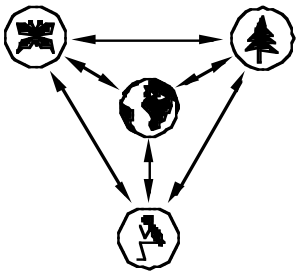

SYSTEMÖKOLOGIE ETHZ

SYSTEMS ECOLOGY ETHZ

Bericht / Report Nr. 23

How to Write Fast Programs

T.J. Löffler

January 1996

Eidgenössische Technische Hochschule Zürich ETHZ
Swiss Federal Institute of Technology Zurich
Departement für Umweltnaturwissenschaften / Department of Environmental Sciences
Institut für Terrestrische Ökologie / Institute of Terrestrial Ecology

The System Ecology Reports consist of preprints and technical reports. Preprints are articles, which have been submitted to scientific journals and are hereby made available to interested readers before actual publication. The technical reports allow for an exhaustive documentation of important research and development results.

Die Berichte der Systemökologie sind entweder Vorabdrucke oder technische Berichte. Die Vorabdrucke sind Artikel, welche bei einer wissenschaftlichen Zeitschrift zur Publikation eingereicht worden sind; zu einem möglichst frühen Zeitpunkt sollen damit diese Arbeiten interessierten LeserInnen besser zugänglich gemacht werden. Die technischen Berichte dokumentieren erschöpfend Forschungs- und Entwicklungsresultate von allgemeinem Interesse.

Adresse des Autors / Addresse of the author:

T.J. Löffler
Systems Ecology
Institute of Terrestrial Ecology
Department of Environmental Sciences
Swiss Federal Institute of Technology ETHZ
Grabenstrasse 3
CH-8952 Schlieren/Zürich
Switzerland

e-mail: sysecol@ito.umnw.ethz.ch

How to Write Fast Programs

A Practical Guide for RAMSES Users

T.J. Löffler†

CONTENTS

INTRODUCTION 2
1. COMPUTER COMPONENTS 2
2. ELEMENTARY RESULTS OF BENCHMARKS 3
3. CONSTANTS 5
4. INVARIANT VALUES 5
4.1 Quasi-constants, Initialization 6
4.2 Variables inside Blocks and Loops 6
4.3 Splitting complex algorithms 7
4.4 Array Copying 7
5. ROUTINES AND FUNCTIONS 8
5.1 Pre-processors and Inline functions 8
5.2 Parameter lists 9
6. MULTI-DIMENSIONAL ARRAYS 10
6.1 Internal index computation, addressing, and accessing 10
6.2 Page faulting 12
6.3 Memory stride 12
7. MATHEMATICAL ALGORITHMS 13
7.1 Sophisticated algorithms 13
7.2 Standard Libraries 13
7.3 Built-in Functions 14
7.4 Arithmetic Programming 15
7.4.1 Simple Arithmetic Techniques 15
7.4.1.1 Unnecessary Operations 15
7.4.1.2 Multiplication and Division 15
7.4.1.3 Mathematical functions 16
7.4.2 Type conversions 16
8. PROGRAM STRUCTURING 17
8.1 Statements for decisions 17
8.1.1 If-Then-Else, Case statements, and loops 17
8.1.2 If-Then-Else and Case statements 17
8.1.3 If-Then-Else statements 17
8.2 Loop statements 18
8.3 Procedures and Functions 18
8.4 Lists and Trees 18
8.6 Recursion 18
9. COMPILER OPTIONS 19
10. BATCH RUNS 19
CONCLUSION 19
REFERENCES 20

† Systems Ecology, Institute of Terrestrial Ecology, Department of Environmental Sciences, Swiss Federal Institute of Technology ETHZ, Grabenstr. 3, CH-8952 Schlieren / Zürich, SWITZERLAND.

Introduction

In this paper aspects for writing fast programs will be discussed and extracted into rules. Each rule will be explained by an example mainly given in Modula-2 if the rule is independent of a special programming language. Some specialities of specific languages will be considered but for special properties we refer to the relevant literature.

The given rules are concepted for saving computation time which may conflict with other aspects of writing programs e.g. producing small, modular, self-explainable code etc.

The efficiency of these rules within a program depends strongly on the code itself as well as on the special problem which has to be solved. The programmer also has to decide about which rules to implement based on his particular programming style. It is not our aim to interfere with such discussions here.

The main rule of writing time optimised programs can be described as ***save operations whenever possible***. This rule will be described more explicitly using basic examples given in the text.

Note that the way of saving operations is not unique. It can be done on several levels as in objects, modules, or routines and functions by local or global variables etc. and in different kinds of implementation.

1. Computer components

In this chapter, significant components of platforms (the composition of hardware and software) which influence the run-time behaviour of programs are briefly described.

The run time behaviour of programs depends not only on the program itself. It is also essential to know details of the elementary hardware components, the system, and used emulation software for a computer as follows.

The CPU is necessary for any kinds of operation, e.g. integer arithmetic. Another possibly available unit is the FPU (numerical co-processor) which is designed for floating point operations. Today both units are often included together within one chip.

If no FPU is available, this function is realized via emulation by the CPU. This leads to

- a highly inefficient run-time behaviour of applications due to the emulation of floating point operations by integer arithmetic;
- the possible use of so called FPU software for effective emulation of the absent numerical co-processor;

Beside the distinction of CPU and FPU we have to differentiate between CISC (**C**omplete **I**nstruction **S**et **C**omputer) and RISC (**R**educed **I**nstruction **S**et **C**omputer) processors. Examples for CISC architecture are Intel's 80x86 CPU, 80x87 co-processor series, Pentium and Motorola's MC68040 for Macintosh computers and for RISC architecture the PPC601 used by Macintosh's PowerPC and the SuperSparc chips of SUN workstations.

The conception of RISC is to replace rarely used processor instructions of their predecessors by a series of other faster processor instructions. By this strategy, the resulting efficiency of the RISC architecture increases and is supposed to be faster than the CISC architecture. However, CISC machines such as Intel's Pentium are real competitors even to the SuperSparc philosophy of SUN [Löffler & Fischlin, 1995].

In order to generate the fastest possible code a compiler must be able to generate so called *native code*. This means that the compiler should be able to create applications which could potentially use all of the available instructions supported by the processor(s) of the machine what can be the guarantor for fastest applications related to that processor(s). For example, an application generated for Intel's 8086 chip will not run as fast on Pentium PC's.

Producing native code is also essential for the FPU, e.g. the Modula compiler MacMETH offers the option *Compile20* [Wirth *et al.*, 1992] for generating such code for the co-processor MC68020.

