Advances in Design and Implementation of Optimization Software

István Maros and Mohammad Haroon Khaliq

Imperial College, London

Email: i.maros@ic.ac.uk, mhk@doc.ic.ac.uk

Departmental Technical Report 2000/11

ISSN 1469–4174

Abstract

Developing optimization software that is capable of solving large and complex real-life problems is a huge effort. It is based on a deep knowledge of four areas: theory of optimization algorithms, relevant results of computer science, principles of software engineering, and computer technology. The paper highlights the diverse requirements of optimization software and introduces the ingredients needed to fulfil them. After a review of the hardware/software environment it gives a survey of computationally successful techniques for continuous optimization. It also outlines the perspective offered by parallel computing, and stresses the importance of optimization modeling systems. The inclusion of many references is intended to both give due credit to results in the field of optimization software and help readers obtain more detailed information on issues of interest.

Keywords: Large scale optimization, optimization software, implementation technology.

1 Introduction

The importance of optimization is growing steadily. In addition to the well-established applications, such as in the petrochemical industry, transport or manufacturing, we are witnessing a rapid escalation of optimization into new areas like medical image processing or financial engineering. The newly emerging problems are increasingly large, complex and difficult to solve.

In the early days of optimization it became clear that the straightforward implementation (coding) of a theoretically favorable algorithm may not perform properly. This has triggered research into the intricacies of what makes an optimizer good that is capable of solving diverse real-life problems.

Developing serious optimization software is a huge effort. It requires deep knowledge in several areas (see section 3). At the beginning, such a work was done as part of academic life and the results were available to the scientific community (e.g. simplex solver XMP [60], nonlinear and simplex solver MINOS [65], and interior point solver OB1 [48]). However, soon it turned out that it is a big business to develop and market high quality optimization software. A couple of competing systems have appeared, developed by people with roots in academia. As a 'result' key algorithmic details have become mostly secret with little published. In many cases it was unknown to the public whether an improvement in the performance of a system was a result of a certain, but not closely specified, new algorithmic element, better coding or something else.

Currently, academic optimization software is catching up with, and temporarily may even supersede, the performance of commercial systems. However, this approach is just asymmetric since the academic results are readily available to commercial developers but not vice versa.

Still there are some important developments that have been published by researchers and they have contributed substantially to the understanding of what is required for good optimization software. Apparently, there is a growing interest in the mathematical programming community for the subject. This fact has motivated the preparation of this paper. The diverse nature of optimization makes it impossible to overview everything essential that has happened. Therefore, the paper concentrates on some important common features of optimization software.

The pioneering work in this area was done by W. Orchard-Hays in his book *Advanced Linear-Programming Computing Techniques* [66] in 1968. This was the first time when issues of building an optimization system were addressed explicitly. Since then the publication of details of implementation has become accepted in scientific literature. Also an important event was a NATO ASI School on *Design and Implementation of Optimization Software* in 1977. The proceedings edited by H.J. Greenberg [36] (published in 1978) is a rich source of papers reflecting the state-of-the-art of that time. Further developments came to light at the *International Workshop on Advances in Linear Optimization and Software*, Pisa, 1980. Since then there have been several meetings on similar topics.

As the theory and solution algorithms of optimization have evolved a number of publications have appeared discussing different questions of computational optimization. Scant few of them also addressed directly issues of implementation. Within the limitations of this paper it is impossible to provide a comprehensive bibliography of all papers that have some relevance to optimization software. A restricted list of papers dealing with implementation is [3, 5, 7, 9, 23, 25, 26, 29, 32, 35, 43, 49, 62, 63, 73].

To give a little idea of what the magnitudes of the development are, we recall that in 1969 the program library of the Wang desktop computers (forerunners of the current PCs) included the simplex method which was 30 lines of code (including input/output) written in Basic. It was a logically correct translation of the rules of the full tableau version of the simplex method. It could solve the supplied 5×6 test problem, but not much more. During testing it failed even on smaller (non-pathological) problems. Nowadays the simplex part of modern optimization systems consists of 10–40,000 lines of code in some high level language (Fortran or C/C++).

Why is there such an enormous difference? How much more do the new systems 'know'? How is that achieved? Here we attempt to outline some answers to these questions in the case of continuous optimization. Integer programming (IP) solvers use continuous algorithms as the main computational engine. Details of strategic techniques of

IP are beyond the scope of the paper.

The organization of the rest of the paper is as follows. In section 2 we provide a brief overview of some characteristics of optimization systems. Section 3 discusses the requirements of optimization software. In section 4 we give a survey of the most important hardware and software features that have relevance to optimization algorithms. This section discusses details that are not widely considered in the mathematical programming community though they have a strong impact on the quality of optimization software. Some computationally successful techniques are overviewed in section 5. The role of modeling systems is briefly discussed in section 6. In section 7 we outline the perspectives offered by parallel computing and, finally, section 8 presents some concluding remarks.

2 Characteristics of optimization systems

The ultimate goal of developing (new) optimization algorithms is to solve real-life problems. These problems can be deterministic or stochastic, linear or nonlinear, continuous or discrete. Though most of them require different solution algorithms there are some common elements in them that make it possible (and reasonable) to combine several methods into one system.

It is typical that the algorithms do linear algebra on vectors and matrices. These entities tend to be *sparse*, i.e. the ratio of the number of nonzero entries to the total possible entries is very small, usually $0.01\% - 1\%$. As such, for computational purposes the number of nonzeros (z) is an important third parameter in describing the size of a sparse problem in addition to the number of rows (m) and columns (n). The size of the matrix of a linear programming (LP) problem can, for instance, be described as 16,000 rows, 25,000 columns and 150,000 nonzeros.

It is the low density (or *sparsity*) that makes it possible to solve huge problems even on desktop personal computers (PCs). Utilizing sparsity is a major feature of optimization software. For example, the explicit inverse of a $10{,}000 \times 10{,}000$ basis in the simplex method with 50,000 nonzeros can be fully dense with 100,000,000 nonzeros requiring 800MB of main memory just to store it which is prohibitively large. With sparse storage and an appropriately chosen equivalent form of the inverse we can expect less than 150,000 nonzeros to represent the inverse which is easy to handle even on a low specification PC.

Only the nonzero coefficients of a problem need to be stored. There are different ways of storing sparse problems as discussed in section 5.1. During solution, transformations are performed on the vectors and matrices generating new nonzeros (*fill-in*). The fill-in can be so dramatic that the storage of the intermediate matrices (and operations with them) may become impossible. This has triggered research for finding algorithms or algorithmic equivalents that reduce the growth of nonzeros and have some other favorable features. Section 5.2 gives some more details of this effort.

An important feature of professional optimization systems is *algorithmic richness*. In the first place, it means that they are able to solve a variety of optimization problems, like linear, quadratic, mixed integer, or network programming. Additionally, for a given problem (e.g. LP) they have several solution algorithms incorporated (e.g. primal and dual simplex and an interior point method for LP), and even within a principal algorithm (e.g. primal simplex) there are several alternatives available for the main algorithmic steps (e.g. choosing the incoming variable—also known as finding the search direction). The aim of this richness is to enable the 'tuning' of the solution algorithm so that it can solve any actual problem in the best possible way. Criteria for the best possible way are discussed in section 3.

