
INSEL
Tutorial

Simulation of Renewable Energy Systems

INSEL 8 :: Tutorial

Jürgen Schumacher

INSEL 8 :: Tutorial



© 1986-2019 Jürgen Schumacher. All rights reserved.

Version: INSEL 8.3.0 June 2021

Please visit us at www.insel.eu

Bison is free software and is available under the GNU General Public License.
Eclipse is a trademark of the Eclipse Foundation, Inc. and copyright protected by Eclipse contributors and others.
GCC is copyright protected by Free Software Foundation, Inc.
Flexx is copyright protected by The Regents of the University of California and The Flex Project.
gfortran is copyright protected by Free Software Foundation, Inc.
gnuplot Copyright 1986 – 1993, 1998, 2004 Thomas Williams, Colin Kelley
HP VEE is a registered trademark of Hewlett-Packard Co.
IISiBat is copyright protected (Centre Scientifique et Technique du Bâtiment CSTB).
INSEL was a registered trademark of doppelintegral GmbH, Stuttgart, Germany (liquidated in December 2019).
Current trademark/copyright holder to be confirmed.
Java is copyright protected by Sun Microsystems.
LabVIEW is a registered trademark of National Instruments Corporation.
Linux is a registered trademark of Linus Torvalds.
Mac OS X is a registered trademark by Apple, Inc. in the United States and other countries.
MATLAB is a registered trademark of The MathWorks Inc.
Microsoft Windows and Visual Studio is a registered trademark of Microsoft Corporation in the United States and other countries.
MiKTex-pdfTeX is copyright protected by D. E. Knuth and Han The Thanh
Ruby is copyright protected free software by Yukihiro Matsumoto
Simulink is a registered trademark of The MathWorks Inc.
Subversion is an open source version control system.
TeX is a trademark of the American Mathematical Society.
TRNSYS is copyright protected (S.A. Klein, F.L Alvarado).
VSEit is copyright protected by Kai Brassel.
Window Builder is provided under the terms and conditions of the Eclipse Foundation Software User Agreement.

All examples presented in this Tutorial can be found in the `examples\tutorial` directory of an INSEL installation.

Preface

This Tutorial is an attempt to enthuse you about a fascinating topic: *Computer Simulation of Renewable Energy Systems*. To be more precise, computer simulation of renewable energy systems on the basis of a graphical programming language.

In contrast to *algorithmic* programming languages like C/C++, for example, *graphical programming languages* are very subtle to use. Simulation models can be created by mouse operations rather than having to implement complex algorithms in a text-based programming environment.

The idea of graphical programming is old, it goes back to the year 1955 when Selfridge published a paper at the *Western Joint Computer Conference*. However, *digital* computers were terribly slow at that time. Today, in times of Digitalization, computers are terribly fast.

The idea of simulating renewable energy systems is old, too. Duffie and Beckman published their first book *Solar Energineering of Thermal Processes* in the year 1974 already and together with Klein they released the first graphical programming language for solar energy systems, named TRNSYS.

During the 1980's renewable energy simulation experienced a boom in Europe, not only because of the Tschernobyl disaster. A graphical programming language that was invented in the year 1986 is the simulation environment INSEL, the software this Tutorial is about – we still keep on celebrating INSEL's 25th birthday.

Today, after the Fukushima Daiichi nuclear disaster, renewable energy systems are more vital than ever. Not only the German government has decided to abandon nuclear power and to support renewable energy in the best possible way. Actually, 2011 renewable energy production has overtaken power from nuclear plants in Germany.

We are convinced that simulation can contribute to this renewable energy future – not only with a single tool like INSEL but on the basis of a combination using the advantages of different approaches and strengths of programs like TRNSYS, MATLAB & Simulink, and LabVIEW, to mention a few. INSEL supports them all, i. e., all INSEL models can be used in other programming environments.

Milestone

The presentation of this Tutorial can really be called a milestone in the history of INSEL. It is the first time that a complete documentation is available which covers all aspects of INSEL programming. Why's that?

INSEL is the result of a German research project. Funded by the German Research Ministry and afterwards by the German Volkswagen Stiftung – without a plan for further development and marketing, however. In consequence, the user circle of the software has been restricted to university people basically.