For Apple computers [Apple Computer, 1986; Wirth *et al.*, 1992] there is the **S**tandard **A**pple **N**umerical **E**nvironment (SANE). Prior to the compilation SANE can be switched on or off. This environment provides high accuracy for arithmetical operations (64 bit arithmetic) but applications compiled without SANE are significantly faster [Löffler & Fischlin, 1995] The reason for this is that the transformations from 32 to 64 bit representation and vice versa done by SANE for floating point variables are implemented within *every* arithmetical operation.

```
VAR
       dValue1, dValue2, dValue3, dValue4 : REAL;
       ...
       dValue1       = dValue1 * (dValue2 - dValue3) / (dValue4 + dValue2);
```

In this example the steps by using SANE are transforming

- *dValue4* and *dValue2* to 64 bit representation, then adding them together and converting the result back to 32 bit representation;
- *dValue2* and *dValue3* to 64 bit representation, then subtracting them and converting the result back to 32 bit representation;
- the results from the described 2 operations to 64 bit representation, then dividing them and converting the result back to 32 bit representation;
- the result of this last operation and *dValue1* to 64 bit representation, then multiplying them together and converting the result back to 32 bit representation;

All of these transformations slow down arithmetical operations significantly.

The efficiency with which data can be transferred to and from memory is an essential limiting factor for the performance of a computer. This is of particular importance if operations are repeated on a large number of data elements as in linear algebra.

For efficient memory management modern computers have *cache memory* which is an intermediate store between the CPU and the main memory designed to hold contiguous data values (e.g. arrays) for immediate processing by the arithmetic unit (cf. chapter 6.2).

There are two kinds of multitasking, systems, the cooperative (e.g. Windows and MacOS) and the preemptive (e.g. Unix). In a cooperative multitasking system a running application has control of the computer and has to return this control to the system actively for other applications to work. In preemptive multitasking systems a master process controls the machine completely and gives the active applications time slices for running.

2. Elementary results of benchmarks

Aside from the optimisation of programs described in the following chapters, we give additional information of the run-time behaviour of applications in a short summary based on elementary benchmark experiments [Löffler & Fischlin, 1995] . For more details refer to this publication.

The tested machines were Macintosh computers (SE/30, IIfx, Quadra700, Quadra950, PowerPC, PowerBook170, PowerBook520, and PowerBook540c), IBM workstations (80486 and Pentium), and SUN workstations (SPARCstation 10, SPARCserver 630 MP, and SPARCstation ipx).

The compilers used were for Macintosh machines *MacMETH V3.2.2*, *Symantec C/C++ V7.0.4*, and *MPW C V3.3*, for SUN workstations *em2 V2.0.2*, and *gcc/g++ V2.4*, and for IBM workstations *Borland Pascal V7.0*. Important results in the context of this paper are

- Not using SANE for Macintosh computers increases speed by approximately 50%. More precisely the range of the increase for *mathematical operations* lies between 1% to 180%, dependent on the particular mathematical operation.

- Using Compile20 brings increases the mean speed of Macintosh computers by approximately 60%. More precisely the range of the increase for *mathematical operations* lies between -12% to 1700%, dependent on the particular mathematical operation.

- Using the optimised library DMMathLib20 [Fischlin *et al.*, 1988; Fischlin *et al.*, 1994] combined with the Compile20 option of *MacMETH* brings in a large *mean* increase in speed of around 500%. More precisely the range of the increase for *mathematical operations* lies between 0% to 4000%, dependent on the particular mathematical operation.

- Using no extensions for Macintosh computers brings a mean increase of approximately 400%. This increase arises mainly from the mathematical subtests.

- Being unconnected to a network increases the means speed of Macintosh computers by approximately 30%. This increase arises mainly from the mathematical subtests.

- The benchmark for the case of using no extensions and being unconnected to a network shows *no* measurable effect for Macintosh computers.

- For the PowerPC, the FPU emulation shareware SoftwareFPU V3.02 and V3.03 allows to running of applications which need a FPU but brings no advantage in speed. The shareware PowerFPU V1.01 increases the mean speed of applications by an additional 100%. That means PowerFPU V1.01 doubles the speed of a PowerPC for non-native applications which need a FPU.

- Applications for PowerPC not generated as native code cannot compete with native code applications for SUN workstations or IBM workstations with Pentium processor.

- CISC machines (IBM workstations) can compete with RISC workstations (SUN SPARCstation).

- Different but equivalent language elements (e.g. loop constructions etc.) of the tested programming languages can have different run-time behaviour. Therefore it must be borne in mind that similar constructions in a programming language can have a different run-time behaviour. For instance, different loop constructions in Modula differ in their speed in a range of 20% to 50% contrary to loop constructions in C/C++.

- Built-in functions (e.g. the functions *Inc* and *Dec* in Modula) are about 30% faster then normal constructions (e.g. *i:=i+1* and *i:=i-1* in Modula).

- By increasing the parameter list, calls of routines significantly slow down an application. For example, the difference in speed of calling a routine without a parameter list and for such a list transferring 4 integers is about 25%.

- Checking ranges of arrays, pointers etc. slow down an algorithm significantly (about 40% to 70% speed depending on the operations).

- The access to one-dimensional arrays versus matrices can be *significantly* faster (about 140%). Also, the access to one-dimensional arrays versus lists (via pointer) is *significantly* faster (about 1210%)

- The difference of computing with real or integer variables is about 2270% for simple operations as multiplication, division, addition and substraction.

- The C/C++ compilers generate code which runs by a factor 1.4 to 4 faster than the code of the Modula compilers.

- Different profiler sessions show that users normally write their applications not with respect to run-time even if they need fast programs. An experiment with an deterministic forest model easily achieved a 70% increase in speed.

The run-time behaviour of applications depends strongly on the compiler used for the programming language. Thus several rules presented in the following sections are not universal, they also depend on the quality of the compiler. For these cases refer to publications about benchmarks.

3. Constants

One of the easiest ways to save computation time is to use constants for expressions like $\sqrt{\Pi / 2}$ etc. and not routines like Pi() for computing such values.

Rule 3(1) *Determine all values used in the implemented algorithms which can be computed beforehand and store them as constants.*

Rule 3(2) *Never use routines which compute constant values (e.g.* ***Pi ()****) more then once in a program.*

For example use the routine Pi() for declaration.

```
CONST
       sqrtPi = Sqrt (Pi ());
```

Alternatively such routines can be used for the initialisation of variables which are used like constants as it is described in the next chapter.

Another example is storing an array of constant values like the faculties 1!,...,5!.

```
const int
       aiFaculty     = (1,2,6,24,120);
```

4. Invariant values

Further way to save on the number of operations is given by the invariance of values for the complete program, objects, modules, blocks, loops etc. Such values can be computed at the beginning of the program, block or loop etc. evaluation. This is possible whenever values depend only on data which are not changed by the specific algorithm in use.

The equation $t_{save} = n_{calls}(t_{compute} - t_{get}) - t_{ini}$ describes the advantage of using initialised values and it has 2 statements.

- Run-time can only be saved ($t_{save} > 0$) if the time t_{ini} for the initialisation of the value is smaller than the difference $t_{compute} - t_{get}$ of the frequencies $t_{compute}$ which we would have to investigate if we use no initialisation structures and t_{get} needed to get the value from the variable.

