unknown variables are defined with the standard Python functions construct "def" and return a string containing the analytical definition of the function that, when evaluated (by means of the `eval` construct), returns a value of type scalar/vector/matrix of `double`, supposing that the unknown variables has been already computed.

- differential operators on those unknown variables appearing in problems (1) e (2), and therefore: partial time derivative, first order and second order partial space derivatives, ordinary time derivative; they become objects of classes specifically designed in CFL for these operators.

- the problem domain, which is space-temporal, and therefore it is necessary to define space and time intervals, and their specific use in ODES and PDEs. For example, the class `I` represents a space interval and its constructor allows to define a 1-d interval; the product of two objects from this class,

`I1 * I2`, implements the Cartesian product of the two 1-d intervals and returns an object of the same type, but that represents an interval of dimension $dim(I_1) + dim(I_2)$.
The problem domain is defined in the continuum and then discretized. The discretization in time is managed implicitly, and therefore not explicitly represented. The discretization in space is instead represented by the class `Mesh`. For problems (2) it contains a grid of nodes if the discretization method is the Finite Differences or a mesh of Finite Elements if the chosen method is the FEM.

- differential equations: the class `PdeEquation` contains a field with a dictionary of coefficients, i.e. those defined in problem (2). This is a useful feature given by Python dictionaries, to use the coefficient name as the index, instead of an anonymous numerical index in a vector of coefficients. The class `OdeSystem` contains a matrix of coefficients (for the linear part of the system), a vector of nonlinear terms and a vector of forcing terms.

4.1.2 Mathematical problems

Each family of mathematical problems has a related resolution pattern that CFL implements in Python. When the parser has finished his job, CFL has a list of mathematical objects recognized by the parser. From this list, the code generator understands which is the resolution pattern to be applied. Here we see the Python implementations of a few significant resolution patterns:

- evaluation of some nested functions and or linear algebraic problems: the resolution pattern becomes a sequence of Python instructions that evaluate the defined functions respecting the correct sequence;
- Initial value problems for systems of ODEs:

```
<unknown function of independent variables> = OdeSolution()
x_CfL = [float(ic) for ic in OdeSystem.current.initcond]
for t_CfL in np.arange(t_0_CfL,t_f_CfL,dt_CfL):
    <definition/update of ODEs>
    pr.method.set_tDiscr(<chosen method>,timeInterval(0.0,dt_CfL),dt_CfL)
    x_CfL = pr.method.solve(t_CfL)
#endfor
```

- Initial/Boundary value problems for PDEs:

```
<unknown function of independent variables, e.g.: T = PdeSolution()>
<PDE, e.g.: d * tDer(T) + -mu * laplacian(T) +b * xder(T,0) +sigma * T == f in Omega>
<initial conditions, e.g.: T == T_0 in t(0)>
<boundary conditions, e.g.: T == g_1 in Gamma_D_1, gradn(T) == c_N in Gamma_N>
pr.method = <chosen method, e.g.: builtInFEM()>
pr.method.set_tDiscr(<chosen method, e.g.: 'IEuler'>,t,dt)
Omega_h = <space discretization, e.g.: Mesh.build_structured_rectangular_mesh(Omega,h)>
pr.method.set_mesh(Omega_h)
pr.method.solve()
```

4.2 Design and implementation of the execution engine

In figure 1 it is shown the UML scheme of the main classes involved for the solution of problems (1) and (2).

The module *Problems* contains one Python class for each family of problems recognized by CFL . Each of them is a subclass of the `Problem` class. Each run of the code generated by CFL for the solution of the input mathematical problem, works with the global object from this class, that the Python environment creates by default. In particular, the field `Problem.current` will contain a list of objects from some classes of the module *Problems*, that describe the input mathematical problem. The creation process of such objects is here briefly described.