In the early years of the 21st century the company doppelintegral has been founded with the aim to turn INSEL into a product and make it available to a broader public. One of

the most enjoyable results of this effort is that this Tutorial is now almost complete.

Thank you's

A sheer endless number of people have contributed to the fact that INSEL exists, most of them are mentioned by name in the INSEL 8 Block Reference Manual.

Without the engagement of my doctor father Prof. Dr. Joachim Luther INSEL would definitively not exist. Jochen, you are the first person to thank. Without you, my whole life would have taken a different course.

Dr. Hans Karl combines the forenames of my father Hans Karl Schumacher who died far too early. I assume that it was my destiny that you showed me the path to the beauty of graphical programming languages.

Prof. Dr. Ursula Eicker, my wife and inspiration—without you INSEL would have departed like a grain of sand in the desert.

Another "without" goes to Kai Brassel. Without your work on the Java-based VSEit framework INSEL would not have survived the 16 bit INSEL 7 world.

The University of Applied Sciences Stuttgart and its Research Center zafh.net kept INSEL alive from 2002 to 2019.

Now INSEL is going to be further developed into an Urban Modeling Patform named insel4D at Concordia University, Montréal, under the new Canada Excellence Research Chair Next Generation Cities. If you wish to keep track of that development, visit www.insel4D.com from time to time.

A big hug goes to my friend Mike Barker for correcting my broken English in the final version—all temporary shortages are up to myself.

Jürgen Schumacher

Oldenburg/Stuttgart/Montréal 2019

Contents

Preface i

Introduction vii

PART I :: Fundamentals 1

1 Getting started with INSEL 8 3
1.1 Installation . . . 3
1.1.1 Windows . . . 3
1.1.2 macOS . . . 4
1.2 Starting and ending INSEL 8 . . . 4
1.3 Running a first example . . . 6
1.4 INSEL blocks in VSEit . . . 8
1.4.1 The Palette . . . 10
1.4.2 Block entities . . . 11
1.4.3 Entity editors . . . 13
1.4.4 Errors in networks . . . 15
1.5 Macros . . . 16

2 INSEL programming concepts 18
2.1 INSEL block groups . . . 18
2.2 Basic photovoltaics . . . 19
2.3 The INSEL concept of time . . . 23
2.4 Nested Timer blocks . . . 26
2.5 The Timer blocks CLOCK and FDIST . . . 30
2.6 Solar radiation . . . 32

3 Reading and writing data files 42
3.1 Reading data . . . 44
3.1.1 Fortran format conventions . . . 49
3.1.2 The READN block . . . 54
3.1.3 The READD block . . . 56

3.1.4 File name qualifiers . . . 58
3.2 Writing data to files . . . 61
3.2.1 Monitoring and simulation . . . 63
3.3 Plotting data . . . 64

4 If blocks 68
4.1 At end If blocks . . . 68
4.2 If blocks with a parameter . . . 75
4.3 Conditional If blocks . . . 78
4.4 General if conditions . . . 81
4.4.1 Load profiles . . . 83
4.5 Calculation list . . . 87

5 Delay and Loop blocks 90
5.1 Handling control cycles . . . 91
5.1.1 The DELAY block . . . 91
5.1.2 PID controller . . . 94
5.2 Solving differential equations . . . 95
5.2.1 The Jentsch rocket . . . 96
5.2.2 Solar collector equation . . . 97
5.3 Loop block concept . . . 97

PART II :: Applications and exercises 99

6 Solar meteorology 101
6.1 Global radiation . . . 101
6.2 Radiation time series generation . . . 107
6.3 Diffuse radiation . . . 112
6.4 Radiation on tilted surfaces . . . 117
6.5 Ambient temperature time series generation . . . 120

7 Photovoltaics 124
7.1 Grid-connected PV generators . . . 124
7.2 Optimum tilt angle . . . 129
7.3 Parameter identification methods for PV modules . . . 134
7.4 Module mismatch and shading problems . . . 134
7.5 Thin-film modules . . . 134
7.6 Stand-alone PV systems . . . 134
7.6.1 Batteries in INSEL . . . 135
7.6.2 Implementation of load profiles . . . 149
7.6.3 System sizing . . . 151
7.6.4 System studies . . . 160
7.6.5 Parameter variations . . . 163