A high degree of comprehensiveness is a typical characteristic of professional optimization software. Academic optimization systems usually concentrate on fewer algorithmic options and are designed to enable deeper testing of new algorithmic ideas. An excellent collection of academic optimization software, which is freely available for academic users, can be found in [64].

3 Requirements of optimization software

As the development of optimization software is a major effort it is reasonable to expect that the codes possess some desirable properties. The most important ones can briefly be summarized as follows.

Robustness: A wide range of problems can be solved successfully. If the attempt is unsuccessful a meaningful answer is returned (e.g. unrecoverable numerical troubles). Robustness implies reliability and accuracy.

Efficiency: In simple terms, efficiency means execution speed. A fast algorithm implemented with code optimized by good design and profiling permits a solution to be obtained quickly.

Capacity: Large (size includes the number of nonzeros) and difficult (e.g. mixed integer) problems can be solved on a given computer configuration.

Portability: The system can be used on different hardware and software platforms. *Binary portability* can be achieved if the same or a similar hardware/software combination is used. *Source level portability* is easier to achieve and can be done by using some of the standard high level languages that are available on nearly all types of computers (independent of hardware and operating system).

Memory scalability: The system can adjust to the size of the main memory and operate reasonably efficiently if at least a minimum amount of memory is available. At the same time, it can adapt well if more memory is allocated. (The system is not critically memory-hungry.)

Modularity: The implementation is divided into data structures, functions, and translation units according to a design that is both intuitive and adheres to good software engineering practice.

Extensibility: The code permits both the simple addition of new features internally to the product and the straightforward integration of that product as a building block for other code.

Code stability: The behaviour of the optimization system code is well-defined even in an error situation. At the least, the system will not leak resources under an exception condition, and should attempt to continue running safely and meaningfully.

Maintainability: The impact of changes or additions to the code should be minimal. Source code is well annotated.

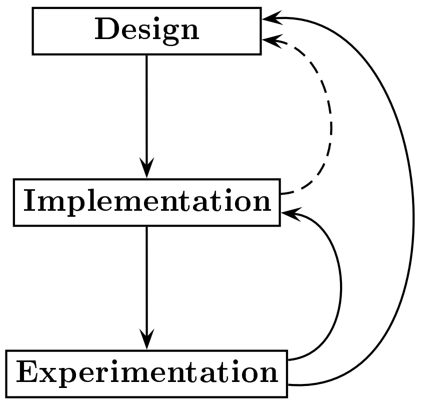
User friendliness: Sometimes also referred to as *ease of use*. This soft notion has several interpretations and has different meanings for a beginner and an advanced user. Under this heading the availability of a stand-alone and callable subroutine version is also understood.

To satisfy these requirements deep knowledge of four areas is imperative: theoretical background of optimization algorithms, relevant results of computer science, principles of software engineering, and the state-of-the-art in computer technology.

The aforementioned properties can be ensured partly in algorithm design and partly in an implementation phase as shown in the following table.

	Algorithm design	Implementation
Reliability	x	x
Accuracy	x	x
Efficiency	x	x
Capacity	x	x
Portability		x
Modularity		x
Code stability		x
Maintainability		x
Scalability	x	x
User friendliness	x	x

The phases of developing optimization software and their relationship can be depicted as follows:



The diagram shows that experimentation plays an important role in this process. On the other hand, nowadays it is commonplace that implementation is a rich source of new algorithmic ideas.

Evaluation of optimization software is an important issue for both developers and users. Over the years, libraries of standard test problems have been established that serve this purpose. These libraries are as follows: linear programming [28], quadratic programming (QP) [57], mixed integer programming (MIP) [10], and general nonlinear programming (NLP) [12]. These sets are freely available, therefore they are widely used in reporting achievements in algorithmic development. The rationale is one of the basic requirements of scientific publications: the claimed results must be reproducible. Unfortunately, this requirement is sometimes not fully respected and reports on proprietary problems get published.

4 The computing environment

Optimization problems cannot be solved without computers. In fact, the solution of real-life LP problems significantly inspired the development of computers in the '50s [20]. The crucial role of computers makes it necessary to survey some features of the hardware and software environment the solvers have to operate in. Here we discuss some details that are not widely known to the mathematical programming community, or their impact on optimization software is not fully recognized. It is important to remember that computing environments, especially the hardware, change very rapidly. What is covered in this section reflects the status quo at the time of writing the paper (August 2000).

4.1 Computer hardware

The proliferation of inexpensive, feature-rich computer hardware and its continuous evolution has had a significant impact not only on the use of optimization software, but also its further development. Processing power tends to double every 18 months (Moore's Law) as does data storage capability (Parkinson's Law of Data). Nevertheless, the price of components tends to stay low. The result is that economical and powerful workstations are readily available that can perform large-scale optimization—something that was the domain of expensive mainframe computers alone until the late '80s.

For optimization software the hardware features of the memory hierarchy, and the central processing unit (CPU) of a computer are most important. Knowledge of how to exploit these technologies now, and directions they will take in the future is crucial in the production of portable code that not only executes quickly today, but also scales well onto future machines.

Considering the memory hierarchy, in decreasing order of speed, decreasing order of cost per memory unit, and increasing capacity we may enumerate the components as follows: registers, L1 cache, L2 cache, RAM, and the hard disk ([15, page 266] and [82, pages 43–45]).

Each CPU has a small set of registers which are best used for rapidly changing and/or heavily accessed variables. Unfortunately, registers are not addressable by languages other than machine-specific assembler, but efficient code coupled with an optimizing compiler can give good register usage (e.g. allocation of loop variables) [15, page 268].

The L1 cache resides on a CPU, is marginally slower than a register, and can be either a unified instruction and data, or a separate instruction block and a data block ([4] and [82, page 12]). The slower L2 cache has recently come to exist in the CPU too, with a size of 256KB or more compared with 32–128KB for the L1 cache [4]. The caches keep data close to the CPU for rapid access, otherwise that data would be brought in time and again from slower main memory [69, page 545]. A frequently accessed variable held in a cache exhibits *temporal locality*. Each time the CPU wants a variable, the cache is queried and if the data is there we have a *cache hit*, with the item supplied to the CPU from the cache location. Since contiguous data tends to be accessed in programs, a cache will insert not only data currently needed, but also some items around that too to get more cache hits by guessing it will need what is nearby—this is *spatial locality* [83]. If data is not consistently accessed, is widely separated from other data, or is requested for the first time, we have the risk of a *cache miss*. For a direct-mapped cache, data is retrieved from memory, a location is isolated in the cache and before it is placed there any existing item is ejected from the cache. Fully associative and set-associative caches have the capacity to allow several items to be held at a cache location, but they are slower than the direct-mapped cache due to scanning operations that must be performed to try to get a cached item or signal a cache miss. Cache misses can delay a CPU for hundreds of cycles [38], but they are easily caused, such as through sparse matrix computations [83]. Research into software manipulation of caches is underway which will be of benefit to optimization software [38].

