Exploration of the Suitability of O-O Techniques for the Design and Implementation of a Numeric Math Library using Eiffel

Diploma Thesis

Peter Häfliger

advised by Prof. Dr. Bertrand Meyer and Susanne Cech

ETH Zürich

March 9, 2003

Abstract

This diploma thesis is organized in three parts. The first part defines a problem which Bertrand Meyer calls the 'Scientific Software Paradox' and shows its historic evolution. The creation of an object-oriented mathematics library is proposed as a possible path out of the paradox. The second part describes the work done on two sample components of the proposed library. The third part evaluates the work on these components and tries to induce from this experience the perspectives of a full library.

Declaration of Authenticity

I submit this report as the thesis for my diploma in Computational Science and Engineering at the Department of Mathematics at the Swiss Federal Institute of Technology ETHZ. It describes the work I have done between November 1^{st}, 2002, and March 1^{st}, 2003, at the Chair of Software Engineering at the Department of Computer Science at ETHZ. I declare that the work described is my original contribution to the field. Foundations laid by others upon which my work is built have been correctly cited and referenced according to my best knowledge.

Zürich, March 9^{th}, 2003 Peter Häfliger

Acknowledgements

To prevent this list from filling up my whole report, I will restrict it to the three people who have inspired me most in the course of the last four months:

First and most of all, I thank Prof. Bertrand Meyer for having offered me the very first diploma project at his newly established Chair of Software Engineering. It has been a great privilege to work in such a close collaboration with one of the pioneers of our field.

I thank Susanne Cech for the loyal and generous support she has always given me, starting with a task as boring as the installation of Microsoft® Windows XP on my machine, moving on to careful reviews of all my code and up to the final proof-reading of my report at the unusual working hour of a Sunday morning.

Finally, I thank my friend and fellow-student Torsten Hofmann. Besides the work on his own diploma thesis in Fluid Dynamics and the pursuits of diverse interests such as Perl hacking, he has always found the time to make a comment on my various design attempts.

Contents

Part I

Introduction

Chapter 1

The Problem to be Addressed

The first chapter tries to motivate the research done within the scope of this diploma thesis by sketching the problem that has been addressed and by placing it in its historic context. The ground shall be prepared for the solution proposed in the second chapter.

1.1 The Scientific Software Paradox

The problem which this diploma thesis addresses is the 'Scientific Software Paradox' as described by Bertrand Meyer at the beginning of his preface to Paul Dubois' book on the EiffelMath library ([OTSC]):

> "Scientific software is a paradoxical product. The people creating it are, as one may expect, scientists: physicists, mathematicians, chemists, experts in mechanical, electrical, or aeronautical engineering. . . . In most cases, however, the product itself is surprisingly unscientific. Software engineering advances of the past three decades have penetrated this field more slowly than areas such as compilers, systems software, user interfaces, telecommunications, or even Management Information Systems. Two reasons can be found to explain the Scientific Software Paradox.
>
> The first is the age of the field: being the oldest application of computers — actually, the one that prompted the very idea of a computer – it has had to bear the weight of tradition, like an ancient and prestigious industrial metropolis that finds itself bypassed by upstart cities unencumbered by existing infrastructure. In this case, a large part of the infrastructure is Fortran, the language of choice of the scientific software community, once a brilliant invention which today finds it hard to keep up with the progress of modern software ideas.
>
> The other reason for the paradox is that many scientific programmers, unlike their colleagues in other application areas, do not have programmer or software engineer on their business cards: their job title reads rocket scientist or the like, and they think of themselves as professionals in fields other than software even if – as is often the case – much or most of their time is spent dealing with computers."

1.2 Historical Background and State of the Art

1.2.1 Programming Languages and Software Engineering

The Search for ever Higher Levels of Abstraction

Since the invention of the first programmable computers more than 60 years ago, the designers of new Programming Languages and the developers of new Software Engineering Techniques have strived at ever more abstraction. [1]

In the late 1940s and the better part of the 1950s, computers were first programmed directly in machine language, later in assembly language. Assembly languages already improved the writability and readability of computer programs for human programmers by replacing the bit series for each machine instruction with a three-letter mnemonic. But the programmer still had to think in individual machine instructions when he was designing his programs. No support for abstraction was offered yet.

IBM's FORmula TRANslating System (FORTRAN), for which the first compiler was released in April 1957, was a quantum leap: FORTRAN I was the first major high-level language which offered some basic high-level constructs we all take for granted today:

- user-definable subroutines
- conditional branching (IF clause)
- loops (DO clause)

One year after the release of FORTRAN I, FORTRAN II added support for independent compilation of subroutines. In 1962, FORTRAN IV added type declarations and the capability of passing subprograms as parameters to other subprograms. FORTRAN 77 added a few more features like string handling and an optional ELSE clause to the IF clause.

FORTRAN 90 was a radical change to the language:

- A large set of functions for array operations was built in.
- New control statements like the multiple selection CASE were added.
- Modular programming is supported like in Ada or Modula-2.
- Arrays can be dynamically allocated and deallocated on command if they have been declared to be ALLOCATABLE.
- Procedures can now be called recursively.

[1]For more background on the History of Programming Languages, see [Sebesta], chapter 2, from which I have learned a lot of what I have tried to summarize in this subsection.

Especially the last two changes are dramatic: They depart from the original FORTRAN concept of allowing static data only. In the earlier FORTRANs, no recursive subprograms were allowed and no new variables could be allocated at run time. This made it difficult to implement data structures that grow or change shape dynamically. But since the types and storage for all variables could be determined at compile time, the construction of highly optimizing compilers was possible. Mathematicians were willing to trade design flexibility for better performance, but software engineers soon turned to more expressive languages. With FORTRAN 90, the FORTRAN family tried to re-enter the club of state-of-the-art programming languages [2], but as far as I can judge, the scientific computing community has not followed. Their reason for not making the transition from FORTRAN 77 to FORTRAN 90 is exactly their reason for favouring FORTRAN in the first place: Performance is their top priority, more important than flexibility or modularity. Where I have seen FORTRAN at work, it was usually FORTRAN 77. If I write FORTRAN without any further qualification in the following text, I always refer to FORTRAN 77.

In the wake of the first FORTRAN versions, many other early high-level languages were developed. This 'babylonic' situation made communication between different communities difficult. Furthermore, these languages were usually specific to a single computer architecture. People started realizing the need for a 'universal' programming language. FORTRAN was no candidate because at the time it was actually owned by IBM. ALGOL was a transatlantic joint effort of ACM[3] and GAMM[4] to design a language subject to the following criteria:

- It should be as close as possible to standard mathematical notation, and programs written in it should be readable with little further explanation.
- It should be possible to use the language for the description of computing processes in publications.
- Programs in the new language must be mechanically translatable into machine language.

The first criterion highlights the leading role of Scientific Computing, which was at that time still the primary computer application area, in the development of programm languages and software engineering techniques. The second and the third criteria show the long way that the programming discipline had already gone in the first twenty years of automatic computation: Programs that could be read and reasoned about and printed in articles had to be at a far higher level of abstraction than machine or assembly language.

ALGOL 58 had only generalized FORTRAN's constructs where they had been too dependent on IBM hardware. But ALGOL 60 introduced some important innovations:

- the concept of block structures which allowed the localization of programm parts by introducing new data environments (scopes)

[2]According to [DDJ], the next standard — FORTRAN 2000 — even offers object-oriented programming

[3]Association for Computing Machinery

[4]Gesellschaft für Angewandte Mathematik und Mechanik

- two different possibilities of passing parameters to subprograms: pass by value and pass by reference
- recursive procedure calls
- semi-dynamic arrays where the subscript range could be specified by variables instead of constants

The last two developments show that although ALGOL 60 was mainly developed for Scientific Computing and although GAMM was a key player in its design, the design principles differed greatly from those of the FORTRAN designers: Expressiveness and flexibility were given higher priority than speed optimization.

Pascal was a descendant of Algol 60 developed by Niklaus Wirth between 1971 and 1973. Wirth did not add any major features or new abstractions, but stressed on simplicity, safety, readability and teachability. Pascal was primarily an educational language and enjoyed wide-spread use at many universities all over the world.

The C language was another descendant of Algol 60 which added only little to the established set of language features and provided no new abstractions. It was developed by Dennis Ritchie between 1969 and 1973 at Bell Laboratories mainly as an implementation language for the UNIX operating system. For this reason, very low-level features were included, e.g. for direct memory manipulation. Safety had not been a priority issue in the design of the language: C is notorious for its lack of complete static type checking or for the dangerous hacks that are possible with its pointers. C was very successful because it was almost as small as Pascal. The whole language could quickly be learned. Besides that, its success was closely tied to the success of the UNIX operating system and the wide-spread availability of compilers (often free with UNIX). C enjoys still wide-spread use today, mainly in two areas:

- in system-level programming
- as an intermediate representation in the compilation of today's even higher-level languages [5]

As average programs grew bigger, the need for modularization and separate compilation arose. C already offered some support for this through the way program storage could be split over several files. As with other issues, the C designers chose the low-level approach to modularization. The development of more abstract modularization techniques, regarding software modules as implementations of abstract data types, was the contribution of Ada and Modula-2 around 1980.

Modular software development has many advantages. Software can be split into parts with clearly defined interfaces which can then be implemented by different programmers in parallel. Data encapsulation is a powerful facility which enforces separation of concerns and allows specialization of the team members: Only part of the team has

[5]See [OOSC-2], section 34.4, where Bertrand Meyer emphasizes this role of C today and looks at its history from yet another angle, also taking into account more business-oriented languages like Cobol and PL/I.

to worry about specific details of an application — like access to the underlying hardware or network technology or the implementation of a very difficult algorithm for a specific computation. These programmers can hide the tedious details of their trade in a software module implementing a higher-level abstraction and providing access to this more abstract functionality rather than to the gory details of the underlying technology. In our world of rapid technological innovation, modularity even allows exchange of the implementation with an improved or altogether different version without causing any trouble in any other part of the system, because each module only knows about the other modules' interfaces and remains happily ignorant of the implementation.

The next big step was the advent of Object-Oriented Programming. Classes first appeared in Simula 67, yet another descendant of Algol 60, in 1967, even before the development of Pascal and C! But the class concept was not yet used for the purpose of data abstraction. Classes had been introduced by Kristen Nygaard and Ole-Johan Dahl, the designers of Simula 67, to allow control flow to pass from a called routine back to the caller and then back to the callee again, exactly to the point where the execution had been left in the first call. *Sub*routines could thus be elevated to *co*routines. The need for coroutines had come from discrete-event simulation programs which gave the language their name 'Simula'. The disadvantage of this naming was that most programmers thought of it as a language *exclusively* applicable to simulation. Only few people recognized the powerful possibilities it offered for the modelling of systems in general.

Though not very widely used itself, Simula 67 did have a tremendous effect on the development of subsequent languages: C.A.R. Hoare was the first to link the concept of abstract data type to the class concept [Hoare] and Alan Kay realized its applicability to graphics programming and user interface design, which led him to the development of Smalltalk.

On top of the benefits of modularity mentioned above, object technology adds two new concepts:

1. **Inheritance** allows code reuse between similar modules: Circles and squares are both figures; they both have a color and can be drawn on the screen. The three classes form a hierarchy, figure being the ancestor, circle and square being two of its descendants. Color is an attribute and the capacity of being drawn a behaviour of figures in general; they are placed at the higher level in the hierarchy (in the *figure* class) and simply inherited at the lower level (from both circles and squares). Diameter, however, is an attribute of circles only, not of squares; it is placed in the *circle* class and not shared by the *square* class. Circles and squares are both *specializations* of the more general concept of figure: They inherit all attributes and behaviour of *figure* and may add supplementary attributes or behaviour. Features that are common to all descendants of *figure* are implemented only once (in the *figure* class). Code duplication and inconsistencies are avoided: The code is written and maintained in one single place and reused wherever appropriate.

2. **Polymorphism** is the substitution of an instance of a class by an instance of a child class. This means that it is possible to assign to variable *my_figure* of type *figure* an instance of type *circle* or *square*. If the feature *draw* is then called on *my_figure*, either a circle or a square is drawn on the screen, depending on the

> type of the instance currently attached to the variable. This implies that I can write a code that manipulates abstract figures without having to know all the concrete figures that are implemented at the moment or will be implemented by other people in the future. Polymorphism thus ensures flexibility and exendibility of software systems in a world of ever-changing requirements.

With the new concepts of inheritance and polymorphism came a whole new method of structuring and designing software. Class hierarchies are a powerful modelling tool. Many people still think at first that the method only applies to simulation software where a lot of so-called 'real-world' objects are modelled. But once you have more fully digested object technology, the way you perceive software systems changes: Your focus shifts from algorithms to data types. The 'objects' in an object-oriented software system do not have to correspond to 'real-world' objects like trucks or cars. Figures are objects of a much more abstract type: You do not really see circles in nature; 'circle' is just a geometric abstraction. A C-programmer confronted with the task of designing a graphics application would probably think of a subroutine *draw* with an argument of type *figure* and a huge switch-case statement in its body to call a different drawing routine for each type of figure. You have seen the O-O approach above. But object technology applies to even more abstract kinds of objects. Mathematicians tend to think that algorithms can only be implemented in functional or procedural languages as functions. In chapter 4, you will see that after some training in object technology, you even look at things like numeric integration in a different way: You'll see data types like integrators or intervals first and regard the algorithms as features offered by the instances of these data types.

Of course I cannot give a more detailed description of object technology in the course of this general introduction. If you want the big picture, have a look at [OOSC-2].

Reasoning about Programs: Safety vs. Expressivity

Some language designers argue that the search for ever higher levels of abstraction is also a strive for ever higher 'expressivity' or 'power'. The following quotes are taken from a very interesting article ([Graham]) by Paul Graham, where he describes the secret behind the success of his startup company 'Viaweb' (which was later bought and still runs today as 'Yahoo!Store'):

> "All languages are equally powerful in the sense of being Turing equivalent, but that's not the sense of the word programmers care about. (No one wants to program a Turing machine.) The kind of power programmers care about may not be formally definable, but one way to explain it would be to say that it refers to features you could only get in the less powerful language by writing an interpreter for the more powerful language in it. If language A has an operator for removing spaces from strings and language B doesn't, that probably doesn't make A more powerful, because you can probably write a subroutine to do it in B. But if A supports, say, recursion, and B doesn't, that's not likely to be something you can fix by writing library functions.
>
> ...

> Few would dispute, at least, that high level languages are more powerful than machine language. Most programmers today would agree that you do not, ordinarily, want to program in machine language. Instead, you should program in a high-level language, and have a compiler translate it into machine language for you.
>
> . . .
>
> Everyone knows it's a mistake to write your whole program by hand in machine language. What's less often understood is that there is a more general principle here: that if you have a choice of several languages, it is, all other things being equal, a mistake to program in anything but the most powerful one.
>
> . . .
>
> Languages fall along a continuum of abstractness, from the most powerful all the way down to machine languages, which themselves vary in power. Consider Cobol. Cobol is a high-level language, in the sense that it gets compiled into machine language. Would anyone seriously argue that Cobol is equivalent in power to, say, Python? It's probably closer to machine language than Python.
>
> . . .
>
> Lisp code, after it's read by the parser, is made of data structures that you can traverse. If you understand how compilers work, what's really going on is not so much that Lisp has a strange syntax as that Lisp has no syntax. You write programs in the parse trees that get generated within the compiler when other languages are parsed. But these parse trees are fully accessible to your programs. You can write programs that manipulate them. In Lisp, these programs are called macros. They are programs that write programs. Programs that write programs? When would you ever want to do that? Not very often, if you think in Cobol. All the time, if you think in Lisp.
>
> . . .
>
> But I think I can give a kind of argument that might be convincing. The source code of the Viaweb editor was probably about 20-25% macros. Macros are harder to write than ordinary Lisp functions, and it's considered to be bad style to use them when they're not necessary. So every macro in that code is there because it has to be. What that means is that at least 20-25% of the code in this program is doing things that you can't easily do in any other language. . . . We weren't writing this code for our own amusement. We were a tiny startup, programming as hard as we could in order to put technical barriers between us and our competitors.
>
> A suspicious person might begin to wonder if there was some correlation here. A big chunk of our code was doing things that are very hard to do in other languages. The resulting software did things our competitors' software couldn't do. Maybe there was some kind of connection."

Programs that manipulate the abstract syntax tree at run time? I am not so sure whether I am really dreaming of that. I am not so sure whether I am dreaming of Lisp anyway. Common Lisp today combines object-oriented, functional and procedural programming within a single language. Many language designers seem to think that the best design principle is just to 'throw everything into the soup'. That's not only true of Common Lisp, but also of C++. Bjarne Stroustrup, the inventor of the C++ language, is quoted at the ANSI Lisp website [6]:

> "No single language can support every style, but a variety of styles can be supported within the framework of a single language. Where this can be done, significant benefits arise from sharing a common type system, a common toolset, and so forth. These technical advantages translate into important practical benefits such as enabling groups with moderately differing needs to share a language rather than having to apply a number of specialized languages."

An even more extreme example is Perl where you can basically do everything you want, but even the designers themselves warn of some of its more dangerous features (like the possibility of changing the inheritance hierarchy at runtime).

Paul Graham may be right that Lisp is the language of choice in an environment where three very gifted and very experienced hackers (like authors of Lisp textbooks) collaborate day and night over a limited period of time in a startup and are then lucky enough to sell their business to 'Yahoo!' and never have to worry about maintenance again. The same applies to Perl and C++.

In an environment where large teams develop *and maintain* large software systems over a long period of time, Paul Graham's observations may not be right anymore. Smaller, safer languages with a consistent design may then be favourable. They might not be as 'powerful' as others, because there are things that require more effort to write in a small language, and there are even things that you cannot do with them. But what you gain from these restrictions is readability and understandability. Bruce Eckel ([Eckel]) calls Perl a 'write-only' language because sometimes even after a few hours, you cannot read your own programs anymore. Having to maintain a program once written by some crazy Perl hacker must be a nightmare.

Besides the trend towards higher abstraction, the last decades have seen a trend towards more restriction of features considered to be dangerous. In 1968, Edsger Dijkstra ([Dijkstra]) described why he considered the *goto* statement to be harmful: It makes it too difficult to reason about programs because it introduces a level of complexity which is too difficult to grasp for the (average) human mind. The *goto* statement was subsequently removed from most serious high-level languages.

[Wirth] shows how to every program language construct corresponds a data structure: fixed-sized arrays correspond to *for*–loops, variable-sized arrays to *while*-loops, trees to recursive feature calls, etc. Explicit pointers correspond to the explicit *goto* statement. So it may not come as a surprise that after the successful suppression of the explicit *goto*, explicit pointers are more and more disappearing.

The hybrid (O-O and procedural) approach and the requirement of utmost backwards compatibility with C do not allow C++ to discard the possibility of explicit pointer manipulations. Side by side with high-level O-O abstraction possibilities, it still has to

[6]See: http://www.lisp.org/table/objects.htm

offer low-level systems programming facilities. This explains for a lot of its dangers and inconsistencies. The survival of explicit pointers in C++ also makes automatic garbage collection impossible, introducing the problem of memory leaking.

Java is a step in the right direction. It constrains the programmer a bit, for example by the lack of explicit pointers, but offers automatic garbage collection and complete static type checking in return. Some people call Java 'C++ with the bugs fixed'.

Eiffel is a purely object-oriented language by design. It is a lot more powerful than procedural languages like C. To express it according to Paul Graham's definition of 'power': If you wanted to get the features of Eiffel in C, you would have to write an interpreter for Eiffel in C. You can use an Eiffel compiler instead, written fully in Eiffel, which does exactly this: It compiles Eiffel to C. But Eiffel is at the same time a lot safer than C, for example by hiding pointer manipulations from the programmer. And it is probably even easier to learn.

