Using Design Patterns to Solve Newton-type Methods

Ricardo Serrato Barrera[1], Gustavo Rodríguez Gómez[2], Saúl Eduardo Pomares Hernández[2], Julio César Pérez Sansalvador[3], and Leticia Flores Pulido[4],

[1] Estratei Sistemas de Información, S.A. de C.V., Virrey de Mendoza 605-B, Col. Las fuentes, 59699, Zamora, Michoacán, México
[2] Instituto Nacional de Astrofísica, Óptica y Electrónica, Coordinación de Ciencias Computacionales, Luis Enrique Erro 1, 72840, Tonantzintla, Puebla, México
[3] Instituto Nacional de Astrofísica, Óptica y Electrónica, Laboratorio de Visión por Computadora, Luis Enrique Erro 1, 72840, Tonantzintla, Puebla, México
[4] Universidad Autónoma de Tlaxcala, Facultad de Ciencias Básicas, Ingeniería y Tecnología, Calzada Apizaquito, Colonia Apizaquito, 90300, Apizaco, Tlaxcala, México
{rsbserrato, grodrig, spomares, tachidok}@ccc.inaoep.mx, leticia.florespo@udlap.mx

Abstract. We present the development of a software system for Newton-type methods via the identification and application of software design patterns. We measured the quality of our developed system and found that it is flexible, easy to use and extend due to the application of design patterns. Our newly developed system is flexible enough to be used by the numerical analyst interested in the creation of new Newton-type methods, or the engineer that applies different Newton--type strategies in his software solutions.

Keywords: Newton-type methods, design patterns, object-oriented, software design, scientific software.

1 Introduction

Newton-type methods are a family of numerical methods widely used to solve nonlinear systems of equations, unconstrained optimisation and data fitting problems. The popularity of these methods has led to the development of software packages implementing variations of them, e.g., HOMPACK [1], TENSOLVE [2] and NITSOL [3] are software packages to solve optimisation and nonlinear problems, all of them implemented in Fortran using *the procedural programming paradigm*. This approach generates complex and hard to reuse interfaces, inappropriate data structures and code that does not captures the implemented algorithms [4]. Software packages such as COOL [5], PETSC [6] and OPT++ [4] are object-oriented implementations of Newton-type methods. When the object-oriented approach is applied focusing on the details rather than on the bigger picture the produced software is commonly highly coupled, hard to modify and difficult to understand.

Software design patterns are expert solutions to software design problems. The development of scientific software lead by the application of design patterns is not common due to the difficulty on the identification and mapping of problem-specific concepts into software patterns. The relations established by the objects identified at the design stage establish the structure and quality of the final software design.

Software patterns guide us through the design stage, they expose relations between the entities that describe a problem in a particular context [7].

Currently, few works have introduced design patterns in the field of scientific software. Some have identified and proposed new patterns for the development of scientific software, [8]-[10]. Others have successfully applied design patterns for the development of scientific software, [11]-[13].

In this work we take the ideas and principles described by design patterns and apply them for the development of scientific software, we design a novel pattern object-oriented software system design for Newton-type methods. The key requirement is the flexibility of this design to permit the inclusion of new Newton-type methods. We use Martin's metric [14] to evaluate the abstractness (or generality) and the instability (or ability to reuse its existing parts) of the software system.

The remaining of this document is organised as follows: in section 2 we present the mathematical background regarding Newton-type methods and the variants considered in this work; in section 3 we present the development of the software system and the application of design patterns to solve software design problems; in section 4 we validate the newly developed software system; and in section 5 we present our main conclusions.

2 Mathematical Background

Newton's method may be considered as the standard technique used by the scientific, engineering and software development community to solve problems presenting nonlinear behaviour. We focus on three classes of nonlinear problems:

- Nonlinear equations problems involve to find x_* such that the vector-valued function F of n variables satisfies $F(x_*)=0$.
- Unconstrained optimisation problems comprise to find x_* such that the real-valued function f of n variables satisfies $f(x_*)\leq f(x)$ for all x close to x_*.
- Nonlinear least-square problems require to find x_* such that $\sum_{i=1}^{m}(r_i(x))^2$ is minimised, r_i denotes the i-th component function of $G(x)=(r_1(x),r_2(x),\ldots r_m(x)), x\in R^n, m\geq n$.