- If run-time is saved by the program structure than the saved time t_{save} increases linearly with how often the values is used, n_{calls}.

Note that $n_{calls} * t_{compute}$ is not necessarily identical to t_{ini}.

4.1 Quasi-constants, Initialization

By quasi-constants we mean variables which can be determined by an algorithm in the initialisation phase of the program run if these values are independent of the effects of the simulation. Note, that such variables can be dependent on initial values of a run.

Another possibility is the replacement of constant declarations which are, by definition, totally independent of effects as well as initial values of the program run by quasi-constants.

This may be useful if the algorithm for determining the constants is too complicated to be solved by hand.

Rule 4(1) *Determine all possible values in the initialisation phase of the program run (dependent on given initial conditions).*

As an example the array **adInitialArray** depends on the variable **rStartValue** and the contents of the former will not change during the program run.

```
IMPLEMENTATION MODULE InitialArray;
CONST
       dInitialArrayDimC    = 50;
VAR
       adInitialArray       : ARRAY[1..dInitialArrayDimC] OF REAL;

PROCEDURE GetInitialArray (iDim: INTEGER; rStartValue: REAL);
VAR
       I      : INTEGER;
BEGIN
       FOR I:=1 TO iDim DO
              BEGIN
                     adInitialArray[I]    := rStartValue / FLOAT (I);
              END;
END GetInitialArray;

BEGIN
       GetInitialArray (100, 12.456);
END InitialArray.
```

4.2 Variables inside Blocks and Loops

Similar to section 4.1 it is often possible to define variables which are constant within a block, or loop etc.

Rule 4(2) *For blocks and loops determine all possible values which are independent of the algorithm outside the block/loop.*

```
PROCEDURE Loop(iArrayDim: INTEGER; VAR adTheArray: ArrayType; dIniValue: REAL);
VAR
       I, iSaveIndex, iEndIndex    : INTEGER;
       dExpIni                     : REAL;
BEGIN
       dExpIni       := Exp (dIniValue);
       iEndIndex     := iArrayDim - 1;
       FOR I:=0 TO iEndIndex DO
              BEGIN
                     iSaveIndex    := I + 1;
                     adTheArray[I] :=
                            FLOAT (iSaveIndex) + dExpIni - Sqrt (adTheArray[iSaveIndex]);
              END;
       ...
END Loop;
```

This example includes three points. First, there will be saved run-time by not computing **Exp (dIniValue)** within the loop, second, by computing **iSaveIndex** one saves an additional operation per loop, and third, by introducing **iEndIndex** another computation of **iArrayDim-1** after every iteration can be saved for testing whether or not the loop has ended. This leads to the rule

Rule 4(3) *Within loops avoid all computations of indices.*

Remark Sophisticated compilers provide optimisation options for doing rule 4(3) automatically, especially saving the index computation **iArrayDim-1**.

4.3 Splitting complex algorithms

If we can split a complex algorithm into pieces for using them for different operations within the algorithm then it is possible to evaluate this algorithm faster.

Rule 4.3 *Generally split algorithms into pieces which can be used several times (to save evaluations).*

```
       ...
       Diam                 := Diameter (trueHeight, minHBirth, spec, treeSpec);
       LnDiam100            := Ln (Diam * 100);
       LnDiam100a2          := LnDiam100 * a2;
       a1ExpLnDiam100a2     := a1 * Exp (LnDiam100a2);
       leaveAreaInH[height] := c2 * a1ExpLnDiam100a2;
       folW                 := c1 * a1ExpLnDiam100a2;
       stemW                := 0.12 *Exp (LnDiam100 * 2.4);
       biomassInH[height]   := (folW + stemW) * 10.0 / patchSize;
       ...
```

In this example a part of a complex algorithm is split such that the pieces **LnDiam100** and **a1LnDiam100a2** can be used twice.
In the case of complicated algorithms writing efficient programs by splitting the algorithm can also make the code more understandable because it is easier to understand simple pieces rather then the whole algorithm.
Some compilers avoid programming code which is grouped together, e.g.

```
       ...
       dValue := (dValue1 * dValue2) * dValue3;
       dValue := dValue4 * (dValue1 * dValue2);
       ...
```

In such cases the compiler will not evaluate **dValue1*dValue2** twice in the final executable binary.

4.4 Array Copying

Array copying can be completely avoidable which will be demonstrated by the following example.

```
VAR
       dArray               : ARRAY[1..10000][1..17][1..2] OF REAL;
       dGKS                 : ARRAY[1..17] OF REAL;
       dYP, dY              : ARRAY[1..10000] OF REAL;
       iI, iJ, iOld, iNew   : INTEGER;
       ...
iIndexOld     := 1;
iIndexNew     := 2;
       ...
(******** begin of iterated block ********)
       ...
       FOR iI=1 TO 10000 DO
              dArray[iI,1,iNew] = Y[I];
              FOR iJ=1 TO 16 DO
                     YP[iI]              = YP[iI] + (Y[iI] -
                                                 dArray[iI,iJ,iOld]) * GKS[iJ];
                     dArray[iI,iJ+1,iNew] = YP[iI];
              END
       END
       ...
iOld   := 3 - iNew;
iNew   := 3 - iOld;
       ...
(******** end of iterated block ********)
       ...
```

This example shows a block in which the computed values from the last iteration are necessary for generating the new values. Instead of using 2 arrays and making copies of the resulting array in each iteration, we simply combine both arrays together and change the values of those indices to access the old values and store the new ones (needed in the next iteration) into the right places.

Rule 4(4) *Avoid array copying by a simple technique as shown.*

For further essential points related to arrays see section 6.

Some programming languages such as Modula and Pascal provide the possibility to copy whole arrays of the same type simply by using the command *array*1 = *array*2 .

This is possible by special *fast* copy routines in the run-time library of these languages.

Rule 4(5) *Use the fast variants of array copying (instead of element copying) if the programming language provides it.*

Note that this can be widely used for array initialisations. For example, if an array often has to be reset to defined numbers, an array can be created in the initialisation phase of the run (storing these numbers) and then used for fast copying.

Remark *Fast* copying routines can also be given like normal routines. Borland Pascal ad ditionally provides *Move* which allows *fast* copying of every type of variable [Borland International, 1992].