7.7 The hybrid system Energielabor 168

8 Solar heating and cooling 169
8.1 Solar collectors 169
8.2 Storage tanks 172
8.3 Heat exchangers 174

9 INSEL GUI's with VSEit 181

10 INSEL in MATLAB 182
10.1 MATLAB 182
10.2 Running INSEL models from MATLAB 185

PART III :: Advanced concepts 186

11 INSEL without GUI 188
11.1 Running `.insel` files 188
11.2 `.include/.insel` applications 193
11.3 Optimization with GenOpt 196
11.4 Direct calls of INSEL blocks 197

12 Programming INSEL blocks 199
12.1 A Fortran crash course 201
12.1.1 The principle form of a Fortran program 202
12.1.2 Fortran data types 205
12.1.3 If-Then-Else structures 211
12.1.4 Structuring program projects 219
12.1.5 Guidelines for writing INSEL Fortran code 230
12.2 Programming INSEL blocks (cont.) 235
12.2.1 Block wizard 235
12.2.2 Templates 239
12.2.3 Call modes 245
12.2.4 Properties 248
12.2.5 Documentation 250
12.3 Text output from INSEL 254
12.3.1 Message files 255
12.3.2 The INSEL message system 256
12.4 INSEL block source code examples 260
12.4.1 The CONST block 263
12.4.2 The SUM, MUL, MAX, and MIN blocks 264
12.4.3 The DIV block 266
12.4.4 The ROOT, GAIN, ATT, and OFFSET blocks 269
12.4.5 The T-block DO 270

12.4.6 The I-block IF . . . 273
12.4.7 The D-block DELAY . . . 274
12.4.8 The L-block NULL . . . 274
12.5 Interfacing INSEL with Python . . . 279

13 Programming INSEL extensions in Eclipse 281
13.1 Java Development Kit . . . 282
13.2 Eclipse . . . 282
13.2.1 A first Java project . . . 283
13.2.2 Installing Eclipse plugins . . . 289
13.3 C/C++ Development Tools (CDT) . . . 290
13.4 Fortran Development Tools (Photran) . . . 295
13.5 WindowBuilder . . . 299
13.6 Eclipse as INSEL block IDE . . . 303
13.6.1 A makefile project for user block development . . . 303
13.6.2 Debugging user blocks in Eclipse . . . 303

PART IV :: Workshops 312

14 PV Heat Pump Storage System 314

15 TRNSYS Restaurant 324

Résumé 330

A Appendix 332
A.1 Directory structure, dependencies, and paths . . . 332
A.1.1 File Handling . . . 335

Introduction

INSEL is an acronym for INtegrated Simulation Environment Language. INSEL is not a simulation program but provides an integrated environment and a graphical programming language for the creation of simulation applications.

The basic idea of INSEL is to connect blocks to block diagrams that express a solution for a certain simulation task.

INSEL was originally developed for modeling of renewable energy systems, the first versions being written at the former Renewable Energy Group at the Faculty of Physics of Oldenburg University, Germany.

What makes INSEL special?

Program flow

The classical approach to computer programming is based on algorithmic programming languages like Fortran or C, for example. From a set of elementary statements a program is written with an ASCII text editor, compiled and finally linked together to build an executable. Program flow is the main aspect which dominates the whole development stage.

Data flow

Graphical programming languages like INSEL use a totally different approach where data flow plays the key role. Instead of statements these languages provide graphical symbols which can be interconnected by mouse operations to build up larger structures. The graphical symbols can represent mathematical functions, real components like solar thermal collectors, photovoltaic modules, wind turbines and batteries, for example, or even complete technical systems of any kind. The graphical elements of INSEL are blocks.

The following list gives a first impression of the modular organization and the currently main application fields of INSEL.

inselEngine

:: The core component of INSEL is the inselEngine which is a full compiler that can interpret and execute applications written in the INSEL language or created from the graphical pre-processor VSEit. INSEL 8 provides an import function for INSEL 7 models written in HP VEE.