In Algorithm 1 we present the generic form of Newton's method given in [15].

Algorithm 1. Newton's method generic steps.

```
Require. Initial guess $x_0$.
1: Initialise iteration counter $k=0$.
2: while stopping condition is not satisfied do
3:    Compute Newton direction $s$.
4:    Calculate the step length $\lambda$.
5:    Get a new approximation $x_{k+1}=x_k+\lambda s$.
6:    Increase the iteration counter $k=k+1$.
7: end while
8: Return $x_k$ as the approximated solution of $x$.
```

The selection of particular strategies, such as line search or trust regions methods, to compute the Newton direction (step 3) and the step length (step 4) gives rise to specific variations of Newton-type methods; each solving an specific nonlinear problem.

In what follows we use $JF(x) = (\partial f_i / \partial x_j)_{i,j}$ to represent the Jacobian matrix of the function $F(x)$. The gradient of a function $f(x)$ is denoted by $\nabla f(x) = (\partial f / \partial x_1, \ldots, \partial f / \partial x_n)^T$, and the Hessian of $f(x)$ is the matrix $Hf(x) = (\partial^2 f(x) / \partial x_i \partial x_j)_{i,j}$.

3 Newton-type Methods Software Design

We start by decomposing the problem into sub-systems by applying the commonality and variability analysis (CVA) of Coplien [16] to identify key concepts and study their variations in different scenarios of the problem domain. Then we apply the Analysis Matrix of Shalloway [7] to determine relationships between the previously identified concepts. We study these relations to gain an indication of potential design problems and identify design patterns that may be applied to solve them. The relations found at this stage dictate the structure of the software system design.

3.1 Base System Design

The base system design is generated from the sub-systems decomposition of the generic form of Newton's method, see Algorithm 1. The studied variations of Newton-type methods have steps 3, 4 and 5 in common, what varies is their particular implementation. By applying a CVA we identify key concepts and their variations in different scenarios, see Table 1.

Table 1. CVA for Newton methods.

Scenarios	**Concept**: Newton direction s	**Concept**: Step length λ
Line search methods	N/A	Compute a step length to guarantee convergence.
Trust region methods	The Newton direction is obtained by solving the linear or quadratic model within a trust ratio.	Constrain the step length by the trust radio.
Damped methods	Solve the system $JF_k \Delta x_k = -F_k$ and find the Newton direction, where $\Delta x_k = x_{k+1} - x_k$.	Use line search methods to obtain the step length.
Quasi-Newton methods	Obtain the Newton direction by solving the system $A\Delta x_k = -F_k$, where A is a matrix representing the Jacobian or the Hessian matrix of F_k.	Obtain the step length using search or trust region methods.
Inexact methods	The Newton direction is obtained by solving the system $JF_k \Delta x_k = -F_k$ using an iterative method.	Use a line search method to obtain the step length.

Based on Algorithm 1 we identified two additional concepts from those presented in Table 1: the stopping condition and the evaluation function, see Figure 1.

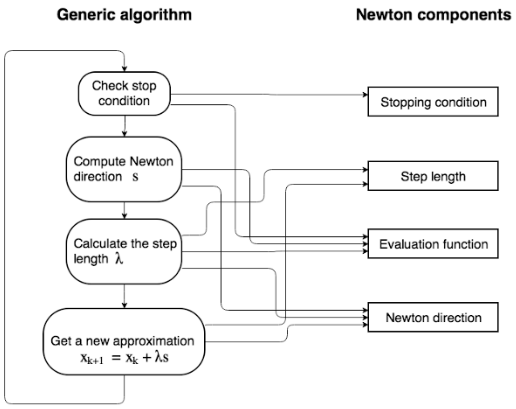

Figure. 1. Relations between the identified concepts or *Newton components* and the generic steps of Newton's method.