5. Routines and Functions

5.1 Pre-processors and Inline functions

Some compilers provide the use of inline functions by

- using pre-compiler/pre-processor (e.g. C/C++);
- marking routines as *inline* (e.g. C++);
- having compiler options for expanding the program code of used subroutines and functions (e.g. FORTRAN);[1]

FORTRAN compilers often have a pre-processor that expands each routine call in the program by it's source code if possible. C/C++ allows the definition of different kinds of macros (e.g. numbers, C code). For instance a function **AbsDiff** is definable as a macro through

```
#define AbsDiff(a,b) ((a)>(b) ? (a)-(b) : (b)-(a))
       ...
double
       X, Y, dAbsOfXY;
       ...
dAbsOfXY = AbsDiff (X,Y);
```

The pre-compiler of C/C++ replaces the *identifier* **AbsDiff** of that macro in the C/C++ code by its definition. At all places **AbsDiff** is used there is no function call and therefore the cost of the evaluation a function (which could also be used instead of a macro) is saved by this replacement [Kernighan & Ritchie, 1990; Kölsch, 1994].

---

[1] cf. chapter 9;

Rule 5(1) *If a pre-processor is available define short functions and subroutines always as pre-compiler macros.*

Note that the use of the pre-compiler of C/C++ is not without dangers [Kernighan & Ritchie, 1990; Kölsch, 1994].

Remark That there exist compilers on workstations which are able to expand a program by optimised standard libraries like the BLAS (cf. chapter 7.1). This means that nested loops in your program will be replaced by a call of a subroutine of that libraries if possible [Willé, 1992; Willé, 1993].

The declaration of routines in C++ as being *inline* means that the compiler tests whether or not it is an advantage for the run-time behaviour of the program to insert the *object code* at places where the routine has to be called relative to the disadvantage of an increased executable binary.

Rule 5(2) *If possible declare routines as inline code.*

This can easily be done for all routines in a program. Hence the resulting application can be large, in contrast to the aim of producing small code.

Remark FORTRAN provides *statement functions* as another possibility to avoid function calls.

Such possibilities as pre-compilers, declaration as in-line etc. save function calls in the application which will especially speed up codes which are highly procedural or modular written.

5.2 Parameter lists

The cost of calling subroutines increases by transferring large variables such as big arrays or records throughout the parameter list.
The often unseen problem of transferring large variables is that not referencing such variables by a pointer means that with the call of the routine the parameter has to be copied (call-by-value) which requires more time with increasingly large parameters. Obviously, it is much faster to transfer a reference of a variable, which is an integer, than the variable itself.

Rule 5(3) *Always transfer large variables to and from subroutines referenced by a pointer (call-by-reference).*

Remark C/C++ provides the user with a safety feature. In both programming languages a *constant reference* to a variable can be made, e.g. as **function (const *aArray)**. In C++ the reference operator "&" **function (const &aArray)** can alternatively be used. The effect is that the transferred variable cannot be manipulated by the routine **function** although it is a call-by-reference. So such routine calls are both, fast *and* save.

Object oriented programming languages such as C++ can have a problem of transferring (large) variables to routines of one more dimension. With C++ it is usual to call subroutines such as constructors and destructor *implicitly* [Stroustrup, 1992].The next example where **cString** itself is a class demonstrates this property.

```
class cXYZ    {
       cString       sName, sFirstName;
       int           iOld;
       double        dLink;
              };
       ...
cXYZ   a;
       ...
void function1(double a)           {      ...    };
void function2(double *a)          {      ...    };
void function3(const double &a)    {      ...    };
       ...
function1 (a);       // this call makes a copy of the instance a
function2 (a);       // this call makes no copy and allows the manipulation of the
                     // instance a
```

```
function3 (a);       // this call makes no copy and does not allow the manipulation of the
                     // instance a
```

With each copying of the variable **a** of type class **cXYZ**, e.g. by the call of **function1**, this instance will be initialised by constructors. In our example no copy-constructor defined, and the default copy-constructor of C++ will therefore be used which makes a copy of **a** element by element. The point here is, that the copy-constructor of the class **cXYZ** will call the copy-constructor of the class **cString** and this will possibly be continued with copy-constructors of instances of further classes declared in **cString** (and so on). Such implicit multiple initialisation calls can need much time.

Rule 5(4) *Be careful in transferring instances of classes to routines using "call-by-value" because of implicit calls of subroutines like constructors.*

Remarks

- Programmers are completely unable to estimate whether and how often such implicit calls of subroutines will occur in their programs if they use object oriented libraries, especially of standard ones, if the source code is not available.
- The same argument as used for transferring variables to a subroutine can be true for returning values from subroutines.

For further information about this problem see e.g. [Stroustrup, 1992].

6. Multi-dimensional Arrays

6.1 Internal index computation, addressing, and accessing

The ordering in which matrices are stored and accessed is central to the efficient implementation of a code.
The use of multi-dimensional arrays has the following disadvantages.

- Different programming languages may have different strategies for storing values into multi-dimensional arrays, e.g. Modula and FORTRAN. Such *incompatibilities* can be significant if someone uses different programming languages for building an application.
- To obtain the position for storing a value in a multi-dimensional array, programming languages use an *internal index computation* (for storing the values internally in an one-dimensional array).

Rule 6(1) *If possible, use one-dimensional arrays instead of multi-dimensional ones (in order to save internal index computations).*

Again, rule 6(1) is greatly in conflict with the aim of writing intuitively understandable programs.
Some compilers store multi-dimensional arrays by column- (e.g. FORTRAN) and others row-wise (e.g. Modula, Pascal, Oberon, or C/C++). As 2 examples, the necessary index computation for storing a number *m* into a *n* dimensional array *aArray[i_1]...[i_n]* of size *[N_1]...[N_n]* will be presented for

- C/C++ by computing $i = \sum_{j=1}^{n} i_{n+1-j} * \prod_{v=1}^{j-1} N_v$ with leading dimensions $N_2,...,N_n$;
- FORTRAN by computing $i = 1 + \sum_{j=1}^{n} (i_j - 1) * \prod_{v=1}^{j-1} N_v$ with leading dimensions $N_1,...,N_{n-1}$;

and storing *m* into *aArray[i]*.[2] Note that the index counting in C/C++ starts normally with 0, in FORTRAN with 1, whereas in Modula, Pascal, and Oberon the first index has to be defined by the programmer and is represented internally by 0.

Remark The use of one-dimensional arrays and then doing the index computations like the compiler does has *no advantage.* The advantage lies in the possibility of formulating the problem in such a way so as to store values *consecutively* into the array which needs less index computations than *unconsecutive* storage.

The following example, using C, demonstrates that different implementations in the same programming language can have *implicitly* different numbers of operations to solve the problem [Kernighan & Ritchie, 1990; Kölsch, 1994].

```
int iI, iSum, iNo[1000], *pNo;
       ...
/* version 1 */
for (iI=0, iSum=0; iNo[iI]; iI++)
       iSum   += iNo[iI];
/* version 2 */
for (iI=0, iSum=0; *pNo; iNo++)
       iSum   += *pNo;
       ...
```

Version 1 of this example makes 4 additions in every iteration whereas in version 2 only 2 additions are necessary. This comes from the additional internal index computation of using **iNo[i]** in version 1 which operation ***pNo** does not have.

