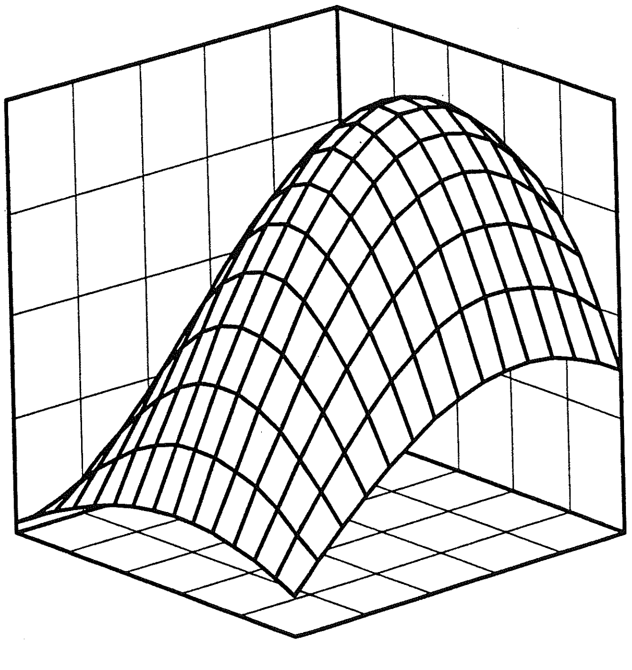
# The Fortran Simulation Translator

## FST version 2.0

Introduction and Reference Manual



PRODUCTION
ECOLOGY

ab-dlo

## Quantitative Approaches in Systems Analysis

*Address for ordering copies of volumes in the series:*
H.E. de Ruiter
TPE-WAU
Bornsesteeg 47
NL-6708 PD Wageningen
Phone: 31(0).317.475770
Fax: 31(0).317.484892
E-mail: hennie.deruiter@staff.tpe.wau.nl

*Addresses of editorial board (for submitting manuscripts):*

B.A.M. Bouman
AB-DLO
P.O. Box 14
NL-6700 AA Wageningen
Phone: 31(0).317.482382
Fax: 31(0).317.423110
E-mail: bouman@ab.dlo.nl

M.K. van Ittersum
TPE-WAU
Bornsesteeg 47
NL-6708 PD Wageningen
Phone: 31(0).317.475972
Fax: 31(0).317.484892
E-mail: martin.vanittersum@staff.tpe.wau.nl

# The Fortran Simulation Translator

## FST version 2.0

Introduction and Reference Manual

C. Rappoldt & D.W.G. van Kraalingen

Quantitative Approaches
in Systems Analysis No. 5
June 1996

**CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG**

C. Rappoldt & D.W.G. van Kraalingen

The Fortran Simulation Translator, FST version 2.0 / C. Rappoldt &
D.W.G. van Kraalingen. - Wageningen : DLO Research Institute for
Agrobiology and Soil Fertility ; Wageningen : The C.T. de
Wit Graduate School for Production Ecology. -
(Quantitative approaches in systems analysis ; no. 5)
ISBN 90-73384-46-X
NUGI 835
Subject headings: description of FST programs, array variables in FST,
subroutines called from FST, examples, user guide, reference manual of FST,
the generated FORTRAN sources

**Keywords**
Continuous simulation language, simulation, FORTRAN, crop growth simulation, modelling

## Guidelines 'Quantitative Approaches in Systems Analysis'

Manuscripts or suggestions should be submitted to the editorial board (B.A.M. Bouman, AB-DLO, or M.K. van Ittersum, TPE-WAU). The final version of the manuscripts should be delivered to the editors camera-ready for reproduction. The submission letter should indicate the scope and aim of the manuscript (e.g. to support scientific publications in journals, program manual, educational purposes). The costs of printing and mailing are to be borne by the authors.

The English language is preferred. Authors are responsible for correct language and lay-out. Overall guidelines for the format of the texts, figures and graphs can be obtained from the publication editor at AB-DLO, or from the PE office:

J.M. Braber
AB-DLO

P.O. Box 14
NL-6700 AA Wageningen
Phone: 31(0).317.475723
Fax: 31(0).317.423110
E-mail: braber@ab.dlo.nl

Th.H. Jetten
Secretariat C.T. de Wit Graduate School
for Production Ecology
Lawickse Allee 13
NL-6701 AN Wageningen
Phone: 31(0).317.485116
Fax: 31(0).317.484855
E-mail: theo.jetten@beleid.spp.wau.nl

# Preface

The easiest way of learning the principles of modeling and simulation is by using a simulation language. The equations of the model are written in a file and the simulation language calculates the behaviour of the system through time. The user does not have to worry about the details of the calculation method and about a lot of trivial matters concerning input and output.

The programs used in practical research, however, serve other than educational needs. They have to run on the computer types used now and in the future. Standardization of relatively well-established parts of a model is essential for progress and cooperation. And finally, simulation models are combined more and more with user-friendly shells, with procedures for parameter optimization and with Geographic Information Systems. All this requires the flexibility of a well-defined and widely available computer language.

For writing a simulation model in Fortran 77, many tools are in use at AB-DLO and TPE-WAU. The numerical procedure and the parts for input and output do not have to be written again and again for every new program. Still, one has to know how to use these tools and one has to understand the overall Fortran structure in which new equations are plugged in. It was the late Prof. C.T. de Wit who convinced us that standardized "simulation modules" were nice, but not enough. There still is a conflict between the requirements of good teaching and the tools used in research.

It is this conflict that made us produce the Fortran Simulation Translator (FST), a program which translates the statements of a simulation language into a standard Fortran simulation module, which can be immediately combined with all the existing tools. Like any other simulation language, FST can be used as a black box between model equations and simulation results. Experienced users, however, can take the intermediate Fortran source as a starting point for further work.

The lack of array variables in the first version of the FST translator appeared to be a more serious limitation than we first thought. In version 2, we have implemented array variables and state arrays. In doing so, we have kept in mind the purpose of FST which is the solution of a teaching problem in the first place. This has led to some restrictions on the use of array variables.

The first proposals for FST have been discussed with Herman van Keulen, Jan Goudriaan, Frits Penning de Vries and Peter Leffelaar. Gon van Laar did a lot of testing work while translating CSMP programs into FST. Her enthusiasm for our error messages and bugs has helped us to finish the job. We thank Michel Verbeek for writing the FST shell for DOS machines. And we thank many users_without_manual for their comments and patience.

We hope FST will prove to be a valuable tool, in teaching, in research and in between.

Haren, Wageningen, June 1996
Kees Rappoldt, Daniel van Kraalingen

# CONTENTS

# Samenvatting

Het eerste deel van dit rapport vormt een inleiding in het gebruik van FST, de "Fortran Simulation Translator". FST is een programma dat de statements van een eenvoudige simulatietaal omzet in een equivalent Fortran-programma met enkele daarbij behorende datafiles. Dit Fortran-programma is goed gestructureerd en bevat slechts een minimale "overhead". Het bevat in feite weinig meer dan de vergelijkingen van het simulatiemodel in een zodanige vorm dat het model doorgerekend kan worden met behulp van standaard subroutines. Het gegenereerde Fortran-programma kan daarom uitstekend als uitgangspunt worden gebruikt voor verdere modelontwikkeling. Op deze manier is gepoogd de afstand te verkleinen tussen enerzijds het onderwijs in systeemanalyse en anderzijds de praktijk van het onderzoek. Ter ondersteuning van het onderwijs is veel aandacht besteed aan goede foutmeldingen.

Iets ingewikkelder onderwerpen zoals het gebruik van array-variabelen en het aanroepen van Fortran-subroutines komen in afzonderlijke hoofdstukken aan de orde. Er is een hoofdstuk met voorbeelden, waarvan een aantal is bedoeld voor gebruikers met ervaring in het maken van modellen. Een meer formele beschrijving van de regels voor het maken van een FST-model vormt een apart hoofdstuk: de "Reference Manual". Tenslotte wordt het gegenereerde Fortran beschreven.

De structuur van de gegenereerde Fortran-modules sluit nauw aan bij de in deze serie rapporten gedocumenteerde FSE-structuur. Ten behoeve van de simulatie van gewasgroei is het gebruik van weersgegevens in FST teruggebracht tot het opgeven van een land, een station en een startjaar. Naast FSE, kan er ook Fortran worden gegenereerd dat geschikt is voor Runge-Kutta integratie en waaraan door de ervaren gebruiker "state" en "time events" kunnen worden toegevoegd.

# Summary

The first part of this report is an introduction in the use of FST, the Fortran Simulation Translator. FST is a program which translates the statements of a simple simulation language into an equivalent Fortran program with datafiles. This generated Fortran program is well-structured and contains little more than the equations of the simulation model in a form which can be executed by standard numerical subroutines. Therefore, the generated Fortran program may well be used as a starting point for further model development. In this way we tried to close the gap between the needs of teaching at one hand and the Fortran programs used in practical research at the other hand. The use of FST in teaching implies that much attention has been paid to the quality of error messages.

Somewhat more complicated subjects like array variables and subroutine calls are treated in separate chapters. There is also a chapter with example programs, some of them meant for experienced modellers. A formal description of the rules for FST is given in the "Reference Manual". Finally, the generated Fortran is documented.

The generated Fortran makes use of FSE, a Fortran structure documented in this report series. FSE supports crop growth simulation, for instance by reducing the use of weather data to specifying a country, a station and a start year. The same FST model can also be used to generate a Fortran module which can be combined with Runge-Kutta integration. The experienced user can add time or state events to this generated Fortran

# 1. Introduction

## 1.1 Outline of this manual

In this manual the Fortran Simulation Translator (FST) is introduced and documented. Chapter 2 explains the use of FST by means of a simple example program. In Chapter 3, the various aspects of FST are described in more detail. Chapters 4 and 5 are entirely devoted to the topics "Array variables" and "Subroutine call's". Chapter 6 gives some ideas and examples.

Chapter 7 introduces the file structure behind a model run with FST. This file structure is important for users who want to understand the installation of FST. But also other users who merely inspect error messages, input files, output files and listings will appreciate some understanding of the underlying structure. Chapter 8 is the Reference Manual of FST with a systematic description of the various variable types and statements.

For the advanced user, Chapter 9 provides a description of the generated Fortran subroutines, the generated data files and a brief description of the “drivers”, the numerical subroutines that organize the calculations. A brief documentation of the linked libraries TTUTIL and WEATHER is also given, but for details on these libraries the reader is referred to other manuals.

The idea of a simulation language is far from new and in this introductory chapter we give our reasons for developing this one. Then some remarks are made on the type of models suitable for FST and the limitations of FST.

As part of our testing work, various biological and physical models have been implemented in FST. The cover pages of each chapter contains a graph of simulation results with a brief description of the underlying model. Most of the programs used for these graphs are discussed in this manual.

## 1.2 Why another simulation language?

A simulation language enables a concise computer representation of a mathematical model. Therefore, de Wit and his co-workers at Theoretical Production Ecology (TPE-WAU) developed their crop growth models in CSMP (Continuous Systems Modeling Program, (IBM,1975)). By using a simulation language, the details of the numerical method and the procedures for input and output tables remain separated from the scientific model itself.

Nevertheless, during the last 10 years, the models used at TPE and AB-DLO have slowly evolved from programs written in CSMP to programs entirely written in Fortran. Several reasons can be given for that change. The first one is that Fortran subroutines form a good way to standardize well-established parts of large models. This standardization is essential for progress and cooperation, both within and between institutes. Examples are the subroutines for crop photosynthesis which are widely used in crop growth models. The second reason for the gradual shift to Fortran is the impossibility to maintain CSMP on new types of computers and in combination with new Fortran compilers. The third reason is that a simulation language tends to be closed in itself, whereas simulation models are combined more and more with procedures for parameter optimization and GIS.

Simulation models in Fortran are difficult to handle, however. They require experience and a lot of discipline from the user. Within an organization, efficiency can be preserved only if the structure of the models and their technical, non-scientific parts are standardized. Our contribution to that standardization can be found in van Kraalingen & Rappoldt (1989), Rappoldt & van Kraalingen (1990), van Kraalingen *et al.* (1991), and van Kraalingen (1995). This work has reduced the writing of a new model to writing a single "simulation module", a Fortran subroutine in a precisely defined form containing the equations of the model and some necessary "overhead".

Even a standardized simulation module appeared to be more difficult to handle than a simulation language. As also mentioned in the Preface, there still appeared to be a conflict between the requirements of research groups and the tools needed in teaching the principles of simulation and crop growth. We hope that FST, the Fortran Simulation Translator, closes this gap. It has been designed to form a bridge between the use of a simulation language on the one hand and research models in Fortran on the other.

FST translates the statements of a simulation language into a Fortran simulation module. Hence, the central part of the Fortran program, the part containing the actual equations, is generated by the computer running FST. The generated Fortran module is linked with procedures for numerical integration, for reading input files and producing output files. Then the model can be executed.

To a generated simulation module belong several data files, also generated. And for reading these data files, the generated Fortran program contains calls to input subroutines from the utility library TTUTIL (Rappoldt & van Kraalingen, 1990 ; van Kraalingen & Rappoldt, 1996). This illustrates the high degree of integration between FST and the Fortran tools developed earlier.

FST is not meant as a general simulation language. On the contrary, FST is limited to the state variable approach of continuous systems and only two integration methods have been implemented. FST generates an interesting intermediate product, however: a well structured Fortran program with data files. The possibility to use this generated Fortran as a starting point for further work makes FST a valuable tool. As far as we know, none of the existing simulation languages offers this possibility, although they may well be more powerful from a mathematical or numerical point of view.

The high quality of the FST-generated Fortran model is reflected, for instance, by the possibility of reruns for (many) different values of arbitrary input parameters. The new parameter values just have to be specified in a data file. This implies that FST models can be easily combined with procedures for parameter optimization and GIS. Calling the generated Fortran model from any higher level system (like a GIS system) does not require changes in the generated Fortran.

The main function of FST, a bridge between teaching and research, implies that there is no need for extending its possibilities indefinitely. We feel that the present version 2.0 serves the purposes it was meant for and we do not foresee major extensions in the future.

## 1.3 Principles and limitations of FST

The FST translator is a program, which reads the statements of an "FST simulation model". Obviously, there are rules for writing a model in FST and these rules define the simulation language. For historical reasons, part of the FST simulation language is equal to CSMP. Other aspects have been adapted or are completely new.

An FST simulation model follows the so-called "state variable approach". The status of a system is described by one or more continuous state variables. These state variables change with time. The rates of change depend on the current values of the state variables, on time itself and possibly on a number of "model parameters". Clearly, the state variables have to be given initial values. Starting from these initial values, future values of the state variables are then calculated by integrating the rates of change.

An FST simulation model contains little "overhead". A model may just consist of equations for the rates of change, the start values of the state variables, the start and finish time and the names of the output variables (see also Chapter 2). After reading the statements of the FST model, the translator "thinks them over" and writes an equivalent Fortran module with associated data files.

The FST translator does more than just copying the model equations to a Fortran subroutine. The internal consistency of the model is carefully verified. A quantity A is expressed in terms of B and C, for instance. In an ordinary programming language, A can be redefined many times. In an FST model, however, there can be only one such statement. If, in addition, B is expressed in terms of A and C, the FST translator produces an error message because A depends on B and B depends on A. The FST translator can produce more than 350 different error messages. Much attention was paid to their clarity. An FST model without translation errors is, at least technically, a good model.

The rate of change belonging to a state variable is in fact its first order time derivative. Formally, the equations for the rates of change form a set of coupled, first order differential equations, the "equations of change". Analytical solutions of the equations of change are the most elegant ones but they are impossible in most situations. In the literature on numerical solutions, simulation is sometimes described as "solving an initial value problem" and many different methods are known. For a readable introduction we refer to Press *et al.* (1986). In the technical literature, the programs which implement the various methods are known as "ODE solvers" (ODE means Ordinary Differential Equation). In FST, only the basic Euler integration and a Runge-Kutta method with controlled integration error have been implemented.

The two integration methods used in FST have been implemented in the form of “model drivers”. These are Fortran subroutines that “drive” the simulated process from beginning to end. The integration methods used in FST are both explicit methods. This implies that FST is not suitable for solving equations that require an implicit integration method. A second limitation is that time and state events during the simulated process are not possible. Such events are moments or system states at which special things happen. Some of the model drivers themselves, however, can handle time and state events. Hence, experienced users can add events to an FST-generated Fortran module (see Chapter 9).

The simulation language FST lacks control structures. An FST model consists of a number of mathematical relations between quantities without IF..ELSE..END IF or other control structures. In case of two alternative formulas there are simple tricks to get around this limitation. In complicated cases, however, one needs a separate Fortran subroutine which contains the desired structure. The subroutine is then called from the main FST program. The Fortran statements of the subroutine are not verified by the FST translator.

# Chapter 2

# Getting Started

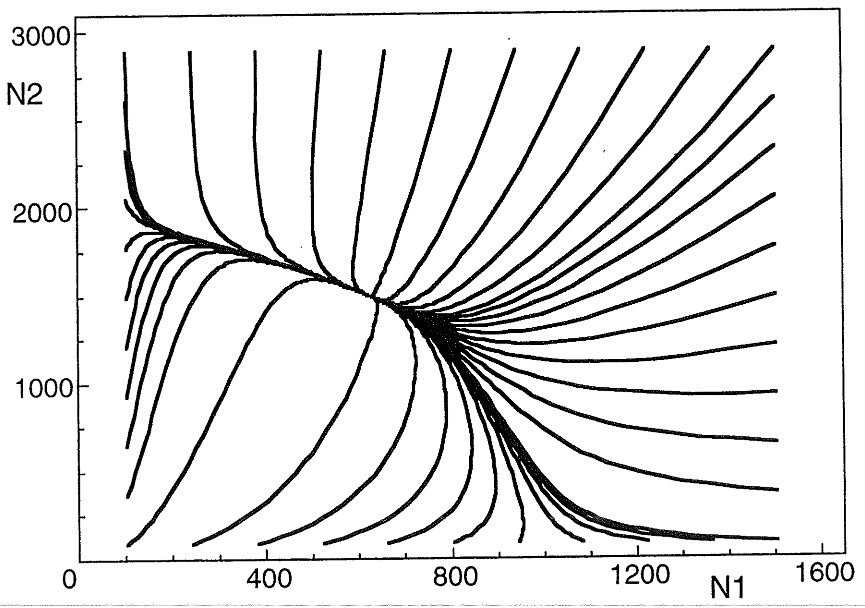


*Figure 2.1 on the title page of this chapter.* Two biological populations with sizes N1 and N2 according to the Lotka-Volterra competition model in Listing 2.3 on page 15. The model has been initialized at 40 start positions along the edge of the graph. The 40 curves describe the status of the system through time. From each start position the two populations eventually reach the same equilibrium status given by (N1,N2) = (625,1500).

# 2 Getting started

The Fortran Simulation Translator (FST) has to be installed on your computer. If this has not been done yet, you need the installation instructions for the computer type you use. These instructions are provided with the floppy disks.

Once FST has been installed you need to know the commands for running an FST program. The first step is always to type in the program with a text editor. This results in a file with a name usually ending with extension .FST. The second step, running the model, is usually an automated procedure which is started by pressing a single key. In other cases one has to give a sequence of commands. Some details for Macintosh, MS-DOS and VAX/VMS can be found in Chapter 7. Also for an overview of the various computer programs and files involved in a model run, we refer to Chapter 7 (especially Figure 7.2 on page 88).

Although the procedure for running an FST model differs among different types of computers, the FST simulation language itself is the same on all machines. This chapter begins with a simple example which is then extended in two different ways.

## 2.1 A small FST program

The model in Listing 2.1 shows a small FST model, which is easily typed into a file. It describes exponential growth according to the well-known equation

$$\frac{\mathrm{d}x}{\mathrm{d}t} = ax$$

with, as initial condition, the value 1.0 for the state variable X.

The first statement, INITIAL, marks the beginning of the initial calculations. These calculations have to be carried out before the actual process simulation is started. This simple model does not require initial calculations, however, and the initial section is empty.

The second statement, DYNAMIC, marks the beginning of the dynamic calculations. There are two dynamic calculation statements, the INTGRL ("integral") statement and the calculation of the rate of change RX. The INTGRL statement tells the FST translator that X is a state variable which has to be initialized as IX and whose rate of change is RX. The calculation of RX is the actual equation of change, expressing RX as function of the state variable X.

The indentation of the two calculation statements in Listing 2.1 is optional. The indentation is used in this chapter to emphasize the difference between calculation statements and other types of statements. Later in this manual indentation is not used.

The INCON statement defines the initial constant IX. The PARAMETER statement defines the yet undefined model parameter A, appearing in the expression for RX. This completes the description of exponential growth. Note that the values assigned to IX and A in the INCON and PARAMETER statement need to be real numbers and cannot be replaced by expressions. Hence, the INCON and PARAMETER statements are not calculations.

The last four statements are necessary for a numerical solution. The TRANSLATION_GENERAL GENERAL statement selects a numerical method, i.e. Runge-Kutta integration method implemented in the driver RKDRIV. This is a "work horse" suitable for many problems involving ordinary differential equations. RKDRIV adapts its time step in order to control the accuracy of the numerical solution. The other general driver EUDRIV (activated by writing DRIVER='EUDRIV') and the FSE driver (activated with a TRANSLATION_FSE statement) use the value of DELT as a fixed time step.

Clearly, the translator also has to know a start and finish time for the simulation (STTIME and FINTIM). The time step DELT is used as a first guess only. The actual time steps taken are made smaller or larger according to the accuracy of the integration steps. Finally, X is selected as output variable and the time interval between successive output values is set by setting the timer variable PRDEL to 0.5

Listing 2.1 Exponential growth in FST

```
INITIAL                                         <-calculations from here are initial
DYNAMIC                                         <-calculations from here are dynamic
  X  = INTGRL (IX,RX)                           <-state, initial value and rate of change
  RX = A * X                                    <-calculation of rate of change
INCON IX=1.0                                    <-set initial value
PARAMETER A=0.1                                 <-a single model parameter
TRANSLATION_GENERAL DRIVER='RKDRIV'             <-the translation mode is selected
TIMER STTIME=1.0 ; FINTIM=10.0 ; DELT=0.1       <-see text on DELT
PRINT X                                         <-some output is required
TIMER PRDEL=0.5                                 <-time interval between output times
```

Execution of the model in Listing 2.1 leads to an output file RES.DAT in which the value of X is given between the start time STTIME and the final time FINTIM with intervals 0.5. Before extending this model in the next sections, there are a few remarks to be made.

The FST program in Listing 2.1 shows that a variable can be used in calculations before it is defined as a model parameter or as an initial constant with INCON. In fact, the order of the statements in Listing 2.1 is almost arbitrary. The only two requirements are that the INITIAL statement precedes the DYNAMIC statement and the DYNAMIC statement precedes the two calculation statements. All statement orders that satisfy these two requirements lead to the same result. This implies that the INCON, PARAMETER, TRANSLATION_GENERAL, TIMER and PRINT statements can be put anywhere in any order, also in between the calculation statements.

Although statement order is almost free, it is useful to adopt a certain style. In Listing 2.1, for instance, the first four statements give the equations of the model. The INCON and PARAMETER statement specify the numerical input values and the last four statements control the simulation and its output.

The most important exception on the rule of a free statement order is the distinction between initial and dynamic calculations. This distinction is made by means of the section statements INITIAL and DYNAMIC. There are several other FST section statements. Some of them are introduced in Section 2.2 below. The full structure of an FST program is described in Section 3.1.1.

## 2.2 Extension 1: Weather dependent growth

The exponential growth model of Listing 2.1 uses a constant relative growth rate A. Now, this relative growth rate is made dependent on temperature leading to the program in Listing 2.2. In addition to the temperature dependency, also technical changes have been made, which illustrate the possibilities of FST. An overview of the changes is:

- The program has been given a title by means of a TITLE statement.
- Comment statements beginning with a "*" have been added to the model
- The MODEL and END statements are introduced.
- The initial amount IX is calculated.

- The relative growth rate A depends on the average daily temperature. This dependency is described as a FUNCTION specified by a few points. Between these points function values are estimated by means of linear interpolation.
- The average daily temperature is estimated from the daily minimum and maximum value.
- A WEATHER statement is used specifying a country, a station and a year. The use of weather data implies that the unit of time is day.
- A FINISH condition is set.
- There are a few more output variables.
- The selected driver is changed into EUDRIV.
- Two more runs are made, for 1986 and 1987, by means of two rerun sections following the MODEL section.

Below, these changes are clarified in greater detail.

## 2.2.1 A program TITLE

The TITLE statement describes the content of the model. The descriptive text could be written also in comment statements. The advantage of a TITLE statement, however, is that the text appears in the output file and in the header of the generated Fortran program.

## 2.2.2 Program sections and statement order

The second statement in Listing 2.2 is the MODEL statement, which marks the beginning of the model section. The model section is terminated by the first END statement. For clarity, the model section of Listing 2.2 has been indented. The model section of an FST program contains all calculation statements, input statements, control statements and output statements. Note that the program of Listing 2.1 consists of a model section only.

In fact, the MODEL statement is optional. In short FST programs, like the one in Listing 2.1, the MODEL statement is often omitted, but in more complicated programs it is useful to distinguish between the model section (the core of the program) and the other sections.

The distinction between initial calculations and dynamic calculations was introduced already in the previous section 2.1. Listing 2.2 shows that these sections (in fact subsections) lie entirely within the model section. The INITIAL section is not empty this time. It contains the calculation of the initial value IX, which depends on parameter B. The PARAMETER statement defining B immediately follows the calculation of IX, thus expressing that IX and B belong together. The PARAMETER statement could have been placed anywhere in the model section, however.

The three calculation statements for X, RX and TDMEAN form the DYNAMIC section. They describe the dynamic behaviour of the system. After finishing the simulation, it is possible to carry out so-called terminal calculations for evaluating the results of the model. These terminal calculations form the terminal section, which is preceded by the section statement TERMINAL. A terminal section is not present in Listing 2.2.

Hence, the calculation statements of a model are organized in three groups: the initial, dynamic and terminal calculations. The three groups are preceded by the section statements INITIAL, DYNAMIC and TERMINAL. Within each group the order of the calculations is free. The indentation of the calculation statements in Listing 2.2 is optional and the other statements of the model section may be placed in between the calculation statements, like for instance the PARAMETER statement defining B.

### 2.2.3 An interpolation FUNCTION

The relative growth rate A is no longer a model parameter but is calculated as function of the average daily temperature TDMEAN. This dependency is specified by of means an interpolation function. The statement "A=AFGEN(TFUN,TDMEAN)" interpolates between the points of the function TFUN which in turn is defined in the FUNCTION statement.

Listing 2.2 Exponential growth depending on average temperature

```
TITLE Temperature Dependent Growth               <- model title appears in output file
MODEL                                            <- the FST model starts from here
  INITIAL                                        <- calculations from here are initial
    IX = ABS(B)                                  <- calculate initial value IX
  PARAMETER B=1.0                                <- the model parameter B required for IX
  DYNAMIC                                        <- calculations from here are dynamic
    X = INTGRL (IX,RX)                           <- state, initial value and rate of change
    RX = A * X                                   <- calculation of rate of change
    A = AFGEN(TFUN,TDMEAN)                       <- AFGEN interpolates between points of TFUN
* empirical function relating the relative
* growth rate to daily average temperature
  FUNCTION TFUN = -20.0, 0.0, ...                <- note the statement continuation in FST !!
                   +0.0, 0.0, ...
                  +20.0, 0.1, ...                <- between 0 and 20 degrees the relative
                  +40.0, 0.1, ...                   growth rate increases. Above 40 degrees it
                  +42.0, 0.0, ...                   steeply decreases, reaching 0.0 at 42
                  +50.0, 0.0                        degrees.
* estimate the average daily temperature
  TDMEAN = (TMMN + TMMX) / 2.0
  WEATHER CNTR='NLD' ; ISTN=1 ; IYEAR=1985       <- country, station Wageningen and year
  WEATHER WTRDIR='C:\SYS\WEATHER\'               <- weather data directory

  FINISH X > 100.0*IX                            <- a finish condition
  TIMER STTIME=1.0 ; FINTIM=1000.0 ; DELT=1.0    <- unit of time is a day, due to WEATHER use
  TIMER  PRDEL=1.0
  PRINT X,TMMX,RX                                <- three output variables
  TRANSLATION_GENERAL DRIVER='EUDRIV'            <- translation mode and driver
END                                              <- end of MODEL section
  WEATHER IYEAR=1986                             <- rerun for 1986
END                                              <- end of first rerun
  WEATHER IYEAR=1987                             <- rerun for 1987
END                                              <- end of second rerun
STOP                                             <- end of all calculations
```

### 2.2.4 WEATHER

The next step is to calculate the average daily temperature TDMEAN. In Listing 2.2 this is done as the average of the daily maximum temperature TMMX and daily minimum temperature TMMN. TMMX and TMMN are predefined variables. They cannot be defined by the user and their values are taken from data files specified by the WEATHER statement(s).

The function of the WEATHER statements is to select a country, a station and a start year. Further, the directory with weather data files has to be specified. This information enables FST to access weather data files by means of the procedure described in van Kraalingen *et al.* (1991). In that report also the format of the weather data files is described. Note these files are not read by the FST translator itself. The data files are accessed only during the actual model runs. Lacking weather files or absent data items may lead to a runtime error message of the weather system as described in van Kraalingen *et al.* (1991).

Apart from the two daily temperatures, there are four other predefined weather variables: the total global radiation RDD, the vapor pressure at 9 a.m. VP, the average wind speed WN and the daily rainfall RAIN. Further, some weather station data and some calendar data are supplied by means of similarly predefined variables (see for details Section 3.2.4.4).

The use of weather data implies that the unit of time is a day. The value of STTIME should be between 1.0 and 365.999... for an ordinary year and between 1.0 and 366.999... for a leap year. During the simulation, the values of the weather variables are automatically updated each day. If a next year is reached, the data of that next year are accessed.

### 2.2.5 A FINISH condition

In order to keep the output values within limits, the simulation is terminated as soon as X becomes 100 times as large as IX. This is achieved by means of the FINISH statement. The ordinary finish time FINTIM, defined in the first TIMER statement, is used as a safeguard against excessive simulation times. It is set to a 1000 days and the time step DELT is 1 day.

More than a single FINISH statement may occur in an FST model (see Section 3.2.4.5). The simulation is terminated as soon as at least one of the conditions is valid.

### 2.2.6 Why EUDRIV?

The choice of the driver EUDRIV requires some explanation. We have a rough description of the temperature dependency of A, making use of the average daily temperature approximated as (TMMN+TMMX)/2.0. The daily temperature cycle is not taken into account and, as a consequence, the growth process within each day is not really considered. Therefore, the driver EUDRIV with a fixed time step of 1.0 day is a suitable choice.

For a truly continuous model, like the one in Listing 2.1, the driver RKDRIV is the right choice. In Listing 2.2, however, we have jumps in TMMX and TMMN while going from one day to the other. Therefore there are also jumps in TDMEAN and in the relative growth rate A. The driver RKDRIV would interpret these jumps as the result of a lack of accuracy and would then reduce its time steps around midnight. Clearly, the result of such calculations would still be close to the result obtained by EUDRIV. The use of RKDRIV for a growth rate specified per full day, however, is at least inefficient and may also be seen as a conceptual error.

### 2.2.7 Programming style

The END statement marks the end of the model section. Overviewing the entire model section it should be noted that the state variable X is the first dynamic variable calculated, then its rate of change RX depending on A, then A depending on TFUN and TDMEAN. Then the function TFUN is defined and TDMEAN is found from weather data. Finally, the required weather data are specified. This is an example of a top-down approach to the problem. At first, the relations between the important model variables are given. Then the variables appearing in those relations are calculated, and finally, the remaining undefined parameters are specified. This programming technique heavily relies on the fact that the statement order in the model section is essentially free (with the exception that initial, dynamic and terminal calculations cannot be mixed).

A free statement order has its price. In the model section, each "user variable" has to be defined once and only once. The FST translator carefully verifies the completeness of a model and does not produce results if variables are left undefined, are defined twice, or if there are any loops in their interdependency (for instance, A depends on B, B on C and C on A).

In the dynamic section of Listing 2.2, one might see an interdependency loop in the INTGRL statement, defining the state variable X, in combination with the calculation of RX, in which X occurs. INTGRL statements, however, are kept outside the evaluation of dependency loops. The reason is that the interdependency between X and RX belongs to the state variable approach: given the value of the state variable(s), the rate(s) of change can be calculated. The INTGRL statement in Listing 2.2 just means that a state variable X has to be initialized as IX and that RX is its rate of change. In a sense, the interdependency of X and RX is the very reason for writing the model.

## 2.2.8 Rerun sections

A very useful facility of FST is the possibility to specify reruns. After a complete model section a rerun can be specified by simply redefining one or more variables. In Listing 2.2 the start year IYEAR has been redefined twice in two rerun sections. Similarly, parameter B could be redefined by means of another PARAMETER statement. Each rerun section is terminated with an END statement and the whole sequence may be terminated with a STOP statement.

Calculations may not occur in rerun sections. The calculations define the mathematical structure of the model, which cannot be changed. Reruns are always based on new variable values. These are not necessarily numerical values, however. Also the value of DRIVER or the country name in a WEATHER statement can be redefined in a rerun section.

## 2.2.9 Results

Results of the model in Listing 2.2 are shown in Figure 2.2. The vertical scale is logarithmic which implies that the curves show the relative increase of X (exponential growth would lead to a straight line). The curves indeed show that the relative growth rate strongly increases in the northern temperate summer starting around day 80.

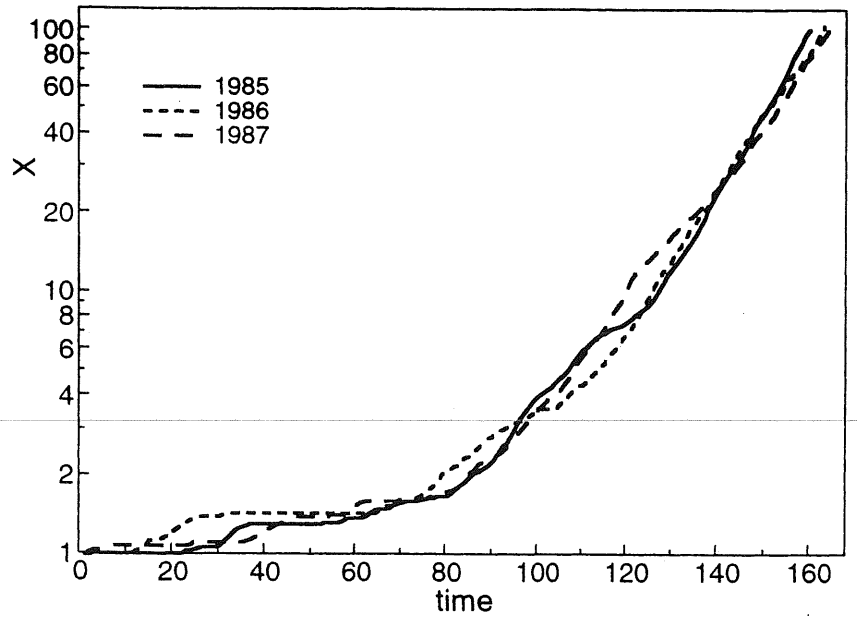


Figure 2.2 The status variable X as function of time for the program in Listing 2.2.

## 2.3 Extension 2: A second state variable

We add a second state variable X2 to the model in Listing 2.1. The equations of change for the two state variables X1 and X2 are coupled, which means that the rate of change for X1 depends on both X1 and X2. Listing 2.3 gives the adapted model. Again, we make some remarks on the changes.

The TITLE, MODEL and DYNAMIC statements have been discussed already in Section 2.2. The dynamic calculations in Listing 2.3 show that the inclusion of a second state variable is straightforward. A second INTGRL statement is added and the rates of change are just calculated according to the two equations of change describing the system.

Listing 2.3 Growth with intra- and interspecific competition

```
TITLE Lotka-Volterra competition                     <- model title appears in output file
MODEL                                                <- the FST model starts from here
  INITIAL                                            <- there are no initial calculations
  DYNAMIC                                            <- calculation from here are dynamic
*   the state variables
    X1 = INTGRL (IX1,RX1)                            <- time integration for state variable X1
    X2 = INTGRL (IX2,RX2)                            <- time integration for state variable X2
  INCON IX1 = 100.0 ; IX2 = 100.0                    <- two initial conditions
*   the growth rates
    RX1 = RGR1 * X1
    RX2 = RGR2 * X2
*   the relative growth rates reduced by competition
    RGR1 = A1 * (1.0 - X1/K11 - X2/K12)
    RGR2 = A2 * (1.0 - X2/K22 - X1/K21)
* the maximum relative growth rates                   <- the model parameters with their meaning
  PARAMETER A1=0.1 ; A2=0.2                             indicated in comment statements
* the carrying capacities for the species alone
  PARAMETER K11=1000.0 ; K22=2000.0
* the competition parameters
  PARAMETER K12=4000.0 ; K21=2500.0
* simulation control
  TIMER STTIME=0.0 ; FINTIM=100.0 ; DELT=0.1 ; PRDEL=1.0
  TRANSLATION_GENERAL DRIVER='RKDRIV'                 <- the model is continuous
  PRINT X1,X2
END
  INCON IX1 = 240.0 ; IX2 = 100.0                     <- reruns for a series of start values
END
  INCON IX1 = 380.0 ; IX2 = 100.0
END
  ....
  INCON IX2 = 2340.0 ; IX1 = 1500.0
END
  INCON IX2 = 2620.0 ; IX1 = 1500.0
END
  INCON IX2 = 2900.0 ; IX1 = 1500.0
END
```

### 2.3.1 On the model

The implemented equations of change are

$$\left| \begin{aligned} \frac{dx_1}{dt} &= A_1 x_1 \left(1 - \frac{x_1}{K_{11}} - \frac{x_2}{K_{12}}\right) \\ \frac{dx_2}{dt} &= A_2 x_2 \left(1 - \frac{x_2}{K_{22}} - \frac{x_1}{K_{21}}\right) \end{aligned} \right.$$

The parameters A1 and A2 are the relative growth rates at low values of both X1 and X2. For larger values, the two growth rates are reduced. It can be shown that there is a stable equilibrium when K21>K11 and K12>K22. This means, roughly speaking, that stability requires the intraspecific competition to be stronger than the interspecific competition.

## 2.3.2 The INCON and PARAMETER statements

An INCON or PARAMETER statement can be used to define several variables in a single statement. The various definitions are separated by means of a semicolon ";". In Listing 2.3, each PARAMETER statement defines a pair of model parameters, which expresses the symmetry between the two equations of change. Alternatively, each model parameter can be defined by means of a separate PARAMETER statements. There is no functional difference between the two methods. It is just a matter of style.

## 2.3.3 The reruns

Listing 2.3 shows just a part of the 39 rerun sections in the original FST file. In each rerun section the initial conditions are redefined. Figure 2.1 on the title page of Chapter 2 shows the results of the 40 simulation runs, not as function of time, however, but as lines in the (X1,X2) plane.

Each point of that plane represents a possible status of the system. The 40 initial conditions were chosen along the edge of a rectangle in the plane. From each of the 40 initial positions, the system has followed a path through "state space". The figure shows that all these paths end up in the same point (625,1500), the stable equilibrium between the populations X1 and X2. This equilibrium is clearly independent of the initial conditions. It depends, however, on the values of the four parameters K11, K12, K22 and K21.

# Chapter 3

# Description of FST programs

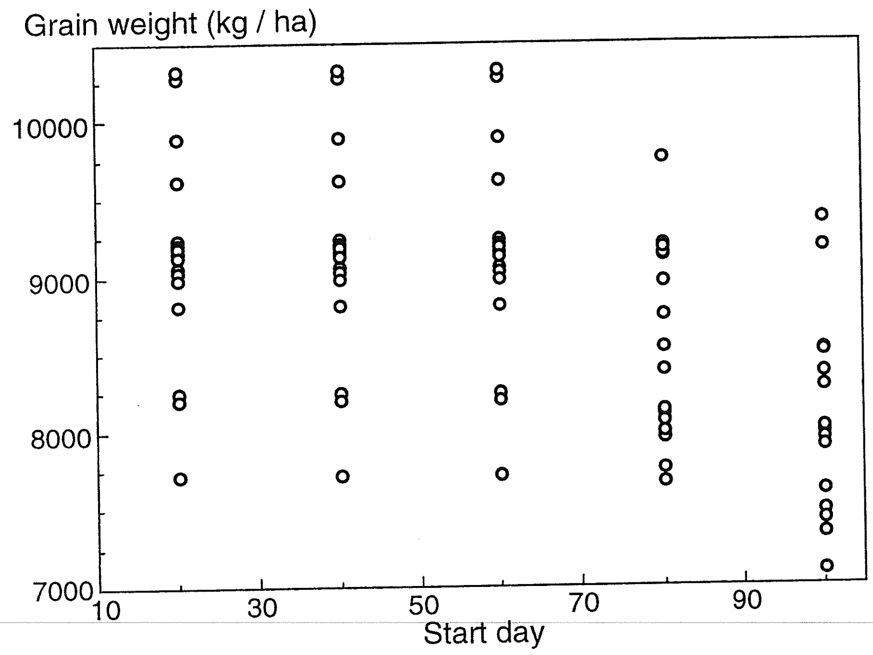


*Figure 3.1 on the title page of this chapter.* The potential yield of Spring Wheat as a function of the start day of the simulation. The results for a number of years are combined in a single graph. The temperature and radiation differences seem to be more important than the start day (for potential yield!). The crop growth model was taken from Goudriaan & van Laar (1994).

# 3 Description of FST programs

In this chapter, the program sections and FST statements are introduced in greater detail. The calculations, the control statements and the model input and output are described from a practical point of view. For a formal description of statement types and variable types, the FST Reference Manual (Chapter 8) is more suitable.

Not much is said on variable arrays and the use of Fortran subroutines since these topics are separately discussed in the Chapters 4 and 5.

## 3.1 Program structure

### 3.1.1 Program sections

In Listing 3.1 the skeleton of an FST program is given. This skeleton consists of the section statements at the beginning of the program sections. A full FST program starts with a DECLARATIONS section, which may only be preceded by TITLE statements. In the DECLARATIONS section array variables and subroutines are declared.

The MODEL section describes the actual model by means of calculation statements, input statements, output statements and simulation control statements. In the previous chapter a few examples have been extensively discussed. The calculations in the MODEL section are usually subdivided in INITIAL, DYNAMIC and TERMINAL ones. If this subdivision lacks, all calculations are assumed to be dynamic. The input, output and control statements may occur anywhere in the MODEL section, also in between or following the calculations.

Listing 3.1 The skeleton of an FST program

```
TITLE Skeleton  <- one or more TITLE statements may be anywhere before the first END
DECLARATIONS
...             <- declaration statements
MODEL
...             <- input & output statements and simulation control statements may occur here
...                and throughout the INITIAL, DYNAMIC and TERMINAL section in arbitrary order.
    INITIAL
    ...         <- initial calculations in arbitrary order
    DYNAMIC
    ...         <- dynamic calculations in arbitrary order
    TERMINAL
    ...         <- terminal calculations in arbitrary order
...
END
...             <- reruns section(s), each ending with END
END
STOP
...             <- Fortran subprograms
ENDJOB
```

Following the END of the MODEL section, reruns can be specified by means of statements which redefine one or more of the input or control variables. Each rerun section is terminated by another END statement. The list of reruns is terminated by STOP. Behind STOP one or more Fortran subprograms may be included in the FST file. The Fortran section is terminated by the ENDJOB statement, the last statement of the file.

Not all section statements are necessary in each FST program. If the first statement of an FST program is a PARAMETER statement, for instance, the FST translator “knows” that there are no declaration statements and that the MODEL section begins. In general, a section statement is required only if the rest of the program cannot be properly classified without it.

This implies that the section statements DECLARATIONS, MODEL and ENDJOB can always be omitted. INITIAL and TERMINAL are required only if there are any initial or terminal calculations. And END and STOP are required only if there is anything following them.

Hence, an FST program with a MODEL section and dynamic calculations only, does not require section statements (e.g. Listing 2.1 on page 10). Large FST programs become more readable, however, by including the section statements, even if some of them are not necessary.

## 3.1.2 The smallest FST program

The program in Listing 3.2 consists of a TIMER and a PRINT statement only. The TIMER statement defines the timer variables STTIME, FINTIM and DELT. These three timer variables control the start time, the finish time and the time step of the simulation (in case of a variable time step method, DELT is the initial time step). The second statement is an output statement. An FST program should produce at least some output.

Listing 3.2 The smallest FST program

```
TIMER STTIME=0.0; FINTIM=10.0; DELT=0.1
PRINT STTIME
```

The program in Listing 3.2 works, although no process is simulated. Only time itself is simulated between 0.0 and 10.0 but nothing else happens. Listing 3.3 gives the same, empty model written now as a complete FST program.

Listing 3.3 The program from Listing 3.2 with all section statements added and with a translation statement.

```
DECLARATIONS
MODEL
TIMER STTIME=0.0; FINTIM=10.0; DELT=0.1
INITIAL
DYNAMIC
TERMINAL
PRINT STTIME
TRANSLATION_GENERAL DRIVER='RKDRIV'
END
STOP
ENDJOB
```

The TRANSLATION_GENERAL statement in Listing 3.3 represents the default choice for translation mode and driver. In Section 3.2.4.1 the TRANSLATION_GENERAL statement is described in more detail.

## 3.1.3 Classification of FST statements

The FST statements can be classified according to their function in the program. The types of statements are:

**Comment statements**. Comment statements begin with an asterisk "*" and contain a description in words or any other comments on the program. Comment statements may occur anywhere in an FST program. Useful comments are literature references, units and dimensions, names of well known equations etc.
Comment statements preceding a calculation statement are transferred to the generated Fortran. Comment statements above other statements are neglected.

**Section statements**. The FST sections have been introduced already in Section 3.1.1. The section statements are DECLARATIONS, MODEL, INITIAL, DYNAMIC, TERMINAL, END, STOP and ENDJOB. For details see Section 3.1.1 and Section 8.4.

**Declaration statements**. These statements may only occur in the declaration section of the FST program. The FST declaration statements are ARRAY and DEFINE_CALL. For details see Section 3.2.2.

**Input statements**. These statements define input variables of the model. They should be part of the MODEL section of the program. Some of them also may occur in rerun sections where they redefine model input. The FST input statements are PARAMETER, INCON, CONSTANT, FUNCTION and ARRAY_SIZE. For details see Section 3.2.3.

**Simulation control statements.** These statements control the simulation by means of giving values to simulation control variables or otherwise. They should be part of the MODEL section of the program. Some of them also may occur in rerun sections where they redefine control variables. The simulation control statements are TRANSLATION_GENERAL, TRANSLATION_FSE, TIMER, WEATHER and FINISH. For details see Section 3.2.4.

**Output statements.** These statements control the model output. The output statements are PRINT, OUTPUT and TITLE. PRINT and OUTPUT statements should be part of the MODEL section of the program. TITLE statements may be anywhere before the rerun sections. For details see Section 3.2.5.

