ELSEVIER

Mathematics and Computers in Simulation 36 (1994) 401–411

MATHEMATICS AND COMPUTERS IN SIMULATION

Intelligent object-oriented scientific computation

Annie Cuyt [a,*], Brigitte Verdonk [a,1], Jan Verelst [b]

[a] *University of Antwerp (UIA), Universiteitsplein 1, B-2610 Antwerp, Belgium*

[b] *University of Antwerp (RUCA), Middelheimlaan 1, B-2020 Antwerp, Belgium*

Abstract

Our goal is the integration of accurate scientific computation and intelligent object-oriented techniques to obtain powerful tools that can be used for the successful development of scientific expert systems, i.e. systems which provide expertise in the choice of algorithms within a particular problem domain and generate reliable, robust and maximally accurate results of full integrity. The need for such scientific expert systems is increasing significantly [1,4,11,12,15,20,21]. Through this study we want to identify some general principles which must be taken into account both concerning the computational aspects, such as automatic validation of the results (see Section 1), and aspects of programming methodology, especially with respect to classification and inheritance (see Section 2). The usefulness and power of such intelligent object-oriented scientific expert systems will be tested on a prototype which will be developed for the problem domain of multivariate rational data fitting [8]. A more detailed description of this prototype is given in Section 3.

Keywords: Object-orientation; Scientific computation; Expert systems; Validation

1. Scientific computation

Although computers were historically developed for scientific computation, extensive interest in the subject of computer arithmetic has mostly developed in the early eighties, yielding some important improvements in the accuracy of floating-point computations. Two IEEE floating-point standards were agreed upon [13] in order to formalize the representation of floating-point numbers and the implementation of the basic floating-point operations. However, many computations in numerical algorithms are carried out in product spaces (vectors, matrices, ...). The arithmetic operations in these spaces are traditionally performed in terms of the given elementary floating-point operations. It is well known that the computational error due to the accumulation of rounding errors of each of the basic floating-point operations can become quite large. To overcome this accumulation of rounding

* Corresponding author. Research Director NFWO. Email: cuyt@wins.uia.ac.be.

[1] Post-Doctoral Researcher NFWO.

0378-4754/94/$07.00
SSDI 0378-4754(93)E0038-7

errors, a new definition of the operations in the product spaces is given in [19]. It is proven there that to implement the operations in all the product spaces according to the new definition, the set of basic operations in the IEEE standard has to be extended with the scalar product for vectors.

The advantages of highly accurate computations can be combined with interval arithmetic to obtain self-validating numerical methods. Interval arithmetic is the only computational tool so far available that incorporates guarantees as part of the basic computational process. Interval arithmetic which has been around for a long time has often been criticized since its naive use may deliver bounds which are unreasonably large and thus do not contain much information about the solution of the problem. If one combines interval computation, the process of defect correction [22] and the optimal scalar product this criticism is superseded. In this approach the role of the optimal scalar product is again crucial.

As a consequence of the above, different programming languages have been extended and libraries developed to include the optimal scalar product as a basic operation and to provide tools for interval arithmetic, computation in product spaces, etc. We refer among others to Pascal-XSC (eXtension for Scientific Computation) [17], C-XSC [18], Acrith-XSC [2], Modula-SC [9], Fortran-SC [10]. Using these extended programming languages, problem solving routines with automatic result verification can and have been implemented [18,25]. It is our goal when developing a scientific expert system to complement automatic result verification tools with the possibility to prescribe the accuracy required for the output of a selected algorithm. This means that for the selected algorithm the precision t used in the floating-point arithmetic has to be determined in terms of the prescribed accuracy of r bits for the output. This step is called the *prerun* of the algorithm. Some initial ideas on this problem were written down in [14]. After the prerun, the algorithm is rerun using t digits of precision, yielding the desired output accuracy. In order to achieve this goal it is clear that the existing tools for scientific computation need to be extended with tools for higher dynamic accuracy.