Libraries

:: Fundamental blocks, basic operations and mathematical functions of the environment are provided in a dynamic library called inselFB. It contains tools like blocks for date and time handling, access to arbitrary files, blocks for performing mathematical calculations and statistics, blocks for data fitting, plotting routines, and so on.

:: Energy meteorology and data handling is available as library inselEM. This library contains algorithms, like the calculation of the position of the Sun, spectral distribution of sunlight, radiation outside atmosphere. A large data base provides monthly mean values of irradiance, temperature and other meteorological parameters. Generation of hourly radiation, temperature, wind speed, and

humidity data from monthly means is possible. Further, diffuse radiation models, conversion of horizontal data to tilted are included.

- Solar electricity components like photovoltaic modules, maximum-power- point tracker, wind turbines, batteries, battery charge regulators, hydrogen storage components, water pumps, inverters, motors, and generators, are available in the dynamic library inselSE.
- Solar thermal components such as thermal flat-plate and vacuum water and air collectors, storage tanks. A full set of models for the simulation of thermal solar cooling plants, like desiccant and evaporative cooling systems, absorption cycles is implemented in the library inselST.
- Solar thermal power plants for solar electricity generation is under development as library inselPP which will contain models for parabolic trough, solar tower, dish-sterling and other solar power plant technologies.
- A building simulation library called inselBS is under development and will contain models for walls, windows, convective and radiative heat exchange between surfaces, thermo-active components like cooling and heating floors and ceilings.
- A data processing library inselDP for Internet communication via different protocols is currently under development.
- A programmable environment with a user-written library inselUB in which practically all fields of engineering applications can be built in a very structured way. All standard programming languages like Fortran, C/C++, which can be compiled into object code, are supported.

Data bases

- Data bases for simulation parameters of components that are available on the market are included INSEL 8.

Structure of the Tutorial

This Tutorial is an introduction to programming with INSEL. No previous knowledge of INSEL or of any other graphical programming language is required.

Prerequisite

All examples and exercises of this Tutorial can be solved and tested in practice by using the 30-days trial period of the INSEL Trialware, which can be downloaded from www.insel.eu. When the 30 days are expired a license must be available, otherwise the software can no longer be used.

The Tutorial is organized in three parts.

Part I teaches the fundamentals of INSEL. Part II is task-oriented, so that you can go directly to the section that suits your interest most. Part III covers advanced programming techniques like implementation of user-written INSEL blocks and

functions into dynamic libraries. The construction techniques for the creation of user interfaces is also presented in the third part.

The goal of this Tutorial is to enable you – the reader – to program applications with INSEL as soon as possible. You can learn how to use the modular simulation tool INSEL and apply it to renewable energy systems. This knowledge can then be applied to all other fields of numerical engineering.

Time required to study the Tutorial

First programming steps are achieved very fast. To work through the first two parts of the Tutorial intensively will take about a week to complete. We have used guided examples for the most part. The exercises in Part II are a challenge to solve problems on your own. Solutions are provided with explanations. Part III is optional for advanced INSEL programmers or MATLAB users.

Although INSEL supports the operating systems Windows, Mac OS X and Linux, for example, the description given here assumes that you are working under Microsoft Windows.

Whereas INSEL is the calculation engine to solve mathematical models, the commercial visualisation tool VSEit (Versatile Simulation Environment for the Internet) is used extensively to graphically construct INSEL models. Since the VSEit framework is completely written in Java it can be used with Windows, Mac OS X and Linux, too.

The Tutorial is organized in Modules, which treat different subject areas. Most Modules also contain concrete pre-programmed examples which should be analyzed and run. The user can then learn to reconstruct these examples following the given examples. In a second stage, exercises are given, where the reader should find own solutions in model construction. However, the solutions are also presented in this Tutorial.

Part I

Module 1 shows how to basically install and handle the simulation environment. The interaction between the graphical tool VSEit and the INSEL calculation engine is explained and demonstrated in simple examples. The basic handling of the graphical interface is explained in detail.

