Software System Design based on Patterns for Newton-Type Methods

Ricardo Serrato Barrera[*,a], Gustavo Rodríguez
Gómez[†,b], Julio César Pérez Sansalvador[‡,b,c], Saul E.
Pomares Hernández[§,b], Leticia Flores Pulido[¶,d], and
Antonio Muñoz[‖,e]

[a] *Estratei Sistemas de Información S.A. de C.V., Virrey de Mendoza 605-B, Col. Las fuentes, 59699, Zamora, Michoacán, México*
[b] *Instituto Nacional de Astrofísica, Óptica y Electrónica, Computer Science Department, Luis Enrique Erro 1, 72840, Tonantzintla, Puebla, México*
[c] *Cátedra CONACyT - Instituto Nacional de Astrofísica, Óptica y Electrónica, Computer Science Department, Luis Enrique Erro 1, 72840, Tonantzintla, Puebla, México*
[d] *Universidad Autónoma de Tlaxcala, Facultad de Ciencias Básicas, Ingeniería y Tecnología, Calzada Apizaquito, Colonia Apizaquito, 90300, Apizaco, Tlaxcala, México*
[e] *Departamento de Ingenierías, Universidad de Guadalajara, Ave. Independencia Nacional, Autlán, Jalisco, México*

Abstract

A wide range of engineering applications uses optimisation techniques as part of their solution process. The researcher uses specialized software that implements well-known optimisation techniques to solve his problem. However, when it comes to develop original optimisation techniques that fit a particular problem the researcher has no option but to implement his own new method from scratch. This leads to large development times and error prone code that, in general, will not be reused for any other application. In this work, we present a novel methodology that simplifies, fasten and improves the development process of scientific software. This methodology guide us on the identification of design patterns. The application of this methodology generates reusable, flexible and high quality scientific software. Furthermore, the produced software becomes a documented tool to transfer the knowledge on the development process of scientific software. We apply this methodology for the design of an optimisation framework implementing Newton's type methods which can be used as a fast prototyping tool of new optimisation techniques based on Newton's type methods. The abstraction, re-useability and flexibility of the developed framework is measured

[*] rsbserrato@inaoep.mx

[†] grodrig@inaoep.mx

[‡] jcp.sansalvador@inaoep.mx, correspondence author

[§] spomares@inaoep.mx

[¶] leticia.florespo@udlap.mx

[‖] jose.munoz@cucsur.udg.mx

by means of Martin's metric. The results indicate that the developed software is highly reusable.

Keywords: Scientific Software Design, Object–Oriented Programming, Design Patterns, Newton's methods, Optimization techniques

1   Introduction

A wide range of applications are solved using a variation of Newton's method [4], these include: trust regions methods and line search methods [27], damping methods, inexact or truncated methods [26], quasi–Newton methods [41], or hybrid methods [6], [32]. Each variation presents specific features that make it suitable for the solution of a particular problem: unconstrained optimisation, solution of nonlinear systems of equations, or data fitting problems.

Name	Language	Comments
MINPACK 1.0 [24]	Fortran	Optimization package
MINOS 5.0 [25]	Fortran	Optimization package
UNCMIN [36]	Fortran	Unconstrained minimisation, evolved to TENSOLVE
HOMPACK [40]	Fortran	Homotopies package
TENSOLVE [7]	Fortran	Tensor methods package
NITSOL [31]	Fortran	Nonlinear equations solver package
Matlab software routines [19]	Matlab	Nonlinear equations solvers
COOOL [20]	C++	Optimization and matrix operations package
PETSc [1]	C++	Differential equations and nonlinear systems solver package
OPT++ [23]	C++	Optimization package

Tab. 1: A selection of software packages implementing Newton–type methods to solve specialized problems.

Currently, there is a large collection of specialized software packages that implement variations of Newton–type methods; see Table 1. Note that most of these software packages are implemented following the *procedural programming paradigm* which main drawback is the lack of flexibility of the developed code; that is, the software is hard to extend or reuse [23, 8, 13, 16, 22]. Generally, software packages developed using the procedural programming paradigm present the following problems:

- Code that is difficult to read and understand due to the use of unsuitable data structures, [13].
- Code that does not describe the algorithms that implements, [16].
- Code that is hard to reuse or modify since it is based on very restrictive programming concepts and the large number of parameters in the interfaces [8, 22].

Only three of the software packages presented in Table 1 implement *the object-oriented programming* approach; this shows us the preference of the procedural programming paradigm over the object-oriented paradigm in the scientific community. The three software packages implementing the object-oriented approach are: COOOL (The CWP -Center for Wave Phenomena- Object-Oriented Optimization Library) [20], PETSc (Portable, Extensible Toolkit for Scientific Computation) [1] and OPT++ [23]. COOOL only handles unconstrained optimisation problems and do not fully exploit the object-oriented paradigm; concepts such as inheritance and polymorphism are not exploited in its software design. PETSc implements Newton's method for the solution

Work	Context	Proposed patterns
Blilie, 2002, [5]	Dynamic systems	Particle–Particle, Particle–Mesh, Mesh.
Rodriguez et al. 2004, [33]	Dynamic systems	Model-Solver, System-Modularization
Matthey et al. 2005, [9]	Nodes three-dimensional simulation of morphogenesis, Molecular Dynamics	Generic Automation, Plugins, Dynamic Class Nodes, Policy, Multigrid, Strategy Chain
Heng and Mackie 2009, [17]	Finite Element	Model-Analysis separation, Model-UI separation, Modular element, Composite element, Modular analyzer
Rouson et al. 2010, [35]	Multiphysics Modeling	Semi-Discrete, Surrogate, Puppeteer
Rouson et al. 2011, [34]	Multiphysics Modeling	Abstract Calculus, Surrogate, Puppeteer

Tab. 2: Works that propose new scientific software patterns.

of nonlinear systems of equations, it is a set of routines and data structures for the solution of PDEs that is widely used for the scientific community. `OPT++` is a software library for nonlinear optimisation in `C++`, it uses inheritance and polymorphism to provide an interface hierarchy to implement quasi–Newton methods, inexact Newton methods, line search methods, and trust region methods.

The application of object-oriented techniques for the development of scientific software started in the late 90's; `Diffpack` [8] was a pioneer project in the application of object-oriented techniques, it was a software for the simulation of engineering and scientific applications.