Rule 6(2) *Use constructions which need minimal internal index computations.*

Normally in higher languages arrays are of a type, e.g. int, double or more complex ones such as record, objects etc. The time for physical addressing of array elements depends on this type which has to be taken into consideration in order to reduce run-time costs.

Rule 6(3) *Minimise the cost of addressing/accessing if using multi-dimensional arrays.*

In order to explain rule 6(3) we use an example written in pseudo code.

```
       I1      = 2;
       ...
       FOR I=1 TO NVAR DO                  ***** version 1 *****
              DEY[1,I,I1] = Y[I]
              FOR J=1 TO K DO
                     YP[I]          = YP[I] + (Y[I] - DEY[J,I,I0]) * GKS[J]
                     DEY[J+1,I,I1] = YP[I]
              END
       END
       ...
       FOR I=1 TO NVAR DO                  ***** version 2 *****
              DEY[I,1,I1] = Y[I]
              FOR J=1 TO K DO
                     YP[I]          = YP[I) + (Y[I] - DEY[I,J,I0]) * GKS[J]
                     DEY[I,J+1,I1] = YP[I]
              END
       END
```

Wild jumping in the physical addresses of a computer for storing values into an array slows down an algorithm. It is better to address it *consecutively within the multi-dimensional array* therefore reducing the time needed for physical addressing.
Version 1 is therefore the faster variant for languages storing multi-dimensional column oriented, whereas version 2 is faster for row oriented languages.

---

[2] The necessary information to store a value *m* into *aArray[i]* are the leading dimensions, the starting point for counting the elements of the array and the physical address of the first element.

6.2 Page faulting

An essential point is the use of cache memory. If, for example, consecutively accessed values are very widely distributed many unwanted adjacent elements may be copied into cache without being operated upon.
In the following example we assume a cache memory of size 2^{10} = 1024 elements of type REAL.

```
 VAR
       aArray : ARRAY[1..1024] OF ARRAY[1..1024] REAL;
       dSum   : REAL;
       I1, I2 : INTEGER;
       ...
(*     version 1     *)
FOR I1:=1 TO 1024 DO
       FOR I2:=1 TO 1024 DO
              dSum := dSum + aArray[I2,I1];
       ...
(*     version 2     *)
FOR I1:=0 TO 1024 DO
       FOR I2:=1 TO 1024 DO
              dSum := dSum + aArray[I1,I2];
       ...
```

Both loops perform the same task but they address the array elements in a different order. In version 2 elements are accessed consecutively, that is they appear in storage, whereas in version 1 they are accessed every 1024^{th} element at a time. Thus, in version 1, one paging operation must be performed for every addition. This is clearly very inefficient. Compare this with version 2 in which only one page copy is needed for every 1024 operations.
Excessive page operations of the type illustrated by version 1 are called *page faults.*

Rule 6(4) *Try to limit the size of arrays so that they can fit directly into cache.*

This problem clearly depends on the way in which the programming language stores array elements and on the use of sophisticated algorithms for realising rule 6(4).
To minimize the number of paging actions is often possible to split an original large matrix problem into a sequence of sub-problems on smaller cache sized sub-matrices by means of special block or partitioned algorithms. For such special algorithms refer to the literature e.g. the numeric of linear algebra.

Remark The efficient use of cache memory (thereby avoiding *page faults*) is essential in numerical linear algebra (vector- and matrix operations etc.).

6.3 Memory stride

The example in chap. 6.2 illustrates the importance of the order in which array elements are accessed. Formally speaking, in version 1 of this example the matrix is accessed with unit stride or unit length 1 whereas in version 2 the matrix is accessed with stride 1024 or stride length 1024. Generally, the stride of any structure (e.g. a row or column) in a host structure (e.g. a matrix) can be defined as the increment in addresses between successive elements.

7. Mathematical Algorithms

7.1 Sophisticated algorithms

Chapters 7 and 8 include a wide field of *essential* points. For instance, how to implement a differential equation solver efficiently or which linear algebra solvers perform no page faults are

highly specialized tasks, which are not described here; for such issues refer to the special literature. Here we give an introduction to widely used libraries.

7.2 Standard Libraries

A simple technique to optimise program code is to use *optimised (standard)* libraries [Dongarra & Walker, 1995].
As an example the BLAS (**B**asic **L**inear **A**lgebra **S**ubprograms) library is conceived to provide highly machine-optimised code to programmers for elementary but often needed operations with vectors and matrices.
Programmers have to keep the following problem in mind as illustrated with BLAS. BLAS is available under Unix workstations and works together with several programming languages, but the representation of arrays in BLAS routines is compatible with FORTRAN and incompatible with Modula, Pascal, and C/C++ (cf. chapter 6).

Rule 7(1) *Whenever possible use machine optimised (standard) libraries.*

There are various gains in using (standard) libraries, such as

- improved portability. For most workstations such libraries are available at least as source code in FORTRAN 77, FORTRAN 90, or C/C++.
- increased program readability. For example, BLAS forces the programmer to break up the program into intuitively understandable vector or matrix operations instead of nests of DO-loops and the resulting codes are therefore often better structured.
- safer code development. Writing your own code or copying it from books may introduce program errors which can be very difficult to detect, especially in cases of complex algorithms.
- extensive documentation. The documentation of such libraries is often extensive.

Note that libraries written by specialists are often optimised with respect to (mathematical) algorithms. By using them you can directly access state-of-the-art technology without special knowledge of what includes several person years of development. Today, the development of new libraries as ScaLAPACK is done in modern programming languages such as FOTRAN 90 and C++.
Well-known examples for such *de facto* standard libraries are given in Table 1.

Short Name	Long Name	Short Description	Availability
BLAS	**B**asic **L**inear **A**lgebra **S**ubprograms	Elementary routines for vector-vector, vector-matrix, and matrix-matrix;	Machine optimised code and public domain as FORTRAN version;
BLACS	**B**asic **L**inear **A**lgebra **C**ommunication **S**ubprograms	MIMD message-passing linear algebra communication;	Machine optimised code; Commercial
NAG	**N**umerical **A**lgorithms **G**roup	Collection of general numerical routines;	Public Domain;
IMSL	**I**nternational **M**athematical **S**tandard **L**ibrary	American version of NAG, but smaller;	Commercial;
EISPACK	**Ei**genvalue**s**/Eigen vectors **Pack**age	For computation of eigenvalues and eigenvectors of matrices;	Public Domain;
LINPACK	**Lin**ear Algebra **Pack**age	General linear algebra package;	Public Domain;
LAPACK	**L**inear **A**lgebra **Pack**age	The successor to LINPACK and EISPACK;	Public Domain;
ScaLAPACK	**Sca**lably **L**inear **A**lgebra **Pack**age	Extended LAPACK;	Public Domain;
HARWELL	÷	Collection of routines with spe-cial ones for operations with sparse matrices;	Commercial, in parts public domain;

Table 1: Collection of some of the well-known *de facto* standard libraries.