RAM and the hard disk drive have the slowest access times, but sizes typically in hundreds of megabytes, and many tens of gigabytes respectively. RAM works at the nanosecond level, yet currently CPUs are so fast that the RAM slows execution! Hard disk drives give persistent storage, but work at the millisecond level. Code that leads to thrashing in virtual memory mechanisms or needs considerable disk access clearly will have poor execution times ([8, pages 7–10] and [51, page 122]).

The features of the CPU that most affect optimization software are: the type of the processor, the number of *floating point units* (FPUs) and *arithmetic logic units* (ALUs), and the number and depth of pipelines.

A processor is typically described as a *Complex Instruction Set Computer* (CISC) or a *Reduced Instruction Set Computer* (RISC), with a given clock frequency and instruction bit width. For example, the Intel Pentium III family are all CISC CPUs, but the 64 bit SUN SPARC series are RISC processors. CISC CPUs have the advantage of low memory requirements for code, but although RISC machines have higher memory demands, they are faster due to hardware optimization of key operations [82, pages 91–95]. CISC-RISC hybrid processors are available that wrap a CISC shell over a RISC core, such as CPUs in the AMD Athlon series [4]. Recently, reconfigurable processors are in production, the Crusoe chip being an example, that have the property of being able to work as CISC or RISC depending on the code they are given at the expense of some CPU speed [45]. Many factors, such as the system bus speed, affect the speed of a machine. However, in general a higher clock frequency means a faster processor, as does a higher bit width of instructions.

Floating point (FP) calculations on a processor occur in the FPUs, built to implement the IEEE 754 standard usually [41, page 85]. Double precision (DP) numbers, typically 8 bytes in length (around 16 decimal digits accuracy), are processed here, but it should always be remembered that calculations—addition especially—are subject to errors that can accumulate ([70,71], and [86, pages 4–9]). More details on issues of numerical accuracy are discussed in section 5.3. Complementing FPUs are ALUs which undertake exact arithmetic on integers only, and sometimes they are employed for address calculations ([4] and [82, pages 348–350]). The more FPUs and ALUs a chip has, the greater the throughput of number processing that can be attained.

Most CPUs now have at least one pipeline made up of about 5–10 structural units, designed to perform an operation and pass the result to the next unit. Superscalar architectures have multiple pipelines. A pipeline can exist in many parts of a CPU, such as the ALU or the FPU. The depth, however, is fixed by factors such as the format of instructions and the amount of instruction unrolling to be done if the pipeline has to be stopped—as explained shortly [69, pages 438–440]. A pipeline increases CPU performance by allowing a sequence of instructions to be in the CPU per clock cycle, thereby boosting instruction throughput. A pipeline entity, however, is efficient only when the results of one operation are not pending on the results of another—this would cause a *stall*. *Branching*—jumps from boolean evaluation, function calls, and function returns—in a program is a major cause of these halts in the running of pipelines [69, page 442]. The hardware and software methods to contain this problem are discussed next.

A CPU with a pipeline can stall as one option in a problem situation, but other options are *branch prediction* or *dynamic branching*. Branch prediction makes an assumption, e.g.

a boolean test is always false, and bypasses a stall. If the guess was wrong, then operations have to be 'rewound' and the correct instructions used ([40, pages 379–381] and [69, page 496]). In dynamic branching, a small hardware lookup table is used holding the depth of branching and the number of times a branch held true or false. By using the lookup table a pipeline can be adjusted to follow the branch that occurs most often—this is useful in looping constructs since the wrong decision occurs only at the final iteration. The trouble with this method is that too many branches in code will overfill the table [69, pages 498–503]. The branching problem is one of several issues significant enough for the number of pipeline units to be kept low [69, page 521].

In software, the best method to utilize pipelines is to avoid branching, especially nested, as much as possible. Loop unrolling is a well-known technique to speed code by reducing boolean tests. This technique is implemented by optimizing compilers when the number of iterations is known at compile time. If the number of iterations is known only at run time, then code safety issues arise that necessitate employing loop unrolling with caution, e.g. in C++ if a loop is unrolled to a level of 4 but iterates through an array of length 3 at run time then an exception arises unless boolean tests are performed for each iteration. If simple branching can be avoided by a single operation in code, then it is preferable to replace the branch [15, pages 280–281]. For example in C, line `/* 2 */` below can replace the implied `if-else` structure of line `/* 1 */` in the following code excerpt (used in a presolve routine for MPS files when the objective row—of conditional code “N”—needs to be ignored by setting a flag to zero) :

```
/* characters are one of : 'E', 'G', 'L', or 'N' */


/* 1 */
theResultChar = (theTestChar == 'N') ? '\x0' : '\x1';


/* 2 */
theResultChar = (theTestChar / 'N') ^ '\x01';
```

Since branching can be caused by function call and return, reducing the number of functions in code is useful [8, page 11]. As an example, in C++, the keyword `inline` allows compilers to check the syntax of a short function and then embed its code in place of all function calls [15, pages 107–152]. This is one way to approach a trade-off in dealing with branching in code: a program should be modular and maintainable, but also must exhibit minimal branching. Sometimes, it is better to leave branching in place, such as when branches call different functions or alter different data types, since program transformations can leave the code unreadable. If possible, placing all of the disjoint code of a function that has had simple branching removed into one location can be of use, as streamlined execution occurs once the branch direction has been found by the CPU.

There is an interesting development in hardware technology under the heading of *reconfigurable computing*. This notion refers to hardware reconfiguration as opposed to a software one (see Crusoe processor above). Boolean satisfiability problems can directly be represented by reconfigurable computing units so that the evaluation of a Boolean expression is carried out in hardware at very high speed. For each problem instance the hardware is reconfigured (the equivalent of traditional 'problem input'). Early indications show that a speedup of up to 100 times can easily be achieved by this method. Investigations are underway on how to exploit this powerful new tool for better computational algorithms in integer programming.

The importance of the knowledge of hardware for optimization software was recognized by some researchers (c.f., [23, 73]) but little has been published in this area.

4.2 Computer software

Computer software is a tool for converting algorithms into computer programs. Nowadays optimization programs are written in some high level language, mostly Fortran or C/C++. These languages are highly standardized so their code can easily be compiled on different hardware/software platforms resulting in identically working programs. This is the basis of source level portability.

The efficiency of optimization code is affected by the following software factors: the operating system, the choice of programming language, and the compiler employed.

Operating systems act as the interface between user software and hardware, typically allowing multiple users access to resources in a way that gives the impression that each user has sole access. This latter feature—known as *multitasking*—is achieved today through intelligent scheduling of processes or threads to a processor. Operating systems differ markedly from each other, but some of the differences can be categorized thus: size of the code base which affects kernel size and the resources the kernel uses (e.g. 128MB RAM minimum for Windows 2000), support for threads or processes (e.g. Linux has process-based operation but the speed it attains rivals thread-based operating systems like Windows NT), the performance they guarantee as the number of users on a system increases, and the quality of the interface provided for programming highly optimized code (e.g. Visual C++ has access to many features of Windows-based system kernels).

The operating system best suited for an optimizer is difficult to isolate early. It is of great benefit therefore that coding be undertaken in a standardized language, like C, so that through portability the best operating system will be found by compiling and testing on different platforms.