1.1 Software Design Patterns and Scientific Software Development

Software design patterns were firstly introduced by Gamma *et al.* in the book "*Design patterns: elements of reusable object–oriented software*" [15]. Design patterns are software design solutions to common and repeatable software design problems. Software design patterns benefit low coupling and strong cohesion relations between software components; therefore, the resulting software is easy to reuse, modify and extend.

The application of design patterns for the development of business software dates back to the 1990's [34]. Even though, currently there is neither a recognised pattern language for the numerical specialists nor a research area focused on the specification or application of design patterns for the development of scientific software.

In recent years, the scientific community has shown interest in the application of design patterns to benefit from the knowledge of expert software designers for the development of scientific software [23, 13, 16, 22, 34, 35, 14, 2]. Tables 2 and 3 show a list of selected works in scientific computing that apply design patterns, we organised them into two main categories:

- Works that *find, create and specify* new scientific software patterns, see Table 2.[1]

- Works that *borrow, modify and apply* software patterns from other areas (such as those for the development of administrative software) for the development of scientific software, see Table 3

---

[1] According to Vlissides [39], any new proposed scientific pattern can not be immediately recognised as real patterns until they are validated by specialist, and their offered solutions are applied in other contexts.

Work	Context
Padula et al. 2004, [29]	Simulation and optimization
Decyk et al. 2008, [11]	Particle in cell simulation
Sansalvador et al. 2011, [30]	Multirate integration methods
Rouson et al. 2011, [34]	Multiphysics Modeling
Barbieri et al. 2012, [2]	Matrix Computations
Filippone et al. 2012, [14]	Matrix Computations
Barrera et al. 2017, [4]	Newton-type methods software

Tab. 3: Works that uses software patterns in scientific software applications.

The works listed in Table 3 benefit from the application of design patterns in their software designs, they gain flexibility and reusability. Nevertheless, none of them explains the methodology used to select the software patterns.

1.1.1 Difficulties on the application of design patterns for the development of scientific software

The specification of a pattern is often abstract, hence determining or mapping a pattern to a specific application is a difficult task that mainly relies on the expertise of the software developer. In order to apply a design pattern, we must *identify the software elements that comprise the design.*

In the scientific software field there are no explicit relations between numerical concepts and design patterns, therefore we need to establish such relations in order to develop a numerical software based on software patterns. In the particular case of Newton–type methods, we have to identify or establish relations between the abstract concepts in the domain of the problem, the numerical methods involved in the implementation of Newton–type methods, and the design pattern's domain.

The main contribution of this work is a methodology that guides the researcher through the identification of software design patterns to efficiently apply the object-oriented paradigm to develop reusable and easy to adapt software. This methodology applies the object–oriented paradigm, domain analysis, and Scope Commonality and Variability analysis (SCV) [10]. We start by performing a domain analysis that help us to generate abstractions to understand the problem domain in terms of software patterns. Then we select a subset of design patterns from Gamma's book [15] that can be applied to address the problems in the software design. We provide guidelines to determine what software patterns to use. Finally, we apply Martin's metrics [21] to provide evidence that the resulting software design is easy to reuse, extend and modify.

We apply the mentioned methodology to develop a framework for the implementation of Newton–type methods. We capture the structure of Newton–type methods in a novel software design by means of software patterns. Our newly developed software system design facilitates the implementation of algorithms by providing a flexible and extensible framework to incorporate new Newton–type methods and easily integrate off-the-shelf software libraries.

1.2 Paper Organisation

In Section 2 we introduce the mathematical notation used in the paper and define the mathematical problem regarding Newton–type methods. Then in Section 3 we describe the methodology used to identify concepts and relations to generate software components that will be associated via design patterns. The application of this methodology for the development of a Newton–type methods software system is presented

in Section 4. In Section 5 we present an evaluation of the resulting software design by using Martin's metrics. The conclusions and future work are presented in Section 6.

2 The mathematical problem

We start by identifying the main concepts in problems involving Newton–type methods. We use the following notation throughout this work. The n-dimensional Euclidean space is denoted by $\mathbb{R}^n$. A vector $\mathbf{x} \in \mathbb{R}^n$ is to be understood as a column vector. The vector $\mathbf{x}_* \in \mathbb{R}^n$ denotes a solution, and $\{\mathbf{x}_k\}, k = 0, 1, \ldots, M$ is a sequence of iterates. The i-th component of the vector $\mathbf{x}_k$ is denoted by $\mathbf{x}_{k,i}$. The gradient of the function $f(\mathbf{x})$ is denoted by $\nabla f(\mathbf{x}) = (\partial f/\partial \mathbf{x}_1, \ldots, \partial f/\partial \mathbf{x}_n)^T$. The Hessian of $f(\mathbf{x})$ is the matrix $Hf(\mathbf{x}) = (\partial^2 f/\partial \mathbf{x}_i \partial \mathbf{x}_j)_{i,j}$ and the Jacobian of the function $F(\mathbf{x})$ is the matrix $JF(\mathbf{x}) = (\partial F_i/\partial \mathbf{x}_j)_{i,j}$, where $F_i(\mathbf{x})$ are the i-th component functions of $F(\mathbf{x})$. We use the Euclidean norm: $\|\mathbf{x}\|^2 = \sum_{i=1}^{n} \mathbf{x}_i^2$.

We consider three classes of nonlinear problems that appear in many applications of the real–world.

- The nonlinear equations problem or **NE**, which comprises to find $\mathbf{x}_*$ such that the vector-value function F of n variables satisfies $F(\mathbf{x}_*) = 0$.

- The unconstrained optimisation problem or **UO**, which involves to find $\mathbf{x}_*$ such that the real-value function f of n variables satisfies $f(\mathbf{x}_*) \leq f(\mathbf{x})$ for all $\mathbf{x}$ *close to* $\mathbf{x}_*$.

- The nonlinear least-square problem or **NLS**, which requires to find $\mathbf{x}_*$ for which $\sum_{i=1}^{m} (r_i(\mathbf{x}))^2$ is minimised, where $r_i(\mathbf{x})$ denotes the i-th component function of $R(\mathbf{x}) = (r_1(\mathbf{x}), r_2(\mathbf{x}), \ldots, r_m(\mathbf{x}))^T$, $\mathbf{x} \in \mathbb{R}^n$, $m \geq n$.