**Calculation statements.** These statements describe relations between variables. There are two types of calculation statements: assignments and subroutine calls. The calculations are part of the MODEL section and can be subdivided in initial, dynamic and terminal calculations by means of the section statements INITIAL, DYNAMIC and TERMINAL (see Section 3.1.1). The calculation statements are sorted by the FST translator in order to find a computational order (see Section 3.5). For details see Section 3.2.6

In some FST error messages on statements or statement order the term "keyword" is used. All FST statements contain a keyword, except comments and calculations. The keyword of a statement is simply the first complete word of the statement. The keyword of a TIMER statement is "TIMER" and the keyword of a PRINT statement is "PRINT". Hence, all statements are named after their keyword. Functional descriptions of all statement types are given in Section 3.2. More formal statement structures can be found in Chapter 8.

### 3.1.4 Classification of FST variables

An important principle of FST is that each variable can be defined only once. The definition of a variable means that a value is assigned to it or that it is calculated with a calculation statement. Most statement types directly correspond to certain variable types, e.g. timer variables are defined by means of TIMER statements, parameters are defined by means of PARAMETER statements. This logic leads to a total of 12 variable types. Together with 3 types of functions and subroutines, the number of FST symbol types amounts 15.

If the translator is instructed to make a symbol listing (see Section 7.3 and Section 7.4.1), all symbols in the program (variable names, functions and subroutines) are listed in alphabetical order with their type and other information. A formal description of all symbol types can be found in Section 8.3. The 15 symbol types can be classified in four main groups:

**User-defined variables**. All variables defined by means of **input** statements and **calculation** statements have names given to them by the user. Some restrictions apply to the choice of variable names (see Section 3.1.5 and Section 8.2). The use of an illegal or reserved name always leads to an error message, however (see Section 3.1.5 for an example). It is therefore not necessary to know all reserved names by heart.

The variable types in this group are

| | |
|---|---|
| • FST Parameter, | defined by a PARAMETER statement (Section 3.2.3.1). |
| • Initial constant, | defined by an INCON statement (Section 3.2.3.2). |
| • Constant, | defined by a CONSTANT statement (Section 3.2.3.3). |
| • Interpolation function, | defined by a FUNCTION statement (Section 3.2.3.4). |
| • Array Size Variable, | defined by an ARRAY_SIZE statement (Section 3.2.3.5). |
| • Calculated variable, | defined by an assignment or subroutine call (Section 3.2.6). |

**Variables with prescribed names**. All variables defined by means of **simulation control** statements have prescribed names. Many of these variables have default values and need not to be defined in each FST program.

The variable types in this group are

| | |
|---|---|
| • TIMER variable, | defined by a TIMER statement (Section 3.2.4.3). |
| • WEATHER control variable, | defined by a WEATHER statement (Section 3.2.4.4). |
| • TRANSLATION_FSE variable, | defined by a TRANSLATION_FSE statement (Section 3.2.4.2). |
| • TRANSLATION_GENERAL variable, | defined by a TRANSLATION_GENERAL statement (Section 3.2.4.1). |

The sections mentioned introduce and explain the use of various control variables. In Section 8.3.2 a complete list of variable names is given per variable type.

**Use-only variables.** There are two types of variables that can be used in an FST program without being defined. These two variable types are

- Variables supplied by the driver. (Section 8.3.2.7).
- Weather and calendar data. These variables are made available by using a WEATHER statement (Section 3.2.4.4 and Section 8.3.2.8).

**Subroutines and functions**. Subroutines called from FST have user-defined names. The input/output structure of these subroutines is declared with a DEFINE_CALL statement. This statement is regarded by FST as the definition of a subroutine. Function names, however, cannot be defined by the user. In expressions one can use, however, all Fortran intrinsic functions and a number of FST intrinsic functions.

Hence, the symbol types in this group are

| | |
|---|---|
| • Called SUBROUTINE, | declared by a DEFINE_CALL statement, see Section 3.2.2.2 and Chapter 5. |
| • Fortran Intrinsic function, | see Section 8.6 for an overview. |
| • FST Intrinsic function, | see Section 3.4 on interpolation functions, Sections 4.3.3 and 4.3.4 on array functions and Table 8.2 on page 136 for an overview. |

### 3.1.5 Variable names in FST

Variable names in FST follow the conventions of Fortran 77. They have a maximum length of 6 characters, start with a letter and may further contain digits and underscores. Hence A1, B_3, DNA123 and XX_444 are valid variable names. The names 1AB, _BAT, ABCD123 are invalid. The same rules apply to the names of called subroutines. In future versions the maximum length of the variable names will probably be extended to 31 characters. The other rules will remain unchanged.

The choice of user-defined variable names is further restricted by a list of forbidden names. The most important forbidden names are simply the FST control variables. Their names cannot be used for other purposes. Other names are forbidden because they would lead to errors in the generated Fortran. Appendix B lists some more groups of forbidden names.

The use of a forbidden variable name always leads to an error message, however. Hence, there is no need to memorize them. For instance, if the variable name DELT is used as a calculated variable, the FST translator reports an error since DELT is a timer variable, which can only be defined by means of a TIMER statement. The name DELT has to be replaced then by another, non-reserved variable name.

## 3.2 Description of the FST statements

Using the classification of statements of Section 3.1.3, the statement types are briefly described. Formal details can be found in Chapter 8. Comment statements are not mentioned anymore. They may occur anywhere in and FST program and begin with an asterisk "*".

### 3.2.1 Section statements

The section statements DECLARATIONS, MODEL, INITIAL, DYNAMIC, TERMINAL, END, STOP and ENDJOB have been described already in Section 3.1.1 on the program structure. Each section statement consists of its keyword only. Section statements may occur only once. They are most useful in somewhat larger FST programs. In very small programs they can largely be omitted. The precise rules are given in Section 8.4.

### 3.2.2 Declaration statements

Declaration statements are needed only when the program makes use of array variables or when Fortran subroutines are called. The declaration statements form the DECLARATIONS section at the beginning of the FST program, which may only be preceded by one or more TITLE statements. There are two types of declaration statements: ARRAY statements and DEFINE_CALL statements.

#### 3.2.2.1 ARRAY

ARRAY statements declare one or more variables as an array. Array variables may later be defined and used as parameters, initial constants or calculated variables. Each array declaration sets the lower and upper bound of the array's subscript range. The upper bound is set relative to a so-called array size variable, which is defined later by means of an ARRAY_SIZE statement. For example:

```
DECLARATIONS
ARRAY A(1:N), B(1:N+1), C(2:N+1)
...
MODEL
* setting the actual array sizes
```

```
ARRAY_SIZE N=10
* a parameter array
PARAMETER A=0.0
* an initial constant array
INCON B(1)=0.0 ; B(2:N+1)=3.4
* a calculated array
C(2:N+1) = AP(1:N)
```

The three subscript ranges all depend on the value of the same array size variable N. This means that the arrays A, B and C belong to the same array family. Complete understanding of the above example requires reading Chapter 4, which is entirely devoted to the use of array variables.

#### 3.2.2.2 DEFINE_CALL

A DEFINE_CALL statement declares the input/output structure of a called subroutine. The name of a called subroutine can be chosen by the user. For example

```
DECLARATIONS
* the input/output structure is declared
DEFINE_CALL MYSUB (INPUT,OUTPUT)
...
MODEL
* the subroutine is called with parameter A as input and B as output
CALL MYSUB (A,B)
PARAMETER A=3.4
```

Note that "INPUT" and "OUTPUT" are reserved words which can only occur in a DEFINE_CALL statement. The subroutine MYSUB in the above example has one input and one output argument. The argument description "INPUT" means a real input variable. The argument description "OUTPUT" means a real output variable which is calculated by the subroutine.

In the MODEL section the subroutine is called with actual arguments A and B. With help of the DEFINE_CALL statement, the FST translator concludes that A, as an input argument, is just used and should be defined elsewhere in the program and that B, as output argument, cannot be defined a second time.

In connection with the use of array variables, FST recognizes a few other argument types. The complete list is given in Section 8.5.2.7. Many other details and examples are given in Chapter 5, which fully describes the use of Fortran subroutines in FST.

### 3.2.3 Input statements

Various types of model input can be specified by means of the input statements PARAMETER, INCON, CONSTANT, FUNCTION and ARRAY_SIZE. These statements begin with the FST keyword which gives the statement its name. The keyword is followed by one or more substatements in the form of assignments. An example is the PARAMETER statement

```
PARAMETER A=0.0; B=13.1; WIDTH=1.0
```

After the keyword PARAMETER , three variables are defined by means of three substatements. The substatements are separated with a semicolon ";". The PARAMETER statement above is equivalent to the following three PARAMETER statements:

```
PARAMETER      A =  0.0
PARAMETER      B = 13.1
PARAMETER WIDTH =   1.0
```

Hence, the various substatements can be given also in separate statements of the same type. All input statements take the same form as PARAMETER statements. An overview:

```
* input statements with user-defined variable names
PARAMETER A=0.0; B=13.1; WIDTH=1.0               <- three parameters
INCON IS=6.7 ; ZERO=0.0                          <- two INitial CONstants
CONSTANT PI=3.14159265
ARRAY_SIZE N=6 ; M=5                             <- must be integer numbers !!
FUNCTION FUNTB=1.0, 3.4,  2.0, 3.4, ...          <- statement continues on next line
               5.0, 4.0, 10.0, 4.0
```

PARAMETER and INCON statements may be used in combination with FST array variables. Some example statements below illustrate that possibility. Of course, the example statements are meant to be imitated. If things become more difficult, however, the precise rules can be found in the referenced sections of Chapter 4 and Chapter 8.

### 3.2.3.1 PARAMETER

Model parameters are defined by means of one or more PARAMETER statements. Model parameters are numerical values in the equations of a model. Examples are a relative growth rate, a diffusion coefficient, the gravitation constant, the solubility of a gas etc. Model parameters can be used in all calculation statements and in FINISH conditions. One or more PARAMETER statements can be anywhere in the MODEL section. A few examples are

```
DECLARATIONS
* array variable to be defined later as a parameter array
ARRAY AP(1:N)
...
MODEL
* scalar parameters are defined
PARAMETER G=9.81 ; MASS=12.1
PARAMETER  WIDTH=10.0
PARAMETER HEIGHT=20.0
* a parameter is used in a finish condition:
FINISH X>HEIGHT
* two parameters are used in this dynamic calculation
DYNAMIC
   ENERGY = MASS * G * X
* a parameter array is defined (see Section 8.5.2.17 for precise rules)
PARAMETER AP(1:4)=1.,2.,3.,4. ; AP(5:8)=0.0 ; AP(9:N)=1.0
```

Note that the parameter array AP is defined in a single statement. The definition is broken up into a three substatements with subscript ranges (1:4), (5:8) and (9:N). (In Section 8.5.2.17 the precise rules can be found). Section 4.3 describes calculations with array variables.

### 3.2.3.2 INCON

The initial value of a state variable may be defined by means of an INCON statement. One or more INCON statements can be anywhere in the MODEL section. A typical construction is

```
MODEL
* the initial constant BI is defined
INCON BI=20.0
INITIAL
...
DYNAMIC
*  state variable B with initial value BI and rate of change BR
   B  = INTGRL(BI,BR)
   BR = ....
```

```
...
END
```

The state variable B is calculated by means of an INTGRL function call. The initial value BI is the first argument of the INTGRL statement. The second argument is the rate of change, calculated somewhere in the program. The INCON variable may be used in several INTGRL statements in order to initialize several state variables at the same value.

Note that the initial value of a state (the first argument of an INTGRL statement) may also be a calculated variable. It should then be calculated in the INITIAL section (e.g. Listing 2.2 on page 12).

Except as initial value, INCON variables may be used throughout the FST program in the same way as model parameters. INCON variables may also be arrays. They are used then in combination with a state variable array. The definition of an INCON array is similar to the definition of a parameter array. An example is

```
DECLARATIONS
ARRAY A(1:N), AI(1:N), AR(1:N)
...
MODEL
* the initial constant array AI is defined
* see Section 8.5.2.17 for precise rules
INCON AI(1:4)=1.,2.,3.,4. ; AI(5:8)=0.0 ; AI(9:N)=1.0
INITIAL
...
DYNAMIC
*  state array A with initial value AI and rate of change AR
   A  = INTGRL(AI,AR)
   AR = ....
...
END
```

Note that the structure of this array example is similar to the structure of the non-array example above. The precise rules for INTGRL statements with state arrays are given in Section 8.7.3.

#### 3.2.3.3 CONSTANT

For defining one or more mathematical of physical constants, the CONSTANT statement can be used somewhere in the MODEL section. Constants can be used throughout the program in calculations and finish conditions. An example is

```
* two physical constants and a mathematical one.
CONSTANT PLANCK=6.62608E-34 ; PI=3.14159265 ; AVOGAD=6.0221367E+23
```

At first sight there is little difference between a constant and a parameter. The two types of variables are treated by FST in different ways, however. Contrary to a parameter, a constant cannot be redefined in a rerun section and arrays of constants do not exist. Therefore, the CONSTANT statement is most suitable for mathematical constants like $\pi$, for basic physical constants as the speed of light and for conversion constants between different units.

#### 3.2.3.4 FUNCTION

A function can be specified in FST as a series of (x,y) points. The series of points gets a name and is specified with a FUNCTION statement anywhere in the MODEL section. Functions in this way are also called "interpolation functions" in this manual. For instance,

```
FUNCTION FUNTB = 0.0, 3.9, ...  <- statement continuation is used
                 1.0, 4.9, ...     to get a neat list of (x,y) pairs
                10.0, 7.8
```

Intermediate function values can be estimated by means of two FST intrinsic functions. The FST intrinsic function AFGEN (Arbitrary Function GENerator) uses linear interpolation. The FST intrinsic function CSPLIN uses Cubic SPLINe interpolation (Press *et al.*, 1986). The following example statements show that interpolated function values can be used in expressions.

```
...
B = 3.0*AFGEN (FUNTB,5.6) + 4.5*AFGEN (FUNTB,SIN(A**2)+1.0)
C = 3.0*CSPLIN(FUNTB,5.6) + 4.5*CSPLIN(FUNTB,SIN(A**2)+1.0)
```

The first argument of AFGEN and CSPLIN is the name of the interpolation FUNCTION. The second argument is the value of the "x" variable for which the function value has to be estimated. The second argument may also be an expression itself (e.g. `SIN(A**2)+1.0`).

Note that a function is not an FST array, although it contains a series of values. Interpolation FUNCTIONs can only be used as the first argument of AFGEN and CSPLIN calls. Interpolation FUNCTIONs can be redefined in a rerun section, simply by means of another FUNCTION statement. In Section 3.4 the interpolation of FUNCTIONs is discussed in greater detail.

#### 3.2.3.5 ARRAY_SIZE

Array size variables determine the actual length of arrays in FST (see the ARRAY statement). All array size variables have to be given a value by means of one or more ARRAY_SIZE statements somewhere in the MODEL section. An example is

```
DECLARATIONS
ARRAY A(1:N)                    <- the array size variable N is used
MODEL
...
ARRAY_SIZE N=60                 <- and here N is defined
```

Note that array size variables are integer variables.

### 3.2.4 Simulation control statements

Everything which needs to be controlled and which is not input to the model itself, is regarded as simulation control. The simulation control statements are TRANSLATION_GENERAL, TRANSLATION_FSE, TIMER, WEATHER and FINISH. With the exception of FINISH, all these statements use the same form as the input statements described in the previous Section 3.2.3. Some examples are

```
* simulation control statements (prescribed variable names)
TIMER  STTIME=0.0; FINTIM=10.0; DELT=0.1; PRDEL=1.0
WEATHER ISTN=2 ; WTRDIR='C:\SYS\WEATHER\'
TRANSLATION_GENERAL DRIVER='RKDRIV' ; TRACE=2 ; EPS=1.E-5
TRANSLATION_FSE IOBSD=1965,12
```

- The similarity to input statements refers to the form only, however. A few formal differences are:
- Only prescribed variable names can be used. For each type of control statement there is a list of variable names which may be defined with that statement type. The complete lists of control variables can be found in Section 8.3.2. In the sections below the most widely used variables are introduced.
- Arrays of control variables do not exist.
- TRANSLATION_FSE and TRANSLATION_GENERAL statements exclude each other.

Finally, the FINISH statement can take two forms:

`FINISH` expression < expression
or
`FINISH` expression > expression

### 3.2.4.1 TRANSLATION_GENERAL

The presence of a TRANSLATION_GENERAL statement causes the generation of a so-called GENERAL Fortran module which can be used in combination with a "general driver" from the DRIVERS library. There are two general drivers available, EUDRIV (EUler DRIVer) and RKDRIV (Runge-Kutta DRIVer). They are called "general" because they can handle a broader class of models than the more specialized FSE driver (see Section 3.2.4.2).

After the TRANSLATION_GENERAL keyword the variables DRIVER, TRACE, DELMAX and EPS may be defined. These variables represent settings for the integration procedure. For example:

```
TRANSLATION_GENERAL DRIVER='RKDRIV';TRACE=2; DELMAX=1.0; EPS=1.0E-4
```

By the value of **DRIVER** a driver is selected, either EUDRIV or RKDRIV. RKDRIV is the preferred driver for continuous equations. It uses the fourth order Runge-Kutta integration procedure with accuracy control which is described in Press *et al.* (1986). The method works by comparing the result of two half time steps with the result of a single full step. If the accuracy is not satisfactory (see the description of EPS below), the time step is reduced. The timer variable DELT (Section 3.2.4.3) is used as a first guess of the integration step at the start of the simulation.

EUDRIV uses Euler or rectangular integration with a fixed time step. Usually that fixed time step is equal to DELT, but if DELT does not fit an integer number of times in the output interval PRDEL (Section 3.2.4.3), the time step is reduced. In case of blocky interpolation functions or otherwise discontinuous models, EUDRIV may be preferred above RKDRIV (see Section 2.2 for an example). The behaviour of the two drivers can be compared by means of a rerun on driver choice (write a rerun section with a TRANSLATION_GENERAL statement). In Chapter 9 the generated Fortran program and the function of the driver are further discussed.

With the integer variable **TRACE** one can set the level of logfile output (the default value is 0). Setting TRACE=2 is very useful for testing a model with RKDRIV debugging. It gives on screen (and on logfile) the number of time steps taken between successive output times. The meaning of other settings of TRACE can be found in Section 7.5.5.

**DELMAX** sets an upper limit to the time step (with default value FINTIM-STTIME). **EPS** is the accuracy parameter of the integration (with default value 1.0E-4). The control variables DELMAX and EPS are significant only in combination with the driver RKDRIV. If they are left undefined the default values are used. DELMAX and EPS may be printed (with PRINT) or used in calculations. Reruns on the value of EPS give the possibility of finding a suitable accuracy setting (see Chapter 9 for details on the use of EPS by RKDRIV).

### 3.2.4.2 TRANSLATION_FSE

The presence of a TRANSLATION_FSE statement causes the generation of a Fortran module which can be called by the FSE driver from the DRIVERS library. Such a Fortran module is also called "FSE module". FSE stands for "Fortran Simulation Environment". Hence, the FST translator generates an FSE module if the following statement is present somewhere in the MODEL section

```
TRANSLATION_FSE
```

The FSE driver and the structure of an FSE model have been developed by van Kraalingen (1995). The FSE driver uses Euler integration, like the general driver EUDRIV. FSE was originally developed for crop growth models and is especially useful for the simulation of processes on a daily basis (although time steps smaller than a day can also be used). The use of year numbers and day numbers and weather data is fully integrated in the FSE driver. This is illustrated by the only TRANSLATION_FSE control variable, **IOBSD**, which is defined in the following example

```
TRANSLATION_FSE IOBSD=1985,12,  1995,13,  1996,14
```

The combinations of year and day number are used by the FSE driver for producing additional output for the specified days. The variable name comes from "OBServation Days". The additional model output at these days can be compared with values measured on these days. The variable IOBSD cannot be used in calculations.

The difference between the GENERAL and FSE translation modes is discussed in Section 3.3.

#### 3.2.4.3 TIMER
One or more TIMER statements are used in the first place to control simulation time. In the examples of Chapter 2 this has been shown already. The TIMER statement occurring in most FST models looks like

```
TIMER STTIME=0.0 ; FINTIM=45.0 ; DELT=0.5 ; PRDEL=3.0
```

This TIMER statement defines **STTIME** (the start time), **FINTIM** (the finish time), **DELT** (the time step of integration) and **PRDEL** (the output or PRint interval). The timer variables STTIME, FINTIM and DELT have to be defined in all FST models. The definition of other timer variables, like PRDEL, is optional. If PRDEL is left undefined, the model will produce initial and terminal output only.

The TIMER statement is also used to set some control variables which, strictly speaking, have nothing to do with time control. The timer variable **IPFORM** (Integer Print FORMat) controls the form of the output file RES.DAT. IPFORM is an integer variable which can take the values 4,5 or 6:

4 = spaces between columns.
5 = TAB's between columns (good as spreadsheet input).
6 = two column output.

The default value for IPFORM is 4. An example of a TIMER statement defining IPFORM among others is

```
* some timer variables which are used less often
TIMER IPFORM=5 ; COPINF='Y' ; RGINIT='Y' ; RGSEED=123
```

The timer variable **COPINF** (COPy INput Files) is a character variable with values 'Y' or 'N'. With COPINF='Y' the generated Fortran input files are copied to the output file RES.DAT after completing all reruns. The default value of COPINF is 'N'.

The timer variables **RGINIT** and **RGSEED** are used to control the generation of random numbers, a topic which is separately discussed in Section 3.7.

#### 3.2.4.4 WEATHER
One or more WEATHER statements are used to control the weather data used by the translated simulation model. There are four weather control variables. **WTRDIR** is a character variable which selects the weather data directory. **CNTR** is a character variable which selects the country. **ISTN** is the integer station number and **IYEAR** is the integer year. An example is:

```
WEATHER WTRDIR = 'C:\SYS\WEATHER\'
WEATHER CNTR='NLD' ; ISTN=2 ; IYEAR=1988
```

These two WEATHER statements instruct FST to look for a weather data file with the name `'C:\SYS\WEATHER\NLD2.988'`. After selecting weather data in this way, the following weather and calendar data become available for use in model calculations:

| Variable name | Meaning | Unit |
|---|---|---|
| RDD | Total daily global short-wave radiation | $J\ m^{-2}\ d^{-1}$ |
| TMMN | Minimum air temperature | °Celcius |
| TMMX | Maximum air temperature | °Celcius |
| VP | Vapor pressure at 9 a.m. | kPa |
| WN | Average wind speed | $m\ s^{-1}$ |
| RAIN | Total daily rainfall | $mm\ d^{-1}$ |
| LAT | Latitude of the site | degrees |
| LONG | Longitude of the site | degrees |
| ELEV | Elevation of the site above see level | m |
| DOY | Day Of Year in the form of a real number | d |
| YEAR | Current year number as 1988.0 | y |

Section 2.2 gives an example of the use of the daily minimum air temperature TMMN and the daily maximum air temperature TMMX in an FST model. All weather and calendar data are real variables which may be used in expressions and as arguments of function and subroutine calls. They cannot be (re)defined, however, and the variable names cannot be used for other purposes, even if no weather data are selected.

The use of weather data implies that the unit of time is day. The weather control variable IYEAR is the start year of the simulation and the timer variable STTIME is the start time between 1.0 (January 1, 00:00:00) and 365.999... (December, 31, 23:59:59.99...). During the simulation, TIME may exceed 365.9999... (or 366.999.. in a leap year), which leads to an increase of the current year number YEAR and the use of the next year of weather data (see also the remarks in Section 6.1.4).

The calendar variable DOY jumps back to 1.0 at the beginning of a new year. DOY is the day number ranging from 1.0 to 365.0 (366.0 in a leap year, changing in steps of 1.0) and may be used in combination with interpolation FUNCTIONs which describe seasonal data.

#### 3.2.4.5 FINISH

The FINISH statement does not define control variables, as do the other simulation control statements. The FINISH statement defines a finish condition. Valid FINISH statements are, for instance,

```
FINISH A < 3.4               <- any user defined variable
FINISH TMMX > 40.0           <- maximum daily temperature
FINISH A*B < SQRT(PP)+1.0    <- a relation between two expressions
FINISH NINT(T) > 56          <- an integer condition
```

If during simulation, one or more of the finish conditions becomes valid, the simulation stops, even if FINTIM is not yet reached. The simulation is halted in an orderly way, however. At the reached finish time dynamic output is produced (see Section 3.2.5), then a possibly present TERMINAL section is executed.

A FINISH condition often represents the regular end of the simulated process. In such a case the value of FINTIM is set very high and just acts as a safeguard against an infinite simulation.

## 3.2.5 Output statements

The most important output statement is the PRINT statement which is used to specify the output variables. The OUTPUT statement leads to printplots (plots in the form of a text file) and is almost obsolete. The TITLE statement specifies the text which is included in the header of the output tables in RES.DAT. The structure of the output statements is most easily described by means of the following example

```
TITLE Example Model            <- Text describing the model and appearing in output file
PRINT STTIME, A, B             <- Comma-separated items
OUTPUT A, C                    <- Comma-separated items
```

### 3.2.5.1 PRINT

The PRINT statements in the MODEL section specify the output variables of the model. All variables from the model and many simulation control variables can act as output variable. Valid PRINT statements are for instance

```
PRINT X, Y1, Y2                <- user variables
PRINT STTIME, TMMX             <- a timer variable and weather data
```

Array variables can be printed as a whole or certain subscript ranges can be mentioned. For instance,

```
DECLARATIONS
ARRAY A(1:N), B(1:N)
...
MODEL
PRINT A(3:5), B, N             <- a range of A, the complete array B and the array size N
...
```

The variables listed in the PRINT statements appear in two output files. The first one is an ordinary, readable text file named RES.DAT which contains tables with output values. The other output file is a machine readable file RES.BIN (a so-called unformatted file). For details on these output files see also Section 7.4.

The FST translator distinguishes between initial, dynamic and terminal output. The initial output consists of all output variables which get their values in the initial phase of the model run. This includes model parameters, simulation control variables or variables calculated in the INITIAL section. Such variables form the initial output, independent of the position of the PRINT statements in which they are listed.

The dynamic output consists of the output variables which are defined in the DYNAMIC section of the program. This includes state variables, rates of change, all other dynamic variables and also the weather and calendar data. Finally, the terminal output consists of variables calculated in the TERMINAL section of the model.

Note that a single PRINT statement can contain variables from all three groups and, once more, that the position of the PRINT statement(s) in the MODEL section does affect the output files. It is good programming practice, however, to list different types of output variables in different PRINT statements, like in the following growth model

```
MODEL
INITIAL
```

```
  IX = 2.0
PRINT IX                        <- initial output
DYNAMIC
  X  = INTGRL(IX,RX)
  RX = A * X
PARAMETER A=0.1
PRINT A                         <- initial output, because A is known from the beginning
PRINT X, RX                     <- dynamic output
TIMER STTIME=0.0 ; FINTIM=10.0 ; DELT=0.1
TIMER PRDEL=0.5
TRANSLATION_GENERAL
END
```

All values appearing in the output file are associated with a certain value of the simulation TIME. Initial and terminal output are generated only once and are associated with the start and finish time of the simulation. The dynamic output variables are sent to output many times, for increasing values of TIME.

If you are only interested in the final result of a simulation run, the size of the output tables can be reduced by not defining the timer variable PRDEL. The dynamic output is then generated only two times, at the start and at the end of the simulation run.

A more elegant way is the use of terminal output variables. For instance, from reruns with the above growth model, we want a graph of the final value of X as a function of parameter A. The following TERMINAL section and rerun sections can be added then to the model

```
...
TERMINAL
* the terminal variable A1 is a copy of parameter A
  A1 = A
* the terminal variable XFINAL is the final value of X
  XFINAL = X
PRINT A1, XFINAL                <- terminal output
...
END
PARAMETER A=0.2                 <- rerun on parameter A
END
PARAMETER A=0.3
...
```

After deleting the other PRINT statements, the model produces only terminal output consisting of A1 (a copy of A) and XFINAL. The advantage of printing a copy of A is that both A1 and XFINAL are then printed with the same value of TIME (PRINT A always leads to initial output). In Section 6.6 terminal output is used to compare the analytical and numerical solution of a model.

#### 3.2.5.2 OUTPUT

This statement is almost obsolete. For variables listed in one or more OUTPUT statements printplots are produced. A printplot is a graph in the form of an ordinary text file which does not require a graphical printer. Each OUTPUT statement leads to a separate printplot. Array variables cannot occur in OUTPUT statements.

The use of OUTPUT is superfluous in case there is any other possibility of producing graphs. The tables of the output file RES.DAT (containing all "PRINT" variables) can be imported in any spreadsheet program, for instance, which is used then to produce graphs. You may need to set the timer variable IPFORM to 5 (see Section 3.2.4.3). An easy way of inspecting results is the use of the graphical output utility TTSELECT which is mentioned briefly in Section 7.2. Graphs can be made then for all variables listed in the PRINT statements.

#### 3.2.5.3 TITLE

The text behind the word TITLE appears in the header of each table in the output file RES.DAT. TITLE statements usually contain a brief description of the model. They may occur anywhere before the END statement at the end of the MODEL section. An example is

```
TITLE  ======= Competition Model =======
TITLE  == based on light interception ==
```

Note that the FST program file itself can also be documented by means of comment statements which begin with an asterisk "*".

## 3.2.6 Calculation statements

In calculation statements model variables are calculated. There are two types of calculation statements. The first type is the ordinary assignment like A=B+SIN(C). The second type of calculation statement is a Fortran subroutine call. With both types of statements, model variables are calculated from other model variables and from model input variables, weather data etc.

#### 3.2.6.1 Assignment

An assignment defines a calculated variable in terms of other variables. The statement A=B+SIN(C) defines the variable A, which apparently depends on B and C. The right part of the assignment is a Fortran style expression which may contain calls to Fortran intrinsic functions (like the sine function SIN, see Table 8.1 on page 134) and FST intrinsic functions (see Table 8.2 on page 136 in Section 8.7) Examples of valid assignments are:

```
A = 2.0
A = 3.0 * (B + C)
A = 2.4 * INSW (B, 1.0+SIN(C), 1.0-SIN(C)) + MAX(D**2, AFGEN(RTB,RAIN))
```

The third assignment shows that function calls may be nested: the arguments of the FST intrinsic function INSW contain calls to SIN, a Fortran intrinsic function. The evaluation of the right hand side takes place in the usual way: subexpressions in parentheses and function calls have priority, then exponentiation, multiplication/division and finally addition/subtraction.

Although assignments in FST look very much like Fortran statements, there are minor differences:

- In FST, the use of integer constants in a real expression is impossible in FST except as exponents. Writing 2*A, for instance, leads to an error message. One has to add a decimal point in such situations. Integer variables like array size variables can be used in expressions after converting them with the Fortran intrinsic function REAL.
- In Fortran there are a few intrinsic functions which accept more than one argument type. In FST these functions accept only real arguments, except the modulo function MOD, which is limited to integer arguments in FST (cf. Section 4.3.4.4). The modulo function for real arguments in FST is AMOD, which exists in Fortran as well.

If the calculated variable is an array, its elements are calculated by means of a consecutive list of substatements separated by a semicolon ";". These compound assignments are introduced in Section 4.3 and the precise rules are given in Section 8.5.2.1.

#### 3.2.6.2 Call to Fortran Subroutine

The second type of calculation statement is a call to a Fortran subroutine. Subroutine calls are extensively described in Chapter 5. Here only a few remarks are made. The form of a subroutine call is the same as in Fortran. Within a subroutine complicated calculations may be carried out, but from the point of view of the FST translator a subroutine call is just a single calculation statement. The only difference with an assignment is that several variables may be calculated simultaneously.

The FST translator has to know the difference between input and output variables of the subroutine call. This information is essential because the computational order of the calculation statements depends on it. Therefore, the input/output structure of each subroutine has to be declared in the DECLARATIONS section by means of a DEFINE_CALL statement. See Chapter 5.

## 3.3 State variables and integration method

In FST, state variables are calculated by means of the INTGRL function. Unlike other functions, the INTGRL function cannot be used in expressions. A state variable S is defined as

```
S = INTGRL(SI,SR)
```

where SI is the initial value of S and SR the rate of change. The initial value is often defined be defined by means of an INCON statement but may also be a variable calculated in the INITIAL section.

The rate of change is usually calculated in the DYNAMIC section of the MODEL. Also weather data can be integrated and, mainly for testing purposes, model parameters can act as rate of change. The use of the INTGRL function for arrays is described in Section 8.7.3.

The INTGRL function call above corresponds to the formula

$$S = S_i + \int_{\mathrm{STTIME}}^{\mathrm{TIME}} S_r \, dt$$

For integrating the rate(s) of change over time, two different numerical methods can be used. In TRANSLATION_GENERAL mode, Euler integration and a fourth order Runge-Kutta method are available (see Section 3.2.4.1). In TRANSLATION_FSE mode, only Euler integration is available.

Running an FST model, the difference between the FSE translation mode and the GENERAL mode with driver EUDRIV is negligible. The minor numerical difference is that the FSE driver adapts the specified DELT in order to fit an integer number of times in one day, and EUDRIV adapts the specified DELT in order to fit an integer number of times in output interval PRDEL (see Section 3.2.4.3 for these timer variables).

The GENERAL and FSE translation modes, however, strongly differ with respect to the generated Fortran modules. This implies that the selected mode becomes important if the generated Fortran is extended "by hand" or has to be combined with existing programs.

The structure of FSE has been documented by van Kraalingen (1995). FSE has originally been developed for crop growth simulation (van Kraalingen, 1995). The unit of time used is day and it requires the specification of weather data by means of WEATHER statements. An FSE module has a simple internal structure, which is more easily adapted "by hand" than a GENERAL module. It can also be combined with other (FST-generated) FSE modules. The disadvantage of FSE is that the Euler integration method is build into the simulation modules.

The GENERAL translation mode is completely independent of the calendar and the unit of time is arbitrary (unless WEATHER is used). A (generated) GENERAL module can be combined with an external integration procedure like the two available in EUDRIV and RKDRIV. The price for that flexibility is a more complicated internal structure which is documented in Chapter 9.

## 3.4 Interpolation FUNCTIONs

An interpolation function is defined by means of a FUNCTION statement (Section 3.2.3.4). In this section the actual interpolation, with help of AFGEN or CSPLIN, is discussed further.

As an example the function ATB is used, defined with 6 function points:

```
FUNCTION ATB = 0.0, 0.0, ...      <- statement continuation is used
               2.0, 1.7, ...         to get a neat list of (x,y) pairs
               4.0, 2.3, ...
               6.0, 2.6, ...
               8.0, 2.7, ...
              10.0, 2.7
```

In Figure 3.2 the function points are plotted. The points can be measured values, guessed values or they may be calculated values of a complicated function. In any case, the use of an interpolation function requires a way to find intermediate function values. Two methods have been built into FST: linear interpolation and cubic spline interpolation.

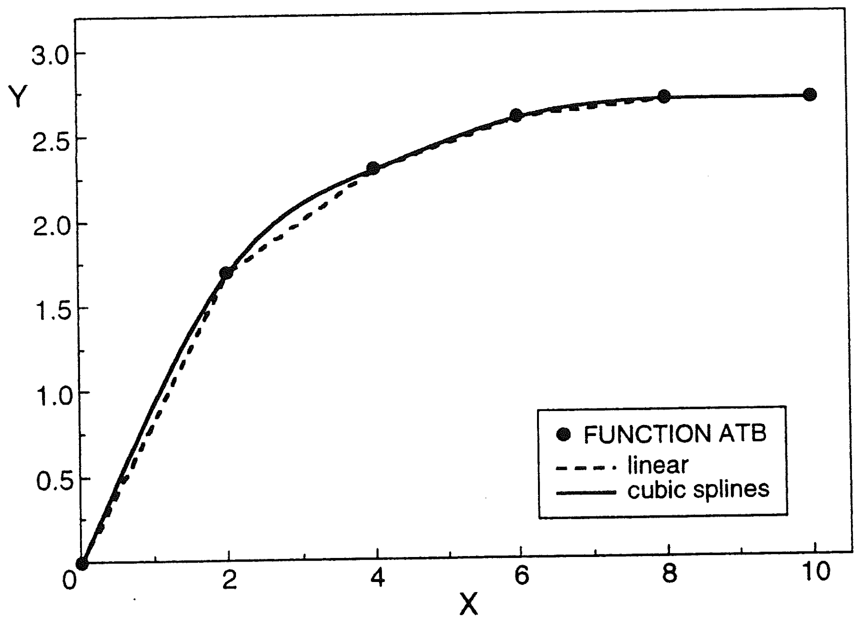


Figure 3.2 Linear and cubic spline interpolation of FUNCTION ATB.

### 3.4.1 Linear interpolation with AFGEN

Function values of the function ATB are calculated as AFGEN(ATB,X) in which the second argument X may be also a real number or an expression. The dashed line in Figure 3.2 shows that straight lines are drawn between the function points given in the FUNCTION statement. If the value of the second argument lies outside the X-range of the function ATB, a warning is given during execution of the program.

An advantage of linear interpolation is that it is a very robust method. The numerical algorithm does exactly what one expects it to do: it "draws" straight lines between function points. In many situations this is fully appropriate. For instance, if little is known about the precise shape of an empirical function it may be roughly described as a few connected straight lines. If many points of a function are known with sufficient accuracy, however, one will be tempted to draw a neat curve through the points. In such situations the use of CSPLIN *may* lead to better results.

### 3.4.2 Natural Cubic Spline interpolation with CSPLIN

Function values of an interpolation function, say ATB, are calculated as CSPLIN(ATB,X) in which the second argument X may be also a real number or an expression. The graph in Figure 3.2 on page 35 shows that a neat curve is drawn through the function points given in the FUNCTION statement. The use of natural cubic splines (Press *et al.* 1986) guarantees that the first and second order derivatives of the function are also continuous. The second derivatives at the begin and at the end of the curve, however, are assumed to be zero. This means there is no curvature at the first and at the last point.

CSPLIN always draws neat curves through the function points. Nevertheless, the method may fail. An example of failure is given in Figure 6.2 on page 69. Between the two last function points the interpolated function behaves wildly. Hence, before CSPLIN is used in model calculations, one has to verify its behaviour. This can be done, for instance, with the FST program for plotting interpolation functions which is given in Section 6.1.1. Special care is required if the X values of the FUNCTION are not equidistant or if the function is curved at the beginning or end of its range. In case of failure one can sometimes repair the interpolation by leaving out slightly deviating function points (or by adding new ones, if available). Otherwise one has to use the more robust linear interpolation.

## 3.5 Sorting calculations

The initial, dynamic and terminal calculations are automatically sorted by FST in order to find a computational order of the statements. This sorting capability implies that the user may start to write down the most important relations of the model and then work downward by specifying the yet undefined variables with new expressions or by defining them as a model parameter. An example of model equations written in such a way is

```
...
* position is the time integral of its rate of change
X = INTGRL(IX,RX)
* the rate of change of the position is the velocity, by definition
RX = V
* the velocity itself is a state variable changing with the acceleration
V = INTGRL(IV,A)        <- note that the use of V instead of RX in the first INTGRL statement would
                           lead to an error. For technical reasons it is impossible in FST to use a
                           state variable as a rate of change. Therefore RX is used, a copy of V.
* Newton's law: Force = Mass * Acceleration
A = FORCE/MASS
* the forces to describe: gravity and friction
FORCE = FDOWN - FRICT
* gravity and friction specified
FDOWN = G*MASS
FRICT = A*V
PARAMETER MASS = 10.0 ; A=0.1
CONSTANT G=9.81
INCON IX=0.0 ; IV=0.0
...
```

The actual calculations are carried out in about the reverse order. FST is only able to sort the calculations, however, if all variables are defined once and only once. If a variable is defined more then once, FST cannot decide in which order the calculations have to be carried out. Therefore, an error message is given.

A computational order requires that each variable is calculated before it is used. Sometimes, although all variables are calculated only once, it is impossible to sort the calculations as a result of a loop in their interdependency. An example of such a loop is: A depends on B, B depends on C and C depends on A. The FST translator finds such loops and gives (in case of the dynamic calculations) the following error message:

```
***           The DYNAMIC calculations cannot be brought in a
              computational order since there is a loop in
              their interdependence. The statements containing
              the loop will be marked with "x" in the listing
              file. Note that a variable depending on itself
              also forms a loop. (%GLOBAL-SORLOOPD).
```

The simplest type of dependency loop is a statement like "A=A+B". This statement implies that A depends on A which is a loop. In Fortran, statements like this are often used, but A would have to be defined earlier as well which is impossible in FST. Usually, the problem can be solved by defining a help variable.

Note that subroutine calls, being calculation statements, are also sorted. In order to find a proper position of the calls among the other calculations, the FST translator has to know which variables are calculated in the call and which ones are just used. Hence, statement sorting is the reason behind the DEFINE_CALL declaration statements in which the Input/Output structure of each called subroutine is declared. Therefore, a wrong DEFINE_CALL statement may lead to a sorting error.

## 3.6 Reruns

Reruns form a very useful feature of FST. After terminating the MODEL section of the program with an END statement, one or more rerun sections may be added. In a rerun section simulation control and/or input variables are redefined by means of another simulation control or input statement. Hence, a timer variable should be redefined by means of another TIMER statement, a model parameter by means of another PARAMETER statement etc. All simulation control statements can be used except FINISH and all input statements can be used except CONSTANT and ARRAY_SIZE.

The program in Listing 2.3 on page 15 gives an example. The value of the initial constants IX1 and IX2 is redefined again and again in order to get a lot of model runs with the state variables starting at different values. Another widely used application of reruns is the use of a series of different weather stations by changing country code and/or station number with a WEATHER statement in each rerun section. Note that the weather data directory WTRDIR cannot be redefined.

It is important to realize that a variable, after being changed in a certain rerun value, retains its new value in all subsequent reruns, until it is changed again. An example is

```
MODEL
PARAMETER A=1.0
INCON XI=34.0
...
END
* second run ; first rerun
```

```
PARAMETER A=2.0
END
* third run ; second rerun
INCON XI=40.0
END
```

In the first rerun section only parameter A is redefined. The second rerun will be done with A=2.0 and XI=40.0.

Variables defined by means of calculation statements cannot be redefined in a rerun section. The reason is that the calculation statements form the structure of the model which is translated by FST into a Fortran program. This generated Fortran program is the same for all reruns and only the values of input and control variables may vary.

In Section 6.6, an example program shows how reruns can be combined with terminal output variables in order to perform a sensitivity analysis of the model. Another interesting application is the redefinition of the variable DRIVER in a TRANSLATION_GENERAL statement (see Section 3.2.4.1). This leads to a model run with another integration method being used.

## 3.7 Random number generation

There are two FST intrinsic functions for the generation of a (pseudo) random numbers. The function **RGUNIF** (Random Generator UNIForm) returns uniformly distributed random numbers. Each time it is called it returns a newly generated number. The function **RGNORM** (Random Generator NORMal) returns random numbers with a normal distribution. Both functions have two arguments which are described in Table 8.2 on page 136.

In the following example a rate of change is calculated as a random number, uniformly distributed between the parameters RMIN and RMAX:

```
DYNAMIC
A = INTGRL (AI,AR)
TRANSLATION_GENERAL DRIVER='EUDRIV'         <- the model is not continuous
AR = RGUNIF(RMIN,RMAX)                      <- a random number between RMIN and RMAX
PARAMETER RMIN=2.3 ; RMAX=2.4
INCON AI=1.0
```

Each time the model runs, the same sequence of random numbers is generated by the function RGUNIF. Hence, the model runs are exactly repeatable which, in general, is important for finding errors. Also if reruns are specified, the behaviour of the random number generator does not change. For instance, the impact of AI on the model results can be investigated by changing the value of AI in reruns. The random number sequence remains the same.

Technically, this means that the random number generators are re-initialized at the beginning of each rerun and that the same 'seed' is used again and again. This behaviour represents the default behaviour of the random number generators in FST. Sometimes this default behaviour has to be changed. One wishes to use different sequences of random numbers in reruns, for instance, in order to include the random process in a sensitivity analysis. In such situations one has to define values for the control variables RGSEED and/or RGINIT.

The first control variable **RGSEED** (Random Generator SEED) is an integer timer variable which is used as the "seed" of the generation process. If RGSEED is not defined, the seed takes its default value 1. Each non-zero integer value results in a different and repeatable sequence of random numbers. A special choice is

```
TIMER RGSEED=0                <-  the seed is derived from system clock at runtime
PRINT RGACTS                  <-  the actual seed used is printed to be able to repeat specific runs
```

This leads to a seed being derived from the computer's system clock during model execution. The actual seed used is stored in the integer variable **RGACTS** (ACTS from ACTual Seed). This variable is supplied by FST and can be printed or used otherwise as an integer variable.

The use of a new seed for every new run means that a different random sequence is used for each model run. Also for equal parameter settings the results of the model runs will not be identical. The differences reflect the influence of the random process on the model results.

The second control variable is **RGINIT**. This timer variable controls whether or not the random number generators are re-initialized at the beginning of each rerun. RGINIT can have the values 'Y' or 'N' (Yes or No). The (default) value 'Y' causes the re-initialization of the generators at the start of each rerun using the current value of RGSEED. RGINIT='N' suppresses re-initialization.

Suppressing re-initialization means that the random sequence of the previous run is continued. This has the disadvantage that an individual model run cannot be repeated since the random sequence used cannot be reconstructed without repeating also the previous runs. Therefore, in most situations, re-initialization with a new seed is the preferred method for including random effects in rerun results.

Finally, it should be noted that RGUNIF and RGNORM are not fully independent of each other. The function RGNORM internally calls RGUNIF and, in fact, only RGUNIF needs to be initialized. Hence, a single pseudo random process underlies the returned values of both functions. This implies that adding a call to RGNORM to a program calling already RGUNIF will influence the values returned by RGUNIF.

# Chapter 4

# Array variables in FST

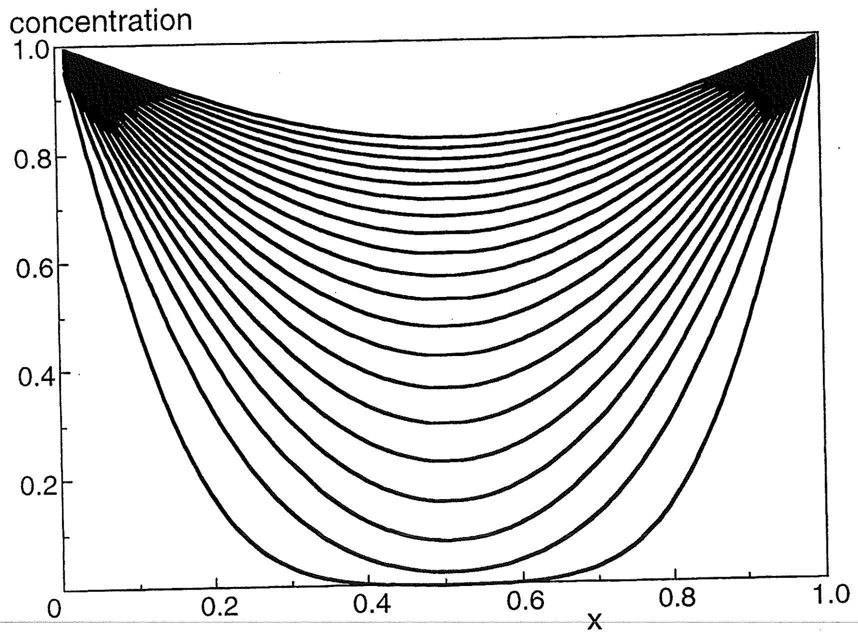


*Figure 4.1 on the title page of this chapter.* The curves are concentration profiles at different values of TIME calculated with the diffusion model in Listing 4.1 on page 44. The model describes diffusion into a plane sheet which is initially at zero concentration. As time increases the concentration approaches the constant outer concentration.