2. Intelligent object-oriented techniques

There are several concepts and principles underlying object-oriented programming: the concept of class or abstract data type, inheritance, overloading which is closely linked with the principle of dynamic binding and many others. We have mentioned these three since it is mainly these which will be put to good use when implementing a scientific expert system [26,8]. For a specific numerical problem domain, usually different problem types can be identified, and with each problem type appropriate algorithms can be associated. Consider for example the problem domain of linear systems of equations where different types of problems can be identified depending on the structure of the matrix. For general square $n \times n$ matrices different algorithms of order n^3 exist while for Toeplitz $n \times n$ matrices more specialized algorithms of order n^2 have been developed [24,16]. Translating the classification of a problem domain in an object-oriented programming language can easily be done. Indeed, each problem type can be implemented as a class, where the algorithms to solve that type of problem are encapsulated within the class. As such, each class certainly defines a method to "solve". The implementation of the *solve*-method in the class C_1 is the algorithm to solve the problem of type C_1, whereas the implementation of the *solve*-method in class C_2 will be the algorithm to solve problems of type C_2. This classification and the partial order between the different problem classes is closely related to the particular problem domain. The overloading of the method name *solve* can be

resolved through dynamic binding. However, for numerical computations the default dynamic binding mechanism must be carefully examined. Indeed, it may be the case that although a particular problem belongs to class C_1 the *solve*-method in a superclass of C_1, instead of in C_1 itself, should be applied for reasons of stability and reliability. In traditional object-oriented programming languages all local methods have a priority higher than that of methods inherited from superclasses. In a scientific expert system the overloading of methods should be conditional on the performance of the algorithms for the particular data.

So far we have only touched upon the roundoff error which results from using floating-point arithmetic. However, in different problem domains, *solve*-methods suffer from a process error, the error incurred because the *solve*-algorithm is only an approximate solution. For such process errors theoretical error bounds are often known. By incorporating process error estimates in the *solve*-method when available, the principle of self-validation referred to in Section 1 is augmented to include not only the arithmetic roundoff error but also the process error. Whereas the former provides the user automatically with information about how well- or ill-conditioned his/her problem is, the latter provides him/her automatically with information on the power of the *solve*-method. Object-orientation can be put to good use to include process error information because of the encapsulation it provides.

Last but not least, as is well known, the principles of object-orientation also allow to improve the programming methodology. Indeed, if more optimal algorithms or data structures can be developed for a particular class, only the implementation within that class needs to be modified, while the rest of the program remains unchanged.

3. Prototype scientific expert system

In order to demonstrate the power of such intelligent object-oriented scientific expert systems, a prototype is being developed. It consists of two main building blocks. Firstly, a generic knowledge based front end (KBFE), capable of assisting users in selecting and applying the most appropriate algorithm for the scientific problem they are trying to solve. Secondly, a numerical library for a particular problem domain, consisting of numerical routines that are equipped with tools for self-validation and automatic process error control. Information flow from the numerical library to the generic KBFE is provided through a proper interface which is described below. We stress that the front end is generic and can be used with numerical libraries for any problem domain, taking into account that a proper interface to the KBFE is provided. Due to time limitations, however, only one library is currently being implemented for the problem domain of multivariate rational data fitting [8].

It is our goal to combine in the prototype tools for extended scientific computation with the power of object-orientation. How does this compare to existing systems like Focus [4], NAExpert [21], Nexus [11], etc.? Essentially, these systems have a structure as illustrated in Fig. 1(a), where a collection of numerical routines is linked to a knowledge based front end serving as intelligent routine selector, in this way obtaining an expert system for numerical software.

The aim of our system is to extend this technology in two essential ways. The first, as depicted in Fig. 1(b), is to upgrade existing numerical libraries or develop new ones according to the XSC-philosophy. This implies taking the computer arithmetic a step further than the traditional IEEE 754

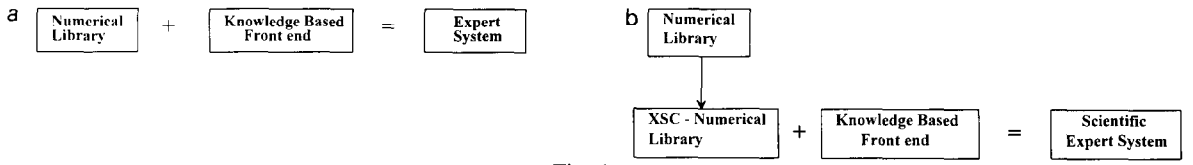

Fig. 1.

arithmetic and including tools for dynamic precision and automatic validation. In essence, we want to let the user prescribe the output accuracy rather than letting the system decide on the reliability of the result.

A second important issue is that we want to develop the system in such a way as to take full advantage of object-oriented technology, even enhancing this technology with extra features for the purpose of numerical reliability and stability as described above.