The above problems are are mathematically equivalent under reasonable hypotheses [12]. For example, the **NE** problem can be transformed into the **UO** problem by using the Euclidean norm and defining the $f : \mathbb{R}^n \to \mathbb{R}$ function as

$$f = \frac{1}{2}\|F(\mathbf{x})\|^2 \tag{1}$$

For the above problems a Newton–type method is involved, in Algorithm 1 we present the generic form of a Newton algorithm given by Kelley in [18]. where s

---

Algorithm 1 Generic Newton Algorithm

1: **while** Stopping condition is not satisfied **do**
2:      $s \leftarrow$ calculate Newton direction
3:      $\lambda \leftarrow$ calculate step length
4:      $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda s$
5: **end while**

---

is called the Newton direction and λ the step length. Different methods exist to compute these variables, for example, Newton's method, Quasi-Newton's method, Newton's method with Cholesky decomposition, method of Steepest Descent, Line Search methods, Trust Region methods, among others.

3 Methodology for patterns identification

One of the key stages when using design patterns for software development is the identification of relations between software components. In the case of scientific software development there are no explicit relations between software patterns and numerical

methods. The methodology presented in this section helps us to identify and establish relations between the abstract concepts in the problem domain and the software patterns.

The main tasks of our proposed methodology are the following:

- Find key concepts of the problem domain.

- Perform an SCV analysis to identify software components that remain invariant through different scenarios, and to identify software components that may change at run time.

- Analyse the resulting software components and their relations to identify software design problems and recognize software design patterns that may solve these problems.

The above listed tasks may be decomposed in the following steps:

1. Create a *general description of the problem* to identify key concepts of the problem domain. These concepts define the *Scope* of the SCV analysis.

2. Study the *commonalities and variabilities* of the concepts in different scenarios by applying the *Analysis Matrix approach* by Shalloway; see [37]. Variations of the concepts generally lead to different versions or implementations of software components. These variations are generally encapsulated to facilitate change and adaptation of the developed software.

3. Find the *most important concepts or participants*, and study their relations. These concepts may represent subsystems in the final software design.

4. *Identify complex relations* between concepts that may lead to software design problems.

5. Recognize *software desing patterns* thay may be applied to solve the design problems found in the previous step.

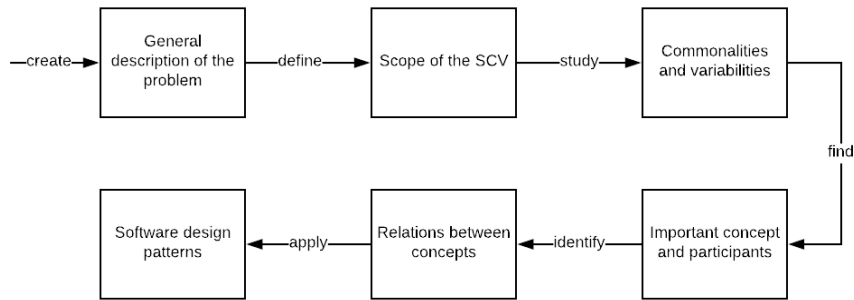In Figure 1 we highlight the key concepts of the above presented methodology.

Fig. 1: Software patterns identification methodology.

4 Software design for Newton–type methods

4.1 UML and the class diagrams

We use class diagrams from the Unified Modelling Language (UML) to present our developed software design. UML diagrams are widely used in the software development community. There are different type of diagrams, each providing specific type

of information. In our case, we are interested in the relations between the objects that compose our software design, therefore we use class diagrams which help us in visualising these relations.

A class is represented by a rectangle with the class name in the upper half of the rectangle. The lower half is used to specify methods and properties associated with the class. [2]

There are two main type of relationships between classes:

- An **is-a** relation, When one class is a *sub-type* of another class.
- A **has-a** relation, when one class *uses* or *contains* another class.

In Figure 2 we present three classes, each represented by a rectangle. The line with the white triangle connecting `Class A` and `Class C` represents an *is-a* relation. The `Class C` is a sub-type or sub-class of `Class A`. The line with the black diamond connecting `Class A` with `Class B` represents an *has-a* relation. In this case we have a *composition* relation, it means that when `Class A` is deleted then `Class B` is deleted as well. A white diamond represents an *aggregation* relation, if that would be the case then `Class A` would not be responsible for the life-cycle of `Class B`.
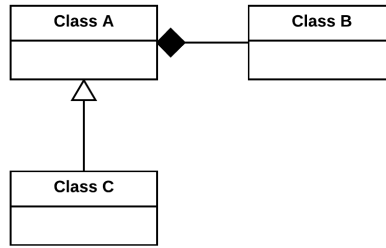

Fig. 2: A simple class diagram showing the two main relationships between classes.

4.2   Software system design foundations

At this stage, our main goal is to generate the foundations of the software design. These foundations should allow us to add and modify software components easily. A software design that supports easy incorporation and exchange of Newton–type methods will help researches when comparing the performance of these type of methods.

We start by analysing the generic form of a Newton–type method presented in Algorithm 1 and performing a top-down analysis of the iterative form of the method. Newton–type methods share the following basic sequence:

$$x_{k+1} = x_k + \lambda s, \tag{2}$$

where x_{k+1} is equal to x_k *shifted* by λs. The **step length** λ is calculated to guarantee convergence of the algorithm, and the **Newton direction** s is found with a linear or quadratic model based on Taylor series. The step length and the Newton direction are components of Newton–type methods. These components will represent subsystems in the whole system design [38]. The variations of these components in different scenarios are summarised in Table 4.

---

[2] The methods and properties of the classes in our software design are not shown to focus the attention in the relations between the classes.

Scenarios	**Concept**: Newton direction s	**Concept**: Step length λ
Line search methods	Solve $\varphi(\lambda) = f(x + \lambda s)$ with fixed x and s	Calculate an step length to guarantee convergence.
Trust region methods	Solve a linear or quadratic model within a trust ratio to find the Newton direction.	The step length is implicit in the Newton direction.
Damped methods	Solve a system of equations using a direct method to find the Newton direction.	The step length is found using Line search methods.
Quasi-Newton methods	Solve the system $Ax = b$, where A is an approximation of the first or second derivative.	Use Line search of Trust region methods to compute the step length.
Inexact or truncated methods	Use an iterative method to solve the system $Ax = b$.	Use a Line search method to calculate the step length.
Hybrid methods	Find a trajectory between a Newton direction and a gradient direction.	Use a Trust region method to compute the step length.