# 4 Array variables in FST

## 4.1 Introduction

An array variable contains a number of values instead of just a single value. Each value is contained in an array element. The values, and array elements, are numbered. If the numbers are just 1,2,3,..N, the array is one-dimensional. If the element numbers are (1,1), (2,1), (3,1), .. (N,M), the array is two-dimensional, etc. Only one-dimensional array variables can be used in FST.

Array variables in FST are treated almost like ordinary, scalar variables. They are defined in one statement and can be referenced in other statements. An array can be used as a calculated variable, as a model parameter or as an initial constant. This abstraction enables the FST translator to verify the completeness of the program, and to find a computational order for scalar and array variables simultaneously. The use of array variables does not change the structure of an FST model.

The example program in Section 4.2 is meant to give an impression of the possibilities. Section 4.3 gives a more systematic treatment of array calculations in FST. Section 4.4 deals with PARAMETER and INCON arrays. The entire chapter has an explanatory character. For a formal description of array variables in combination with the various FST statements Chapter 8 should be consulted.

## 4.2 Example of the use of array variables

Listing 4.1 gives a short FST program describing diffusion into a plane sheet. The model is similar to diffusion models given in Leffelaar (1993). Initially, the concentration in the sheet is zero. From time 0.0 on, the sheet is exposed to a constant concentration CS at its two surfaces. Diffusive transport takes place until the equilibrium concentration CS is reached everywhere. In this section, the statements in Listing 4.1 are discussed with emphasis on array use. The reader is assumed to be familiar with the principle of a diffusion process.

Figure 4.1 on the title page of this chapter shows how the concentration profile changes with time. The figure has been constructed from the output of the FST program in Listing 4.1.

The diffusion problem is solved by distinguishing N layers. Then there is an array of N concentrations C, there are N amounts H, N initial amounts IH, N rates of change NFLOW (net flow into layer), and N+1 fluxes FLUX through the N+1 surfaces. Further, the centers of the N layers are calculated as an array X.

On top of the program all array variables are declared with the ARRAY statements 3 and 4. Each declaration consists of an array name followed by a subscript range in parentheses, e.g. FLUX(1:N+1). The lower boundary has to be an integer constant, the upper boundary is a so called array size variable, here N, to which an integer constant may be added. In Listing 4.1, all arrays make use of the same array size variable N. This implies that the arrays form an "array family", which expresses the fact that all arrays refer to properties of N layers.

All this is still independent of the actual number of layers N, which is set with the ARRAY_SIZE statement 8. Note that the value of N occurs only a single time in the entire program.

The constant concentration CS at the outer surface, the height of the plane sheet HEIGHT, its area AREA and its diffusion coefficient is D are all parameters of the model (statement 7).

Listing 4.1 An example of using arrays. The graph on the title page of Chapter 4 shows output of this program.

```
0001   TITLE   Diffusion into a two-sided plane sheet
0002   DECLARATIONS
       * all arrays are declared ; N layers and N+1 fluxes
0003   ARRAY FLUX(1:N+1), X(1:N), C(1:N)                 <-all arrays are declared in one or
0004   ARRAY H(1:N), IH(1:N), NFLOW(1:N)                 <-more array statements.
0005   MODEL
       *****
0006   INCON   IH=0.0                                    <-full range assignment in INCON
0007   PARAM   HEIGHT=1.0 ; D=0.01 ; AREA=1.0 ; CS = 1.0
       * the number of layers
0008   ARRAY_SIZE N=60                                   <-sets the size of all arrays
0009   INITIAL
       * height of compartment
0010   DEL = HEIGHT / REAL(N)
       * volume of compartment
0011   VOL = DEL * AREA
       * compartment centres
0012   X = (REAL(I) - 0.5) * DEL
0013   DYNAMIC
0014   H = INTGRL(IH,NFLOW)
0015   C = H / VOL
       * the fluxes through the boundaries
0016   FLUX(1)    = D * (CS - C(1)) / (0.5*DEL) ; ...    <-three substatements
       FLUX(2:N)  = D * (C(1:N-1) - C(2:N)) / DEL ; ...
       FLUX(N+1)  = D * (C(N) - CS) / (0.5*DEL)
       * the nett flow into the compartments
0017   NFLOW(1:N) = FLUX(1:N)*AREA - FLUX(2:N+1)*AREA
       * output and simulation control
0018   PRINT   X, C
0019   TIMER   STTIME = 0.0 ; FINTIM=20.0 ; DELT = 1.0 ; PRDEL=1.0 ; IPFORM=5
0020   TRANSLATION_GENERAL DRIVER='RKDRIV' ; TRACE=2
0021   END
0022   STOP
0023   ENDJOB
```

In statement 6, the array of initial amounts is defined. Since the FST translator "knows" which variables are arrays, statements like PARAM and INCON can be used to define array variables, which become then parameter arrays or initial constant arrays. In statement 6 all array elements are equal and the array IH is simply defined by IH=0.0.

In statement 10 and 11, the volume VOL and height DEL of the layers are calculated. The number of layers N can be used in expressions, but has to be converted to a real value by the function REAL.

In statement 12, the array variable X is calculated. The declaration statement 3 shows that the elements of X are numbered from 1 to N. Therefore, the integer variable I in the right hand part takes the values 1...N in successive calculations. The first element of X is calculated as (REAL(1)–0.5)*DEL, the second element as (REAL(2)–0.5)*DEL etc. The variable name I is reserved for this purpose and cannot be defined or used in any other way (see also Section 4.3.2).

The first two assignments in the DYNAMIC section look like ordinary statements from an FST program without arrays. The N initial amounts IH and the N rates of change NFLOW belong to the N amounts H, and the INTGRL statement looks the same as for ordinary, non-array variables. In the same way, statement 15 means C(1)=H(1)/VOL, C(2)=H(2)/VOL, ...... C(N)=H(N)/VOL. Since the translator knows already that C and H are arrays, there is no need for describing the relation between C and H for each layer separately. The concise form C=H/VOL is "expanded" by the translator (see Section 4.3.1.4).

The assignment in statement 16 is more complicated. The calculation of the surface fluxes differs from the calculation of the fluxes inside the sheet. Therefore the statement is divided into three substatements, separated by a semi-colon ";". Each substatement defines a part of the array FLUX. The fist part consists of the first element only, a surface flux. The second part of the array FLUX consists of all N-1 fluxes "inside" the sheet. The second substatement is a concise form for

```
FLUX(2)  = D*(C(1)-C(2))/DEL
FLUX(3)  = D*(C(2)-C(3))/DEL
...
FLUX(N)  = D*(C(N-1)-C(N))/DEL
```

The last substatement defines the last element of FLUX, again a surface flux.

In statement 17, the rate array NFLOW is calculated. Note that the array NFLOW has N elements and that the two ranges of FLUX in the expression, (1:N) and (2:N+1), also have N elements. This illustrates an important principle: all subscript ranges in a substatement need to be equally long. Array calculations are further discussed in Section 4.3 and precise rules are given in Section 8.5.2.1.

The rest of the program in Listing 4.1 looks probably familiar. The PRINT statement sends the initial array variable X and the dynamic array variable C to output. The value of the timer variable IPFORM makes the output file readable for a spreadsheet.

A few general remarks on Listing 4.1 can be made. Like simple, scalar variables, calculated array variables are defined with a single statement (possibly consisting of several substatements). This allows the FST translator to find a computational order of the statements (see also Section 4.3.1.6).

By using a single family of arrays (there is only a single array size variable), the program reflects the fact that all array variables refer to the same number of soil layers. In more complicated situations, it may be necessary to use several array families, i.e. to use more than a single array size variable. Calculations with arrays from different families are less easy, however. Strict rules apply, which are explained in section 4.3.1.7.

## 4.3 Calculations with array variables

In this section examples of array use are discussed. The concept of statement expansion and elementwise calculation is explained in Sections 4.3.1 and 4.3.2. In section 4.3.3, the FST array functions are introduced. They work on entire arrays and return average, maximum value, minimum value etc. In section 4.3.4 the function ELEMNT is introduced which is useful for models with a more than a single array family. The use of Fortran subroutines in combination with arrays is discussed in Chapter 5.

### 4.3.1 Terminology and Syntax rules

#### 4.3.1.1 Array declarations and array families

Before the actual model begins, the FST translator has to know which variables are arrays. A variable becomes an array by means of an ARRAY statement in the DECLARATIONS section of the FST program. An ARRAY statement looks like:

```
ARRAY XVALUE(1:N), YVALUE(0:N+5)
```

This statements declares the variable names XVALUE and YVALUE as arrays. Later, these array names may become calculated variables, model parameters or initial constants. All non-array names in the program are assumed to be ordinary, scalar variables.

To each array in the example above belong two subscript bounds. For XVALUE all subscripts lie between 1 and a yet unspecified number N. For YVALUE the subscript bounds are 0 and N+5. This implies that the actual array sizes depend on the yet unspecified value of the array size variable N. Array size variables are given a value by means of an ARRAY_SIZE statement in the MODEL section. For instance,

```
ARRAY_SIZE N=36
```

Since the length of both arrays depend on the same array size N, the two arrays are said to belong to the same array family. The arrays of an array family usually represent different properties of the same thing, for instance a soil layer, or a biological population. Calculations with arrays from the same family are much easier than with arrays from different families.

#### 4.3.1.2 Two examples of a calculated array variable

The simplest example of a calculated array is

```
DECLARATIONS
ARRAY A(1:N)
MODEL
...
A = 0.0
```

The assignment A=0.0 means that all elements of the array variable A are set equal to zero. A more complicate calculation is shown in the next example

```
DECLARATIONS
ARRAY A(1:N), B(1:N)                              <- The two arrays form a single array family
MODEL
...
A(1:5) = 1.0 ;    A(6:N) = 1.0 + B(6:N)**2
```

The last statement defines array variable A. The first five elements of A are set equal 1.0. The other elements depend on elements of array B. A(6) is equal to `1.0+B(6)**2`, A(7) is equal to `1.0+B(7)**2`, etc. Before this type of statements can be fully explained, some more terminology has to be introduced.

#### 4.3.1.3 Array elements, subscripts, subscript ranges and substatements

Elements of an array variable can be used in expressions in the same way as ordinary variables. For instance,

```
DECLARATIONS
ARRAY A(1:N)
MODEL
...
C = 1.0 + A(5) - A(N-1)
```

The ordinary, non-array variable C is calculated from two elements of array A, A(5) and A(N-1). The array subscript "5" in A(5) is called an absolute subscript. The array subscript "N-1" in A(N-1) is called a relative subscript since it points to an element number relative to the value of N.

Calculation statements may also refer to a series of array elements. A(1:5) means that the calculation must be carried out for A(1), A(2), A(3), A(4) and A(5) separately. The subscript range (1:5) is called an absolute-to-absolute subscript range absolute-to-absolute range, since both subscripts are absolute. Similarly we have absolute-to-relative subscript ranges absolute-to-relative range like in A(6:N-3) or relative-to-relative subscript rangesrelative-to-relative rangerelative-to-relative range like in A(N-2:N). The fourth category, relative-to-absolute, is forbidden in FST.

An example of an array calculation with all possible subscripts and subscript ranges is.

```
DECLARATIONS
ARRAY A(2:N), B(1:N)                <- Arrays A and B and their subscript bounds are declared,
...                                    a declared range is always absolute-to-relative
MODEL
...                                    array A is defined in a single statement:
A(2:4)      = 0.0       ;...        <- absolute-to-absolute subscript range
A(5)        = B(1)      ;...        <- absolute subscripts of A and B
A(6:N-8)    = B(6:N-8)  ;...        <- absolute-to-relative subscript ranges, defined and referenced
A(N-7:N-1)  = 0.0       ;...        <- relative-to-relative subscript range
A(N)        = B(2)                  <- a relative subscript of A and an absolute subscript of B
```

The declared range of A is split up into 5 parts. Each part is defined in a separate substatement and the substatements are separated by means of a semi-colon ";". Note that the use of statement continuation with three points "..." leads to a neat list of defined array elements.

The FST translator compares each subscript with the declared subscript bounds. The ARRAY statement in the above example declares for array A a lower bound 2 and an upper bound N. The use of A(1) or A(N+1), either as an array element or as part of a subscript range, will therefore result in an error message.

The use of a certain array subscript or subscript range also implies a restriction on the value of an array size variable. In the above example, the ARRAY statement itself implies already that N is at least 2. The use of A(5) implies that N is at least 5. Further, the subscript range A(6:N-8) should not be empty, which implies that N is 14 or larger. Finally, the calculations with all arrays of a certain family result in a lower limit for the array size variable. The actual value of the array size variable is tested against that lower limit. Moreover, the test is repeated during execution of the Fortran program as a safeguard against a change of N in the generated Fortran source.

A subscript range which is equal to the declared range may be omitted. This leads to a so-called full-range definition or a full-range array reference. Two examples:

```
DECLARATIONS
ARRAY A(2:N), B(1:N)   <- The two arrays form a single array family
MODEL
...
A=0.0                  <- Full-range definition of A, equivalent to A(2:N)=0.0
B(1)=1.0 ; ...
B(2:N)=A               <- Full-range reference of A, equivalent to B(2:N)=A(2:N)
```

#### 4.3.1.4 Statement expansion

As stated above, the use of a subscript range means that the calculations should be carried out separately for all elements involved. This is realized by the FST translator by means of statement expansion. The first example of Section 4.3.1.2 was

```
DECLARATIONS
ARRAY A(1:N)
MODEL
```

```
...
A = 0.0
```

which contains a full-range definition of array A. The FST translator <u>expands</u> this statement to a Fortran DO-loop. The expansion of A=0.0 is

```
      DO 10 I=1,N                 <- Counter I runs from 1 to N
         A(I) = 0.0               <- The I-th element of A is set at 0.0
10    CONTINUE                    <- Go back for next I
```

which means: a counter "I" runs from 1 to N and for each of these values of I element A(I) is set at 0.0. Strictly speaking, statement expansion belongs to the Fortran translation of an FST program and is not relevant to the user. Some understanding of statement expansion is helpful, however, in working with array variables.

If an array definition involves several substatements, the expansion consists of several DO-loops. For example, the calculation statement in

```
DECLARATIONS
ARRAY A(1:N), B(1:N)              <- The two arrays form a single array family
MODEL
...
A(1:5)      = B(1:5)    ; ...     <- Note the continuation which is used here in order to get
A(6:N-3)    = B(7:N-2)  ; ...        the substatements on different lines
A(N-2:N-1)  = 0.0       ; ...
A(N)        = B(6)+1.0
```

is expanded by the FST translator into

```
      DO 10 I=1,5                 <- The range of the loop counter I is the range of A
         A(I) = B(I)
10    CONTINUE
      DO 20 I=6,N-3               <- The range of the loop counter I is the range of A
         A(I) = B(I+1)            <- Note the subscript of B
20    CONTINUE
      DO 30 I=N-2,N-1             <- The range of the loop counter I is the range of A
         A(I) = 0.0
30    CONTINUE
      A(N) = B(6)+1.0
```

Note that several elements of B are not used and this does *NOT* generate an error or warning. A warning for an unreferenced array is only given if none of its elements is used. This is different, however, for the left part of the assignment, the defined array A. If one or more elements of A are left undefined, an error message is given.

Finally a more complicated example of statement expansion with functions:

```
DECLARATIONS
ARRAY A(1:N), B(1:N+1), C(0:N-1)
MODEL
...
A = SIN(B(2:N+1)) - AFGEN(FTB, 0.1 + MIN(0.0,COS(C)))
```

The array A is defined in a single substatement. Hence, the definition is a full-range definition and the subscript range of A, which is (1:N), can therefore be omitted. Array B contains more elements than array A, however, and therefore the used range of B has to be specified. Array C is equally long as array A and can be referenced without subscript bounds. The expansion by FST into a Fortran DO-loop is

```
      DO 10 I=1,N
         A(I) = SIN(B(I+1)) - AFGEN (FTB, 0.1 + MIN(0.0,COS(C(I-1))))
10    CONTINUE
```

Note that the interpolation function FTB, defined with a FUNCTION statement, is not an FST array (although it is a series of function values).

#### 4.3.1.5 The rules for calculating an array variable

With help of the terminology introduced above, the rules for calculation statements can be stated more precisely now. A few important rules at statement level are:

- Only a single array variable is calculated in the statement.
- The calculation of the array variable must be complete, i.e. all declared elements must be calculated.
- The statement may consist of substatements, each defining a part of the declared range.
- There are no gaps or overlaps in the subscript ranges of the consecutive substatements.
- Only a single substatement occurs with absolute-to-relative ranges, the reverse thing is impossible.

In each substatement, one or more array elements are calculated. The elements of an array may depend on elements of other arrays. The most important rule at the substatement level is that all subscript ranges in a substatement must be of the same type and must have the same length. The following correct substatements contain absolute-to-absolute subscript ranges

```
...; A(1:5)   = B(1:5)+3.4 ; ...                  <- Equal ranges, correct
...; A(2:9)   = B(8:15)+13.6 ;...                 <- From both arrays 8 elements
...; A(3:13)  = B(8:18)+C(102:112) ;...           <- From three arrays 11 elements
...; A(2:6)   = B(8:12)+SIN(C(-1:3)) ;...         <- From all arrays 5 elements
```

Note that the use of absolute-to-absolute subscript ranges does not require the arrays to be part of the same family.

A substatement with absolute-to-relative subscript ranges requires that the arrays belong to the same family (see also Section 4.3.1.7). For a family of three arrays A, B and C, the following substatements are correct:

```
...; A(3:N)    = B(3:N)+3.4 ; ...                 <- Equal ranges, correct
...; A(4:N-5)  = B(6:N-3)+13.6 ;...               <- From both arrays N-8 elements
...; A(8:N+2)  = B(2:N-4)+C(0:N-6) ;...           <- From all arrays N-5 elements
```

A substatement like

```
...; A(3:N)  = B(3:8)+3.4 ; ...                   <- ERROR, different types of subscript ranges
```

contains an absolute-to-relative and an absolute-to-absolute range and is therefore incorrect, even if N has been set to the value 8.

Finally we have the relative-to-relative subscript ranges. Again, for A, B and C in the same array family, the following substatements are correct

```
...; A(N+1:N+5)   = B(N+1:N+5)+3.4 ; ...                   <- Equal ranges, correct
...; A(N+2:N+9)   = B(N+8:N+15)+13.6 ;...                  <- From two arrays 8 elements
...; A(N+3:N+13)  = B(N+8:N+18)+C(N+102:N+112) ;...        <- From all arrays 11 elements
...; A(N+2:N+6)   = B(N+8:N+12)+SIN(C(N-1:N+3)) ;...       <- From all arrays 5 elements
```

The translator verifies all these rules and produces messages in case of errors.

Substatements may also define just a single array element. Such a single element has either an absolute or a relative subscript. Note that single array elements, like B(5) or B(N-3), can be referenced in all types of substatements as if they were ordinary, scalar variables. There are no requirements then with respect to the array family of B.

The complete list of rules is given in Section 8.5.2.1. A final example is

```
DECLARATIONS
ARRAY A(1:N), B(1:N+1)
MODEL
...
A(1:5)      = B(1:5)   ; ...      <-The defined parts of array A are consecutive without gaps or
A(6:N-3)    = B(7:N-2) ; ...        overlaps. Array B is just used. It needs not to be completely
A(N-2:N-1)= 0.0        ; ...        used.
A(N)        = B(6)+1.0
```

The following program lines contain <u>errors</u>:

```
DECLARATIONS
ARRAY A(1:N), B(0:N+10)
ARRAY C(-3:N+200)                      <- A, B and C form an array family
MODEL
...
ARRAY_SIZE N=100
...
A(1:5)=1.0 ; A(7:N)=2.0                <- ERROR: A(6) undefined
B(1:4)=0.0 ; B(5:N+10)=A(5:N+10)       <- ERRORs: B(0) undefined ; A(N+1:N+10) does not exist
C(-3:8)=0.0 ; C(9:N)=5.0 ; ...
C(N+1:130)=2.0 ; C(131:N+200)=1.0      <- ERROR: a relative-to-absolute range is illegal
```

#### 4.3.1.6 Sorting of array calculations

Calculated array variables are defined with a single statement. This allows the FST translator to find a computational order of all scalar and array calculations simultaneously. Sorting takes place, however, at the level of statements and not at the level of substatements. This implies that the following statement contains a dependency loop and leads to an error message.

```
A(1) = 2.3 ; A(2:N) = A(1)**2 + 3.0
```

FST reports a sorting loop: A depends on A because the array A depends on its own element A(1). Clearly, in Fortran the above calculations would be no problem, but in FST the sorting is done at the level of the entire array A, and not at the level of its individual elements. This is a limitation of FST, which can usually be bypassed by defining help variables. The above illegal statement is then replaced by

```
A(1) = A1 ; A(2:N) = A1**2 + 3.0
A1 = 2.3
```

which can be sorted.

#### 4.3.1.7 Arrays from different families

There are important restrictions on the use of arrays from different families in a single substatement. As long as the two subscripts or subscript ranges used are absolute, as in A(5:8) and C(6:9) for instance, there is no problem, even if A and C belong to different families. In case of absolute-to-relative and relative subscript ranges, however, it is forbidden to use arrays from different families in the same substatement.

The following example illustrates this rule for two array families, one consisting of A and B and the other of array C only.

```
ARRAY A(1:N), B(1:N), C(1:M)          <- Array family with arrays A and B and a family with array C
...
A(1:2) = C(1:2) ; ...                 <- Two absolute ranges, which is correct
A(3:N-4) = 2.0 * B(3:N-4) ; ...       <- Correct since A and B both use N
A(N-3:N) = 1.0 - C(N-3:N)             <- ERROR since C is from another family
```

Note that, even if N and M get the same values in an ARRAY_SIZE statement, the calculation of A is still in error. The reason for this strict rule is that the use of C(N-3:N) implies a relation between N and M, i.e. M should be equal to or larger than N. This type of relations is not verified by the FST translator. Therefore, it is impossible to use array size variable N in subscripts of C.

Although this rule limits the possibilities of FST, it has the advantage that users are forced to declare their arrays in meaningful families. Each array family describes properties of a series of simulated things, soil layers or plant species, or whatever. Mixing arrays from different families is then indeed a somewhat odd thing to do. Moreover, within each family one is free to declare arrays in slightly different ways, like in ARRAY A(0:N), B(2:N+20).

In case arrays from different families have to be used in the same expression, the limitations discussed here can be bypassed by using the ELEMNT function, which is explained in Section 4.3.4.

## 4.3.2 Array sizes and loop counter I in expressions

Sometimes it is useful to use the value of an array size variable in calculations. If a system with size LENGTH, for instance, is divided in N compartments with equal size, then the compartment size COMSIZ can be found as

```
ARRAY_SIZE N=48
COMSIZ = LENGTH / REAL(N)
```

This illustrates that array size variables may be used in expressions as long as they are properly converted into floating point numbers (with the REAL function). It is even possible to do calculations with array size variables using the rules for integer arithmetic in Fortran. For instance,

```
ARRAY_SIZE N=8 ; M=12
P = REAL((2*M-N)/6)                   <- Here P gets the value 2.000 !!!
```

Especially integer division easily leads to errors since the results are truncated. Therefore, FST warns against integer division.

A related topic is the use of the array element number in calculations. If we want to calculate the lower boundary of N compartments with size COMSIZ, then the answer for the 4-th layer is 3*COMSIZ, for the 9-th layer 8*COMSIZ etc. FST supports this type of calculations by allowing the element counter I to be used in expressions. This is demonstrated in the following example

```
ARRAY LBOUND(1:N)
PARAMETER SIZE=200.0                  <- the total system is 200 units large
ARRAY_SIZE N=32                       <- 32 compartments are used
COMSIZ = SIZE / REAL(N)               <- compartment size
LBOUND = REAL(I-1) * COMSIZ           <- find all lower bounds
```

It is essential to realize that the values of I are taken from the first subscript range of the statement, i.e. the subscript range in the left-hand part of the expression. In case of A(4:N-7)=..., for instance, I gets the values 4,5,6,...,N-7. As in case of array size variables, the loop counter I may be used in expressions, as long as it is handled as a Fortran INTEGER variable. The use of I in expressions is easier if statement expansion is fully understood (see Section 4.3.1.4). The variable I is nothing else than the DO-loop counter.

## 4.3.3 FST array functions

The FST array functions have been added to FST in order to simplify the calculation of properties of an entire array of values. For instance, the mean of the first 5 elements and the mean of all elements of an array A are calculated in

```
DECLARATIONS
ARRAY A(1:N)
MODEL
...
MEAN5 = ARMEAN(A,1,5)
MEANN = ARMEAN(A,1,N)
```

With similar functions other properties of the array can be calculated. The list of currently implemented FST functions working on a single array variable is:

```
X = ARMAXI (Array_Name, From, To),        <- The maximum value in the subscript range
X = ARMINI (Array_Name, From, To),        <- The minimum value in the subscript range
X = ARSUMM (Array_Name, From, To),        <- The sum of values in the subscript range
X = ARMEAN (Array_Name, From, To),        <- The mean of values in the subscript range
X = ARSTDV (Array_Name, From, To),        <- The standard deviation
X = ARLENG (Array_Name, From, To),        <- The length of the array seen as a vector
X = ARSMPS (Array_Name, From, To, DELX)   <- Integral if array contains function values
```

The precise mathematical definitions of these functions are given in Table 8.2 on page 136. There is one function which works on two array variables. Function ARIMPR calculates the vector improduct of two arrays.

```
X = ARIMPR (Array_Name, Array_Name, From, To),     <-The vector improduct A•B
```

The arguments "From" and "To" are the lower and upper subscript bounds to be used by the function. These arguments are integer constants, integer variables or integer expressions. Note that the first argument is an entire array. The functions work on this entire array and not on individual array elements. Therefore, no statement expansion takes place and the result of the function call is an ordinary scalar value. A few more examples are

```
ARRAY A(1:N)
...
B1 = ARMAXI(A,1,100)                          <-Maximum of the first 100 elements of A
B2 = 2.*ARLENG(A,1,3) + ARMINI(A,1,NINT(X+Y)) <-A calculation with function calls
B3 = ARSMPS (A,1,N,0.1)                       <-An estimated integral (See Section 8.7.1)
```

Clearly, there is a danger that the subscript range sent into ARMAXI, from 1 to 100, is not part of the declared range of A (here 1..N). During translation of the FST program this is NOT verified. The translator, however, adds the declared bounds, here 1 and N, as additional arguments to the ARMAXI function call. During execution of the model, the ARMAXI function then compares the requested subscript range with the declared range. If, in case of the above example, the value of N is less than 100, the function ARMAXI produces a fatal error message during *execution* of the program.

As a more complex example, we consider an array A with 80 elements and calculate the mean of the first 10 elements, the mean of the second 10 elements etc. The 8 average values are stored in array B with 8 elements (another array family). Array B is calculated with a single statement:

```
DECLARATIONS
ARRAY A(1:N), B(1:M)
ARRAY_SIZE N=80 ; M=8
MODEL
...
B = ARMEAN (A,10*I-9,10*I)
```

The calculation of array B is expanded by the FST translator to

```
      DO 10 I=1,M
         B(I) = ARMEAN (A,1,N,10*I-9,10*I)     <- Declared range of A is added !!
10    CONTINUE
```

and we see how the DO-loop counter I is used in FST to calculate the appropriate subscript range of A. For element I of array B the subscript range is `(10*I-9:10*I)`, which becomes (1:10), (11:20), ..(71:80). Note that the ARMEAN function call in Fortran includes the declared bounds of A. These are added by the translator to support the runtime existence check on the elements of A.

### 4.3.4 The function ELEMNT

The rules on array use in FST sometimes lead to error messages on constructions that look natural. Especially the restrictions on the use of arrays from different families in the same expression causes problems in relatively simple situations. An example is given in the first Section 4.3.4.1. The problem is then bypassed in Section 4.3.4.2 with help of the ELEMNT function. In Section 4.3.4.3 the proper use of the ELEMNT function is discussed.

#### 4.3.4.1 Limitation of array use in FST

As a consequence of the rules explained in Section 4.3.1, the following simple calculation is illegal.

```
DECLARATIONS
ARRAY A(1:N), B(1:M)
MODEL
ARRAY_SIZE N=5, M=100
.....
B = REAL(I)                          <- assign values 1,2,3,....M (see Section 4.3.2)
A(1:N) = B(1:N)                      <- ERROR, Range of B contains N instead of M
```

In this example, a subscript or subscript range of array B, which was declared with ARRAY B(1:M), contains the "foreign" array size variable N (cf. Section 4.3.1.7). Of course, the idea of the statements above is that A contains the first N elements of B. Writing A(1:5)=B(1:5) does not solve the problem, since it does not form a complete definition of A (the piece from 6 to N lacks). Writing

```
A(1:4) = B(1:4) ; A(5:N)=B(5)
```

works for N=5. For larger values of N this statement remains to be a valid statement in FST, but it does not do what we want! For N= 6, for instance, both A(5) and A(6) become equal to B(5). In general, all elements of A with numbers 5 and higher are made equal to B(5). Hence, if we want to copy the first N elements of B, the above statement is essentially wrong since it does a different thing for values of N exceeding 5. The solution of the problem is the use of the function ELEMNT.

#### 4.3.4.2 The ELEMNT function as a solution

The only correct way of assigning the first N values of array B to array A is

```
DECLARATIONS
ARRAY A(1:N), B(1:M)
MODEL
ARRAY_SIZE N=5, M=100
.....
B = REAL(I)                         <- Assign values 1,2,3,....
A(1:N) = ELEMNT(B,I)                <- To A(I) is assigned the value of B(I), where I=1...N
```

The function ELEMNT ("the element function") simply returns the value of an array element and has two arguments, an array name and an element number. Hence, ELEMNT(B,3) returns the value of B(3) and ELEMNT(B,N) returns the value of B(N).

The difference between, say, B(N-1) and ELEMNT(B,N-1), is that B(N-1) is valid only if the array B is declared using N as array size variable. The expression ELEMNT(B,N-1) is always valid. The declared bounds of B, however, are added to the function call by the FST translator and the function ELEMNT verifies the existence of the requested element during *execution* of the program. Hence, the existence check on elements of B takes place during *execution* of the generated Fortran program instead during translation.

In the above example, I is the DO-loop counter running from 1 to N (the left-hand subscript range is (1:N)). The integer argument of the ELEMNT call can be replaced by any other integer expression, by I+2, for instance. Also 1.0 + SIN(0.1+ELEMNT(B,I+2)) is a valid construction. Even ELEMNT(B,100+I) does not lead to an error message of the FST translator. It leads to a fatal error during *execution* of the program, however.

#### 4.3.4.3 Disadvantages of the ELEMNT function

In order to prevent error messages about illegal use of array size variables, one could use the ELEMNT function in all expanded array expressions. We strongly advise, however, not to do so. The first reason is that part of the translators' error checking capabilities would simply be ignored. Instead of neat error messages one gets fatal execution errors of ELEMNT (execution stops without proper termination of the model run and without output being generated). The second reason is that the ELEMNT function slows down considerably the execution of the generated Fortran program.

Hence, instead of just plugging in ELEMNT function calls, it is a good idea to look a bit closer to the ARRAY statements. Is it really necessary to use different array families for the problem at hand? If array B is always 5 elements larger than array A, the statement ARRAY A(1:N), B(1:N+5) will solve the problem. An example of such a construction is the array variable FLUX in Listing 4.1 on page 44.

#### 4.3.4.4 A complicated example

Suppose we want to calculate all $N^2$ distances between N objects, each with an X and a Y coordinate. The example below declares the arrays X and Y with N elements and an array DIST with $N^2$ elements (numerically set).

```
DECLARATIONS
ARRAY X(0:N-1), Y(0:N-1), DIST(0:NN-1)
MODEL
ARRAY_SIZE N=10, NN=100
......
DIST = SQRT ((ELEMNT(X,MOD(I,N)) - ELEMNT(X,I/N))**2 + ...
            (ELEMNT(Y,MOD(I,N)) - ELEMNT(Y,I/N))**2))
```

Clearly, a second array size variable is needed here. The expression for DIST shows how the distances can be calculated. The DO-loop counter I runs with the left-hand array DIST, from 0 to 99. Then the integer argument MOD(I,N) takes the values 0,1,2,3...9, 0,1,2,3...9, ....... 0,1,2,3...9. The integer argument I/N makes use of integer arithmetic in Fortran and takes the values 0,0,0...0, 1,1,1...1, ....... 9,9,9....9. These two series together make up all combinations of the N objects (0,0), (1,0), ..... (9,9).

This example illustrates that practical use of the ELEMNT function may require subtle calculations on array subscripts. Of course, the $N^2$ distances between N objects form a matrix and the subscript calculations result from the fact that FST does not allow two-dimensional arrays. An alternative solution to the problem of $N^2$ distances is the use of a Fortran subroutine in which DIST is declared as a two-dimensional array.

Such a subroutine looks like

```
      SUBROUTINE DISTAN (X,Y,N,DIST,NN)
      INTEGER N,NN,I,J
      REAL X(0:N-1),Y(0:N-1),DIST(0:N-1,0:N-1)
      IF (NN.NE.N*N) CALL ERROR ('DISTAN','illegal value of NN')
      DO 20 J=0,N-1
         DO 10 I=0,N-1
            DIST(I,J) = SQRT ((X(I)-X(J))**2 + ((Y(I)-Y(J))**2)
10       CONTINUE
20    CONTINUE
      RETURN
      END
```

This subroutine returns exactly the same array DIST as the FST statement above. In this case, one might consider the use of the ELEMNT function because the subscript calculations are not too difficult. The use of a subroutine leads to much faster calculations, however, and forms the only solution in more complicated cases.

Hence, the ELEMNT function is most appropriate in relatively simple situations. When models grow larger, straightforward numerical work has to be left to subroutines. Complicated subscript calculations in integer arithmetic have to be avoided. In Chapter 5 the use of Fortran subroutines in FST is fully discussed.

## 4.4 PARAMETER and INCON arrays

Like calculated array variables, also PARAMETER and INCON arrays are defined in a single statement. A simple example is

```
DECLARATIONS
ARRAY A(1:N)
MODEL
...
PARAMETER A=0.0
```

The parameter statement contains a full-range definition of array A that looks exactly like a calculation of A. In the same way as for calculated arrays, the array A can be defined in a series of substatements like in

```
PARAMETER A(1:2)=0.0 ; A(3:6)=1.0,2.0,3.0,4.0 ; A(7:N-4)=5.0 ; A(N-3:N)=6.0
```

Clearly, since we define a parameter array, the right part of the substatements cannot contain expressions. The parameter array A is a series of constants. A second difference with a calculated array is that the right part of a substatement may contain more than a single value. The above example contains A(3:6)=1.0,2.0,3.0,4.0. This is equivalent to A(3)=1.0 ; A(4)=2.0 ; A(5)=3.0 ; A(6)=4.0.

Note that several values can only be given if the number of defined elements is independent of N. The range A(3:6) contains four elements. The range A(N-3:N) also contains four elements and might be defined with A(N-3:N)=3.0,4.0,5.0,6.0. The number of elements in A(7:N-4), however, depends on N and A(7:N-4) has to be defined with a single value.

INCON arrays are defined in a similar way. The precise rules for writing PARAMETER (or INCON) statements are given in Section 8.5.2.17 on page 127. The rules for INTGRL statements with array variables acting as states, rates and initial values are given in Section 8.7.3 on page 141.

Chapter 5

# Subroutines called from FST

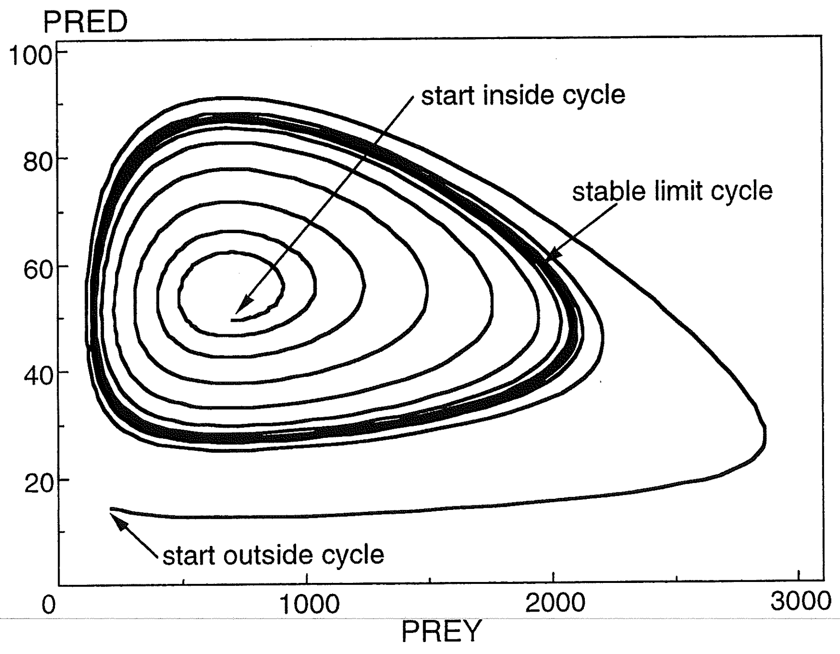


*Figure 5.1 on the title page of this chapter.* The dynamics of the predator-prey system given in Listing 6.5 on page 76. Every point of the (PREY,PRED) space represents a status of the system. The curves are simulated trajectories in the predator-prey space. After a sufficient amount of time, the simulated system ends up in a stable limit cycle. The oscillation of prey and predator populations is independent of the initial conditions.

# 5 Subroutines called from FST

The simplest way of using a Fortran subroutine is in combination with ordinary, scalar variables. The subroutine may require some input and will calculate something which is returned as one or more output variables. Basic subroutine use is described in section 5.1. Section 5.2 describes how to pass entire arrays to a subroutine. Clearly, the subroutine should be prepared to receive an array instead of a single variable.

In section 5.3 arrays are used in combination with a subroutine which expects ordinary, scalar variables. In that case the subroutine call is expanded and the array elements are calculated in separate calls.

## 5.1 Ordinary, scalar variables as input and output

As an example we use the following subroutine, which calculates A*$X^2$ and A*$X^3$ for any value of A and X.

```
      SUBROUTINE EXAMP1 (A,X,Y2,Y3)
      REAL A,X,Y2,Y3
      Y2 = A * X**2
      Y3 = A * X**3
      RETURN
      END
```

The first two arguments, A and X, are input arguments. The !ast two arguments are output arguments. The following FST program section makes use of this subroutine:

```
PARAMETER MULT=3.4
SUM = S2 + S3
CALL EXAMP1 (MULT,X,S2,S3)
X = 4.5
```

In these four lines we make use of the statement sorting capability of FST. At first, X has to be calculated, then the subroutine can be called and finally S2 and S3 are added. In order to choose the right statement order, the FST translator has to know that MULT and X are input variables, and that S2 and S3 are output variables of the subroutine. That is achieved by the following statement which should be on top of the FST program in the DECLARATIONS section.

```
DEFINE_CALL EXAMP1 (INPUT,INPUT,OUTPUT,OUTPUT)
```

This DEFINE_CALL statement is a formal description of the function of the subroutine in the model. Each time the subroutine is used, its first two arguments are input and its last two arguments are output. This shows that the use of a subroutine in FST requires nothing more than telling the translator about the different types of subroutine arguments. In Fortran an input argument can be overwritten and becomes an output argument as well. Subroutines which do that cannot be directly called from FST.

Input arguments may also be constants or expressions. Hence,

```
SUM = S2 + S3
CALL EXAMP1 (3.4, 1.0+SQRT(X), S2,S3)
X = 4.5
```

is a correct piece of program. An output argument, however, can never be an expression.

## 5.2 Entire arrays as input and output

Instead of calculating the sum of S2 and S3 we want to use an entire series of powers of X. In other words, we want to calculate

$$\text{SUM} = \sum_{i=2}^{N} S_i = \sum_{i=2}^{N} \text{MULT} \times X^i$$

and we also want to calculate the terms of the series in a subroutine. A subroutine calculating an entire series is

```
      SUBROUTINE EXAMP2 (A,X,Y,N)
      INTEGER I,N
      REAL A,X,Y(N)
      DO 10 I=1,N
         Y(I) = A * X**REAL(I)
10    CONTINUE
      RETURN
      END
```

Note that the first element of the array Y is Y(1) and the last element is Y(N). This is not necessary, but it simplifies the program. The FST program lines become

```
DECLARATIONS
ARRAY S(1:N)                    <- The series S declared from 1 to N !!
DEFINE_CALL EXAMP2 ...          <- Declare the new subroutine
   (INPUT,INPUT,...             <- The same as for EXAMP1 above
    OUTPUT_ARRAY,...            <- An entire array is calculated
    INTEGER_INPUT)              <- The integer array size is passed to the subroutine
MODEL
...
ARRAY_SIZE N=5
PARAMETER MULT=3.4
SUM = ARSUMM(S,2,N)             <- The sum of the series, starting at the 2-nd term
CALL EXAMP2 (MULT,X,S,N)        <- Calculate the series S
X = 4.5
```

The declared subscript range of S is (1:N) instead of (2:N), which leads to a neater program and a neater subroutine. In *this* FST program S(1) is not used, which is no problem at all.

The third argument of EXAMP2 is declared as OUTPUT_ARRAY, which enables the calculation of array S in a single call to EXAMP2. If a subroutine requires an input array, the argument type is INPUT_ARRAY. The size of an array argument can be passed to the subroutine as an INTEGER_INPUT argument. The argument type INTEGER_OUTPUT does not exist.

After declaring the subroutine, the actual call looks again like Fortran. The required sum is calculated with a call to the FST intrinsic function ARSUMM (See Table 8.2 on page 136). Note that FST sorts the statements and will calculate at first X, then the series S and finally SUM.

In case of this simple series we could have calculated the terms directly in FST, for instance with

```
SUM = ARSUMM(S,2,N)
S = MULT * X**REAL(I)           <- will be expanded by FST
X = 4.5
```

Calculation of a series may require complicated iterative procedures, however, and in such cases a subroutine is the only solution.

## 5.3 Expansion of a subroutine call

Suppose we have available an existing subroutine FLIB1 which uses ordinary, scalar arguments and is declared as

```
DEFINE_CALL FLIB1 (INPUT,OUTPUT,OUTPUT)
```

As discussed in Section 5.1, a call to the subroutine could look like

```
CALL FLIB1 (1.0+X, Y, Z)
```

The input argument is an expression, 1.0+X, from which the variables Y and Z are calculated. Now, suppose that X, Y and Z are declared as arrays with equal subscript bounds. The problem is then the calculation of arrays Y and Z from array X <u>without changing the subroutine</u>. Changes in a well-tested subroutine are undesirable. Making changes is even impossible if the subroutine is only available in compiled form, as part of a library for instance. The first solution to this problem is discussed in this section. A second solution is discussed in Section 5.4.

This first solution is easy. The variables X, Y and Z are declared as arrays and the subroutine is called as if nothing has changed!

```
DECLARATIONS
DEFINE_CALL FLIB1 (INPUT,OUTPUT,OUTPUT)
ARRAY X(1:N), Y(1:N), Z(1:N)
MODEL
....
CALL FLIB1 (1.0+X,Y,Z)                        <-Expanded subroutine call
```

FST treats this call in the same way as a statement like Y=1.0+X, or Z=SQRT(1.0+X). Such statements would be expanded and the calculations would be carried out elementwise in a Fortran DO-loop in the way discussed in Section 4.3.1.4. The call to FLIB1 is expanded in a similar way:

```
      DO 10 I=1,N
         CALL FLIB1 (1.0+X(I),Y(I),Z(I))      <-For each I a separate call
10    CONTINUE
```

The values Y(I) and Z(I) are calculated by means of a series of calls to the subroutine. Together, these calls define the entire arrays Y and Z. The expansion of a subroutine call is essentially nothing else than the expansion of assignments in which arrays or array subscript ranges occur.

### 5.3.1 The rules for expansion of a subroutine call

<u>The two basic rules</u> for expansion of a subroutine call are:

- A subroutine call is expanded if an array subscript range or a full-range array is used at the position of a scalar INPUT or OUTPUT argument.
- A subroutine call does <u>not</u> allow substatements for different array ranges.

In addition to these basic rules, we have the rules for calculation discussed in Section 4.3. They apply to subroutine calls as well. This implies:

- If an array subscript range covers the entire declared range, the subscript range may be omitted. The subroutine call above is an example. It is equivalent to
  `CALL FLIB1 (1.0+X(1:N),Y(1:N),Z(1:N)).`
- The expanded array subscript ranges in the entire call need to be equally long and of the same type (for subscript types see Section 4.3.1.3).
- In case of absolute-to-relative and relative-to-relative ranges all expanded arrays must belong to the same family (for this concept see Section 4.3.1.1)
- The DO-loop counter I may be used in one or more of the arguments of an expanded call. The following rule was given already in Section 4.3.2: The range of I is the range of the first expanded array variable in the statement. An example is given below in Section 5.3.2.1.

Other rules are the consequence of the principles of variable calculation in FST:
- A calculated array variable has to be completely defined in a single statement. Since a subroutine call cannot consist of substatements, a calculated array is necessarily full-range. Again, the above subroutine call is an example. Both Y and Z are full-range.
- A variable can be calculated only once. This implies that a scalar OUTPUT argument is incompatible with statement expansion. A scalar variable would be calculated again and again for each subroutine call in the generated DO-loop.
- For the same reason OUTPUT_ARRAY arguments are incompatible with statement expansion. An OUTPUT_ARRAY argument is an entire array, calculated by means of a single call to a subroutine. It will not be calculated in an expanded call.
- The last three rules imply that all output arguments of an expanded call must be of type OUTPUT with full-range arrays as actual arguments.
- And this implies that the subscript ranges in an expanded subroutine call are always absolute-to-relative, unless the subroutine does not have output arguments.

Finally, an expanded subroutine call is still a subroutine call:
- An INPUT argument may also be an expression. An input expression containing array variables is simply expanded. The example above has the expression 1.0+X(1:N) as input argument.
- INPUT_ARRAY arguments are entire arrays that are passed to the subroutine. This does not interfere with expansion of the call. An INPUT_ARRAY argument is simply passed to the subroutine each time it is called. An INPUT_ARRAY does not need to belong to the family of the expanded arrays in the call. An example is given below in Section 5.3.2.