The practical realization of the prototype can be pictured as follows. The expertise is presented to the expert system in the form of a collection of problem domain specific numerical routines (we chose for C++), which are structured in a class hierarchy. Starting from these routines, the front end

(1) deduces the class hierarchy from the source file;

(2) constructs a knowledge base containing information about the solve routines in order to be able to select the most appropriate routine for a given problem.

Note that, because of the way the expertise is presented to the front end, the knowledge base is constructed dynamically. The front end learns from the expertise that is presented to it and adapts its knowledge base whenever the expertise is upgraded.

Once the expertise is loaded into the KBFE, the scientific expert system is ready for use. During a consultation, it

(3) displays the problem domain tree and prompts for problem data;

(4) classifies the user's problem data by performing the necessary tests;

(5) selects the most appropriate routine to solve the user's problem, taking into account possible unstable behaviour of the suggested numerical procedure or large perturbations in the input data, and sometimes overrules the classical inheritance mechanism by choosing a numerical technique of a superclass instead of a more specific class;

(6) builds the call to the selected numerical procedure, prompting for additional data if necessary;

(7) provides a (sort of natural language) user interface prompting for lacking information or input and explaining decisions upon request.

In the following paragraphs, we will go into more detail on each of these points, using illustrations from several problem domains, thereby emphasizing once more that the front end is generic. To start off, we explain how the front end gets information from the plugged in numerical library and where it stores this information for later use.

3.1. Deduce the class hierarchy from the library source files

First of all, the front end needs to store the structure of the problem domain tree, which, for the problem domain of multivariate rational interpolation, is given in Fig. 2 [5–8].

The front end extracts this structure from the source files of the library. To store this information,

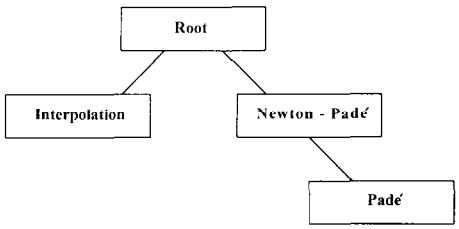

Fig. 2.

a C^{++} template class named *Tree* has been implemented, possessing the standard methods a tree has, such as *addNode*, *goUp*, *goDown*, *searchNode* and so on. Because it is a template, the nodes of *Tree* can be objects of any other C^{++} class. Each time a problem domain specific library is presented to the KBFE, the front end creates an instance of the template class *Tree*, called *KBDomainTree*. For each problem type in the problem domain, a node is created in the *KBDomainTree* and these nodes are connected in such a way that the *KBDomainTree* and the problem domain tree have an identical structure. Each node of the *KBDomainTree* will store information on the corresponding problem type, as described next.

3.2. Construct the knowledge base

In order to be able to assist the user in selecting the most appropriate routine for his/her problem, the front end needs complete information on the implementation of each problem type in the library. This information is best mastered by the library programmer who has developed the numerical library and possesses expertise on the problem domain. In order to make optimal use of this expertise, the library programmer must make it explicitly available to the KBFE. To this end, the programmer uses a specially developed command language which provides the interface between the library and the front end. Using this command language it is also easy to insert updated libraries into the front end. In the following we do not aim at giving a full description of the command language but we will rather illustrate the philosophy behind it. A separate paper will be devoted to the description of the command language.

For each problem type, the library programmer has to add a C^{++} comment to the library source file, containing the command language statements applicable to that class. Fig. 3 contains the command language interface for the different classes in the rational data fitting problem domain as displayed in Fig. 2.