Tab. 4: Scope-Commonality and Variability (SCV) Analysis for Newton methods.

The variation in Newton components produces different Newton methods, for example, a *damped Newton method* is instantiated by varying the Step length and the Newton direction components as presented in Table 4.

We identified two additional components to those detailed above, the **Stopping condition**, and the **evaluation function**. Even though these two additional Newton components do not define Newton's method classes they are part of the iterative method and must be considered in the software design.

In Figure 3 we present the relations between the identified concepts and the steps of the generic form of the Newton–type method. Each component represents a subsystem. The step length component represents a subsystem specialized in the searching step lengths.

Once the problem has been stated and the SCV analysis has been performed, we proceed to identify problematic relationships and interfaces in the software design. Consider the relations presented in Figure 3 and the generic form of a Newton algorithm. Note that we can construct an specialized version of a Newton-type method by specifying the components of each step of the generic algorithm. In order to capture this structure, we apply the **Template method** pattern [15] which defines the skeleton of an algorithm and allows for variations at each of its steps. We apply the **Facade** design pattern [15] to provide a simple and general interface for the Newtons components. This facilitates the variation of Newtons components in the skeleton of the algorithm.

In order to support different implementations[3] of a single Newton component we apply the **Bridge** design pattern [15], this pattern decouples the abstraction from the implementation and lets them vary independently.

The foundation components of the software design are described in Figure 4.

In the following sections we present the software design for some of the identified Newton components.

---

[3] Consider the case that the software developer provides a debugging and release implementation that share a common interface
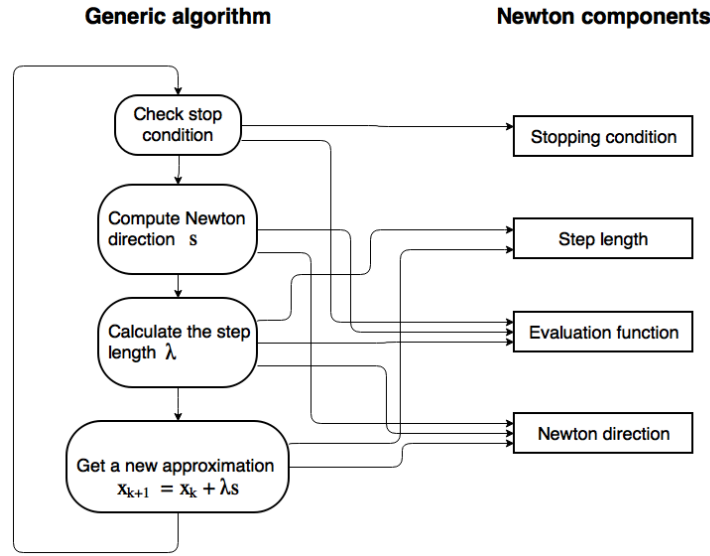
Fig. 3: Relations between Newton components and the generic form of the Newton–type method.

4.3 Software Design for Newton components

4.3.1 Nonlinear Problems and the State pattern

The numerical schemes for solving the three nonlinear problems presented in Section 2 are closely related. The nonlinear equation problem and the nonlinear least-square problem are a particular case of an unconstrained minimization problem. Given a vector-valued function $F(\mathbf{x}) = 0$ of n variables we define $f(\mathbf{x}) = (1/2)\|F(\mathbf{x})\|^2$. Finding $\mathbf{x}_*$ such that $F(\mathbf{x}_*) = 0$ is equivalent to find $\mathbf{x}_*$ such that $f(\mathbf{x}_*) = 0$. In other words, a nonlinear least-square problem may be *transformed* into an unconstrained optimisation problem. This transformation is represented as a state machine depicted in Figure 5.

In each *state* the nonlinear functions and its derivatives are handled accordingly with the nonlinear problem represented by the state.

Using the Analysis Matrix approach by Shalloway [37] and the SCV analysis we identified the concepts and their variations in different scenarios as indicated in Table 5.

Scenarios	**Concept**: Nonlinear equations problem	**Concept**: Unconstrained optimisation problem	**Concept**: Nonlinear least-square problem
Function	$F : \mathbb{R}^n \to \mathbb{R}^n$	$f : \mathbb{R}^n \to \mathbb{R}$	$F : \mathbb{R}^n \to \mathbb{R}^m, n < m,$ $f = \frac{1}{2}\|F\|^2$
First derivative	$JF(x)$	$\nabla f(x)$	$\nabla f(x) = JF(x)^T F$
Second derivative	N/A	$Hf(x)$	$Hf(x) = JF(x)^T JF(x) + \sum_{i=1}^m F_i(HF_i(x))$

Tab. 5: SCV analysis for nonlinear problems.

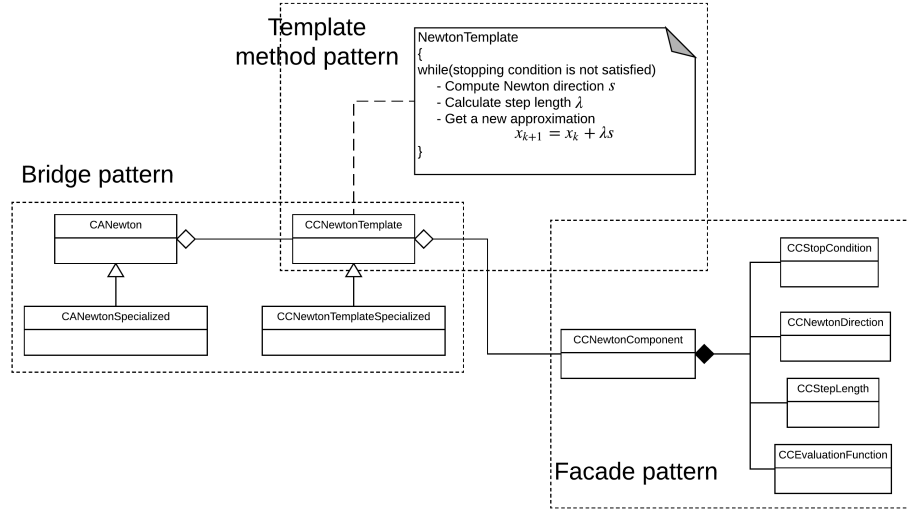Note that for a particular problem we compute either the Gradient, or the Jac-