## 5.3.2 Examples of expanded subroutine calls

The first example once more shows that an INPUT argument may be an expression containing array variables. The arrays Y and Z are calculated from the larger arrays A and B in

```
DECLARATIONS
DEFINE_CALL FLIB2 (INPUT,INPUT,OUTPUT,OUTPUT)
ARRAY A(1:N+10), B(1:N+10), Y(1:N), Z(1:N)
MODEL
....
CALL FLIB1 (2.5*A(1:N)+SQRT(B(1:N)),Y,Z)          <- Expanded call with input expression
```

The subroutine call is expanded to

```
      DO 10 I=1,N
         CALL FLIB1 (2.5*A(I)+SQRT(B(I)), Y(I), Z(I))
10    CONTINUE
```

Not all input arguments of an expanded subroutine call need to be arrays or array expressions. The first argument in the call to subroutine FLIB2 below is a scalar input variable Q:

```
DECLARATIONS
DEFINE_CALL FLIB2 (INPUT,INPUT,OUTPUT,OUTPUT)
ARRAY X(1:N), Y(1:N), Z(1:N)
MODEL
....
PARAMETER Q=2.0
CALL FLIB2 (Q,X,Y,Z)                        <- Expanded call with scalar Q as input
```

The expanded subroutine call is

```
      DO 10 I=1,N
         CALL FLIB2 (Q,X(I),Y(I),Z(I))      <- For each I a separate call
10    CONTINUE
```

In case of a scalar OUTPUT argument, however, the translator reports an error. The same happens for OUTPUT_ARRAY arguments. They cannot be combined with expansion. These errors are illustrated by the following program lines.

```
DECLARATIONS
DEFINE_CALL FLIB3 (INPUT,OUTPUT)
DEFINE_CALL FLIB4 (INPUT,OUTPUT,OUTPUT_ARRAY)
ARRAY Y(1:N), T(1:N), U(1:N)
MODEL
...
CALL FLIB3 (Y,C)            <- ERROR: Array Y is used as INPUT which requires expansion
...                            scalar C is OUTPUT which does not allow expansion.
CALL FLIB4 (A,T,U)          <- ERROR: Array T is used as OUTPUT which requires expansion
...                            array U is OUTPUT_ARRAY which does not allow expansion.
```

INPUT_ARRAY arguments are no problem. In such a case, the subroutine requires an entire array as input. If, in addition, an array is used at the position of a scalar OUTPUT argument, the call is expanded. For example:

```
DECLARATIONS
DEFINE_CALL FLIB5 (INPUT_ARRAY, INTEGER_INPUT, INPUT, OUTPUT)
ARRAY Y(1:N), Z(1:N)
ARRAY X(1:M)             <- Array X belongs to a different family
MODEL
...
CALL FLIB5 (X,M,Y,Z)     <- Expanded: Z(I) is calculated from array X, integer M and Y(I)
```

The expansion of this call is

```
      DO 10 I=1,N
         CALL FLIB5 (X,M,Y(I),Z(I))
10    CONTINUE
```

Indeed, the INPUT_ARRAY X is not expanded but passed to the subroutine. Note that the array X not necessarily belongs to the family of Y and Z.

#### 5.3.2.1 The counter I in an expanded subroutine call

Finally, an example with the DO-loop counter I used in the call. The range of the counter is the range of the first expanded array in the statement. Hence, the declared range of an INPUT_ARRAY argument is insignificant!

```
DECLARATIONS
DEFINE_CALL FLIB6 (INTEGER_INPUT, INPUT_ARRAY, INTEGER_INPUT, INPUT, OUTPUT)
```

```
ARRAY X(1:M+10)
ARRAY Y(8:N+7), Z(1:N)
MODEL
...
CALL FLIB6 (I-7,X,M+10,1.0+Y,Z)
```

This requires some more explanation. The subroutine FLIB6 has 5 arguments. The first argument is an integer value used in the calculations. The second and third argument are an array with a certain length, also used in the calculations. The fourth and fifth argument are scalar input and output arguments. This subroutine is used to calculate the elements of array Z, declared with subscript bounds 1 and N. The elements of Z depend, say, on their own element number, on array X and on the elements of an equally long array Y declared with bounds 8 and N+7.

The DO-loop counter I runs over the declared range of Y because the array Y is the first expanded array of the statement. The expansion then becomes:

```
      DO 10 I=8,N+7
         CALL FLIB6 (I-7,X,M+10,1.0+Y(I),Z(I-7))
10    CONTINUE
```

Hence, if the elements of Z depend on their own element number, the call to FLIB6 should contain the integer number I-7. Of course, if Y would be declared as Y(1:N), the first argument could simply be I. The example demonstrates, however, that the use of I may sometimes be tricky.

## 5.4 An interface subroutine

Sometimes, statement expansion cannot be used. In case arrays from different families have to be used as arguments of the subroutine or in case one or more output arguments remain scalar, FST produces an error message. In such a case, we suggest to hide the subroutine for FST by calling it indirectly, via a small "interface subroutine".

A second reason for hiding a subroutine from FST is an input argument which is overwritten. Such an argument acts as input and output which is not possible for a subroutine directly called from FST.

The problem discussed below requires an interface subroutine. Suppose we have a subroutine FLIB1 described by the following input/output structure:

```
DEFINE_CALL FLIB1 (INPUT,OUTPUT,OUTPUT)
```

This routine calculates two output variables from one input variable. Suppose we want to call it for an array of input values X and produce two arrays of output values, Y and Z. Statement expansion does not work, however, since the input array and the output arrays belong to different array families.

The only way to combine arrays of different families in a subroutine CALL is to write an interface subroutine INTF1

```
      SUBROUTINE INTF1 (X,N,Y,Z,M)
      INTEGER N,M
      REAL X(N), Y(M),Z(M)
      IF (N.LT.M) CALL ERROR ('INTF1','N < M')
      DO 10 I=1,M
         CALL FLIB1 (X(I),Y(I),Z(I)       <- The original is left unchanged !
10    CONTINUE
      RETURN
      END
```

The FST program using this interface then contains

```
DECLARATIONS
DEFINE_CALL INTF1 (INPUT_ARRAY, INTEGER_INPUT, ...
                   OUTPUT_ARRAY, OUTPUT_ARRAY, INTEGER_INPUT)
ARRAY X(1:N)
ARRAY Y(1:M), Z(1:M)                <- Another family
MODEL
ARRAY_SIZE N=100 ; M=10
....
CALL INTF1 (X,N,Y,Z,M)              <- Subroutine FLIB1 is not called directly and is not
                                       described anymore in FST
```

The subroutine FLIB1 is called indirectly and FST does not need to know its input and output structure. The original subroutine can be left unchanged and its Fortran source code is not even needed.

At first sight, interface subroutines for just calling another subroutine may seem undesirable overhead. In general, however, they form a powerful tool in connecting subroutines with different authors, origin and style, without changing the subroutines themselves and creating a maintenance problem. Sometimes, calling a subroutine may require the use of a Fortran COMMON block, for instance. Then, without changing the internal logic of that subroutine (and the thousands of program lines it possibly contains), a small interface subroutine, which contains the COMMON declarations, can be called from FST.

Chapter 6

# Examples

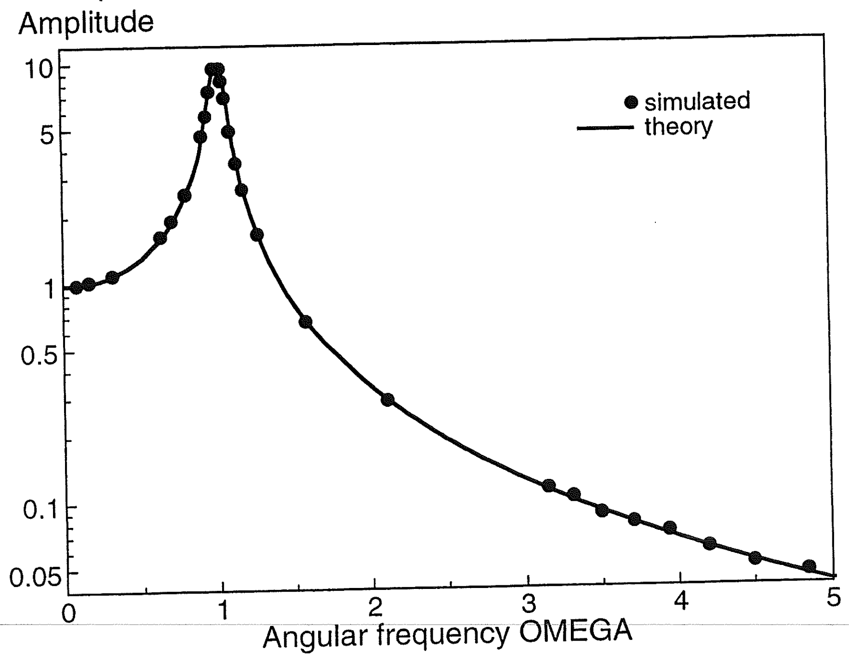


*Figure 6.1 on the title page of this chapter.* The response of a damped, harmonic oscillator to a periodic force. If the period of the force is close to the natural period of the oscillator, the amplitude becomes large. The phenomenon is widely known as resonance. The model is given in Listing 6.7 on page 80.

# 6 Examples

In the first section, Tips and Tricks, a few often returning tricks are briefly discussed. Then full example programs are given. The material in these example programs comes from biology and physics.

The material in this chapter is meant to be interesting for both beginning and experienced users. Complete understanding of the example programs in Section 6.5 and Section 6.6 requires some knowledge of mechanics. These programs have been written as test programs for the translator and give therefore also an impression of the possibilities of FST.

## 6.1 Tips and Tricks

### 6.1.1 Plotting an AFGEN or CSPLIN function

The small program in Listing 6.1 on page 70 calculates interpolated values for the function TEST given as seven (x,y) pairs. The program makes use of the range of TIME, between STTIME and FINTIM, to set the X values for the interpolation. It uses then both interpolation techniques available, linear interpolation and cubic spline interpolation (see Section 3.4). The value of PRDEL can be used to determine the number of X points for which function values are calculated.

Although Listing 6.1 represents a full FST program, it does not contain any state variables. It only simulates TIME, starting at STTIME and ending at FINTIM.

The interpolation results can be inspected in Figure 6.2. During most of the X-range the cubic spline algorithm leads to neat function values. The curve between 6 and 9, however, is fantasy of the cubic spline algorithm which uses a third order polynomial between every two (x,y) pairs without sudden changes in direction. One should be aware of this type of unexpected behaviour.

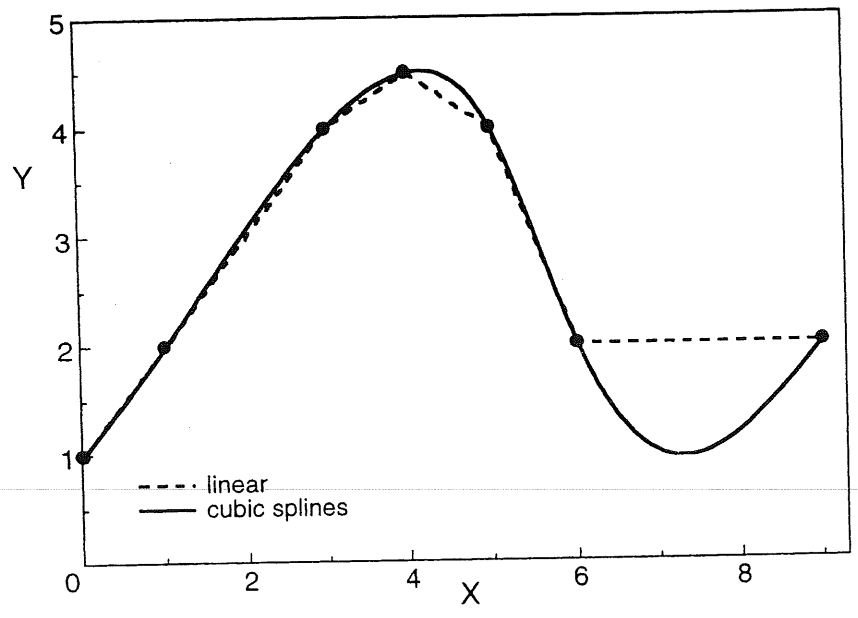


Figure 6.2 Cubic spline interpolation fails between the two rightmost points.

Listing 6.1 Inspecting FUNCTION interpolation

```
TITLE Two Interpolation Methods
MODEL
FUNCTION TEST = 0.0, 1.0, 1.0, 2.0,...
                3.0, 4.0, 4.0, 4.5,...
                5.0, 4.0, 6.0, 2.0, 9.0, 2.0

* make use of TIME to get a range of X-values
TIMER STTIME=0.0 ; FINTIM=9.0 ; DELT=0.1 ; PRDEL=0.1
X = TIME

* linear interpolation
Y1 = AFGEN (TEST,X)

* interpolation with natural cubic splines
Y3 = CSPLIN (TEST,X)

PRINT X,Y1,Y3
TRANSLATION_GENERAL DRIVER='EUDRIV'
END
```

## 6.1.2 Plotting an arbitrary function

An arbitrary function can be studied by using the same technique as in Section 6.1.1. The AFGEN or CSPLIN function call is simply replaced by any other function of X. Listing 6.2 shows that the dependent variable can even be calculated in a subroutine.

Listing 6.2 Inspecting the result of a subroutine which calculates a function y(x)FST program structure

```
TITLE Inspect subroutine result
DECLARATIONS
DEFINE_CALL MYSUB (INPUT,OUTPUT)

MODEL
* make use of TIME to get a range of X-values
TIMER STTIME=0.0 ; FINTIM=9.0 ; DELT=0.1 ; PRDEL=0.1
X = TIME

* subroutine call
CALL MYSUB (X,Y)

PRINT X,Y
TRANSLATION_GENERAL DRIVER='EUDRIV'
END
```

Again X is a copy of TIME. The X-range studied is then [STTIME,FINTIM] with a resolution of PRDEL. There are no state variables and the driver EUDRIV can be used to control TIME.

## 6.1.3 TERMINAL output

Often, the final result of a simulation run is studied as function of the value of an input variable. In the construction below, the variables of interest are model parameter A and the final value of state variable X. By printing simply A and X, we get a single, INITIAL value of A and a list of X values, one at each output interval. A better construction is the following one:

```
MODEL
PARAMETER A=2.0
...
DYNAMIC
X = INTGRL (IX,RX)
...
TERMINAL
AT = A                       <-A copy of parameter A
XT = X                       <-The final value of state X
PRINT AT,XT                  <-Terminal output of these variables
TIMER PRDEL=0.1              <-For AT and XT this value is not relevant
...
TIMER STTIME=0.0 ; FINTIM=10.0 ; DELT=0.1
END
```

The final result of the simulation is the value of X at FINTIM. This value is available for calculations in the TERMINAL section. Here, it is just copied into the terminal variable XT. This avoids a long list of intermediate X values during the simulation. In order to study XT as function of A, also A is copied into a terminal variable, AT. The table in the results file RES.DAT looks like

```
*---------------------------------------------------------------------
 * Output table number  :  0 (=first output table)
 * Output table format  : Table output
 * Simulation results

      TIME           AT           XT

   10.0000       0.10000       2.4596
```

Hence, parameter A and the terminal value of X are printed on the same line. This simplifies the making of graphs with help of spreadsheet programs (use IPFORM=5, see Section 3.2.4.3) or the graphical utility program TTSELECT (see 7.2.1). Note that terminal output is independent of the value of PRDEL, the dynamic output interval. Many of the example programs in this chapter make use of terminal output.

### 6.1.4 The start year IYEAR and the current year YEAR

In case weather data are used, the start year of the simulation is the value of the WEATHER control variable IYEAR. The value of IYEAR is not available in the FST program, however. Instead, the variable YEAR has to be used, which belongs to the group of weather and calendar data (see Section 3.2.4.4). The variable YEAR is the current year and is updated if the simulation reaches a new year.

Often, simulation results will depend on the choice of the start year IYEAR. Printing YEAR, however, leads to a value of YEAR for each output interval and not to a unique value of the start year. The solution is to define an INITIAL variable which stores the value of YEAR at STTIME:

```
WEATHER IYEAR=1985, ....
INITIAL
* define and print start year
SYEAR = YEAR
PRINT SYEAR
...
DYNAMIC
...
END
```

Since SYEAR is a variable calculated in INITIAL, it will be printed only once, at time STTIME. Another interesting application of SYEAR is a FINISH condition which terminates the run at the beginning of a new year

```
FINISH YEAR > SYEAR
```

This simple construction does not require day counting and works also in leap years.

In addition, the start year can be copied in a terminal output variable (see Section 6.1.3). We have to copy SYEAR in a calculated terminal variable. This leads to the following construction:

```
WEATHER IYEAR=1985, ....
INITIAL
* define start year
SYEAR = YEAR
...
DYNAMIC
...
TERMINAL
RESULT = ....
* start year, terminal copy of SYEAR
SY = SYEAR
* print start year and result, both at FINTIM
PRINT SY, RESULT
END
```

Note that just printing a terminal copy of YEAR will give the final year and not the start year in the results. Section 6.2 gives a practical example of the entire construction.

### 6.1.5 Choosing between expressions

Sometimes there are two alternative expressions for a quantity, say Y. Which one is to be used depends on the value of another expression. An example is

```
Y = 1.0 + INSW(X-2.3, SIN(A), COS(B))
```

The FST intrinsic function uses the first argument as a control variable. Depending on the sign of the control variable, the function result is equal to either the second or the third argument. With the above statement, Y is calculated as 1.0+SIN(A) for X<2.3 and as 1.0+COS(B) for X>=2.3. A precise description of the arguments of INSW can be found in Table 8.2 on page 136.

A similar function is FCNSW which switches between three possible expressions, depending on the value its first argument (see Table 8.2 on page 136).

### 6.1.6 Switching integration on and off

The function INSW can be used to switch on a rate of change during a certain time interval. The following program lines give an example.

```
S = INTGRL (IS,RS)
RS = (A*S) * INSW (TIME-TSTART, 0.0, 1.0) * INSW(TIME-TEND, 1.0, 0.0)
PARAMETER A=0.01 ; TSTART=10.0 ; TEND=20.0
```

The rate of change is set equal to a constant A times S, which means that the amount S will grow exponentially. The process works only for values of TIME larger than TSTART but smaller than TEND. Outside that time interval, one of the INSW calls is zero. Of course, in this simple case, one could simply use STTIME and FINTIM to control the simulation. If the process is part of a larger model, however, this is not always possible.

The use of a function like INSW represents conditional calculations. In Fortran or any other general programming language, one would use IF-THEN-ELSE control structures to write down all conditions explicitly. The above example would look like

```
IF (TIME.GE.TSTART .AND. TIME.LT.TEND) THEN
   RS = A*S
ELSE
   RS = 0.0
END IF
```

In FST such conditions have to be written in the form of INSW function calls. In complicated cases this tends to become very tricky, for instance in case of nested conditions. Then the use of a Fortran subroutine will often be a better solution.

## 6.1.7 Subroutines requiring INTEGER input

The Fortran intrinsic functions INT and NINT (see Table 8.1 on page 134) can be used to convert the result of calculations to an integer value. This can be done as part of the subroutine call. Say, we have a subroutine which requires an INTEGER first argument. The following program lines give examples of possible call statements

```
DECLARATIONS
DEFINE_CALL MYSUB (INTEGER_INPUT, OUTPUT)  <- first argument is integer
...
MODEL
...
PARAMETER A=3.4
B = SQRT(A) + 10.0
CALL MYSUB (NINT(A), C)                    <- first argument becomes 3
CALL MYSUB (NINT(1.0+A*B), D)              <- first argument calculated
CALL MYSUB (NINT(YEAR-1.0), E)             <- first argument is previous year
```

Note that the use of INT is always a bit risky. If a the result of a calculation is 12.0, there is always the possibility that the machine code for that number is something like 11.9998. Truncation with INT then leads to the integer value 11. Therefore, the use of NINT (Nearest INTeger), is generally safer than the use of INT.

Subroutines producing INTEGER output cannot be called directly from FST. They have to be changed or called via an interface routine (cf. Section 5.4).

## 6.1.8 Using the loop counter I

A calculation or FINISH statement which is expanded is placed by the translator in a Fortran DO-loop with the integer variable I as its counter. The range of I is the subscript range of the first expanded array in the statement (see Section 4.3.2). In order to show some possibilities of the use of I, some more examples are given in the following program lines.

```
DECLARATIONS
ARRAY X(1:N), Y(1:M)
DEFINE_CALL MYSUB (INTEGER_INPUT, OUTPUT)
...
MODEL
...
FINISH 2.0*X < REAL(I)        <- Expanded to N conditions 2.0*X(I) < REAL(I)
CALL MYSUB (I+1,X)            <- Expanded to N calls CALL MYSUB (I+1,X(I))
Y = 1.0+REAL(I**2)            <- Expanded to N calculations Y(I) = 1.0 + REAL(I**2)
```

With the first FINISH statement, N comparisons are made. For each array element, 2.0*X(I) is compared with the value of I itself. The is called N times, for all elements of X separately and the subscript number I is used in the input argument for the subroutine. The assignment calculates the elements of array Y as the series 1, 5, 10, 17, 26, 37, 50, ..., 1+M*M.

## 6.2 Cumulative rainfall

The program in Listing 6.3 makes use of the easy access to weather data in FST for calculating the yearly rainfall. This is done by using RAIN as a rate of change. The little tricks of Section 6.1.3 and Section 6.1.4 are used to get the start year and the total rainfall as terminal output.

Listing 6.3 Monthly averages of weather data

```
0001  TITLE Cumulative rainfall
0002  MODEL
0003  WEATHER WTRDIR='C:\SYS\WEATHER\'
0004  WEATHER CNTR='NLD' ; IYEAR=1954 ; ISTN=1
0005  TIMER STTIME=1.0 ; FINTIM=370.0 ; DELT=1.0
0006  TRANSLATION_FSE
0007  INITIAL
      * save start year as YEAR1 and stop at end of the (leap) year
0008  YEAR1=YEAR
0009  FINISH YEAR>YEAR1

0010  DYNAMIC
      * integrate rainfall starting at 0.0
0011  CRAIN = INTGRL(ZERO,RAIN)
0012  INCON ZERO=0.0

0013  TERMINAL
      * terminal help variables are copies from start year and total rain
0014  CR = CRAIN
0015  YR = YEAR1
0016  PRINT YR,CR
0017  END
```

The start year selected in statement 4 is 1954. The FINISH condition in statement 9 will halt the simulation at the first day of 1955 (see Section 6.1.3). The rainfall of the last day of 1954 has then been added tot the total and CRAIN is at its final value. Copies of YEAR1 and CRAIN are used for terminal output (see Section 6.1.4). By means of reruns over IYEAR, a graph can be made of the total rainfall as function of the year number.

## 6.3 Monthly means of weather data

The example program in Listing 6.4 on page 74 also integrates weather data, but now per month, by switching on and off an array of 12 rates of change using the technique described in Section 6.1.6.

Listing 6.4 Monthly averages of weather data

```
TITLE Monthly Averages of Weather data
DECLARATIONS
* cumulative radiation, minimum & maximum temperature per month,
* rates of change and monthly averages
ARRAY  CRAD(1:M),  RMRDD(1:M),  MRADM(1:M)
ARRAY CTMMX(1:M), RMTMMX(1:M), MTMMXM(1:M)
ARRAY CTMMN(1:M), RMTMMN(1:M), MTMMNM(1:M)
* help variables
ARRAY MDAYS(1:M) , XDAYS(0:M) , CDAYS(0:M)
ARRAY PULSE(1:M)

MODEL
ARRAY_SIZE M=12

WEATHER WTRDIR='C:\SYS\WEATHER\'
WEATHER CNTR='NLD' ; IYEAR=1954 ; ISTN=1
INCON ZERO=0.0
PARAMETER MDAYS(1:11)=31.,28.,31.,30.,31.,30.,...
                      31.,31.,30.,31.,30. ; MDAYS(12:M)=31.
INITIAL
* variable leap is 1.0 at leap years ; otherwise 0.0
LEAP = REAL((1+MOD(NINT(YEAR)-1,4))/4)

* make february longer when needed
XDAYS(0)=0.0 ; XDAYS(1)=MDAYS(1) ; XDAYS(2)=MDAYS(2)+LEAP ; ...
XDAYS(3:M)=MDAYS(3:M)

* calculate cumulative day numbers and start year
CDAYS = ARSUMM(XDAYS,0,I)
SYEAR = YEAR

DYNAMIC
* month pulse
PULSE = INSW((DOY-CDAYS(0:M-1)-0.5), 0.0, 1.0) * ...
        INSW((DOY-CDAYS(1:M)  -0.5), 1.0, 0.0)
* cumulative radiation switched on for current month only
CRAD  = INTGRL (ZERO, RMRDD)
RMRDD = PULSE * RDD
* cumulative maximum temperature switched on for current month only
CTMMX  = INTGRL (ZERO,RMTMMX)
RMTMMX = PULSE * TMMX
* cumulative minimum temperature switched on for current month only
CTMMN  = INTGRL (ZERO,RMTMMN)
RMTMMN = PULSE * TMMN

* stop at first day of next year (see Section 6.1.4)
FINISH YEAR > SYEAR

TERMINAL
* monthly averages to terminal output (See Section 6.1.3)
MRADM  = CRAD / XDAYS(1:M)
MTMMXM = CTMMX / XDAYS(1:M)
MTMMNM = CTMMN / XDAYS(1:M)

* get year to terminal output (see Section 6.1.4)
Y = SYEAR

PRINT Y, MRADM, MTMMXM, MTMMNM
TIMER STTIME=1.0 ; FINTIM=400.0 ; DELT=1.0
TRANSLATION_FSE
```

```
0036    END
```

In statement 16, the help array XDAYS is calculated as a copy of MDAYS, except for February which may have 28 or 29 days. In statement 17, the cumulative number of days is calculated as an array containing 0., 31., (31.+28.(or 29.)), (31.+28.(or 29.)+31.),...

During the simulation, the value of DOY (Day Of Year, see Section 3.2.4.4) proceeds from 1.0 to 365. (366.). For each day, statement 20 is evaluated. In expanded form, this statement is

```
      DO I=1,M
         PULSE(I) = INSW((DOY-CDAYS(I-1)-0.5), 0.0, 1.0) * ...
                    INSW((DOY-CDAYS(I)-0.5), 1.0, 0.0)
10    CONTINUE
```

For a value of DOY in, say March, the first INSW call returns 1.0 for I=3,4,...12. The second INSW call returns 1.0 for I=1,2,3. Hence, the product of the two INSW calls is 1.0 for I equal to 3 only, which makes the 3-rd element of PULSE equal to 1.0 and all other elements equal to 0.0. This is used in statements 22, 24 and 26 to switch on all integrals for March and to switch off all others. This logic may be verified by printing the array PULSE.

Using the 12 switches, the radiation, minimum temperature and maximum temperature are integrated in twelve parts. In the terminal section, the result is then divided by the lengths of the months.

Note that FINTIM has been set to 400 and that the run is terminated at the end of the year by means of the finish condition on SYEAR discussed in Section 6.1.4. The output variables are TERMINAL variables sent to the output files after terminating the run, at the first day of the next year. In order to have also the start year available at the same value of TIME, SYEAR is copied to the terminal variable Y (see Section 6.1.4). Figure 8.1 on the title page of Chapter 8 shows some results for January.

## 6.4 Plotting in state space

Usually, the result of a simulation model is one or more state variables that change with time. This can be visualized by plotting state variables with TIME as independent variable. In some situations, however, insight in the behaviour of a system can be gained by making a different kind of graph, by plotting the status of the system in state space.

The status of a system is described by the values of its state variables. If a system has two state variables X1 and X2, its status is completely described by a combination (X1,X2). Every value combination (X1,X2), however, corresponds to a point in the state space spanned by an axis X1 and an axis X2. The simulated changes in the system can be visualized then as a trajectory through the state space. An example is the biological population model in Listing 6.5 on page 76.

Listing 6.5 A predator prey model leading to so-called stable limit cycles.

```
0001    TITLE Predator Prey Model with stable limit cycle

        * This Predator prey system leads to a stable limit cycle
        * if the environment carrying capacity for the prey is large.
        * The model originates from Rosenzweig (1972) and is
        * discussed also by May (1972).

        * A stable limit cycle means that, after a sufficient amount
* of time, the periodic behaviour of predator and prey populations
* becomes independent of the initial conditions. This can be
* demonstrated by drawing the populations in (PREY,PRED) space.

* model parameters ; values taken from Roughgarden (1979)
* R - relative growth rate prey at low density
* K - carrying capacity of environment for prey
* A - consumption of preys per prey per predator at low prey density
* C - consumption of preys per predator at high prey density
* B - predator's increase per eaten prey
* D - relative death rate predator

MODEL
PARAM R = 0.5; A = 0.01; C = 10.0; B = 0.02; D = 0.1; K = 3500.
* start outside cycle
INCON IPREY = 200.0; IPRED = 15.0

PRED = INTGRL (IPRED,RPRED)
PREY = INTGRL (IPREY,RPREY)

* the growth rate of the preys with self limitation
RPREY = R * PREY * (1.0 - PREY/K) - FOUND * PRED

* the growth rate for the predators with efficiency factor B
RPRED = (B * FOUND - D) * PRED

* the number of preys eaten per predator per unit of time ; the
* exponential function describes the saturation of predators
FOUND = C * (1.0 - EXP (-1.0 * A * PREY / C))

PRINT  PREY, PRED
TIMER  STTIME=0.0; FINTIM=200.0; DELT=1.0
TIMER   PRDEL=0.5

TRANSLATION_GENERAL DRIVER='RKDRIV'
END
* start inside cycle
INCON IPREY = 700.0; IPRED = 50.0
END
```

The biology of the model is explained in Listing 6.5. The status of the system is completely described by a combination (PREY, PRED). Figure 5.1 on the title page of Chapter 5 shows the changing system status in state space. Starting from different points in state space, the system slowly approaches the same cycle, which is therefore called a stable limit cycle.

## 6.5 "Measuring" model properties

If the qualitative behaviour of a model is well understood, it becomes important to quantify this "understanding". For instance, a system appears to oscillate. If the FST program could determine the size of the oscillation period T, the behaviour of T could be studied as function of model input variables. It would be cumbersome to extract this information from plots of an oscillating state variable as function of time.

In the model of Listing 6.6 on page 78, the movement of a planet around its sun is described by Newton's laws of mechanics and gravitation. Positions and velocities are the state variables of the system. Their rates of change are velocities and accelerations as discussed briefly in Section 3.5. Here, the calculations are done in the (X,Y) plane and positions, velocities and accelerations are arrays with two components.

Statement 29 is the last statement of the actual mechanical model. The equations lead to the movement of the planet around its sun in an ellipse. The other calculation statements “measure” the length of the planets year.

Listing 6.6 Movement of a planet around a star according to Isaac Newton. A nice introduction into mechanics can be found in Feynman *et al.* (1963, Chapters 7, 8 and 9).

```
0001   TITLE   Orbits of sun and planet
0002   TITLE calculated with Newton's law

0003   DECLARATIONS
       * sun
0004   ARRAY  SUN(1:N),   ISUN(1:N), RSUN(1:N)
0005   ARRAY VSUN(1:N), IVSUN(1:N), ASUN(1:N)
       * planet
0006   ARRAY  PLN(1:N),   IPLN(1:N), RPLN(1:N)
0007   ARRAY VPLN(1:N), IVPLN(1:N), APLN(1:N)
       * unit length vector pointing from sun to planet
0008   ARRAY E(1:N)

0009   MODEL
       * calculations are 2-dimensional
0010   ARRAY_SIZE N=2
0011   CONSTANT PI=3.14159265

       * initial position and velocity of planet relative to center of mass
0012   INCON  IPLN(1) = 100.0 ;  IPLN(2:N) = 0.0
0013   INCON IVPLN(1) = 0.0   ; IVPLN(2:N) = 1.0 ; ZERO = 0.0

       * value of gravitation constant ; sun and planet mass
0014   PARAMETER G=2.0 ; MSUN=100.0 ; MPLN=1.0

0015   INITIAL
       * Initial position and velocity of the sun are calculated in
       * such a way that the center of mass (at 0,0) does not move
0016   ISUN  = -MPLN *  IPLN / MSUN
0017   IVSUN = -MPLN * IVPLN / MSUN

0018   DYNAMIC
       * equations of movement for the sun
0019   SUN  = INTGRL ( ISUN,RSUN)
0020   VSUN = INTGRL (IVSUN,ASUN)

       * equations of movement for the planet
0021   PLN  = INTGRL ( IPLN,RPLN)
0022   VPLN = INTGRL (IVPLN,APLN)

       * unit length vector pointing from sun to planet
0023   E = (PLN-SUN) / DIST

       * the acceleration of the sun and planet ; Newton's 2-nd law
0024   ASUN = +FORCE / MSUN * E
0025   APLN = -FORCE / MPLN * E

       * size of force between planet and sun ; Newtonís gravitation law
       * with dist as the distance between planet and sun
0026   FORCE = G * MSUN * MPLN / DIST**2
0027   DIST  = SQRT((SUN(1)-PLN(1))**2 + (SUN(2)-PLN(2))**2)

       * rates of change of position are velocities, by definition
```

```
RSUN = VSUN
RPLN = VPLN

TIMER STTIME = 0.0; DELT = 0.01; FINTIM=1.0E+5; PRDEL=100.0
PRINT SUN,PLN,SURF,ANGLE
TRANSLATION_GENERAL DRIVER='RKDRIV' ; TRACE=2

* Observing model behaviour
* ========================
* This model clearly has to obey Keppler's famous second law
*   "The radius vector of the planet sweeps out equal areas
*              in equal intervals of time"
* The swept area per unit of time is found as a vector cross product
SURF = 0.5 * (PLN(1)*VPLN(2) - PLN(2)*VPLN(1))

* angular velocity of the planet seen from the center of mass
OMEGA = 2.0 * SURF / (PLN(1)**2 + PLN(2)**2)
ANGLE = INTGRL(ZERO,OMEGA)
* the variable BACK falls to zero if the planet is back
BACK  = INSW(ABS(ANGLE)-2.0*PI,1.0,0.0)
* back is integrated
PERIOD = INTGRL(ZERO,BACK)

FINISH BACK<1.0
TERMINAL
T = PERIOD
PRINT T
END
```

The length of the planets year is found by calculating the angular velocity of the planet as seen from the center of mass. This angular velocity OMEGA is nothing else than the rate of change of the angle between the planet and some reference direction. OMEGA is calculated from the planet's position and velocity by means of some vector calculus. As soon as the integral of OMEGA over time reaches the value 2.0*PI, the planet has returned to its initial position. At this moment, BACK switches from 1.0 to 0.0. The state variable PERIOD is then equal to the simulated time so far and will not increase further. This formulation has the advantage that the Runge-Kutta algorithm detects the sudden change of BACK and will reduce its time step towards the end of the year in order to preserve the accuracy of the integral.

Note that the values of the gravitation constant and the masses are not realistic. The parameter values used in Listing 6.6 lead to a PERIOD of 246.73 time units reached in 35 integration steps.

Running the model shows that SURF is independent of time. This is the second of Kepler's laws on planetary movement: "The radius vector of the planet sweeps out equal areas in equal intervals of time". Kepler's incredible analysis was based on a lifetime of observations by Tycho Brahe in Denmark. With Newton's laws it becomes a direct consequence of the conservation of angular momentum.

The variables added for calculating the period are not part of the system description. They are used only for observing the simulated system. This example illustrates that it may sometimes be difficult to use a numerical simulation model for understanding a system. Just simulating system behaviour may not be sufficient.

## 6.6 Sensitivity analysis

This example demonstrates how a series of reruns in combination with terminal output can be used to describe the response of a system on the value of an input parameter.

The model in Listing 6.7 on page 80 describes an damped oscillator on which a periodic force acts. The period of the oscillation is equal to the period of the force, but the oscillation becomes large only if the period of the force is close to the natural period of the oscillator. This phenomenon is called resonance.

Listing 6.7 Forced damped oscillator

```
TITLE Movement of Forced Damped Oscillator

* A harmonic oscillator is brought into movement by an
* external sinusoidal force with period PERIOD. If the
* external period is close to the oscillator time, the
* amplitude reaches large values. There is resonance.
* The oscillator is damped by means of a friction force
* proportional to velocity.
* The parameters of the model:
* MASS   - mass of harmonic oscillator
* PERIOD - period of external sinusoidal force on oscillator
* K      - ratio elastic force and displacement X
* B      - friction constant: ratio between force and velocity
* F0     - amplitude of driving force

MODEL
PARAMETER MASS=1.0 ; PERIOD=80.0 ; K=1.0 ; B=0.1 ; F0=1.0

* Stationary oscillation is reached for 20 times the period
TIMER STTIME=0.0 ; FINTIM=1600.0 ; DELT = 0.01

* A constant
CONSTANT PI=3.14159265

* Initial value used for all state variables
INCON ZERO = 0.0

* Integration method with some extra accuracy
TRANSLATION_GENERAL DRIVER='RKDRIV' ; EPS=1.0E-5

INITIAL
* angular frequency related to period
OMEGA = 2.0*PI / PERIOD

DYNAMIC
* position and velocity are the state variables
X = INTGRL (ZERO,RX)
V = INTGRL (ZERO,RV)

* velocity changes as a result of acceleration
RV = A
* position changes as a result of velocity
RX = V

* acceleration is calculated from external force FORCE
* elastic force K*X and oscillator friction B*V, using Newton's law F=M.A
A = (FORCE - K * X - B * V) / MASS
FORCE = F0 * COS(OMEGA * TIME)
```

```
     * moved distance in last period of simulation is the
     * integral of the absolute speed switched on a time
     * PERIOD before TIME reaches FINTIM.
0017 S  = INTGRL(ZERO,RS)
0018 RS = INSW(TIME-(FINTIM-PERIOD),0.0,1.0) * ABS(V)

0019 PRINT X

0020 TERMINAL

     * terminal variable equal to angular frequency
0021 OMEG = OMEGA
     * a quarter of S should be close to the amplitude
0022 SIML = S / 4.0
     * analytical formula (not derived in this manual)
0023 ANAL = F0 / ...
        SQRT( MASS**2 * (K/MASS-OMEGA**2)**2 + OMEGA**2 * B**2)

     * terminal output
0024 PRINT OMEG,SIML,ANAL

0025 END
0026 PARAMETER PERIOD=40.
0027 TIMER FINTIM=800.0
0028 END
0029 PARAMETER PERIOD=20.
0030 TIMER FINTIM=400.0
0031 END
     .....
0098 PARAMETER PERIOD=1.4
0099 TIMER FINTIM=28.0
0100 END
0101 PARAMETER PERIOD=1.3
0102 TIMER FINTIM=26.0
0103 END
```

The state variables of the oscillator are its position X and its velocity V. The rates of change of the velocity is equal to the acceleration (sum of forces divided by mass) and the rate of change of X is equal to the velocity. The statements 11 to 16 represent this mechanical system.

The next step is to "measure" the amplitude of the oscillation. After a sufficient amount of time, the amplitude of the oscillator approaches a constant. In Listing 6.7, FINTIM is set equal to 20 times the period of the external force. The amplitude is measured during simulation of the last period. That is done in statements 17 and 18. During the last period the absolute velocity is integrated. This leads to an integral equal to four times the amplitude, independent of the phase of the oscillation. This is used in the terminal section for the calculation of the simulated amplitude SIML in statement 22.

In the terminal section, also an analytical expression for the amplitude is evaluated and a copy OMEG of the angular frequency OMEGA is made. The three variables OMEG, ANAL and SIML are terminal output variables. By means of a series of reruns, a graph of the amplitude as function of the OMEGA can now be directly made. Figure 6.1 at the title page of this Chapter 6 shows the result.

The strategy for such a sensitivity analysis is to represent the system behaviour by one or a few terminal output variables. By means of many reruns these output variables are studied as function of certain input values. It is handy to make copies of these input values in the terminal section. Then input and output variables appear in the result at the same value of TIME.

## 6.7 The use of gridpoints in a diffusion model

The diffusion problem of Section 4.2 on page 43 is treated in a somewhat different way. Again, the sheet size is divided in N layers, but in Listing 6.8 the N+1 concentrations at the layer interfaces are used as state variables instead of the amounts presents in each layer. This simplifies the formulation of the rates of change, but complicates the formulation of a mass balance and the role it plays in the program.

In statement 4 the concentrations, their rates of change and their initial values are declared as arrays with N+1 elements. In statement 13, the initial concentrations at the sheet surface (at grid pints 1 and N+1) are set at the outer concentration CS. All interior gridpoints have a zero initial concentration. The diffusion equation is very simple now: statement 16 keeps the concentration at the sheet's surface at CS (rate 1 and rate N+1 are always zero) and for all interior gridpoints the rate of change is equal to the second order spatial derivative. This is a direct implementation of the second order partial differential equation describing diffusion:

$$\frac{\partial c}{\partial t} = \frac{\partial^2 c}{\partial x^2}$$

So far the program looks attractive by its simplicity. The calculation of a mass balance, however, makes things more complicated. We need an estimate for the surface flux into the sheet. This surface flux is the diffusion coefficient times the concentration gradient at the surface. This gradient cannot be directly estimated as a concentration difference since there is no gridpoint present outside the sheet. Therefore, the gradient at point N+1, for instance, is calculated as the gradient at point N plus the change in the gradient between N and N+1. This change is the second order spatial derivative at point N times the layer size. Listing 6.8 contains comment statements in which this has been worked out. The total surface flux consists then of the flux at point 1 and the flux at point N+1.

In statement 20 the surface flux is integrated in order to get the cumulative uptake of the sheet as CFLUX. This uptake is also calculated as the contents of the sheet found by integrating the concentration profile with Simpson's method (statement 18). Finally, statement 22 evaluates the mass balance.

Listing 6.8 The diffusion problem of Section 4.2 on page 43 is treated here in a different way. Instead of layers with amounts as state variables, gridpoints are considered with concentrations as state variables.

```
0001    TITLE         Elementary description of diffusion
0002    TITLE   Two-sided sheet, initially at zero concentration

0003    DECLARATIONS
        * for N layers and N+1 gridpoints
0004    ARRAY C(1:N+1), RC(1:N+1), IC(1:N+1)

0005    MODEL
        *****
0006    PARAM   HEIGHT=1.0 ; D=0.01 ; CS = 100.0
0007    TIMER   STTIME = 0.0 ; FINTIM=40.0 ; DELT = 1.0 ; PRDEL=0.5
0008    PRINT   CHECK, SFLUX, RELUPT, CFLUX
0009    TRANSLATION_GENERAL DRIVER='RKDRIV' ; TRACE=2
0010    ARRAY_SIZE N=40

0011    INITIAL

        * height of compartment
```

```
DEL = HEIGHT / REAL(N)

* set initial concentrations
IC(1)=CS ; IC(2:N)=0.0 ; IC(N+1)=CS

DYNAMIC
C = INTGRL(IC,RC)

* second derivative for interior points ; zero at edge
RC(1)   = 0.0 ; ...
RC(2:N) = D * (C(1:N-1) - 2.0*C(2:N) + C(3:N+1)) / DEL**2 ; ...
RC(N+1) = 0.0

* surface flux at two sides added
* second order estimates
* dc/dx(N):            (1.0*C(N+1)          -1.0*C(N-1)) / (2.0*DEL)
* d2c/dx2(N) * DEL:    (2.0*C(N+1)-4.0*C(N)+2.0*C(N-1)) / (2.0*DEL)
*                      ------------------------------------------- +
* dc/dx(N+1):          (3.0*C(N+1)-4.0*C(N)+1.0*C(N-1)) / (2.0*DEL)
*
* and the same for the other edge at point 1
*
SFLUX = D * (3.0*C(1)   -4.0*C(2) +1.0*C(3)   ) / (2.0*DEL) + ...
        D * (3.0*C(N+1) -4.0*C(N) +1.0*C(N-1)) / (2.0*DEL)

* contents of sheet
UPTAKE = ARSMPS(C,1,N+1,DEL)
RELUPT = UPTAKE / (HEIGHT*CS)

* cumulative surface flux
CFLUX  = INTGRL (ZERO,SFLUX)
INCON ZERO=0.0

* balance check
CHECK  = (UPTAKE - CFLUX) / (HEIGHT*CS)

END
```

The mass balance of the program in Section 4.2 (Listing 4.1 on page 44) is nonzero only in case of a programming error. If the program is correct, the mass balance must be zero since mass transport is described as amounts going from one layer into another layer. In Listing 6.8, a non-zero mass balance also points towards an accuracy problem. Integration errors lead to a (small) deviation from a zero mass balance, also in case of a correct model.

The use of gridpoints in this way is known as the method of lines. The calculation of first and second order spatial derivatives is an art of its own and is preferably done from FST with subroutine calls (e.g. Silebi & Schiesser, 1992). The use of gridpoints enables a straightforward implementation of the diffusion equation. This is an advantage, especially in case of a cylindrical or spherical geometry in combination with several source and sink terms. The use of the mass balance for finding an optimal number of gridpoints is also an attractive aspect. The model gets a more mathematical and less intuitive character, however, and will always have a nonzero mass balance, which may be a disadvantage in certain applications and in teaching.

# Chapter 7

# User Guide

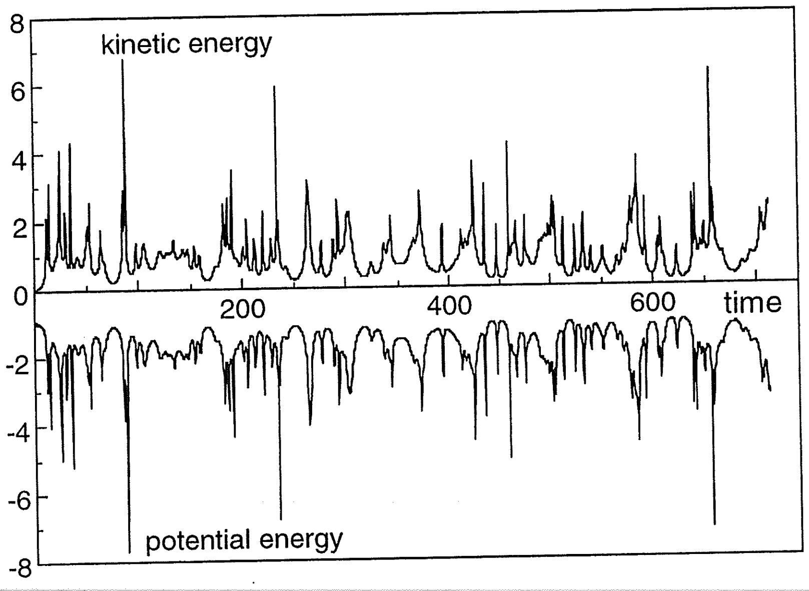


*Figure 7.1 on the title page of this chapter.* The total kinetic and potential energy of a group of 10 stars in their own gravitational field. The orbits of the stars are chaotic and extremely dependent on the initial conditions. The system is therefore no longer predictable. Statistical averages replace the predictable elliptic orbits of a two body system. According to the virial theorem the average kinetic energy is minus half the average potential energy. The FST program is available from the authors.

# 7 User guide

## 7.1 The file structure of a model run

Figure 7.2 on page 88 gives a diagram of the computer operations during the execution of an FST model. The FST program is on top of the diagram. It is usually written with help of a program editor. If a translation command is given, the FST program is read by the FST translator. The translator may produce a listing of the FST program. If there are no errors in the FST program, a Fortran program MODEL.FOR is generated with a set of data files belonging to it.

The next step is to compile the Fortran program. The result is machine code stored in MODEL.OBJ. The Fortran program makes use of the library DRIVERS (containing the simulation drivers), the library TTUTIL (containing standardized procedures for input and output), and the library WEATHER for access to the weather data files. The Linker takes the required code from the libraries and writes an executable program MODEL.EXE.

During the execution of MODEL.EXE, the generated data files are read. The input subroutines produce temporary files with extension .TMP which are not included in Figure 7.2. Sometimes you may find them on the working directory. They can be deleted. Except these temporary files, there are 4 real output files. The file RES.DAT contains the simulation results requested by the PRINT and OUTPUT statements of the FST program. The log files MODEL.LOG and WEATHER.LOG contain remarks and messages. And finally, RES.BIN is a machine readable version of RES.DAT which is used on some machines to produce graphical output.