The choice of programming language has an enormous impact on optimization software. The main contenders in this field are FORTRAN77, C, and recently C++ (e.g. [1, 75, 77]). A language decision must consider at least the following: standardization so that portability is allowed (the aforementioned have ANSI standards), whether the language is interpreted (like APL and MatLab) or compiled (like C) since the former is excellent for prototyping ideas but the latter is the means for fast code, whether there is control of static and dynamic memory (FORTRAN77 is capable only of the former; FORTRAN90 and later versions have dynamic memory support), whether code can be made modular for extensibility (e.g. the ability to create not only a stand-alone optimizer, but the same optimizer as a callable routine that can be integrated into a library), and the amount of safety the language provides against erroneous execution (e.g. the exception safety mechanism of C++).

To illustrate a decision for a programming language for an optimization system, we take the example of C++. The language is standardized to ANSI/ISO level since 1997 with good compiler support on practically all platforms [42]. It is a compiled language so we can expect well-written code to have good execution speed on different systems. Memory management is under the control of the programmer, so we can allocate and deallocate resources at run time, e.g. using functions `new` and `delete` or standard components such as vectors or queues [78, pages 128–129]. Object-oriented programming is allowed using classes so that modular code can be made that is intuitive to expand and maintain. Last of all, there is a well-defined exception mechanism in the language that allows software to recover from errors without program termination which could leak resources [80, pages 38–39].

The quality of a compiler influences the speed of well-written code on a given platform. A code may use generic speed optimizations—such as low function depth to reduce excessive stack usage [2, page 153], use pointers instead of arrays or vice versa, and exhibit minimal branching—but because the optimizing ability of each compiler is different, the run time performance is different [15, pages 5–6]. What must be noted, however, is that the quality of compiler-optimized code is competitive with or exceeds hand-optimized code, and the former is improving. For example, C code optimized by the GNU C compiler "gcc" is very difficult to better with optimization by hand. Languages like C++ allow the inclusion of assembler into code which the compiler then integrates. In this respect hand-optimized code may better compiler-optimized code. However, impressive gains in using assembler will need to be balanced against the loss of portability of a system [2, pages 28–29]. Lastly, compilers differ in the quality and breadth of the libraries they are packaged with. This can lead to speed differences in the same function that is part of a language standard, e.g. `malloc` in C [44, page 167].

To end this section, we should like to state our own experience of utilizing the computational environment. An 'environment aware' code of an optimization algorithm can be—without loss of portability—up to three times faster than an unsophisticated but otherwise carefully coded version of exactly the same algorithm.

5 Techniques

Writing programs that implement optimization algorithms can be a source of major surprises. It can happen that correctly coded, theoretically convergent algorithms among other things:

- run very slowly,
- require a huge amount of memory,
- do not converge,
- give the wrong answer (e.g. problem declared unbounded or infeasible though optimal)
- work only on small test problems (or worse, just on a specific test problem),
- sometimes do not work at all (e.g. loop infinitely or break down due to numerical difficulties)

Why is this? That the code does not follow the principles of implementation technology of optimization software is often the cause. As the ingredients of a good program are good algorithms and data structures, in this section we outline how this recipe can be interpreted for large scale sparse optimization.

5.1 Data structures

Data structures are responsible for the efficient storage and manipulation of sparse vectors and matrices occurring in large scale optimization. Their choice is vital for efficiency and scalability; the right choice depends on the intended usage. The two main types are *static* and *dynamic* data structures, and some examples include:

1. The LP matrix for the simplex method is static—usually a set of sparse row and(!)/or column vectors. If memory allows, it is worth maintaining both to help smooth transition between primal and dual operations and enable utilization of sparsity in updating (an important source of speedup).

2. The nonzero pattern of $\boldsymbol{A}\boldsymbol{A}^T$ in IPMs is created with dynamic structures.

3. Factorization of a sparse simplex (SSX) basis or $\boldsymbol{A}\boldsymbol{A}^T$ for IPM uses dynamic structures, and the result is stored statically.

For static storage, sparse vectors are best kept in *packed* form in which case the nonzero elements are stored in consecutive locations in a DP array and their indices are in an integer (or pointer) array. The length of the vector in terms of the number of nonzeros must also be stored.

Static matrices are stored as a set of packed row/column vectors in a linear array. If these vectors are located consecutively then only their starting positions have to be recorded (i.e. $n + 1$ positions). However, it is more practical to assume that logically neighboring vectors are not stored consecutively. In this case both the starting position and the length (or the position of the last element) of a vector must be recorded (i.e. $2n$ positions). This gives much needed flexibility because there are cases when even a static structure must undergo a one-off change (during presolve, adding constraints or variables, etc.).

Matrices in network optimization are very simple (containing no more than two nonzeros which are usually ± 1) and they can be stored in two linear arrays.

For storing sparse data that keeps changing structurally (e.g. intermediate data during factorization of a simplex basis) dynamic storage schemes are needed. The most efficient ones are *linked lists*—mostly doubly linked. They can be used to avoid, for instance, repeated search for elements with a certain property (e.g. linked list of rows with equal nonzero counts) or shifting arrays during insertion of new elements. More about data structures and operations on them can be found in [22, 35].

Use of linked lists comes with some overhead. Elements are accessed at one more level of indirection than in an array, so their use has to be well organized. If properly used, they are one of the most important factors in the efficiency of sparse optimization.

Sparse data structures have to be designed in such a way that they benefit from caching as much as possible. While with explicitly stored dense vectors or matrices this occurs automatically, it is far from obvious in the case of sparse data. Wrongly designed structures may suffer from heavy cache miss (which, on the other hand, is not easy to detect).

In the early days of computational optimization it was noticed [43] that even among the nonzeros there may be only very few distinct values, e.g. ± 1. This phenomenon is called *super sparsity.* Setting up a (relatively short) list of distinct nonzeros, a matrix entry can be retrieved by referencing it on the list. While in the '70s this method enabled the solution of considerably larger problems, nowadays, with more memory available, its advantages are offset by the loss of access speed due to nested indirect referencing.

5.2 Algorithms

The enormous progress in the capabilities of optimization software is the result of designing computationally more suitable algorithms and implementing them 'professionally' taking full advantage of the computing environment.

When a new algorithmic element (or a complete system) is developed its performance has to be evaluated. One of the most effective methods of monitoring performance is *profiling.* It enables us to measure how much of the solution time is spent in the different functional units of the program thus pointing to locations where improvements can be made. It is also useful in evaluating the effects of new algorithmic developments. Profiling is supported in nearly all software development systems. Therefore, it is widely used among developers of optimization software.

Presenting all the algorithmic achievements that have lead to the current state-of-the-art in optimization software would fill a book. Here we have to restrict the discussion to arbitrarily selecting some of the most important elements.

5.2.1 General framework of optimization

Most continuous optimization algorithms follow the pattern:

1. Input problem.
2. Do preprocessing: presolve ($\approx$ problem simplification), scaling.
3. Determine initial solution.
4. Check stopping criterion / optimality condition.

   If satisfied go to Step 7, otherwise find improving direction.
5. Find steplength. If infinite, terminate.
6. Move to new solution, perform update, return to Step 4.
7. Evaluate outcome, postprocessing, post optimality analysis.

In the case of a branch and bound or branch and cut algorithm for mixed integer programming the above is included in a hierarchy and the first two steps are usually different from the above.