Fig. 4: Foundations of the software design. We highlighted the application of the design patterns Template Method, Facade and Bridge in the software design. In the classes names the prefixes `CA` and `CC` indicate *abstract* and *concrete* classes, respectively.

obian matrix of the function $F(x)$, or the Hessian matrix of the function $f(x)$, *i.e.* the derivative of the function changes according to the nonlinear problem to solve. Changing the nonlinear problem involves different operations to the functions $f(x)$ or $F(x)$. The **State** design pattern [15] implements an state machine to changes the behaviour of an specific object. In our case, changing the nonlinear problem produce a transition in the state machine that switches between strategies to handle the nonlinear function and its derivatives; see Figure 6.

4.3.2 Line Search Methods and the Strategy pattern

The aim of line search methods is to find a search direction s_k and its corresponding step length λ_k. Typical strategies in line search methods test a sequence of candidate
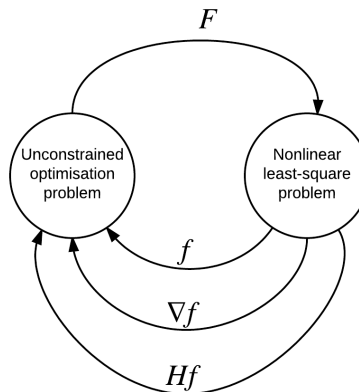

Fig. 5: Transition from an unconstrained optimisation problem to an nonlinear least-square problem, and *vice versa*.
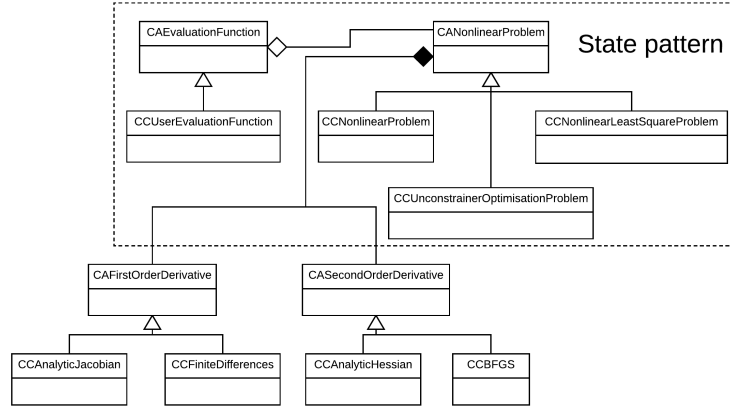
Fig. 6: Software system design implementing transitions between nonlinear problems by means of the State pattern.

values for s and stop when one of them satisfies some condition, for example when $f(x_k + \lambda_k s_k) < f(x_k)$. Popular strategies are Wolfe, Curvature and Goldstein conditions [28, 12]. These strategies involve two main tasks: the computation of the step length, and the step length decrease condition. In order to perform the SCV analysis we regard these two tasks as concepts. The variations of these concepts are presented in Table 6.

Scenario	**Concept**: Step length computation	**Concept**: Step length decreasing condition
Line Search Methods	Bisection Quadratic interpolation Cubic interpolation	Wolfe Goldstein Curvature condition

Tab. 6: Variations of the concepts in the Line Search Methods scenario.

The step length test condition is a sub-step in the computation of the step length. The test condition can vary independently of the selection of the strategy to calculate the step length. In order to allow variations in the test condition separately from the step length computation procedure we apply the **Strategy** pattern [15]. This pattern defines a family of algorithms, encapsulate them and makes them interchangeable. We apply a *double strategy*, one to encapsulate the step length test condition, and the second one to encapsulate the method for the computation of the step length, see Figure 7.

The software design presented in this section belongs to the Newton component: Step length from Figure 3.

4.3.3   Trust Region Methods and the Adapter pattern

Trust region methods construct a mathematical model to approximate the function $f(\mathbf{x})$ in a region around $\mathbf{x}_k$. The size of this region, known as the *trust regions*, is critical for the effectiveness of each step. It is necessary to find a balance between an small and a big region, therefore these methods look for a maximum trust region expansion such that the model provides a good approximation to the function $f(\mathbf{x})$. We have a constrained minimization problem as follows:
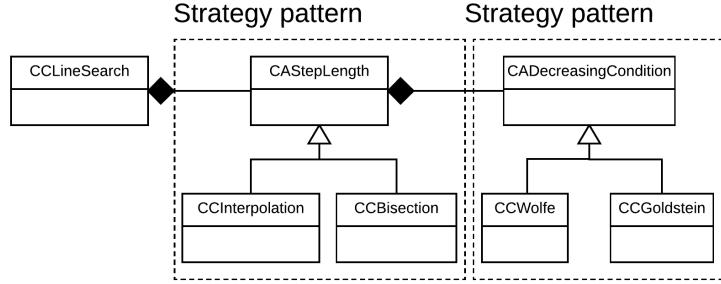
Fig. 7: Software design for Line Search Methods applying a double Strategy Pattern for the implementation of the test condition and the step length computation procedure.

$$\min_{s \in \mathbb{R}^n} m(s) = f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^T s + \frac{1}{2} s^T H f(\mathbf{x}_k) s \tag{3}$$

subject to $\|s\| \leq \Delta$, where $\Delta > 0$ is the trust region radius; see Section 2 for details on the mathematical notation.

The main tasks in trust region methods are the solution of the system of equations derived from (3), the trust region radius update, and the model–function approximation. In Table 7 we present the variations of each of the tasks or concepts identified above.

Scenario	**Concept**: Solution of system of equations	**Concept**: Update trust region radio	**Concept**: Model–function approximation
Trust region methods	Cauchy point method *Dogleg* methods	Adaptive methods using thresholds [12]	Decreasing condition in Line Search Methods
	Two–dimensional sub-space minimisation methods		

Tab. 7: SCV analysis for Trust Region Methods.

The identified concepts are highly related: a) if the trust region radius is decreased too much then the descending direction found to solve equation (3) may not be correct, b) if the solution of the equation (3) is not good enough then there might be many updates of the trust region to increase or decrease the radius, and c) the trust region update depends on how good is the approximation to the function $f(\mathbf{x})$ given by the model (model–function approximation).