Rule 7(2) *Whenever possible use mathematically, algorithmically optimised (standard) libraries.*

Today, many servers offer public domain standard libraries. As an example *Netlib* and the *eLib* are automatic repositories for general and mathematical software accessible directly by electronic mail. For further details simply send the message *send index* or *send index from <library name>* to one of the following internet addresses

- *netlib@ornl.gov*;
- *netlib@nac.no* (preferable from within Europe);
- *eLib@zib-berlin.d*e (preferable from within Germany);

Netlib is also accessible via

- the World Wide Web (WWW). The URL is: *http://www.netlib.org/index.html*;
- anonymous ftp to: *ftp.netlib.org*;
- gopher. Point your gopher browser to: *gopher.netlib.org*;

as well as anonymous rcp, Xnetlib, and CD-ROM. More information can be found on the homepage of *Netlib*.

7.3 Built-in Functions

Some compilers provide so called *built-in* functions for several elementary operations. Those functions are written in machine code and therefore fast. The availability of built-in functions depends on the compiler.

Rule 7(3) *Whenever possible use optimised built-in functions.*

For example *Modula* provides ***DEC(i)*** and ***INC(i)*** and *C/C++* provides ***i--*** and ***i++*** which are fast variants of ***i:=i-1*** and ***i:=i+1***.

7.4 Arithmetic Programming

In order to minimise the computation time of any program we have in general to reduce the arithmetic operations, i.e. mathematical routines such like *POWER, EXP, LN* as well as binary operations such as multiplication and addition.

7.4.1 Simple Arithmetic Techniques

We now examine the arithmetic operations multiplication (division) and addition (subtraction). The cost of these operations can generally be sorted in descending order as:

unnecessary arithmetic operations
> real multiplication (division) > real addition (subtraction)
> integer (index) multiplication (division) > integer (index) addition (subtraction)

7.4.1.1 Unnecessary Operations

By this, we mean operations such as multiplication with 1 or 0.

Rule 7(4) *Avoiding multiplication with 0.0 and 1.0 as well as division by 1.0 and addition/subtraction with 0.0 can speed up the code.*

7.4.1.2 Multiplication and Division

A good example is the evaluation of polynomials $a_{degree}*x^{degree}+...+a_1*x+a_0$.

Rule 7(5) *Use algorithms which provide a minimal number of binary operations.*

```
VAR
       iI     : INTEGER;
       sum    : REAL;
BEGIN
       sum    := Coefficients[0];
       FOR iI:=1 TO degree DO
              sum    := sum + Coefficients[iI] * x;
       END;
       ...
END Polynomial;
```

This example demonstrates the computation of polynomials with the minimal number of multiplication by the Horner scheme $((a_{degree}*x+a_{degree-1})*x+...)*x+a_0$ instead of using $a_{degree}*POWER(x, degree)+...+a_0*POWER(x,0.0)$ or anything similar. Note that there is a wide class of complicated problems in which an algorithm with the minimal number of operations cannot easily be found and described (cf. chap. 7.1).
Often divisions are slower than multiplication. Especially within loops, the use of the rules in chap. 4 may be helpful.

```
       ...
       dDivValue     := 1.0 / x                                (* Version 1 *)
       sum    := Coefficients[0] * dDivValue;
```

```
        FOR iI:=1 TO degree DO
                sum     := sum + Coefficients[iI] * dDivValue;
        END;

        sum     := Coefficients[0] / x;
        FOR iI:=1 TO degree DO                                  (* Version 2 *)
                sum     := sum + Coefficients[iI] / x;
        END;
        ...
```

Rule 7(6) *Use multiplication instead of division (especially within loops).*

In version 1 there is only 1 division whereas in version 2 degree+1 divisions have to be done.

7.4.1.3 Mathematical functions

For instance, in cases of computing $x^{3.0}$ it is faster to use *x*x*x* instead of *POWER(x,3.0)* because a power function often uses the algorithm $10^{3.0\ log_{10}\ x}$.

Rule 7(6) *Think about whether mathematical routines (e.g. POWER, LN) can be replaced by faster algorithms.*

7.4.2 TYPE CONVERSIONS

Pascal and other programming languages convert types within a program code automatically in contrast to Modula in which it is done explicitly due to the strict rules of type conversion. The following Pascal fragment demonstrates the problem.

```
var
        iValue1, iValue2     : integer;
        rValue          : real;
        ...
        iValue1         := iValue2 / rValue;
```

The term **iValue2/rValue** *implicitly* includes 2 type conversions. In Modula this translates to *INT(FLOAT(iValue2) / rValue)* in which *INT* and *FLOAT* are functions for the conversion of variables from one into another type. Such conversions are time consuming. The following rules should therefore be considered:

Rule 7(8) *Avoid type conversions by sophisticated program structuring.*

Rule 7(9) *In order to avoid using type conversions split algorithms into pieces in which operations are evaluated with variables of the same type.*

Note, that similar techniques should also be used with constants.

The next example demonstrates rule 7(8). The faster version 2, which has no type conversion, should be preferred.

```
VAR
        iI      : INTEGER;
        dHelp   : REAL;
        dArray : ARRAY [0..1000] of REAL;
        ...
FOR iI:=0 TO 1000 DO                (* Version 1 *)
        dArray[iI]     := iI;
END;

dHelp   := 1.0;
FOR iI:=0 TO 1000 DO                (* Version 2 *)
        dArray[iI]     := dHelp;
        dHelp          := dHelp + 1.0;
END;
        ...
```

Remark C/C++ has an implicit type conversion as do Pascal and FORTRAN. Nevertheless, my preference is, that it is safer to make type conversions

explicitly which is the better programming style since it allows for easy debugging and makes program code more understandable.

8. Program Structuring

The structure of the program can be built such that the program is fast or slow. This depends on the selected programming language and on the compiler.

8.1 Statements for decisions

8.1.1 IF-THEN-ELSE, CASE STATEMENTS, AND LOOPS

The If-Then-Else and Case construction is widely used in programs. In order to build fast programs it is essential never to use such a statement within a loop or an iterated block (e.g. a routine).

Rule 8(1) *Whenever an If-Then-Else (or Case) statements is independent of the loop (or iterated block) build the loop into that statement. Therefore double the loop (or iterated block) into the If (or all Case) and the Else part(s) of the statement.*

```
VAR
        I       : INTEGER;
        ...
        IF (a < b) THEN
                FOR I:=0 TO 100 DO
                        ...
                END;
        ELSE
                FOR I:=0 TO 100 DO
                        ...
                END;
        END;
        ...
```

This construction has to test the If-Then-Else statement only once. Building the If-Then-Else statement into the loop would cause 100 such tests. Here the code is made faster but bigger.