A more detailed explanation of the commands of our command language will be given hereafter. We remark that the overloading of the method name *solve* referred to in Section 2 occurs here for the methods *coef_problem*, *rec_algor* and *qdg_algor*. For simplicity one can concentrate in the sequel of the text on the method *coef_problem* from the above example, which is defined for each class in the problem domain. Its implementation will vary from class to class, depending on the complexity of the problem for that particular class. Afterwards we will briefly discuss how the expert system deals with the full rational interpolation numerical library, where besides *coef_problem* other aspects of the rational interpolation problem are dealt with by different *solve*-methods, i.e. *rec_algor* and *qdg_algor*.

```
/*
.class root
.label Root
.input point_input_system
.beginsolve
.menu -n1
.label 1 "numerator and denominator coefficients"
.input -n2 1
  "numerator degree " int n_test
  "denominator degree " int d_test
.solve 1 coef_problem(int, int)
.endsolve
*/
/*
.class newton_pade
.label Newton-Pade
.is_instance newton_pade_test
.beginsolve
.menu -n3
.label 1 "numerator and denominator coefficients"
.input -n2 1
  "numerator degree " int root.n_test
  "denominator degree " int root.d_test
.solve 1 coef_problem(int, int)
.label 2 "recursive ε-like algorithm"
.input -n4 2
  "maximal numerator degree " int root.n_test
  "maximal denominator degree " int root.d_test
  "x-coord of point where approximation requested " double
  "y-coord of point where approximation requested " double
.solve 2 rec_algor(int, int, double, double)
.label 3 "continued fraction-based algorithm"
.input -n4 3
  "numerator degree " int root.n_test
  "denominator degree " int root.d_test
  "x-coord of point where approximation requested " double
  "y-coord of point where approximation requested " double
.solve 3 qdg_algor(int, int, double, double)
.endsolve
*/
/*
.class rational_hermite
.label Interpolation
.is_instance rational_hermite_test
.beginsolve
.menu -n3
.label 1 "numerator and denominator coefficients"
.input -n2 1
```

Fig. 3.

```
  "numerator degree " int root.n_test
  "denominator degree " int root.d_test
.solve 1 coef_problem(int, int)
.label 2 "recursive Bulirsch-Stoer-like algorithm"
.input -n4 2
  "maximal numerator degree " int root.n_test
  "maximal denominator degree " int root.d_test
  "x-coord of point where approximation requested " double
  "y-coord of point where approximation requested " double
.solve 2 rec_algor(int, int, double, double)
.label 3 "Thiele-type continued fraction algorithm"
.input -n4 3
  "numerator degree " int root.n_test
  "denominator degree " int root.d_test
  "x-coord of point where approximation requested " double
  "y-coord of point where approximation requested " double
.solve 3 thiele_algor(int, int, double, double)
.endsolve
*/
/*
.class pade
.label Pade
.is_instance pade_test
.beginsolve
.menu -n3
.label 1 "numerator and denominator coefficients"
.input -n2 1
  "numerator degree " int root.n_test
  "denominator degree " int root.d_test
.solve 1 coef_problem(int, int)
.process_error 1 pade_process_error(int, int)
.label 2 "recursive ε-like algorithm"
.input -n4 2
  "maximal numerator degree " int root.n_test
  "maximal denominator degree " int root.d_test
  "x-coord of point where approximation requested " double
  "y-coord of point where approximation requested " double
.solve 2 rec_algor(int, int, double, double)
.label 3 "continued fraction-based algorithm"
.input -n4 3
  "numerator degree " int root.n_test
  "denominator degree " int root.d_test
  "x-coord of point where approximation requested " double
  "y-coord of point where approximation requested " double
.solve 3 qdg_algor(int, int, double, double)
.endsolve
*/
```

Fig. 3. (Continued.)

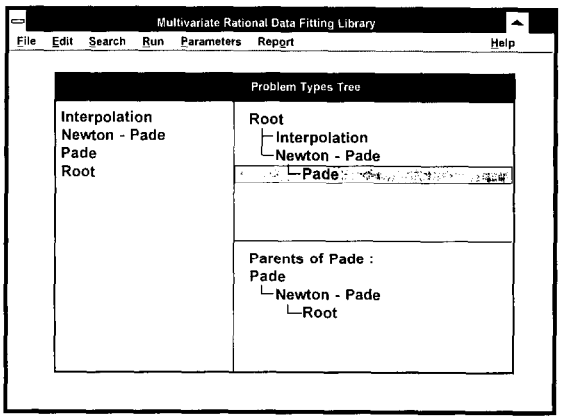

Fig. 4.

3.3. Display the tree and prompt for the problem data

When the expert system is run, the opening screen will display the problem domain tree and a menu bar. Fig. 4 shows the opening screen for the multivariate rational data fitting library.

In the left part of the window the classes are listed in alphabetical order while in the right top part the problem domain tree is displayed. If the user selects a particular class in the problem domain, the right bottom part of the window is used to display the ancestors of that class (class `Pade` in Fig. 4). The display of ancestor information can be especially useful in case of multiple inheritance.