We apply the **Strategy** pattern [15] to encapsulate the algorithms that solve equation (3) in a family of classes that share the same interface. The **Adapter** pattern helps us to reuse algorithms in different contexts, in this case it is applied to reuse the methods that solve nonlinear unconstrained optimisation problems to update the descending direction in equation (3), see Figure 8.

The software design presented in this section belongs to the Newton component: Newton direction from Figure 3.
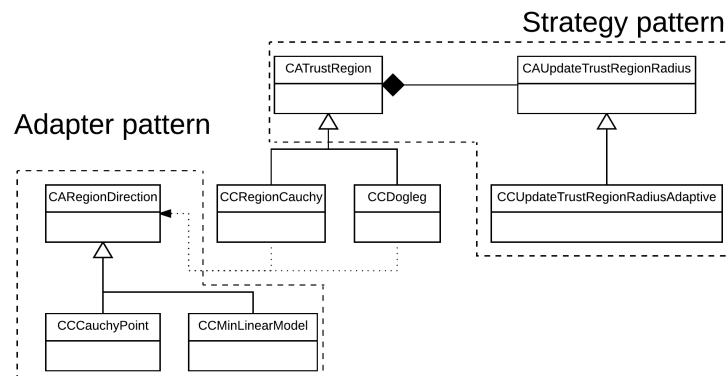
Fig. 8: Software design for Trust Region Methods applying the Strategy and the Adapter patterns.

4.4 Objects Creation, the Abstract Factory and the Singleton Patterns

In order to grant additional flexibility for the integration of new routines or algorithms we apply the Abtract Factory pattern. This pattern provides an interface to create families of objects without specifying their concrete classes [15]. A single global object implemented via the Singleton pattern [15] is used as the interface for each of the factories in the software design [3], see Figure 9.
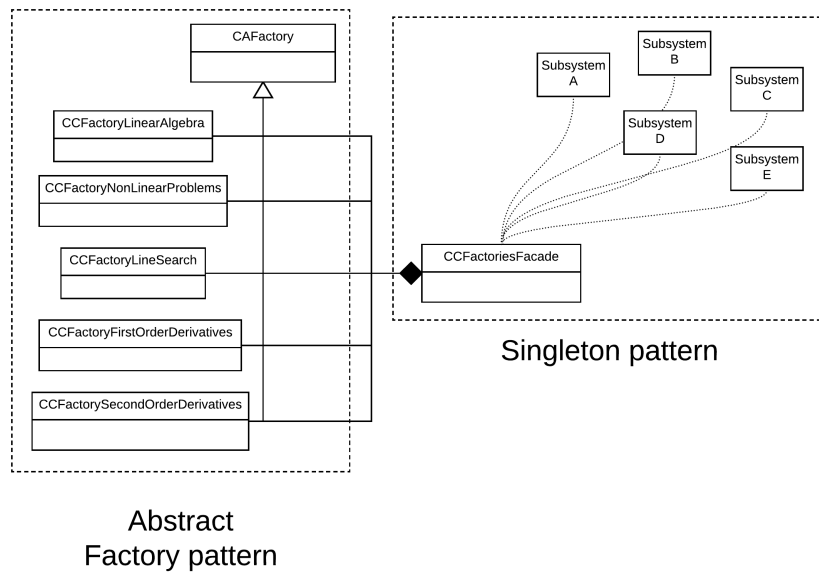

Fig. 9: Software design implementing the Abstract Factory and the Singleton design patterns. Note that the prefix `CA` and `CC` are used to identify abstract and concrete classes, respectively.

4.5   External Packages and the Adapter Pattern

Newton-type methods solve a system of equations as part of its computations, in order to include support for third-party state-of-the-art numerical software libraries we use the Adapter design pattern to convert an interface from an external package into an interface expected by our software design, [15], see Figure 10.
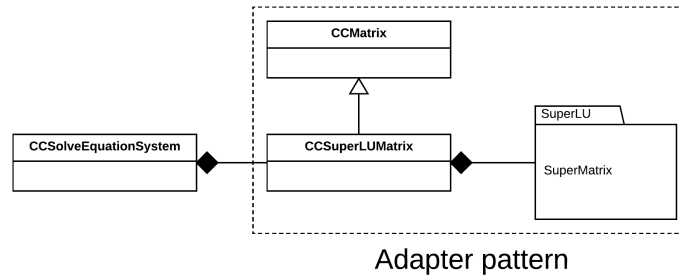

Adapter pattern

Fig. 10: The adapter pattern allows to use external numerical libraries without modifying the existing source code.

4.6   Patterns summary

In the previous section we identified and applied eight design patterns from Gamma's book [15] for the development of a software system design for Newton–type methods. In Table 8 we summarise the identified design patterns and their implications in the designed software.

Desing Pattern	General intent	Specific use
Bridge	Decouples abstraction from implementation such that both can vary independently	Facilitates the implementation of different versions of a method to target the needs of each user (debugging or production version)
Facade	Provides a unified interface to a set of interfaces	Implements a simple and unique interface for the Newton component
Adapter	Wraps an existing interface with a new one	Facilitates the use of third-party numerical software libraries. Also used to reuse Newton direction strategies from Trust Region Methods
Abstract Factory	Provides an interface to create a family of objects	Provides and interface to create and configure the objects of the software system. When combined with the Adapter pattern, they decouple the creation of external packages from the software design
Singleton	Provides a global access points for an only-one instance of a class	Simplifies the the communication with the factory objects
Template Method	Defines the skeleton of an algorithm and lets sub-methods to implement each step	Defines the general structure of the three presented Newton–type methods: nonlinear, trust region and line search
State	Allows an object to modify its behaviour based on the internal state of the object	Switch the strategies to handle nonlinear functions and its derivatives based on the problem to solve
Strategy	Defines a family of algorithms and encapsulate them to make them interchangeable	Encapsulates the step-length test condition and the computation of the step-length methods

Tab. 8: Applied design patterns.

5 Evaluation of the Software Design

In order to evaluate the quality of our software design we applied Martin's metrics [21]. These metrics consider the dependency between subsystems to compute the instability and abstractness of the software design. These measurements provide insightful information regarding the adaptation, extension and reusability capacities of the software design.