Module 2 starts with a description of the programming concepts of the graphical simulation tool INSEL. It explains the simpler block concepts such as Constant and Standard blocks and introduces the often needed Timer block concept. The Module then covers a range of examples using the block concepts explained before. Among the examples the performance of photovoltaic modules will be calculated for grid-connected and stand-alone systems.

Module 3 treats data file handling in INSEL. Reading and writing data from files and the corresponding formatting statements are explained.

Module 4 introduces a further block concept, the If blocks, and shows examples where such blocks are useful.

Module 5 starts with Delay blocks, which are necessary to solve “algebraic loops.” As a last and most complex block concept, Loop blocks are introduced.

At the end of Module 5 you will be familiar with all the block concepts that INSEL offers.

Part II

Part II covers five Modules with extensive applications one Module about graphical user interfaces in INSEL and one about MATLAB.

Module 6 is a course about some aspects of meteorological data processing.

Module 7 touches the topic of photovoltaic system simulation.

Module 8 shows simulation examples from the field of solar heating and cooling.

Module 9 handles the creation of interactive VSEit GUIs.

Module 10 describes how INSEL models can be run from MATLAB and completes the second part.

Part III

Part III is meant as a supplement for the advanced INSEL programmer.

Module 11 introduces INSEL programming using a text editor. The graphical representation in VSEit is then no longer necessary. Very few INSEL language statements need to be learnt to directly program in a text editor and create `.include/.insel` applications. The Module also shows how general-purpose programming language like C/C++ or Ruby, for example, can be used to directly communicate with INSEL blocks – either directly or via the wrapper class `CinselBlock`. The Module ends with an application which is completely independent of the inselEngine.

Module 12 describes how users can write and implement their own INSEL blocks in Fortran, C, C++ or any other compiler language which can generate dynamic libraries as output. INSEL provides a block wizard for the creation of new blocks and tools which are required to fully integrate these blocks into the VSEit environment.

Module 13 introduces the integrated development environment Eclipse for the purpose of INSEL block management and debugging. In addition to the Java, Fortran, and C/C++ languages, the Module introduces the Ruby script language, LaTeX documentation of INSEL blocks, a plug-in named Window Builder for the creation of Java-based graphical user interfaces, and finally Subversion, a plug-in for version control of software development.

PART I :: Fundamentals

1 :: Getting started with INSEL 8

1.1 Installation

The available INSEL 8 installers for supported operating system are

- `win32\setup_insel_8.3.0_32.exe` (Windows 10)
- `win64\setup_insel_8.3.0_64.exe` (Windows 10)
- `macOSX/insel_8.3.0_macOS.pkg` (macOS Mojave)
- `linux32/insel_8.3.0_32.deb` (Debian Linux)
- `linux64/insel_8.3.0_64.deb` (Debian Linux)
- `linux32/insel_8.3.0_32.rpm` (Red Hat Linux)
- `linux64/insel_8.3.0_64.rpm` (Red Hat Linux)

Administrator rights required

INSEL 8.3 will be the last version to support 32-bit operating systems. The setup program requires administrator rights in order to install the software.

1.1.1 Windows

All files, executables and dynamic link libraries which are required by INSEL 8 are copied to the INSEL installation directory, which is typically

`C:\Program Files\INSEL 8.3` or `C:\Program Files (x86)\INSEL 8.3`

when the 32-bit version is installed under 64-bit Windows. The directory includes a copy of the Java Development Kit JDK Version 8 since the VSEit user-interface of INSEL is based on Java 8.

The `resources` directory of the INSEL installation is added to the Windows environment variable `%PATH%`.

Documentation

The INSEL documentation is written in LaTeX and is supplied in .pdf format. The Windows version uses Adobe's Acrobat Reader. In case, no Acrobat Reader can be found a fallback to the reader provided in the `resources` directory is used. In most Debian distributions Document Viewer is installed by default.

1.1.2 macOS

The `.pkg` package can be installed by a double click on its icon. The directory structure is similar to the Windows version. The application is typically installed as `/Applications/INSEL.app` macOS application bundle.