Next the front end processes the user's problem data. As the format and amount of input is problem domain specific, the input routines of the problem domain have to be implemented by the library programmer. The front end is informed of the presence of an input routine by the `.input` command of the command language (see Fig. 3). In the case of the rational data fitting problem domain, a single graphical input routine was written and attached to the `Root` problem type, which is also the root of the problem domain tree. However, other libraries may have multiple input routines, each attached to a different class, in order to supply each corresponding problem type with an input routine that takes full advantage of the typical input data structure inherent to the problem type. These input routines can drastically lower the number of actions the user has to perform to input the data. For instance, when inputting a linear system of equations, the user can select a special input routine for symmetric or band-structured matrices instead of having to use the input routine for general matrices.

3.4. Classify the problem data

After the data has been input, the front end determines of which classes in the problem domain the data is an instance. This is achieved as follows. Each class in the mathematical library is assumed to have a member function implementing an *is_instance* test. This *is_instance* test determines whether the data are an instance of a particular problem type class, based on the structure of the data. The test also returns to the front end additional information on the structure of the data in the form of keyword strings. Via these keywords, the user can obtain a more detailed explanation of decisions from the KBFE's help file, to which the keywords are hyperlinked.

As illustrated in Fig. 3, the front end retrieves the necessary information on the member functions

implementing the *is_instance* tests via the `.is_instance` command of the command language interface. Using this information, the front end traverses the *KBDomainTree* top down, performing the *is_instance* test of each encountered class and eventually classifies the data in the most specific class for which the *is_instance* test succeeds. We note that this class will normally have the solve routine with least complexity for the particular data because this solve routine is allowed to make the most powerful assumptions about the structure of the data.

3.5. Select an appropriate solve routine

Once the data are classified in a particular class C as described above, the solve routine from this class can be applied to solve the user's problem. However, it may be the case that for reasons of stability and reliability, the solve method of a superclass of C instead of the one of the class C itself should be applied, as was described in Section 2. Therefore, a more intelligent dynamic binding mechanism than the standard object-oriented strategy, where more specific methods always have higher priority, needs to be implemented. The front end implements such a numerically reliable dynamic binding mechanism as follows. The most specific numerical routine is first prerun. The output of the prerun is used to check for the applicability of the method, taking into account the accuracy of the data. If the most specific method cannot yield the accuracy required, the front end will automatically check if superclass methods can be applied to meet the reliability requirements.

So far we have only considered the case where each problem type contains a single solve routine. However, it may be the case that a particular problem type contains different solve routines, each solving a different aspect of the problem. For example, for the problem domain of multivariate rational interpolation one may be interested in either the numerator and denominator coefficients of the approximating rational function, or the values of recursively computed rational approximants in a particular point different from the interpolation points. In such a case the command language interface needs to be more elaborate. This is illustrated in Fig. 3, where the `.solve` information for all classes except the `Root` class is much more elaborate because for all these classes different algorithms, which solve different aspects of the problem, exist. The `.menu` command specifies the number of different problem aspects and the user will be prompted for the aspect of the problem he wants to deal with via a menu displaying the strings in the `.label` command. This will enable the front end to automatically prerun and activate the corresponding numerical routine.

Lastly, it may be the case that the same aspect of a particular problem type can be solved using different algorithms. In this case it is recommended that the library programmer uses the `.prerunpriority` slot to specify, by using a set of rules, the order in which the preruns should be done.

3.6. Build the call to the selected solve routine

When building the call to the selected solve routine, the front end prompts the user for the necessary solve function arguments. Information on the name and type of these arguments was provided to the front end through the command language. For example in Fig. 3, the first `.input` command in class `Newton_Pade` specifies that the user must be prompted for two integer arguments, namely the numerator and denominator degree of the rational function.

3.7. Implementation issues

The prototype is being implemented on an IBM PC using the Borland C^{++} version 3.1 compiler and the Turbo Vision for DOS interface library. In the development phase of the prototype transfer of the system to an X-windows environment is kept in mind. The user interface makes elaborate use of windows, dialog boxes, keywords, a context-sensitive hyperlink helpfile, a log book and so on. It also features numerous functions to display and edit the ASCII files that the front end uses. We decided to use C^{++} because of its object-oriented features and because in this way the front end can directly be linked to a numerical library which is either precompiled from Pascal-XSC to C code, or itself written in C^{++} and linked to the C-XSC library or provided directly through our own dynamic precision implementation in C^{++}, which is currently under development.