Let us consider the model problems of section 2; at the beginning of the generated code, it is called the constructor of some objects of type `PdeSolution` or `OdeSolution`, corresponding to the unknown

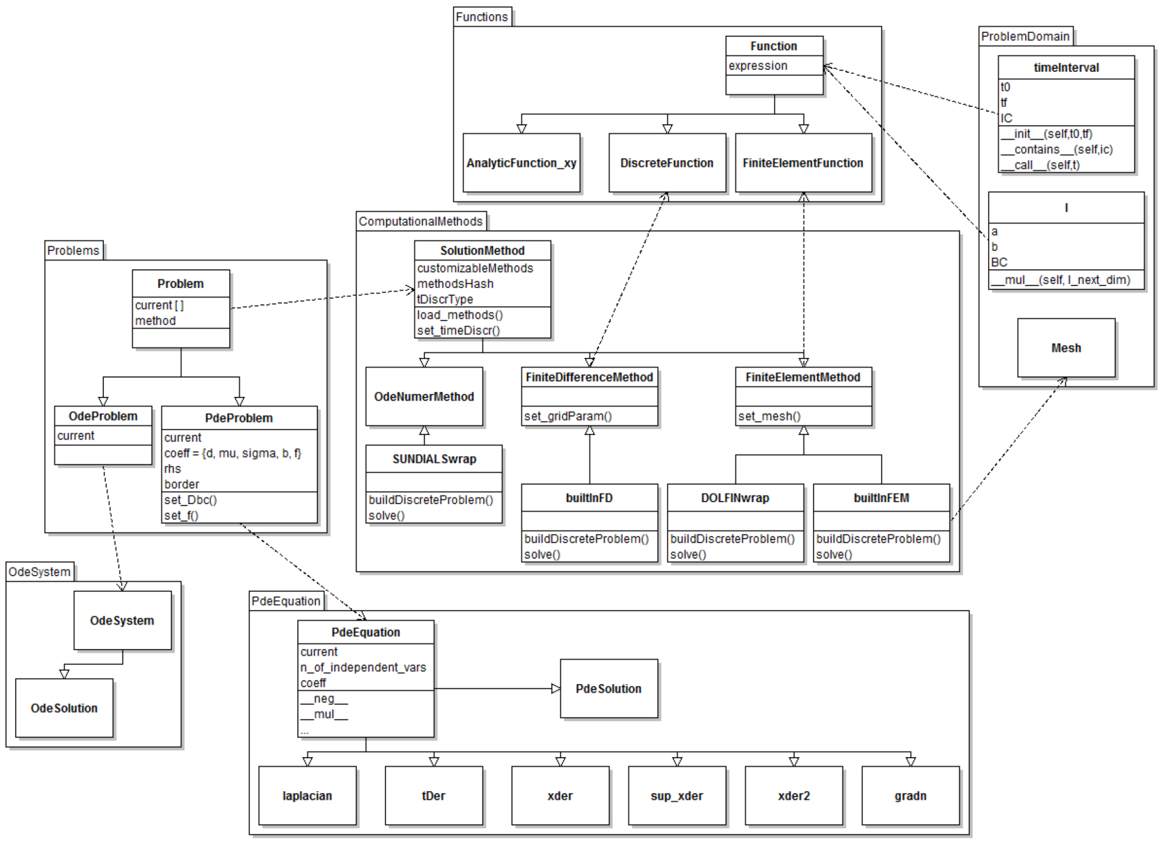

Figure 1: Diagram of the main classes in the CFL engine.

variables, which are subclasses of `PdeEquation` and `OdeSystem`, that are the fundamental classes for these two model problems. Note that there is an entire module dedicated to each of these classes. In this way, when the Python interpreter executes the instructions containing the differential equations (see Table 1), the overloaded operators defined in various classes of these modules will return an object `PdeEquation` or `OdeSystem` more and more rich of details about the differential equations to be solved. For example, the expression `- mu * laplacian(c)` will return an object of type `laplacian` with a coefficient set to `- mu`. When the operator "==" is evaluated, the equation is completed and it is added to the field `PdeEquation.current` or `OdeSystem.current`. Objects like these are created for each PDE and system of ODEs present in the mathematical problem.