This shows that the execution of an FST model requires 4 steps: translation, compilation, linking and running. The first three steps are taken by the FST translator, the Fortran compiler and the Fortran linker. The last step is the actual model execution. Some insight of this procedure makes it a lot easier to understand the installation of FST and to understand the difference between translation errors, compilation errors, linker errors and runtime errors. Below these various aspects are further discussed.

## 7.2 Installation of FST

This text is meant to give some insight into the installation of FST. The actual installation instructions come with floppy and on floppy disks.

The FST translator itself consists of a single executable file. All error messages and also the generated Fortran code are contained in this executable file FST.EXE. This makes a raw installation of FST a trivial thing to do: the file FST.EXE has to be copied to a disk. An FST model may then be translated by just typing in a translation command which starts up the translator (see Section 7.3). According to the scheme in Figure 7.2, this leads to a Fortran program with data files. The rest of the procedure remains to be done "by hand".

Clearly, on most computers an often repeated sequence like translation, compilation, linking and running can be automated. This has been done for PC and Macintosh, making use of a certain compiler.

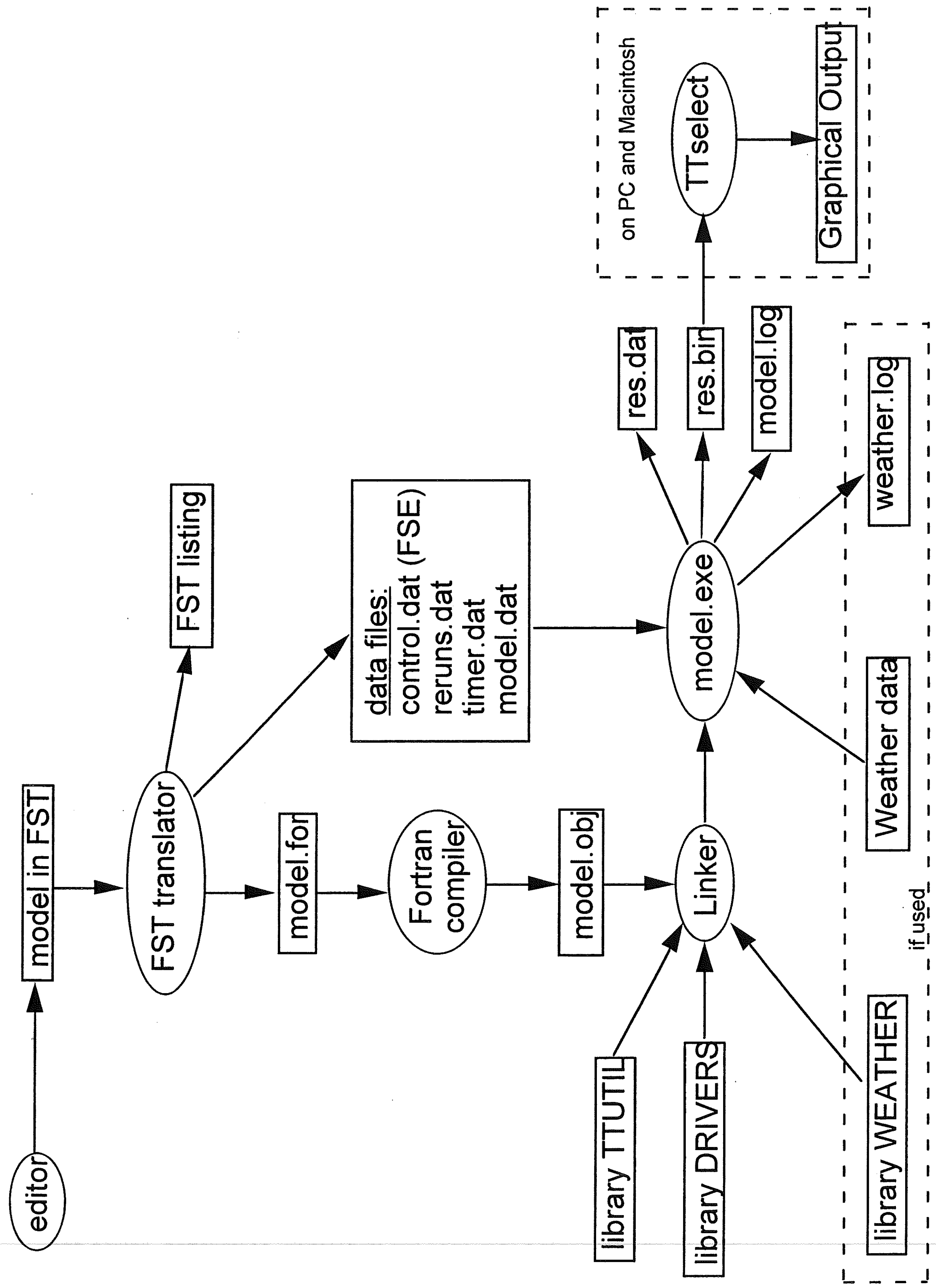


Figure 7.2 Overview of programs (ellipses) and files (rectangles) involved in an FST model run.

## 7.2.1 Automated model runs on PC

For use in combination with Microsoft Fortran 5.1, there is a small shell available which reduces the entire procedure of running an FST model to "press R".

The shell is started by typing under DOS the command FSTS (FST Shell). After typing "E", the shell executes the program editor of your choice for editing an existing or new FST model. Quitting the editor brings you back to the shell menu from which the execution sequence can be started. The linker makes use of the Fortran object libraries DRIVERS, TTUTIL and WEATHER which have been made with Microsoft Fortran 5.1.

After model execution also the graphical utility program TTSELECT can be used from the shell in order to quickly produce graphs at screen resolution. This graphical utility is able to combine the results of several reruns in a single graph.

If you use another Fortran compiler, the translator FST.EXE can still be used, but the source code of the three linked libraries needs to recompiled.

## 7.2.2 Automated model runs on Macintosh

The FST translator is available in 4 forms.
- Application for 68k Macintosh machines, running in about 1.2 Mb RAM
- MPW tool for 68k Macintosh machines, running in MPW application memory
- Native code application for Power Macintosh, running in about 1.7 Mb RAM
- MPW native tool for Power Macintosh, running in MPW application memory

The compiler we use on Macintosh is Language Systems Fortran 3.3 running under MPW. On a Powermac we use a PPC native code version of that compiler, also from Language Systems. For both an 68k Mac and a Powermac, compiled libraries DRIVERS, TTUTIL and WEATHER are available.

For this environment, MPW plus Language Systems Fortran 3.3, we wrote a few MPW script files for automated model runs. Two Startup scripts are used to install additional Fortran, library and FST commands under the MPW menu. FST program files can be edited with the MPW editor or with another one. Selecting then the "Run FST" menu item leads to the following sequence:
- The current directory is searched for .FST files. If there is more than one, a choice has to be made by means of an item selector. A single FST file is automatically selected.
- The selected .FST file is translated with the MPW tool FST (which may be the 68k or PPC version). Error messages appear in the worksheet window.
- An MPW script LSF.INI is generated for compilation, linking and running with the selected compiler options. The compiler options include processor choice (68k or PPC) and linker mode (application or MPW-tool).
- The generated MPW script is executed.

This procedure runs an FST model, but the results have to be visualized and analyzed by means of other programs. Although the graphical utilities used on a PC are also available on the Macintosh, they have not yet been integrated in the MPW scripts.

If you have the model linked as an MPW-tool, also the model is executed in MPW memory and no additional RAM memory is needed. If you link the translated model as application, the memory required by the model is claimed separately.

Once you have available the generated script LSF.INI, a quick method for redoing all this is executing FST in the MPW worksheet without command line (which re-translates the same FST model) and typing Command-0 (the 0 on the keypad), which re-starts the existing script file LSF.INI.

If you use another Fortran compiler, the translator application FST.EXE can still be used, but the source code of the three linked libraries needs to recompiled.

### 7.2.3 Other platforms

As mentioned above, the translator itself is a single executable file which translates an FST model into Fortran. In principle, the FST translator can be produced for all platforms for which a good Fortran 77 compiler is available. The translator itself has also been written in Fortran 77 with use of a few extensions like the INTEGER*2 variable type, the IMPLICIT NONE statement and the INCLUDE statement. Most Fortran compilers accept these extensions.

For VAX/VMS an application FST.EXE is available. Translation, compilation and linking of an FST model can be organized in an easy to write command file. We probably will also port the translator to UNIX. On these machines the installation of FST remains to be a "raw translator installation" for the generation of Fortran for FST. Integration with graphical utilities will not be available.

## 7.3 Translator commands

The FST translator works on the basis of a command line. A command line consists of names of FST files and options. The command lines are almost the same on the different computers. On a DOS machine, the command for translating ORBIT.FST and producing an FST listing is

```
C:\> FST ORBIT /LIST
```

Note that the translator FST.EXE needs to be on the current directory or somewhere in the path. On the Macintosh, options begin with a hyphen "-" instead of a slash "/". The translator application FST.EXE comes with a window and a prompt FST> after which the command line is typed in. We get then:

```
FST> ORBIT -LIST
```

Using the MPW tool on a Macintosh, the command typed is

```
FST ORBIT -LIST
```

Note that several filenames may be given and that file names and options may be given in any order. For instance, the command for producing a full listing of ORBIT.FST, NEW.FST, CROP.FST and WEED.FST is, on a PC

```
C:\> FST /LIST /SYMLIST /STATLIST ORBIT NEW CROP WEED /NOGENERATE
```

This command produces the listing files ORBIT.LIS, NEW.LIS, CROP.LIS and WEED.LIS and does not produce any Fortran files. The generation of Fortran would not make sense because the files produced for ORBIT, NEW and CROP would be overwritten by the ones generated for WEED.FST. Therefore, the Fortran generation is suppressed by means of the option /NOGENERATE. Note that options need not to be separated by a space. One may write /LIST/STATLIST, for instance.

In the FST application window on the Macintosh, the above command looks like

```
FST> -NOGENERATE -LIST -SYMLIST -STATLIST ORBIT NEW CROP WEED
```

and the equivalent MPW command using the tool FST is

```
FST -NOGENERATE -LIST -SYMLIST -STATLIST ORBIT NEW CROP WEED
```

The options in Macintosh command lines must be separated by a space.

## 7.3.1 The translator options

As shown above, options in the command line are preceded by a slash "/" on PC and a hyphen "-" on Macintosh. On other platforms at least one of these two possibilities will work. The available options are

| | | |
|---|---|---|
| LIST | NOLIST | with default NOLIST |
| SYMLIST | NOSYMLIST | with default NOSYMLIST |
| STATLIST | NOSTATLIST | with default NOSTATLIST |
| BATCH | NOBATCH | with default BATCH on Macintosh and NOBATCH elsewhere |
| GENERATE | NOGENERATE | with default GENERATE |

The option /LIST produces a statement listing of the FST program with error messages and warnings. The option /SYMLIST produces a symbol list with variable names, their FST variable types and the statement numbers in which they are defined and used. The option /STATLIST produces a list of state variables, their rates of change and initial values. All three listings are written to the same file with extension .LIS. Listing 7.1 on page 93 is an example of a full listing.

The /NOGENERATE option skips the generation of Fortran and datafiles. This is used if the translator should only verify one or more FST programs. If different FST program names are supplied, the /NOGENERATE option makes sense because the generated Fortran and data files would be overwritten for each model. The /NOBATCH option holds the screen and asks for a <Return> to continue in case there is much screen output from the translator (error messages). On a Macintosh the output appears in a window which can be scrolled and the default there is /BATCH.

A special option is /HELP or -HELP. This produces a list of available options, together with the defaults for the machine you work on. Also the commands "`FST -?`", "`FST /?`" or even "`FST ?`" will produce this list of options (except for the translator in the form of an MPW tool on Macintosh). The /HELP option suppresses all further processing of FST files. Clearly, /NOHELP is the default on all machines.

Without any options, the translator generates Fortran and does not produce a listing.

Options do not have to be typed in fully. There should be enough letters for an unambiguous interpretation by the translator. Instead of /LIST, for instance, just /L can be given. Typing /S, however, does not make clear what you want.

## 7.3.2 The file ABFORUTL.INI

The FST translator stores its last command line in a file named ABFORUTL.INI on the current directory. Each time the translator is started on that same directory it reads and rewrites this file. If the translator is started without any filenames or options, i.e. without a command line, the command stored in the file ABFORUTL.INI is repeated. This "INI file" is an ordinary text file which can be edited. It contains, for instance,

```
LATEST_FST_COMMAND: ORBIT /LIST
USER_FST_DEFAULTS :
```

Behind "USER_FST_DEFAULTS: " one can type options to be used as a default (this overwrites the built in defaults). For instance, if one always wants a listing and a state variable listing, (but no symbol listing) the INI file should be

```
LATEST_FST_COMMAND: ORBIT /LIST            <- last command given
USER_FST_DEFAULTS : /LIST /STATLIST
```

The user defaults can still be overwritten by command line options.

## 7.4 The output of FST

The diagram in Figure 7.2 on page 88 shows that the translator produces a listing and generates Fortran (with data files). After compilation and linking, the model produces several output files.

### 7.4.1 The FST listing

In Listing 7.1 the full listing is given of the diffusion model in Listing 4.1 on page 44. Two errors have been deliberately added to the model: A typing error in the value of HEIGHT and the calculated variable VOL has been changed in VOLUM without adapting the references to VOL.

The first part is the statement listing produced with the option /LIST (or -LIST). A statement listing consists of:
- Global errors and warnings. These errors and warnings do not belong to specific statements. In Listing 7.1 there is a global warning on the use of the loop counter I.
- If there are any errors, there is a message telling you that no Fortran can be generated.
- The actual statement listing with numbered statements. The warnings and errors appear just below the statements to which they belong. In Listing 7.1 there is one error below statement 7.
- The errors found during a consistency check. These are messages on undefined variables, a lacking WEATHER statement, a lacking PRINT statement, variables used in INITIAL calculations and calculated in DYNAMIC, a lacking definition of DELT, etc. In Listing 7.1, VOL is undefined and VOLUM unreferenced.
- The number of errors and warnings found.

In listings warnings and errors can be found by searching for a percentage sign "%".

The second part is the state variable listing produced with the option /STATLIST (or -STATLIST). This is a list of state variables, their initial values and rates of change. Variables marked with a star "*" are array variables.

The third part is the symbol listing produced with the option /SYMLIST (or -SYMLIST). This symbol listing consists of 3 parts:
- Symbol list 1. A list of variables with values defined by the user. For each variable the section and statement of definition are given and the statements in which it is used. Variables marked with a star "*" are array variables. Note that the symbol listing in Listing 7.1 also contains the used, but undefined variable VOL. Its type is “???”.
- Symbol list 2. A list of symbols which can be used only (weather variables, driver supplied variables, Fortran and FST intrinsic functions).
- Symbol list 3. A list of declared and called subroutines (not present in Listing 7.1).

If there are any array variables in the program, an overview over the declared bounds is given behind the state variable listing and/or Symbol list 1. The listing file ends with a list of abbreviations used for the various variable types.

Listing 7.1 This is the full listing of the diffusion model in Listing 4.1 on page 44 with two errors deliberately added to the program. The listing has been produced by giving all listing options /LIST /STATLIST and /SYMLIST (or -LIST -STATLIST and -SYMLIST). Errors and warnings appear in the listing (and on screen).

```
*--------------------------------------------------------------------*
* General info about this file                                       *
*                                                                    *
* Contents      : FST translator listing                             *
* Creator       : FST translator version 2.0                         *
* Creation date : 9-May-1996, 22:26:26                               *
* Source file   : LIST41.FST                                         *
*--------------------------------------------------------------------*

The Fortran Simulation Translator (FST) translates an FST program
into a structured Fortran subroutine with data files containing timer
variables, model parameters and rerun specifications.

      ***              Warning: The loop counter I is used in an
                       expression or subroutine argument. The range of
                       values assigned to I is the range of the first
                       expanded subscript range in the substatement or
                       call, e.g. in "A(2:5)=0.3*REAL(I)*B(4:7)" the
                       counter I takes the values 2,3,4,5 and NOT 4,5,6
                       and 7. (%GLOBAL-RANGWARN).
      ***              Due to one or more errors in the program no
                       Fortran source and no data files are generated.
                       Also some checks on the values of some TIMER and
                       WEATHER variables could not be carried out.
                       (%GLOBAL-FINALS2).

0001  TITLE   Diffusion into a two-sided plane sheet
0002  DECLARATIONS
      * all arrays are declared ; N layers and N+1 fluxes
0003  ARRAY FLUX(1:N+1), X(1:N), C(1:N)
0004  ARRAY H(1:N), IH(1:N), NFLOW(1:N)
0005  MODEL
      *****
0006  INCON   IH=0.0
0007  PARAM   HEIGHT=1..0 ; D=0.01 ; AREA=1.0 ; CS = 1.0

      [GHT=1..0 ; D=], This point is illegal here. (%SYNTAX-REAL.PNT).
               ^

      * the number of layers
0008  ARRAY_SIZE N=60
0009  INITIAL
      * height of compartment
0010  DEL = HEIGHT / REAL(N)
      * volume of compartment
0011  VOLUM = DEL * AREA
      * compartment centres
0012  X = (REAL(I) - 0.5) * DEL
0013  DYNAMIC
0014  H = INTGRL(IH,NFLOW)
0015  C = H / VOL
```

```
       * the fluxes through the boundaries
0016   FLUX(1)    = D * (CS - C(1)) / (0.5*DEL) ; ...
       FLUX(2:N)  = D * (C(1:N-1) - C(2:N)) / DEL ; ...
       FLUX(N+1)  = D * (C(N) - CS) / (0.5*DEL)
       * the nett flow into the compartments
0017   NFLOW(1:N) = FLUX(1:N)*AREA - FLUX(2:N+1)*AREA
       * output and simulation control
0018   PRINT   X, C
0019   TIMER   STTIME = 0.0 ; FINTIM=20.0 ; DELT = 1.0 ; PRDEL=1.0 ; IPFORM=5
0020   TRANSLATION_GENERAL DRIVER='RKDRIV' ; TRACE=2
0021   END
0022   STOP
0023   ENDJOB

 %ERROR: Undefined variable(s):
         VOL    first used in statement  15

 %WARNING: Unreferenced variable(s):
           VOLUM  defined in statement  11 as Calculated variable

Translation completed with
2 errors
2 warnings


 List of STATE variables, their INITIAL values and RATES of change
 =================================================================
 Name     INTGRL    Initial value    Rate of change
 ------   -------   -------------    --------------
 H      *   14      IH     *   6     NFLOW  *  17

 - State variables are variables calculated with the INTGRL function.
 - The initial value of a state variable is either a calculated variable
   or an initial constant set by means of an INCON statement.
 - A rate of change may be a calculated variable or a variable made available
   by the WEATHER statement (weather and calendar data). For testing purposes,
   also a model PARAMETER is accepted as a rate of change.


 Symbol list 1: USER DEFINED VARIABLES
 =====================================
                   Defined        Referenced in
 Name     Type    Stmt Sect    Sect Out?  Statements (including use in reruns)
 ------   ------  ----------   -------------------------------------------------
 AREA     Param      7   I      ID.   .    11  17
 C      * Calcu     15   D      .D.   P     3  16  18
 CS       Param      7   I      .D.   .    16
 D        Param      7   I      .D.   .    16
 DEL      Calcu     10   I      ID.   .    11  12  16
 DELT     Timer     19   I      ...   .
 DRIVER   TrGEN     20   I      ...   .
 FINTIM   Timer     19   I      ...   .
 FLUX   * Calcu     16   D      .D.   .     3  17
 H      * Calcu     14   D      .D.   .     4  15
 HEIGHT   Param      7   I      I..   .    10
 IH     * Incon      6   I      .D.   .     4  14
 IPFORM   Timer     19   I      ...   .
 N        ArSiz      8   I      ID.   .     3   4  10  16  17
 NFLOW  * Calcu     17   D      .D.   .     4  14
 PRDEL    Timer     19   I      ...   .
 STTIME   Timer     19   I      ...   .
 TRACE    TrGEN     20   I      ...   .
 VOL       ???      --   -      .D.   .    15
```

```
VOLUM     Calcu      11   I        ... .
X       * Calcu      12   I        ...  P    3  18

  * = This is an FST array.
  The declared ranges are given in the following table.
                subscript bounds
  Array name    (Lower : Upper)
  ----------    ------   ----------
     C            (1   : N)
     FLUX         (1   : N+1)
     H            (1   : N)
     IH           (1   : N)
     NFLOW        (1   : N)
     X            (1   : N)


Symbol list 2: USE-ONLY SYMBOLS
===============================
Name      Type     Sect Out?   Statements
------    ------   ------------------------------------------------
I         DRsup     I..  .     12
INTGRL    FSFun     .D.        14
REAL      FFunc     I..        10  12


Abbreviations:
--------------
 ???  = Undefined variable
Calcu = Calculated variable
Incon = Initial constant
Param = FST Parameter
Const = FST Constant                         -----------------------
ArSiz = Array Size Variable                  ! I = INITIAL section  !
IntPL = Interpolation function               ! D = DYNAMIC section  !
UsSub = Called SUBROUTINE                    ! T = TERMINAL section !
Timer = TIMER variable                       ! P = PRINT or OUTPUT  !
Weath = WEATHER control variable             -----------------------
TrFSE = TRANSLATION_FSE variable
TrGEN = TRANSLATION_GENERAL variable
FFunc = Fortran Intrinsic function
FSFun = FST Intrinsic function
DRsup = Variable supplied by the driver
WSsup = Weather and calendar data
```

### 7.4.2 The generated Fortran with datafiles

The generated Fortran is not just a "dump" of the model in Fortran form. A well-structured and commented Fortran program is generated with a complete declaration section. All numerical values of FST parameters, initial constants, interpolation functions and simulation control variables are written to data files from which they are read back by the generated Fortran program. There are four files: MODEL.FOR, MODEL.DAT, TIMER.DAT and RERUNS.DAT. In case of translation in FSE mode there is a fifth file, CONTROL.DAT.

The technical structure of the generated Fortran is discussed in Chapter 9. Here, a brief description is given indicating the contents of each file.

### 7.4.2.1 MODEL.FOR

The first part of this file is the Fortran main program. The function of the main program is to call a simulation driver, either directly (in FSE mode) or indirectly (via subroutine RERUNS in GENERAL mode).

The Fortran main is followed by the actual model subroutine, which is used by the drivers. In FSE mode there is an intermediate subroutine MODELS in between the driver and the actual user model. This subroutine models may contain calls to various FSE modules. In case of an FST-generated program, however, there is only a single FSE module and the intermediate subroutine MODELS just passes the driver calls further down to the user MODEL.

The actual model subroutine, both in FSE and GENERAL mode, contains all calculations of the model in computational order. The three sections INITIAL, DYNAMIC and TERMINAL can be found back in the generated Fortran. Further, from each section, calls to output routines are made for writing values of output variables to the output file RES.DAT.

Especially the Fortran initial section contains other things than calculations. The initial section reads the values of model parameters, initial constants and interpolation functions from the data file MODEL.DAT. The output procedure is initialized and finally the state variables of the model are set at their initial values.

For the FSE translation mode, technical details can be found in van Kraalingen (1995). Chapter 9 gives details for GENERAL translation.

### 7.4.2.2 MODEL.DAT

This file contains the values of the model parameters, the initial constants and the interpolation functions. The file is in TTUTIL format, which implies that values can be read into Fortran programs by means of a few calls to subroutines from the TTUTIL library.

### 7.4.2.3 TIMER.DAT

This file contains the values of all simulation control variables. Also this file is in TTUTIL format. The file is somewhat different for the two translation modes since the simulation control variables for TRANSLATION_FSE and TRANSLATION_GENERAL differ from each other.

### 7.4.2.4 RERUNS.DAT

Values of input and simulation control variables in the rerun sections appear in the reruns file. This file is an application of the TTUTIL rerun facility. There is one important difference, however, between the rerun sections of the FST program and the so-called rerun sets of the generated rerun file RERUNS.DAT. This difference can be best explained by means of an example. Two reruns are defined:

```
MODEL
PARAMETER A=1.0
TIMER STIME=0.0
...
END
PARAMETER A=3.4                 <- first rerun
END
TIMER STTIME=10.0               <- additional change for second rerun
END
```

The first rerun section redefines parameter A and the second section redefines also timer variable STTIME. In FST, once a variable has been redefined, the new value is kept until it is changed again in a rerun section further down. Hence, the second rerun has to be done for A=3.4 and STTIME=10.0. In the file RERUNS.DAT, however, all reruns are defined relative to the standard run

## 7.5 Debugging an FST model

As shown in Figure 7.2 on page 88, the execution of an FST model requires translation into Fortran by the FST translator, compilation of the generated Fortran with a Fortran compiler, linking with DRIVERS, TTUTIL and WEATHER, and finally the actual execution of the model. In case of errors, the most important thing is to know during which part of the execution sequence things went wrong. The three sections below give some hints. In general however:

**READ THE ERROR MESSAGES**

### 7.5.1 Translation errors

Translation errors are the result of errors or inconsistencies in the FST program. Before starting the translation, the FST translator presents itself on the screen with a version number. Then error messages appear and finally the number of errors and warnings is given.

If there are more then one or two errors it is worthwhile to inspect the FST statement listing (see Listing 7.1 for an example). In that listing errors and warnings can be found by searching for a percentage sign %. If a large number if errors is reported, the first ones are usually the most important ones. The other ones may be the side effects resulting from previous errors and disappear if the first errors are corrected (See Chapter 8 for reference information about the FST syntax rules).

A special type of error is a sorting error. If statements cannot be brought in a computational order, the unsortable statements are marked with a cross "x" and on top of the listing a sorting error is reported for the INITIAL, DYNAMIC or TERMINAL section. Note that a sorting error may sometimes be the result of a wrong DEFINE_CALL statement. Note that wrong information on the Input/Output structure of a subroutine may lead to a sorting error. Hence, if a subroutine call is marked as unsortable, also the DEFINE_CALL statement for that subroutine should be inspected.

The number of errors given for a single statement is limited. Hence, it is possible that new errors are reported after correcting the first ones.

### 7.5.2 Compilation errors

The Fortran program generated by the FST translator should be free of Fortran errors. We have done everything we could to prevent Fortran being generated for a wrong FST program, since such a program is likely to lead to Fortran errors. Hence, in most cases, compilation errors will disappear if appropriate compiler options are used. Note that the generated Fortran is not completely compatible with standard Fortran 77. The exceptions are

- The IMPLICIT NONE statement in all generated subprograms, which forces the user to declare all variables added to the program.
- The calls to the error subroutine FATALERR of TTUTIL.

This implies that the generated Fortran should be compiled without standard switch. We regard remaining errors as bugs in the FST translator. If such errors occur, you are kindly requested to send the generated Fortran, the translator's version number and the FST model producing the error to the authors of this manual.

of the MODEL section. In case of the example above, the two rerun sections in RERUNS.DAT both contain values for A and STTIME. The first rerun is made for A=3.4 and STTIME=0.0 (the MODEL value) and the second rerun is made for A=3.4 and STTIME=10.0.

#### 7.4.2.5 CONTROL.DAT

In FSE mode this file contains the names of the model data file (MODEL.DAT), the simulation control file (TIMER.DAT), the rerun file (RERUNS.DAT), the output file (RES.DAT) and the logfile (MODEL.LOG). The FSE simulation driver reads these filenames from CONTROL.DAT. The TRANSLATION_GENERAL drivers use the same, but built-in filenames and do not read them from a control file.

### 7.4.3 Runtime output

The executed model typically creates three output files: RES.DAT, RES.BIN and MODEL.LOG. If weather data are used there is a fourth file, WEATHER.LOG. In case of unexpected termination of the model execution, there may be also files with extension TMP. These are temporary files created by TTUTIL library routines.

#### 7.4.3.1 RES.DAT

This file contains the output of the model with the output of reruns merged below each other in the file. The internal format of the output file RES.DAT depends on the value of the TIMER variable IPFORM written to the timer  file TIMER.DAT. If printplots are made with OUTPUT, they appear before the output tables. If the TIMER variable COPINF was set to 'Y', also copies of the input files will be present at the end of the RES.DAT output file.

#### 7.4.3.2 RES.BIN

This file is made during the simulation and contains in machine readable form all the values of the variables in the PRINT and OUTPUT statements. In fact, the tables of RES.DAT are based on the contents of RES.BIN. Some graphical utilities use RES.BIN instead of reading formatted values from RES.DAT. If you do not use such graphical utilities, RES.BIN can be deleted.

The file RES.BIN is also useful in case of a system crash during model execution, for instance, as a result of a divide by zero. No output file RES.DAT is available then since this file is produced during the terminal phase of the simulation. From RES.BIN a part of the output may be recovered, however. This may be useful for finding the reason of the crash.

#### 7.4.3.3 MODEL.LOG

This file may contain error and warning messages from routines used during the simulation. Messages about input variables whose values have been replaced by the rerun facility can be particularly useful. In case of problems it is a good idea to inspect also this file. In TRANSLATION_GENERAL mode the log file may also contain a detailed integration report written by the driver (see Section 7.5.5).

#### 7.4.3.4 WEATHER.LOG

This file contains all the messages generated by the weather system. By default, all the comment headers of the data files, all warnings and all errors from the weather system are written to this log file. If shortly before termination of the model a message is displayed about possible errors and warnings from the weather system one has to look into this file and interpret the messages. Messages may be as unimportant as rainfall not being available when it was not used by the model but they can also be of a much more severe type.

In case of problems one can begin with setting TRACE=2. This gives the number of steps between the successive output times. If the model "hangs" somewhere, it becomes at least clear during which time interval the problem occurs. A detailed report of the integration steps taken by the driver is generated with TRACE=3. Listing 7.2 gives an example of a report written by RKDRIV. It shows how the time step is adapted in order to maintain a certain integration accuracy.

The value TRACE=4, does not add anything if used from FST. If state events have been added to the generated Fortran, however, the driver produces a detailed iteration report.

Listing 7.2 Example of a log file MODEL.LOG. The log file contains an initialization report and a dynamic integration report. The latter contains a column with time steps, the reached TIME value and (with RKDRIV) a proposal for a next step. This example comes from a run with the model in Listing 6.6. It shows that the time steps taken by the driver vary with the position of the planet.

```
  Run    0                                                   <-basic run
  =======                                                    <-first rerun is run 1
Data file   TIMER.DAT with   11 variables parsed by RDINDX   <-main program reads data
  RKDRIV 2.0: Initialize model                               <-driver RKDRIV selected
Contents of TIMER.DAT recovered from TMP file                <-RKDRIV reads data
Data file   MODEL.DAT with    7 variables parsed by RDINDX   <-MODEL.FOR reads data
  Initialization of RKDRIV and user MODEL completed:
        --------------------------------------               <-simulation control
              Number of states:    10
        Integration starts at:   0.00000
        Output interval PRDEL:    100.00
           Finish time FINTIM:   1.00000E+05
            Relative accuracy:   1.00000E-04
         State event accuracy:   1.00000E-05
                 Maximum step:   1.00000E+05
        --------------------------------------

  RKDRIV 2.0: DYNAMIC loop                                   <-integration report
  ===========                                                  by driver
     time step            time       try as next step
   -------------------------------------------------
 Output flag set =====  0.00000                              <-dynamic output at
                                                               STTIME
    +  1.00000E-02 -->  1.00000E-02    ( 4.00000E-02)
    +  4.00000E-02 -->  5.00000E-02    ( 0.16000    )
    +  0.16000     -->  0.21000        ( 0.60515    )
    +  0.60515     -->  0.81515        (   2.4206   )        <-proposed step is taken
    +    2.4206    -->   3.2358        (   7.2895   )
    +    7.2895    -->   10.525        (   18.316   )
    +    18.316    -->   28.841        (   24.002   )
    +    24.002    -->   52.843        (   25.057   )        <-proposed step NOT
                                                               taken!
    +    15.423    -->   68.267        (   16.516   )
    +    12.211    -->   80.477        (   12.682   )
    +    10.098    -->   90.575        (   10.223   )
    +    9.4251    -->   100.00        (   10.223   )
 Output flag set =====   100.00                              <-at 1 x PRDEL
    +    6.0044    -->   106.00        (   6.0712   )
    +    6.0712    -->   112.08        (   7.0388   )
    +    5.2448    -->   117.32        (   5.5947   )
    +    3.9047    -->   121.23        (   4.1512   )
    +    3.1824    -->   124.41        (   3.2998   )
    +    2.8723    -->   127.28        (   2.8849   )
    +    2.8849    -->   130.16        (   4.0388   )
    +    4.0388    -->   134.20        (   5.1377   )
```

### 7.5.3 Linker errors

These errors are more likely to occur than compilation errors. Especially old versions of the linked libraries DRIVERS, TTUTIL and WEATHER may lead to a linker error. It is also possible that the FST model calls a Fortran subroutine which is neither present in the FST source, nor mentioned in the linker command. The linker then reports an unresolved external symbol (it cannot find the called subroutine).

Note that the automated execution procedures on PC and Macintosh, automatically link a library USERSUBS.LIB (on PC) and USERSUBS.OLB (on Mac) with the generated Fortran. This "default user library" may contain subroutines called from the FST model. If other libraries should be linked with the model, the linking has to be done "by hand".

### 7.5.4 Runtime errors

A few categories of runtime errors can be distinguished:

- No weather data can be found. Usually the WEATHER variable WTRDIR has been set to a wrong directory. This also leads to a translation warning.
- Datafiles are lacking, they may have been accidentally deleted.
- Variables are lacking in the datafile MODEL.DAT or TIMER.DAT. The datafiles used will probably belong to another model which has been translated but which is not the one being executed.
- Range error of an interpolation function. These are messages from the functions CSPLIN or LINT2. You have to find out why the X variable is out of range.
- Array subscripts out of declared bounds. The array functions ELEMNT, ARMEAN, ARSUMM, etc. verify the existence of the requested array elements and report an error in runtime.
- The Runge-Kutta driver RKDRIV sometimes reports an insignificant time step. The time step has become so small that no progress is made. Increasing the value of TRANSLATION_GENERAL variable EPS may help. The accuracy check of RKDRIV also involves the SCALE array in MODEL.DAT which is set to default values by the translator (see Section 9.2.2.3 on page 152). You may have to change these values in case of integration problems with RKDRIV.
- Illegal arguments of the SQRT or LOG function are usually the result of conceptual errors in the model equations (which cannot be found by the translator). The same applies to "division by zero". The FST translator only verifies the form of the expressions, not their numerical results. Hence, errors resulting from zero or negative values appear in runtime.

### 7.5.5 The use of TRACE in TRANSLATION_GENERAL mode

In case of problems with a TRANSLATION_GENERAL model, the driver can be instructed to produce a detailed integration report on the log file MODEL.LOG. This is achieved by means of the TRANSLATION_GENERAL variable TRACE. The definition of this variable is simply added to the TRANSLATION_GENERAL statement, for instance,

```
TRANSLATION_GENERAL .... ; TRACE=3
```

The different values of TRACE may be regarded as levels of detail. Increasing TRACE gives a more detailed report on log file. Here are the values with their meaning:

| | |
|---|---|
| `TRACE=0` | -> message on Initial, Dynamic, Terminal |
| `TRACE=1` | -> also short initialization report |
| `TRACE=2` | -> also output times and number of steps to screen ('follow simulation') |
| `TRACE=3` | -> also reports integration steps + output times to logfile |
| `TRACE=4` | -> also reports on state event iteration (events are not possible from FST) |

```
    +   5.1377      -->    139.34        (   5.6155     )
    +   5.6155      -->    144.96        (   6.2606     )
    +   6.2606      -->    151.22        (   6.8276     )
    +   6.8276      -->    158.04        (   7.7086     )
    +   7.7086      -->    165.75        (   9.1083     )
    +   9.1083      -->    174.86        (   10.893     )
    +   10.893      -->    185.75        (   13.215     )
    +   13.215      -->    198.97        (   16.290     )
    +   1.0310      -->    200.00        (   16.290     )      <-at 2 x PRDEL
Output flag set =====    200.00
    +   16.290      -->    216.29        (   20.247     )
    +   20.247      -->    236.54        (   24.429     )
    +   5.4848      -->    242.02        (   21.939     )
    +   2.0901      -->    244.11        (   7.9015     )
    +   2.3710      -->    246.48        (   8.9665     )
    +  0.26769      -->    246.75        (  0.24574     )
Output flag set =====    246.75                                <-last dynamic output
  RKDRIV 2.0: Terminate model

Temporary file TIMER.TMP deleted by RDDTMP                     <-temporary files created
Temporary file MODEL.TMP deleted by RDDTMP                     <-by TTUTIL are deleted
```

# Chapter 8

# Reference Manual

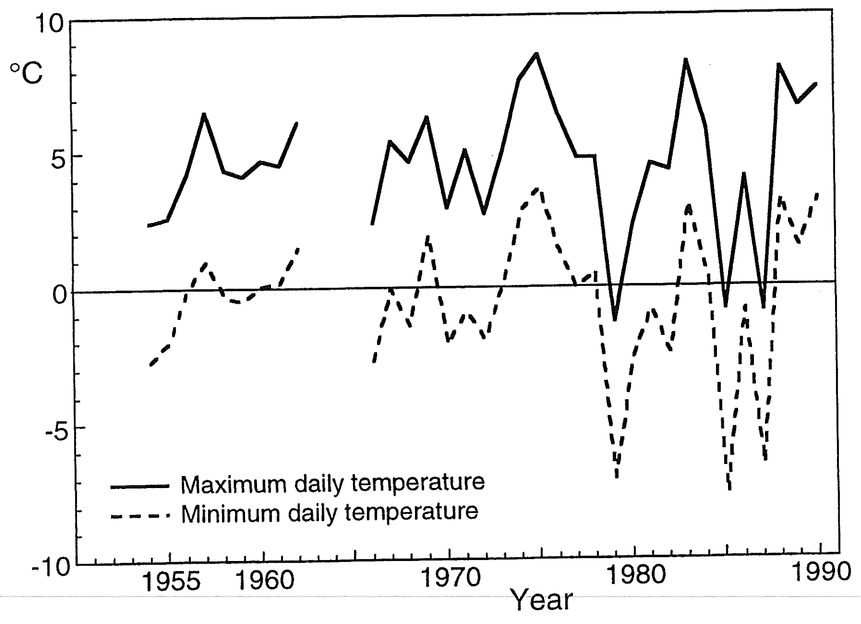


*Figure 8.1 on the title page of this chapter.* The January average of the daily minimum and the daily maximum temperature for Wageningen (The Netherlands) between 1954 and 1990. An FST program calculating monthly means of weather data can be found in Listing 6.4 on page 75.

# 8 Reference manual of FST

This Chapter contains a more systematic treatment of the various aspects of FST. The text provides a full reference in case of doubt.

A consequent distinction has been made between FST variable types and FST statement types although, in practice, there is much overlap between the two. Without this distinction we could not write down the exact rules of FST, however.

## 8.1 FST Programming conventions

### 8.1.1 Statements, program lines and continuation

An FST program consists of statements. Between the statements blank lines and comment lines may occur. A statement may begin in the first column of the file and continue up to column 72. If it ends with 3 dots ". . ." it is continued on the next line of the program.

### 8.1.2 Comment lines

Comment lines begin with an asterisk in the first column. They are usually written in lowercase characters above the statements they describe. In FST listings they appear unchanged without statement number. Comment lines above calculations appear in the generated Fortran program. Comment lines above input and control statements are not copied to any of the output files.

### 8.1.3 Upper case characters

FST statements are written in UPPERCASE characters. The use of lowercase characters leads to a warning from the FST translator with the following exceptions:

- For comment lines there are no rules. It is advised, however, to write them in lowercase characters.
- TITLE statements contain a description of the FST model in words, which may contain lower- and uppercase characters.
- The value of character variables like CNTR (country code in WEATHER statement), or DRIVER (the name of the driver in TRANSLATION_GENERAL) appears unchanged in the generated data files and may contain lowercase characters.

### 8.1.4 File names in lowercase

All filenames are converted to lowercase characters before passing them to the operating system. On DOS systems all filenames are converted to uppercase by the system. On Macintosh, new files get the suggested names, here in lowercase, but existing files in uppercase are recognized. UNIX is truly case sensitive. Hence, on UNIX systems, FST program files should have a name in lowercase characters. The generated Fortran source and data files have lowercase names and the compiled and linked Fortran program will also use lowercase filenames.

## 8.2 FST symbol names

Symbols are names of variables, functions or subroutines. Symbol names in FST follow the conventions of Fortran 77. They have a maximum length of 6 characters, start with a letter and may further contain digits and underscores. Hence A1, B_3, DNA123 and XX_444 are valid variable names. The names 1AB, _BAT, ABCD123 are not valid. The same rules apply to the names of called subroutines. In future versions the maximum length of the variable names in FST will probably be extended to 15 or 31 characters.

### 8.2.1 Forbidden variable names

Although the user is generally free to invent his or her FST variable names, there are three categories of forbidden names:

- The keywords of FST, Fortran and some keywords of old CSMP. It would be confusing to use variable names like TIMER and INCON. The same applies to names like WRITE, COMMON, READ and ENDIF.
- The variable names used in the standard framework of the generated Fortran programs. Examples are OUTPUT, TERMNL, WSTAT, ITASK and IULOG. The full list contains almost 30 names. The use of these names for user-defined variables would lead to errors in the generated Fortran.
- The subroutine names and the names of the common blocks in the libraries DRIVERS, TTUTIL and WEATHER. Strictly speaking it would be possible to use these names as FST variable names and, consequently, as local variable names in the generated Fortran program. To prevent confusion, however, the use of names from the three linked libraries is forbidden.

The precise list of reserved variable names is given in Appendix B. It will be adapted from time to time to reflect changes in the linked libraries, look on the floppy disk for the latest information. The use of a forbidden variable name leads to an error message of the FST translator.

### 8.2.2 Reserved variable names

Except forbidden variable names there are also names reserved for special purposes. STTIME, for instance, is the start time of the simulation and this variable name cannot be defined or used in any other way. Other variables can be used only, for instance weather variables. There are eight groups of reserved variable names:

- The names of the TIMER variables. See Section 8.3.2.1.
- The WEATHER control variables. See Section 8.3.2.2.
- The TRANSLATION_FSE variable IOBSD. See Section 8.3.2.3.
- The TRANSLATION_GENERAL variables. See Section 8.3.2.4.
- The variables supplied by the driver (the integration procedure). The most important of them is TIME. It is impossible to use TIME as the name of a user-defined variable, but the variable TIME can be used in expressions. The same holds for a few other variables. See Section 8.3.2.7.
- A number of weather variables and weather station data. See Section 3.2.4.4.
- The names of the Fortran intrinsic functions like SIN and TAN. The complete list is given in Table 8.1 on page 134.
- The names of the FST intrinsic functions like INTGRL, AFGEN and ARMEAN. The complete list is given in Table 8.2 on page 136.

Improper use of any of these special variable names leads to an FST error message.

## 8.3 FST symbol types

Symbols are names of variables, functions or subroutines. The symbols in an FST program are classified by the translator according to their function in the program. FST distinguishes between 15 different types. Seven symbol types have names supplied by the user and eight symbol types have prescribed names. These prescribed names are reserved for that purpose. They were described in Section 8.2.

### 8.3.1 Symbols with names supplied by the user

The description of each variable type below consists of the following parts: the variable type is defined, an example is given, its place in the Fortran translation is described, the possibility or impossibility of reruns is mentioned and array use is described. Names given to the model variables have to comply with the conventions mentioned in Section 8.2.

---

#### 8.3.1.1 Array Size Variable

An FST array size variable is defined by means of an ARRAY_SIZE statement. The variable may be used as a integer variable in subroutine calls and expressions in the INITIAL, DYNAMIC and TERMINAL section of the FST program, independent of the position of the CONSTANT statement.

Example:

```
ARRAY A(1:N+3), B(1:M)
ARRAY_SIZE N=10 ; M=100
X = REAL(N) / REAL(M**2)     <- Integer variables in expressions have to be converted to real variables
```

An array size variable appears in the generated Fortran program as an integer Fortran parameter which is used in the Fortran declaration of the arrays. It cannot be redefined in a rerun section by means of another ARRAY_SIZE statement. An array size variable cannot be an array.

See also: Chapter 4, Section 8.5.2.3

---

#### 8.3.1.2 Calculated variable

An calculated variable is defined by means of an assignment or a subroutine call in the INITIAL, DYNAMIC or TERMINAL section of the FST program. It can be used in expressions or as input arguments for functions or subroutines in the section in which it is defined or in a later section. Note that also state variables are classified as calculated variables. They are calculated by means of an INTGRL function call.

Example:

```
XX = SIN(A) + COS(B)          <- Definition of XX
P = XX**2 + SQRT(A)           <- Use of XX
```

The statement defining a calculated variable is written in the initial, dynamic or terminal section of the generated Fortran program. Calculated variables cannot be redefined in a rerun section.

The name of a calculated variable can be declared as an array with an ARRAY statement in the DECLARATIONS section. In such a case all array elements have to be defined in a single statement, by means of one or more substatements, or by a subroutine call.

See also: Section 3.2.6, Section 4.3, Chapter 5, Section 8.5.2.1, Section 8.5.2.4.

#### 8.3.1.3 Called SUBROUTINE

The name of a called subroutine is also an FST symbol appearing in the symbol list of an FST program. The I/O structure of a called subroutine has to be declared to the translator by means of a DEFINE_CALL statement in the DECLARATIONS section. Actual subroutine calls then follow the Fortran syntax. It is not necessary to include a called subroutine in the FST source file. It should then be linked with the compiled MODEL.FOR. This implies that a called subroutine may be part of an object library, possibly without source code available.

Example:
```
DEFINE_CALL MYSUB (INPUT, OUTPUT_ARRAY, INTEGER_INPUT)
ARRAY A(1:N), B(1:M)
...
CALL MYSUB (X,A,N)          <- Calculation of output array variable A
CALL MYSUB (X,B,M)          <- Calculation of array variable B
```

A subroutine call is a calculation statement which appears in the initial, dynamic or terminal section of the generated Fortran program together with the other calculations. The information in the DEFINE_CALL statement does not appear in Fortran.

See also: Chapter 5, Section 8.5.2.4, Section 8.5.2.7.

#### 8.3.1.4 Constant

An FST constant is defined by means of a CONSTANT statement. A constant can be used in the INITIAL, DYNAMIC and TERMINAL section of the FST program, independent of the position of the CONSTANT statement.

Example:
```
CONSTANT PI=3.14159265
```

A constant appears in the generated Fortran program as a Fortran parameter. It cannot be redefined in a rerun section. An FST constant cannot be an array.

See also: Section 3.2.3.3, Section 8.5.2.5

#### 8.3.1.5 Initial constant

An initial constant defined by means of an INCON statement somewhere in the MODEL section. An initial constant can be used in the INITIAL, DYNAMIC and TERMINAL section of the FST program, independent of the position of the INCON statement. An initial constant should act at least once as the initial value of a state variable by being used as the first argument of an INTGRL function call.

Example:
```
DYNAMIC
X = INTGRL (IX,RX)          <-  IX is used as initial constant
...
INCON IX = 4.5              <-  IX is defined by means of an INCON statement
```

An initial constant is written to the file MODEL.DAT which is read by the initial part of the generated Fortran program. An initial constant can be redefined in a rerun section by means of another INCON statement.