We identified 17 sub-systems or packages in our software system. For each of these packages we measured their instability and abstractness as indicated by Martin in [21].

5.1 Abstractness

The level of abstractness of a software package is given by the ratio between the number of abstract classes in the package and the total number of classes in the package, that is

$$A = \frac{N_a}{N_c}, \tag{4}$$

where N_a represents the number of abstract classes in the package and N_c represents the total number of classes in the package. When $A = 1$ then we have an abstract packages, conversely, when $A = 0$ then we have a concrete package.

The level of abstractness of a package is an indicator of its capacity for extension and reuse.

5.2 Instability

The instability of a package is given by the ratio

$$I = \frac{C_e}{C_e + C_a},\tag{5}$$

where C_e is the number of classes inside the package that depend on classes outside the package, and C_a is the number of classes outside the package that depend on classes inside the package [21].

If $C_e = 0$ then $I = 0$, therefore we have an stable package, on the contrary, if $C_a = 0$ then $I = 1$, which indicates that we have an unstable package.

5.3 The main sequence

Considering Martin's metrics, a good software design is that when $D = 0$, where $D = |A + I - 1|$. D is an indicator on the facility of a package to be adapted or extended. A package value of $D = 1$ indicates that the package is difficult to adapt or modify. In Table 9 we present the obtained values for the main packages of the developed software design.

Package name	A	I	D
TrustRegionMethods	0.29	0.80	0.09
BaseArchitecture	0.75	0.31	0.06
LineSearchMethods	0.20	0.70	0.10
NonlinearMethods	0.29	0.46	0.25

Tab. 9: Martin's metric applied to the main packages of the architecture.

We observe that $D = 0.06$ for the package BaseArchitecture, it indicates that the package is mostly abstract and has low dependency with other packages, therefore the package is easy to extend or adapt. Note that the BaseArchitecture package is part of the foundation of the software design. The other packages are also close to $D = 0$.

The D metric is better understood as a function of I and A as shown in Figure 11. The closer the points representing the packages are to the *main sequence*, the more easy to adapt or extend they are.
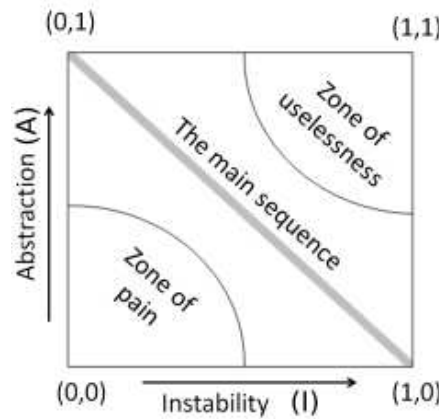

Fig. 11: The main sequence. Note that I and A should add to one in order to stay close to the main sequence.

6   Conclusions

We presented a new methodology based on a domain analysis and a Scope, Commonality and Variability analysis to transfer the knowledge of scientific experts into simple, flexible and effective object-oriented software designs. We applied the presented methodology to develop a pattern-object-oriented software design for Newton–Types methods. We evaluated the instability and the flexibility of the developed software design by means of Martin's metric. The results shown that the software design is stable enough to be extended without loss of flexibility.

Newton's method has many applications in scientific computing; in this work we used for solving optimisation problems. Another of its main uses is the approximation of solutions of systems of equations arising in simulation problems involving the finite difference method, the finite element method or the finite volume method. In this regard, the presented software design may be used as the core of an specialized software for the simulation of physical phenomena or industrial processes.

Regarding the identification of design patterns for the development of scientific and engineering software, we successfully applied eight patterns from Gamma's book [15], namely: bridge, facade, adapter, abstract factory, singleton, template method, state and strategy. In this regard, a main contribution of this work is the identification and application of the **state** pattern; after a thorough search in the relevant literature we found no reports of the application of this pattern for the development of scientific software.

As part of our future work, in order to overcome the abstraction penalty, introduced by the application of design patterns, in the code performance, we are working in the use of parallel technologies, the integration of third-party state-of-the-art software libraries and code optimisation techniques for the development of high-performance numerical software.

7   Acknowledgements

References

[1] BALAY, S., ABHYANKAR, S., ADAMS, M. F., BROWN, J., BRUNE, P., BUSCHELMAN, K., DALCIN, L., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MAY, D. A., MCINNES, L. C., MILLS, R. T., MUNSON, T., RUPP, K., SANAN, P., SMITH, B. F., ZAMPINI, S., ZHANG, H., AND ZHANG, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.9, Argonne National Laboratory, 2018.

[2] BARBIERI, D., CARDELLINI, V., FILIPPONE, S., AND ROUSON, D. Design patterns for scientific computations on sparse matrices. In *Euro-Par 2011: parallel processing workshops* (Bordeaux, France, 2012), Springer–Verlag, pp. 367–376.

[3] BARRERA, R. S. Arquitectura de Software Flexible y Genérica para Métodos del tipo Newton. Master's thesis, Instituto Nacional de Astrofísica, Óptica y Electrónica, 2011.

[4] BARRERA, R. S., GÓMEZ, G. R., HERNÁNDEZ, S. E. P., SANSALVADOR, J. C. P., AND PULIDO, L. F. Using design patterns to solve newton–type methods. In *Trends and Applications in Software Engineering: Proceedings of CIMPS 2016* (2017), J. Mejia, M. Muñoz, Á. Rocha, T. San Feliu, and A. Peña, Eds., Springer International Publishing, pp. 101–110.

[5] BLILIE, C. Patterns in scientific software: an introduction. *Compt. Sci. Eng. 4*, 3 (2002), 48–53.

[6] BLUE, J. L. Robust methods for solving systems of nonlinear equations. *SIAM J. Sci. Stat. Comput. 1*, 1 (Mar. 1980), 22–33.

[7] BOUARICHA, A., AND SCHNABEL, R. B. Algorithm 768: Tensolve: A software package for solving systems of nonlinear equations and nonlinear least-squares problems using tensor methods. *ACM Trans. Math. Softw. 23*, 2 (1997), 174–195.

[8] BRUASET, A. R. E. M., AND LANGTANGEN, H. P. Object-Oriented Design of Preconditioned Iterative Methods in Diffpack. *ACM Trans. Math. Softw. 23*, 1 (1997), 50–80.