5.2.2 Some algorithmic variants and their implementation

It is typical that certain algorithmic steps have several mathematically equivalent forms, and at times a step is not uniquely defined. In some cases it is known that one alternative is better than the other(s), like the elimination form of the inverse (EFI) in SSX is better than the simple product form (PFI). However, in most other cases the best choice is problem dependent, like the choice of the normal equation or the augmented system form in IPMs. The purpose of having this sort of selection is to be able to identify the most suitable variant to the problem and situation (more effective, faster, better accuracy, lower space overhead, etc.).

Due to the limitations of the paper, in this section we give just examples of some important algorithmic variants and their implementations.

5.2.2.1 Preprocessing. The purpose of preprocessing is to bring the problem into a computationally more suitable form. *Presolve* is used to eliminate redundant constraints, loosen/tighten existing ones without (significantly) increasing the density of the matrix involved. The criteria of presolve are different for LP SSX, IPM and MIP [6,13,31,74]. For instance, in LP the loosening of bounds, up to generating implied free variables, is advantageous. However, in MIP tightening is the main issue. By *scaling* the numerical properties of the problem can be improved so that the chances of numerical troubles are substantially reduced [7, 18].

In IPMs, obtaining sparse factorization of $\boldsymbol{A}\boldsymbol{A}^T$ is also considered under 'preprocessing'. The importance of this step cannot be underestimated. This is one of the most heavily researched area in IPMs (see [50] or [5] for an overview). The success of this step fundamentally determines the performance of the IPM code.

5.2.2.2 Starting procedures. They are used to find a 'good' initial solution with little computational effort. How is 'good' defined? It can mean several different things. It is usually said that a starting procedure is good if an initial solution is quick to obtain, close to optimality/feasibility, least degenerate, has few nonzeros, or enables quick iterations [9, 33, 59, 66]. In IPMs, the starting point must be well centered [5, 72]. In SSX a (near) triangular basis is advantageous (it can be found by variants of the 'crash' technique) or a block triangular basis (with matrix reordering and 'mini' LPs). Efficiency of starting procedures is by no means a negligible issue.

Example: A 'Lower Triangular Symbolic' crash with feasibility awareness [59] in the case of a problem with 16,676 constraints and 15,695 variables found all columns can be reordered in triangular fashion. Timing on a low specification PC was 29.45s with 'straightforward' code, but 0.33s with appropriate doubly linked lists—a speedup of nearly 100 times. For larger problems the rate of improvement is expected to grow further.

In IPMs, the computational cost of determining a starting solution is equivalent to the work of one IPM iteration.

5.2.2.3 Finding a search direction. It is the most time consuming operation in the majority of optimization algorithms. The choice heavily influences the number of iterations required to solve a problem. Generally, there is a trade-off between the quality of the direction and the effort of finding it. Appropriate data structures and carefully designed algorithmic details (e.g. coding to achieve a high level of cache hits) can contribute to the speedy operation of this functional unit.

In SSX, for instance, there is a huge variety of 'pricing' techniques to find an improving direction ('incoming variable'), such as (i) first improving candidate, (ii) Dantzig rule, (iii) (dynamic) partial pricing, (iv) sectional pricing, (v) normalized pricings (simple dynamic, Devex, steepest edge), (vi) multiple pricing, (vii) composite pricing, (ix) anti-degeneracy column selection. Some relevant references are [11,19,24,30,34,39,52–54,66, 81]. Many of these strategies have additional parameters and some of them can be used in combination (e.g. partial sectional partial pricing (PSPP) is very efficient in network simplex). In case of full single pricing (Dantzig, Devex, steepest edge) the reduced cost vector can explicitly be maintained. This speeds up pricing but puts an additional workload on updating. The trade-off depends on the way the updating is implemented and sometimes also on the nature of the problem.

While some of the above pricing techniques are known to be inferior to others they need not be discarded completely. Recent investigations [58] have shown that the application of several pricing techniques in combination can considerably improve the quality of column selection. In such cases even the 'weak methods' can contribute to the overall success. The important message of this observation is that a good combination of algorithmic elements can be more effective than any single one alone.

Determining a search direction is susceptible to numerical error and can wrongly indicate a direction profitable. It can lead to all sorts of troubles. As a precautionary measure (i) the corresponding linear algebra must be implemented in a numerically stable way, (ii) a reliable quality check is needed to catch an error as early as possible and corrective action must be included for the case of a failed check.

5.2.2.4 Finding the steplength. This algorithmic unit serves to determine the displacement of the solution along the search direction. It has to satisfy a number of criteria, like maintaining feasibility (if already feasible), making largest progress towards optimality/feasibility, staying close to the central path, and taking care of numerical stability. Some references are [5,19,32,34,52,55,66,87–89]. Here, various logical tests are performed on computed numerical values. Therefore, it is also a source of numerical troubles.

In SSX, this step is known as the *ratio test* for determining the pivot position. It requires the updated incoming column (primal) or the updated pivot row (dual). From a numerical and efficiency point of view it is important that the matrix is stored both column- and rowwise. If the updated row/column vector is not accurate enough the ratio test may have a qualitatively and/or quantitatively wrong outcome (e.g. division by small quantities that otherwise should be zero or quantities with the wrong sign).

An example of an algorithmic alternative that is theoretically intuitive and has very desirable computational features can be introduced at this point. In both phases of the dual and in phase-1 of the primal simplex it is possible to choose the pivot position by maximizing a piecewise linear (gain) function at each iteration that gives the largest possible progress in the direction of the actual (phase-1 or 2) objective [34,52,55,87]. The distinguishing features of this method are: (i) using proper data structures this method can be implemented very efficiently with hardly more work than the traditional pivot method, (ii) it has inherently better numerical stability because it creates a large flexibility in finding a pivot element (if the pivot at the maximum of the function is not 'good' enough, usually several other elements are still available along the break points of the gain function), (iii) it is very effective in coping with degeneracy as it can bypass degenerate vertices more easily than the traditional pivot procedures. This method is the basis of the great success of the dual SSX for MIP.

5.2.2.5 Update. At the end of each iteration not only the solution but also the intermediate data needed for the iterations has to be updated. This work can be as simple as adding a new factor to the PFI in the SSX method, but can also be quite involved like updating the explicitly kept reduced costs (requiring the computation of the updated pivot row) or using Forrest-Tomlin or Bartels-Golub update for maintaining triangularity of the EFI. These complex operations require carefully designed data structures for efficient performance. The speedup between straightforward and sophisticated, sparsity-exploiting implementations can be up to 50 times.

At this stage the periodical (or extraordinary) regeneration of some of the intermediate data also has to be done, like refactorization of the SSX basis, resetting Devex weights, recomputing the reduced costs, etc. The most critical of these is the refactorization (reinversion). It has to produce a sparse and accurate representation of the inverse of the basis and has to be very fast. While it was an area of intensive research in the '70s and '80s, nowadays it is widely accepted that computationally the most suitable form is the one proposed by Suhl and Suhl [79].