1.2.2 Scientific Computing

The motivation for building programmable computing machines came from ever larger computations to be done in science and engineering. The first computers that Konrad Zuse constructed between 1935 and 1945 (the famous 'Z1' – 'Z4' series) were used for the the solution of static systems in civil engineering and aerodynamic systems in the aircraft industry [Zuse]. And for quite some time, Scientific Computing remained the main application area for computing machines [7]. The key part played by GAMM in the design of ALGOL 60 has been sketched above.

Some large numerical library projects were started in the 1960s and 1970s, for example the NAG ('Nottingham Algorithms Group', renamed 'Numerical Algorithms Group' in 1973) project which began in 1970 as a collaborative effort of the Universities of Birmingham, Leeds, Manchester, Nottingham and Oxford and the Atlas Computer Laboratory. Mark 1 of the NAG Library contained 98 user-callable routines and was released on October 1st, 1971 — implemented both in ALGOL 60 and in ANSI FORTRAN. The aims of the NAG effort were the following [8]:

- to create a balanced, general-purpose library of algorithms which meets the numerical and statistical needs of computer users
- to support the library with documentation giving advice on problem identification, algorithm selection, and routine usage
- to provide a substantial test suite, including example test programs, for certification of the library
- to implement the library as widely as user demand required

I don't know what the state of the art in subroutine libraries was at the time, but these aims seem to me to express a concern for quality which may not have been standard back then: A lot of emphasis is laid on documentation and test suites.

[7] For a nice collection of essays about the early days of Scientific Computing, see [Nash].
[8] See: http://www.nag.co.uk/about_EarlyYears.asp

I am not an expert in Scientific Computing and may have a very limited view, but it seems to me that in the last years research has mostly been focused in the following two areas:

- Theoretical foundations like definitions of functional spaces and other theoretical frameworks, proofs of computability, existence proofs of solutions, convergence proofs of approximations, etc. This is research at the highest level of abstraction: It is the roof which shields me from unexpected behaviour of algorithms and the protection without which I would not even think of writing scientific software.
- Algorithmic improvements like ever faster matrix inversion and Fast Fourier Transform (FFT). This is research at the lowest technical level, what some people disrespectfully call 'number crunching': It is the ground on which I build my software when the systems get big and speed is critical.

Ground and roof? That's nice, but something seems to be missing in between: The high-level concepts of mathematics and the sophisticated implementations of lowest-level computation tasks should be combined in a framework that allows system modelling by providing programs that are easy to use, reuse, maintain and adapt to new needs. But as Bertrand Meyer has been quoted at the beginning of this text, Scientific Computing does not seem to have contributed to nor profited much from advances in Software Engineering during the last thirty years.

In his book on EiffelMath ([OTSC]), Paul Dubois explains why the arguments for object technology weigh even more in Scientific Computing than in other computing environments:

- The code stays around for a long time, sometimes for decades.
- The code evolves: Most systems start out small and specific and then get larger and more general.
- The people who write scientific software do not usually have a degree in computer science: They are neither hardware nor memory management experts and should be shielded from these things as much as possible.
- There is usually no explicit specification and no consistent design effort: Somebody writes a tool and then somebody else adjusts it for his own purposes or integrates it with other tools.
- Collaborators do often not have a full-time contract, but are students doing a semester project, building on the code of other students who maybe did a PhD thesis at the same institute years ago.

If object technology is useful in the area of compilers or management information systems where projects are usually done with a small team of full-time expert programmers, starting with a specification document and making a lot of design considerations before even writing the first line of code, then object technology would certainly be useful in the field of scientific computing as described above.

As a student of Computational Science and Engineering at ETH's Department of Mathematics, I have had the possibility to get a look into several different departments and institutes during the last four years. In addition, I have had the chance of doing a semester project at ABB Corporate Research in Baden-Dättwil, a private, company-funded and thus more "applied" and "industrial" research center, as well as an internship at the University of Campinas UNICAMP in São Paulo State, Brazil. In this time I observed the following:

- Where the problem size is not too big, scientists and engineers use very high-level languages and tools like MathWorks' MATLAB® and Simulink® packages:
  - Institute of Robotics at ETH
  - Theoretical Physics Group at ABB Corporate Research
  - Phonak Hearing Systems Company (according to company presentation)
- Where the systems are large and speed is important, functional decomposition is the predominant design method and FORTRAN and C are the implementation languages of choice:
  - Institute of Fluid Dynamics at ETH
  - Parallel Computing and Applied Mathematics Groups at ETH (e.g. Dr. Wesley Petersen)
  - Computational Chemistry at ETH (Prof. Dr. Wilfred van Gunsteren)
- There is a vanguard that has already made the transition to object technology:
  - Institute of Mechatronics and System Dynamics, Gerhard-Mercator-University Duisburg, Germany (Prof. Dr.-Ing. habil. Manfred Hiller): Development of the 'MOBILE' package for the simulation of mechanical systems of complex topology [9]
  - Group of Prof. Dr. Christoph Schwab at the Seminar of Applied Mathematics at ETH: Development of the 'Concepts' package for the solution of partial differential equations [10]
  - Group of Prof. Dr. Marco Lúcio Bittencourt at the Departamento de Projeto Mecânico at the University of Campinas UNICAMP, Brazil

  In all three examples, 'object technology' means C++. If you have read the whole document so far, you know why I think that Eiffel would be the better choice.

The number of observations I have had the possiblity to make is far from being large enough for statistical analysis and may not be representative for Scientific Computing in general. Nevertheless I dare the conclusion that object technology is steadily entering the field, at least in individual vanguard research groups and for certain individual applications.

However, I have not found evidence of any big object-oriented library covering a large range of mathematical and numerical areas:

[9]See: http://www.mechanik.tu-graz.ac.at/~mobile

[10]See: http://www.sam.math.ethz.ch/~concepts/

- Repositories like Netlib[11] offer mostly public-domain libraries that have originally been written in FORTRAN in the 1960s, '70s or '80s and maybe ported to C in the meantime. Libraries like BLAS, LINPACK, LAPACK, EISPACK and the like are fast and stable because they are highly optimized and have been publicly tested for years. They might be wrapped in higher-level languages, but nobody dares a re-implementation.

- One of the most successful and widely used commercial set of numeric libraries is developed and maintained by the Numeric Algorithms Group (NAG)[12]. They offer implementations in FORTRAN 77, FORTRAN 90 and C, but not in any object-oriented languages.

- Numerical Recipes, another provider of a widely used set of numeric libraries, offer implementations in many languages, among them C++ [13]. But a superficial leaving through their documentation gave me the impression that probably just the language has changed, not the design method. As an example, they offer no real support classes for vectors and matrices, because they cannot think of an implementation which is at the same time fast and general. They advise the user to write his own classes instead.

- Systems like MathWorks' MATLAB® and Simulink® packages offer high-level abstractions at the user interface level but delegate the actual computations to legacy code behind the scenes. That is the reason why they are at once very user-friendly and for some tasks even very performant.

- The same technique has been applied by Paul Dubois for the creation of his EiffelMath library which he describes in [OTSC]: He designed a large set of abstract data types like vectors and matrices, integrators, interpolators or distributions and implemented them in an Eiffel class library. But the actual computations are performed by internal calls to the NAG C library. Looked at from the opposite perspective, EiffelMath is just a very sophisticated wrapper for large parts of the NAG C library.

- In a book which I have unfortunately only discovered three weeks ago ([Besset]), Didier Besset describes the implementation of a library covering many areas of mathematics (like NAG or EiffelMath), but offering only relatively few algorithms per area (where NAG and EiffelMath offer many more). It can probably be said that his project proved the point that I am trying to make in my own project: The principal applicability of object-oriented design and implementation techniques to the creation of a large math library. But as far as I can judge, his implementation has not yet reached the full functionality provided by big commercial libraries like NAG or NR.

To dare a broader conclusion: Object Technology is certainly creeping around the corners, but it has not yet entered the mainstream of Scientific Computing. The "Scientific Software Paradox" is still waiting for a solution.

[11]See:http://www.netlib.org
[12]See: http://www.nag.com
[13]See: http://www.nr.com

Chapter 2

The Solution to be Investigated

After the definition of the "Scientific Software Paradox" and the outline of its historic evolution in the last chapter, the concrete project shall now be sketched with which we hope to show a path towards a solution of the paradox. At the end of the chapter, an outline of the second and third part of this thesis will be given.

2.1 Evolution of the Basic Idea

When I first talked to Bertrand Meyer about the possibility of doing my diploma thesis within his research group, he immediately suggested that I work on a re-implementation of Paul Dubois' EiffelMath. This was an idea he had carried around for some time. At that stage, the main goal was to free EiffelMath from the underlying NAG implementation in order to make it freely distributable. It was then thought that the best scheme would be to leave the design of EiffelMath untouched and replace NAG with a public-domain alternative to NAG. When it turned out that it would be difficult and probably impossible to find a library which covered all the functionality provided by NAG, the possibility of using different libraries for EiffelMath's individual clusters was investigated. But it was obvious that this would lead to a dangerous gap: Paul Dubois had had to corrupt his pure abstract design in order to comply to the peculiarities of the NAG C library. If we now were to implement this NAG-specific interface with other C or FORTRAN libraries, each with its own peculiarities, friction was unavoidable. So this scheme was quickly abandoned as well.

This was the moment when Bertrand first suggested a complete re-implementation of EiffelMath purely in Eiffel. This was a shocking idea to me at first, because I had been discouraged to re-invent the numeric wheel by my professors from the Department of Mathematics for years: "Don't even *think* of re-implementing the algorithms!" they would say, "People have spent their lives on the old implementations, so you'll *never* be able to beat them in performance!" Bertrand thought differently: With the advance of Eiffel Compilers within the last ten years, it might today be possible to implement the algorithms fully in Eiffel with only a small performance penalty. And this performance penalty would not actually be a 'loss', but rather an *investment* in some of the most critical qualities of larger software systems like reusability, readability, understandability and maintainability. A program written by a genius in FORTRAN in the 1960s and later automatically translated to C might today still be fast code, but it will probably not get any better in the future, because nobody understands it and nobody is brave enough to start modifying it. Our code, on the contrary, would just be a starting point: An understandable and transparent implementation of clear and simple concepts. Our task would be to prove the principle feasibility to mathematicians in order to attract the real experts in individual fields which could then come spend as much of their lives at the improvement of a single algorithm as they pleased.

“The Scientific Community has relied on FORTRAN implementations for more than forty years now”, Bertrand said, “we cannot go on relying on FORTRAN for another 150 years, just interfacing every new technology back to the old implementations. One day we’ll have to re-implement the algorithms from scratch, and this day may well have come now!” In the meantime, I had caught fire ... The project was set!

2.2 Evolution of the Project

At the time Paul Dubois had designed EiffelMath, he had not only had a PhD in Mathematics, but also twenty-five years of experience in Scientific Computing, mainly in FORTRAN, but also in various higher-level languages. He had worked hard to come up with good abstractions for mathematical and numerical concepts, and he had consulted with various specialists for the design of individual clusters. So our first thought was that it was impossible for me — not even having a diploma yet and only very little experience — to come up with better abstractions than he had. So we agreed that I would only dare slight improvements of Paul Dubois’ design. My task would be to abstract his pure design from the concrete NAG C implementation, i.e. to try to find out what he would have come up with if he had not been constrained by NAG. This purified design I would then try to implement fully in Eiffel.

It was decided that two sample clusters should be chosen for re-implementation. It was clear from the beginning that we wanted to do integration, because it offered an ideal application to test the new Eiffel agent mechanism. We were unsure about the best-possible second cluster, and so this decision was left open at first.

After some browsing through the whole library and based on the understanding that my task was re-implementation rather than re-design, I decided to start with the ‘special’ cluster, trying to understand the applications for the rich set of special functions offered by EiffelMath. I believed that if we wanted to re-implement EiffelMath, we would have to take every effort necessary to provide good implementations for the special functions it offers. I thought that special functions served as building blocks for many problems in many areas, and that low performance in the evaluation of special functions could thus destroy the performance of the other clusters. The more I worked myself into the area of special functions, however, the more insecure I got about how important they really are.

Special functions pose mathematical and implementation problems, but no design challenges. They are special *functions* and have to be offered as user-callable functions, just like elementary functions (sin, cos, exp, etc.). Even in the most rigid object-oriented approach, nobody has yet suggested turning the cosine into an abstract data type and implementing it as a class, rather than as a feature of a class called something like *ELEMENTARY_FUNCTIONS* or *DOUBLE_MATH*.

Integration is just the opposite: For a start, it is possible to get quite good results with the most simple algorithms, so the topic is not too difficult neither from the mathematics nor from the implementation point of view. But integration can supply material for almost endless design discussions. When I presented my first design attempt, it was highly criticized and it was seen that the 'EiffelMath purification approach' was wrong. It was realized that it *is* necessary to start from scratch with the most simple concepts and then work on to more complex abstractions.

This meant that after almost two thirds of my project time, the task changed: It was not a re-implementation of EiffelMath anymore, but the design and implementation of a new library from scratch. From this point of view, the worth of my work on special functions diminished further: If we write our own library, we have a lot more freedom to decide what special functions to offer. The decision made by Paul Dubois (who himself has had to adopt the decision made by the Numeric Algorithms Group) suddenly becomes kind of arbitrary: Some libraries offer a smaller set of special functions than NAG, others an even larger set. At some time, the people who fully implement the new library started with this diploma thesis will have to address the special functions issue, but in retrospect it should not have been such a large part of a pure feasibility study.

In the last part of my project, I really started thinking about basic design issues. That was the most challenging and most interesting part of my whole project. I would have some seemingly great idea late at night and be eager to present it to other members of the group first thing in the morning. The more the idea was criticized by the others, the more I learned from these discussions. In the end we came up with something useful, I believe.

2.3 Tasks to be Accomplished and Questions to be Answered

In the course of this diploma project, the feasibility of a fully object-oriented numeric math library was to be investigated by designing and implementing one or two representative sample components. The design and implementation goals were

- immediate useability of the components for public testing and scrutiny
- high quality
- high performance

These components were to be precisely assessed in an attempt to identify the benefits of a fully object-oriented approach to Scientific Computing, as well as possible drawbacks, challenges and open issues.

Great importance had to be given to performance analysis, because it had to be feared that no mathematician would even consider our approach if the performance overhead was more than a few percent.

From the sample components, a generalization to the feasibility, advantages, challenges and open issues in the creation of a full object-oriented numeric math library was to be tried. If possible, a project plan was to be sketched.

2.4 Outline of the Thesis

After the detailed introduction in the first part of this document, the second part will describe work on two sample components:

- Special functions (chapter 3)
- Integration (chapter 4)

The work on special functions was mainly reading, discussing with mathematicians, trying to understand, and then writing. It led to almost no implementation. Discussions with mathematicians convinced me that the topic is not trivial. Besides, it is a mathematical area that is not 'en vogue'. The corresponding literature is old and the relevant information is somewhat dispersed in an ocean of other things. Since I have dived into it for a considerable part of the duration of this project, I found it important to record in detail what I have found, to facilitate the work of potential successors as much as possible.

The work on integration was completely different: Less reading, less writing, a lot of discussions and even more programming. I only describe the concepts in this report; my main achievement is written in a different language: Eiffel programs are a literary genre of their own.

The third and final part will describe an assessment

- of the the two sample components described in the second part (chapter 5), trying to give an answer to the following questions:
  - What are the benefits of a fully object-oriented approach to Scientific Computing?
  - What are possible drawbacks, challenges and open issues?
  - How does the performance of a fully object-oriented implementation compare to the following alternative implementations:
    - Modular Eiffel (Eiffula-2)
    - Procedural Eiffel (EIFTRAN)
    - C
    - Assembly language

- of the perspectives for a full object-oriented library (chapter 6):
  - Feasibility, advantages, challenges and open issues
  - Specification of basic platform requirements to support a full library
  - Estimate of the work
  - Modularizability of the work
  - A possible project plan

Part II

Sample Components

Chapter 3

Special Functions: The Last Resort of C

As mentioned in the introduction part, two clusters of EiffelMath were investigated more closely: 'special' and 'integrat'.

Many mathematical texts use the term *elementary functions* to denote a set of basic functions like the trigonometric, exponential and logarithmic functions. These functions are used on a daily basis in every area of analytical modelling. They serve as the building blocks of approximations of more complicated functions because they are easily integrable and differentiable. The term *special functions* is used for functions like Gamma, Bessel and error functions which appear as special solutions to model problems in mathematical physics. They serve as the elements from which solutions of more general problems can be constructed.[1]

Although the mathematic behavior of elementary and special functions is well understood, their numeric computation is not trivial, especially since they appear so many times in every program that efficiency is more critical than ever. In the words of Paul Dubois:

> "If you ask an old Fortran programmer if they might consider using a new language, one of the points that is sure to be mentioned is that most other languages do not have a full library of special mathematical functions . . . Special functions are very common in scientific programs, and *most of the algorithms are difficult — too difficult to just type in as a recipe.*"[2]

For efficiency reasons and in a deviation from our overall approach, elementary and special functions should therefore not be implemented in Eiffel. We should profit from the implementations provided by the standard C libraries shipped with every compiler. They should be the fastest implementations available because they are usually directly tuned to a particular CPU and operating system. But because of their standardization,

[1]For a detailed description of special functions, see [MOR], [Bateman], [Abramowitz] or [Wolfram]. All the formulae appearing in this section are taken from [MOR], chapters III and IX, where not stated otherwise.

[2][OTSC], pg. 223, highlighting is mine.

this individual machine-dependance of their implementations should in no way harm the portability of our code.

3.1 C Language Standards

As mentioned before, the C language was developed by Dennis Ritchie between 1969 and 1973 at Bell Laboratories. For a long time, the only "standard" was the book by Kernighan and Ritchie ([KandR]). Between 1982 and 1988, the American National Standards Institute ANSI developed a new description of C, which was adopted as a standard in 1989. Versions of C that comply to this standard are called ANSI C or C89. The standard does not only include the core language, but also a set of basic libraries for input/output, date and time, error handling, etc, that are implemented and shipped with compilers. Of major interest to us is the C89 standard math library which contains about 20 elementary functions.

The C++ language was standardized separately in 1998.

In 1999, a new ISO/ANSI standard for C was published. This new standard is very interesting for us, because the C99 standard math library contains an additional set of about 40 new functions. Besides elementary functions, some special functions are now provided. Of course this new math library standard only applies to C compilers so far and not to C++ compilers yet, there is reasonable hope that the same math library standard will be adopted for C++ in the next few years as well:

> "**C99**: The second official ISO/ANSI C Standard, published in 1999. This standard contains much that the C++ committee can be expected to adopt wholesale, or with minor modifications, as part of C++0x. After all, it's clear that the C++ committee values C compatibility, and the C committee has helped us by likewise valuing C++ compatibility, which has made some of C99's features easier to integrate into C++0x than they might otherwise have been. There are still some C99 features, however, that C++0x cannot easily adopt in their C99 form, because conflicting facilities already exist in C++98 (for example, complex is a class template in C++98 and a keyword in C99)." [3]

> "The most important standardization effort we should coordinate with, of course, is C. C99 has added a number of library components that were not present in C90; all of them should be considered as possible C++ extensions." [4]

Since the C99 standard math library has been implemented for several C compilers [5] and can be expected to be implemented for at least the most widespread C++ compilers

[3]Herb Sutter, independent consultant and secretary of the ISO/ANSI C++ standards committee, quoted from [Sutter].

[4]JTC1/SC22/WG21, usually shortened to WG21, the technical working group responsible for C++ within ISO, quoted from [ISOWG21].

[5]e.g. [C99MATH.H], from which all the following C99 math function signatures are quoted.

like Microsoft Visual C++ within the next few years, the functions it offers should be taken into account when thinking about the implementation of special functions for our library.