The name of an initial constant can be declared as an array with an ARRAY statement in the DECLARATIONS section. The subscript range used for an initial constant should be identical to the range of the state array(s) for which it is used. Note that a state array can be combined either with an initial constant array of the same length or with a scalar initial constant. For the definition of an array of initial values by means of an INCON statement the rules for parameter arrays apply (see Section 8.5.2.17)

See also: Section 3.2.3.2, Section 3.3, Section 8.5.2.13

#### 8.3.1.6 Interpolation function or AFGEN function

An interpolation function is defined by means of a FUNCTION statement in the MODEL section of FST. The FUNCTION statements specifies a number of points of the function. Actual function values are calculated by means of linear interpolation (with AFGEN) or cubic spline interpolation (with CSPLINE) between the function points given. AFGEN or CSPLIN function calls may occur in calculation statements or FINISH statements.

Example:

```
RGR1 =  AFGEN(RGRFUN,TEMP)                           <-The function value for TEMP is estimated
DIFF = CSPLIN(RGRFUN,TEMP) - AFGEN(RGRFUN,TEMP)  <-The difference between the 2 methods
FUNCTION RGRFUN =   0.0, 0.1, 10.0, 0.2, ...         <-Four data points specifying the function
                   20.0, 0.2, 30.0, 0.0
TERMINAL
RGRT = AFGEN(RGRFUN,TEMP)                            <-Terminal calculation
```

The data points of an interpolation function appear as array of real values in the file MODEL.DAT. The array is read by means of a call to TTUTIL subroutine RDAREA in the initial part of the generated Fortran program. An interpolation function can be redefined in a rerun section by means of another FUNCTION statement. The numbers of function points in the various reruns do not need to be the same. An interpolation function is not an FST array.

See also: Section 3.2.3.4, Section 3.4, Section 6.1.1, Section 8.5.2.12, Section 8.7.2

#### 8.3.1.7 Parameter

An FST parameter is defined by means of a PARAMETER (or PARAM) statement. A parameter can be used in calculation statements in the INITIAL, DYNAMIC and TERMINAL section, independent of the position of the PARAMETER statement.

Examples:

```
PARAMETER A = 3.4 ; BOB=2.3
```

A parameter is written to the file MODEL.DAT which is read by the initial part of the generated Fortran program. A parameter can be redefined in a rerun section by means of another PARAMETER statement.

The name of a model parameter can be declared as an array with an ARRAY statement in the DECLARATIONS section. The array of values should be fully specified in a single PARAMETER statement by means of one or more consecutive substatements each covering part of the declared subscript range.

See also: Section 3.2.3.1, Section 8.5.2.17

### 8.3.2 Symbols with names prescribed by FST

The symbol types below have prescribed names, which means that only specific names are accepted by the translator as a symbol of that type. For each of the symbol types, the list of prescribed names is given with a brief description of the function of each variable.

#### 8.3.2.1 TIMER variable
A timer variable can only be defined by means of a TIMER statement. The timer variables STTIME, FINTIM and DELT have to be defined in each FST program. They can be used in INITIAL, DYNAMIC or TERMINAL, independent of the position of the TIMER statement(s). The other timer variables can only be defined and cannot be used. The following timer variables exist:

`STTIME` The start time of the simulation, given as a real number. STTIME is often defined as 0.0 unless some special TIME dependent functions are used. If a WEATHER statement is used, the unit of time is day and STTIME is the start time as a day number between 1.0 and 366.999.... The values between 366.0 and 367.0 are valid only in a leap year. The first day of the year corresponds to TIME values between 1.0 and 2.0STTIME has always to be specified.

`FINTIM` The finish time of the simulation, given as a real number. Note that the duration of the simulation is FINTIM-STTIME. In case a WEATHER statement is used, a FINTIM value of 366.0 means the beginning of the next year (except after a leap year), etc. FINTIM has always to be specified.

`DELT` Time step used for the integration of the rates of state. In case of Runge-Kutta integration DELT is just the first guess of the time step used. DELT has always to be specified.

`COPINF` A flag given as a character variable 'Y' or 'N'. When 'Y' is given the generated input files CONTROL.DAT (FSE mode only) TIMER.DAT, MODEL.DAT and RERUNS.DAT are copied to the output file after completing all reruns. The default value of COPINF is 'N'.

`IPFORM` A code for the form of the output tables, given as an integer number in the range [4,6].
4 = spaces between columns
5 = TAB's between columns (spreadsheet output)
6 = two column output
The default value for IPFORM is 4.

`PRDEL` The period between output times, given as a real number. The default value of PRDEL is the simulated period FINTIM-STTIME.

`RGINIT` A flag given as a character variable 'Y' or 'N'. When 'Y' is given the random generators are reset at the start of each rerun. The default value of RGINIT is 'Y'. See also Section 3.7.

`RGSEED` The integer seed used to reset the random generators (if a reset is done). A zero value leads to a generated seed value derived from the system clock at runtime. The default value of RGSEED is 1. See also Section 3.7.

All timer variables are written by FST to the generated data file TIMER.DAT.

All timer variables can be redefined in a rerun section by means of another TIMER statement. Timer variables cannot be declared as an FST array.

See also: Section 3.2.4.3, Section 8.5.2.21

#### 8.3.2.2 WEATHER control variable
A WEATHER control variable can only be defined by means of a WEATHER statement. A weather control variable cannot be used in expressions or printed. The following WEATHER control variables exist:

`CNTR` The country code as a string variable, e.g. 'NLD'

`ISTN` The station number as an integer number

`IYEAR` The year as an integer number

`WTRDIR` The directory with weather data as a string variable. The form of this directory string depends on the machine on which FST is used. On a DOS machine it will be something like WTRDIR='C:\SYS\WEATHER\' and on a Macintosh WTRDIR='Macintosh HD:weather data:'. When WTRDIR is not specified or a space is given, the default directory is used.

The FST translator makes use of the weather data files and the WEATHER subroutine library described by van Kraalingen *et al.* (1991). All WEATHER control variables are written by FST to the generated data file TIMER.DAT. In FSE mode the WEATHER control variables are read by the FSE driver. In TRANSLATION_GENERAL mode they are read by the weather interface routine WTRINT, called from the generated model subroutine.

The country code, station number, year and weather directory provide all information required for access to the weather data files. These weather data files are formatted files which can be installed on all computer types and which are read by the subroutines of the Fortran weather system which has to be linked with the generated Fortran.

The WEATHER control variables CNTR, ISTN and IYEAR can be redefined in a rerun section by means of WEATHER statement(s). WEATHER control variables cannot be declared as an FST array.

See also: Section 3.2.4.4, Section 8.5.2.25

---

#### 8.3.2.3 TRANSLATION_FSE variable

A TRANSLATION_FSE variable can only be defined by means of a TRANSLATION_FSE statement. There is only one TRANSLATION_FSE variable:

`IOBSD` One or more combinations of year and day given as integer numbers. At the specified days the FSE driver produces (additional) output. The variable IOBSD is meant to get output at days on which observations were made. The variable IOBSD cannot be used in expressions or in PRINT or OUTPUT statements. See also van Kraalingen (1995, Section 7.3.4). Example:

```
IOBSD = 1984, 11, 1985, 117       <-(Additional) output days
```

The TRANSLATION_FSE variable can be redefined in a rerun section by means of another TRANSLATION_FSE statement. A TRANSLATION_FSE variable cannot be an FST array.

See also: Section 3.2.4.2, Section 3.3, Section 8.5.2.23

---

#### 8.3.2.4 TRANSLATION_GENERAL variable

A TRANSLATION_GENERAL variable can only be defined by means of a TRANSLATION_GENERAL statement.

`DRIVER` The name of the subroutine which organizes the simulation, given as a string variable. Subroutine RKDRIV uses Runge-Kutta integration with adaptive time step according to an error criterion (EPS). Subroutine EUDRIV uses Euler integration. The default value is 'RKDRIV'. Example:

```
TRANSLATION_GENERAL DRIVER='RKDRIV'
```

`DELMAX` The maximum time step used by the Runge-Kutta integrator. A limitation of the time step is useful when phenomena can be completely missed by large steps. The default value of DELMAX is the simulated period FINTIM-STTIME which implies "no limitation".

`EPS` The accuracy variable of the Runge-Kutta integrator with a default value of 1.0E-4. See for the meaning of this variable Section 9.2.2.

`TRACE` An integer value in the range [0,4] specifying the level of detail in the logfile output of the drivers EUDRIV and RKDRIV. See Section 7.5.5 on page 99 for the meaning of the TRACE values. The default value is 0.

The integer and real TRANSLATION_GENERAL variables DELMAX, EPS and TRACE can be used in calculations and can be printed. All TRANSLATION_GENERAL variables can be redefined in a rerun section by means of another TRANSLATION_GENERAL statement. TRANSLATION_GENERAL variables cannot be declared as an FST array.

See also: Section 3.2.4.1, Section 3.3, Section 7.5.5, Section 8.5.2.24

---

**8.3.2.5 Fortran Intrinsic function**

The intrinsic functions of Fortran 77 are available in FST. Number and type of arguments is verified by the translator. The list of implemented functions is given in Table 8.1 on page 134.

There is a minor limitation: Some intrinsic functions in Fortran allow different argument types, for instance ABS, which may have a real or integer argument. In such cases FST accepts only one of the two types.

See also: Section 8.6

---

**8.3.2.6 FST Intrinsic function**

Examples of FST intrinsic functions are the important INTGRL function, the functions AFGEN and CSPLIN for interpolation. In Table 8.2 on page 136 all FST intrinsic functions are described.

The FST intrinsic functions have been implemented via the libraries DRIVERS and TTUTIL which are linked with the generated Fortran program. Note that the arguments of the Fortran subprograms may differ from the arguments used in FST, since the translator adds additional arguments to various function calls. The easiest way of learning the Fortran function calls is to inspect a Fortran module generated by FST.

See also: Section 8.7

---

**8.3.2.7 Variable supplied by the driver**

These variables are build into the Fortran program which is generated by the FST translator. They can be used but not defined.

`I` Integer variable which cannot be defined. It may only be used in expanded assignments and subroutine calls. The variable I is actually the counter of the DO-loop in the generated Fortran program. The use of I is a bit tricky and the following warning is given by the translator: Warning: The loop counter I is used in an expression or subroutine argument. The range of values assigned to I is the range of the first expanded subscript range in the substatement, e.g. in "A(2:5)=0.3*REAL(I)*B(4:7)" the counter I takes the values 2,3,4,5 and NOT 4,5,6 and 7. The counter I cannot be printed. See also Section 4.3.2, Section 6.1.8.

`TIME` Real variable which cannot be defined by the user. It is the system time running from the start time STTIME to the finish time FINTIM. It can be used in expressions in the INITIAL, DYNAMIC or TERMINAL section of the FST program. TIME is printed always.

`DELDID` In TRANSLATION_GENERAL mode the variable DELDID is the size of the last completed time step. It can be used in the DYNAMIC and TERMINAL section of the program. In FSE mode this variable is not available. DELDID can be printed.

`RGACTS` The last seed used to initialize the random number generators RGUNIF and RGNORM. RGACTS is an integer variable. Usually RGACTS is equal to the seed supplied as the TIMER variable RGSEED. But if RGSEED is defined as 0.0, a seed is generated from system clock. The generated seed is available then as RGACTS (See also Section 3.7).

---

**8.3.2.8 Weather and calendar data**

By specifying the weather control statements, set of weather data become available for use in expressions. The list of weather, station and calendar data is given in 3.2.4.4.

The value of these variables corresponds always with the current value of TIME, measured in days. At the start of a year , at 00:00:00, TIME is 1.000 and a normal year ends after 365 days at the value 365.999... (366.999... for a leap year). Larger values of time then correspond to the next year. The value of DOY is 1.0 throughout the first day of the year and 365.0 throughout the last day of a normal year. In the next year it then starts with 1.0 again. Hence, the real variable DOY can be used, for instance, to interpolate an interpolation FUNCTION with measurements as function of day number.

See also: Section 2.2, Section 3.2.4.4, Section 8.5.2.25

## 8.4 FST program sections

An FST program may consist of several sections. The various sections may be separated from each other by section statements. A section statement, however, is needed only if the program becomes ambiguous without it. In Listing 8.1 the rules for FST sections are summarized. Then the function of the various sections is briefly explained.

Listing 8.1 The structure of an FST program. Many section statements can be omitted. See the text for further details.

```
DECLARATIONS                  <-  On top of the declaration statements ; optional section statement
...                           <-  ARRAY and DEFINE_CALL statements
MODEL                         <-  Begin of model description ; optional section statement
...
INITIAL                       <-  Begin of initial calculations, needed if there are initial calculations
...                           <-  Initial calculations
DYNAMIC                       <-  Begin of dynamic calculations ; needed with INITIAL
...                           <-  Without INITIAL the first calculation is assumed to be DYNAMIC
TERMINAL                      <-  A terminal section needed if there are terminal calculations
...                           <-  Terminal calculations
END                           <-  End of MODEL section ; end of run 0, needed if there is more
...                           <-  Changes for rerun 1 are specified here
END                           <-  End of first rerun, needed if there is more
...                           <-  Changes for rerun 2 are specified here
END                           .......
...                           <-  Additional changes for last rerun
END                           <-  End of last rerun, needed if there is more
STOP                          <-  End of model calculations, needed if there is more
       SUBROUTINE ...         <-  Linked Fortran subroutines
       ........
       END                    <-  End of the last Fortran subroutine: a Fortran END statement
ENDJOB                        <-  To the memory of punch cards ; optional
```

### 8.4.1 DECLARATIONS section

The DECLARATIONS section consists of just the ARRAY and the DEFINE_CALL statements. The DECLARATIONS statement itself is optional. See also Section 3.1 and Section 3.2.2.

### 8.4.2 MODEL section

An FST program always contains a MODEL section. The MODEL section starts with the MODEL statement or with any other statement which is not allowed in the DECLARATIONS section. The MODEL section contains the actual FST model: the calculations, the output variables, the translation mode, etc. The MODEL section ends with and END statement or at the end of the FST file.

The calculations in the MODEL section are organized in three groups: initial, dynamic and terminal calculations. These groups are separated from each other by the section statements INITIAL, DYNAMIC and TERMINAL. Non-calculation statements as TIMER or PARAMETER are not affected by these three sections. They can be written anywhere in the MODEL section. Timer variables and model parameters are in fact always initial variables and their section of definition in the symbol listing is the initial section, although their definitions may be among dynamic or terminal calculations.

#### 8.4.2.1 INITIAL

Initial calculations have to be preceded by the section statement INITIAL. The initial assignments and subroutine calls are brought in a computational order by the translator and are written to the initial section of the generated Fortran program.

The initial calculations are carried out before the actual simulation starts. Initially calculated variables may be used in the DYNAMIC and TERMINAL sections of the model.

#### 8.4.2.2 DYNAMIC

The dynamic section of the calculations starts after a DYNAMIC statement. It also starts at the first calculation when no INITIAL section is present. This implies that (short) FST programs without initial calculations do not need the section statements INITIAL and DYNAMIC. The dynamic assignments and subroutine calls are brought in a computational order by the translator and are written to the dynamic section of the generated Fortran program.

In the dynamic section the state variables are defined with INTGRL function calls and the rates of change are calculated from the state variables. The dynamic section is the core of the model. During the simulation, the calculations in the dynamic section are carried out many times. After completion of the simulation, usually at FINTIM, the dynamic calculations are carried out a last time, however, to calculate the rates of change and all other dynamic variables for the final state values. These final values are available in the terminal section.

#### 8.4.2.3 TERMINAL

Terminal calculations have to be preceded by the section statement TERMINAL. The terminal assignments and subroutine calls are brought in a computational order by the translator and are written to the terminal section of the generated Fortran program.

Terminal calculations are useful for the evaluation of the simulation results. Both initial variables and the final values of the dynamic variables are available in the terminal section.

## 8.4.3 Reruns section

One or more rerun sections may follow the MODEL section. Each rerun section is terminated by an END statement. In a rerun section the following statements may occur: INCON, TIMER, PARAMETER, TRANSLATION_FSE, TRANSLATION_GENERAL, FUNCTION and WEATHER. Variables defined in the MODEL section with these statements can be (repeatedly) redefined in the rerun sections. For each of the rerun sections a new model run is defined in the generated rerun file RERUNS.DAT which is read by the simulation driver. See also Section 3.6.

## 8.4.4 Fortran subroutines

After a STOP statement one or more Fortran subroutines may be added. The Fortran subroutines should comply with the Fortran conventions for statement begin, statement continuation and statement labels. The Fortran code is not analyzed in any way by the FST translator, except for the number of arguments of subroutines which are directly called from the model section of FST. The Fortran section optionally ends with an ENDJOB statement.

The Fortran section is especially useful for subroutines written specifically for the FST program. If the used subroutines have a more general character, we strongly advise to link them with the compiled Fortran and NOT to include them in the FST file.

## 8.5 The FST statements

### 8.5.1 Classification of FST statements

The following types of statements can be distinguished:

- Declaration statements
  ARRAY, DEFINE_CALL
- Section statements
  DECLARATIONS, MODEL, INITIAL, DYNAMIC, TERMINAL,END, STOP, ENDJOB
- Calculation statements
  Assignment, CALL to a subroutine
- Input statements
  PARAMETER, INCON, CONSTANT, FUNCTION, ARRAY_SIZE
- Simulation control statements
  TRANSLATION_GENERAL, TRANSLATION_FSE,TIMER, WEATHER, FINISH
- Output statements
  PRINT, OUTPUT, TITLE

### 8.5.2 The FST statements in alphabetical order

---

#### 8.5.2.1 Assignment

*Syntax*

defined_variable = {real_constant | real_variable | real_expression}

*Example:*

```
A = 5.0
B = C
C = 1.0 + SIN(AFGEN(RTB,X-1.0)) + ARSUMM(X,1,N-1)
D = INTGRL (ID,RD)
```

*Type of statement and position in program*

An assignment is a calculation statement. All calculation statements should be contained in the MODEL section. They may be organized in three groups: INITIAL calculations, DYNAMIC calculations and TERMINAL calculations. Without INITIAL statement the first assignment in the program is considered to be a dynamic one. Calculations cannot occur in a rerun section.

*Description*

Each assignment in FST uniquely defines an initial, a dynamic or a terminal variable. Together with the subroutine calls, the three groups of assignments are separately sorted in order to find a computational order for the calculations.

Assignments with an INTGRL function form an exception. The INTGRL function call cannot be part of a larger expression and "INTGRL statements" are not sorted together with the other dynamic calculations. In fact, INTGRL statements are somewhat related to declarations since integration of rates of change is the function of the entire FST program.

*Use in combination with array variables*

Also calculated array variables have to be defined in a single statement. The statement may then be subdivided into substatements. Each substatement consists of the assignment for a part of the declared range. The rules for array assignments are:

- Only a single array variable can be calculated in the statement.
- The calculation of the array variable must be complete, i.e. all declared elements must be calculated.
- The substatements are separated from each other by a semi-colon ";".
- Each substatement defines one or more array elements.
- There should be no gaps or overlaps in the subscripts and subscript ranges covered by consecutive substatements. For types of subscripts and subscript ranges, see Section 4.3.1.3.
- There is only a single substatement defining an absolute-to-relative range.
- The right hand part of each substatement consists of a real constant, a real (array) variable or a real expression.
- In a right-hand expression, single elements of other arrays may be used as scalar variables.
- In a right-hand expression, also subscript ranges of other arrays can be referenced.
- Subscript ranges in the left-hand side and in the right-hand side of a substatement must have the same length.
- Subscript ranges in the left-hand side and in the right-hand side of a substatement must be of the same type. If case of absolute-to-relative or relative-to-relative ranges, the various arrays must belong to the same family.

The FST sorting algorithm works on the basis of entire statements. This implies that all variables appearing in the right-hand side of the substatements should be known before the array assignment can be evaluated. As a consequence, array elements may not depend on other elements of the same array.

*Example of use with array variable*

```
ARRAY A(0:N+1), B(1:N)
...
A(0)=0.0 ; A(1:N)=2.0+SQRT(B) ; A(N+1)=1.0
```

More examples are given in Section 4.3.

*See also*

Section 3.2.6, Section 4.3.

#### 8.5.2.2 ARRAY

*Syntax*

ARRAY array_declaration [, array_declaration] ...

*array_declaration* = array_name (*lowerbound* : *upperbound*)
*lowerbound* = integer_constant [+ integer_constant]
*upperbound* = array_size_variable [+ integer_constant]

*Example:*

```
ARRAY AA(3:N-4), SPEED(-5:NUMBER+20), BB(1:M), CC(1+2:M)
```

*Type of statement and position in program*

The ARRAY statement is a declaration statement and may occur only in the declaration section on top of the program. It cannot occur in a rerun section.

*Description*

The array statement declares one or more array names to the FST translator. Each array also becomes associated with an array_size_variable. In the example the array_size_variables used are N, NUMBER and M. These variables automatically become integer variables in the program. Array_size_variables get a numerical value by means of an ARRAY_SIZE statement. Different arrays may make use of the same array_size_variable (e.g. the arrays BB and CC in the example). Such arrays are said to belong to the same array family.

The ARRAY statement declares only the names of the arrays. Later in the program, the array variables can be used as state variables, rate variables, initial constants, parameters or auxiliary variables.

*Use in combination with array variables*

In the description of all statements, special attention is given to the use of array variables in combination with each statement. Clearly, the array statement itself has no function when no array variables are used.

*See also*

Section 3.2.2.1, Chapter 4, Section 8.5.2.3

### 8.5.2.3 ARRAY_SIZE

*Syntax*

ARRAY_SIZE array_size_variable = integer_constant [; ...]

*Example*

```
ARRAY_SIZE  N=4 ; N2=16
```

*Type of statement and position in program*

The ARRAY_SIZE statement is an input statement and it may occur in the MODEL section of FST. It cannot occur in a rerun section.

*Description*

Defines the value of one or more array size variables. Each array size defines the size of one or more arrays forming together an array family. An array size variable can be used as an integer variable in expressions.

*Use in combination with array variables*

Array size variables belong to arrays. Without an ARRAY declaration in which it is used, an array size variable is not accepted by the translator.

*See also*

Section 3.2.3.5, Chapter 4, Section 8.3.1.1, Section 8.5.2.2

### 8.5.2.4 CALL subroutine

*Syntax*

CALL *subroutine_name* ({*input_arg* |*output_arg*} [,{*input_arg* |*output_arg*} ]...)

*subroutine_name* = name of a declared subroutine
*input_arg* = constant, variable or expression
*output_arg* = variable

*Examples:*

```
CALL MYSUB1 (X, Y)
* here, only the last argument is output, the others input
CALL MYSUB2 (2.34, REAL(N), 1.0+X-ARMEAN(A,1,N), SIN(B), RESULT)
```

*Type of statement and position in program*

A subroutine CALL is a calculation statement. It needs to be in the MODEL section in one of the subsections INITIAL, DYNAMIC or TERMINAL. Without an INITIAL statement, a subroutine call causes a jump to the dynamic section. A subroutine call cannot occur in a rerun section.

*Description*

A subroutine CALL makes use of a Fortran subroutine included in the FST file or linked with the generated Fortran model. Subroutine CALL's are used exactly in the same way as in Fortran, with the exception that a subroutine argument must be either input or output. In order to sort a subroutine CALL together with the other calculations, the FST translator has to know the I/O structure of the CALL. Therefore, each subroutine which is called directly from the FST program has to be declared by means of a DEFINE_CALL statement in the DECLARATIONS section. Subroutines which are called indirectly, via other Fortran subprograms, need not to be declared.

*Use in combination with array variables*

Also array variables may serve as arguments of a subroutine. Such arguments need to be declared then as INPUT_ARRAY or OUTPUT_ARRAY (see the DEFINE_CALL statement) INPUT_ARRAY arguments, however, cannot be constants or expressions. They must be array variables.

If an array variable is used as an argument known to FST as a simple, scalar INPUT or OUTPUT argument, the subroutine call is expanded to a Fortran DO-loop and the array elements are used or calculated in a series of calls to the subroutine. This also happens in case of array subscript ranges used in INPUT expressions. However, all expanded arrays in the subroutine CALL need to be in the same array family and all subscript ranges need to be of the same type and of equal length.

Expanded calls cannot have OUTPUT_ARRAY arguments or non-expanded OUTPUT arguments, since these arguments would be calculated many times in the generated DO-loop.

*Example of use with array variable*

```
DEFINE_CALL MYSUB1 (INPUT,OUTPUT)
DEFINE_CALL MYSUB2 (INPUT_ARRAY, OUTPUT_ARRAY, INTEGER_INPUT)
DEFINE_CALL MYSUB3 (INPUT_ARRAY, INTEGER_INPUT, INPUT, OUTPUT)
ARRAY XX(1:N), YY(1:N), ZZ(1:N)
ARRAY KK(1:P)
....
CALL MYSUB1 (XX, ZZ)                <- Expanded: ZZ(I) calculated from XX(I),
                                       I=1...N
CALL MYSUB2 (XX, YY, N)             <- Array YY calculated from array XX,
CALL MYSUB3 (XX, N, REAL(I), KK)    <- Expanded: KK(I) calculated from array
                                       XX, integer N  and REAL(I) with I=1...P
```

*See also*

Section 3.2.2.2, Section 8.5.2.7, Chapter 5.

### 8.5.2.5 CONSTANT

*Syntax*

CONSTANT    FST_constant = real_constant  [; ...]

*Example*

CONSTANT PI = 3.14159265

*Type of statement and position in program*

The CONSTANT statement is an input value statement and it may occur in the MODEL section of FST. It cannot occur in a rerun section.

*Description*

The CONSTANT statement can be used for the definition of mathematical or natural constants, which are used in the program. A constant is not written to a Fortran data file, but it appears in the generated Fortran program. This is different from an FST parameter which is written by the FST translator to the Fortran data file MODEL.DAT.

*Use in combination with array variables*

A CONSTANT cannot be an array variable.

*See also*

Section 3.2.3.3, Section 8.3.1.4

### 8.5.2.6 DECLARATIONS

*Syntax*

DECLARATIONS

*Example*

```
TITLE My FST program          <- may be on top of the program
DECLARATIONS
ARRAY .....                   <- declaration statement
```

*Type of statement and position in program*

The DECLARATIONS statement is a section statement, which may occur only once.

*Description*

A DECLARATIONS statement is optional on top of the declarations section of an FST program. The declaration section ends with the MODEL statement or with any non-declaration statement, except TITLE which may occur in any section.

*Use in combination with array variables*

A section statement has no relation to arrays.

*See also*

An overview of the FST program structure in Section 8.4.

#### 8.5.2.7 DEFINE_CALL

*Syntax*

DEFINE_CALL *subroutine_name* (*argument_type* [,*argument_type*] ...)

```
argument_type = { INPUT | REAL_INPUT|
                  INPUT_ARRAY | REAL_INPUT_ARRAY}
                  INTEGER_INPUT |
                  OUTPUT | REAL_OUTPUT |
                  OUTPUT_ARRAY | REAL_OUTPUT_ARRAY }
```

*Examples*

```
DEFINE_CALL MYSUB1 (INPUT,OUTPUT)
DEFINE_CALL MYSUB2 (INPUT_ARRAY, OUTPUT_ARRAY, INTEGER_INPUT)
DEFINE_CALL MYSUB3 (INPUT_ARRAY, INTEGER_INPUT, INPUT, OUTPUT)
```

*Type of statement and position in program*

The DEFINE_CALL statement is a declaration statement, which may occur in the DECLARATIONS section only.

*Description*

The DEFINE_CALL statement declares the input/output structure of a Fortran subroutine which is called from the FST program. The main function of the declaration is to enable the FST translator to find a computational order for subroutine calls in combination with the other calculations. Subroutine arguments are either input or output arguments. The prefix "REAL_" in the argument type is an FST default and may be left out, e.g. the argument types INPUT and REAL_INPUT are exactly the same.

There are three types of input arguments:

- INPUT or REAL_INPUT. An ordinary real input variable. In the actual subroutine call(s), this argument may also be a real constant, like 2.3456, or a real expression, like SIN(A)+1.0. If the expression contains an array variable, possibly with subscript range, the translator will expand the subroutine call (see below).
- INPUT_ARRAY or REAL_INPUT_ARRAY. An entire array is sent to the subroutine. Clearly, the subroutine argument has to be declared as an array in the subroutine. In the actual subroutine call(s), this argument should be an array variable.
- INTEGER_INPUT. This argument type is primarily meant to communicate subscripts or subscript bounds to Fortran subroutines, but it may be used for other purposes as well. Except variables, also integer constants, like 34, or integer expressions, like NINT(A+B), may be used as actual input arguments in a subroutine call.

There are two types of output arguments:

- OUTPUT or REAL_OUTPUT. An ordinary real output variable. Also the actual argument in a subroutine call has to be a variable. Constants or expressions are illegal. If an array variable is used, the translator will expand the subroutine call (see below).
- OUTPUT_ARRAY or REAL_OUTPUT_ARRAY. An entire array is calculated by the subroutine.

If an input argument of the subroutine is overwritten by the subroutine (i.e. becomes an output argument as well), the subroutine cannot be called directly from FST. Instead, one has to write an interface routine between FST and the desired subroutine. An example is given in Section 5.4.

*Use in combination with array variables*

INPUT_ARRAY's and OUTPUT_ARRAY's are entire arrays, used or calculated by means of a single call to the subroutine. The subroutine should be prepared to receive an entire array which usually means that the array size should also be passed to the subroutine as one or two INTEGER_INPUT arguments.

Array variables, however, can also be used as ordinary, scalar argument types INPUT and OUTPUT. In case of INPUT, the array variable may even be contained in an input expression. In that case, the call statement is expanded in a Fortran DO-loop and the array elements are used or calculated elementwise. The range of the DO-loop counter I is the range of the first expanded array in the subroutine call. Clearly, if more than a single array is used in such a way, all subscript ranges should have the same size.

Note that the use of an INPUT_ARRAY does not interfere with statement expansion. In such a case the array is passed to the subroutine as many times as the routine is called. Both non-expanded OUTPUT and OUTPUT_ARRAY arguments are incompatible with an expanded call, however.

*Example of use with array variable*

```
DEFINE_CALL MYSUB1 (INPUT,OUTPUT)
DEFINE_CALL MYSUB2 (INPUT_ARRAY, OUTPUT_ARRAY, INTEGER_INPUT)
DEFINE_CALL MYSUB3 (INPUT_ARRAY, INTEGER_INPUT, INPUT, OUTPUT)
ARRAY XX(1:N), YY(1:N), ZZ(1:N)
ARRAY KK(1:P)
....
CALL MYSUB1 (XX, ZZ)               <- Expanded: ZZ(I) calculated from XX(I),
                                      I=1...N
CALL MYSUB2 (XX, YY, N)            <- Array YY calculated from array XX,
CALL MYSUB3 (XX, N, REAL(I), KK)   <- Expanded: KK(I) calculated from array
                                      XX, integer N  and REAL(I) with I=1...P
```

*See also*

Section 3.2.2.2, Chapter 5.

---

### 8.5.2.8 DYNAMIC

*Syntax*

DYNAMIC

*Example*

```
IX = SQRT(A**2+B**2)           <- initial calculation
DYNAMIC
```

```
        CALL MYSUB (IX,RX)              <- dynamic calculations
        X = INTGRL(IX,RX)
```

*Type of statement and position in program*

The DYNAMIC statement is a section statement, which may occur only once.

*Description*

The DYNAMIC statement defines the beginning of the dynamic part of the calculations. If there are no initial calculations, the DYNAMIC statement is optional. Dynamic calculations after initial ones, however, have to be preceded by the section statement DYNAMIC. The dynamic section ends with a TERMINAL statement, with an END statement or at the end of the FST file.

Note that input statements, simulation control statements and output statements are not affected by the distinction between initial, dynamic and terminal calculations.

*Use in combination with array variables*

A section statement has no relation to arrays.

*See also*

An overview of the FST program structure in Section 8.4.

---

**8.5.2.9 END**

*Syntax*

END

*Example*

```
        ....
        END                        <- end of MODEL section
        * rerun 1
        PARAMETER A=2.0
        END                        <- end of rerun 1
        * rerun 2
        PARAMETER A=3.0
        END                        <- end of rerun 2
        ....
```

*Type of statement and position in program*

The END statement is a section statement at the end of the MODEL section and at the end of each rerun section.

*Description*

The MODEL section contains the initial, dynamic and terminal calculations and all input, output and simulation control statements of the FST program. It is terminated with an END statement. An END statement is also at the end of each rerun section. The END statement may be the last statement in the FST file. In that case it may also be omitted.

*Use in combination with array variables*

A section statement has no relation to arrays.

*See also*

An overview of the FST program structure in Section 8.4.

### 8.5.2.10 ENDJOB

*Syntax*

ENDJOB

*Example*

```
..........
*     last lines of Fortran section
100   CONTINUE
      RETURN
      END           <- No FST, but Fortran END Statement beginning in column 7.
ENDJOB              <- FST ENDJOB statement
```

*Type of statement and position in program*

The ENDJOB statement is an optional section statement, at the end of the FST file.

*Description*

The ENDJOB statement may be used to terminate a Fortran section included in the FST file between the STOP and the optional ENDJOB statement. Without any Fortran, the ENDJOB statement may directly follow the STOP statement at the end of all reruns.

*Use in combination with array variables*

A section statement has no relation to arrays.

*See also*

An overview of the FST program structure in Section 8.4.

### 8.5.2.11 FINISH

*Syntax*

FINISH {constant | variable | expression} { < | > } {constant | variable | expression}

*Example*

```
FINISH B > 2.0
FINISH YEAR > 1887.
FINISH A+B < 1.0+SIN(X)
FINISH RAIN > 10.0
```

*Type of statement and position in program*

The FINISH statement is a simulation control statement. One or more FINISH statements may occur in the MODEL section of FST. A FINISH statement cannot occur in a rerun section.

*Description*

The FINISH statement describes a finish condition of the simulation. When the condition becomes true, the simulation is halted. Almost all variables and functions can be used in the expressions of a finish statement, except character variables like WTRDIR or DRIVER and variables defined in the TERMINAL section of the program. From the functions the INTGRL function cannot be used.

During the simulation, the finish conditions are evaluated after the calculation of all rates of change. Then, the value of TIME, all states, the dynamic variables and all rates of change constitute a synchronous set of values. Since the FINISH conditions are evaluated after the rate calculations, a FINISH condition cannot be used as a safeguard against runtime errors as "divide by zero".

*Use in combination with array variables*

The use of one or more array variables in a FINISH statement leads to elementwise expansion of the condition, i.e. the condition is separately evaluated for all array elements. The array ranges used have to be of the same type and of equal length. If relative subscripts are used, all arrays in the expression have to be part of the same family.

*Example of use with array variable*

```
ARRAY A(1:N), B(1:N), C(1:P)
...
FINISH A+1.0 < LOG(B)            <- N conditions A(I)+1.0<LOG(B(I))
FINISH A(1:3) > C(1:3)**2        <- 3 conditions A(I) > C(I)**2
```

*See also*

Section 3.2.4.5, Section 6.2

### 8.5.2.12 FUNCTION

*Syntax*

FUNCTION function_name = Xvalue, Yvalue, Xvalue, Yvalue [, Xvalue, Yvalue...]

*Example*

```
FUNCTION REDUCT = 0.0, 0.0, 10.0, 0.9, 20.0, 0.9, 30.0, 0.0
FUNCTION EVAPTB = 1.0, 1.0, ...
                  100.0, 2.0, ...     <- Use of statement continuation to
                  200.0, 2.5             write a neat list of (x,y) pairs
```

*Type of statement and position in program*

The FUNCTION statement is an input statement, which may occur in the MODEL section of the FST program. It defines an interpolation function by means of a number of (X,Y) pairs. The defined function is available for interpolation by means of AFGEN or CSPLIN calls in initial, dynamic or terminal calculations. An interpolation function can be redefined in a rerun section by means of another FUNCTION statement.

*Description*

At least two (X,Y) pairs have to be provided in order to have a valid interpolation function. The list of function points should be ordered according to an increasing X value. By means of statement continuation, one can keep an overview over the function points.

The defined (empirical) functions can be used in AFGEN or CSPLIN calls only. The FST function AFGEN calculates a value by means of linear interpolation. The FST function CSPLIN uses natural cubic splines for interpolation between the provided (X,Y) points.

*Use in combination with array variables*

Functions defined by means of a FUNCTION statement cannot be FST arrays. The result of an AFGEN or CSPLIN interpolation is an ordinary, scalar value.

*See also*

Section 3.2.3.4, Section 3.4, Section 6.1.1, Table 8.2 on page 136

### 8.5.2.13 INCON

*Syntax*

INCON initial_constant_name = real_constant [; ...]

*Example*

```
INCON AI= 3.4 ; SPEED=5.0
```

*Type of statement and position in program*

The INCON statement is an input statement, which may be used in the MODEL section of FST. Initial constants can be redefined in a rerun section by means of another INCON statement.

*Description*

The INCON statement defines the initial value of a state variable. By FST, the initial value is assigned to the state variable during the initial phase of the simulation. An initial constant can also be used in initial, dynamic or terminal calculations.

*Use in combination with array variables*

An INCON statement can be used to define an array of initial constants belonging to a state array. An initial constant array is defined by means of one or more substatements, each covering a part of the declared subscript range. The rules for such substatements are the same as for the PARAMETER statement in Section 8.5.2.17.

*Example of use with array variable*

```
ARRAY AI(0:N+1)
...
INCON AI(0:5)=0.0 ; AI(6:N-1)=1.0 ; AI(N:N+1)=2.0,3.0
```

*See also*

Section 3.2.3.2, Section 3.3, Section 8.3.1.5

### 8.5.2.14 INITIAL

*Syntax*

INITIAL

*Example*

```
INITIAL
IX = SQRT(A**2+B**2)          <- initial assignment
CALL MYSUB (IX,IY)            <- initial subroutine call
DYNAMIC
```

*Type of statement and position in program*

The INITIAL statement is a section statement, which may occur only once.

*Description*

The INITIAL statement marks the beginning of the initial calculations. If there are no initial calculations, the INITIAL statement is optional. The initial section ends with a DYNAMIC statement, with an END statement or at the end of the FST file.

Note that input statements, simulation control statements and output statements are not affected by the distinction between initial, dynamic and terminal calculations.

*Use in combination with array variables*

A section statement has no relation to arrays.

*See also*

An overview of the FST program structure in Section 8.4.

#### 8.5.2.15 MODEL

*Syntax*

MODEL

*Example*

```
DEFINE_CALL MYSUB (INPUT,OUTPUT)      <- declaration statement
MODEL
PARAMETER A=1.0 ; ...                 <- input statement
```

*Type of statement and position in program*

The MODEL statement is a section statement, which may occur only once.

*Description*

The MODEL statement marks the beginning of the actual FST program. The MODEL section contains all calculations and all input, output and simulation control statements, except those which define reruns. The calculations are organized in three groups: initial, dynamic and terminal calculations. The other statements are not affected by that distinction. The MODEL section ends with an END statement or at the end of the FST file.

*Use in combination with array variables*

A section statement has no relation to arrays.

*See also*

An overview of the FST program structure in Section 8.4.

#### 8.5.2.16 OUTPUT

*Syntax*

OUTPUT variable_name [,variable_name ...]

*Example*

```
OUTPUT A, B, PETER
```

*Type of statement and position in program*

The OUTPUT statement is an almost obsolete output statement. There should be at least one PRINT or OUTPUT statement in the MODEL section of an FST file. OUTPUT statements cannot occur in rerun sections.

*Description*

The OUTPUT statements creates a printplot for a line printer. The printplot is written to the output file RES.DAT. Each OUTPUT statement leads to its own printplot. Hence, it may be useful to use different OUTPUT statements for a few groups of variables.

Variables listed in OUPUT statements are also included in the formatted output tables written in RES.DAT by the PRINT statement.

The variable types accepted by OUTPUT are:
- array_size variables
- calculated variables
- constants
- initial constants
- parameters
- the driver supplied variables except I
- Weather and calendar data
- Translation_general variables as far as they are numbers
- the TIMER variables STTIME, FINTIM and DELT

Variable values are always associated with a value of TIME. Therefore, variables calculated in the INITIAL section are plotted at STTIME only. Variables calculated in DYNAMIC are plotted as function of TIME (with a resolution controlled by the timer variable PRDEL). And variables calculated in the TERMINAL section are plotted at the final value of TIME, which is either FINTIM or a smaller value due to a finish condition. All this is independent of the position of the OUTPUT statement in the FST program.

*Use in combination with array variables*

The OUTPUT statement does not accept array variables.

*See also*

Section 3.2.5, Section 8.5.2.18

#### 8.5.2.17 PARAM, PARAMETER

*Syntax*

PARAM[ETER] parameter_name = real_constant [; ...]

*Example*

```
PARAMETER B=3.5 ; C=8.9
```

*Type of statement and position in program*

The PARAMETER statement is an input statement, which may be used in the MODEL section of FST. Model parameters can be redefined in a rerun section by means of another PARAMETER statement.

*Description*

The PARAMETER statement defines one or more model parameters. A model parameter can be used in initial, dynamic or terminal calculations. The position of the PARAMETER statement in the MODEL section of FST does not influence its function.

*Use in combination with array variables (also applies to INCON)*

The rules are
- In a single PARAMETER statement one or more parameter arrays and scalar parameters can be defined.
- The definition of a parameter array consists of a consecutive list of substatements.
- The substatements are separated from each other by a semi-colon ";" in the same way as definitions of different scalar parameters.

- The parameter array must be completely defined by the series of consecutive substatements.
- There should not be gaps or overlaps in the subscript ranges covered by consecutive substatements.
- If the elements of a defined range are the same, only a single value needs to be given.
- If several values are given, they should be separated by commas.
- If several values are given, the number of values should be equal to the number of array elements defined. For instance 'A(2:5)' refers to 4 array elements and 4 values are given. and 'A(N-3:N-1)' refers to 3 elements.
- The array definition should be independent of the actual value of N. Hence, a substatement like 'A(16:N-4)=...' may contain only a single value, since the number of elements in A(16:N-4) depends on the value of N which is assumed to be variable.
- There is only a single absolute-to-relative range 'A(16:N-4)=0.0' in which the left bound is a constant and the right bound refers to the array size variable.

*Example of use with array variable*

```
ARRAY A(-3:N+3)
...
PARAMETER A(-3:0)=0.0,1.0,2.0,3.0 ; A(1:5)=4.0 ; ...
          A(6:N-1)=1.0 ; A(N:N+1)=2.0,3.0
```

*See also*

Section 3.2.3.1, Section 8.3.1.7.

#### 8.5.2.18 PRINT

*Syntax*

PRINT variable_name [,variable_name ...]

*Example*

```
PRINT A, B, PETER
```

*Type of statement and position in program*

The PRINT statement is an output statement. There should be at least one PRINT or OUTPUT statement in the MODEL section of an FST file. PRINT statements cannot occur in rerun sections.

*Description*

The PRINT statements lists variables that should be included in the output tables produced in file RES.DAT. Printed variables are also written to the machine readable file RES.BIN. The variables types accepted by PRINT are:

- array_size variables
- calculated variables, arrays or scalar
- constants
- initial constants, arrays or scalar
- parameters, arrays or scalar
- the driver supplied variables except I
- Weather and calendar data
- Translation_general variables as far as they are numbers
- the TIMER variables STTIME, FINTIM and DELT

Variable values are always associated with a value of TIME. Therefore, variables calculated in the INITIAL section are printed at STTIME only. Variables calculated in

DYNAMIC are printed during the simulation at increasing values of TIME (at a series of output times controlled by the timer variable PRDEL). And variables calculated in the TERMINAL section are printed at the final value of TIME, which is either FINTIM or a smaller value due to a finish condition. This is independent of the position of the PRINT statement in the FST program.

Input and control variables are initially known and they are printed at STTIME only, even if the input and control statements are at the end of the MODEL section. If the value of a model parameter A, for instance, has to be printed at the same time as a terminal variable, one needs to make a copy of A in the TERMINAL section and print that copy.

Integer numbers like array size variables are printed by means of a utility which has been written for the Fortran real data type. Hence, integer numbers are printed with a decimal point behind them.

*Use in combination with array variables*

The PRINT statement accepts array variables with or without a subscript range. In the latter case all elements are printed.

*Example of use with array variables*

```
PRINT X(1:5), Y(1), Y(N), Z(N-4:N)
```

*See also*

Section 3.2.5, Section 8.5.2.16

### 8.5.2.19 STOP

*Syntax*

STOP

*Example*

```
............
* rerun 12                     <- rerun sections
PARAMETER A=8.0
END
* rerun 13
PARAMETER A=9.0
END
STOP
      SUBROUTINE MYSUB (X,Y)   <- Fortran subroutine(s)
      ....
      END                      <- Fortran END statement
ENDJOB                         <- FST ENDJOB statement
```

*Type of statement and position in program*

The STOP statement is a section statement, which may occur only once.

*Description*

The STOP statement marks the end of all rerun sections of an FST program. Between the STOP statement and the (optional) ENDJOB statement, Fortran subroutines may be included in the FST file. The Fortran subroutines should comply with the Fortran conventions for statement begin at column 7, statement continuation and statement labels.

Without Fortran and without the ENDJOB statement, the STOP statement is optional and the FST file ends with the last rerun section.

*Use in combination with array variables*

A section statement has no relation to arrays.

*See also*

An overview of the FST program structure in Section 8.4.

---

#### 8.5.2.20 TERMINAL

*Syntax*

TERMINAL

*Example*

```
TERMINAL
RES1 = SQRT(X**2-IX**2)        <- terminal assignment
CALL MYSUB (RES1,RES2)         <- terminal subroutine call
.....
END
```

*Type of statement and position in program*

The TERMINAL statement is a section statement, which may occur only once.

*Description*

The TERMINAL statement marks the beginning of the terminal calculations. If there are no terminal calculations, the TERMINAL statement is optional. The terminal section ends with an END statement or at the end of the FST file.

Note that input statements, simulation control statements and output statements are not affected by the distinction between initial, dynamic and terminal calculations.

*Use in combination with array variables*

A section statement has no relation to arrays.

*See also*

An overview of the FST program structure in Section 8.4.

---

#### 8.5.2.21 TIMER

*Syntax*

TIMER timer_variable = value [; ...]

*Examples*

```
TIMER STTIME=0.0 ; FINTIM=100.0 ; DELT=0.1  <- these have to be there
TIMER COPINF='Y'                            <- default 'N'
TIMER IPFORM=5                              <- default 4
TIMER PRDEL=1.0                             <- default (FINTIM-STTIME)
```

*Type of statement and position in program*

The TIMER statement is a simulation control statement, which must be present in the MODEL section of FST. Timer variables can be redefined in a rerun section by means of another TIMER statement.

*Description*

A TIMER statement defines one or more timer variables. FST requires values for at least STTIME, FINTIM and DELT (also in TRANSLATION_FSE mode). They have to be defined with one or several TIMER statements. The other timer variables have defaults and their definition is optional.

*Use in combination with array variables*

A timer variable cannot be an array.

*See also*

Section 3.2.4.3, Section 3.7
The list of TIMER variables in Section 8.3.2.1

#### 8.5.2.22 TITLE

*Syntax*

TITLE any text without quotes

*Example*

```
TITLE Crop growth model                <- may occur on top of the FST model
TITLE with prescribed crop development
...
MODEL
TITLE and without crop diseases        <- or in the MODEL section
...
```

*Type of statement and position in program*

The TITLE statement is an output statement. TITLE statements may be present in the DECLARATIONS or in the MODEL section and also on top of the program, before the DECLARATIONS section. TITLE statements cannot occur in a rerun section.