Symbolic links

Symbolic links are created to all dynamic libraries and three executables which INSEL uses. A complete list can be found in the shell script `INSEL.app/Contents/_createRequiredSymbolicLinks.sh`.

GNUPLOT GOES HERE: SEE PROGRAMMERS GUIDE (TEMPORARILY)

1.2 Starting and ending INSEL 8



In the Windows version the installation program has created an icon on your desktop. INSEL 8 can simply be started with a double click on the icon.

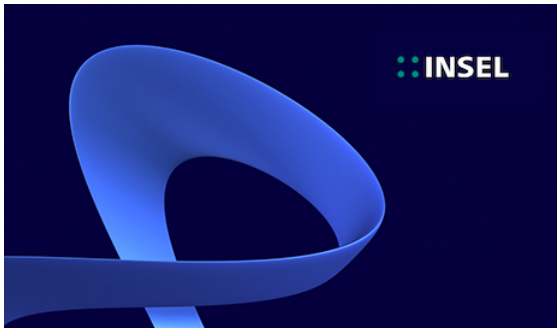
Another possibility is to start INSEL 8 via Windows' Start button and the link to the executable in the Programs list (group `insel 8`).

Yet another possibility is to browse to the installation directory (typically `C:\Program Files\INSEL 8.3`) with the Windows Explorer and double-click on the executable `insel_8.exe`.

There is another executable named `insel.exe` in the `resources` subdirectory. This executable is meant to run INSEL 8 from a console prompt or in batch mode.

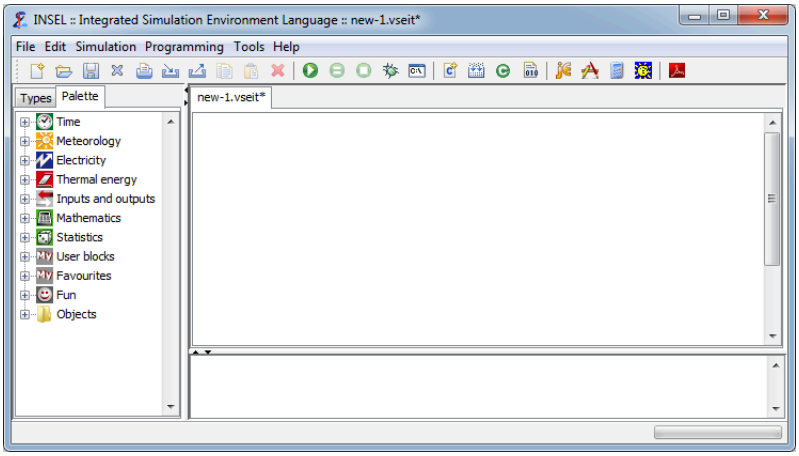
Splash screen

During start INSEL 8 will display a splash screen.



INSEL window

After a moment the INSEL window appears.



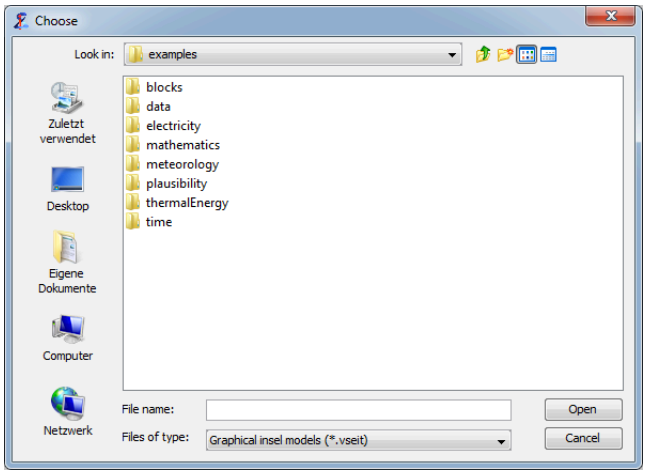
- The *Title* bar contains the program name, the name of the currently open document (like `new-1.vseit` for the first open, empty model), and the standard Windows buttons to minimize, maximize and close the window.
- The *Menu* bar with the *File*, *Edit*, *Simulation*, *Programming*, *Tools*, and *Help* menu can be used to access items and features.
- The *Tool* bar and its icons provide buttons for the most frequently used functions.
- On the left-hand side the *Palette* is displayed. Here all INSEL blocks can be found. They are organized by *categories*, like *Time*, *Meteorology*, *Electricity* etc.
- Next to the palette is the *Types* tab. Similar to the palette the *Types* pane lists INSEL blocks, but only those used in the current model.
- The white space on the right side is the *Work area* which can be used to create INSEL block diagrams.
- The *Output window* below the work area is used by INSEL for text output.
- The *Status* bar is displayed at the bottom of the INSEL window. It is used for temporary text messages and includes the *Progress* bar which shows the progress of a running INSEL model.