After the instructions containing the differential equations, those specifying the initial and boundary conditions follow to satisfy the mathematical well-posedness of the problem. They contain objects of type `PdeSolution` or `OdeSolution` (e.g. `c` in Table 1). When the Python interpreter executes these instructions, the overloaded operators of these objects create an object `PdeProblem` or `OdeProblem` and add to its field `current` the object `PdeEquation.current` or `OdeSystem.current` and these initial and boundary conditions. Then, these new objects are inserted in the list contained in `Problem.current`.

4.3 Extension to new classes of problems

The aim of this research has been to create a software tool capable of automatically translating the *specification* of a mathematical problem belonging to one specific family of problems into an executable computer code inspired to an appropriate *resolution pattern* for that family. The tractable number of families is unlimited, but it is not conceivable to predetermine the whole set of mathematical problems which can be solved by a computer; it is evolving continuously. Therefore, we consider the extension of CFL a fundamental issue: given an initial configuration of CFL , which recognizes the families of problems presented in section 2, extend it to a new family without remaking the things already done.

It turns out that the steps to be done to introduce in CFL a new family of problems are always the same; the difference is the creation of new entities specific to the class added, that are:

- new mathematical objects (terms and expressions), written in LaTeX , and new LaTeX commands added to the package *computable* (sec. 2.1); they must contain all the information required to specify a mathematical program of this new family.
- the corresponding parsing rules; they must recognize all the new objects without confusing them with the objects of the previously defined families of problems;
- new Python classes, corresponding to the new mathematical objects;
- a new resolution pattern for the generation of the solution code in Python.

Let us see the main rules to follow to extend CFL to a new family of problems.

4.3.1 Adding a new class of problems

Assuming that a new family of problems has been defined, all the necessary LaTeX expressions (and the corresponding regexp) can be either comprised in previously existing families of problems, or be new. The latter must be defined without interfere with the parsing of the existing ones. In particular, there is one main rule that must be kept in mind, leaving to the documentation [13] for other minor rules:

- the parser of CFL accepts different styles for the same mathematical object and the text items that participate to the construction of the object can be put in arbitrary order. For example, it accepts `Let $p=1.3$ and define $f(t)=p t$`, but also `Let us define $f(t)=p t$, where $p=1.3$`.

This means that old and new mathematical objects can be mixed arbitrarily. A correct and unambiguous formulation of the new mathematical problem in the LaTeX text is sufficient for the parser to recognize the mathematical objects described.

For the creation of new Python classes corresponding to the new mathematical objects, it should be analysed if and how they interact with existing ones: if the mathematical objects of the new family of problems are all new there are no interactions, otherwise it must be checked that the overloading of the operators in the existing reused classes is not incompatible with the new family of problems, which is likely. Reusing existing classes may be subtle, and therefore we stated a general rule that can help in compatibility

- all the variables and functions, known and unknown, return a value of type scalar/vector/matrix of `double` or a string that, when evaluated, returns a value of type scalar/vector/matrix of `double`.

For the creation of a new resolution pattern according to which generate the Python code, i.e. the problem solver, it is necessary to understand the mechanism that builds the solver, starting from the problem in the LaTeX text recognized as a member of a determined family of problems. This mechanism, reflects the structure of an abstract resolution pattern, made for being generic and therefore allowing the addition of new resolution patterns to the existing ones. It can be described as a list of code blocks, where each is present only if required by the specific problem:

```
1. import of external Python modules
2. assignment of: - known variables/functions,
                 - spatial/temporal domains of the problem
3. set an iteration on subproblems
4. compute the unknown variables of the sequence of algebraic problems
6. compute the functions of these unknown variables
5. compute the unknown variables of the differential problems
6. compute the functions of these unknown variables
7. end the iteration on subproblems
```

```
    8. print the results.
```
A new resolution pattern can be added by extending (i.e. adding instructions to) each of these code blocks in the solver code generator.

4.3.2 Using a new solution library

The aim of this work is to reuse as much as possible the existing numerical libraries and not to create necessarily new ones. The module `CodeGeneration`, independent from the parser, can be used to generate code for any existing library, be it native in Python or wrapped.