*Description*

The text of all TITLE statements is used as a header in the output files. FST also writes the TITLE texts in the header of the generated Fortran module. Besides comment lines, TITLE statement are also used as a short description of the process described by the FST model. Therefore TITLE statements may appear also in the DECLARATIONS section or even on top of the DECLARATIONS section statement.

*Use in combination with array variables*

The TITLE has no relation to arrays.

*See also*

Section 3.2.5.3

#### 8.5.2.23 TRANSLATION_FSE

*Syntax*

TRANSLATION_FSE [translation_fse_variable = value]

*Examples*

```
TRANSLATION_FSE
TRANSLATION_FSE IOBSD=1988,23, 1989,24
```

*Type of statement and position in program*

The TRANSLATION_FSE statement is a simulation control statement, which may be used in the MODEL section of FST. The TRANSLATION_FSE variable IOBSD can be redefined in a rerun section by means of another TRANSLATION_FSE statement. The two translation modes, TRANSLATION_GENERAL and TRANSLATION_FSE, exclude each other.

*Description*

A TRANSLATION_FSE statement has the following consequences

- The FST translator generates an FSE simulation module in Fortran according to van Kraalingen (1995). Also the data files belonging to an FSE program are generated.
- During execution of the model, the simulation is controlled by the FSE driver from the DRIVERS library.
- The unit of time is a day. Hence, STTIME, FINTIM and DELT have to be specified in days (DELT may be a fraction of a day).
- Weather data have to be specified by means of one or more WEATHER statements.

*Use in combination with array variables*

A TRANSLATION_FSE variable cannot be an array.

*See also*

Section 3.2.4.2, Section 3.3
The list of TRANSLATION_FSE variables in Section 8.3.2.3

---

### 8.5.2.24 TRANSLATION_GENERAL

*Syntax*

TRANSLATION_GENERAL [trans_general_variable = value [; ...]]

*Examples*

```
TRANSLATION_GENERAL                       <- the default driver RKDRIV will be used
TRANSLATION_GENERAL DRIVER='EUDRIV' ; DELMAX=0.01
```

*Type of statement and position in program*

The TRANSLATION_GENERAL statement is a simulation control statement, which may be used in the MODEL section of FST. Translation_general variables can be redefined in a rerun section by means of another TRANSLATION_GENERAL statement. The two translation modes, TRANSLATION_GENERAL and TRANSLATION_FSE, exclude each other.

*Description*

A TRANSLATION_GENERAL statement has the following consequences

- The FST translator generates an "GENERAL" simulation module in Fortran. The structure of these modules is described in Section 9.2. Also the data files belonging to the Fortran program are generated.
- During execution of the model, the simulation is controlled by a "GENERAL driver" from the DRIVERS library. There are two such drivers: EUDRIV and RKDRIV.
- Weather data are optional.

The default general driver is RKDRIV which uses the fourth order Runge Kutta scheme with step size control described in Press *et al.* (1986). The second available driver is

EUDRIV which uses Euler integration. Note that reruns can be made also on the driver choice.

*Use in combination with array variables*

A TRANSLATION_GENERAL variable cannot be an array.

*See also*

Section 3.2.4.1, Section 3.3
The list of TRANSLATION_GENERAL variables in Section 8.3.2.4

#### 8.5.2.25 WEATHER

*Syntax*

WEATHER   weather_control_variable = value  [; ...]

*Example*

```
WEATHER WTRDIR='C:\SYS\WEATHER\'
WEATHER ISTN=2 ; CNTR='NLD' ; IYEAR=1990
```

*Type of statement and position in program*

The WEATHER statement is a simulation control statement, which may be present in the MODEL section of FST. Weather_control_variables can be redefined in a rerun section by means of another WEATHER statement.

*Description*

One or more WEATHER statements define the weather control variables. Together, the weather control variables point to a weather data file. Only WTRDIR, the weather data directory, has a default value (the directory of the executed model). The other three, CNTR, ISTN and IYEAR have to be specified. Details on weather data files can be found in van Kraalingen *et al.* (1991).

In TRANSLATION_GENERAL mode the use of weather data is optional.
In TRANSLATION_FSE mode weather data have to be selected.

*Use in combination with array variables*

A weather control variable cannot be an array.

*See also*

Section 3.2.4.4, Section 6.2, Section 6.3
The list of WEATHER control variables in Section 8.3.2.2

## 8.6 Fortran intrinsic functions

In Table 8.1, a list is given of the Fortran intrinsic functions supported by FST. For readers familiar with Fortran there is a minor difference in the use of generic function names. In Fortran, so-called generic functions accept various types of arguments: integer, real or double precision real. In FST most of these function accept real arguments only. The exception is the function MOD which, in FST, accepts only integer arguments. For real arguments AMOD has to be used.

Table 8.1 List of Fortran intrinsic functions with explanation. The input arguments I and J denote an integer constant, variable or expression. The input arguments X, X1, X2,... are real constants, variables or expressions.

| Fortran function | Explanation | Mathematical notation | Restrictions |
|---|---|---|---|
| `ABS(X)` | absolute value of x | $\lvert x\rvert$ | |
| `INT(X)` | the integer part of x, result is integer | $\operatorname{int}(x)$ | |
| `AINT(X)` | the integer part of x converted to real | $\operatorname{int}(x)$ | |
| `NINT(X)` | the nearest integer, result is integer | $\operatorname{int}(x)$ | |
| `ANINT(X)` | the nearest integer converted to real | $\operatorname{int}(x)$ | |
| `MAX(X1,X2,...,Xn)` | maximum value among the real arguments | $\max(x_1,x_2,\ldots,x_n)$ | $n \geq 2$ |
| `AMAX1(X1,X2,...,Xn)` | maximum value among the real arguments | $\max(x_1,x_2,\ldots,x_n)$ | $n \geq 2$ |
| `MIN(X1,X2,...,Xn)` | minimum value among the real arguments | $\min(x_1,x_2,\ldots,x_n)$ | $n \geq 2$ |
| `AMIN1(X1,X2,...,Xn)` | minimum value among the real arguments | $\min(x_1,x_2,\ldots,x_n)$ | $n \geq 2$ |
| `MOD(I,J)` | remainder of i/j with sign of i, result is integer | $i \bmod j$ | $j \neq 0$ |
| `AMOD(X,Y)` | remainder of x/y with sign of x, result is real | $x \bmod y$ | $y \neq 0$ |
| `COS(X)` | cosine of x, x in radians | $\cos(x)$ | |
| `COSH(X)` | hyperbolic cosine of x | $\cosh(x)$ | |
| `ACOS(X)` | arc cosine of x in range [0,π] | $\arccos(x)$ | $-1 \leq x \leq 1$ |
| `EXP(X)` | exponential function | $e^x$ | |
| `LOG(X)` | natural logarithm of x | $e\log x$ | $x > 0$ |
| `ALOG(X)` | natural logarithm of x | $e\log x$ | $x > 0$ |
| `LOG10(X)` | base 10 logarithm of x | $10\log x$ | $x > 0$ |
| `ALOG10(X)` | base 10 logarithm of x | $10\log x$ | $x > 0$ |
| `REAL(I)` | the real number nearest to integer I | | |
| `SQRT(X)` | square root of x | $\sqrt{x}$ | $x \geq 0$ |
| `SIN(X)` | sine of x, x in radians | $\sin(x)$ | |

| | | | |
|---|---|---|---|
| `SINH(X)` | hyperbolic sine of x | $\sinh(x)$ | |
| `ASIN(X)` | arc sine of x<br>in range [-π/2, π/2] | $\arcsin(x)$ | $-1 \le x \le 1$ |
| `TAN(X)` | tangent of x, x in radians | $\tan(x)$ | $x \bmod \frac{\pi}{2} \ne \frac{\pi}{4}$ |
| `TANH(X)` | hyperbolic tangent of x | $\tanh(x)$ | |
| `ATAN(X)` | arc tangent of x<br>in range [-π/2, π/2] | $\arctan(x)$ | |
| `ATAN2(X,Y)` | arc tangent of x/y<br>in range [-π/2, π/2] | $\arctan\left(\frac{x}{y}\right)$ | |

## 8.7 FST intrinsic functions

Table 8.2 on page 136 lists the FST intrinsic functions. Function calls can be included in mathematical expressions with the exception of the INTGRL function. For a few functions more information is given in separate subsections below.

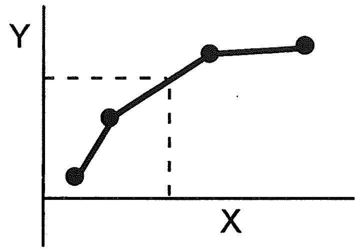
The arguments are described in the table below as variables. Input arguments can also be constants or expressions, however, possibly involving other function calls. There are three exceptions, however:
- The arguments of the INTGRL function call have to be variables or array variables.
- The first argument of the functions AFGEN and CSPLIN has to be an interpolation function defined with a FUNCTION statement.
- The input arrays of the array functions AR* and ELEMNT have to be FST arrays without calculations or subscript range.

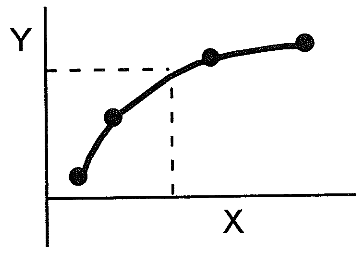
Ordinary real input arguments may also contain array variables with or without a subscript range indicated. In that case the entire statement is elementwise evaluated. The use of array variables in calculations is extensively discussed in Section 4.3.

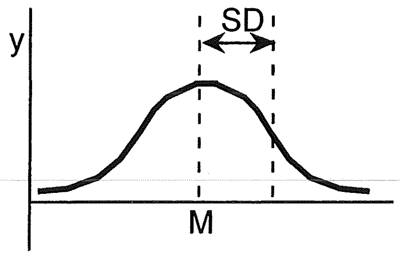
### 8.7.1 Simpson integration by means of ARSMPS

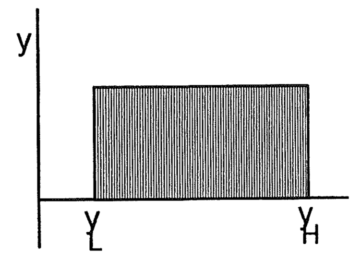
If the elements of an array contain the value of a function at a series of N equidistant x values, the integral can be estimated as a weighted sum of the N function values, multiplied by the interval width. The weights are 1.0 for the function values "in the middle of the range". At the edges of the interval, the weights differ from unity. The weights cannot be all unity since there are N points and N-1 intervals (See Chapter 4.1 in Press *et al.* ,1989).

Table 8.2 List of FST intrinsic functions with explanation. The input arguments K and L denote an integer constant, variable or expression. Array arguments are written as A or B. The symbol F means an FST interpolation function and all other input arguments are real constants, variables or expressions.

| FST function | Mathematical notation or Graph |
|---|---|
| `Y = AFGEN(F,X)`<br>Linear interpolation between (x,y) function points. See Section 3.4.<br>Y - Result of interpolation, estimated F(X)<br>F - Table of (x,y) values specified with FUNCTION statement<br>X - Value of independent variable |  |
| `Y = ARIMPR(A,B,K,L)`<br>Returns the improduct of a vector A and a vector B calculated over the subscript range K,...,L. See Section 4.3.3.<br>Y - Returned improduct<br>A - Array variable seen as vector<br>B - Array variable seen as vector<br>K - Start of subscript range<br>L - End of subscript range with $L^3K$ | $y = \sum_{i=K}^{L} A_i B_i$ |
| `Y = ARLENG(A,K,L)`<br>Returns the length of the vector with elements A(K),...., A(L) in (L-K+1)-dimensional space. See Section 4.3.3.<br>Y - Returned length<br>A - Array variable seen as vector<br>K - Start of subscript range<br>L - End of subscript range with $L^3K$ | $y = \sqrt{\sum_{i=K}^{L} A_i^2}$ |
| `Y = ARMAXI(A,K,L)`<br>Returns the maximum value among the array elements A(K),...., A(L). See Section 4.3.3.<br>Y - Resulting number<br>A - Array variable<br>K - Start of subscript range<br>L - End of subscript range with $L^3K$ | $y = \max\limits_{i=K}^{L} A_i$ |
| `Y = ARMEAN(A,K,L)`<br>Returns the arithmetic mean of the array elements A(K),...., A(L). See Section 4.3.3.<br>Y - Maximum value<br>A - Array variable<br>K - Start of subscript range<br>L - End of subscript range with L≥K | $y = \frac{\sum_{i=K}^{L} A_i}{L-K+1}$ |

| | |
|---|---|
| `Y = ARMINI(A,K,L)`<br>Returns the minimum value among the array elements A(K),...., A(L). See Section 4.3.3.<br>Y - Resulting number<br>A - Array variable<br>K - Start of subscript range<br>L - End of subscript range with $L^3K$ | $y = \min_{i=K}^{L} A_i$ |
| `Y = ARSMPS(A,K,L,D)`<br>Simpson's integral of a function over L-K closed intervals $(x_K,x_{K+1})$, $(x_{K+1},x_{K+2})$,...$(x_{L-1},x_L)$. At the L-K+1 points the function takes the values A(K),...., A(L). All intervals have equal width D. See Section 4.3.3 and Section 8.7.1.<br>Y - Approximate integral<br>A - Array containing function values<br>K - Start of subscript range<br>L - End of subscript range with L>K<br>D - Interval width | $y = D\sum_{i=K}^{L} w_i A_i$<br>in with the coefficients $w_i$ follow<br>- trapezoidal rule for n=2,3 (n=L-K+1)<br>- extended $1/n^3$ rule for n=4,5,6,7<br>- alternative extended Simpson's rule for $n^38$<br>See Chapter 4.1 in Press *et al.* (1989) |
| `Y = ARSTDV(A,K,L)`<br>Returns the standard deviation of the array elements A(K),...., A(L) seen as a sample. See Section 4.3.3.<br>Y - Returned standard deviation<br>A - Array variable<br>K - Start of subscript range<br>L - End of subscript range with L>K | $y = \sqrt{\frac{\sum_{i=K}^{L}\left(A_i - \overline{A}\right)^2}{L-K}}$ |
| `Y = ARSUMM(A,K,L)`<br>Returns the sum of the array elements A(K),...., A(L). See Section 4.3.3.<br>Y - Resulting sum<br>A - Array variable<br>K - Start of subscript range<br>L - End of subscript range with $L^3K$ | $y = \sum_{i=K}^{L} A_i$ |
| `Y = CSPLIN(F,X)`<br>Natural cubic splines interpolation between (x,y) function points according to Press *et al.* (1989). See Section 3.4 and Section 8.7.2.<br>Y - Result of interpolation, estimated F(X)<br>F - Table of (x,y) values specified with FUNCTION statement<br>X - Value of independent variable |  |
| `Y = ELEMNT(A,K)`<br>Returns value of the K-th element of array A after verifying its existence by comparing K with the declared array bounds. See Section 4.3.4.<br>Y - Returned element value<br>A - Array variable<br>K - Subscript | $y = A_K$ |

| | |
|---|---|
| `Y = FCNSW(X,Y1,Y2,Y3)`<br>Input switch. Y is set equal to Y1, Y2 or Y3 depending on the value of X.<br>Y - Returned as either Y1, Y2 or Y3<br>X - Control variable<br>Y1 - Returned value of Y if X<0<br>Y2 - Returned value of Y if X=0<br>Y3 - Returned value of Y if X>0 | $\begin{cases} y = y_1, & x < 0 \\ y = y_2, & x = 0 \\ y = y_3, & x > 0 \end{cases}$ |
| `Y = INTGRL(YI,YR)`<br>Integration command in the form of a function call. The algorithm of the numerical integration depends on the selected translation mode and driver. See Section 3.3 and 8.7.3.<br>Y - State variable<br>YI - Initial value of Y, must be a variable<br>YR- Rate of change, must be a variable | $y(t) = y(0) + \int_0^t \frac{dy(t')}{dt}\,dt$ |
| `Y = INSW(X,Y1,Y2)`<br>Input switch. Y is set equal to Y1 or Y2 depending on the value of X.<br>Y - Returned as either Y1 or Y2<br>X - Control variable<br>Y1 - Returned value of Y if X<0<br>Y2 - Returned value of Y if X³0 | $\begin{cases} y = y_1, & x < 0 \\ y = y_2, & x \geq 0 \end{cases}$ |
| `Y = LIMIT(XL,XH,X)`<br>Y is equal to X but limited between XL and XH<br>Y - Returned as X bounded on [XL,XH]<br>XL - Lower bound of X<br>XH- Upper bound of X | $\begin{cases} y = x, & x_l \leq x \leq x_h \\ y = x_l, & x < x_l \\ y = x_h, & x > x_h \end{cases}$ |
| `Y = NOTNUL(X)`<br>Y is equal to X but 1.0 in case of X=0.0. Note that X is evaluated without any tolerance interval.<br>Y - Returned result<br>X - Checked for being zero | $\begin{cases} y = x, & x \neq 0 \\ y = 1, & x = 0 \end{cases}$ |
| `Y = REAAND(X1,X2)`<br>Returns 1.0 if both input values are positive, otherwise Y=0.0. | $\begin{cases} y = 1, & x_1, x_2 > 0 \\ y = 0, & x_1 \leq 0 \text{ or } x_2 \leq 0 \end{cases}$ |
| `Y = REANOR(X1,X2)`<br>Returns 1.0 if both input values are less than or equal to zero, otherwise Y=0.0. | $\begin{cases} y = 1, & x_1, x_2 \leq 0 \\ y = 0, & x_1 > 0 \text{ or } x_2 > 0 \end{cases}$ |
| `Y = RGNORM(M,SD)`<br>Random number Generator which returns numbers with an univariate normal distribution. See Section 3.7 and 8.7.4.<br>Y - Returned random number<br>M - Mean of the normal distribution<br>SD- Standard deviation of the distribution |  |

| | |
|---|---|
| `Y = RGUNIF(YL,YH)`<br>Random number Generator which returns numbers with a uniform distribution on (YL,YH). See Section 3.7 and 8.7.4.<br>Y - Returned random number<br>YL - Lower bound of interval<br>YH - Upper bound of interval |  |

The function ARSMPS combines three different formulas for the integral. For 2 or 3 points there is nothing better than the trapezoidal rule. Up to 7 points the extended 1/n$^3$ rule is used and for N$^3$8, the alternative extended Simpson's rule is used (Chapter 4.1 in Press *et al.* ,1989). This is

$$\begin{cases} \mathrm{ARSMPS}(A,1,N,D) = D\left\{\frac{1}{2}A_1 + A_2 + \cdots + A_{N-1} + \frac{1}{2}A_N\right\} & , N=2,3 \\ \mathrm{ARSMPS}(A,1,N,D) = D\left\{\frac{5}{12}A_1 + \frac{13}{12}A_2 + A_3 + \cdots + A_{N-2} + \frac{13}{12}A_{N-1} + \frac{5}{12}A_N\right\}, & N=4,5,6,7 \\ \mathrm{ARSMPS}(A,1,N,D) = D\left\{\frac{17}{48}A_1 + \frac{59}{48}A_2 + \frac{43}{48}A_3 + \frac{49}{48}A_4 + A_5 + \cdots\right. & \\ \qquad\left.\cdots + A_{N-4} + \frac{49}{48}A_{N-3} + \frac{43}{48}A_{N-2} + \frac{59}{48}A_{N-1} + \frac{17}{48}A_N\right\} & , N \geq 8 \end{cases}$$

Note that for N=2, N=4 and N=8, the array elements with weight 1.0 (in the middle) are absent.

The following FST program uses function ARSMPS to calculate the integral of SIN(X) over the interval [0,$\pi$]. There are only INITIAL calculations and the time integration is dummy. In statement 5 the interval width is calculated. In statement 7 all x values and in statement 9 the integral over the x-interval [0,$\pi$] is estimated. The array size variable N is the number of points used.

```
0001    TITLE Test Simpson FST program structure
0002    ARRAY X(1:N), Y(1:N)
0003    INITIAL
0004    CONSTANT PI=3.14159265
0005    DELX = PI/REAL(N-1)
0006    ARRAY_SIZE N=20
0007    X = REAL(I-1) * DELX
0008    Y = SIN(X)
0009    SURF = ARSMPS(Y,1,N,DELX)
0010    PRINT SURF
0011    TIMER STTIME=0.0; FINTIM=1.0; DELT=1.0
0012    TRANSLATION_GENERAL DRIVER='EUDRIV'
0013    END
```

The result of this program, depends on the value of N. The following table gives an idea of the accuracy for the function SIN(X). The analytical result is 2.0.

| N | SURF |
|---|---|
| 4 | 1.9649 |
| 5 | 1.9887 |
| 6 | 1.9953 |
| 7 | 1.9977 |
| 8 | 2.0007 |
| 9 | 2.0004 |
| 10 | 2.0002 |

Note that for the integration of known functions more efficient methods are available (various forms of Gaussian integration). If the function values are simulated values on a regular grid, however, the use of ARSMPS is an easy way of estimating the total amount present.

## 8.7.2 Cubic spline interpolation with CSPLIN

The FST intrinsic function CSPLIN is based on Chapter 3.3 of Press *et al.* (1986). For each two points the coefficients of a third order polynomial are calculated in such a way that the first and second order derivative are also continuous in the given (x,y) points. The word "natural" refers to the assumption that the second order derivative is zero at the first and last point. This implies that a graph of the function is not curved at the first and last of the (x,y) points of the interpolation FUNCTION.

Note that there is no smoothing of data points and a warning is given on extrapolation outside the X-range of the defined function. Examples are given in Section 3.4 and Section 6.1.1.

## 8.7.3 The INTGRL function (and array variables)

A **state variable** S is formally calculated as

```
S = INTGRL (SI,SR)
```

in which SI is the initial value and SR the rate of change for the state variable. Unlike other functions, the INTGRL function cannot be part of a larger expression. The rules for the first argument of an INTGRL call are

- It has to be a variable, it cannot be a constant (like 0.0) or an expression.
- A variable calculated in the INITIAL section may act as initial value
- An INCON variable may act as initial value

The rules for the second argument of an INTGRL call are

- It has to be a variable, it cannot be a constant (like 0.0) or an expression like 2.0*SR
- A variable calculated in the DYNAMIC section may act as rate of change
- WEATHER and calendar data variables may act as rate of change
- A model PARAMETER may act as rate of change

The INTGRL function call can also be used to define a **state array**. An array S becomes a state array by writing

```
S = INTGRL (SI,SR)
```

which is exactly the same statement as for a scalar, non-array state. The rules for a state array are

- A state array is defined by means of an INTGRL function call in a single substatement ; unlike other array calculations, the declared range cannot be split up among a different substatements.
- The INTGRL function call cannot be part of a larger expression.
- The first argument is a variable and cannot be a constant or expression.
- The first argument is either a scalar variable or an array variable belonging to the same array family as the state variable and with exactly the same declared bounds.
- The first argument must be either a calculated (array) variable or an INCON (array) variable.
- If the first argument is a calculated variable, it must be calculated in the INITIAL section.
- The second argument is a variable and cannot be a constant or expression.
- The second argument is either a scalar variable or an array variable belonging to the same array family as the state variable and with exactly the same declared bounds.
- The second argument must be a calculated (array) variable, a weather variable or a parameter (array).
- If the second argument is a calculated variable, it must be calculated in the DYNAMIC section.

A often needed structure is

```
ARRAY S(1:N), SI(1:N), SR(1:N)
...
DYNAMIC
S = INTGRL (SI,SR)
```

This makes S a state array, SI an initial value array and SR a rate array.

It also happens often that all elements of a state array have to be initialized at a single value. Then there is no need for declaring and defining an entire array of initial values. The following structure is in accordance with the above rules

```
ARRAY S(1:N), SR(1:N)
...
INCON ZERO=0.0
DYNAMIC
S = INTGRL (ZERO,SR)
```

The first argument of the INTGRL function is scalar.

Although an INTGRL function call has the form of a calculation statement (and has to be in the DYNAMIC section), it is not sorted with the other calculations. An INTGRL function call acts as a declaration of the relation between a state variable, its initial value and its rate of change. See also Section 3.3.

## 8.7.4 The random number generators RGUNIF and RGNORM

The algorithm for generating random numbers with a uniform distribution (RGUNIF) originates from L'Ecuyer (1988). It is implemented in Bratley *et al.*, 1983, (UNIFL), and in Press *et al.*, 1992 (RAN2). Note that the value 1.0 is never returned.

The algorithm used in RGNORM is known as the Box-Muller method for generating random normal deviates (Box & Muller, 1958). Two random normal deviates are derived from two uniform deviates using the inverse of the bivariate normal distribution. The method is also well explained in Bratley *et al.* (1983).

We have adapted the procedures for seed generation to the FST rules for random number generation as explained in Section 3.7. In the linked DRIVERS library there is a seed generating function RGRSET which is called during the initial phase of model execution. The arguments of this function are the TIMER variables RGINIT and RGSEED. The communication between RGRSET and RGUNIF takes place by means of a small common block.

# Chapter 9

# The generated Fortran

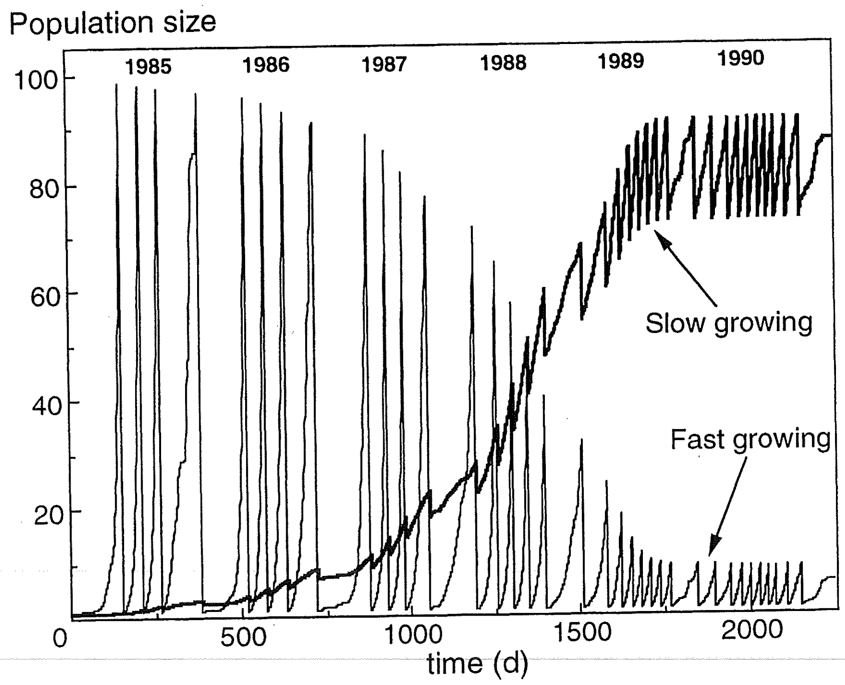


*Figure 9.1 on the title page of this chapter.* Two exponentially growing populations with relative growth rates depending on daily average temperature in the same way. From the FST program in Listing 9.1 on page 146 the Fortran program in Listing 9.3 was generated by the FST translator. To the Fortran program a state event has been added “by hand” according to the description in Section 9.2.4. The event occurs when the sum of the two population sizes reaches 100.0. The slow-growing population is then reduced by 20% and the fast-growing one is cut down to its initial value 1.0. The state events are a form of interaction between the two species. The slow- growing one suffers little from each event and is able to overgrow the fast-growing population in about 5 years.

# 9 The generated Fortran sources

This chapter is especially meant for those readers who want to use the generated Fortran as a starting point for further modeling work. Some introductory remarks on the generated Fortran have been made in Section 7.4.2. Here we assume a serious interest and we give some background on the structure of the drivers and the model routine.

The scheme in Figure 7.2 on page 88 showed already that the generated Fortran code needs to be linked with the libraries DRIVERS, TTUTIL and WEATHER. Subroutines called from the FST program need to be included in its Fortran section or need to be linked with the translated program.

The TTUTIL library is primarily a library of input and output subroutines and has been documented in Rappoldt & van Kraalingen (1990) and van Kraalingen & Rappoldt (1996). The I/O subroutines are called from the driver and from the generated Fortran module itself for reading values from the generated data files and for writing the output files. The WEATHER library has been documented by van Kraalingen *et al.* (1991).

The DRIVERS library contains the simulation drivers for both translation modes. The DRIVERS library also contains most of the FST intrinsic functions. The intrinsic functions LIMIT, INSW, REANOR, REAAND and AFGEN are part of TTUTIL for historical reasons (AFGEN is renamed by the translator in LINT2).

The Fortran generated in FSE mode has been documented in van Kraalingen (1995) and is discussed briefly in Section 9.1. The GENERAL drivers are discussed in greater detail in Section 9.2. Section 9.2.4 describes how time and state events can be added to a generated Fortran model.

## 9.1 FSE module

The FST translator generates a standard FSE program. The structure of FSE simulation models has been fully documented in van Kraalingen (1995). Here, we just explain the function of the various Fortran subroutines and give an example which can be compared in Section 9.2 with the Fortran program generated in GENERAL mode.

### 9.1.1 Overview of the FSE mode Fortran structure

The hierarchy and function of the Fortran subroutines in FSE translation mode is

- The generated main program **MAIN** which calls the FSE driver.
- Subroutine **FSE** from the DRIVERS library. This subroutine organizes the simulation. Initialization, calculation of rates of change, integration and termination of each model run are realized by calls to the subroutine MODEL<u>S</u>, which in turn calls all FSE modules under it. The FSE driver calls the WEATHER library to get weather data and supplies these to the models. The driver takes care of the timing and the calendar variables and also contains the loop over all reruns. This rerun loop is based on calls to subroutines from I/O library TTUTIL.
- The generated subroutine **MODEL<u>S</u>**. In case of FST-generated Fortran, the subroutine MODELS is a trivial one. It contains little more than a call to the actual FSE module MODEL. The real function of MODELS, however, is to call <u>several</u> FSE modules, which may interact. The different hand-written, or generated FSE modules are not called directly from the driver, via the intermediate subroutine MODELS, which represents the integrated model from the drivers point of view. The interaction between the various FSE modules may be organized via parameter exchange or common blocks added to the modules.

- The generated subroutine **MODEL** is an FSE module and represents the actual model.
- Below the model subroutine, there are the subroutines called from the model, the intrinsic functions, and the utilities for output and for reading data from MODEL.DAT.

## 9.1.2 Example of a generated FSE module

The example program in Listing 9.1 is an adapted version of the FST model in Listing 2.2 on page 12. Here, the weather dependent exponential growth is simulated for two populations, making use of arrays for population size, initial value, rate of change and maximum relative growth rate. The two populations are independent. The FINISH condition will stop the simulation as soon as one of the populations reaches 200.0 times its initial value.

Listing 9.1 Model of two populations growing exponentially. It is an extension of the model in Listing 2.2 for a single population. Note that the temperature dependent interpolation function REDUCE is used to reduce the maximum relative growth rate RGRMAX of both populations. Between 20°C and 30°C the function is 1.0 and there is no reduction. Outside that range the function is below 1.0.

```
TITLE  Temperature Dependent Growth
TITLE for N independent populations
DECLARATIONS
   ARRAY X(1:N), IX(1:N), RX(1:N)                <-Program suitable for N populations
   ARRAY RGRMAX(1:N)
MODEL
ARRAY_SIZE N=2                                   <-Number of populations
  INITIAL
    IX = ABS(B)
  PARAMETER B=1.0 ; RGRMAX(1)=0.01 ; RGRMAX(2:N)=0.1     <-RGRMAX values
  DYNAMIC
    X  = INTGRL (IX,RX)
    RX = RGRMAX * AFGEN(REDUCE,TDMEAN) * X       <-Calculation of growth rate

*   sum of populations
    XSUM = ARSUMM (X,1,N)                        <-The sum of all populations

* empirical function reducing the relative
* growth for extreme daily average temperatures
  FUNCTION REDUCE = -20.0, 0.0,  0.0, 0.0, 15.0, 0.9, 20.0, 1.0, ...
                     30.0, 1.0, 40.0, 0.8, 42.0, 0.0, 50.0, 0.0

* estimate the average daily temperature
  TDMEAN = (TMMN + TMMX) / 2.0
  WEATHER CNTR='NLD' ; ISTN=1 ; IYEAR=1985
  WEATHER WTRDIR='C:\SYS\WEATHER\'

* integration of average temperature
  TCUM = INTGRL(ZERO,TDMEAN)                     <-The average temperature is integrated
  INCON ZERO=0.0

  TERMINAL
*   average temperature in simulation period
    TMEAN = TCUM / (TIME-STTIME)                 <-Mean temperature over simulated period

  FINISH X > 200.0*IX
  TIMER STTIME=1.0 ; FINTIM=3000.0 ; DELT=1.0
  TIMER  PRDEL=1.0
  PRINT X,XSUM,TDMEAN,TMEAN
  TRANSLATION_FSE                                <-FSE translation selected
END
```

The FSE module MODEL generated by the FST translator for this simple growth model is given in Listing 9.2. It is part of the file MODEL.FOR which also contains the generated MAIN program and the generated subroutine MODELS. A few remarks on the Fortran code:

- The TIMER variables STTIME, FINTIM and DELT, the weather and calendar data and the driver supplied variable TIME are made available to the Fortran model as input parameters.
- The IMPLICIT NONE statement in line 55 is a non-standard Fortran statement. It forces the user to declare all variables.
- The IF statement in line 122 and 123 is a safeguard against values of N too low for the subscript ranges used (see Section 4.3.1.3). Array size variables are Fortran PARAMETERS (line 70), which can be changed by the user corresponding changes are made in the file MODEL.DAT.
- The error routine FATALERR represents a non-standard subroutine name.
- There are several groups of declarations (line 57-96). All variables of the model are systematically declared. The SAVE statement in line 97 guarantees that the values of all variables retain their values between successive calls to MODEL.
- The string WUSED in line 101 is a code for the use of weather variables in the model. It contains 6 characters, one character for each weather variable, which are either "-" for a non-used variable and "U" for a used one. The lines 103-114 compare WUSED with the weather status string WSTAT supplied by the FSE driver.
- The rest of the program is organized in 4 sections: INITIAL, RATES of change, INTEGRATION and TERMINAL. They contain the calculation statements from the FST program in a computational order.
- The output section (line 179-184) is at the end of the rate of change section. At that point a consistent set of values is available for TIME, all states, rates of change and other calculated variables.
- Calls to the FST intrinsic functions may differ from the FST source code. Examples are in line 164 and line 169. The FST translator has added arguments for improved runtime error messages or for runtime checks on array subscript ranges.
- The program does not contain any write or read statements. All Input/Output operations make use of the library TTUTIL. This improves the readability program and greatly simplifies writing it.

Listing 9.2 A Fortran compiler listing of subroutine MODEL in FSE format. This subroutine was generated by the FST translator from the program in Listing 9.1 on page 146.

```
0001    *----------------------------------------------------------------------*
0002    * SUBROUTINE MODEL                                                     *
0003    * Authors: FST translator                                              *
0004    * Date    :                                                            *
0005    * Purpose: This subroutine is the translated FST file                  *
0006    *                                                                      *
0007    * FORMAL PARAMETERS:  (I=input,O=output,C=control,IN=init,T=time)      *
0008    * name   type meaning                                    units  class *
0009    * ----   ---- -------                                    -----  ----- *
0010    * ITASK   I4  Task that subroutine should perform          -       I  *
0011    * IUNITD  I4  Unit of input file with model data           -       I  *
0012    * IUNITO  I4  Unit of output file                          -       I  *
0013    * IUNITL  I4  Unit number for log file messages            -       I  *
0014    * FILEIT  C*  Name of timer input file                     -       I  *
0015    * FILEIN  C*  Name of file with input model data           -       I  *
0016    * OUTPUT  L4  Flag to indicate if output should be done    -       I  *
0017    * TERMNL  L4  Flag to indicate if simulation is to stop    -     I/O  *
0018    * DOY     R4  Day number within year of simulation (REAL)   d      I  *
0019    * IDOY    I4  Day number within year of simulation (INTEGER) d     I  *
0020    * YEAR    R4  Year of simulation (REAL)                     y      I  *
0021    * IYEAR   I4  Year of simulation (INTEGER)                  y      I  *
0022    * STTIME  R4  Start time of simulation (=day number)        d      I  *
0023    * FINTIM  R4  Finish time of simulation (=day number)       d      I  *
```

```
* DELT    R4  Time step of integration                     d        I  *
* LAT     R4  Latitude of site                         dec.degr.    I  *
* LONG    R4  Longitude of site                        dec.degr.    I  *
* ELEV    R4  Elevation of site                            m        I  *
* WSTAT   C6  Status code from weather system              -        I  *
* WTRTER  L4  Flag whether weather can be used by model    -        O  *
* RDD     R4  Daily shortwave radiation                 J/m2/d      I  *
* TMMN    R4  Daily minimum temperature               degrees C     I  *
* TMMX    R4  Daily maximum temperature               degrees C     I  *
* VP      R4  Early morning vapour pressure                kPa      I  *
* WN      R4  Daily average windspeed                      m/s      I  *
* RAIN    R4  Daily amount of rainfall                    mm/d      I  *
*                                                                      *
* Fatal error checks: if one of the characters of WSTAT = '4',         *
*                     indicates missing weather                        *
* Warnings          : none                                             *
* Subprograms called: models as specified by the user                  *
* File usage        : IUNITD,IUNITD+1,IUNITO,IUNITO+1,IUNITL           *
*----------------------------------------------------------------------*

      SUBROUTINE MODEL (ITASK , IUNITD, IUNITO, IUNITL,
     &                  FILEIT, FILEIN,
     &                  OUTPUT, TERMNL,
     &                  DOY   , IDOY  , YEAR  , IYEAR,
     &                  TIME  , STTIME, FINTIM, DELT ,
     &                  LAT   , LONG  , ELEV  , WSTAT, WTRTER,
     &                  RDD   , TMMN  , TMMX  , VP   , WN, RAIN)

*     Title of the program
*     <fill in your title here>

      IMPLICIT NONE

*     Formal parameters
      INTEGER  ITASK , IUNITD, IUNITO, IUNITL, IDOY, IYEAR
      LOGICAL  OUTPUT, TERMNL, WTRTER
      CHARACTER*(*) FILEIT, FILEIN, WSTAT
      REAL DOY, YEAR, TIME, STTIME, FINTIM, DELT
      REAL LAT, LONG, ELEV, RDD, TMMN, TMMX, VP, WN, RAIN

*     Standard local variables
      INTEGER IWVAR
      CHARACTER WUSED*6

*     Array size variables
      INTEGER N
      PARAMETER (N=2)

*     State variables, initial values and rates
      REAL TCUM, ZERO, TDMEAN
      REAL X(1:N), IX(1:N), RX(1:N)

*     Model parameters
      REAL B
      REAL RGRMAX(1:N)

*     Other calculated variables
      REAL TMEAN, XSUM

*     Interpolation functions used in AFGEN en CSPLIN functions
      INTEGER IMREDU, ILREDU
      PARAMETER (IMREDU=40)
      REAL REDUCE(IMREDU)
```

```


*     Declarations and values of constants
*     none

*     Used functions
      REAL LINT2, ARSUMM, INTGRL

*     DO loop counter for array integrals
      INTEGER I
      SAVE

*     Code for the use of RDD, TMMN, TMMX, VP, WN, RAIN  (in that order)
*     a letter 'U' indicates that the variable is Used in calculations
      DATA WUSED/'-UU---'/

*     Check weather data availability
      IF (ITASK.EQ.1.OR.ITASK.EQ.2.OR.ITASK.EQ.4) THEN
         DO 10 IWVAR=1,6
*           is there an error in the IWVAR-th weather variable ?
            IF (WUSED(IWVAR:IWVAR).EQ.'U' .AND.
     &          WSTAT(IWVAR:IWVAR).EQ.'4') THEN
               WTRTER = .TRUE.
               TERMNL = .TRUE.
               RETURN
            END IF
10       CONTINUE
      END IF

      IF (ITASK.EQ.1) THEN

*        Initial section
*        ===============

*        Check values of array size variables
         IF (N.LT.2) CALL FATALERR
     &      ('MODEL','Array size symbol N is too small')

*        Open input file
         CALL RDINIT (IUNITD, IUNITL, FILEIN)

*        Read initial states
         CALL RDSREA ('ZERO', ZERO)

*        Read model parameters
         CALL RDSREA ('B', B)
         CALL RDFREA ('RGRMAX', RGRMAX, N, N)

*        Read LINT functions
         CALL RDAREA ('REDUCE', REDUCE, IMREDU, ILREDU)

         CLOSE (IUNITD)

*        Initial calculations
         DO 20 I=1,N
            IX(I) = ABS(B)
20       CONTINUE

*        Initially known variables to output
*        none

*        Send titles to OUTCOM
         CALL OUTCOM (' Temperature Dependent Growth')
```

```
            CALL OUTCOM ('for N independent populations')

*           Initialize state variables
            TCUM = ZERO
            DO 30 I=1,N
               X(I) = IX(I)
30          CONTINUE

         ELSE IF (ITASK.EQ.2) THEN

*           Rates of change section
*           ======================

*     sum of populations
            XSUM = ARSUMM ('X',X,1,N,1,N)

*   estimate the average daily temperature
            TDMEAN = (TMMN + TMMX) / 2.0
            DO 40 I=1,N
               RX(I) = RGRMAX(I) * LINT2('REDUCE',REDUCE,ILREDU,TDMEAN) *
     $          X(I)
40          CONTINUE

*           Finish conditions

            DO 50 I=1,N
               IF (X(I).GT.200.0*IX(I)) TERMNL = .TRUE.
50          CONTINUE

*           Output
            IF (OUTPUT) THEN
               CALL OUTDAT (2, 0, 'XSUM', XSUM)
               CALL OUTDAT (2, 0, 'TDMEAN', TDMEAN)
               CALL OUTAR2 ('X',X,1,N,1,N)
            END IF

         ELSE IF (ITASK.EQ.3) THEN

*           Integration section
*           ===================

*   integration of average temperature
            TCUM = INTGRL (TCUM, TDMEAN, DELT)
            DO 60 I=1,N
               X(I) = INTGRL (X(I), RX(I), DELT)
60          CONTINUE

         ELSE IF (ITASK.EQ.4) THEN

*           Terminal section
*           ================

*           Terminal calculations
*     average temperature in simulation period
            TMEAN = TCUM / (TIME-STTIME)

*           Terminal output
            CALL OUTDAT (2, 0, 'TMEAN', TMEAN)

*           Printplot output
*           none
         END IF

```

```
      RETURN
      END
```

## 9.2 General module

After an overview of the Fortran subroutine structure, the internal structure of the drivers EUDRIV and RKDRIV is explained in Section 9.2.2. Section 9.2.3 deals with the structure of the generated model subroutine MODEL. In Section 9.2.4 we show how time and state events can be added to generated Fortran.

A somewhat older version of both drivers has been extensively discussed by Leffelaar *et al.* (1993). Appendix 10.1 of that book contains the Fortran source code of the drivers. The changes made during the subsequent development of FST are small. The older drivers do not work, however, in combination with an FST-generated MODEL routine.

### 9.2.1 Overview of the GENERAL mode Fortran structure

The hierarchy and function of the Fortran subroutines in GENERAL translation mode is
- Main program **MAIN** which opens a logfile, opens the formatted output file RES.DAT, selects a block of free unit numbers for file I/O, calls the RERUNS routine and finally deletes all temporary files. This main program is easily changed into a subroutine called by other Fortran programs. MAIN is independent of the actual model, but it is generated by FST since a main program cannot be included in a linked library.
- Subroutine **RERUNS** from the DRIVERS library. This subroutine sets I/O unit numbers in more detail, reads the rerun file RERUNS.DAT and contains the loop over the model runs. For each run it calls either EUDRIV or RKDRIV.
- The simulation driver **EUDRIV** or **RKDRIV** from the DRIVERS library. The driver organizes the simulation run. It initializes the MODEL routine, takes care of the integration and terminates the run. It uses the simulation control variables to set the TIME variable and to determine output times. The driver terminates the model run if MODEL reports a valid terminal condition.
- The generated model subroutine **MODEL** is called by the drivers, both directly and, in case of RKDRIV, also via the Runge-Kutta integration routines.
- Below the actual model subroutine, there are the Fortran subroutines belonging to the model, the intrinsic functions, and the utilities for reading data from MODEL.DAT.

Hence, only two of these units are generated by FST, the main program MAIN and the actual model subroutine MODEL. Both are contained in the file MODEL.FOR created by the translator.

### 9.2.2 The driver subroutines EUDRIV and RKDRIV

#### 9.2.2.1 Arguments of the drivers

These two drivers carry out a single model run. The driver EUDRIV uses Euler integration with a fixed time step. The driver RKDRIV calls the subroutines RKQCA and RK4A for Runge-Kutta integration. These integration routines are adapted versions of the subroutines RKQC and RK4 from Press *et al.* (1986). The arguments of the two drivers EUDRIV and RKDRIV are the same.

| | |
|---|---|
| IUL | An integer input argument. It is the unit number of an open log file. |
| IUR | An integer input argument. The unit number of open results file. If the flag OUTDFL is .TRUE., the TTUTIL output routine OUTDAT will open a file RES.DAT if unit IUR is not found to be open already. Note that OUTDAT also IUR+1. |
| IUT | An integer input argument. First of 2 free unit numbers used by the driver to read data from the timer file TIMER.DAT. |

| | |
|---|---|
| IUM | An integer input argument. First of at least 4 free unit numbers for use by the external model subroutine. |
| ITRACE | An integer input argument. The level of output to log file. The variable ITRACE corresponds with the TRANSLATION_GENERAL variable TRACE (see Section 7.5.5). |
| OUTDFL | A logical input argument. If .TRUE., the driver initial and terminal calls to the TTUTIL subroutine OUTDAT, a facility used in driver use by FST. If the model subroutine does not use OUTDAT for ouput, the flag can be .FALSE. |
| MODEL | Name of EXTERNAL model subroutine. |

#### 9.2.2.2 Interface with the EXTERNAL subroutine MODEL
There are three ways in which the drivers communicate with the user supplied (or FST-generated) subroutine MODEL. The first way consists of the arguments in the call to MODEL. The drivers themselves and the numerical subroutines used by RKDRIV, use the same calls to the external model routine. All these calls look like

```
CALL MODEL (ITASK,OUTPUT,TIME,STATE,RATE,SCALE,NDEC,NEQ)
```

The INTEGER argument ITASK has the value 1 for initialization, 2 for rate calculation and 4 for run termination. OUTPUT is a flag turned on by the driver at output times. TIME is the simulated time.

The array STATE contains the system status variables. The external model sets them at their initial values during the ITASK=1 call. The rates of change in the RATE array are calculated by the external model for ITASK=2. NDEC is the declared size of the arrays STATE and RATE. The integer NEQ (Number of EQuations) is the actual size of these arrays and is set by MODEL at initialization.

Also the SCALE array has declared length NDEC and actual length NEQ. Its elements are given a value in the external MODEL during initialization and should later not be changed by MODEL. The SCALE array is used by RKDRIV for the evaluation of the integration accuracy (see Section 9.2.2.3). EUDRIV does not use SCALE.