Each identified Newton component represents a sub-system. We observe that the Newton-type methods studied in this work share the same generic structure; an specific Newton-type versions can be created by varying the Newton components in the generic steps of Newton's method. From this statement we recognise the *template method* pattern [17] to represent the generic structure of Newton's method, and the *facade* design pattern [17] to define a simple and general interface for each of the steps or Newton components of the generic Newton algorithm steps.

Now suppose that we have two different implementations for the same Newton component, one used for development and another used for high-performance applications. In order to provide a *black-box* design where we hide the technical details of the implementation to the users we apply the *bridge* design pattern which decouples the abstraction from the implementation and allows them to vary independently, [14]. The base software system design for Newton-type methods is presented in Figure 2.

3.2 Newton's Method Sub-systems Software Design

Nonlinear methods. We observe that the three nonlinear problems presented in section 2 are particular or general cases of each other; *e.g.*, a nonlinear least-square problem is a particular case of an unconstrained optimisation problem. Consider a function $F(x)=0$ and define $f=1/2\|F\|^2$, finding an x such that $F(x_*)=0$ is equivalent to find an x_* such that $f(x_*)=0$. We represent this relation as a *transition* between different nonlinear problems; in this case from a nonlinear least-square problem to an unconstrained optimisation problem, see Figure 3.

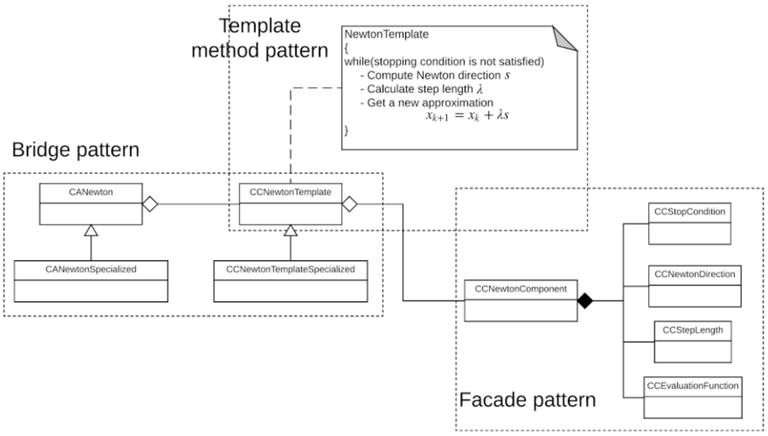

Figure. 2. Base system design for Newton-type methods showing the application of the template, facade and bridge design patterns.[1]

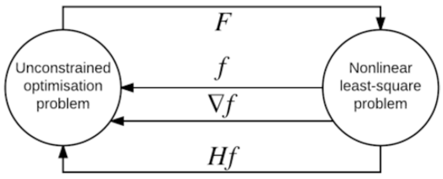
F
f
∇f
Hf
Unconstrained optimisation problem
Nonlinear least-square problem

Figure. 3. Transitions between a nonlinear least-square problem and an unconstrained optimisation problem. The nonlinear functions and its derivatives are handled accordingly to the nonlinear problem to solve.

We recognise that three studied nonlinear problems are mathematically equivalent, thus they can be reduced to the solution of an unconstrained optimisation problem. However, following this approach requires the user to perform this transformation. Our goal is to develop a software system that provides the user with the tools to treat and solve his nonlinear problem using different strategies (without him having to implement these strategies), thus he can select the one that satisfies its application requirements.

In order to identify key concepts and relations between the three studied nonlinear problems we performed a CVA, see Table 2. We observe that for a particular scenario we compute either the Jacobian, the gradient or the Hessian of the nonlinear function. We recognise and apply the *state* design pattern [17] to implement transitions between strategies to handle the nonlinear function and its derivatives. Additionally, the user may provide the analytical form of the derivative or we could approximate it via finite differences or quasi-Newton method updates, we added these strategies to the software system design defined by the state pattern, see Figure 4.

[1] We use the prefix CA and CC to indicate abstract and concrete classes, respectively.

Table 2. CVA for nonlinear problems.