Ending INSEL 8

The INSEL main window can be moved, resized and closed in the usual fashion. The *File – Exit...* menu item and the *Close* button in the window's title bar are equivalent options to end the program.

1.3 Running a first example

A good starting point to become familiar with INSEL are the examples which can be found in the `examples` directory. The easiest way to access the examples is via the *File – Open example...* menu item which will open a file chooser dialog similar to the one in the next figure.

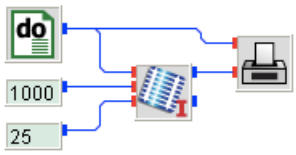


`blocks` directory

In the `blocks` directory one basic example for each INSEL block is available. It is certainly a good idea to browse through the subdirectories of the `blocks` directory and get an overview which blocks are available in INSEL 8 and what their functions are.

`pvi.vseit`

As a first example we choose `pvi.vseit` from the `electricity` directory. The block diagram looks like this:

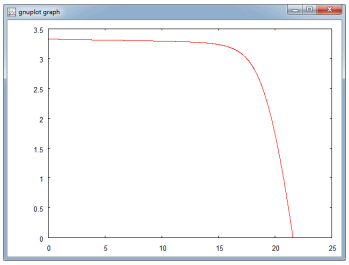


The application will plot the I-V curve of a PV module under standard test conditions. Before going into details let us execute the model.



In the tool bar there are five buttons dealing with the execution of INSEL models.

Since `pvi.vseit` is the current not-yet-running INSEL model the *Run* button is highlighted. Your first INSEL simulation run is now only one mouse click away. The result should be a Gnuplot graph.



Should it disturb you that Gnuplot displays mouse coordinates in the lower left corner by default, press the *m* key and they disappear. Pressing the *m* key again brings them back.

Debugging

Another option to execute INSEL models is via the *Debug* button. In this case the block which has the focus is highlighted by a green frame. The current values of all inputs and outputs are shown next to the respective ports. In addition, INSEL will display a list in the output window which block is called at the moment.

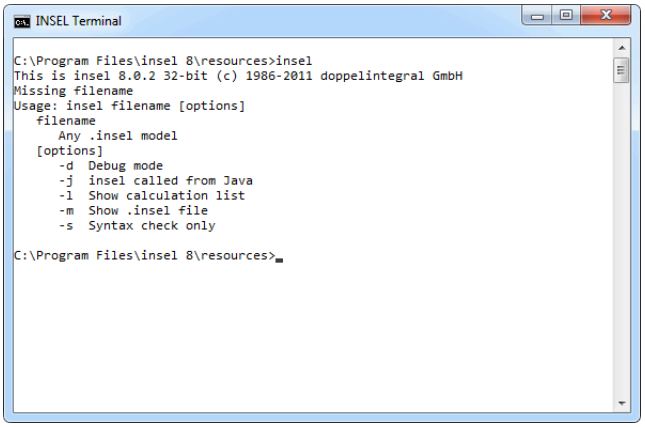
When you try out the debug mode, you can observe

- How much slower the execution of the model is.
- How the progress bar indicates the status of the running INSEL model.
- That the *Pause* button and the *Stop* button are highlighted during model execution.

Obviously, a click on the *Pause* button pauses model execution and a econd click on the *Pause* button continues model execution. The *Stop* button terminates model execution. However, INSEL prompts you to confirm this action.