5.2.3 Control of optimization

The operation of modern optimization systems is controlled by a number of parameters. The purpose of the parameters is to adjust the algorithm to the problem to solve it accurately and efficiently. There are numeric and strategy parameters. Typical representatives of the first category are all sorts of tolerances and density handling parameters. In the second category we find decision variables describing which algorithm or algorithmic element to use and how (primal or dual SSX, normal equation or augmented system in IPM, strategy of presolve, scaling, ordering in IPM, pricing in SSX, number of candidates in different algorithmic steps, various MIP strategies, etc.).

Modern optimization systems have a large number (up to 100) of such parameters. Most of them are accessible to the users through a parameter file (*specs file*) so that they can be modified if needed, though this action requires knowledge and experience. Parameters have default values (worked out by the developers) that ensure robust operation of the solver for problems without extreme features.

5.2.4 Adaptivity

An algorithmic step can be made in several different ways as shown above. It is quite problem- and situation-dependent which alternative is the most suitable. Therefore, it is worth implementing several techniques and letting a supervisory algorithm choose the most appropriate one.

The choice can be a pre-selection of an algorithm, algorithmic element or numeric parameter based on a preliminary analysis of the problem at hand. This choice is then applied throughout the solution.

Examples:

1. IPM: normal equations or augmented system.
2. SSX: use both column- and rowwise storage of $\boldsymbol{A}$ to speed up column and row updates if there is enough memory available.
3. SSX: choose between primal and dual method.
4. SSX: choose pricing strategy (and its parameters).

The other possibility is to make selections 'on-the-fly' based on the continual evaluation of the entire process of optimization. As an example, we can think of the dynamic choice of pricing strategies as mentioned earlier and discussed in [58].

The supervisory algorithm can be equipped with some intelligence such that it can learn from the problem being solved. A knowledge base can also be set up to store experience from previous solutions and use it in future runs. MIP solvers are expected to be the main beneficiaries of this idea but other algorithms can also take advantage of it.

5.3 Numerical issues

Optimization algorithms use the standard IEEE 754 normalized double precision floating point arithmetic. While it ensures an accuracy of about 16 digits, in several cases it may not be sufficient. The reason is that computations suffer from *rounding error* and *cancellation error*, and errors in computed quantities can magnify and propagate. As a result the solution algorithms may run into numerical difficulties. In a 'fortunate' case the troubles lead just to reduced accuracy (few accurate digits in the, otherwise correct, solution). In the worst case, they make the algorithm give a wrong answer (wrong optimal basis, feasible problem declared infeasible, optimal problem found unbounded, etc.) or no answer at all (divergence, oscillation, etc.).

The main difficulty lies in the distinction between computed small numbers and computational garbage that otherwise should be zero. Dividing by such a quantity simply blows up the solution. Additionally, algorithms can make wrong branching decisions if some computed quantities do not have the correct sign. To alleviate the problem, a wide variety of tolerances are used to determine which computed value is considered zero. To avoid the scale dependency, relative tolerances are much better than absolute ones.

All of these precautions do not eliminate the problem completely. Therefore, optimization systems have to use such algorithmic components that have proven better numerical characteristics (e.g. elimination form of inverse is preferred to the straight product form in SSX or Cholesky factorization in IPM is preferred when the normal equations are not too dense) and must be prepared to handle situations when errors do occur.

The main source of inaccuracy lies in computing dot products, or more specifically, the 'additions' in them. It is well known that addition is not an associative operation in normalized floating point computing and it can cause serious numerical troubles. There are several methods to reduce the chance of catastrophic errors but none of them is a cure-all. There is a software technique that can accumulate dot products in arbitrary precision without loss of computational speed. This technique enabled the solution of intractable LP problems [56]. The disadvantage of this method is that it uses assembly level coding which reduces portability. Recently, a new processor has been introduced that includes dot product with arbitrary precision as the fifth arithmetic operation in hardware. It certainly will have an impact on optimization software [46].

6 Modeling systems and optimization

Creating, modifying, maintaining and analyzing large scale real-life optimization problems is a major task. This activity is receiving increasingly efficient support from tools called *modeling systems*. They have evolved from the matrix generators of the '70s and '80s. Modeling systems allow model formulation in a similar way as humans naturally do it (algebraic equations, sets, logic expressions, etc.). This enables better understanding of the model and also serves as its documentation. These are quite important features because nowadays optimization systems can solve much larger problems than humans can easily comprehend.

Modeling is done using the specific modeling language of a given system. Though the languages are different they share many common features.

Modeling systems also serve as a working environment. Without leaving them, different solvers can be activated and the solution can be fed back to the model for further analysis. They enable the maintenance of several versions of a model, even the creation of a larger model from 'sub-models'.

The few existing modeling systems are commercial products developed mostly by the involvement and based on the results of academic researchers. It is beyond the scope of this paper to review commercial modeling systems. Interested readers are referred to the following publications and web sites: [14, 27, 37, 61, 68, 85].

7 Parallel computing and optimization

Parallel computing has much to offer optimization software. This is an active area of research because there is the potential to: (i) solve problems faster, and (ii) solve larger problems [47]. A parallel computer has more than one CPU with the aim of achieving a computational task more efficiently than a *uniprocessor* system through dividing the work to the processors ([16, pages 13–24], [17, pages 4–23] and [84]). Parallel machines can be differentiated at the hardware and the software level—these will be the topics discussed in this section.

At the hardware level, there are two *parallel architectures* currently available: *shared memory* and *distributed memory* [67, pages 16–24]. The former has one system memory used by a number of identical processors (e.g. the Cray T90 has up to 32 CPUs). Such machines benefit from fast interprocessor communication, and the need for only a single copy of any data for execution. Their drawbacks lie not only in their cost, but also in the fact that due to a limited bandwidth they cannot be scaled up to very many processors [67, page 16]. Distributed memory parallel computers, however, can have thousands of processors (e.g. the Fujitsu AP3000 has up to 1024 CPUs) with each one holding its own system memory. These machines offer economy (e.g. heterogeneous CPUs can be connected), and scalability—to the extent that many machines can be connected despite wide geographic separation. However, this architecture tends to have slower interprocessor communication speeds. Thus, if a computational task cannot be split into relatively independent blocks, then communication costs will contribute substantially to run times. A further difficulty is that at least some and at worst all program and problem data must reside at each CPU for execution. In optimization software, these two problems, aggravated by the inherent unsuitability of sparse computing on distributed machines, have created an active area of research.

Parallel computers tend to be supplied with software tools specifically created to exploit the hardware best. The operating system can be UNIX, or a faster custom-made one for a given machine. Numerical computing is supported not only by compilers of C, C++, and FORTRAN, but also optimized libraries of code that abstract away the details of hardware interface from programmers [21]. In fact, the MPI communication protocol standard allows source-level portability to any parallel machine of C or FORTRAN code through the use of library functions [67, pages 363–397]. Current software tools are advanced enough to even allow code to configure how it views the hardware, e.g. a hypercube will be viewed as the parallel topology even though the hardware has a ring form [76, pages 319–352].

8 Conclusions

Current optimization systems, whether academic or commercial, are immensely more capable than their predecessors a few decades ago. This is the result of developments in computational algorithms for optimization, incorporation of advanced techniques and methods of computer science and software engineering, and last, but not least, the achievements of computer technology (speed, capacity, and architecture).