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

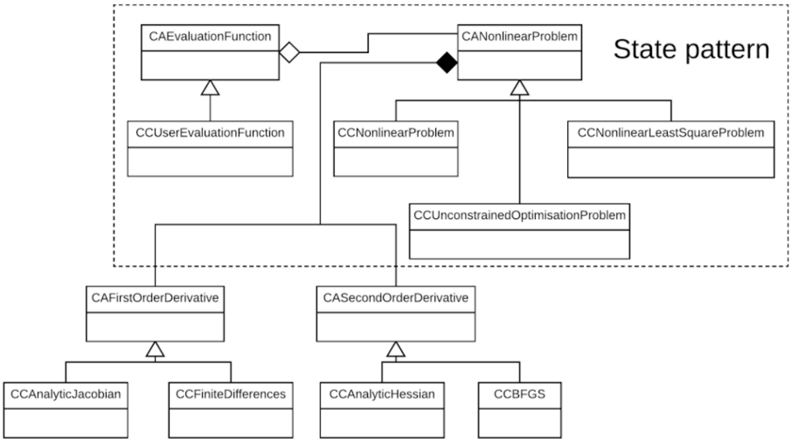

Figure. 4. Software system design implementing the state pattern to allow transitions between the studied nonlinear problems.

The software system design developed in this section correspond to the Newton component: evaluation function, presented in Figure 1.

Line Search Methods. Line search methods are strategies to find the step length λ to move along a direction s_k. The common schemes are based on bisection and interpolation. In order to determine whether the selected step length is appropriate, Wolfe, curvature and Goldstein test are applied [18], [19]. In Table 3 we present the concepts and its variations for the line search method scenario.

The *step length test condition* is applied as part of the computation of the step length, however, the test condition can vary independently of the step length approximation method. The *strategy* design pattern [17] allows us to define a family of algorithms, encapsulate them and make them interchangeable. We use a *double-strategy*, one strategy to encapsulate the step length approximation methods, and another strategy to encapsulate the decreasing condition methods, see Figure 5.

Table 3. CVA for line search methods.

Scenario	**Concept**: Step length approximation	**Concept**: Step length decreasing condition
Line search methods	Bisection Quadratic interpolation Cubic interpolation	Wolfe Goldstein Curvature condition

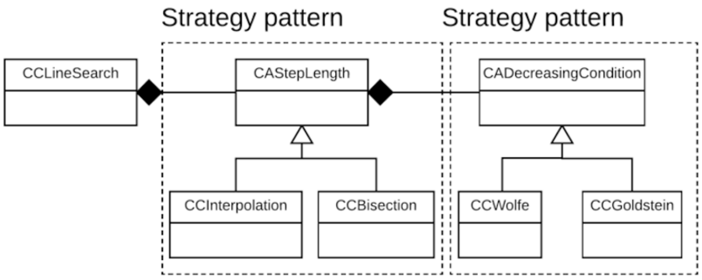

Figure. 5. A double-strategy pattern for the implementation of the selection of the step length.

The resulting software system design from this section corresponds to the Newton component: step length, presented in Figure 1.

Trust region Methods. Trust region methods are based on constructing a model to approximate a function $f(x)$ in a region around x_k. These methods can be reduced to solve a constrained minimisation problem

$$\min_{s\in R^n} m(s) = f(x_k) + \nabla f(x_k)^T s + \frac{1}{2} s^T Hf(x_k)s \tag{1}$$

such that $\| s \| < \Delta$, where $\Delta > 0$ is the trust region radius. In Table 4 we present the identified concepts and its variations associated with the trust region scenario.

Table 4. CVA for trust region methods.

Scenario	**Concept**: Solve constraint problem	**Concept**: Update trust region radius
Trust region methods	Cauchy point method *Dogleg* methods Two-dimensional sub-space minimisation methods	Adaptive methods using threshold [18]

We encapsulate the methods that solve the minimisation problem and provide them with the same interface to make them interchangeable. We apply the *strategy* pattern to handle the methods to update the trust region radius and facilitate the addition of future methods. We apply the *adapter* design pattern to reuse the methods to solve nonlinear unconstrained optimisation problems from the nonlinear methods section; this pattern adapts the interface of an object such that it can be used in different contexts, [17], see Figure 6.