An interesting aspect of this work is that in this way different libraries can cooperate even to the solution of the same problem, and their interface is abstracted to the mathematical problem level.

5 An example

Let us consider the example presented at section 3.2:

Let $p = 10$, $r = 64$, $b = 8/3$. Let $y_1(t)$, $y_2(t)$ and $y_3(t)$ be the convection intensity, the maximum temperature difference and the stratification change respectively.

The system equations are:

$$\dot{y_1}(t) = py_2(t) - py_1(t)$$

$$\dot{y_2}(t) = ry_1(t) - y_2(t) - y_1(t)y_3(t)$$

$$\dot{y_3}(t) = y_1(t)y_2(t) - by_3(t)$$

The discretization in time is made with the `\setTimeDiscrMethod{Implicit Euler}` method. Simulation duration is set being $t_0 = 0, T = 2$ with $dt = 0.005$s.

The system initial conditions are the following: $y_1(0) = 1$, $y_2(0) = 2$ and $y_3(0) = 3$.

Results are shown in Figure `\createFigure{y_1(t)}{y_2(t);y_3(t):'r--'}`.

When CFL executes this example, it generates and executes this Python code:

```
# declaration of known variables:
p = 10
b = 8/3
r = 64
# defintion of the time domain of the problem:
T = 2
dt = 0.005
t_0 = 0
# initialization of the problem unknown variables:
y_1 = OdeSolution(Function('1'))
y_2 = OdeSolution(Function('2'))
y_3 = OdeSolution(Function('3'))
y_1_hist1 = [0 for i in np.arange(t_0,T,dt)]
y_2_hist1 = [0 for i in np.arange(t_0,T,dt)]
y_3_hist1 = [0 for i in np.arange(t_0,T,dt)]
# setting of initial conditions:
x = [float(ic) for ic in OdeSystem.current.initcond]
x_cfl_past = np.copy(x)
dtmin = dt
it = 0
# solution of the initial-value problem:
for t in np.arange(t_0,T,dtmin):
    1*tder(y_1) == p*y_2 - nnl_term(p*nonLinear(y_1))
    1*tder(y_2) == r*y_1 - 1*y_2 - nnl_term(nonLinear(y_1)*nonLinear(y_3))
    1*tder(y_3) == y_1*y_2 - nnl_term(b*nonLinear(y_3))
    pr = Problem.current[0]
    pr.method.set_tDiscr('IEuler',timeInterval(0.0,dtmin),dtmin)
    x = pr.method.solve(t)
    Problem.current = []

    y_1_hist1[it] = x[0]
    y_2_hist1[it] = x[1]
    y_3_hist1[it] = x[2]

    y_1 = OdeSolution(Function(str(x[0])))
    y_1.SetPastValue(x_cfl_past[0])
    y_2 = OdeSolution(Function(str(x[1])))
    y_2.SetPastValue(x_cfl_past[1])
    y_3 = OdeSolution(Function(str(x[2])))
    y_3.SetPastValue(x_cfl_past[2])

    x_cfl_past = np.copy(x)
    it += 1
#endfor
# print of the results:
import pylab as p
p.clf()
pfig=p.figure(1); p.hold(True); p.title('figure $(y_{1}(t),y_{2}(t);y_{3}(t))$');
pfig.set_size_inches(12,8)
p.plot(y_1_hist1,y_2_hist1,label='y_2');
p.plot(y_1_hist1,y_3_hist1,'r--',label='y_3');
p.legend();
p.savefig('ce0test1m1fig1.png',bbox_inches=0,dpi=100);
```

and generates Figure 2 with the computed results.

6 Conclusions and related work

Even with a simple example like that of sec. 5, the overhead required by CFL for parsing and code generation is a small percentage of the total time, and this is a big advantage considering the time required to write manually the solver code. With examples that are more computational demanding, the overhead becomes even more negligible.