The implementation of optimization algorithms is a nontrivial activity, it is sometimes referred to as an art. However, this technology is based on the knowledge of a variety of good optimization algorithms that are implemented using the results of computer science and advanced hardware and software features of computers. As such, it is in the territory of science.

The primary purpose of the paper was to highlight the complex nature of designing and implementing optimization software and discuss briefly the sources of its successes. It was necessary to give a relatively detailed account of the computing environment in order to make the aims and nature of the efforts understandable. We pointed out that though optimization problems and solution algorithms are quite different they still have many common features that makes it reasonable to combine several of them into comprehensive systems. We also defined the criteria of good optimization software. New algorithms and algorithmic elements are designed with these criteria in mind.

A secondary purpose of the paper was to give sufficient guidance in the form of references to readers interested in more details of the subject.

Taking into account the growing need for optimization and the evolution of the computing environment, we can expect that the development of general and special purpose optimization software will continue with great momentum. New systems will be able to solve problems more reliably and quickly than their predecessors. Also, currently intractable problems will become solvable. This progress will certainly be assisted by new algorithmic discoveries and more advanced computers.

References

[1] O. Aberth and M. J. Schaefer. Precise Computation Using Range Arithmetic, Via C++. *ACM Transactions on Mathematical Software*, 18(4):481–491, December 1992.

[2] M. Abrash. *Michael Abrash's Graphics Programming Black Book*. The Coriolis Group Inc., Special edition, 1997.

[3] I. Adler, N. Karmarkar, M.G.C. Resende, and G. Veiga. Data Structures and Programming Techniques for the Implementation of Karmarkar's Algorithm. *ORSA Journal on Computing*, 1:84–106, 1989.

[4] AMD Athlon Processor Technical Brief, December 1999. Publication #22054 Rev:D.

[5] E.D. Andersen, J. Gondzio, Cs. Meszaros, and X. Xu. Implementation of Interior-Point Methods for Large Scale Linear Programs. In T. Terlaky, editor, *Interior Point Methods of Mathematical Programming*, volume 5 of *Applied Optimization*, chapter 6, pages 189–252. Kluwer Academic Publishers, 1996.

[6] E.D. Anderson and K.D. Anderson. Presolving in Linear Programming. *Mathematical Programming*, 71(2):221–245, 1995.

[7] M. Benichou, J.M. Gautier, G. Hentges, and G. Ribiere. The efficient solution of large-scale linear programming problems. *Mathematical Programming*, 13:280–322, 1977.

[8] A. Binstock and J. Rex. *Practical Algorithms for Programmers.* Addison-Wesley Publishing Company, 1995.

[9] R.E. Bixby. Implementing the Simplex Method: The Initial Basis. *ORSA Journal on Computing*, 4(3):267–284, Summer 1992.

[10] R.E. Bixby, S. Ceria, C.M. McZeal, and M.W.P. Savelsbergh. An Updated Mixed Integer Programming Library: MIPLIB 3.0. *Optima*, 54:12–15, 1998.

[11] R.G. Bland. New finite pivot rule for the simplex method. *Mathematics of Operations Research*, 2:103–107, 1977.

[12] I. Bongartz, A.R. Conn, N.I.M. Gould, and Ph.L. Toint. CUTE: constrained and unconstrained testing environment. *ACM Transactions on Mathematical Software*, 21(1):123–160, 1995.

[13] A.L. Brearley, G. Mitra, and H.P. Williams. Analysis of Mathematical Programming Problems Prior to Applying the Simplex Method. *Mathematical Programming*, 8:54–83, 1975.

[14] A. Brooke, D. Kendrick, A. Meeraus, and R. Raman. *GAMS.* GAMS Development Corp., `http://www.gams.com/`, 1998.

[15] D. Bulka and D. Mayhew. *Efficient C++ : Performance Programming Techniques.* Addison-Wesley Longman Inc., 2000.

[16] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures.* International Thompson Computer Press, 1995.

[17] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., 1999.

[18] A.R. Curtis and J.K. Reid. On the automatic scaling of matrices for gaussian elimination. *J. Inst. Maths. Applics*, 10:118–124, 1972.

[19] G.B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, 1963.

[20] G.B. Dantzig. Impact of linear programming on computer development. *OR/MS Today*, pages 12–17, August 1988.

[21] J. J. Dongarra and D. W. Walker. Software Libraries for Linear Algebra Computations On High Performance Computers. *SIAM Review*, 37(2):151–180, June 1995.

[22] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications. Clarendon Press, Oxford, 1986.

[23] J.J. Forrest and J.A. Tomlin. Vector Processing in Simplex and Interior Point Methods. *Annals of Operations Research*, 22:71–100., 1990.

[24] J.J.H. Forrest and D. Goldfarb. Steepest edge simplex algorithms for linear programming. *Mathematical Programming*, 57(3):341–374, 1992.

[25] J.J.H. Forrest and J.A. Tomlin. Implementing Interior Point Linear Programming Methods in the Optimization Subroutine Library. *IBM Systems Journal*, 31:26–38, 1992.

[26] J.J.H. Forrest and J.A. Tomlin. Implementing the Simplex Method in the Optimization Subroutine Library. *IBM Systems Journal*, 31:11–25, 1992.

[27] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 1993.

[28] D.M. Gay. Electronic mail distribution of linear programming test problems. *COAL Newsletter*, 13:10–12, 1985.

[29] P.E. Gill, W. Murray, M.A. Saunders, and M.H. Wright. A Practical Anti–Cycling Procedure for Linearly Constrained Optimization. *Mathematical Programming*, 45:437–474, 1989.

[30] D. Goldfarb and J. Reid. A Practicable Steepest-Edge Simplex Algorithm. *Mathematical Programming*, 12:361–371, 1977.

[31] J. Gondzio. Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method. *INFORMS Journal on Computing*, 9(1):73–91, 1997.

[32] J. Gondzio and T. Terlaky. A computational view of interior point methods. In *Advances in Linear and Integer Programming*, Oxford Lecture Series in Mathematics and its Applications, chapter 3, pages 103–144. Clarendon Press, 1996.

[33] N.I.M. Gould and J.K. Reid. New crash procedures for large systems of linear constraints. *Mathematical Programming*, 45:475–501, 1989.

[34] H.J. Greenberg. Pivot selection tactics. In *Design and Implementation of Optimization Software* [36], pages 143–174.

[35] H.J. Greenberg. A tutorial on matricial packing. In *Design and Implementation of Optimization Software* [36], pages 109–142.

[36] H.J. Greenberg, editor. *Design and Implementation of Optimization Software*. Sijthoff and Nordhoff, 1978.

[37] H.J. Greenberg. *A Computer-Assisted Analysis System for Mathematical Programming Models and Solutions: A User's Guide for ANALYZE*. Kluwer Academic Publishers, Boston, MA, 1993.

[38] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. *Computer Architecture News*, 28(2):107–116, May 2000.

[39] P.M.J. Harris. Pivot Selection Method of the Devex LP Code. *Mathematical Programming*, 5:1–28, 1973.

[40] J. P. Hayes. *Computer Architecture and Organization*. The McGraw-Hill Companies, Inc., Third edition, 1998.

[41] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., Second edition, 1996.