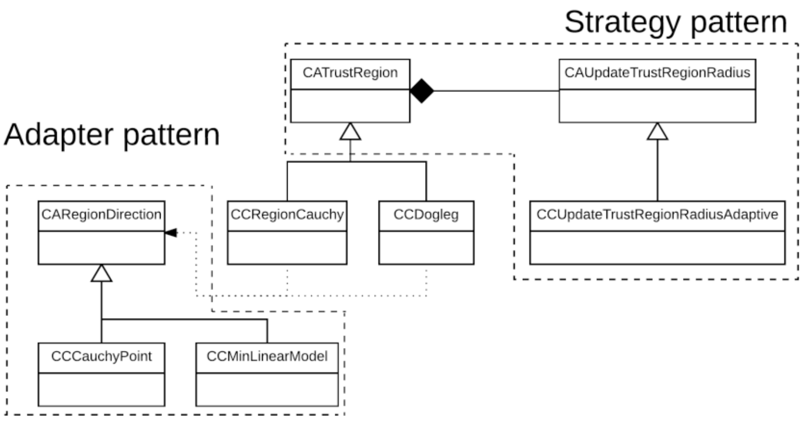

Figure. 6. Trust regions methods implemented via the strategy and the adapter pattern.

The resulting software system design from this section corresponds to the Newton component: Newton direction, presented in Figure 1.

Instantiating Objects and Interaction with External Packages. We use three more design patterns: the *abstract factory* to facilitate the creation and configuration of objects; the *singleton*, to supply a unique access point to all the factories; and the *adapter* pattern, to support the interaction with external packages for high-performance computations. The details of the application of these patterns are presented in [20] (the master's degree thesis of the first author).

4 Evaluation of the Software System Design

The developed software system design is formed by seventeen packages, see details in [20]. Martin defines in [14] the level of abstractness of a software system as

$$A = \frac{N_a}{N_c} \tag{2}$$

where N_a is the number of abstract classes in the package and N_c is the total number of classes in the package. The range of A is [0, 1]. $A = 0$ indicates a concrete package and $A = 1$ an abstract package. This metric is related to the capacity of extension of a software system, the more abstract the package the easier to extend. We also measure the instability of each package which is defined by Martin [14] as

$$I = \frac{C_e}{C_e + C_a} \tag{3}$$

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. The range of I is [0, 1]. $I = 0$ indicates an stable package and $I = 1$ an unstable package. This metric shows the ability of a package to support change. Martin combines these two metrics as $D = |A + I - 1|$, where $D = 0$ indicates a package easy to adapt or extend, and $D = 1$ a package difficult to adapt or modify, see Table 5.

Table 5. Martin's metric applied to the main packages of the architecture.

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

We observe that most of the packages are near the main sequence, in particular, the package `BaseArchitecture` has $D = 0.06$, which indicates that the main package of the system is easy to extend, reusable and does not overuse abstraction.

5 Conclusions

We have presented the development of a software design for Newton-type methods; we applied eight design patterns from the book of Gamma *et. al.* [17]. The *template method pattern* defines the generic structure of the three studies Newton-type methods, the *facade* pattern supplies a simple interface for the Newton components, the *bridge* pattern allow us to implement different versions of a method to target the interest of different users, the *state* pattern hides the details of computing the first and second order derivatives of nonlinear functions, the *strategy* pattern allows us to change algorithms to compute the step length and decreasing condition in line search methods, it also let us add new methods to update the trust region radius, the *adapter* pattern provides a medium to communicate with third-party software libraries, it also allows us to reuse the strategies to compute the Newton direction from trust region methods, the *abstract factory* pattern provides an interface to create and configure the objects of the software system, and finally, the *singleton* pattern provides a unified and single interface for the communication with the factories.