8.1.2 IF-THEN-ELSE AND CASE STATEMENTS

If-Then-Else statements are faster than Case statements in some programming languages. Therefore we have the rule

Rule 8(2) *Use the faster If-Then-Else instead of a Case statement.*

8.1.3 IF-THEN-ELSE STATEMENTS

If-Then-Else statements can be used quite inefficiently as the following example demonstrates.

```
        ...
        IF (i < a) ...
        IF (i < b) ...
        IF (i < c) ...
        ...
```

In this code each of a sequence of If-Then-Else statements have to be evaluated. A construction such as

```
...
IF (i < a) THEN
      ...
      IF (i < b) THEN
            ...
            IF (i < c) ...
...
```

is faster.

Rule 8(3) *Used nested if-statements instead of consecutive ones.*

8.2 Loop statements

Applications made by some Modula compilers (MacMETH, em2) have different run-time behaviour when different loop construction are used, whereas C/C++ has no such differences [Löffler & Fischlin, 1995].

Rule 8(4) *Use the fastest loop construction provided by the compiler*.

Refer to publications about benchmarks.

8.3 Procedures and Functions

The call of a routine costs more time as its parameter list increases in size. Therefore we get the rule

Rule 8(5) *Design your routines so that the required list of variables is minimized.*

This rule can conflict with efforts to produce safe programs which do not use too many global variables. However, an increase in safety by minimising interfaces which could lead to failures in manipulating variables can also be achieved.

8.4 Lists and Trees

Elements of lists and trees are logically structured by pointers in which each element stores the address in which another element of the list or tree is located in the memory. Standard examples of structuring a list by such pointers are the single and double interconnection. In contrast to arrays, lists and trees do not allow direct access to an element by addressing it. To get an element out of lists or trees their logical structure must be scanned element by element via these pointers. Working with lists and trees can therefore be very slow because of such extensive scanning.

Elementary bench marking demonstrates that working with arrays is faster then using lists or trees of the same size.

Rule 8(6) *Think about whether the use of lists and trees can be replaced by arrays.*

However, by using lists and trees instead of arrays a program gives a more flexible structure.

8.6 Recursion

Is easy to implement. The problem lies in the copying of (parts of) the recursive routine into the RAM for execution which is time consuming and needs storage. Every recursion can be rewritten in a non-recursion algorithm, thus

Rule 8(7) *Avoid recursions.*

Remark Lists, trees, and recursions are often the intuitive answer to the algorithmical question. Thus, the sections 8.4 and 8.5 are often difficult and unwanted to realise.

9. Compiler options

Normally compilers provide several options. For example it is possible to

- generate native code for the special CPU and FPU of the used machine;

to check or uncheck e.g.

- array indices, sub ranges, pointer etc.;
- integer and real arithmetic for under- and overflow;
- division by zero;

which slows down the application by checking. Other options are special for code optimisation in relation to run-time or to producing small code.

Rule 9(1) *Inform yourself about and use the options which the compiler provides for optimisation.*

Rule 9(2) *For the final (debugged and stable) version avoid all options for generating debug information, especially ones for checking ranges, under-, and overflow etc.*

10. Batch Runs

This short chapter covers with the alternatives *interactive* or *batch* run. Both of these have their advantages and disadvantages. In relation to speed, batch is the faster alternative because many time consuming processes as e.g. window handling, graphical output etc. are not used.

Rule 10(1) *Think about running programs in batch mode.*

Which of these alternatives is preferred is clearly a question of the purposes for which the program has to be used.

Conclusion

In this paper we dealt with the problem of writing time optimised programs. The main rule for writing such programs mentioned in the introduction is ***save operations whenever possible***. In 10 chapters we presented rules and examples of how to achieve this. Therefore it becomes clear

- optimisation of programs can be difficult because of not seeing the problem(s);
- the aim of optimisation is highly compiler and system dependent;
- optimisation also depends on personal preferences of the programmer;

- the given rules and optimisation in general may in conflict with other aims of programming such as writing safe, short, or readable code;

We did not include special points of programming languages (e.g. of C++), but ones which are relevant in general. For specialities please refer to the relevant literature.

References

Apple Computer, I., 1986 *APPLE Numerics Manual*. Addison-Wesley: 336 pp.

Apple Computer, I., 1993 *Building and Managing Programs in MPW, For MPW V 3.3*. Developer Technical Publications:

Blaschek, G., Pomberger, F.R. & Ritzinger, F., 1987 (Zweite, korrigierte Auflage ed.). *Einführung in die Programmierung mit Modula-2*. Springer Verlag:

Borland International, I., 1992 *Borland Pascal With Objects*:

Dongarra, J.J. & Walker, D.W., 1995. Software Libaries for Linear Algebra Computations on High Performance Computers. *SIAM Review*:

Fischlin, A., 1986. Simplifying the usage and the programming of modern working stations with Modula-2: The Dialog Machine. *Dept. of Automatic Control and Industrial Electronics, Swiss Federal Institute of Technology Zurich (ETHZ)*:

Fischlin, A. (ed.), 1991. *Interactive modeling and simulation of environmental systems on workstations*. Springer: Berlin a.o.: Ebernburg, Bad Münster am Stein-Ebernburg, BRD, pp. 131-145.

Fischlin, A. *et al.*, 1994. *ModelWorks 2.2: An interactive simulation environment for personal computers and workstations*. Systems Ecology Report No. 14, Institute of Terrestrial Ecology, Swiss Federal Institute of Technology ETH, Zurich, Switzerland, 324 pp.

Fischlin, A., Vancso-Polacsek, K. & Itten, A., 1988. *The "Dialog Maschine" V1.0*. Projekt-Zentrum IDA/CELTIA, Swiss Federal Institute of Technology ETH, Zurich, Switzerland, 8 pp.

GNU, 1993. *GNU project C and C++ Compiler (v2.4), GCC manual, G++ manual*. .

Keller, D., 1989 *Introduction to the Dialog Machine*. Bericht Nr. 5, Proijektzentrum IDA, ETH Zürich: 37 pp.

Kernighan, B.W. & Ritchie, D.M., 1990 (Zweite Ausgabe ed.). *Programmieren in C*. Carl Hanser Verlag:

Kölsch, T., 1994. *Programmiersprache C*. TÜV Akademie Pfalz, neox GmbH.

Kreutzer, W., 1986 *System simulation: programming styles and languages*. Sydney a.o.: Addison-Wesley: 366 pp.

Löffler, T.J. & Fischlin, A., 1995. *Performance of RAMSES*. Systems Ecology Report Institute of Terrestrial Ecology, Swiss Federal Institute of Technology ETH, Zurich, Switzerland.

Pudlatz, H., 1990 (2., verbesserte Auflage ed.). *Modula-2, Einführung in die Programmiersprache*: 213 pp.

Stroustrup, B., 1992 (2. überarbeitete Auflage ed.). *Die Programmiersprache C++*. Addison-Wesley:

Thöny, J., 1994. *Practical considerations on portable Modula-2 code*. : Institute of Terrestrial Ecology, Swiss Federal Institute of Technology ETH, pp. 6.