[42] International Standard for Information Systems—Programming Language C++. ISO/IEC JTC1/SC22/WG21, November 1997. Morristown, N.J.

[43] J.E. Kalan. Aspects of Large-Scale In-Core Linear Programming. In *Proceedings of the 1971 annual conference of the ACM*, pages 304–313. ACM, 1971.

[44] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall PTR, 2nd edition, 1988.

[45] A. Klaiber. The Technology Behind Crusoe Processors, January 2000. Transmeta Corporation.

[46] U. Kulisch. The Fifth Floating-point Operation for Top-Performance Computers. Technical report, Institut fur Angewandte Mathematik, Universität Karlsruhe, 1997.

[47] I. J. Lustig and E. Rothberg. Gigaflops in Linear Programming. *Operations Research Letters*, 18(4):157–165, 1996.

[48] I.J. Lustig, R.E. Marsten, and D.F. Shanno. Computational Experience with a Primal–Dual Interior Point Method for Linear Programming. *Linear Algebra and its Applications*, 152:191–222, 1991.

[49] I.J. Lustig, R.E. Marsten, and D.F. Shanno. On Implementing Mehrotra's Predictor-Corrector Interior Point Method for Linear Programming. *SIAM Journal on Optimization*, 2:435–449, 1992.

[50] I.J. Lustig, R.E. Marsten, and D.F. Shanno. Interior point methods for linear programming: Computational state of the art. *ORSA Journal on Computing*, 6(1):1–14, 1994.

[51] M. J. Karels M. K. McCusick, K. Bostic and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Inc., 1996.

[52] I. Maros. A general Phase–I method in linear programming. *European Journal of Operational Research*, 23:64–77, 1986.

[53] I. Maros. A multicriteria decision problem within the simplex method. In Gautam Mitra, editor, *Mathematical Models for Decision Support*, pages 263–272. Springer Verlag, 1988.

[54] I. Maros. A structure exploiting pricing procedure for network linear programming. Research Report 18–91, RUTCOR, Rutgers University, NJ, USA, May 1991. 15 pages.

[55] I. Maros. A Piecewise Linear Dual Procedure in Mixed Integer Programming. In R. Schaible F. Giannesi and S. Komlosi, editors, *New Trends in Mathematical Programming*, pages 159–170. Kluwer Academic Publishers, 1998.

[56] I. Maros and Cs. Mészáros. A numerically exact implementation of the simplex method. *Annals of Operations Research*, 58:3–17, 1995.

[57] I. Maros and Cs. Mészáros. A repository of convex quadratic programming problems. *Optimization Methods and Software*, 11&12:671–681, December 1999.

[58] I. Maros and G. Mitra. Investigating the Sparse Simplex Algorithm on a Distributed Memory Multiprocessor. *Parallel Computing*, 26:151–170, 2000.

[59] I. Maros and G. Mitra. Strategies for creating advanced bases for large-scale linear programming problems. *INFORMS Journal on Computing*, 10(2):248–260, Spring 1998.

[60] R.E. Marsten. XMP: A Structured Library of Subroutins for Experimental Mathematical Programming. *ACM Transactions on Mathematical Software*, 7:487–497, 1981.

[61] Maximal Software, Inc., `http://www.maximal-usa.com/mpl/`. *MPL Modeling System.*

[62] K.A. McShane, C.L. Monma, and D.F. Shanno. An Implementation of a Prima-Dual Interior Point Method for Linear Programming. *ORSA Journal on Computing*, 1:70–89, 1989.

[63] S. Mehrotra. On the Implementation of a Primal-Dual Interior Point Method. *SIAM Journal on Optimization*, 2:575–601, 1992.

[64] H.D. Mittelmann and P. Spellucci. Decision Tree for Optimization Software. `http://plato.la.asu.edu/guide.html`.

[65] B.A. Murtagh and M.A. Saunders. MINOS 5.1 User's Guide. Technical Report SOL 83–20R, Stanford University, 1987.

[66] W. Orchard-Hays. *Advanced Linear-Programming Computing Techniques.* McGraw-Hill, 1968.

[67] P. S. Pacheco. *Parallel Programming With MPI.* Morgan Kaufmann Publishers Inc., 1997.

[68] Paragon Decision Technology, `http://www.aimms.com/index.html`. *AIMMS.*

[69] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design : The Hardware-Software Interface.* Morgan Kaufmann Publishers Inc., 2nd edition, 1998.

[70] F. Ris, C. Barkmeyer, and P. Farkas. When Floating-Point Addition Isn't Commutative. *SIGNUM Newsletter*, 28(1):8–13, January 1993.

[71] T. G. Robertazzi and S. C. Schwartz. Best Ordering for Floating-Point Addition. *ACM Transactions on Mathematical Software*, 14(1):101–110, March 1988.

[72] C. Roos, T. Terlaky, and J.-Ph. Vial. *Theory and Algorithms for Linear Optimization.* Discrete Mathematics and Optimization. Wiley, 1997.

[73] E. Rothberg and A. Gupta. Efficient Sparse Matrix Factorization on High Performance Workstations—Exploiting the Memory Hierarchy. *ACM Transactions on Mathematical Software*, 17(3):313–334, 1991.

[74] M.W.P. Savelsbergh. Preprocessing and Probing for Mixed Integer Programming Problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[75] B. F. Smith. The Transition of Numerical Software : From Nuts-and-Bolts to Abstraction. *SIGNUM Newsletter*, 33(1):7–15, January 1998.

[76] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference, Volume 1, The MPI Core*. Scientific and Engineering Computation Series. The MIT Press, 2nd edition, 1999.

[77] R. S. Solanki and J. K. Gorti. Implementation of an Integer Optimization Platform Using Object-Oriented Programming. *Computers and Operations Research*, 24(6):549–557, 1997.

[78] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 3rd edition, 1997.

[79] U.H. Suhl and L.M. Suhl. Computing Sparse LU Factorizations for Large-Scale Linear Programming Bases. *ORSA Journal on Computing*, 2(4):325–335, Fall 1990.

[80] H. Sutter. *Exceptional C++*. The C++ In-Depth Series. Addison-Wesley Longman Inc., 2000.

[81] A. Swietanowski. A New Steepest Edge Approximation for the Simplex Method for Linear Programming. *Computational Optimization and Applications*, 10(3):271–281, 1998.

[82] D. Tabak. *Advanced Microprocessors*. McGraw-Hill, Inc., Second edition, 1995.

[83] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal Of Research And Development*, 41(6):711–725, November 1997.

[84] R. R. Trippi and E. Turban. Parallel Processing and OR/MS. *Computers and Operations Research*, 18(2):199–210, 1991.

[85] P. Van Henteryck. *The OPL Optimization Programming Language.* MIT Press, 1999.

[86] P. J. L. Wallis. *Improving Floating-Point Programming.* John Wiley & Sons Ltd., 1990.

[87] Ph. Wolfe. The composite simplex algorithm. *SIAM Review*, 7(1):42–54, 1965.

[88] S.J. Wright. *Primal-Dual Interior Point Methods.* SIAM, 1996.

[89] H. Yamashita and H. Yabe. Superlinear and quadratic convergence of some primal-dual interior point methods for constrained optimization. *Mathematical Programming*, 75:377–397, 1996.