3.2 *DOUBLE_MATH*

The class *DOUBLE_MATH* in the 'support.classic' cluster of ISE's EiffelBase offers the following 14 elementary functions:

3.2.1 Absolute value and nearest integer

dabs (v: DOUBLE): DOUBLE
-- Absolute of 'v'

ceiling (v: DOUBLE): DOUBLE
-- Least integral greater than or equal to 'v'

floor (v: DOUBLE): DOUBLE
-- Greatest integral less than or equal to 'v'

3.2.2 Square root

sqrt (v: DOUBLE): DOUBLE
-- Square root of 'v'

3.2.3 Exponential function and logarithm

exp (x: DOUBLE): DOUBLE
-- Exponential of 'v'

log (v: DOUBLE): DOUBLE
-- Natural logarithm of 'v'

log10 (v: DOUBLE): DOUBLE
-- Base 10 logarithm of 'v'

log_2 (v: DOUBLE): DOUBLE
-- Base 2 logarithm of 'v'

3.2.4 Trigonometric and inverse trigonometric functions

cosine (v: DOUBLE): DOUBLE
-- Trigonometric cosine of radian 'v' approximated

sine (v: DOUBLE): DOUBLE
-- Trigonometric sine of radian 'v' approximated

tangent (v: DOUBLE): DOUBLE
-- Trigonometric tangent of radian 'v' approximated

arc_cosine (v: DOUBLE): DOUBLE
-- Trigonometric arccosine of radian 'v'
-- in the range [0, pi]

arc_sine (v: DOUBLE): DOUBLE
-- Trigonometric arcsine of radian 'v'
-- in the range [-pi/2, +pi/2]

arc_tangent (v: DOUBLE): DOUBLE
-- Trigonometric arctangent of radian 'v'
-- in the range [-pi/2, +pi/2]

log_2 internally calls *log* twice. The other 13 functions can all be implemented as external calls to the following functions from the C89 standard math library [C89MATH.H], respectively:

double __cdecl fabs(double);	*double __cdecl cos(double);*
double __cdecl ceil(double);	*double __cdecl sin(double);*
double __cdecl floor(double);	*double __cdecl tan(double);*
double __cdecl sqrt(double);	*double __cdecl acos(double);*
double __cdecl exp(double);	*double __cdecl asin(double);*
double __cdecl log(double);	*double __cdecl atan(double);*
double __cdecl log10(double);	

3.3 *SPECIAL_FUNCTIONS*

The class *SPECIAL_FUNCTIONS* in EiffelMath's 'special' cluster wraps the complete set of 43 special functions offered by the NAG C library's 's' chapter [NAGC]. These functions have to be examined individually, and different schemes for a self-contained implementation have to be considered.

3.3.1 Hyperbolic functions

frozen *cosh (x: DOUBLE): DOUBLE*
-- hyperbolic cosine

frozen *sinh (x: DOUBLE): DOUBLE*
-- hyperbolic sine

frozen *tanh (x: DOUBLE): DOUBLE*
-- hyperbolic tangent

The hyperbolic functions can be implemented as external calls to three of the remaining functions from the C89 standard math library [C89MATH.H]:

```
double __cdecl cosh(double);        double __cdecl tanh(double);
double __cdecl sinh(double);
```

3.3.2 Inverse hyperbolic functions

frozen *arccosh (x: DOUBLE): DOUBLE*
-- inverse hyperbolic cosine

frozen *arcsinh (x: DOUBLE): DOUBLE*
-- inverse hyperbolic sine

frozen *arctanh (x: DOUBLE): DOUBLE*
-- hyperbolic tangent

The inverse hyperbolic functions can be implemented as direct calls to their respective implementations in the C99 standard math library:

```
double acosh(double x);        double atanh(double x);
double asinh(double x);
```

3.3.3 Bessel functions of the first kind

Bessel functions are special solutions of Bessel's differential equation [6]:

$$z^2\frac{d^2\omega}{dz^2} + z\frac{d\omega}{dz} + (z^2 - \nu^2)\omega = 0 \tag{3.1}$$

where ν and z can be arbitrarily complex.

The definition of the Bessel functions (of the first kind) is:

$$J_\nu(z) := \sum_{m=0}^{\infty} \frac{(-1)^m (\frac{z}{2})^{\nu+2m}}{m!\Gamma(\nu+m+1)} \tag{3.2}$$

where ν and z can again be arbitrarily complex.

The first two real Bessel functions of the first kind ($J_0(x)$ and $J_1(x)$, where $\nu = 0$ or 1, respectively, and the variable x is of type *DOUBLE*) appear in *SPECIAL_FUNCTIONS* as:

[6] For a detailed physically motivated derivation (in German), see [Jänich], part 3.

```
frozen bessel_j0 (x: DOUBLE): DOUBLE
        -- Bessel function j0

frozen bessel_j1 (x: DOUBLE): DOUBLE
        -- Bessel function j1
```

They can be implemented as direct calls to their respective implementations in the C99 standard math library:

```
double j0(double x);          double j1(double x);
```

The C99 standard math library even offers an implementation of the n^{th} real Bessel function of the first kind ($J_n(x)$, where $\nu = n$ is of type *INTEGER* and the variable x is of type *DOUBLE*):

```
double jn(int n, double x);
```

It can now be offered additionally in *SPECIAL_FUNCTIONS* as:

```
frozen bessel_jn (n: INTEGER; x: DOUBLE): DOUBLE
        -- Bessel function jn
```

3.3.4 Bessel functions of the second kind

Bessel functions of the second kind are nothing more than a combination of Bessel functions of the first kind. They are sometimes called Weber or Neumann functions. Their definition is:

$$Y_\nu(z) := \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)} \tag{3.3}$$

The first two real Bessel functions of the second kind appear in *SPECIAL_FUNCTIONS* as:

```
frozen bessel_y0 (x: DOUBLE): DOUBLE
        -- Bessel function Y0

frozen bessel_y1 (x: DOUBLE): DOUBLE
        -- Bessel function Y1
```

They can be implemented as direct calls to their respective implementations in the C99 standard math library:

double y0(double x); *double y1(double x);*

The C99 standard math library even offers an implementation of the n^{th} real Bessel function of the second kind:

double yn(int n, double x);

It can now be offered additionally in *SPECIAL_FUNCTIONS* as:

frozen *bessel_yn(n: INTEGER; x: DOUBLE): DOUBLE*
-- Bessel function YN

3.3.5 Modified Bessel functions

The modified Bessel functions are special solutions of the modified Bessel differential equation:

$$z^2\frac{d^2\omega}{dz^2} + z\frac{d\omega}{dz} + (z^2 + \nu^2)\omega = 0 \tag{3.4}$$

The definition of the modified Bessel functions (of the first kind) is:

$$I_\nu(z) := e^{-i\frac{\pi}{2}\nu} J_\nu(ze^{i\frac{\pi}{2}}) = \sum_{m=0}^{\infty} \frac{(\frac{z}{2})^{\nu+2m}}{m!\Gamma(\nu+m+1)} \tag{3.5}$$

The first two real modified Bessel functions of the first kind appear in *SPECIAL_FUNCTIONS* as:

frozen *bessel_i0 (x: DOUBLE): DOUBLE*
-- modified Bessel function of the first kind, I0

frozen *bessel_i1 (x: DOUBLE): DOUBLE*
-- modified Bessel function of the first kind, I1

In addition, scaled versions appear as:

frozen *bessel_i0_scaled (x: DOUBLE): DOUBLE*
-- scaled modified Bessel function of the first kind, e^(-|x|)*I0(x)

frozen *bessel_i1_scaled (x: DOUBLE): DOUBLE*
-- scaled modified Bessel function of the first kind, e^(-|x|)*I1(x)

Again, modified Bessel functions of the second kind are nothing more than a combination of modified Bessel functions of the first kind. Their definition is:

$$K_\nu(z) := \frac{\pi[I_{-\nu}(z) - J_\nu(z)]}{2\sin(\nu\pi)} \tag{3.6}$$

The first two real modified Bessel functions of the second kind and their scaled versions appear in *SPECIAL_FUNCTIONS* as:

```
frozen bessel_k0 (x: DOUBLE): DOUBLE
        -- modified Bessel function of the second kind, K0

frozen bessel_k1 (x: DOUBLE): DOUBLE
        -- modified Bessel function of the second kind, K1

frozen bessel_k0_scaled (x: DOUBLE): DOUBLE
        -- scaled modified Bessel function of the second kind, e^x*K0(x)

frozen bessel_k1_scaled (x: DOUBLE): DOUBLE
        -- scaled modified Bessel function of the second kind, e^x*K1(x)
```

Unfortunately, neither C standard math library provides an implementation of any of the modified Bessel functions. This is a real pity:

In the complex plane, Bessel's functions can easily be transformed into their modified counterparts, e.g. for positive integer indices $\nu = n$ simply by inserting iz into the definition of $J_n(z)$ and multiplying with i^{-n}: [7]

$$I_n = i^{-n} J_n(iz), \qquad n \in \{1, 2, 3, \ldots\} \tag{3.7}$$

But we are only dealing with Bessel functions for real arguments here, and in the realm of real numbers I have not found any finite combination of Bessel functions that yields modified Bessel functions [8].

Since the C99 standard math library does not offer an implementation of modified Bessel functions and since there seems to be no way of constructing them from a small number of calls to non-modified Bessel functions, they will have to be implemented separately [9]. Two approaches seem possible:

[7]See: http://mathworld.wolfram.com/ModifiedBesselFunctionoftheFirstKind.html

[8]I have not found a proof for the non-existence of such a finite combination, but none of the reference works that I have consulted ([Abramowitz], [Bateman], [MOR], [Wolfram]) lists any. Considering the wealth of formulae they provide, I take this as a strong indication that either no such combination exists or that at least none has been found yet. My discussions on this subject with Prof. Dr. Jörg Waldvogel from ETH's Seminar of Applied Mathematics have shown me that constructing a non-existence proof would probably not be an easy task and far beyond the scope of this diploma thesis. So I have decided to look for alternative approaches without having proven that the most direct approach is impossible in theory.

[9]A large repository of algorithms for the computation of special functions can be found at [ACM].

1. **Low-level implementation in C**: Creation of a variant of the C standard math library implementation of the non-modified Bessel function:

   + If the implementation is done with the series expansion in eq. (3.2), the variation of the algorithm will be small: The comparison of eq. (3.2) with eq. (3.5) shows that the only difference in each summand is just a factor $(-1)^m$.
   – If the implementation is not done with series expansion, the variation of the two implementations might be bigger.
   – To be efficient, a separate implementation will have to be provided for every compiler and platform on which the new library is to run. This means in effect that an extension of the standard C math library will have to be provided for every C compiler with which the library is to run.

2. **High-level implementation in Eiffel**: For example based on an integral representation of modified Bessel functions like

   $$K_\nu(z) = \int_0^\infty e^{-z\cosh(t)} cosh(\nu t) dt \qquad Re\ z > 0, \tag{3.8}$$

   making use of the integration algorithms provided by the library's own 'integrat' cluster. This solution will be:

   + fully portable
   – probably less efficient for lack of CPU-specific tuning

3.3.6 Airy functions

The Airy functions $Ai(z)$ and $Bi(z)$ are linearly independent solutions of the differential equation

$$\frac{d^2\omega}{dz^2} - z\omega = 0 \tag{3.9}$$

The signatures of the Airy functions and their derivatives in *SPECIAL_FUNCTIONS* read:

```
frozen airy_ai (x: DOUBLE): DOUBLE
        -- Airy function Ai

frozen airy_ai_deriv (x: DOUBLE): DOUBLE
        -- derivative of Airy function Ai

frozen airy_bi (x: DOUBLE): DOUBLE
        -- Airy function Bi

frozen airy_bi_deriv (x: DOUBLE): DOUBLE
        -- derivative of Airy function Bi
```

Since the Airy functions are not provided by the C99 standard math library and cannot be easily constructed from existing functions, they will have to be implemented from scratch. If the low-level implementation scheme has been chosen for the modified Bessel functions, with the consequence that for every target compiler a separate extension to the standard C math library will have to be provided, then the Airy functions may as well be included in this same C extension to enable platform-specific fine-tuning. On the other hand, since no existing C algorithm can easily be modified for this task, an implementation in Eiffel seems more consistent with the overall approach to the library.

3.3.7 Kelvin functions

Kelvin functions are the building blocks of solutions to the following complex second-order differential equation ([Abramowitz], pg. 379):

$$x^2\frac{d^2\omega}{dx^2} + x\frac{d\omega}{dx} - (ix^2 + \nu^2)\omega = 0 \tag{3.10}$$

where this time ν is real and x is real and non-negative.

There are four real-valued Kelvin functions. The signatures of their restrictions to the set of real values in *SPECIAL_FUNCTIONS* read:

```
frozen kelvin_ber (x: DOUBLE): DOUBLE
        -- Kelvin function ber(x)

frozen kelvin_bei (x: DOUBLE): DOUBLE
        -- Kelvin function bei(x)

frozen kelvin_ker (x: DOUBLE): DOUBLE
        -- Kelvin function ker(x)

frozen kelvin_kei (x: DOUBLE): DOUBLE
        -- Kelvin function kei(x)
```

where the missing index ν means $\nu = 0$.

Kelvin functions can be combined to solutions of eq. (3.10) ([Abramowitz], pg. 379):

$$\begin{aligned}
\omega(z) &= ber_\nu(z) + ibei_\nu(z) && (3.11)\\
&= ber_{-\nu}(z) + ibei_{-\nu}(z) && (3.12)\\
&= ker_\nu(z) + ikei_\nu(z) && (3.13)\\
&= ker_{-\nu}(z) + ikei_{-\nu}(z) && (3.14)
\end{aligned}$$

Kelvin functions can be expressed as the real and imaginary parts of Bessel functions of complex values. There is a whole collection of formulae connecting them ([Abramowitz], pg. 379), e.g. for the special case of $\nu = 0$ [10]:

$$ber(x) + ibei(x) = J_0(i\sqrt{i}x) = J_0(\frac{1}{2}(i-1)x) \tag{3.15}$$

[10] See http://mathworld.wolfram.com/KelvinFunctions.html

which again doesn't help us because we only have Bessel functions of real arguments at our disposal. So the comments that have been made for Airy functions also apply to Kelvin functions.

3.3.8 Gamma function

We have encountered the gamma function $\Gamma(x)$ before in the series expansion of the Bessel function and the modified Bessel function (eqs. (3.2) and (3.5)). The gamma function itself is defined:

$$\Gamma(z) := \int_0^\infty e^{-t} t^{z-1} dt \tag{3.16}$$

The gamma function is a generalization or interpolation of the factorial because it satisfies the functional equation

$$z\Gamma(z) = \Gamma(z+1) \tag{3.17}$$

and together with $\Gamma(z) = \int_0^\infty e^{-t} t^{z-1} dt = 1$, it can be shown by induction that [11]

$$\Gamma(n+1) = n!, \qquad n \in \{0, 1, 2, \ldots\} \tag{3.18}$$

SPECIAL_FUNCTIONS offers the gamma function as well as its logarithm:

```
frozen gamma (x: DOUBLE): DOUBLE
        -- gamma function

frozen log_gamma (x: DOUBLE): DOUBLE
        -- log of gamma function
```

The C99 standard math library provides two signatures

```
double gamma(double x);        double lgamma(double x);
```

which refer to the *same* function, namely the *logarithm* of the gamma function, contrary to what the names might suggest.

So the logarithm of the gamma function can be implemented in *SPECIAL_FUNCTIONS* through a direct call to the C function, whereas the gamma function itself simply becomes:

```
frozen gamma (x: DOUBLE): DOUBLE
        -- gamma function
    require
        x_not_a_negative_integer: x >= 0.0
            or else x /= x.truncated_to_integer
    do
        Result := exp(log_gamma(x))
    end
```

The double function call is stable and its double cost does not hurt too much.

[11] For a detailed derivation (in German), see again [Jänich], part 3.

3.3.9 Incomplete gamma functions

If we change the improper integral in the definition of the gamma function (eq. (3.16)) into a proper integral with upper bound x as parameter, we get the definition of the lower incomplete gamma function

$$\gamma(a,x) := \int_0^x e^{-t}t^{a-1}dt \qquad \mathit{Re}\ a > 0 \tag{3.19}$$

The definition of the upper incomplete gamma function is still an improper integral:

$$\Gamma(a,x) := \int_x^\infty e^{-t}t^{a-1}dt = \Gamma(a) - \gamma(a,x) \tag{3.20}$$

The regularized incomplete gamma functions are then defined as: [12]

$$P(a,x) := \frac{\gamma(a,x)}{\Gamma(a)} \tag{3.21}$$

$$Q(a,x) := \frac{\Gamma(a,x)}{\Gamma(a)} = 1 - P(a,x) \tag{3.22}$$

Their signatures appear in *SPECIAL_FUNCTIONS* as:

```
frozen p_incomplete_gamma (a, x, tol: DOUBLE): DOUBLE
        -- Compute incomplete gamma function 'P(a,x)'
        -- with relative error 'tol'.

frozen q_incomplete_gamma (a, x, tol: DOUBLE): DOUBLE
        -- Compute incomplete gamma functions 'Q(a,x)'
        -- with relative error 'tol'.
```

Since they are not implemented in the standard C99 math library, the comments that have been made for Airy functions also apply to the incomplete gamma functions.

3.3.10 Error functions

The error function $erf(x)$ and its complement, the complementary error function $erfc(x)$, are defined as follows:

$$erf(x) := \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}dt \tag{3.23}$$

$$erfc(x) := \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}dt = 1 - erf(x) \tag{3.24}$$

[12]See: http://mathworld.wolfram.com/RegularizedGammaFunction.html

In statistics, $erf(x)$ for real positive x is the probability that an observation will fall within x standard deviations of the mean in a standard normal distribution. $erfc(x)$ then is of course the complementary probability, i.e. the probability that an observation will fall *outside* x standard deviations of the mean. Both functions also appear in very different contexts, e.g. as building blocks of solutions to certain partial differential equations. The reason why most numeric libraries offer both functions is to avoid the loss of precision that would result from subtracting large probabilities (on large x) from 1.

The signatures in *SPECIAL_FUNCTIONS* read

```
frozen erf (x: DOUBLE): DOUBLE
        -- error function erf(x)

frozen erfc (x: DOUBLE): DOUBLE
        -- complementary error function erfc(x)
```

and they can be implemented through direct calls to the corresponding functions of the C99 standard math library:

```
double erf(double x);          double erfc(double x);
```

3.3.11 Cumulative normal distribution

Different authors use different definitions of the cumulative normal distribution. EiffelMath naturally follows the NAG C library definitions: [13]

$$\text{cum. normal distr.} \; : \; P(x) := \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-t^2}{2}} dt \qquad (3.25)$$

$$\text{complement} \; : \; Q(x) := \frac{1}{\sqrt{2\pi}} \int_{x}^{\infty} e^{\frac{-t^2}{2}} dt = 1 - P(x) \qquad (3.26)$$

The cumulative normal distribution and its complement can easily be related to the complementary error function:

$$P(x) = \frac{1}{2} erfc(-\frac{x}{\sqrt{2}}) \qquad (3.27)$$

$$Q(x) = \frac{1}{2} erfc(\frac{x}{\sqrt{2}}) \qquad (3.28)$$

These formulae can be translated directly into implementations:

[13] See [NAGC], functions *s15abc* (*nag_cumul_normal*) and *s15acc* (*nag_cumul_normal_complem*).

```
frozen cumulative_normal (x: DOUBLE): DOUBLE
        -- cumulative Normal distribution
    do
        Result := 0.5 * erfc(-x/Sqrt2)
    end
frozen cumulative_normal_complement (x: DOUBLE): DOUBLE
        -- complement of cumulative Normal distribution
    do
        Result := 0.5 * erfc(x/Sqrt2)
    end
```

3.3.12 Elliptic integrals

An elliptic integral is an integral of the form [14]

$$\int \frac{A(x) + B(x)\sqrt{S(x)}}{C(x) + D(x)\sqrt{S(x)}}dx \qquad or \qquad \int \frac{A(x)}{B(x)\sqrt{S(x)}}dx \tag{3.29}$$

where $A(x)$, $B(x)$, $C(x)$ and $D(x)$ are polynomials in x and $S(x)$ is a polynomial of degree 3 or 4.