[9] CICKOVSKI, T., MATTHEY, T., AND IZAGUIRRE, J. Design patterns for generic object-oriented scientific software. Tech. Rep. TR05-12, Departament of Computer Science and Engineering, University of Notre Dame, 2005.

[10] COPLIEN, J., HOFFMAN, D., AND WEISS, D. Commonality and variability in software engineering. *IEEE Softw. 15*, 6 (Nov. 1998), 37–45.

[11] DECYK, V. K., AND GARDNER, H. J. Object-oriented design patterns in fortran 90/95. *Comput. Phys. Commun. 178*, 8 (2008), 611–620.

[12] DENNIS, J., AND SCHNABEL, R. *Numerical methods for unconstrained optimization and nonlinear equations.* Society for Industrial Mathematics, Philadelphia, USA, 1996.

[13] DONGARRA, J., LUMSDAINE, A., NIU, X., POZO, R., AND REMINGTON, K. A sparse matrix library in c++ for high performance architectures. In *Proceedings of the 2nd Annual Object-Oriented Numerics Conference* (Oregon, USA, 1994), pp. 214–218.

[14] FILIPPONE, S., AND BUTTARI, A. Object oriented techniques for sparse matrix computations in fortran 2003. *ACM Trans. Math. Softw. 38*, 4 (2012), 23:1–23:20.

[15] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design patterns: elements of reusable object–oriented software.* Addison–Wesley, Massachusetts, Massachusetts, USA, 1995.

[16] GOCKENBACH, M. S., PETRO, M. J., AND SYMES, W. W. C++ classes for linking optimization with complex simulations. *ACM Trans. Math. Softw. 25*, 2 (1999), 191–212.

[17] HENG, B. C. P., AND MACKIE, R. I. Using design patterns in object-oriented finite element programming. *Computers and Structures 87*, 15-16 (2009), 952–961.

[18] KELLEY, C. *Iterative methods for optimization.* Society for Industrial Mathematics, Philadelphia, USA, 1999.

[19] KELLEY, C. *Solving nonlinear equations with Newton's method.* Society for Industrial Mathematics, Philadelphia, USA, 2003.

[20] LYDIA DENG, H., GOUVEIA, W., AND SCALES, J. *An object-oriented toolbox for studying optimization problems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 320–330.

[21] MARTIN, R. *Agile software development: principles, patterns, and practices.* Prentice Hall, NJ, USA, 2003.

[22] MATTHEY, T., CICKOVSKI, T., HAMPTON, S., KO, A., MA, Q., NYERGES, M., RAEDER, T., SLABACH, T., AND IZAGUIRRE, J. A. Protomol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Trans. Math. Softw. 30*, 3 (2004), 237–265.

[23] MEZA, J., OLIVA, R., HOUGH, P., AND WILLIAMS, P. Opt++: An object-oriented toolkit for nonlinear optimization. *ACM Trans. Math. Softw. 33*, 2 (2007), 12–27.

[24] MORÉ, J., GARBOW, B., AND HILLSTROM, K. User guide for minpack-i. Tech. Rep. ANL-80-74, Argonne National Laboratory, 1980.

[25] MURTAGH, B., AND SAUNDERS, M. Minos 5. 0 user's guide. Tech. rep., Systems Optimization Laboratory, Stanford University, 1983.

[26] NASH, S. G. A survey of truncated-newton methods. *Journal of Computational and Applied Mathematics 124*, 1–2 (2000), 45 – 59.

[27] NOCEDAL, J., AND WRIGHT, S. *Numerical Optimization*, 1st ed. Springer–Verlag, 1999.

[28] NOCEDAL, J., AND WRIGHT, S. *Numerical Optimization*, 2nd ed. Springer–Verlag, 2006.

[29] PADULA, A., SCOTT, S., AND SYMES, W. The standard vector library: a software framework for coupling complex simulation and optimization. Tech. Rep. TR05-12, Rice University, 2004.

[30] PÉREZ-SANSALVADOR, J., RODRÍGUEZ-GÓMEZ, G., AND POMARES-HERNÁNDEZ, S. Pattern object-oriented architecture for multirate integration methods. In *CONIELECOMP* (Puebla, México, 2011), pp. 158–163.

[31] PERNICE, M., AND WALKER, H. F. Nitsol: A newton iterative solver for nonlinear systems. *SIAM J. Sci. Comp. 19*, 1 (1998), 302.

[32] POWELL, M. J. A hybrid method for nonlinear equations. *Numerical methods for nonlinear algebraic equations 7* (1970), 87–114.

[33] RODRÍGUEZ-GÓMEZ, G., MUÑOS ARTEAGA, J., AND FERNÁNDEZ, B. Scientific software design through scientific computing patterns. In *Fourth IASTED International Conference* (Hawai, USA, 2004).

[34] ROUSON, D., XIA, J., AND XU, X. *Scientific Software Design*, 1st ed. Cambridge University Press, New York, USA, 2011.

[35] ROUSON, D. W. I., ADALSTEINSSON, H., AND XIA, J. Design patterns for multiphysics modeling in fortran 2003 and c++. *ACM Trans. Math. Softw. 37*, 1 (2010), 3.

[36] SCHNABEL, R. B., KOONATZ, J. E., AND WEISS, B. E. A modular system of algorithms for unconstrained minimization. *ACM Trans. Math. Softw. 11*, 4 (1985), 419–440.

[37] SHALLOWAY, A., AND TROTT, J. *Design Patterns Explained: A New Perspective on ObjectOriented Design (2Nd Edition) (Software Patterns Series)*. Addison-Wesley, 2002.

[38] SOMMERVILLE, I. *Software Engineering*, 8th ed. Addison-Wesley, 2004.

[39] VLISSIDES, J. *Pattern hatching: design patterns applied*. Addison-Wesley, Essex, UK, 1998.

[40] WATSON, L. T., BILLUPS, S. C., AND MORGAN, A. P. Algorithm 652: Hompack: A suite of codes for globally convergent homotopy algorithms. *ACM Trans. Math. Softw. 13*, 3 (1987), 281–310.

[41] XU, C., AND ZHANG, J. A survey of quasi-newton equations and quasi-newton methods for optimization. *Annals of Operations Research 103*, 1 (2001), 213–234.