The second way of communication is via the common block /DRVCOM/. The declarations of the common variables are

```
      INTEGER          IULOG,  IUMOD, IURES,   KEEP, TRACE
      REAL            STTIME, FINTIM,  DELT, DELMAX,   EPS, DELDID
      LOGICAL         TERMNL
      COMMON /DRVCOM/  IULOG,  IUMOD, IURES,   KEEP, TRACE, TERMNL,
     $                STTIME, FINTIM,  DELT, DELMAX,   EPS, DELDID
```

This common /DRVCOM/ communicates unit numbers (copies of driver input arguments IUL, IUM and IUR), the TIMER variables STTIME, FINTIM and DELT, the TRANSLATION_GENERAL variables TRACE, DELMAX and EPS and the size of the last time step DELDID supplied by the driver. There are two more variables. The integer variable KEEP is equal to 1 at precisely one rate call (ITASK=2) at the beginning of each time step. Otherwise KEEP is 0. The logical flag TERMNL can be set .TRUE. by the user model. If that happens, the driver will terminate the simulation run in an orderly way.

The third way of communication is via the common block /DRVEVT/ for time and state events. Time and state events are not possible in FST. If events are added to the generated Fortran, however, this common block is also included in MODEL (see Section 9.2.4.1).

#### 9.2.2.3 The array SCALE ; accuracy control by RKDRIV
The elements of array SCALE must get a value during the INITIAL call to MODEL and should later not be changed by MODEL. A generated MODEL routine reads the SCALE array from the input file

MODEL.DAT. The translator has written values for the SCALE array in that file: NEQ times the value zero.

A zero value of a certain element SCALE(I) means that the estimated integration error in STATE(I) is taken relative, i.e. is divided by the value of STATE(I). The relative error is then compared with the value of the TRANSLATION_GENERAL variable EPS or its default. For small values of STATE(I), however, this would lead to unrealistic accuracy requirements or even to division by zero. Therefore, if the absolute value of STATE(I) is less then 1.0, the estimated integration error is taken as an absolute value. Hence, for small values of STATE(I), the absolute integration error is compared with EPS, which leads to a weak accuracy criterion for low STATE(I) values.

If the INITIAL value of STATE(I) is larger than zero, the estimated integration error in STATE(I) is always taken relative to the value of SCALE(I). Hence, in that case the value of SCALE(I) is used to indicate an order of magnitude for the various STATE variables. This explains the name SCALE.

The integration error is estimated by integration routine RKQCA (called by RKDRIV) for each time step by comparing the result of a full step with two half ones (See for details Press *et al.*, 1986).

In case of integration problems with an FST-generated MODEL, it may be necessary to set certain SCALE array elements to non-zero values. This requires a proper understanding of the order in which the STATE, RATE and SCALE array elements are related to the user variable names in MODEL (see Section 9.2.3 below).

### 9.2.3 The structure of a GENERAL module MODEL

The arguments of the user subroutine MODEL have been discussed already in Section 9.2.2.2. The main difference with an FSE module is that there is no integration call (with ITASK=3) and that the user supplied state and rate variables are copied to and from the overall STATE and RATE array used by the GENERAL drivers. How this works can be best explained by means of the example MODEL in Listing 9.3 on page 154. That generated Fortran is based on the same FST model as the FSE module in Listing 9.2 on page 147 (except of course the FST translation mode).

The line numbers in the following remarks refer to Listing 9.3 on page 154.

- The IMPLICIT NONE statement in line 25 is a non-standard Fortran statement. It forces the user to declare all variables.
- The TIMER variables STTIME, FINTIM and DELT are part of the common block /DRVCOM/ (line 32-51). See Section 9.2.2.2 for some more remarks on this common.
- Again, there are several groups of declarations (line 53-91). All variables of the model are systematically declared. The SAVE statement in line 92 guarantees that the values of all variables retain their values between successive calls to the model.
- The string WUSED has the same function as in the FSE module. The GENERAL drivers, however, do not supply weather and calendar variables. They come out subroutine WTRINT (WeaTher INTerface) called in line 99. This subroutine from the DRIVERS library also takes care of the check on WUSED and sets the WTRTER flag in case of error.
- The rest of the program is organized in 3 sections: INITIAL, RATES of change and TERMINAL. They contain the calculation statements from the FST program in a computational order.
- The IF statement in line 117 and 118 is a safeguard against values of N too low for the subscript ranges used (see Section 4.3.1.3). Array size variables are Fortran PARAMETERS (line 59), which can be changed by the user corresponding changes are made in the file MODEL.DAT.
- The error routine FATALERR represents a non-standard subroutine name.
- At the end of INITIAL, in lines line 167-173, all local state variables (with user supplied names) are copied into the overall STATE array. At the beginning of the RATES of change section, in line

177-185, the values of the STATE array calculated by the driver are copied into the local variables.

- The sections for copying to and from the STATE array give information on the order in which FST has placed the local state variables in the overall state array. If non-zero SCALE values are needed (see Section 9.2.2.3), this order is essential. The SCALE array is read in the INITIAL section from input file MODEL.DAT (line 143).
- The output section (line 208-214) is at the end of the rate of change section. At that point a consistent set of values is available for TIME, all states, rates of change and other calculated variables.
- At the end of the RATES of change section, in lines 216-222, the calculated rates of change are copied into the overall RATE array.
- Terminal calculation may require both state variables and rates of change. Therefore, the STATE and the RATE array are copied to the local variables at the beginning of the TERMINAL section (line 228-240).
- Calls to the FST intrinsic functions may differ from the FST source code. Examples are in line 190 and line 195. The FST translator has added arguments for improved runtime error messages or for runtime checks on the subscript range of arrays.
- The program does not contain any write or read statements. All Input/Output operations make use of the library TTUTIL. This improves the readability program and greatly simplifies writing it.

Listing 9.3 A Fortran compiler listing of subroutine MODEL in GENERAL format. This subroutine was generated from the program in Listing 9.1 on page 146 using TRANSLATION_GENERAL.

```
0001  ************************************************************************
0002  *                 TRANSLATED SIMULATION MODEL                           *
0003  ************************************************************************
0004        SUBROUTINE MODEL (ITASK,OUTPUT,TIME,STATE,RATE,SCALE,NDEC,NEQ)
0005
0006  *     Model subroutine for use with driver EUDRIV or RKDRIV
0007  *
0008  *     ================ TITLE of the FST model ================
0009  *      Temperature Dependent Growth
0010  *     for N independent populations
0011  *
0012  *     The STANDARD (!!) parameter list of this model subroutine:
0013  *
0014  *     ITASK  - task of model routine                      I
0015  *     OUTPUT = .TRUE. output request                      I
0016  *     TIME   - time                                       I
0017  *     STATE  - state array of model                      I/O
0018  *     RATE   - rates of change belonging to STATE        I/O
0019  *     SCALE  - size scale of state variables             I/O
0020  *     NDEC   - declared size of arrays                    I
0021  *     NEQ    - Number of state variables, for ITASK=1     O
0022  *                                          otherwise      I
0023
0024  *     all variables have to be declared
0025        IMPLICIT NONE
0026
0027  *     Formal parameters
0028        INTEGER ITASK, NDEC, NEQ
0029        REAL TIME,STATE(NDEC),RATE(NDEC),SCALE(NDEC)
0030        LOGICAL OUTPUT
0031
0032  *     Declarations of the variables in common /DRVCOM/
0033  *     IULOG  - logfile unit number
0034  *     IUMOD  - first of a free block of unit numbers used by MODEL
0035  *     IURES  - unit number used for OUTDAT when OUTDFL is up
0036  *     KEEP   = 1: rate call (ITASK=2) at begin of new time step
```

```
*             = 0: rate calls during numerical integration by driver
*     TRACE  - integration logging level of driver
*     TERMNL - terminal flag, may be set by model
*     STTIME - start of simulation
*     FINTIM - end of simulation
*     DELT   - The time step on file TIMER.DAT
*     DELMAX - The maximum time step on file TIMER.DAT
*     EPS    - Accuracy level on file TIMER.DAT
*     DELDID - size of last completed time step
*
      INTEGER          IULOG,  IUMOD, IURES,   KEEP, TRACE
      REAL            STTIME, FINTIM,  DELT, DELMAX,   EPS, DELDID
      LOGICAL         TERMNL
      COMMON /DRVCOM/  IULOG,  IUMOD, IURES,   KEEP, TRACE, TERMNL,
     $                STTIME, FINTIM,  DELT, DELMAX,   EPS, DELDID

*     Number of state variables NSV
      INTEGER NSV
      PARAMETER (NSV=3)

*     Array size variables
      INTEGER N
      PARAMETER (N=2)

*     State variables, initial values and rates
      REAL TCUM, ZERO, TDMEAN
      REAL X(1:N), IX(1:N), RX(1:N)

*     Model parameters
      REAL B
      REAL RGRMAX(1:N)

*     Other calculated variables
      REAL TMEAN, XSUM

*     Interpolation functions used in AFGEN en CSPLIN functions
      INTEGER IMREDU, ILREDU
      PARAMETER (IMREDU=40)
      REAL REDUCE(IMREDU)


*     Declarations and values of constants
*     none

*     Variables supplied by the weather system
      REAL LAT, LONG, ELEV, YEAR, DOY, RDD, TMMN, TMMX, VP, WN, RAIN
*     Variables for check on weather data
      CHARACTER*6 WUSED
      LOGICAL WTRTER

*     Used functions
      REAL LINT2, ARSUMM

*     DO loop counter and STATE/RATE array subscript
      INTEGER I,IARELN
      SAVE
*     Code for the use of RDD, TMMN, TMMX, VP, WN, RAIN  (in that order)
*     a letter 'U' indicates that the variable is Used in calculations
      DATA WUSED/'-UU---'/

*     Get weather and calendar data
      IF (ITASK.EQ.1.OR.ITASK.EQ.2.OR.ITASK.EQ.4) THEN
         CALL WTRINT (ITASK, IUMOD+2, IULOG, IURES, WUSED, WTRTER,
     $    TIME, LAT, LONG, ELEV, YEAR, DOY,
     $    RDD, TMMN, TMMX, VP, WN, RAIN)
         IF (WTRTER) THEN
*           weather data error; prevent initial error on NEQ
            NEQ    = NSV
            TERMNL = .TRUE.
            RETURN
         END IF
      END IF


      IF (ITASK.EQ.1) THEN

*        Initial section
*        ===============

*        Check values of array size variables
         IF (N.LT.2) CALL FATALERR
     &      ('MODEL','Array size symbol N is too small')

*        Check size of NDEC against number of states NSV
         IF (NDEC.LT.NSV) THEN
*           driver capacity too low ; stop program
            WRITE (*,'(A,I4,/,A,I4)')
     $       '  The number of state variables is',NSV,
     $       ' and the capacity of the driver is',NDEC
            CALL FATALERR ('MODEL','Driver capacity too low')
         END IF

*        Open input file
         CALL RDINIT (IUMOD,IULOG,'MODEL.DAT')

*        Read initial states
         CALL RDSREA ('ZERO', ZERO)

*        Read model parameters
         CALL RDSREA ('B', B)
         CALL RDFREA ('RGRMAX', RGRMAX, N, N)

*        Read LINT/CSPLIN interpolation functions
         CALL RDAREA ('REDUCE', REDUCE, IMREDU, ILREDU)

*        Read SCALE array and close datafile
         CALL RDFREA ('SCALE',SCALE,NDEC,NSV)
         CLOSE (IUMOD)

*        Set number of state variables
         NEQ = NSV

*        Initial calculations
         DO 20 I=1,N
            IX(I) = ABS(B)
20       CONTINUE

*        Initially known variables to output
*        none

*        Send title(s) to OUTCOM
         CALL OUTCOM (' Temperature Dependent Growth')
         CALL OUTCOM ('for N independent populations')

*        Initialize state variables
         TCUM = ZERO
         DO 30 I=1,N
            X(I) = IX(I)
30       CONTINUE

*        Assign local variable names to state array
         STATE(1) = TCUM
         IARELN = 1
         DO 40 I=1,N
            IARELN = IARELN + 1
            STATE(IARELN) = X(I)
40       CONTINUE

      ELSE IF (ITASK.EQ.2) THEN

*        Rates of change section
*        ======================
*        Assign state array to local variable names
         TCUM = STATE(1)
         IARELN = 1
         DO 50 I=1,N
            IARELN = IARELN + 1
            X(I) = STATE(IARELN)
50       CONTINUE

*        Dynamic calculations

*   sum of populations
         XSUM = ARSUMM ('X',X,1,N,1,N)

* estimate the average daily temperature
         TDMEAN = (TMMN + TMMX) / 2.0
         DO 60 I=1,N
            RX(I) = RGRMAX(I) * LINT2('REDUCE',REDUCE,ILREDU,TDMEAN) *
     $       X(I)
60       CONTINUE

*        Finish conditions
         IF (KEEP.EQ.1) THEN

            DO 70 I=1,N
               IF (X(I).GT.200.0*IX(I)) TERMNL = .TRUE.
70          CONTINUE
         END IF


*        Output
         IF (OUTPUT) THEN
            CALL OUTDAT (2, 0, 'TIME  ', TIME  )
            CALL OUTDAT (2, 0, 'XSUM', XSUM)
            CALL OUTDAT (2, 0, 'TDMEAN', TDMEAN)
            CALL OUTAR2 ('X',X,1,N,1,N)
         END IF

*        Assign calculated rates to rate array
         RATE(1) = TDMEAN
         IARELN = 1
         DO 80 I=1,N
            IARELN = IARELN + 1
            RATE(IARELN) = RX(I)
80       CONTINUE

      ELSE IF (ITASK.EQ.4) THEN

```

```
*        Terminal section
*        ================
*        Assign terminal states and rates to local variable names
         TCUM = STATE(1)
         IARELN = 1
         DO 90 I=1,N
            IARELN = IARELN + 1
            X(I) = STATE(IARELN)
90       CONTINUE
         TDMEAN = RATE(1)
         IARELN = 1
         DO 100 I=1,N
            IARELN = IARELN + 1
            RX(I) = RATE(IARELN)
100      CONTINUE

*        Terminal calculations
*    average temperature in simulation period
         TMEAN = TCUM / (TIME-STTIME)

*        Terminal output
         CALL OUTDAT (2, 0, 'TIME  ', TIME  )
         CALL OUTDAT (2, 0, 'TMEAN', TMEAN)

*        Printplot output
*        none
      END IF

      RETURN
      END
```

## 9.2.4 Adding events to an FST-generated module

State events are certain system states which cause a sudden change of the system. A state event is assumed to be momentarily. The problem of handling such an event in a continuous simulation model is that the system should “exactly” reaches the moment in time at which the state event occurs. Then the appropriate changes are made to the state variables and the simulation continues. Time events are just certain moments in time at which the system status suddenly changes.

In FST, state and time events cannot take place. The drivers EUDRIV and RKDRIV, however, are prepared to handle such events. In some cases it may be worthwhile to change the generated MODEL in order to have events taking place. This is not difficult but requires great care since the Fortran compiler only verifies the Fortran code and not the consistency of the model.

Note that a simple check like TIME.EQ.EVENT_TIME in the user MODEL would not work since the integration procedure calls the user MODEL at a lot of times during a time step. A valid time event can only take place at the beginning of a new time step (when KEEP=1, see Section 9.2.2.2). During the calls of a numerical integration step no events may happen.

A proper implementation of state and time events requires additional communication between the driver and the user MODEL, which is realized by means of the “driver event” common /DRVEVT/. In Section 9.2.4 an example is given, but at first the variables in common /DRVEVT/ are introduced.

#### 9.2.4.1 The event common /DRVEVT/

The declaration of the variables in common /DRVEVT/ are

```
*     declarations of the variables in common /DRVEVT/
      INTEGER          NSEV
      PARAMETER        (NSEV=20)
      LOGICAL          TEVREQ, SEVREQ(NSEV), SECROS(NSEV), STEVNT, TEVENT
      REAL             TEVTIM, SEVFUN(NSEV)
      COMMON /DRVEVT/  TEVREQ, TEVTIM, TEVENT,
     $                 SEVREQ, SEVFUN, STEVNT, SECROS
```

The meaning and function of these variables is

| | |
|---|---|
| TEVREQ | Request time event ; flag which may be set .TRUE. by MODEL in order to generate a time event at time TEVTIM. The flag is initialized by the driver at .FALSE. At the beginning of each time step (just after each KEEP=1 call), the driver checks the value of the request flag. |
| TEVTIM | Time of next event set by MODEL ; initialized by driver at zero. |
| TEVENT | = .TRUE. time event signaled by driver. This means that TIME has reached the value TEVTIM and the system is at the beginning of a new step.<br>= .FALSE. no time event. |
| NSEV | The maximum number of different state event conditions. |
| SEVREQ | Request state event. This is an array of NSEV flags which may be set by MODEL to request a future state event for one or more of the NSEV event functions SEVFUN. The flags are usually set during the INITIAL call but may be changed at the beginning of each new time step (during each call for which KEEP=1). Before initializing MODEL, all NSEV flags are set at .FALSE. by the driver and, by default, no events occur. |
| SEVFUN | Array of NSEV state event functions. Each of these values is a function of one or more variables, calculated in initial or dynamic. Also input variables read from MODEL.DAT may be used. When the flag SEVREQ(J) is set for a value of J, zero- crossings of SEVFUN(J) are detected by the driver. The precise time of the crossing is then found by means of iteration. Then ,at the moment of the state event, the event flag SECROS(J) is set by the driver. The values of SEVFUN have to be calculated in a special call to the user MODEL with ITASK=3. |
| STEVNT | = .TRUE. a state event occurs ; which state event can be seen by checking the SECROS flag array.<br>= .FALSE. no state event. |
| SECROS | array of state event flags ; for each of the NSEV state events a flag can be set by the driver. Event flags are set only during rate call's for which also KEEP=1. |

#### 9.2.4.2 Example of an added state event

A state event is added to the FST model in Listing 9.1 on page 146 which has been translated into the Fortran subroutine MODEL in Listing 9.3 on page 154. The model describes two populations growing exponentially. The state event added is the following:

- If the sum of the two population sizes XSUM reaches 100.0, the slowly growing population X(1) is reduced by 20% and the fast growing population is initialized at its start value 1.0.

This state event is included in the user MODEL of Listing 9.3 by means of the following five steps:

**Step 1**. The above declarations of common /DRVEVT/ are inserted in the user MODEL, preferably at line 52 of Listing 9.3, after the declarations of common /DRVCOM/.

**Step 2**. At the end of the INITIAL section, an event request flag is set by means of

```
30         CONTINUE                  <-existing statement (line 165)

*          request state event 1
           SEVREQ(1) = .TRUE.        <-event function 1 will be monitored by driver
```

This change will instruct the driver to flag a state event as soon as the function SEVFUN(1) crosses zero (see step 5 below). If different events may take place, different flags are set. To each flag corresponds an element of SEVFUN.

**Step 3**. The Rates of change section (ITASK=2) begins with copying the contents of the STATE array into local variable names. After that, the state event is handled by means of

```
50      CONTINUE                            <-existing statement (line 185)

*       state event handling
        IF (STEVNT) THEN                    <-a state event happens ?
           IF (SECROS(1)) THEN              <-handling event 1
*             event 1
              X(1) = 0.8 * X(1)             <-the event takes place here
              X(2) = 1.0
           END IF
        END IF

*       Dynamic calculations                <-existing comment (line 187)
```

Within the IF (STEVNT) THEN - END IF structure different state events may be distinguished by just adding sections for other SECROS flags. Here, only event 1 was requested and no others can take place. The Fortran code added in this step is the actual “event handling section”.

**Step 4**. Since a state event implies that changes in the system status are made, the STATE array has to be updated before leaving the MODEL subroutine. The Fortran code for doing that is copied from the end of the INITIAL section:

```
        END IF                              <-existing statement (line 214)

*       Assign local variable names to state array    <-copy of line 167
        STATE(1) = TCUM                     <-copy of line 168
        IARELN = 1                          <-copy of line 169
        DO 75 I=1,N                         <-copy of line 170 with label number changed
           IARELN = IARELN + 1              <-copy of line 171
           STATE(IARELN) = X(I)             <-copy of line 172
75      CONTINUE                            <-copy of line 173 with label number changed

*       Assign calculated rates to rate array <-existing statement (line 216)
        RATE(1) = TDMEAN                    <-existing statement (line 217)
```

Note that each state array requires its own DO-loop and that the an exact copy has to be made of the similar section in INITIAL. Otherwise the STATE array becomes a mess. Clearly, the copying has only to be done in case of an event. Hence, the entire new section may be placed into an `IF(STEVNT.OR.TEVENT) THEN...ENDIF` control structure. It is not necessary, however, as long as state variables are never changed outside event handling sections.

**Step 5**. The last step is the calculation of the state event functions. This is done by means of a new “ITASK section” in the large control structure around INITIAL, DYNAMIC and TERMINAL:

```
80      CONTINUE                        <-last statement of DYNAMIC (line 222)

     ELSE IF (ITASK.EQ.3) THEN          <-this value of ITASK is used by the drivers
*       calculate state event functions <-for calculating the event functions
        SEVFUN(1) = (100.0-XSUM)/100.0  <-this event function is 0.0 for XSUM=100.0

     ELSE IF (ITASK.EQ.4) THEN          <-start of the TERMINAL section (line 226)
```

Preferably, the state event functions are dimensionless numbers in the range [-1.0,+1.0]. The reason is that their zero-crossing is detected by the driver with a certain tolerance, specified by variable SEVTOL in the file TIMER.DAT. The default value of this control variable is 1.0E-5. The tolerance is an absolute number and proper scaling should be done by means of the event functions SEVFUN.

After completing the five steps, the resulting Fortran program can be compiled and linked with the libraries DRIVERS, TTUTIL and WEATHER. The result of the program can be found as Figure 9.1 on the title page of this chapter. The state events are a form of interaction between the two formerly independent species. The slowly growing one suffers little from each event and is able to overgrow the fast growing population in about 5 years.

#### 9.2.4.3 Time events

The inclusion of a time event proceeds along the first 4 steps of Section 9.2.4.2. Only the variables involved are different. In the INITIAL section, a time event is requested by setting TEVREQ. Also the event time TEVTIM needs to be set. If TEVTIM is reached the driver sets TEVENT and the event is handled in an event handling section similar to the section for a state event. Note that a new event time must be set, otherwise there will never be a time event again! Step 5 of Section 9.2.4.2 is not needed. Leffelaar *et al.* (1993) give an example of a time event involving harvest of a crop.

#### 9.2.4.4 What does the driver do ?

The driver checks the event request flags at the beginning of each time step. Hence, also during the simulation, event request flags may be set or reset. You may for instance use a time event to request a state event, or the other way, at a state event you may request a future time event.

A time event is a relatively simple thing. Approaching TEVTIM, the final integration step is reduced by the driver in order to reach precisely TEVTIM. If a state event is requested the driver keeps track of the corresponding event function. At the beginning of each timer step an ITASK=3 call is done to the MODEL in order to calculate the event functions. Then an integration step is made and the driver verifies if one or more of the event functions crossed zero as a result of the integration. This requires a rate call (ITASK=2) and an event function call (ITASK=3).

If a zero-crossing is detected, the precise time of the state event is iteratively found by the driver through bisection of the time interval. Hence, the integration step taken is rejected, and somewhere between the old and the new TIME, the precise moment of zero-crossing is looked for. By setting TRACE=4, a detailed report of the iteration is written to MODEL.LOG. An exactly zero value is not looked for, however. The deviation from zero which is accepted by the driver is SEVTOL, which can be changed in the file TIMER.DAT.

After finding the moment of the state event, the driver accepts the corresponding STATE as a valid status of the system, it sets the output flag, does a rate call to the model (ITASK=2), then sets also the appropriate state event flag(s) and starts the new time step with the event handling rate call. The reason for the intermediate rate call will be explained below.

In the event handling section (see step 3 above), and in calculating event functions, all variables of the model can be used except the variables calculated in the terminal section. This means that all dynamic variables are at their proper values, although the event handling section is placed at the beginning of the Rates of change section. This freedom is the result of an additional rate call carried out by the driver just before an event takes place or just before event functions are calculated. These intermediate rate calls are done with the same values of TIME and STATE and therefore update all dynamic variables which are retained in MODEL as a result of the SAVE statement in Fortran.

The additional rate call to MODEL, just before a state (or time) event, also has the output flag set (but not yet the event flags). This produces additional output values at the time of the discontinuity, just before the event takes place. Also during the subsequent event call the output flag is set by the driver, which leads to a second set of output values for the same value of TIME, but now <u>after</u> the event has taken place. This is necessary to make good graphs of the a possible discontinuity in values caused by the event.

Note that state events can take place only during an integration step, i.e. during continuous system behaviour. Zero-crossing of an event function as a result of another state event or a time event is not detected by the driver. It is also important to realize that crossing zero implies a non-zero value at the beginning. An event function must be at least SEVTOL away from zero, before it can cross zero (again).

# References

Box, G.E.P. & M.E.Muller, 1958. A note on the generation of random normal deviates. Annals of Mathematical Statistics 29: 610-611.

Bratley, P., B.L. Fox & L.E. Schrage. 1983. A guide to simulation. Springer-Verlag, New York Inc., 397 pp.

Feynman, R.P., R.B. Leighton & M. Sands, 1963. The Feynman lectures on physics. Volume 1. Addison-Wesley Publishing Company, Reading, Massachusetts., 52-12 pp.

Goudriaan, J. & H.H. van Laar, 1994. Modelling Potential Crop Growth Processes. Textbook with Exercises. Current Issues in Production Ecology, Volume 2, Kluwer Academic Publishers, Dordrecht, 238 pp.

IBM, 1975. Continuous System Modelling Program III. General system information manual (GH19-7000) and users manual (SH19-7001-2). IBM Data Processing Division, White Plains, New York.

Kraalingen, D.W.G. van & C. Rappoldt. 1989. Subprograms in simulation models. Simulation Report CABO-TT nr. 18. Centre for Agrobiological Research and Dept. of Theoretical Production Ecology. P.O.Box 14, 6700 AA Wageningen, The Netherlands. 54 pp. (available on request).

Kraalingen, D.W.G. van, W. Stol, P.W.J. Uithol & M. Verbeek, 1991. User Manual of CABO/TPE Weather System. CABO/TPE internal communication. 27 pp. (available on request).

Kraalingen, D.W.G. van, 1995. The FSE system for crop simulation, version 2.1. Quantitative Approaches in Systems Analysis No. 1. DLO Research Institute for Agrobiology and Soil fertility; The C.T.de Wit graduate school for Production Ecology. Wageningen. The Netherlands. 58 pp. (available on request).

Kraalingen, D.W.G. van & C. Rappoldt, 1996. The Fortran utility library TTUTIL 4.1. In prep.

L'Ecuyer, P., 1988. Efficient and portable combined random number generators. Communications ACM 31:742-749,774.

Leffelaar, P.A. (ed.), 1993. On systems analysis and simulation of ecological processes. Current Issues in Production Ecology, Volume 1, Kluwer Academic Publishers, Dordrecht, 294 pp.

Leffelaar, P.A., C. Rappoldt & D.W.G. van Kraalingen, 1993. Simulation using FORTRAN. Chapter 10 in: Leffelaar, P.A. (ed.), 1993. On systems analysis and simulation of ecological processes. Current Issues in Production Ecology, Volume 1, Kluwer Academic Publishers, Dordrecht, 294 pp.

May, R.M., 1972. Limit cycles in predator-prey communities. Science 177: 900-902.

Press, W.H., B.P. Flannery, S.A. Teukolsky & W.T. Vetterling, 1986. Numerical Recipes, the art of scientific computing. Cambridge University Press, 818 pp.

Press, W.H., B.P. Flannery, S.A. Teukolsky & W.T. Vetterling, 1992. Numerical Recipes, the art of scientific computing. second edition. Cambridge University Press.

Rappoldt, C. & D.W.G. van Kraalingen, 1990. FORTRAN utility library TTUTIL. Simulation Report CABO-TT no. 20. Centre for Agrobiological Research and Dept. of Theoretical Production Ecology, Wageningen, The Netherlands, 54 pp. (available on request).

Silebi, C.A. & W.E. Schiesser, 1992. Dynamic Modeling of Transport Process Systems. Academic Press, Inc., New York, 518 pp.

Rosenzweig, M.L.,1972. Stability of enriched aquatic ecosystems. Science 175: 564:565.

Roughgarden, J., 1979. Theory of population genetics and Evolutionary Ecology: An introduction. MacMillan Publishing Co., Inc., New York, 634 pp.

# Appendix A: Glossary

**array variable** The name of a row of numbered elements, which each may contain a value. In FST only one-dimensional arrays variables can be used.

**array element** One of the parts of an array. The array A may have 5 elements, for instance, with values 1.0, 2.0, 4.0, 12.0 and 3.3.

**array family** One or more arrays with a subscript range defined relative to the same array size variable form an array family. Calculations with arrays from within an array family are much easier than with arrays from different families.

**array subscript** An array subscript is the number of the array element. It may be an integer constant, like in A(3) or B(100), or a subscript variable like A(N).

**assignment** A statement of the form "A=expression" is an assignment. The value of the expression is assigned to the variable A. An expression may consist of just a single constant like 2.45 (a real constant), 'New York' (a character constant) or a complicated mathematical expression containing various function calls.

**compiler** A computer program which translates a program written in Fortran, Pascal, C or in any other programming language into machine code, so called object code.

**constant** A value like 2.45 is a real constant, a string like 'New York' is a character constant and an integer like 345 is an integer constant. Constants are the actual values which are contained in variables or arrays of variables. The FST keyword

**CONSTANT** An FST keyword used for the definition of mathematical constants used in the model. For instance CONSTANT PI=3.14159265.

**CSMP** Continuous System Modeling Program. A simulation language developed by IBM (1975). Part of the FST statement structure originates from CSMP.

**cubic spline interpolation** Between each two points a third order polynomial is used for interpolation. The coefficients are calculated in such a way that first and second order derivatives are continuous in the function points.

**declaration** A statement which tells a translator or compiler something about a variable or program name which is used further down in the program.

**driver** A program which translates an abstract command into a lot of instructions for another program or for a piece of hardware. The integration drivers of FST translate the simulation command into a series of call's to the simulation module. Hence, they drive the simulated process forward.

**DYNAMIC** An FST statement at the begin of the dynamic calculations, the calculations which specify the equations of change.

**expansion** See statement expansion.

**expression** The combination of one or more variables with mathematical operations on them.

**FSE** The Fortran Simulation Environment for crop growth simulation.

**FST** The Fortran Simulation Translator, which generates either FSE modules or so called GENERAL modules suitable for an Euler or Runge-Kutta integration driver.

**FUNCTION** An FST statement used for specifying a function by means of a series of (x,y) points.

**INITIAL** An FST statement at the begin of the initial calculations, the calculations which are carried out before the driver starts the actual dynamic calculations.

**initial constant** In FST a variable which is defined by means of an INCON statement.

**initial value** The start value of a state variable (the first argument of an INTGRL function call). It can be an INCON variable or a variable calculated in the INITIAL section.

**integration method** A numerical method for calculating the integral of a function by means of a finite number of function calls.

**interface routine** A program or procedure which forms the connection between two other programs.

**interpolation** The calculation of intermediate function values

**keyword** FST keywords are TIMER, WEATHER, INCON, PARAMETER etc.

**library** In this manual a collection of one or more compiled Fortran subprograms contained in one file, the library file.

**linker** A computer program which combines the machine code from different subprograms with machine code from user libraries and standard machine code libraries in order to form a program which can be executed by the computer. An FST model is first translated into Fortran, then compiled into machine code, then linked with libraries and finally executed by the computer.

**model** A simplified description of the reality. It can be a physically constructed model or a mathematical description. For the numerical evaluation of a mathematical model, a computer program is often necessary. Such computer programs are sometimes called models, although this is confusing. A model can be represented by many different computer programs and all these programs should in fact not be called different models.

**natural cubic splines** A method for interpolating between points of a smooth function. For each two points the coefficients of a third order polynomial are calculated in such a way that the first and second order derivative are also continuous. The word "natural" refers to the assumption that the second order derivative is zero at the first and last point.

**object library** See library.

**parameter** A variable in a model on which the model behaviour depends. The value of this variable is not part of the model.

**rate of change** The change per unit of time of a status variable.

**rerun** Another run of the program, usually for (slightly) different input values.

**section** An FST program consists of different sections: a DECLARATIONS section, a MODEL section, consisting of INITIAL, DYNAMIC and TERMINAL, and a RERUN section. Sometimes one of the three MODEL calculation sections (INITIAL, DYNAMIC or TERMINAL) is meant.

**simulation** Following the changes of a model system in time, often by means of a computer program.

**sorting** FST sorts the calculations in the INITIAL, DYNAMIC and TERMINAL section of the program in order to bring them in a computational order. Writing an FST program one may work "top-down": at first the main equations are written in the form of FST statements, then the variables in them are specified in terms of other variables etc. The actual calculations have to be carried out in the reverse order. FST takes care of that and verifies the completeness of the description.

**statement expansion** A statement containing one or more (equally long) array subscript ranges is expanded in order to perform the calculations element wise.

**subroutine** A subprogram in Fortran. A subprogram is a program unit which can be called by other program units (subroutines). A subroutine call requires a list of input and output arguments, with which the subroutine works. The internal structure of the called subroutine is fully independent of the program from which it is called.

**symbol** In this manual usually a variable name in a computer program. The symbol list of a program is just the list of names which occur in it.

**TERMINAL** The last part of the MODEL section of an FST program is the TERMINAL section. It contains the calculations to be carried out after finishing the simulation.

**translator** In this manual it is the FST program itself, which translates an "FST" model description in into a Fortran program with associated data files.

**TTUTIL** A library of utility subprograms, most of them for input and output. The name TTUTIL originates from the department of "Theoretische Teeltkunde", now called Theoretische Productie Ecologie of the Agricultural University in Wageningen.

**utility** A subroutine or function which takes care of a scientifically trivial part of the calculations, for instance input from file, or linear interpolation between two function points.

**variable** A symbol in a mathematical expression corresponding to a name in a computer program.

**weather system** A subroutine library which may be called from a Fortran program in order to read data from a large set of data files containing weather data for different countries, stations and years. (van Kraalingen *et al.* 1991).

# Appendix B: Reserved variable names

This Appendix lists the names that cannot be freely used as user-defined names. Note that it is not necessary to know all these names. Improper names always lead to an error message of the FST translator. There are a few categories of reserved names:

**Names reserved for special purposes in FST**
These names have a meaning in FST programs. The names in this category are FST control variables, FST driver supplied variables and FST weather and calendar data, FST intrinsic functions or Fortran intrinsic functions. The list is

| | | | | | |
|---|---|---|---|---|---|
| ABS | fortran intrinsic | DELMAX | trans_GEN. | NOTNUL | fst intrinsic |
| ACOS | fortran intrinsic | DELT | timer | PRDEL | timer |
| AFGEN | fst intrinsic | DOY | weather, calendar | RAIN | weather, calendar |
| AINT | fortran intrinsic | DRIVER | trans_GEN. | RDD | weather, calendar |
| ALOG | fortran intrinsic | ELEMNT | fst intrinsic | REAAND | fst intrinsic |
| ALOG10 | fortran intrinsic | ELEV | weather, calendar | REAL | fortran intrinsic |
| AMAX1 | fortran intrinsic | EPS | trans_GEN. | REANOR | fst intrinsic |
| AMIN1 | fortran intrinsic | EXP | fortran intrinsic | RGACTS | driver supplied |
| AMOD | fortran intrinsic | FCNSW | fst intrinsic | RGINIT | timer |
| ANINT | fortran intrinsic | FINTIM | timer | RGNORM | fst intrinsic |
| ARIMPR | fst intrinsic | I | driver supplied | RGSEED | timer |
| ARLENG | fst intrinsic | INSW | fst intrinsic | RGUNIF | fst intrinsic |
| ARMAXI | fst intrinsic | INT | fortran intrinsic | SIN | fortran intrinsic |
| ARMEAN | fst intrinsic | INTGRL | fst intrinsic | SINH | fortran intrinsic |
| ARMINI | fst intrinsic | IOBSD | translation_FSE | SQRT | fortran intrinsic |
| ARSMPS | fst intrinsic | IPFORM | timer | STTIME | timer |
| ARSTDV | fst intrinsic | ISTN | weather control | TAN | fortran intrinsic |
| ARSUMM | fst intrinsic | IYEAR | weather control | TANH | fortran intrinsic |
| ASIN | fortran intrinsic | LAT | weather, calendar | TIME | driver supplied |
| ATAN | fortran intrinsic | LIMIT | fst intrinsic | TMMN | weather, calendar |
| ATAN2 | fortran intrinsic | LOG | fortran intrinsic | TMMX | weather, calendar |
| CNTR | weather control | LOG10 | fortran intrinsic | TRACE | trans_GEN. |
| COPINF | timer | LONG | weather, calendar | VP | weather, calendar |
| COS | fortran intrinsic | MAX | fortran intrinsic | WN | weather, calendar |
| COSH | fortran intrinsic | MIN | fortran intrinsic | WTRDIR | weather control |
| CSPLIN | fst intrinsic | MOD | fortran intrinsic | YEAR | weather, calendar |
| DELDID | driver supplied | NINT | fortran intrinsic | | |

**Names used in the generated Fortran**
These are program names, subroutine names, variable names and common block names used in the generated Fortran code. The use of these names in FST would lead to Fortran errors. The list of names in this category is

| | | | | | |
|---|---|---|---|---|---|
| DRVCOM | IULOG | LINT2 | OUTCOM | RDSINT | STEVNT |
| DRVEVT | IUMOD | MAIN | OUTDAT | RDSREA | TERMNL |
| FILEIN | IUNITD | MODELS | OUTPLT | RGRSET | TEVENT |
| FILEP | IUNITL | NDEC | RATE | SCALE | TEVREQ |
| IARELN | IUNITO | NEQ | RDAREA | SECROS | TEVTIM |
| IDOY | IURES | NSEV | RDFREA | SEVFUN | WSTAT |
| ITASK | IWVAR | NSV | RDINIT | SEVREQ | WTRINT |
| ITOLD | KEEP | OUTAR2 | RDSCHA | STATE | WTRTER |

WUSED

**FST and Fortran keywords**
These are a number of names which, strictly speaking, need not to be forbidden. Their use would be confusing, however since they are also FST and Fortran keywords. An example is the use of a variable name PRINT. In order to preserve clarity, the use of keywords as variable names has also been forbidden in FST.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ACCESS | fortran | ENDJOB | fst | LE | fortran | RETURN | fortran |
| AND | fortran | ENDPRO | fst | LEN | fortran | REWIND | fortran |
| ARRAY | fst | ENTRY | fortran | LGE | fortran | SAVE | fortran |
| BLOCK | fortran | EQ | fortran | LGT | fortran | SORT | fst |
| BYTE | fortran | EQV | fortran | LLE | fortran | STATUS | fortran |
| CALL | fortran | ERR | fortran | LLT | fortran | STOP | fst |
| CHAR | fortran | EXIST | fortran | LT | fortran | TABLE | fst |
| CLOSE | fortran | FILE | fortran | MACRO | fst | THEN | fortran |
| COMMON | fortran | FINISH | fst | MODEL | fst | TIMER | fst |
| DATA | fortran | FIXED | fst | NE | fortran | TITLE | fst |
| DECODE | fortran | FMT | fortran | NEQV | fortran | TYPE | fortran |
| DELETE | fortran | FORM | fortran | NOSORT | fst | UNIT | fortran |
| DIRECT | fortran | FORMAT | fortran | NOT | fortran | UNTIL | fortran |
| DO | fortran | GE | fortran | OPEN | fortran | WHILE | fortran |
| DOUBLE | fortran | GOTO | fortran | OR | fortran | WRITE | fortran |
| ELSE | fortran | GT | fortran | OUTPUT | fst | XOR | fortran |
| ELSEIF | fortran | ICHAR | fortran | PAGE | fst | FALSE | fortran |
| ENCODE | fortran | IF | fortran | PARAM | fst | TRUE | fortran |
| END | fst | INCON | fst | PRINT | fst | | |
| ENDDO | fortran | INDEX | fortran | RAN | fortran | | |
| ENDIF | fortran | IOSTAT | fortran | READ | fortran | | |

**Symbols in linked libraries**
For user-defined subroutines there is an additional restriction. The name of a user-defined subroutine (declared with a DEFINE_CALL statement) cannot be equal to the name of a subroutine, function or common block in one of the linked object libraries DRIVERS, TTUTIL or WEATHER. The list of these names is build into the FST translator and is updated from time to time. Hence, the following list may be incomplete

| | | | | | |
|---|---|---|---|---|---|
| ADDINT | DELFIL | FLNAME | LOWERC | RDDECD | RDFTIM |
| ADDREA | DTARDP | FOPENG | MOFILP | RDDECI | RDINDT |
| ADDREF | DTDPAR | FOPENS | OUTARR | RDDECL | RDINDX |
| ADDSTF | DTDPST | FOPERR | OUTSEL | RDDECT | RDINLV |
| ADDSTR | DTLEAP | FSE | PLTFUN | RDDTMP | RDINNE |
| AFINVS | DTNOW | FSECM1 | PLTHIS | RDERR | RDINQR |
| AMBUSY | DTSYS | GETCH | POS | RDFCHA | RDLEX |
| ARBCHK | ENTCHA | GETREC | RDACHA | RDFDOR | RDMCHA |
| CHKTSK | ENTDCH | GETUN | RDADOR | RDFDOU | RDMDEF |
| CLS | ENTDIN | GETUN2 | RDADOU | RDFIL1 | RDMDOU |
| COPFIL | ENTDRE | IFINDC | RDAINR | RDFIL2 | RDMINT |
| COPFL2 | ENTINT | IFINDI | RDAINT | RDFINR | RDMLOG |
| DECCHK | ENTREA | ILEN | RDALOG | RDFINT | RDMREA |
| DECINT | ERROR | ISTART | RDARER | RDFLOG | RDMTIM |
| DECREA | EUDRIV | IUNIFL | RDATIM | RDFRER | RDPARS |
| DECREC | EXTENS | LINT | RDDATA | RDFROM | RDREC1 |

RDREC2
RDSCTB
RDSDOR
RDSDOU
RDSETS
RDSINR
RDSLOG
RDSRER
RDSTA
RDSTIM
RDTBL1
RDTBL2
RDTMP1
RDTMP2
RDTOK1
RDTOK2
REMOVE
RERUNS
RG
RK4A
RKDRIV
RKQCA
SFINDG
SORTCH
SPLINE
SPLINT
STINFO
STRIP
SWPI4
TIMER2
TTUDD1
TTUDD2
TTURR1
TTURR2
TTUVER
UNIFL
UPPERC
USEDUN
WEATHR
WORDS
WRACHA
WRAINT
WRALOG
WRAREA
WRINIT
WRSCHA
WRSINT
WRSLOG
WRSREA
WSATTR
WSCFIL
WSCONI
WSDAOP
WSFLGS
WSFUN
WSGREC
WSILEN
WSITOA
WSMESS
WSNBUF
WSNFIL
WSOPEN
WSRDDA
WSSTAR

# Appendix C: Converting CSMP into FST

For users familiar with CSMP we have listed a few differences from a practical point of view. Simple programs are easily converted to FST. Programs involving PROCEDURE's or array variables are more difficult to convert but serious problems did not show up so far.

- The **METHOD** statement of CSMP is replaced by a TRANSLATION_FSE or a TRANSLATION_GENERAL statement (See Section 3.2.4.1 and Section 3.2.4.2)
- CSMP allows constants or expressions as arguments of the **INTGRL** function. In FST the two arguments have to be variables or array variables. This problem can be solved with help of a few additional variables for the initial value and/or the rate of change.
- The **FIXED** statement of CSMP for declaring integer variables does not exist in FST. The use of arbitrary integer variables is impossible in FST. In FST, a loop counter I can be used in array calculations (see Section 4.3.2) and array size variables are also integer by default (see Section 4.3.1, Section 8.3.1.1 and Section 8.5.2.3). They do not have to be declared separately.
- The **TABLE** statement of CSMP for filling an array with data can be easily replaced by an FST PARAMETER or statement.
- The **STORAGE** statement of CSMP for declaring an array is replaced by the ARRAY statement of FST. The upper bound of an FST array is declared relative to an array size variable. This means that actual array sizes are set at precisely one place in the FST program, by means of an ARRAY_SIZE statement. Errors with array calculations are therefore less likely in FST than in CSMP.
- The **NOSORT** sections of CSMP do not exist in FST. As far as NOSORT sections are used for array calculations (with Fortran DO-loops), they can be easily replaced by FST statements. In FST, array variables can be used in expressions as if they were scalar variables. The translator then creates the appropriate loops in the generated Fortran (see Section 4.3). If the CSMP program contains complicated control structures (with nested IF-THEN-ELSE blocks, for instance), it will usually be necessary to write a Fortran subroutine called by FST.
- The **PROCEDURE's** in a CSMP program have to be converted into Fortran subroutines with an I/O structure declared in a DEFINE_CALL statement.
- The use of **MACRO**'s in FST is impossible. The macro has to be expanded for the different cases in which it is used, or, if it is used many times, array variables may be used.

# Appendix D: Bug report form

| | |
|---|---|
| Name | : |
| Software product | : |
| Version | : |
| Date | :____________________ |
| E-mail | :____________________ |

Do you consider the bug (choose one): **minor / annoyance / serious / catastrophic**

Is the bug reproducible ? Y / N

Description of the problem:

Sequence of events that will recreate the bug:

Please send with your bug report form:
1) printscreen where bug is visible,
2) a floppy disk with all the relevant files (including source code).

Return this bug report form to one of the suppliers.

# Appendix E: FST Capacity settings

The capacity settings for the FST translator are given in the table below. We had to find a reasonable compromise between the maximum capacity and the amount of RAM memory the translator uses.

Table E.1. Capacity settings of the FST translator

| Capacity | Description |
|---|---|
| 3000 | Maximum number of cross references |
| 10 | Maximum number of finish conditions |
| 22 | Maximum number of called subroutines |
| 218 | Maximum total number of argument descriptions in DEFINE_CALL statements |
| 5000 | Maximum statement length in characters (source or generated) |
| 60 | Maximum number of variables to be sent to OUTDAT |
| 70 | Maximum number of variables in PRINT statements |
| 30 | Maximum number of variables in OUTPUT statements |
| 200 | Maximum number of redefinitions in the rerun sections |
| 1000 | Maximum number of calculation statements (except INTGRL statements) |
| 100 | Maximum number of state variables |
| 100 | Maximum number of lines in statement (source or generated) |
| 1000 | Maximum number of symbols |
| 10 | Maximum number of titles |
| 150 | Maximum number of substatements in program |
| 100 | Maximum number of (X,Y) pairs for CSPLIN interpolation |
| 500 | Maximum number of state variables+state array elements |
| 50 | Maximum number of observation days |
| 100 | Maximum number of interpolation functions |
| 12 | Maximum nesting depth of expressions |
| 500 | Maximum number of sorted statements per section |
| 200 | Maximum number of array declarations |
| 500 | Maximum number of array ranges |
| 500 | Maximum number of values in an assignment or FUNCTION statement |
| 1000 | Maximum number of statements |
| 1000 | Maximum number of value assignments |

Note that some of the capacity settings refer to the DRIVERS library rather than to the FST translator itself. These are the number of state variables / state array elements, which is limited by the EUDRIV/RKDRIV integration capacity, the number of observation days IOBSD in FSE mode and the number of data pairs allowed in cubic spline interpolation of a FUNCTION with CSPLIN.

# Index