> “Elliptic integrals can be viewed as generalizations of the inverse trigonometric functions and provide solutions to a wider class of problems. For instance, while the arc length of a circle is given as a simple function of the parameter, computing the arc length of an ellipse requires an elliptic integral. Similarly, the position of a pendulum is given by a trigonometric function as a function of time for small angle oscillations, but the full solution for arbitrarily large displacements requires the use of elliptic integrals. Many other problems in electromagnetism and gravitation are solved by elliptic integrals.” [14]

The NAG C library defines four specific elliptic integrals: [15]

$$R_C(x, y) := \frac{1}{2}\int_0^\infty \frac{dt}{\sqrt{t+x}(t+y)} \tag{3.30}$$

$$R_D(x, y, z) := \frac{3}{2}\int_0^\infty \frac{dt}{\sqrt{(t+x)(t+y)(t+z)^3}} \tag{3.31}$$

$$R_F(x, y, z) := \frac{1}{2}\int_0^\infty \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}} \tag{3.32}$$

$$R_J(x, y, z, \rho) := \frac{3}{2}\int_0^\infty \frac{dt}{(t+\rho)\sqrt{(t+x)(t+y)(t+z)}} \tag{3.33}$$

They are all offered by *SPECIAL_FUNCTIONS*:

[14] See: http://mathworld.wolfram.com/EllipticIntegral.html

[15] See [NAGC], functions *s21bac* (*nag_elliptic_integral_rc*), *s21bcc* (*nag_elliptic_integral_rd*), *s21bbc* (*nag_elliptic_integral_rf*) and *s21bdc* (*nag_elliptic_integral_rdj*).

frozen *elliptic_integral_rc (x: DOUBLE; y: DOUBLE): DOUBLE*
-- Rc(x,y) special case of elliptic integral

frozen *elliptic_integral_rd*
(x: DOUBLE; y: DOUBLE; z: DOUBLE): DOUBLE
-- Rd(x,y,z) symmetrised elliptic integral of the second kind

frozen *elliptic_integral_rf*
(x: DOUBLE; y: DOUBLE; z: DOUBLE): DOUBLE
-- Rf(x,y,z) symmetrised elliptic integral of the first kind

frozen *elliptic_integral_rj*
(x: DOUBLE; y: DOUBLE; z: DOUBLE; r: DOUBLE):
DOUBLE
-- Rj(x,y,z,r) symmetrised elliptic integral of the third kind

But since they are not implemented in the standard C99 math library, the comments that have been made for Airy functions also apply to the elliptic integrals.

3.3.13 Exponential, cosine and sine integrals

Again, different authors use different definitions of these integrals. EiffelMath naturally follows the NAG C library definitions: [16]

$$exp.\ integral \quad : \quad E_1(x) := \int_x^\infty \frac{e^{-u}}{u} du, \qquad x > 0 \tag{3.34}$$

$$cos.\ integral \quad : \quad Ci(x) := \gamma + ln(x) + \int_0^x \frac{cos(u) - 1}{u} du, \quad x > 0 \tag{3.35}$$

$$sine\ integral \quad : \quad Si(x) := \int_0^x \frac{sin(u)}{u} du \tag{3.36}$$

where γ denotes Euler's constant.

Again, all three integrals are offered by *SPECIAL_FUNCTIONS*:

frozen *exp_integral (x: DOUBLE): DOUBLE*
-- exponential integral E1(x)

frozen *cos_integral (x: DOUBLE): DOUBLE*
-- cosine integral Ci(x)

frozen *sin_integral (x: DOUBLE): DOUBLE*
-- sine integral Si(x)

[16] See [NAGC], functions *s13aac* (*nag_exp_integral*), *s13acc* (*nag_cos_integral*) and *s13adc* (*nag_sin_integral*).

But since they are not implemented in the standard C99 math library, the comments that have been made for Airy functions also apply to the exponental, cosine and sine integrals.

3.3.14 Fresnel integrals

The NAG C library provides two more integrals involving cosines and sines:

$$fresnel\_c \quad : \quad C(x) := \int_0^x cos(\frac{\pi}{2}t^2)dt \tag{3.37}$$

$$fresnel\_s \quad : \quad C(x) := \int_0^x sin(\frac{\pi}{2}t^2)dt \tag{3.38}$$

Both are offered by *SPECIAL_FUNCTIONS*:

> ***frozen*** *fresnel_c (x: DOUBLE): DOUBLE*
> -- Fresnel Integral C(x)
>
> ***frozen*** *fresnel_s (x: DOUBLE): DOUBLE*
> -- Fresnel Integral S(x)

And again, since they are not implemented in the standard C99 math library, the comments that have been made for Airy functions also apply to the Fresnel integrals.

3.4 Conclusion

The statistics on *SPECIAL_FUNCTIONS*' 43 features do not look too promising:

- Only 13 can be implemented through direct calls to the standard C math library.
- Only 3 more can be implemented through calls to existing functions.
- 27 functions will have to be implemented separately, which will be difficult.

Paul Dubois' estimation of just *how* difficult a re-implementation of special functions will be is very pessimistic [Dubois]:

> "Implementing any of these functions requires very special knowledge not known by most numerical mathematicians. At one point when I had an employee who worked on this sort of thing he corresponded with I think two other people, both professors. If you think about it, who in their right mind would write a thesis today in special functions? There is no research to do. No company has any need for such a person. No educational institution would give you a degree or tenure. Without access to the NAG source or some equivalent at Netlib, you are stuck with just the published algorithms. But the published algorithms are notoriously not the whole story. Some of these required time/space tradeoffs that may or may not still be valid, and there were heuristics that are not reported in journals."

Paul Dubois advises clearly against re-implementation of special functions, but for the search of an alternative to NAG at Netlib. A quick search of Netlib shows that there are only a handful of special functions packages written in FORTRAN and only one package in C: The CEPHES Mathematical Function Library [CEPHES]:

> "...a collection of more than 400 high quality mathematical routines for scientific and engineering applications. All are written entirely in C language. Many of the functions are supplied in six different arithmetic precisions:
>
> - 32 bit single (24-bit significand)
> - 64 bit IEEE double (53-bit)
> - 64 bit DEC (56-bit)
> - 80 or 96 bit IEEE long double (64-bit)
> - extended precision formats having 144-bit and 336-bit significands.
>
> The library treats about 180 different mathematical functions. In addition to the elementary arithmetic and transcendental routines, the library includes a substantial collection of probability integrals, Bessel functions, and higher transcendental functions. There are complex variable routines covering complex arithmetic, complex logarithm and exponential, and complex trigonometric functions."[17]

All of *SPECIAL_FUNCTIONS*' features are provided by the CEPHES Library in single[18] as well as double[19] precision except Kelvin functions and the elliptic integrals *exactly* as defined by NAG. In addition, CEPHES offers a vast amount of functions not implemented by the NAG C library. However, CEPHES is offered for download as source code and would have to be compiled for the platforms the library is supposed to target. The implementation is not tuned for any specific CPU, which puts the same limitations on efficiency as would a high-level implementation in Eiffel. In addition, careful testing of the correctness of the implementations would have to be done since of course nothing is guaranteed about them. The advantage of CEPHES is that it is not under the terms of the GPL, but completely free to use [20].

In terms of mathematical areas, the statistics of what can be done with the standard C math library look a bit more favorable:

- 4 areas can easily be covered:
  - Hyperbolic functions and their inverses
  - Bessel functions
  - Gamma function

[17]Cephes package documentation: http://netlib2.cs.utk.edu/cephes/cephes.doc.

[18]Documentation of cephes single precision special functions: http://www.netlib.org/cephes/singldoc.html.

[19]Documentation of cephes double precision special functions: http://www.netlib.org/cephes/doubldoc.html

[20]See http://www.netlib.org/cephes/doubldoc.html for the exact terms of use.

  - Error functions and cumulative normal distribution

- 5 areas will need a separate implementation:
  - Modified Bessel functions
  - Airy functions
  - Kelvin functions
  - Incomplete Gamma functions
  - 9 basic integrals

The important question is whether we really need to provide them all at any cost. What distinguishes these particular 43 special functions from the thousands of other special functions that exist in the literature is that they have been chosen for implementation by NAG. Whether all the 43 functions offered by NAG really are of the same vital importance or whether some of them could be dismissed without great harm is a question I lack the experience to answer [21].

Since the Kelvin functions do not seem to be the most important, what can be accomplished with the CEPHES Library should be more than enough.

Starting the design from the rock-bottom, a question related to the possibility of dismissing some of EiffelMath's special functions is whether we should include any additional special functions not provided by NAG. At least the functions provided at no cost by the C99 standard math library should be included:

- Bessel functions *jn* and *yn* have been mentioned above.
- *atan*2 is used for example in solid mechanics for the computation of polar coordinates from cartesian coordinates: It takes two arguments x and y — not just their ratio like the normal *atan* function — and returns the correct angle in the range $[-\pi, \pi]$, not just $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

In addition, for many of the functions offered by the CEPHES Library providing an interface in our library might be worth considering.

As I have already mentioned in the introduction part, I have probably spent too much time on special functions. I get more and more convinced that their importance for many mathematical areas is not that vital. They seem to be only loosely connected to the other numerical components we would want to offer. An implementation would have to be provided in the end, but I do not believe anymore that they would have top priority. Having reconsidered their importance, I believe that the possible schemes I have been able to show for their implementation are promising enough.

[21]The question seems to be difficult indeed, as highlights the comment made by Paul Dubois, a man with more than 30 years of experience in Scientific Computing [Dubois]: “I don’t know who could answer such a question. I believe from my limited experience that Airy functions are the most general case and are necessary for a lot of other ones. The incomplete Gamma and Modified Bessel are important.”

Chapter 4

Integration: A Playground for Agents

As mentioned in the introduction part, integration was chosen for re-implementation because it seemed ideally suited for a demonstration of the power of the new Eiffel agent mechanism. An agent is a function wrapped in an object, something like a high-level function pointer with all the benefits of being an object, like static type checking.

Assume you have a function

```
my_function (x: DOUBLE): DOUBLE
```

somewhere in your program text in a class *A* and you have in some other class a reference *a* to an object of type *A*. You can now create an agent of your function by writing

```
my_agent := agent a.my_function
```

my_agent is a normal object which can be sent as an argument to features of other objects. Assume that in the same class you have an object of some integrator type:

```
my_integrator: SOME_INTEGRATOR

...

create my_integrator.make
```

You can now send your agent to the integrator:

```
my_integrator.set_integrand(my_agent)
```

my_integrator will assign your agent to its *integrand* attribute. At some later time, when you want to integrate the function over a certain interval, you simply call

```
my_integrator.integrate
```

which will evaluate the integrand many times by calling the agent's *item* feature:

```
current_value := integrand.item([current_node])
```

The writer of the integrator class does not know anything about the function to be integrated except that it is a function of one argument of type *DOUBLE* returning a value of type *DOUBLE*. But he can provide for calls to this function through calls of feature *item* on the agent object. The writer of the client class (the class that uses the functionality of the integrator class) on the other hand does not have to know at which nodes the integrator object will evaluate the function. He does not have to specify the arguments himself. He just creates an agent of his function and sends it to the integrator object without specifying the arguments to the function. The arguments are not specified until the function is actually called through the agent object. Because of this gap (spacial in the programm text and temporal in the program execution) between the denomination of the function and its actual call with specified arguments, agents are sometimes referred to as 'delayed calls'. [1]

If you have never been exposed to object technology before, something might have struck you as even more fascinating than the agent mechanism itself: The fact that you can set the integrand with one call to *set_integrand* on the *my_integrator* object and then ask the integrator to integrate the integrand at some later time by calling another feature *integrate* on it without any arguments. If you come from the C and NAG world, you are used to call integration routines ('functions' in the programming language sense of the word) with up to twenty or more arguments, some of them function pointers to other functions with their own arguments each. And the worst thing is: Every time you call the function, you have to specify the whole set of arguments again. This is especially annoying if all the arguments but one remain constant between two calls, for example if you want to perform the same integration routine over the same integrand and interval just with a slightly higher precision. Object technology puts a halt to this annoying and even dangerous redundancy: Objects are *instantiations* of modules (classes) with a *state* which is persistent between calls. If you want to integrate several functions over the same interval, you just exchange the integrand between calls to *integrate* and leave the interval unchanged. And for many of the arguments of the old C style functions which are rather options than operands [2], a sensible default can be provided by the integrator class. So you only have to set the attributes for which you wish something other than the default. This allows the library designer (the writer of the integrator class) to provide facilities for extreme fine-tuning for expert library users about which the novice users will not have to worry at all because they will just work with the default settings [3].

[1]There's a lot more to the agent mechanism than the features introduced here which are the ones of greatest interest to our project. To get the full picture, see [ETL3].

[2]For object technology principles like command-query separation or the option principle, see [OOSC-2].

[3]For more information on the principles of library design, see [Base].

4.1 A first Approach: View Inheritance Type Hierarchy

For some clusters of EiffelMath, Paul Dubois ([OTSC]) explains the internal design and motivates the decisions that led to it. Not so for the integration cluster ('integrat'). He only presents integrators from a user's perspective: He demonstrates how to instantiate them, how to set their properties, how to call their features.

I had to browse through his classes and generate inheritance diagrams in trying to re-engineer the underlying design of the framework for one-dimensional integration. I first removed the NAG-specific parts. Then I got rid of some auxiliary classes which had served to provide an own high-level function pointer, which had been necessary because Eiffel had not included agents at the time Paul had written EiffelMath. When I had removed all I considered to be implementation-specific corruption of the original design, I was left with what is shown in figure 4.1.

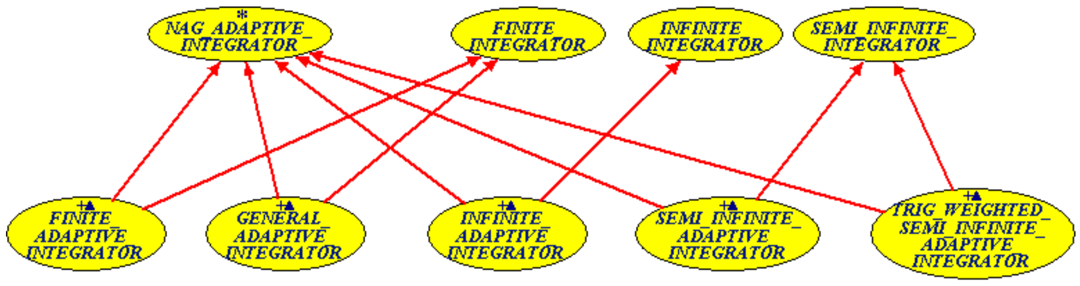
NAG_ADAPTIVE_INTEGRATOR
FINITE_INTEGRATOR
INFINITE_INTEGRATOR
SEMI_INFINITE_INTEGRATOR
FINITE_ADAPTIVE_INTEGRATOR
GENERAL_ADAPTIVE_INTEGRATOR
INFINITE_ADAPTIVE_INTEGRATOR
SEMI_INFINITE_ADAPTIVE_INTEGRATOR
TRIG_WEIGHTED_SEMI_INFINITE_ADAPTIVE_INTEGRATOR

Figure 4.1: Integrator hierarchy in EiffelMath (small purified extract)

Class *NAG_ADAPTIVE_INTEGRATOR* is ancestor to all integrators that use an adaptive-step (as opposed to fixed-step) integration scheme. The other three classes at the same level (*FINITE_*-, *SEMI_INFINITE_*- and *INFINITE_INTEGRATOR*) are called 'mix-in classes' in the documentation. Clemens Szyperski explains mix-in classes in his book ([Szyperski], pg. 101):

> "The idea is that a class inherits interfaces from one superclass and implementations from several superclasses, each focusing on distinct parts of the inherited interface. The latter classes are called *mixins*, as they are used to mix implementation fragments into a class."

If I understand Szyperski right, *NAG_ADAPTIVE_INTEGRATOR* should be viewed on an equal level with the other three classes: It mixes its implementation with that of one of the other three classes. To get the full picture, I gave it two peers by adding a class for fixed-step and a class for random methods. EiffelMath contains similar classes, but they are not treated on the same level. I ended up with two groups of three classes, each focusing on distinct parts of integration: the integration scheme (fixed-step, adaptive-step or random) and the interval over which to integrate (finite, semi-infinite or infinite). To structure the six mix-in classes in these two groups, I added an individual parent class to each group: *INTERVAL_INTEGRATOR* and *METHOD_INTEGRATOR*. To make the hierarchy fully compliant with Szyperski's definition of mix-in, I added a top level class *INTEGRATOR* from which the parents of the two the mix-in groups inherit. That's the class Szyperski calls 'interface' in the Java world. EiffelMath also contains a very high-level *INTEGRATOR* class, but not all integrators inherit from it. The result of my re-engineering can be seen in figure 4.2: The mix-in classes can be combined in any possible way because each group *focuses on distinct parts of the inherited interface.* The top-level class is *INTEGRATOR*: There are different ways of classifying integrators:

- according to the integration 'method' or 'scheme' they use:
  - fixed-step
  - adaptive-step
  - random
- according to the type of interval over which they operate:
  - finite
  - semi-infinite
  - infinite

The three classes in each classification cover the whole set of integrators fully and without overlapping. The two classifications are orthogonal or independent *views* on integrators. I see a strong connection between View Inheritance and mix-in. [4]

When this design was presented to the rest of the group, it was criticized for several reasons. One reason was that it is a mere type hierarchy, structuring integrators into groups, but not factoring out common behaviour. This is true: The top four layers of classes only implement features for setting parameters and contain invariants to guarantee some consistency properties, but the main feature *integrate* is completely deferred unto the lowest level (which consists only of three classes in figure 4.2 because the approach was not pursued further). It is true that this design only benefits from half of the advantages of object technology: It does enforce polymorphism with the possibility of substituting integrators within the same sub-group. And because of the stress on *view*, it even offers two possible grouping criteria with the possibility of polymorphism within each. But it does not reuse any substantial amount of code. The 'flaw' of deferring the whole implementation unto the lowest level is of course inherited from EiffelMath, where Paul Dubois has had no other choice than to create mere type hierarchies and delegate the implementation to the NAG C library at the lowest possible level.

This was a clear sign the the 'EiffelMath purification approach' was wrong in the attempt of creating a purely object-oriented library. The real problem with EiffelMath is *not* the corruption at the lowest level through some technical NAG peculiarities. The *real* problem is that its design is only object-oriented on the surface, a mere type hierarchy, a beautiful gown to let the user forget that it is still old ugly NAG inside. It already offers most of the benefits of object technology to the user, but not to the maintainer or the extender. If we want *everybody* to enjoy the pleasures of object-orientation, we have to start from scratch by finding useful abstractions and factoring out common behaviour.

[4] In fact, the top three levels in figure 4.2 look exactly like the figure illustrating View Inheritance on pg. 853 of [OOSC-2] to me, if I replace *INTEGRATOR* with *EMPLOYEE*, *INTERVAL_INTEGRATOR* with *CONTRACT_EMPLOYEE*, *METHOD_INTEGRATOR* with *SPECIALTY_EMPLOYEE*, etc.