A main contribution of this work is the identification and application of the *state* pattern for the development of scientific software, to the best of these authors knowledge the identified instance of this pattern has not been reported in related works. The instability and abstractness values of this pattern are those of the *NonlinearMethods* package, the one implementing the *state* pattern. The results show that the system design is stable enough to be extended without loss of flexibility. With the design of the presented software system we demonstrate that the knowledge of the scientific expert can be exploited by the software engineer through the application of design patterns to generate simple, flexible and effective object-oriented software. As part of our future work is the application of parallel technologies, integration of third-party state-of-the-art software libraries, use of templates and code optimisation techniques for the development of high-performance numerical software.

Acknowledgments. We sincerely thank the observations of the two anonymous reviewers that helped to improve and clarify this work.

References

1. L.T. Watson, S.C. Billups and A.P. Morgan, Algorithm 652: hom-pack: a suite of codes for globally convergent homotopy algorithm, ACM Trans. Math. Soft., vol. 13, no. 3, pp. 281-310 (1987).
2. A. Bouaricha and R.B. Shnabel, Algorithm 768: tensolve: a software package for solving systems of nonlinear equations and nonlinear least-square problems using tensor methods, ACM Trans. Math. Soft., vol. 23, no. 2, pp. 174-195 (1997).
3. M. Pernice and H. F. Walker, Nitsol: a Newton iterative solver for nonlinear systems, SIAM J. Sci. Comp., vol. 19, no. 1, p. 302 (1998).
4. J. Meza, R. Oliva, P. Hough and P. Williams, Opt++: an object-oriented toolkit for nonlinear optimization, ACM Trans. Math. Soft., vol. 33, no. 2, pp. 12-27 (2007).
5. L. Deng, W. Gouveia and J. Scales, The cwp object-oriented optimization library, Center for Wave Phenomena, Technical report (1994).
6. S. Balay, W. Gropp, L. McInnes and B. Smith, Petsc 2.0 users manual, Argonne National Laboratory, Technical report (1995).
7. A. Shalloway and J. Trott, Design Patterns Explained: A New Perspective on Object-Oriented Design (2Nd Edition) (Software Patterns Series). Addison-Wesley (2002).
8. C. Blilie, Patterns in scientific software: an introduction, Compt. Sci. Eng., vol. 4. no. 3, pp. 48-53, 2002.
9. G. Rodríguez-Gómez, J. Muños-Arteaga and B. Fernández, Scientific software design through scientific computing patterns, in Fourth IASTED International Conference, Hawai, USA, 2004.
10. T. Cickovski, T. Matthey and J. Izaguirre, Design patterns for generic object-oriented scientific software, Department of Computer Science and Engineering, University of Notre Dame, Technical report, TR05-12 (2005).
11. V. K. Decyk and H. J. Gardner, Object-oriented design patterns in Fortran 90/95, Comput. Phys. Commun., vol. 178, no. 8, pp. 611-620 (2008).
12. J. Pérez-Sansalvador, G. Rodríguez-Gómez and S. Pomares-Hernández, Pattern object-oriented architecture for multirate integration methods. In CONIELECOMP, Puebla, Mexico, 2011, pp., 158-163.
13. D. Rouson, J. Xia and X. Xu, Scientific Software Design, 1st. New York, USA: Cambridge University Press, 2011.
14. R. Martin, Agile software development: principles, patterns, and practices. NJ, USA: Prentice Hall (2003).
15. C. Kelley, Iterative methods for optimization, Philadelphia, USA: SIAM (1999).
16. J. Coplien, D. Hoffman and D. Weiss, Commonality and variability in software engineering, IEEE Software, vol. 15, no. 6, pp. 37-45 (1998).
17. E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: elements of reusable object-oriented software. Massachusetts, USA: Addison-Wesley, Massachusetts (1995).
18. J. Dennis and R. Schnabel, Numerical methods for unconstrained optimization and nonlinear equations, Philadelphia, USA: SIAM (1996).
19. J. Nocedal and S. Wright, Numerical optimization, 2nd. Springer-Verlag (2006).
20. R. S. Barrera, Arquitectura de Software Flexible y Genérica para Métodos del tipo Newton, Master's thesis, Instituto Nacional de Astrofísica, Óptica y Electrónica, Mexico (2011).