CFL is a natural tool for people doing computational mathematics: we have not created a new DSL; we have used the existing "mathematical language" in a computing system. The mathematical language is not strictly tied to a grammar, like computer languages are, but it is sufficiently formalized and abstract that a mathematical problem can be automatically translated in a computer language implementation of its solution. Moreover, the mathematicians use naturally the overloading of the operators, e.g. the multiplication operator is used with different mathematical object types.

CFL offers a serie of practical advantages:

- it allows an automatic verification that the problem written in the LaTeX text is well defined and complete, since it is executed by a computer;
- it guarantees that the results reported in the LaTeX text are corresponding to the algorithms described therein, since they are automatically generated from the text;
- it allows to do numerical experiments even to whom does not possess any programming skills, without the need of complicated graphical user interfaces, since CFL interfaces directly the LaTeX text with the computing system.

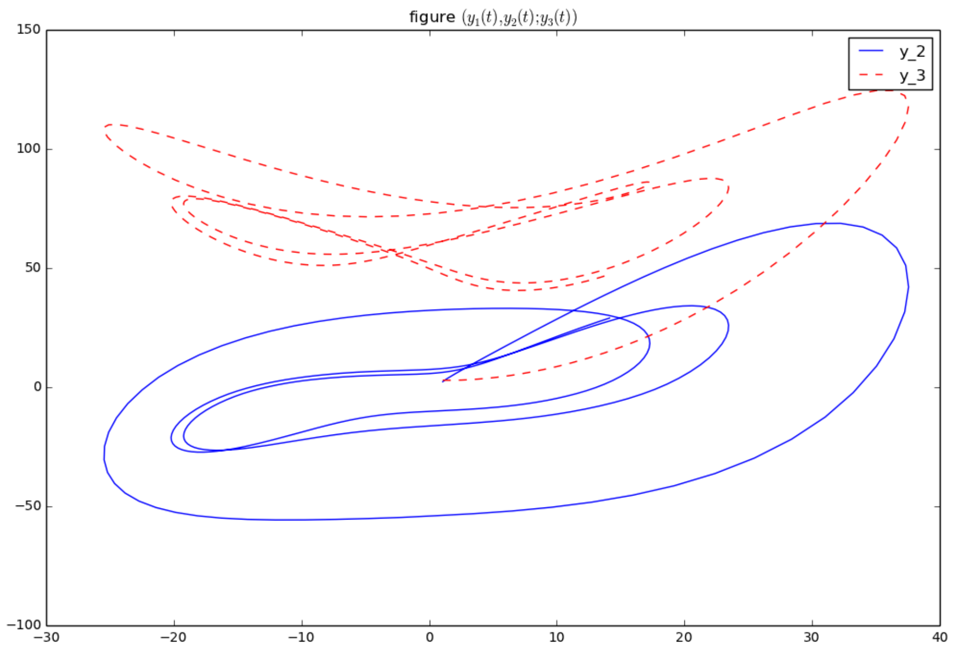

Figure 2: Computed solution for the Lorenz example.

References

[1] Domain-Specific Languages, A Theoretical Survey - CoRTA09DSLsurvey_vf.pdf

[2] C.PRUD'HOMME, *A Domain Specific Embedded Language in C++ for Automatic Differentiation, Projection, Integration and Variational Formulations,* 2006.

[3] Mathematica website, http://www.wolfram.com/mathematica/

[4] Matlab website http://www.mathworks.it/products/matlab/

[5] R Project website, *http://www.r-project.org/*

[6] Maxima website, *http://maxima.sourceforge.net/*

[7] Modelica website, https://www.modelica.org/

[8] Freefem++ website, http://www.freefem.org/ff++/

[9] Getfem++ website, http://download.gna.org/getfem/html/homepage/

[10] Logg A., Wells G.N., “DOLFIN: automated finite element computing”, *ACM Trans. Math. Software* **37** (2010), no. 2, Art. 20, 28 pp.

[11] A. Hindmarsh et al., “SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers”, *ACM Trans. Math. Software* **31** (2005), no. 3, pp. 363-396

[12] NumPy website, *http://www.numpy.org/*

[13] http://www.math.unipd.it/˜marcuzzi/Computing_from_LaTex.html