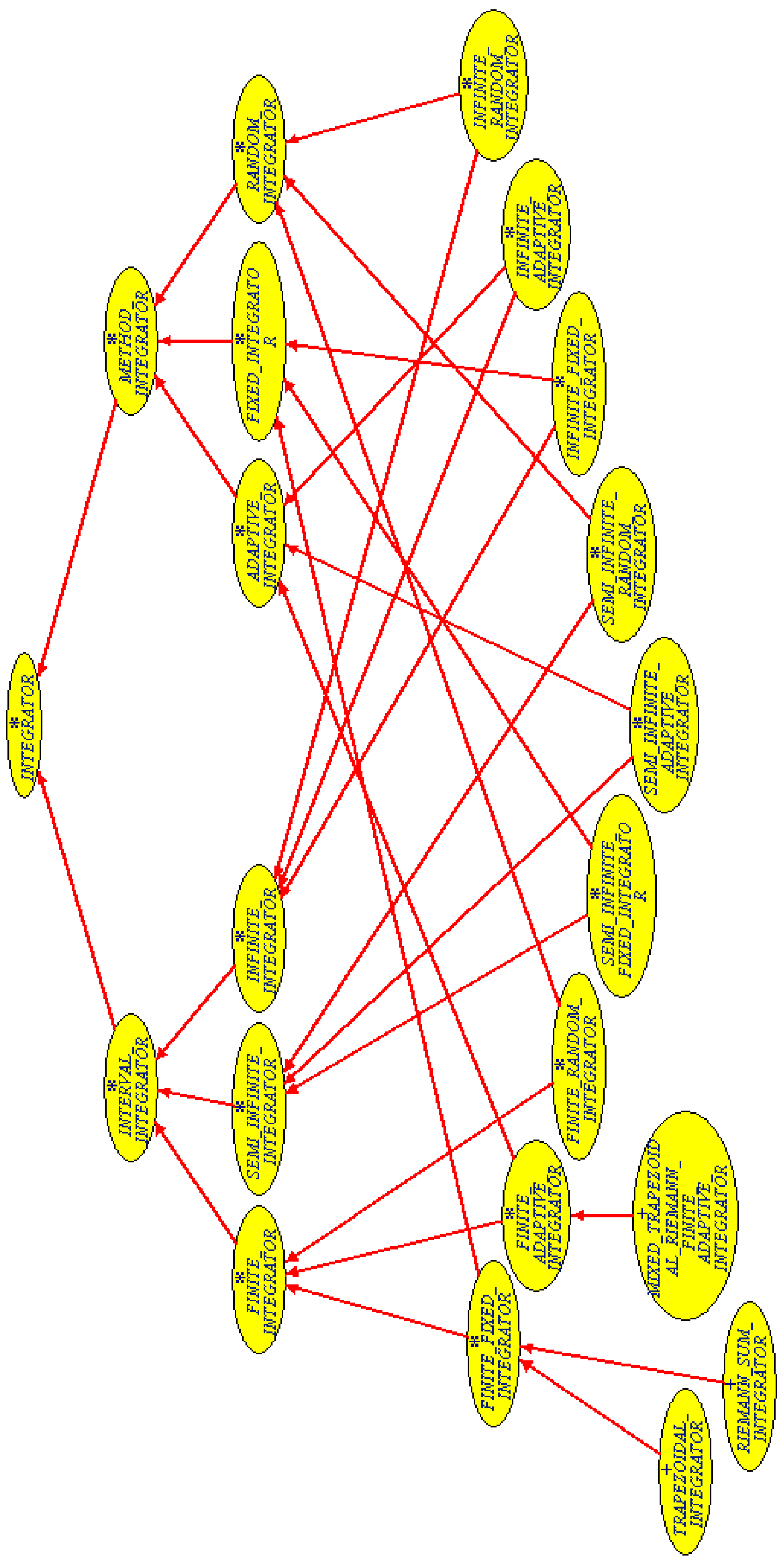
*INTEGRATOR
*METHOD_INTEGRATOR
*INTERVAL_INTEGRATOR
*RANDOM_INTEGRATOR
*FIXED_INTEGRATOR
*ADAPTIVE_INTEGRATOR
*INFINITE_INTEGRATOR
*SEMI_INFINITE_INTEGRATOR
*FINITE_INTEGRATOR
*INFINITE_RANDOM_INTEGRATOR
*INFINITE_ADAPTIVE_INTEGRATOR
*INFINITE_FIXED_INTEGRATOR
*SEMI_INFINITE_RANDOM_INTEGRATOR
*SEMI_INFINITE_ADAPTIVE_INTEGRATOR
*SEMI_INFINITE_FIXED_INTEGRATOR
*FINITE_RANDOM_INTEGRATOR
*FINITE_ADAPTIVE_INTEGRATOR
+MIXED_TRAPEZOIDAL_RIEMANN_FINITE_ADAPTIVE_INTEGRATOR
*FINITE_FIXED_INTEGRATOR
+RIEMANN_SUM_INTEGRATOR
+TRAPEZOIDAL_INTEGRATOR

Figure 4.2: View inheritance type hierarchy for integrators

4.2 Start from Scratch: An Integrator is an Iterator

How do you find a good abstraction? There does not seem to be a recipe. You look at your problem, you look at it somewhat longer; you look at other things, at your problem again; you try something, try something else; then you talk it over with your friends, think it over again, and in the course of all this, you might come up with something useful. If you are lucky enough to have found something good, it will probably look very simple in retrospect and make you wonder why it has taken you so long to find it.

What is an integrator? It was Bernd Schoeller, one of the PhD students in Bertrand Meyer's group, who first suggested that an integrator is nothing more than a *dispenser*, just giving you point after point according to its own rules which contain the knowledge that constitutes the integration 'scheme' or 'method'.

I was not quite convinced yet, and so I asked a related question: What is *integration*? Integration is continuous summation, i.e. the summation of an infinite number of infinitesimal terms. And what is *numerical* integration? Numerical integration replaces continuous by discrete summation:

$$\int_a^b f(x)dx \qquad \longrightarrow \qquad \sum_{i=1}^{N} w_i * f(x_i) \tag{4.1}$$

The expression on the right side of the arrow is what is usually called a *weighted sum*: a summation of terms $f(x_i)$, each multiplied by (or weighted with) a corresponding weight w_i. So Bernd was actually right! An integrator *is* a dispenser: It is a little machine with a button (called *forth*). Every time you press this button, the machine gives you two things: a *value* (the result of an evaluation of the integrand at some node which is secret to the machine) and a *weight*. All you have to do is multiply the two and add the product to the integral. How the current value and the current weight are computed is the secret of the little machine: It is what distinguishes it from other integrator machines who also return a weight and a value when their *forth*-button is pressed. It is the machine's concrete integration scheme. The *forth* feature and the fact that a weight and a value are returned when it is pressed is the behaviour they all have in common. It is a very general description of numeric integration. It is an *abstraction* for what is going on.

This immediately translates into the following feature *integrate* in our top-level *INTEGRATOR* class:

```
integrate is
    require
        integrand_set: integrand /= Void
        interval_set: interval /= Void
    do integral := 0
        if
            interval.length /= 0
        then
            from
                start
            until
                done
            loop
                forth      -- updates current_weight and current_value
                integral := integral + current_weight * current_value
            end
            finish
        end
    end
```

It now becomes obvious that an *integrator* is an *iterator*: A machine that iterates over an interval, choosing points along the way according to its inner knowledge, weighing each and summing them up.

The code shown above is a fully effective feature: No concrete integrator class down the inheritance hierarchy will ever have to touch it again; all they need to do is to effect the features that are called by *integrate* and left without implementation at this highest level of abstraction: *start*, *forth*, *done* and *finish.* These are the four features that will contain the knowledge of the concrete integration scheme. It is one of the benefits of object technology that this beautiful abstraction is not rotting away somewhere in a long forgotten design document, but that it is preserved right here in our code.

The Eiffel feature shown above mentions in its *require* clause two more features of the same class: *integrand* and *interval. integrand* refers to the function to be integrated. Our integrator abstraction is so general that it does not only apply to the integration of functions explicitly given by the program text of an Eiffel feature, but also of functions given by a table of (abscissa, ordinate)-pairs. These are just two different *representations* of the abstract mathematical notion of 'function'. The wish to treat both cases within the same code framework above creates the necessity of a completely abstract 'function' class of which agents are just one possible specialization, namely the specialization of functions given explicitly by program text. This leads to the class hierarchy shown in figure 4.3 [5]: An abstract class *NOOCL_FUNCTION* to designate a function at the highest level of abstraction and two concrete representations *FUNC-*

[5]The class hierarchy depicted in figure 4.3 is not part of the 'integrat' cluster: Since functions do not only appear in the context of integration, but in many other mathematical areas as well, they belong into a cluster of basic data structures available to the whole library. EiffelMath contains such a cluster named 'eifmath'. I kept this cluster but renamed it 'noocl' for 'Numeric Object-Oriented Component Library'.

TION_GIVEN_BY_PROGRAM_TEXT and *FUNCTION_GIVEN_BY_TABLE* [6]. They inherit *NOOCL_FUNCTION*'s interface but delegate the implementation to *FUNCTION[ANY, [DOUBLE], DOUBLE]* (an agent class) or *HASH_TABLE[DOUBLE, DOUBLE]*, respectively [7]. In more concrete descendants of the *INTEGRATOR* class, *integrand* will have to be redefined covariantly.

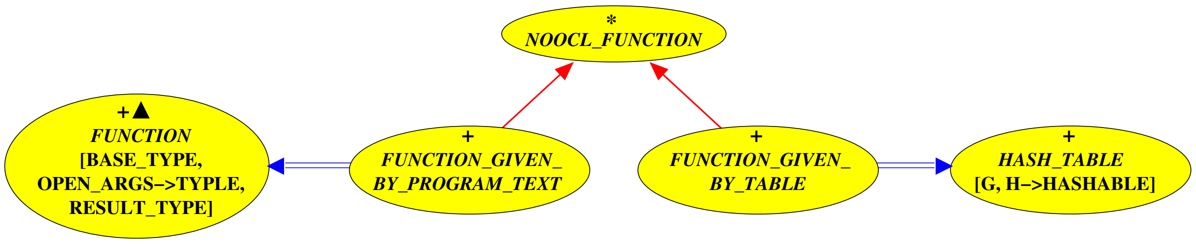

Figure 4.3: Class hierarchy of different function representations

The same considerations apply to the *interval* feature and lead to the class hierarchy shown in figure 4.4 [8].

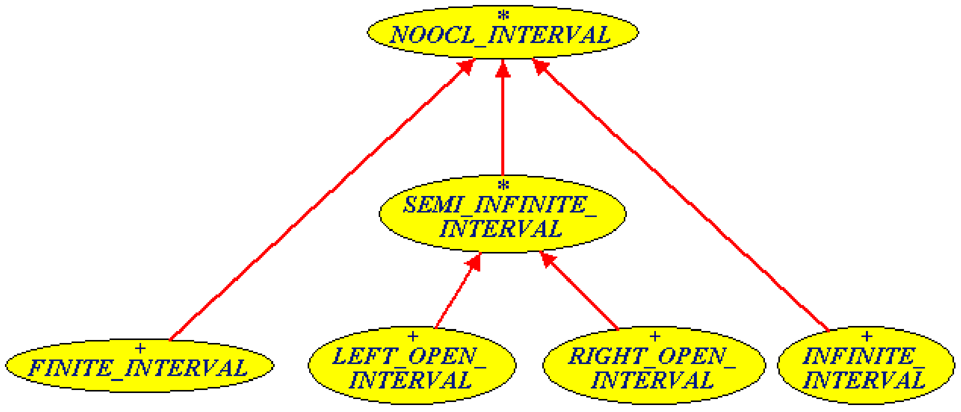

Figure 4.4: Class hierarchy of intervals

It is questionable whether the simple notion of 'interval' really justifies such a big abstraction. Some members of the group find it too much of an overhead for what is basically just two *DOUBLE*s (*lower_boundary* and *upper_boundary*). I have kept the interval hierarchy so far, because to me these intervals represent more than just two *DOUBLE*s: They are abstractions with well defined properties that can be expressed as class invariants, for example:

[6]Class names can be a subject of endless discussions. If you think that these two names are already too long, prepare for greater horrors to come ... I reject names like *TABULAR_FUNCTION* because the function itself is not tabular; it is just *represented* or *given* by a table.

[7]'Marriage-of-Convenience inheritance with *NOOCL_FUNCTION* as the noble and either *FUNCTION[ANY,[DOUBLE], DOUBLE]* or *HASH_TABLE[DOUBLE, DOUBLE]* as the rich partner would be an alternative if it were not for technical problems when inhering from *FUNCTION[ANY,[DOUBLE], DOUBLE]*

[8]To be found in the 'noocl' cluster as well.

- in class *NOOCL_INTERVAL*:

  lower_boundary <= upper_boundary

- in class *FINITE_INTERVAL*:

  length = upper_boundary – lower_boundary

  lower_boundary > – Infinity ***and*** *upper_boundary < Infinity*

- in class *SEMI_INFINITE_INTERVAL*:

  length = Infinity

  lower_boundary = – Infinity ***or else*** *upper_boundary = Infinity*

  lower_boundary > – Infinity ***or else*** *upper_boundary < Infinity*

- in class *INFINITE_INTERVAL*:

  length = Infinity

  lower_boundary = – Infinity ***and*** *upper_boundary = Infinity*

I am convinced of the necessity of *NOOCL_INTERVAL*, *FINITE_INTERVAL* and *INFINITE_INTERVAL*. Since no algorithm for semi-infinite intervals has been implemented yet, I am less sure about the absolute necessity of *SEMI_INFINITE_INTERVAL* or of its two specializations *LEFT_OPEN_INTERVAL* and *RIGHT_OPEN_INTERVAL*.

After the development of this high-level 'Integrator Machine' abstraction, the framework had to be applied to concrete integration algorithms to check whether it was general enough or whether it had to be generalized further. Once you have found an abstraction you like, you run the danger of viewing the world through the glasses of your abstraction and forcing every member of your problem set into its mold instead of adjusting the abstraction to a broader view of the problem set. To avoid this, you have to question the applicability of your abstraction carefully again every time you apply it to a new member of your problem set.

The rest of this section describes briefly how our framework has been applied successfully and — as I believe — without any forced distortions to a number of algorithms. For a more detailed look at individual classes, you should browse the code yourself. You will see that Eiffel code is very readable and almost self-documenting. In addition, the code has been commented abundantly and contains references to literature for the mathematical background, namely to [Schwarz] and [Gander].

The full class hierarchy can be seen in figure 4.5.

4.2.1 Integrators with a fixed number of terms over a finite interval

The algorithm that springs first to most people's minds when they think of numerical integration is the trapezoidal method: The area below the graph of the integrand is

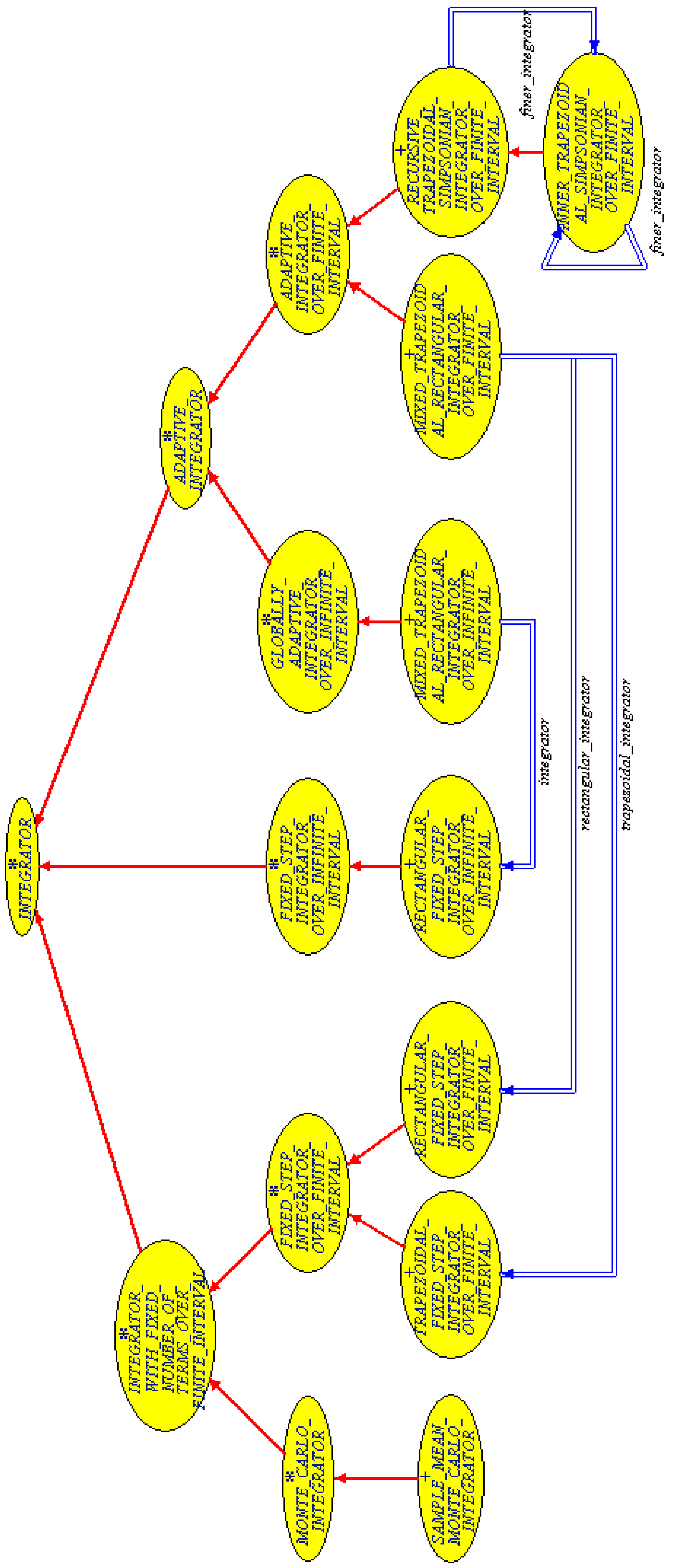
* INTEGRATOR
* ADAPTIVE_INTEGRATOR
* ADAPTIVE_INTEGRATOR_OVER_FINITE_INTERVAL
+ RECURSIVE_TRAPEZOIDAL_SIMPSONIAN_INTEGRATOR_OVER_FINITE_INTERVAL
INNER_TRAPEZOIDAL_SIMPSONIAN_INTEGRATOR_OVER_FINITE_INTERVAL
finer_integrator
finer_integrator
+ MIXED_TRAPEZOIDAL_RECTANGULAR_INTEGRATOR_OVER_FINITE_INTERVAL
* GLOBALLY_ADAPTIVE_INTEGRATOR_OVER_INFINITE_INTERVAL
+ MIXED_TRAPEZOIDAL_RECTANGULAR_INTEGRATOR_OVER_INFINITE_INTERVAL
integrator
rectangular_integrator
trapezoidal_integrator
* FIXED_STEP_INTEGRATOR_OVER_INFINITE_INTERVAL
+ RECTANGULAR_FIXED_STEP_INTEGRATOR_OVER_INFINITE_INTERVAL
+ RECTANGULAR_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL
* FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL
+ TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL
* INTEGRATOR_WITH_FIXED_NUMBER_OF_TERMS_OVER_FINITE_INTERVAL
* MONTE_CARLO_INTEGRATOR
+ SAMPLE_MEAN_MONTE_CARLO_INTEGRATOR

Figure 4.5: 'Integrator Machine' class hierarchy

approximated by a series of trapezoids of equal width:

$$step\_size := \frac{interval.upper\_boundary - interval.lower\_boundary}{total\_number\_of\_steps} \tag{4.2}$$

The approximation of the integral is

$$T(h) := step\_size[\frac{1}{2}f(lower\_boundary) + \sum_{j=1}^{total\_nuber\_of\_steps-1} f(x_j) + \frac{1}{2}f(upper\_boundary)] \tag{4.3}$$

where

$$x_j := lower\_boundary + j * step\_size \tag{4.4}$$

This algorithm can very well be split over the four features to be effected in *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL*: *start* initializes the loop counter, current weight and current node. *finish* takes care of the first and the last elements and scales the whole integral with *step_size*. Only the 'inner' elements are treated in the loop. This allows *current_weight* to be set constant to 1 for the whole loop in *start* and avoids costly conditional branching within *forth*:

```
start is
    do
        number_of_terms := 1
        current_node := interval.lower_boundary
        current_weight := 1
    end

forth is
    do
        number_of_terms := number_of_terms + 1
        current_node := current_node + step_size
        current_value := integrand.item([current_node])
    end

done is
    do
        Result := number_of_terms = (total_number_of_terms – 1)
    end

finish is
    do
        integral := integral
            + 0.5*(integrand.item([interval.lower_boundary])
                + integrand.item([interval.upper_boundary]))
        integral := integral * interval.length / total_number_of_steps
        number_of_terms := number_of_terms + 1
    end
```

A slight variation of the trapezoidal method is the 'midpoint sum' algorithm: This time the area below the graph of the integrand is approximated by a series of rectangular boxes of equal width. The hight of each box corresponds to the value of the integrand at the node in the middle of the subinterval which constitutes the lower box edge:

$$M(h) := step\_size \sum_{j=0}^{total\_nuber\_of\_steps-1} f(x_{j+\frac{1}{2}}) \tag{4.5}$$

where

$$x_{j+\frac{1}{2}} := lower\_boundary + (j + \frac{1}{2}) * step\_size \tag{4.6}$$

In the trapezoidal method, the number of terms was equal to the number of steps (subintervals) plus 1. In the midpoint sum or 'rectangular' approximation, the number of terms is now equal to the number of steps. Another difference is that the weight is now constant 1 for all terms. Therefore all terms can be evaluated within the loop; no special treatment for the boundary terms is necessary. This changes *start*, *finish* and *done* in *RECTANGULAR_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL* slightly:

```
start is
    do
        number_of_terms := 0
        current_node := interval.lower_boundary - 0.5 * step_size
        current_weight := 1
    end

done is
    do
        Result := number_of_terms = total_number_of_terms
    end

finish is
    do
        integral := integral * interval.length / total_number_of_terms
    end
```

But *forth* looks exactly the same as in *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR _OVER_FINITE_INTERVAL*. Our abstraction allows us to factor out common behaviour and move feature *forth* from *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER _FINITE_INTERVAL* up the inheritance hierarchy to a common ancestor of both *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL* and *RECTANGULAR_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL*: *FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL* (see figure 4.5).