Thöny, J., Fischlin, A. & Gyalistras, D., 1995. *Introducing RASS - The RAMSES Simulation Server*. .

Vancso-Polacsek, K., Fischlin, A. & Schaufelberger, W. (eds.), 1987. *Die Entwicklung interaktiver Modellierungs- und Simulationssoftware mit Modula-2, Simulationstechnik*, Berlin, Springer Verlag, Vol. **150**, 239-249 pp.

Weber, F., 1990. Wenn Rechnen Glückssache ist. *ETH Zürich*, **Nr. 225**:

Willé, D.R., 1992. *A Short Course in Advanced FORTRAN Programming.* University of Heidelberg.

Willé, D.R., 1993. *Advanced Scientific Fortran.* University of Heidelberg.

Wirth, N., 1981. *The Personal Computer Lilith*. : Departement für Informatik ETH Zürich, Switzerland, pp. 70.

Wirth, N., 1983 (2 (corr.) ed.). *Programming in Modula-2*. Springer-Verlag, Berlin a.o: 176 pp.

Wirth, N., 1984. *Revisions and amendments to Modula-2*. . Zürich, Switzerland: Institut für Informatik ETHZ, Swiss Federal Institute of Technology, pp. 27-28.

Wirth, N. *et al.*, 1992. *MacMETH. A fast Modula-2 language system for the Apple Macintosh. User Manual. 4th, completely revised ed*. : Departement für Informatik ETH Zürich, Switzerland, pp. 116.

Zeigler, B.P., 1976 *Theory of modelling and simulation*. Wiley, New York a.o.: 435 pp.

BERICHTE DER FACHGRUPPE SYSTEMÖKOLOGIE
SYSTEMS ECOLOGY REPORTS
ETH ZÜRICH

Nr./No.

1 FISCHLIN, A., BLANKE, T., GYALISTRAS, D., BALTENSWEILER, M., NEMECEK, T., ROTH, O. & ULRICH, M. (1991, erw. und korr. Aufl. 1993): Unterrichtsprogramm "Weltmodell2"

2 FISCHLIN, A. & ULRICH, M. (1990): Unterrichtsprogramm "Stabilität"

3 FISCHLIN, A. & ULRICH, M. (1990): Unterrichtsprogramm "Drosophila"

4 ROTH, O. (1990): Maisreife - das Konzept der physiologischen Zeit

5 FISCHLIN, A., ROTH, O., BLANKE, T., BUGMANN, H., GYALISTRAS, D. & THOMMEN, F. (1990): Fallstudie interdisziplinäre Modellierung eines terrestrischen Ökosystems unter Einfluss des Treibhauseffektes

6 FISCHLIN, A. (1990): On Daisyworlds: The Reconstruction of a Model on the Gaia Hypothesis

7 * GYALISTRAS, D. (1990): Implementing a One-Dimensional Energy Balance Climatic Model on a Microcomputer *(out of print)*

8 * FISCHLIN, A., & ROTH, O., GYALISTRAS, D., ULRICH, M. UND NEMECEK, T. (1990): ModelWorks - An Interactive Simulation Environment for Personal Computers and Workstations *(out of print[] for new edition see title 14)*

9 FISCHLIN, A. (1990): Interactive Modeling and Simulation of Environmental Systems on Workstations

10 ROTH, O., DERRON, J., FISCHLIN, A., NEMECEK, T. & ULRICH, M. (1992): Implementation and Parameter Adaptation of a Potato Crop Simulation Model Combined with a Soil Water Subsystem

11 * NEMECEK, T., FISCHLIN, A., ROTH, O. & DERRON, J. (1993): Quantifying Behaviour Sequences of Winged Aphids on Potato Plants for Virus Epidemic Models

12 FISCHLIN, A. (1991): Modellierung und Computersimulationen in den Umweltnaturwissenschaften

13 FISCHLIN, A. & BUGMANN, H. (1992): Think Globally, Act Locally! A Small Country Case Study in Reducing Net CO_2 Emissions by Carbon Fixation Policies

14 FISCHLIN, A., GYALISTRAS, D., ROTH, O., ULRICH, M., THÖNY, J., NEMECEK, T., BUGMANN, H. & THOMMEN, F. (1994): ModelWorks 2.2 – An Interactive Simulation Environment for Personal Computers and Workstations

15 FISCHLIN, A., BUGMANN, H. & GYALISTRAS, D. (1992): Sensitivity of a Forest Ecosystem Model to Climate Parametrization Schemes

16 FISCHLIN, A. & BUGMANN, H. (1993): Comparing the Behaviour of Mountainous Forest Succession Models in a Changing Climate

17 GYALISTRAS, D., STORCH, H. v., FISCHLIN, A., BENISTON, M. (1994): Linking GCM-Simulated Climatic Changes to Ecosystem Models: Case Studies of Statistical Downscaling in the Alps

18 NEMECEK, T., FISCHLIN, A., DERRON, J. & ROTH, O. (1993): Distance and Direction of Trivial Flights of Aphids in a Potato Field

19 PERRUCHOUD, D. & FISCHLIN, A. (1994): The Response of the Carbon Cycle in Undisturbed Forest Ecosystems to Climate Change: A Review of Plant–Soil Models

20 THÖNY, J. (1994): Practical considerations on portable Modula 2 code

21 THÖNY, J., FISCHLIN, A. & GYALISTRAS, D. (1994): Introducing RASS - The RAMSES Simulation Server

---

* Out of print

22 GYALISTRAS, D. & FISCHLIN, A. (1996): Derivation of climate change scenarios for mountainous ecosystems: A GCM-based method and the case study of Valais, Switzerland

23 LÖFFLER, T.J. (1996): How To Write Fast Programs

24 LÖFFLER, T.J., FISCHLIN, A., LISCHKE, H. & ULRICH, M. (1996): Benchmark Experiments on Workstations

25 FISCHLIN, A., LISCHKE, H. & BUGMANN, H. (1995): The Fate of Forests In a Changing Climate: Model Validation and Simulation Results From the Alps

26 LISCHKE, H., LÖFFLER, T.J., FISCHLIN, A. (1996): Calculating temperature dependence over long time periods: Derivation of methods

27 LISCHKE, H., LÖFFLER, T.J., FISCHLIN, A. (1996): Calculating temperature dependence over long time periods: A comparison of methods

28 LISCHKE, H., LÖFFLER, T.J., FISCHLIN, A. (1996): Aggregation of Individual Trees and Patches in Forest Succession Models: Capturing Variability with Height Structured Random Dispersions

29 FISCHLIN, A., BUCHTER, B., MATILE, L., AMMON, K., HEPPERLE, E., LEIFELD, J. & FUHRER, J. (2003): Bestandesaufnahme zum Thema Senken in der Schweiz. Verfasst im Auftrag des BUWAL

30 KELLER, D., 2003. *Introduction to the Dialog Machine, 2nd ed.* Price,B (editor of 2nd ed)