References

[1] H. Abelson, M. Eisenberg, M. Halfant, J. Katzenelson, E. Sacks, G. Sussman, J. Wisdom and K. Yip, Intelligence in scientific computing, Comm. ACM 32 (1989) 546–561.
[2] Acrith-XSC: IBM High Accuracy Arithmetic - Extended Scientific Computation. Version 1, Release 1, IBM Deutschland GmbH, Department 3282, Böblingen, 1990.
[3] L. Atanassova and J. Herzberger, eds., Computer Arithmetic and Enclosure Methods (Elsevier, Amsterdam, 1992).
[4] C.W. Cryer, The ESPRIT project FOCUS, in: P.W. Gaffney and E.N. Houstis, eds., Programming Environments for High-Level Scientific Problem Solving (Elsevier, Amsterdam, 1992) 371–380.
[5] A. Cuyt, Padé approximation in one and more variables, in: [23] 41–68.
[6] A. Cuyt, Rational Hermite interpolation in one and more variables, in: [23] 69–104.
[7] A. Cuyt and B. Verdonk, Rational interpolation on general data sets in $\mathbb{C}^n$, in: C. Brezinski, ed., Numerical and Applied Mathematics (Baltzer, Basel, 1989) 415–429.
[8] A. Cuyt and B. Verdonk, Multivariate rational data fitting: general data structure, maximal accuracy and object-orientation, Numer. Algorithms 3 (1992) 159–172.
[9] C. Falcó Korn, S. Gutzwiller, S. König and C. Ullrich, Modula-SC motivation, language definition and implementation, IMACS Ann. Comput. Appl. Math. 12 (Baltzer, Basel, 1992) 161–179.
[10] FORTRAN for scientific computation, Language description and sample programs, Institute for Applied Mathematics, University of Karlsruhe, 1988.
[11] P.W. Gaffney, C.A. Addison, B. Andersen, S. Bjørnestad, R.E. England, P.M. Hanson, R. Pickering and M.G Thomason, NEXUS, Towards a problem solving environment (PSE) for scientific computing, ACM SIGNUM Newsletter 21 (1986) 13–24.
[12] E. Gallopoulos, E. Houstis and J.R. Rice, Future research directions in problem solving environments for computational science, Report Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry and Artificial Intelligence for Computational Science, Washington, DC, 1991.
[13] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Comp. Surv. 23 (1991) 5–48.
[14] R. Hammer, Pascal-XSC: From accurate expressions to the accurate evaluation of program parts, in: [3] 119–128.
[15] E. Huber, C. Kulikowski, J.M. David and J.P. Krivine, eds., Artificial Intelligence in Scientific Computation, IMACS Ann. Comput. Appl. Math. 2 (Baltzer, Basel, 1989).
[16] T. Kailath, S.-Y. Kung and M. Morf, Displacement ranks of matrices and linear equations, J. Math. Anal. Appl. 68 (1979) 395–407.
[17] R. Klatte, U. Kulisch, M. Neaga, D. Ratz and Ch. Ullrich, Pascal-XSC Language Reference (Springer, Berlin, 1992).
[18] R. Klatte, U. Kulisch, A. Wiethoff, C. Lawo and M. Rauch, C-XSC, A C^{++} Class Library for Extended Scientific Computing (Springer, Berlin, 1993).
[19] U. Kulisch and W. Miranker, Computer Arithmetic in Theory and Practice (Academic Press, New York, 1981).

[20] J.R. Rice, Mathematical aspects of scientific software, in: J.R. Rice, Ed., Mathematical Aspects of Scientific Software, IMA Volumes Math. Appl. 14 (Springer, New York, 1988) 1–40.
[21] K. Schulze and C.W. Cryer, NAExpert: a prototype expert system for numerical software, SIAM J. Sci. Stat. Comput. 9 (1988) 503–515.
[22] G. Schumacher and J. Wolff von Gudenberg, Highly accurate numerical algorithms, in: [25] 1–58.
[23] S.P. Singh, ed., Approximation Theory, Spline Functions and Applications, NATO ASI Series (Kluwer Academic Publishers, Dordrecht, 1992).
[24] W. Trench, An algorithm for the inversion of finite Toeplitz matrices, SIAM J. Appl. Math. 12 (1964) 515–522.
[25] Ch. Ullrich and J. Wolff von Gudenberg, eds., Accurate Numerical Algorithms: A Collection of Research Papers (Springer, Berlin, 1989).
[26] J. Wolff von Gudenberg, Object-Oriented Concepts for Scientific Computation, IMACS Ann. Comput. Appl. Math. 12 (Baltzer, Basel, 1992) 181–192.