RECTANGULAR_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL is more general than *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL*: It only shares *forth* with *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER _FINITE_INTERVAL*, but it shares *done* and *finish* with a large group of algorithms which all use a fixed number of terms (but not necessarily a *fixed step*, i.e. not necessarily a uniform step size) over a finite interval. Another subgroup of this large group of algorithms is random (or 'Monte Carlo') integration where a fixed number of nodes is chosen randomly from within the interval, evaluated, equally weighted and summed up. Changing the terms *group* and *subgroup* for *class* and *subclass* directly suggests moving both *done* and *finish* from *RECTANGULAR_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL* up the inheritance hierarchy to *INTEGRATOR_WITH_FIXED_NUMBER_OF_TERMS_OVER_FINITE_INTERVAL*, an ancestor common to all fixed-step and random-step classes. Imposing *done* and *finish* from *RECTANGULAR_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL* as default behaviour on *all* descendants of *INTEGRATOR_WITH_FIXED_NUMBER_OF_TERMS_OVER_FINITE_INTERVAL* does not prohibit that individual classes like *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL* redefine it according to their needs.

The bottom-up development of this leftmost major branch of the integrator hierarchy shown in figure 4.5 is similar to the development of the other branches seems to be typical for object-oriented library development: First a specific problem is solved (trapezoidal integration), then another specific task is accomplished (rectangular integration), then commonalities are factored out into a common ancestor class. Other algorithms are implemented and more commonalities found, and slowly but steadily the class hierarchy not only broadens but deepens at the same time.

4.2.2 The library user's perspective: Integrators as Service Providers

The main difference between the three major branches of the integrator hierarchy in figure 4.5 is best described from a library user's perspective: The library user only sees feature *integrate*; the four features that *integrate* calls internally are not exported to clients[9]. To the user, a completely effective integrator class is flat: He does not care how far up the class hierarchy a feature's implementation resides. He is not interested in the factoring out of common behaviour, all that matters to him is the final behaviour at the lowest level. However, he does group integrators by the interface they present to him, and this interface is determined by the number and kind of parameters that he can set before calling *integrate*. Of course he always has to set the interval and the integrand to even fulfill *integrate*'s precondition: There is no reasonable default for either of them. But apart from that, not all integrators offer the same possibilities.

Integrators with a fixed number of terms, as described in the last subsection, let the user fix this number of terms, which is of interest to him because it is about proportional to the execution time and it determines the precision of the approximation. If he then calls *integrate*, the integrator will compute an approximation of the integral with the predefined number of terms and put the result to its attribute *integral*. This result will always be a valid approximation of the integral, although arbitrarily inaccurate if the

[9]I use the two terms 'user' and 'client' more or less synonymously in this context. Actually the term 'user' refers to a *human* library user, whereas 'client' rather refers to client *code*. But here I am thinking of a human user working with the library directly, i.e. writing just one layer of client code which calls library features directly. Thus the two terms become almost interchangeable.

number of terms is small. But since no precision has been specified, every approximation is valid and allows *integrate* to fulfill its (implicit) contract: to approximate the integral according to the 'method' it implements and as precise as possible with the number of terms specified [10].

Integrators using a fixed-step method over an *in*finite interval (middle main branch in figure 4.5) evaluate an integrand (which should be rapidly decreasing on both sides) starting from a specified point and proceeding in both the positive and negative directions until the function values drop below a certain 'cut-off error' δ. To prevent *integrate* execution time from exploding if the integrand is not decreasing fast enough, the user can additionally set the maximum number of steps to be used. These are completely different parameters than were present in the leftmost branch of the class hierarchy. It is immediately evident to the user that the two sub-hierarchies are two different specializations of the general *INTEGRATOR* abstraction.

The user-settable parameters for integrators using a fixed-step method over an infinite interval [11] present a new problem: They actually allow the user to *overspecify* the contract by asking something impossible from the *integrate* feature: What is it supposed to do if the maximum number of subintervals has been reached and the function value at least on one side has still not dropped below δ? Two different schemes seem possible:

1. Let *integrate* put the value to *integral* nonetheless and set a corresponding error flag which it is the client's responsibility to check if he wants to be sure that the result he uses is not complete non-sense.

2. Throw an exception which it is the client's responsibility to check.

After some debate, the second scheme was chosen. The default for the maximum number of terms to be used (provided by the integrator class as supplier of the service) corresponds to the largest integer of the system the library is run on. If the user explicitly chooses value smaller than the default, he must know that he might be overspecifying the contract. If the client asks something of *integrate* which the feature is just not able to fulfill with the scarce resources granted, then *integrate* must be allowed to throw an exception.

The same considerations apply to the rightmost branch of the class hierarchy in figure 4.5: Adaptive integration. These classes allow the client to specify a certain precision ϵ to be reached in the approximation and adapt the step-size globally or locally to reach the specified precision. For the same reason as in the case of fixed-step integrators over infinite intervals, the user has the possibility of additionally specifying the maximum number of terms to be used as an indirect upper boundary for execution time. The same danger of overspecifying the contract arises, and the same measure has been taken to encounter it.

[10]See [OOSC-2] for the concept of contracts between software modules.

[11]If you read these descriptions, you may understand why I still defend the class names shown in figure 4.5 although they are certainly too long to be user-friendly: I am just reluctant to designate an integrator as being an infinite-fixed-integrator for the same reasons mentioned in the case of functions: the integrator itself is no more 'infinite' or 'fixed' than any other integrator: It is just integrating over an infinite *interval*, using a fixed *step size*.

4.2.3 Globally adaptive integration

In *globally* adaptive integration, the step-size (starting with *interval.length*) is *uniformly* decreased until the specified precision has been reached.

[Schwarz] describes an algorithm in which the step size is successively cut in half:

> Let T(*step_size*) := an integral computed with trapezoidal method
> and M(*step_size*) := the same integral computed with rectangular method
>
> for the *same step_size*, then the relation
>
> $$T(\frac{step\_size}{2}) = (\frac{1}{2})[T(step\_size) + M(step\_size)] \qquad (4.7)$$
>
> holds.

This relation allows the improvement of the trapezoidal approximation by successively cutting the step size in half by computing M(*step_size*) and adding it to T(*step_size*) for a given *step_size*. For each cutting in half of the step size, the execution time approximately doubles, but the function values that have already been computed can economically be reused.

It is usually okay to stop the refinement as soon as

$$|T(step\_size) - M(step\_size)| <= \epsilon \qquad (4.8)$$

for a given tolerance $\epsilon > 0$, because then the relation

$$error = |T(\frac{step\_size}{2}) - I| <= \epsilon \qquad (4.9)$$

usually holds. I in this relation denotes the exact integral.

At first sight, this algorithm seems to break our abstraction: In the end, the approximation is a trapezoidal approximation as in the very first case we examined. But this time, the order of terms added is not from left to right anymore. The algorithm performs multiple passes through the interval, each time with a smaller step-size and ignoring the nodes already evaluated. This poses the problem that at the moment a term is added to the sum, its weight is yet unknown. Eq. (4.7) shows that if the precision demands another pass through the interval, all the terms that have already been added to the sum lose half their weight after the fact.

We can reconcile the algorithm with our abstraction by switching to a higher-level perspective: For the globally adaptive integrator, the current term to be added is not an individual evaluation of the integrand but the current rectangular approximation $M(step\_size)$, to be added to the last approximation $T(step\_size)$. Since our *INTEGRATOR*'s main loop

```
loop
    forth        -- updates current_weight and current_value
    integral := integral + current_weight * current_value
end
```

does not allow us to weight *integral*, but only *current_value*, eqn. (4.7) has to be scaled by a factor 2:

$$2T(\frac{step\_size}{2}) = T(step\_size) + M(step\_size) \tag{4.10}$$

Applied recursively, this yields:

$$\begin{aligned} 4T(\frac{step\_size}{4}) &= 2T(\frac{step\_size}{2}) + 2M(\frac{step\_size}{2}) && (4.11) \\ 8T(\frac{step\_size}{8}) &= 4T(\frac{step\_size}{4}) + 4M(\frac{step\_size}{4}) && (4.12) \\ &\cdot \\ &\cdot \\ &\cdot \\ 2^kT(\frac{step\_size}{2^k}) &= 2^{k-1}T(\frac{step\_size}{2^{k-1}}) + 2^{k-1}M(\frac{step\_size}{2^{k-1}}) && (4.13) \end{aligned}$$

The first approximation T is calculated in feature *start* and assigned to *integral*. An approximation M is added in every loop cycle, i.e in every call to *forth*. Feature *finish* has to scale the integral as usual, this time by a factor 2^{k-1}, where k corresponds exactly to the number times the loop has been executed and *forth* has been called. Because floating-point numbers are stored as a mantissa and an exponent (of base 2), and because the scaling in this algorithm only has to be done with powers of 2 (corresponding to an increase or decrease of the exponent, but leaving the mantissa and thus the precision unchanged), the scaling should not cause any numeric problems.

The algorithm described has been implemented for finite intervals in class *MIXED_TRAPEZOIDAL_RECTANGULAR_INTEGRATOR_OVER_FINITE_INTERVAL*. Of course the class does not compute the individual trapezoidal and rectangular approximations itself: in a fully object-oriented approach, the class creates an instance of *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL* and an instance of *RECTANGULAR_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL* and lets them do the work. Within an object-oriented library, code is not only reused through inheritance, but also through client-supplier-relationships between individual classes. The library user is thus not the only client to the library: Parts of the library may well be clients to other parts, as the horizontal arrows at the bottom of figure 4.5 show.

For infinite intervals, the algorithm has been implemented in *MIXED_TRAPEZOIDAL _RECTANGULAR_INTEGRATOR_OVER_INFINITE_INTERVAL*.

4.2.4 Locally adaptive integration

The algorithm presented in the last subsection can also viewed with a slightly different focus: Two approximations were computed for each step size. The results were compared, and if the difference was above a given tolerance ϵ, both approximations were recomputed with half the step size. The step size was halved by doubling the number of steps over the entire interval.

Another possibility of halving the steps is to cut the whole interval into two equal halves and perform the two initial integration schemes on each half-interval (with the same number of steps initially applied to the whole interval now applied to each half-interval).

At first sight this looks like exactly the same thing: Globally, the number of steps has been doubled, and the steps have all been cut into halves. The partition of the initial interval is still uniform. But there is a gain in information: Now we have two approximations for each half-interval which we can compare individually. If the precision is good enough for one half but not for the other, only the the half that is not precise enough will be subdivided further. This is what is called *locally* adaptive integration.

In principle, any two different approximation techniques could be used. For global adaption, trapezoidal and rectangular method were chosen because of their nice property that the average of the two directly yield the next-finer trapezoidal approximation. This property is of no use for local adaptation.

The most simple example to demonstrate the principle uses trapezoidal approximation and simpsonian approximation which is given by:

$$\begin{aligned} simpsonian &:= \frac{1}{3}(T + 2 \cdot step\_size \cdot f(center)) && (4.14) \\ where\ \ center &:= \frac{1}{2}(inteval.lower\_boundary + interval.upper\_boundary) && (4.15) \end{aligned}$$

As for the stopping criterion of the algorithm [Gander] observes that the interval can successively be cut into halves until for a given subinterval,

$$(trapezoidal + coarse\_approximation) = (simpsonian + coarse\_approximation) \quad (4.16)$$

holds in machine arithmetic, where *coarse_approximation* is a coarse approximation of the whole integral. This criterion has the effect that small contributions (in absolute value) do not need to be computed with a small relative error, and that the final approximation of the whole integral is obtained with almost full machine accuracy.

If the client only wishes to approximate the integral with a certain $relative\_precision > 0$, this can be achieved by setting

$$coarse\_approximation := coarse\_approximation * relative\_precision / Machine\_precision$$

where $Machine\_precision$ is the smallest positive DOUBLE representable on your machine, such that

$$1 + Machine\_precision \neq 1 \quad (4.17)$$

[Gander] contains the following first sketch of a recursive ‘divide and conquer’ algorithm [12]:

[12] [Gander] uses Pascal syntax. The reproduction here uses Eiffel syntax and variable names which are consistent with the rest of the text.

```
integral(lower_boundary, upper_boundary: DOUBLE): DOUBLE is
    local
        trapezoidal, simpsonian, center: DOUBLE
    do
        step_size := upper_boundary – lower_boundary
        trapezoidal := 0.5 * step_size *
            (integrand.item([lower_boundary]) + integrand.item([upper_boundary]))
        center := 0.5 * (lower_boundary + upper_boundary)
        simpsonian := (1/3) * (trapezoidal + 2*step_size*integrand.item([center]))
        if
            (trapezoidal + coarse_approximation) = (simpsonian + coarse_approximation)
        then
            Result := simpsonian
        else
            Result := integral(lower_boundary, center)
                    + integral(center, upper_boundary)
        end
    end
```

What is recursive ‘divide and conquer’ in a functionally decomposed program translates into a binary object tree in an O-O runtime structure: Every object contains references to objects of the same class:

```
class FIRST_SKETCH_OF_A_RECURSIVE_INTEGRATOR
feature -- Parameters
    trapezoidal: DOUBLE
    simpsonian: DOUBLE
    center: DOUBLE
    integrator: FIRST_SKETCH_OF_A_RECURSIVE_INTEGRATOR
feature {NONE} -- Implementation
    integrate is
        do
            trapezoidal := . . .
            center := . . .
            simpsonian := . . .
            if
                (trapezoidal + coarse_approximation)
                    = (simpsonian + coarse_approximation)
            then
                Result := simpsonian
            else
                create integrator.make
                integrator.set_integrand(integrand)
                integrator.set_relative_precision(relative_precision)
                integrator.set_interval(create
                        {FINITE_INTERVAL}.make(interval.lower_bound, center))
                integrator.integrate
                **Result** := integrator.integral
                integrator.set_interval(**create**
                        {FINITE_INTERVAL}.make(center, interval.upper_bound))
                integrator.integrate
                **Result** := **Result** + integrator.integral
            end
        end
    end -- class FIRST_SKETCH_OF_A_RECURSIVE_INTEGRATOR
```

The final implementation in *RECURSIVE_TRAPEZOIDAL_SIMPSONIAN_INTEGRATOR_OVER_FINITE_INTERVAL* is slightly more complex than this first sketch for two reasons:

1. Although feature *integrate* in the top-level *INTEGRATOR* class is not ***frozen*** and could thus be redefined by an individual descendant, I didn't want to make use of this possibility because my main interest was exactly to find out whether the framework provided by *INTEGRATOR*'s *integrate* could be retained even for a recursive algorithm. It could indeed be retained, but then the recursion runs not only over one feature *integrate*, but over the conglomerate of the five tightly coupled individual features *integrate*, *start*, *forth*, *done* and *finish*.

2. The first sketch above would not be an efficient implementation, because the support abscissae of the integrand that have already been evaluated would have to be evaluated again and again [13]. If these values can be passed on to the child node in the object tree, the child will only have to evaluate one new node instead of three: It receives the values of the lower and upper boundaries and only has to compute the value of the new center. This is true for all nodes except the root of the tree: It will not receive any precomputed values and will have to evaluate all three support abscissae itself.

   Two schemes were implemented for this distinction between root and non-root objects:

   (a) In environments like 'Message Passing Interface' (MPI), all processors run the same programm, but one of them is the 'king' which distributes the work load to all the other processors (and to itself). In the actual work phase, all 'instances' of the programm run the same code, but in the distribution and collection phases, the behaviour differs, based on whether the instance is 'king' or not. The program therefore has the following structure:

```
feature execute_some_task_in_parallel is
    do
        if king then
            -- Distribute workloads evenly among all processors.
        end
```

[13]The 'support abscissae' are called 'nodes' in the rest of this report. [Schwarz] uses the two terms synonymously. Short English names are usually preferable to long Latin names. However, in the description of this class the longer Latin name is preferable simply to avoid any confusion with the 'nodes' of the object tree described in the same context.

```
        -- Do your share of the work. (Everybody, including the king.)

        if king then
            -- Collect the results from all the processors.
        else
            -- Send your partial result to the king.
        end
    end
```

The code in the first implementation of *RECURSIVE_TRAPEZOIDAL_SIMPSONIAN_INTEGRATOR_OVER_FINITE_INTERVAL* makes a similar distinction of whether an instance (a 'node') is the 'root' or not. A root can be created by any client. The root will then create all the the other nodes to build up the binary tree, through a creation routine which is exclusively exported to the class itself. The root will itself do all the evaluations of the integrand it needs, whereas all the other nodes will receive the values that have already been computed (and which are of interest to this particular node) in the creation routine.

(b) The second version of *RECURSIVE_TRAPEZOIDAL_SIMPSONIAN_INTEGRATOR_OVER_FINITE_INTERVAL* implements this difference in behaviour through two different classes: The class of 'root' objects which can be created by any client and the class of 'inner' nodes which is a descendant of the root class and whose creation routine is only exported to the class itself and to the root class. No external client may hold a reference to an inner node. The descendant class redefines *start*, where the evaluation of the boundary abscissae is omitted, and in turn adds setter methods for the precomputed boundary values.

The second implementation has been chosen for inclusion with the library. Its design seems more compliant with object technology: Conditional branching based on the runtime subtype of an object is exactly the way object-oriented software should *not* be built. And in the end, 'root' is nothing more than a specialization of the more general 'node'. The recursive client-supplier relationship between the two classes can be seen in the lower right corner of figure 4.5.

4.3 The Important Question: Is it General Enough?

The general 'integrator machine' framework has successfully been applied to a number of integration algorithms, without any forced distortion, except maybe in the last case of a recursive algorithm. This last scheme would have been easier to implement in one single routine *itegrate*, i.e. without any implementation framework at all, like in the case of EiffelMath or the EiffelMath-based view inheritance type hierarchy. But if an individual algorithm is too 'exotic', nothing prevents it from ignoring the framework by redefining *integrate*. This should not be an argument for not providing a framework in the first place.

INTEGRATOR's main loop

```
loop
    forth       -- updates current_weight and current_value
    integral := integral + current_weight * current_value
end
```

might even be simplified to

```
loop
    forth       -- updates current_term := current_weight * current_value
    integral := integral + current_term
end
```

The explicit weighting of the term within the main loop is illustrative to the abstraction of numeric integration as weighted summation, but it is irrelevant to the implementation. Integration might just as well be regarded simply as summation, the weighting of the terms being packed into *forth* and hidden from the main loop in *integrate*.

Further research will have to show how well the abstraction applies to even more classes of numeric integration, like for example to schemes not only involving evaluations of the integrand, but also of its derivatives.

Part III

Evaluation

Chapter 5

Evaluation of the Sample Components

This chapter evaluates the work on the two sample components described in the second part of this document. It tries to assess the benefits, drawbacks, challenges and open issues in a fully object-oriented approach to Scientific Computing. Importance is given to careful performance analysis.

5.1 Benefits of a Full O-O Approach to Scientific Computing

5.1.1 Special Functions

As the name suggests, special functions are *functions* not only in the mathematical, but also in the programming language sense of the word. Even in an object-oriented environment, they will be implemented as function calls. No benefit can be gained from object technology.

5.1.2 Integration

Integration is an area where the benefits of object technology can be seen on various levels:

- As already described by Paul Dubois in [OTSC] and briefly recapitulated at the beginning of chapter 4 of this diploma thesis, an object-oriented library provides a much simpler interface to its *clients* through modules (classes) that can be *instantiated* and have a *state* that is persistent between consecutive calls, allowing them to 'remember' parameters between consecutive calls and to provide defaults for options.
- As shown in section 4.2, the *full* O-O approach offers the full benefits of object technology not only to clients, but also to the library designers, programmers and

maintainers: The possibility to factor out common behaviour of related algorithms, leading to reusable code and facilitating maintenance and extension.

5.2 Possible Drawbacks, Challenges and Open Issues

5.2.1 Special Functions

According to Paul Dubois ([Dubois]), the challenge of actually re-*implementing* special functions in an object-oriented language is so big, that he strongly advises against it. Considering that object technology has no benefits to offer in this area, it would probably be a useless investment. In a deviation from our overall approach, a new library should only wrap existing implementations in this area. If the original goal is kept of freeing the library from the NAG implementation in order to make it freely distributable, the challenge will consist in finding equivalent implementations. Chapter 3 mentions the CEPHES Library as a possible alternative.

After all, I am unsure of the importance of special functions to the whole of a numeric library.

5.2.2 Integration

No drawbacks have been encountered in the partial implementation of an integration class hierarchy as described in section 4.2. Since there are still many schemes of numeric integration which it has not been possible to explore in the course of this time-limited project, the issue is still open whether the abstraction that has been developed is general enough to encompass the whole field. If not, the challenge will be to come up with a better abstraction.

5.3 Performance Analysis

5.3.1 Special Functions

If existing C implementations for special functions are reused, the performance overhead of calling the corresponding function once is negligible.

5.3.2 Integration

Extensive performance analysis has been performed on one exemplary integration algorithm: Trapezoidal integration over a finite interval. The implementation of *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER_FINITE_INTERVAL* described in subsection 4.2.1., which will be called 'noocl' or 'nOOcleus'[1] makes heavy use of object

[1]'nOOcleus' is the acronym for 'Numeric Object-Oriented Component Library: Eiffel Used for Science'.

technology. For comparison, this implementation has been gradually stripped down along Paul Graham's 'power line' in the following steps:

Unwrapped agents

As shown in figure 4.3, the agent class *FUNCTION[ANY,[DOUBLE], DOUBLE]* has been wrapped with an additional class *FUNCTION_GIVEN_BY_PROGRAM_TEXT* in order to treat it within the same framework as functions given by a hash table. The 'Unwrapped agents' implementation is equivalent to 'noocl' except for not wrapping the agent in an additional class.

Classic Eiffel

The 'Classic Eiffel' implementation consists of a class hierarchy equivalent to 'noocl', but does not use agents. Agents are a relatively new contribution to the Eiffel language, and their effect on performance can thus be isolated. In 'classic' Eiffel, *integrand* is declared as a deferred feature which has to be effected by the library user in an own class which inherits from the class the library provides for trapezoidal integration. Note that this approach necessitates more work from the library user: He has to write a special class which inherits from the library class he wants to use. He has to write the integrand explicitly into the text of this new class. With agent technology, he could just call a function already written down somewhere in an arbitrary class, wrap it in an agent and send it to an instance of an unmodified class as provided by the library.

Modular Eiffel (Eiffula-2)

As shown in figure 4.5, *TRAPEZOIDAL_FIXED_STEP_INTEGRATOR_OVER_FINITE _INTERVAL* inherits from three generations of ancestors to which feature calls have to be dynamically dispatched [2]. The 'Modular Eiffel' implementation is a completely flat class. It can be instantiated, which still fits into the purely modular approach: Modula-2 allows you to write modules that contain functions and a record which can be instantiated. You can then call the module's functions to manipulate attributes of the record, just like you call an an object's features to manipulate its state. You could still write the main *integrate* loop in such a module and let it dispatch the work to four other features like *start*, *forth*, *done* and *finish*. But this is not the approach taken here: Apart from features to set parameters, the module contains just a single function named *integral* which does all the computation itself and returns the result directly to the client. This is almost like the implementation resulting from the view inheritance approach described in section 4.1.

The only deviation from the modular approach that the 'Modular Eiffel' implementation takes despite its name is that *integrand* is a deferred feature which the user has to effect in an own descendant class as in the 'classic Eiffel' approach. This result in one

[2]In an approach that differs greatly from the policy chosen by C++, all binding is dynamic in Eiffel by default. Binding cannot be influenced by the programmer in the program text, but a compiler option can allow the compiler to automatically bind feature calls statically where this is safely possible.

dynamically dispatched feature call (to *integrand* per evaluation of the integrand in this evaluation, as opposed to five in all higher-level approaches (to *start*, *forth*, *done* and *finish* and from *forth* to *integrand*).

Procedural Eiffel (EIFTRAN)

The 'Procedural Eiffel' approach is the semantics of C in the disguise of Eiffel syntax: No extra parameter setting and no persistent state, but a single routine *integral* which takes all the necessary parameters as input arguments, computes the integral approximation and returns it to the client. No class to be derived and no function to be effected: The integrand is a function directly written into the program text of the integration algorithm (some people call this 'hard-coded' or 'hard-wired') [3].

C

Semantically the same as 'Procedural Eiffel', but now in its workday dress of C syntax, with the full beauty of braces and the '+='-operator. A single *.c file with a main test routine that calls the routine that performs the integration. The integrand is hard-coded as a macro[4].

Assembly language

Real hard-core IA32 assembly language, nothing high-level like MASM[5] or HAL[6]. Since the program contains neither recursive function calls nor conditional branchings, all variables can be declared global and static in a single stack frame.

The program makes full use of the numeric coprocessor's stack of registers for floating point numbers. The variables that are most often needed are kept in a register during the whole program execution.

Model Problem and Test Systems

The model problem consists of 100 runs of the following problem:

Approximation of the integral of

$$f(x) = \exp(0.1x)\sqrt{x}\cos(2x) \tag{5.1}$$

over the interval (1, n) where n is initially equal to 1 and incremented by 1 in each run. The number of terms is 10'000, which means that the function is evaluated 1 million times in the whole model problem:

[3]Actually the function text resides in a class of test functions from which the integrator class inherits. But since this is a case where the compiler can bind the call statically, it comes down to what I have described above.

[4]Macros are expanded by the preprocessor. This corresponds to function *inlining*. The Eiffel approach differs again: There is no ***inline*** keyword, but a compiler option can allow the compiler to automatically inline functions depending on their size and the number of calls.

[5]Microsoft® Macro Assembler

[6]'Higher Level Assembler', see [AoA].

```
 from
     counter := 0
until
     counter = 100
loop
     counter := counter + 1
     create interval.make(0, counter)
     integrator.set_interval(interval)
     integrator.set_total_number_of_terms(10000)
     integrator.set_integrand(create
          FUNCTION_GIVEN_BY_PROGRAM_TEXT.make(agent finite_test_function))
     integrator.integrate
     integral := integrator.integral
end
```

The model problem has been run on two different test systems:

1. Intel® Mobile Pentium® III CPU 933 MHz
   128 MB Memory
   Microsoft® Windows 2000, Version 5.0: Build 2195: Service Pack 2
   Microsoft® 32-bit C/C++ Optimizing Compiler Version 13.00.9466 for 80x86
   ISE EiffelStudio 5 (5.2.1123 Enterprise Edition)
   SmallEiffel The GNU Eiffel Compiler Release -0.74
   Microsoft® 32-bit C/C++ Optimizing Compiler Version 13.00.9466 for 80x86
   Options: /Ox /Og /Ob2 /Oi /Ot /Oy /GT /G5 /GA /D "WIN32"
   /D "_DEBUG" /D "_CONSOLE" /D "_MBCS" /FD /MLd
   /GS /J /Yu"stdafx.h" /Fp"Debug/Trapezoidal_integration.pch"
   /Fo"Debug/" /Fd"Debug/vc70.pdb" /W3 /nologo /c /Wp64 /TC

2. Intel® Pentium® 4 CPU 1.80 GHz
   512 MB Memory
   GNU/Linux (Red Hat 8.0), Kernel 2.4.18-24.8.0
   ISE EiffelStudio 5 (5.2.1402 Enterprise Edition)
   SmallEiffel The GNU Eiffel Compiler Release -0.74
   gcc 3.2
   options: -O3 -pipe

ISE EiffelStudio automatically calls the underlying C compiler with the appropriate options in finalization mode. On Linux, SmallEiffel automatically calls gcc. On Windows, it would call gcc on a GNU emulation as provided by CYGWIN or MinGW[7]. To get a fair performance comparison, this was not the facility used here: SamllEiffel's C-output was manually compiled with the Microsoft® C/C++ Compiler instead. The same options were used as for the compilation of the direct C code. The assembler program was only run on Linux using gcc.

[7]See http://www.cygwin.com and http://www.mingw.org for more information.

Results

The model problem was run ten times in a row and the resulting execution time divided by 10. Start-time was taken after system start-up right before the first problem execution run. End-time was taken after completion of the last run. The whole procedure was repeated five times and the average taken. The tests are very well reproducible. The standard deviation is less than 1%. Figures 5.1–5.4 show a graphical representation of both the absolute as well as the relative execution times, where fully optimized C code is taken as the reference.

Interpretation

The measurements are consistent across both test systems, and they are also consistent across Eiffel compilers, with the exception of the ISE implementation of the agent mechanism, which seem to have some fundamental problem which must be solved by the compiler implementors and can be solved, as the SmartEiffel[8] implementation shows [9].

Apart from ISE agents, the results look promising:

- Procedural Eiffel can even be faster than direct C[10]. This is in accordance with the fact that even in my best attempt, I was unable to beat fully optimized C with my direct assembly program. Unless you really are an expert hacker in the lower-level language, you shouldn't try to beat the compiler of the next higher-level language.
- For SmartEiffel code,
  - the cost of object technology is less than 2%
  - the cost of the agent mechanism is less than 15%
- ISE code is usually slower, but the overhead is less than 30% and all Eiffel implementations are faster than unoptimized C code.

Additional tests to be done

As mentioned in the introduction to this report, the reference language for Scientific Computing is not C but FORTRAN. An implementation in FORTRAN 77 should therefore be done. Since I was initially working under Microsoft® Windows only, it would

[8]'SmallEiffel' has been renamed 'SmartEiffel', but the name of their latest compiler release is still 'SmallEiffel The GNU Eiffel Compiler'. That's an inconsistency I reproduce in my text on purpose in a desire for utmost correctness.

[9]The SmartEiffel implementation does not fully comply to the official Eiffel syntax specification provided by the Nonprofit International Consortium for Eiffel (NICE). It is for example not possible to pass an object of type *TUPLE[DOUBLE]* as an argument to *item*. An argument of type *DOUBLE* has to be passed instead, explicitly enclosed by manifest braces. I don't know whether there is a connection between this restriction and the striking performance results, but this might well be worth looking at for the compiler implementors at ISE.

[10]If I saw results like these in somebody else's report, the first suspicion to come to my mind would be that the tests were not implemented fairly in order to market the proposed solution. All I can say to that is that I did execute the tests unprejudiced and according to my best knowledge. Anybody is welcome to submit a faster C implementation.

have been difficult to find a good native FORTRAN compiler. Now that I have gotten used to GNU/Linux, I could do it with the g77 compiler. To make the test fair, I would have to get some experience with the language first. Since I don't think the performance will differ greatly from my C and assembler implementations, it had no priority as part of this diploma project. But it should of course be done for completeness.

A comparison to EiffelMath and to NAG should be done. Unfortunately, we only got a working EiffelMath distribution too late in the project for this analysis to be included. NAG's and thus EiffelMath's implementations are so diverse and so complex that it would take some time to get an exemplary algorithm from both EiffelMath and nOOcleus that would be comparable.

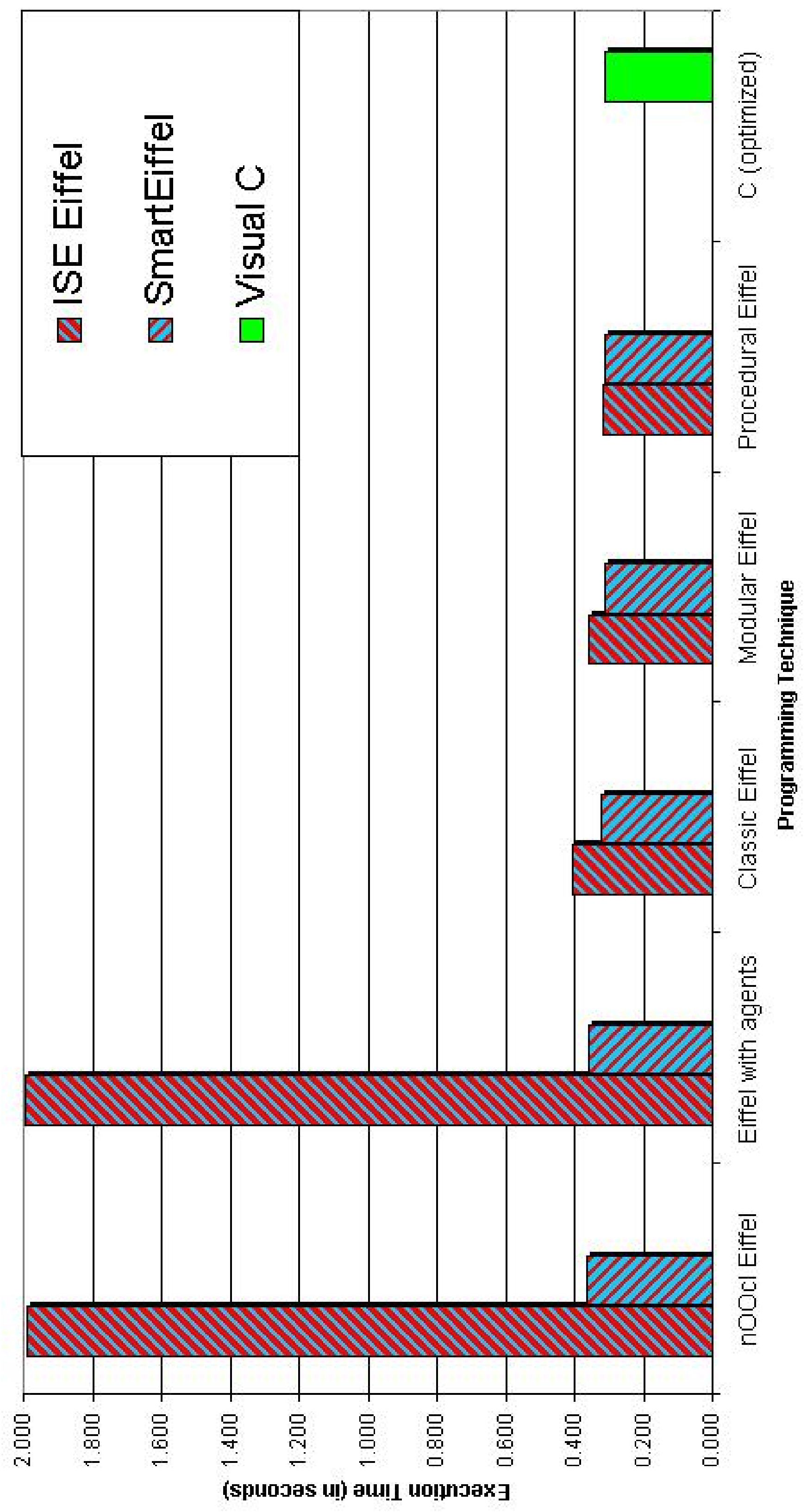

Figure 5.1: Absolute execution times on system 1

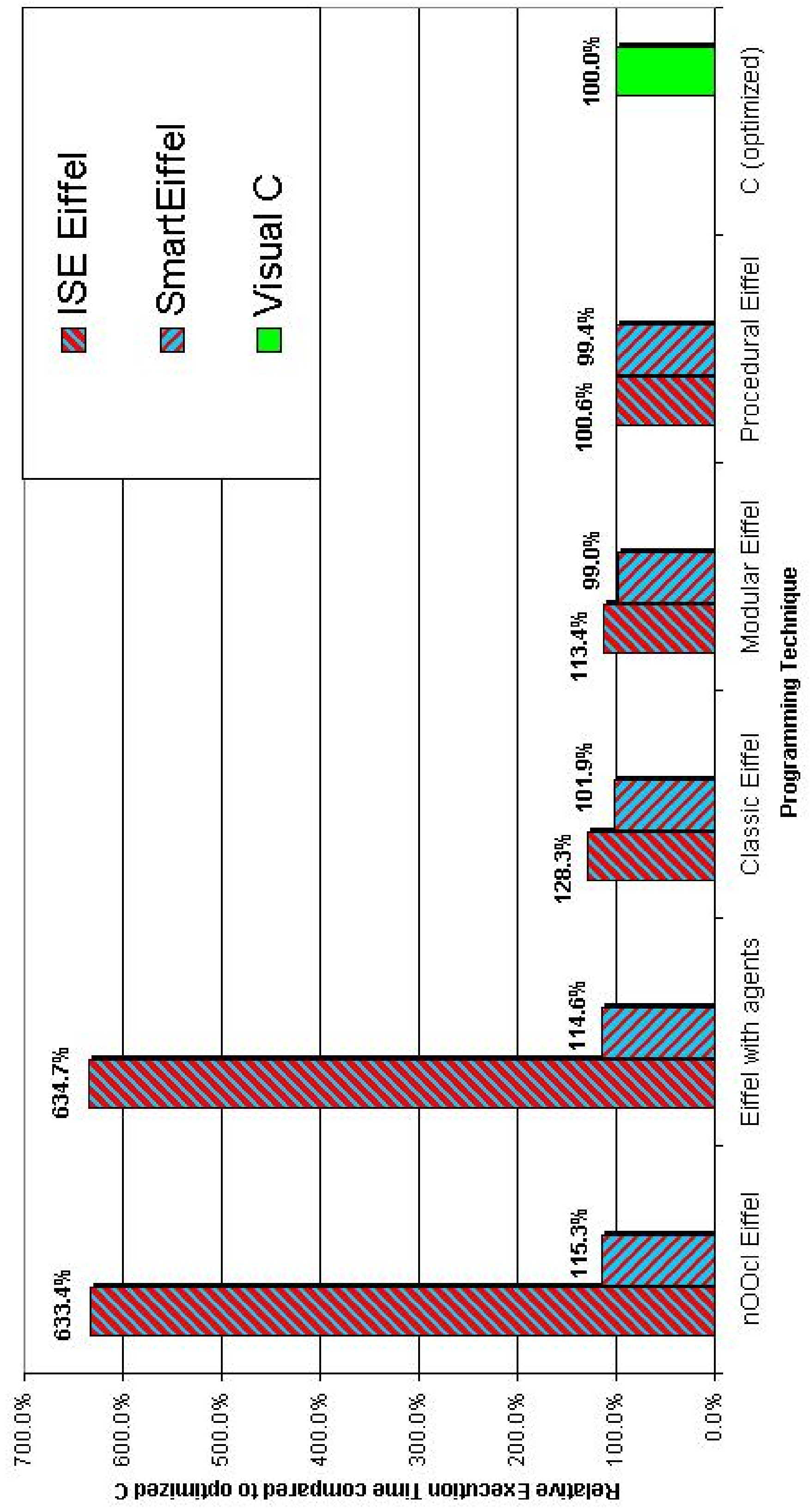

Figure 5.2: Relative execution times on system 1

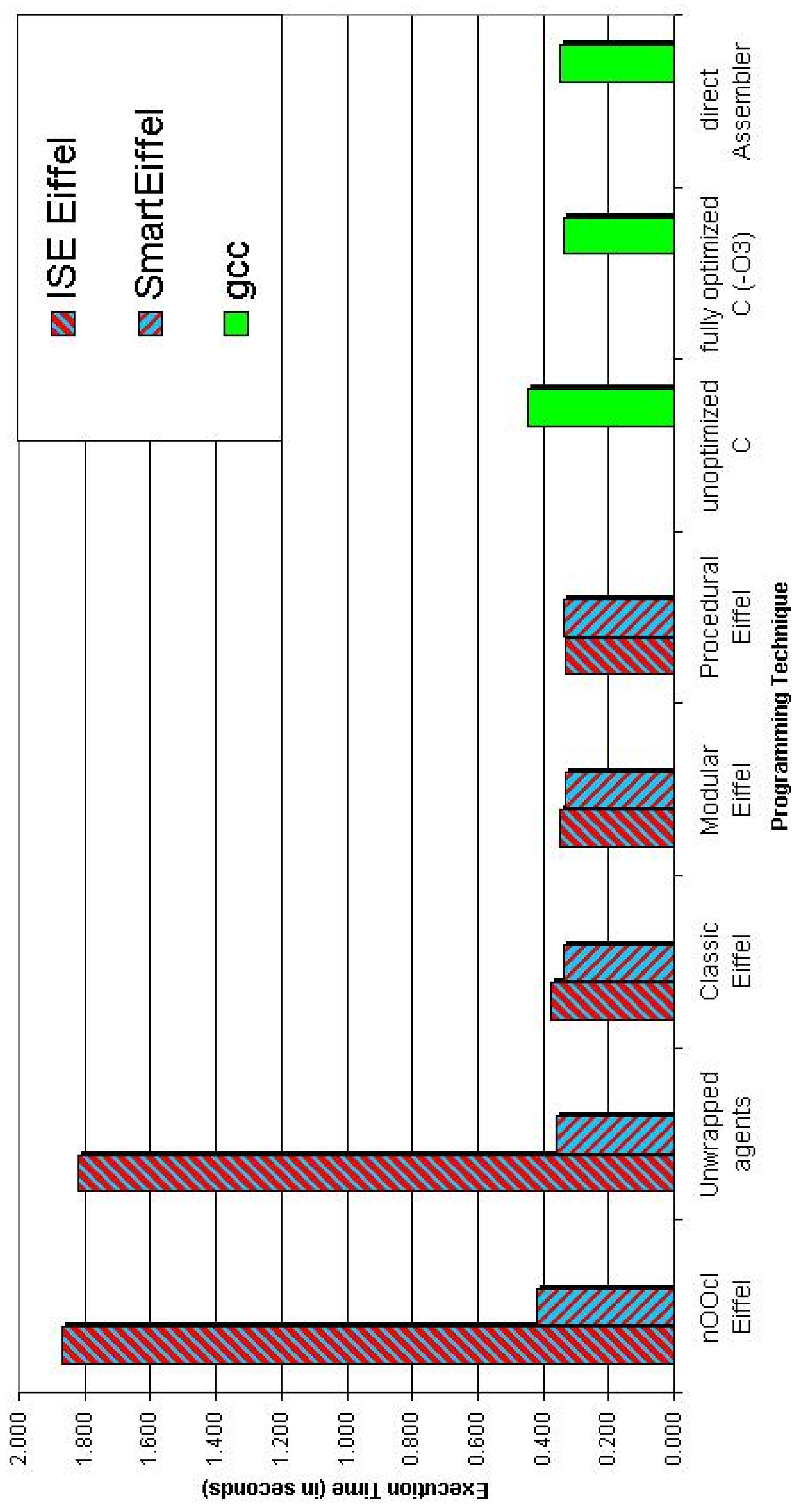

Figure 5.3: Absolute execution times on system 2

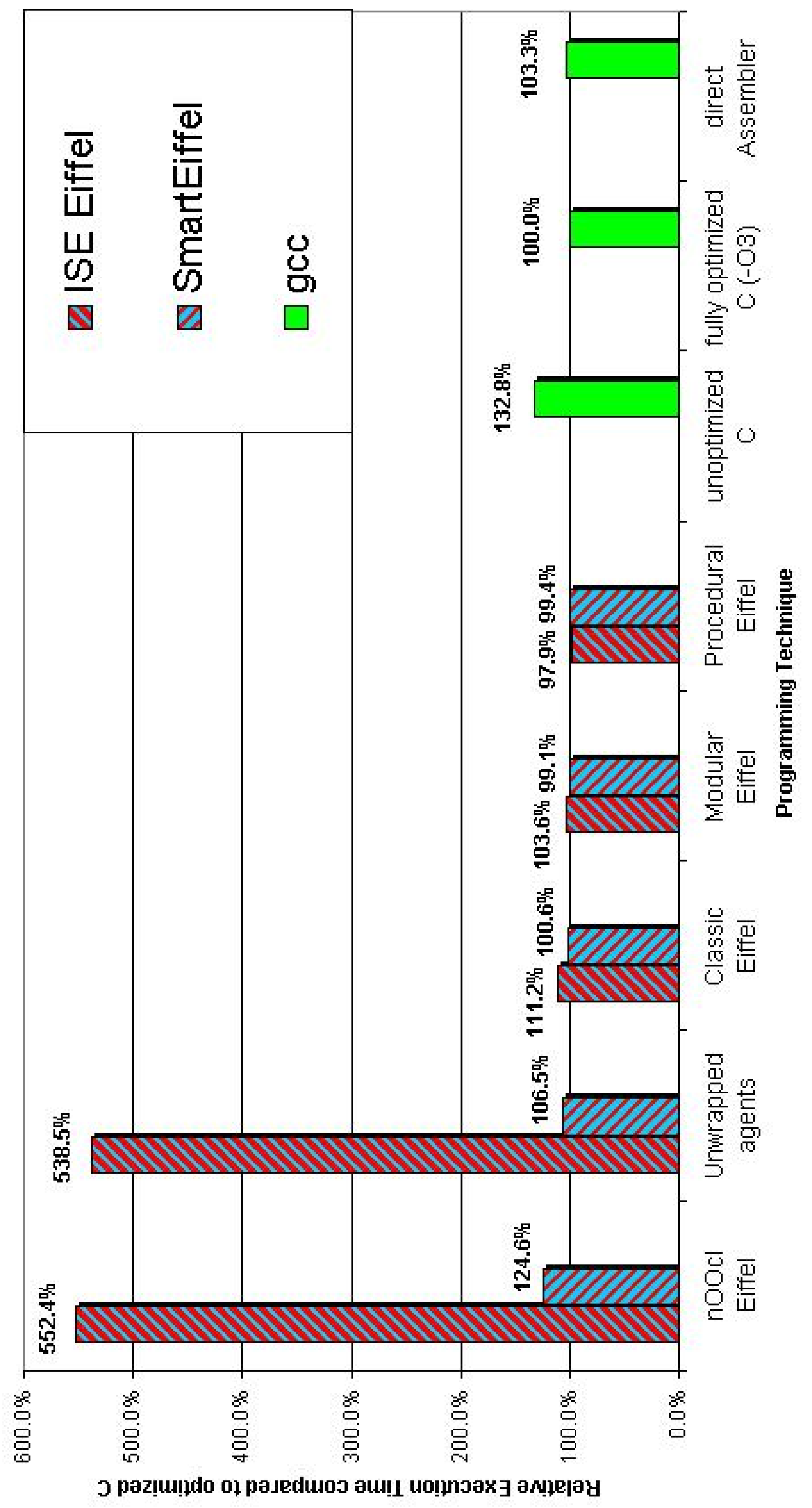

Figure 5.4: Relative execution times on system 2

Chapter 6

Perspectives of a Full O-O Math Library

The final chapter attempts an extrapolation of the experience gained from the work on the two sample clusters and an assessment of the perspectives of a full object-oriented math library. Although an estimate of the work is difficult, some basic requirements for the launch of a concrete project will be described.

6.1 Feasibility, Advantages, Challenges

Considering the very first goal of freeing EiffelMath from underlying *commercial* implementations in order to make it freely distributable, I don't see any feasibility problems there. Considering the second goal of implementing a *purely* object-oriented library, I don't think that this would be feasible at any reasonable cost if the purity of the approach were turned into a religious issue.

To have spent as much time on special functions as on integration was maybe not that bad after all. It has given me some feeling for the broad variety of numerical areas: Some fields pose interesting conceptual problems, the work on which can lead to design innovations. Integration is one such area. Curve-fitting, interpolation and the solution of ordinary differential equations should be similar. On the other hand, all the areas that involve the solution of large systems and the manipulation of matrices will be a lot more difficult. Where 'number crunching' is more important than the understanding of abstract concepts, object technology may not have much to offer.

The computation of special functions should be delegated to legacy implementations in C. This can be done without harm, since the area offers no opportunity for object-oriented abstractions anyway. In other areas — like linear algebra, FFT and optimization — the compromise will hurt more: The delegation of the lowest-level computations to public-domain packages like BLAS, LINPACK or LAPACK might have a disastrous effect on the internal design of the library: It might be hard to come up with anything better than EiffelMath.

Before making the decision whether to favour a purely object-oriented or a more hybrid approach, it should be decided what users are to be targeted. There are applications where speed is less important: Scientists and engineers do a lot of small calculations every day. If you just want to quickly try an idea on a small example, the intuitiveness of the library's interface is much more important than whether the execution time is one or two seconds. On the other hand, if the library is to be competitive in high-performance computing areas like weather simulation, solvers for big systems should not be re-implemented.

Eiffel as a language and a method is not the performance problem. This has clearly been shown in chapter 5. Object technology is fast enough! The problem in high-performance computing of large systems may be the knowledge of the algorithms that the original implementors have taken home when they have retired. This point can hardly be emphasized enough. I will therefore repeat Paul Dubois' quote on special functions ([Dubois]):

> "Implementing any of these functions requires very special knowledge not known by most numerical mathematicians. At one point when I had an employee who worked on this sort of thing he corresponded with I think two other people, both professors. If you think about it, who in their right mind would write a thesis today in special functions? There is no research to do. No company has any need for such a person. No educational institution would give you a degree or tenure. Without access to the NAG source or some equivalent at Netlib, you are stuck with just the published algorithms. But the published algorithms are notoriously not the whole story. Some of these required time/space tradeoffs that may or may not still be valid, and there were heuristics that are not reported in journals."

If high performance is to be reached in all areas, it seems wise to allow a hybrid approach for components that fall more into the 'number-crunching' area and follow the pure approach where implementation is more straight-forward.

However, the development of a library *as purely* object-oriented *as possible* does seem a good long-term investment to me. As should have become clear from chapter 4, it offers a great opportunity to re-think existing algorithms from a completely different perspective and gain a whole new understanding of the topics involved. This may even lead to the discovery of new recipes for old numeric tasks. It will certainly be a stimulation for the field of numeric analysis.

6.2 Specification of Basic Platform Requirements

I don't see any reason at the moment why the library should not run on any system on which the Eiffel Base Libraries run. However, this brings us to the topic of compatibility between different Eiffel compilers. Not even the small test system for trapezoidal integration, developed under ISE EiffelStudio, could be compiled without modification under SmartEiffel. Other numeric components may need more sophisticated data structures from Base than the integrator hierarchy, making the problem only worse.

Serious consideration should be given to the idea of only using gobo[1] data structures to ensure compatibility with all major Eiffel systems.

6.3 Estimate and Modularizability of the Work to be Done

To give a precise estimate of the work to be done is very, very difficult.

I have tried to get an overview over the whole set of clusters offered by EiffelMath at the beginning of my four-months' diploma project. I have then worked on two individual components. Within just about one month, I have been able to come up with a system for numerical integration that is still very small, but of good quality and relatively high performance, and which can immediately be put to use.

To get substantial design work done in an area, not more than one component should be looked at in the course of a four-month's diploma or master's thesis. Work in groups of two is an alternative that should seriously be considered. If a design is talked about and defended and justified and started over again, it certainly improves.

The one month reserved for semester projects may not be enough to start a component. Good design cannot mature under the pressure of a short deadline. However, once a component is started and a basic framework has been provided (like the current state of the integration hierarchy), the focus will shift to the proof of applicability of this framework to alternative algorithms, to a further generalization of the underlying abstractions and to pure implementation. That's the stage where semester projects can be of interest and where experts can contribute their particular knowledge of single algorithmic details.

EiffelMath offers about 15 numeric clusters. They seem to be only loosely coupled, not only on the surface, but also 'behind the scenes'. This means that the development of a full library may well be split into individual clusters although some will of course require the services of others. It also means that *at least* 15 master's theses will be needed to get them all started, plus some some in reserve for fundamental problems that will pop up along the way, plus a steady flow of semester projects as well as a group of voluntary contributors to carry on.

6.4 A Possible Work Plan

If many components are to be started in parallel, the supervision overhead will be considerable. Ideally, this supervision should be taken by an assistant who does his PhD thesis in this area. Alternatively, voluntary contributors could take responsibility for individual clusters, and one of them could take overall-responsibility. Semester and diploma students come and go. It is important to ensure consistency by giving responsibility to voluntary contributors who are willing and capable of committing themselves for a few years rather than months.

[1]For more information see: http://www.gobosoft.com

'nOOcleus' should be launched as an open-source project and publicly announced. A document should be published that summarizes the project philosophy and goals and the basic design guidelines and quality requirements. A role model should be elaborated to separate responsibilities, and a process should be defined in which designs can pass through various levels of acceptance like

- experiment
- proposition
- consolidation
- freeze

or something the like. Components that are ready for use and public scrutiny should immediately be made available on a central components server.

As I believe that this library offers a wealth of interesting topics for diploma and semester projects, other institutes inside and outside ETH should be informed and encouraged to contribute.

6.5 Related Fields

6.5.1 SCOOP

Two members of the SE group are currently working on the SCOOP[2] project. The more 'number crunching'-oriented part of Scientific Computing offers many interesting fields for parallelization. Collaboration between the two areas could be beneficial to both. A parallel high-level implementation of some computationally intensive component like FFT could be explored.

6.5.2 Non-Functional Contracts

[Szyperski] describes how an underlying math library can break your graphics application by improving the precision of some algorithm in the new release, because higher precision comes at the price of longer execution time. The possibility of introducing non-functional contracts into the Eiffel language should therefore be explored, and again our library offers a rich set of fields where the concepts under development could be tested right away.

[2]'Simple Concurrent Object-Oriented Programming': See [OOSC-2], chapter 30

Bibliography

[Abramowitz] Milton Abramowitz and Irene A. Stegun (editors): *Handbook of Mathematical Functions with formulas, graphs and mathematical tables*, Dover Publications, 5th printing, 1968.

[ACM] The vast collection of articles published by the Association for Computing Machinery (ACM) is accessible for subscribers at http://portal.acm.org.

[AoA] Randall Hyde: *Art of Assembly Language Programming and HLA*, published online at http://webster.cs.ucr.edu.

[Base] Bertrand Meyer: *Reusable Software: The Base Object-Oriented Libraries*, Prentice Hall Object-Oriented Series, 1994.

[Bateman] Arthur Erdlyi et al., California Institute of Technology Bateman Manuscript Project: *Higher Transcendental Functions, based in part on notes left by Harry Bateman, Volume II*, McGraw-Hill, 1953–1955.

[Besset] Didier H. Besset: *Object-Oriented Implementation of Numerical Methods. An Introduction with Java & Smalltalk*, Morgan Kaufman, 2000.

[C89MATH.H] C89/C++98 standard math library header file as implemented in Microsoft Visual Studio .NET version 7.0.9466.

[C99MATH.H] C99 standard math library header file as implemented at Red Hat (The Red Hat newlib C Math Library, libm 1.10.0, July 2002) by Steve Chamberlain, Roland Pesch, Red Hat Support, Jeff Johnston: http://sources.redhat.com/newlib/libm_toc.html#TOC1. This implementation does not set all the error flags as specified in the Open Group Specification (http://www.opengroup.org/onlinepubs/007908799/xsh/math.h.html).

[CEPHES] CEPHES Mathematical Function Library: http://netlib2.cs.utk.edu/cephes.

[Dijkstra] Edsger Dijkstra: 'Go To Statement considered harmful', *Communications of the ACM*, Vol 11 (2), pp. 147–148.

[DDJ] Dr. Dobb's Journal, February 2002: *Programming Languages*, article by Dan Nagle: *The FORTRAN 2000 standard.*

[Dubois] Paul Dubois in an e-mail sent to the author on December 6, 2002.

[Eckel] Bruce Eckel: *Thinking in Java*, 2nd edition, 2000, http://www.mindview.net/Books

[ETL3] Bertrand Meyer: *Eiffel: The Language*, 3rd edition, work in progress, not yet published as a book; updates can be read electronically at http://www.inf.ethz.ch/∼meyer/ongoing/.

[Gander] Walter Gander: *Computermathematik*, 1st edition, Birkhuser Verlag, Basel, 1985 (in German)

[Graham] Paul Graham: Article 'Beating the averages': http://www.paulgraham.com/avg.html

[Hoare] C.A.R Hoare: 'Proof of Correctness of Data Representations', *Acta Informatica*, Vol.1, pp. 271–281.

[ISOWG21] JTC1/SC22/WG21: *Notes on Standard Library Extensions*: http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2001/n1314.htm.

[Jänich] Klaus Jänich: *Analysis für Physiker und Ingenieure*, Springer, 2001 (in German).

[KandR] B.W. Kernighan and D.M. Ritchie: *The C Programming Language*, Prentice Hall, 1978.

[MOR] Wilhelm Magnus, Fritz Oberhettinger, Raj Pal Soni: *Formulas and Theorems for the Special Functions of Mathematical Physics*, Springer, 1966.

[NAGC] *NAG C library manual, Mark 5*, The Numerical Algorithms Group Limited, Oxford, UK, 1998.

[Nash] Stephen G. Nash (editor): *A History of Scientific Computing*, ACM Press, 1990.

[OOSC-2] Bertrand Meyer: *Object-Oriented Software Construction, 2nd Edition*, PTR Prentice Hall, 1997.

[OTSC] Paul F. Dubois: *Object Technology for Scientific Computing*, PTR Prentice Hall, 1996.

[Schwarz] Hans-Rudolf Schwarz: *Numerische Mathematik*, 4th, revised and enlarged edition, Teubner Verlag, Stuttgart, 1997 (in German); *Numerical Analysis. A comprehensive Introduction*, translation based on the 2nd German edition, John Wiley & Sons Ltd., 1989.

[Sebesta] Robert W. Sebesta: *Concepts of Programming Languages*, 5th edition, Addison-Wesley, 2002.

[Sutter] Herb Sutter: *The new C++*, article in *C/C++ Users Journal*, 2002: http://www.cuj.com/experts/2002/sutter.htm.

[Szyperski] Clemens Szyperski: *Component Software — Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

[Wirth] Niklaus Wirth: *Algorithmen und Datenstrukturen mit Modula-2*, 4^{th}, revised and enlarged edition, Teubner, Stuttgart, 1986 (in German; translations in many languages available).

[Wolfram] A lot of information on many mathematical and scientific topics can be fount at: http://mathworld.wolfram.com.

[Zuse] Konrad Zuse: *Der Computer — Mein Lebenswerk*, 3^{rd}, unmodified edition, Springer, 1993 (in German).