Run INSEL from DOS prompt

It is also possible to execute INSEL models in batch mode or in a terminal or DOS box window. Such a terminal or DOS box window can be opened via the rightmost button C:\, for example. At this point it is sufficient to have a look at the usage of the `insel` command.



More information on this topic can be found in section *INSEL without GUI* of the Tutorial's Module 13.

1.4 INSEL blocks in VSEit

As mentioned before, the graphical user interface of INSEL 8 – the INSEL window – is based on the VSEit framework. Before we take a closer look at the usage of INSEL blocks in VSEit, a basic understanding of the term INSEL block is required.

Question

So let us ask the question "What is an INSEL block?" and try to answer it. Well, in principal, an INSEL block is nothing but a representation of a mathematical function, let's say f.

Inputs and outputs

Most functions depend on independent variables $x = (x_1, x_2, \ldots)$. In INSEL, these independent variables are called inputs. When the function f is applied to x, the resulting dependent variables $y = (y_1, y_2, \ldots)$ are called outputs. Please notice, that the x_i and y_i are scalars (4-byte real numbers in INSEL).

Hence, we may write $y = f(x)$, and interpret this as a representation of an INSEL block named f with inputs x and outputs y.

Parameters

In addition, the function – or INSEL block – f can have a set of constant parameters $p = p_1, p_2, \ldots)$ which influence the current value of the output y. Hence, we can write $y = f(x, p)$. Parameters in INSEL can be real numbers and strings.

History

Finally, blocks may have a history, which can make their results time-dependent, i. e., every INSEL block can be represented by the equation

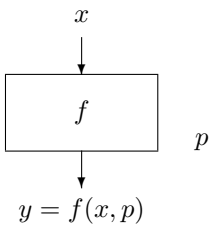
$$y = f(x, p, t)$$

Answer 1

In conclusion, it follows from these remarks that an INSEL block is a representation of

an explicit function, i. e., $y = f(x, y, p, t)$ is not allowed.[1]

Answer 2 Because it's a graphical programming language, INSEL represents the equation $y = f(x, p, t)$ by a graphical element. This picture is called INSEL block, too:



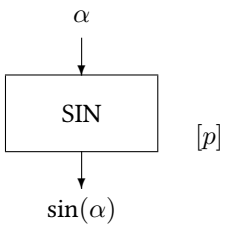
In the documentation, a rectangle is used to represent an INSEL block named f, for example. Inputs x come into the block via arrows which point into the rectangle from the top. Outputs y are represented by arrows pointing out of the block.

It is convenient to write the names of the inputs and outputs close to the corresponding arrows. Parameters are frequently written at the right edge of a block. As mentioned before, the number of inputs, outputs and parameters can range from zero to any positive number and depend only on the specific requirements of the block f.

By convention, block names in INSEL use all capital letters.

SIN block In the simple case of the function $y = \sin(\alpha)$ the block representation looks like





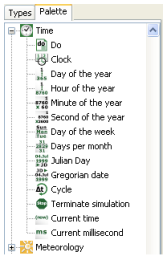
The SIN block requires exactly one input, namely an angle α. It returns exactly one output, the sine of α. The parameter $[p]$ is written in square brackets, which means that the parameter p is optional, i. e., not necessarily required.

Angles can be given in either degrees or radians. How does the SIN block know what is meant? By default, the SIN block assumes that α is given in degrees. This is equivalent to not specifying p at all, or by setting the parameter p equal to zero. If you want the SIN block to recognize α in radians you have to set p equal to 1. Any other value for p will result in an error message.

[1] There are exceptions to this rule in INSEL, but this is not the place to discuss them.

1.4.1 The Palette

The most convenient way to access and use INSEL blocks is from the Palette.

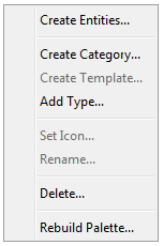


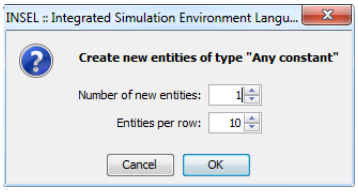
A category (like *Time*, for instance) can be opened by a left-mouse click on the small + symbol next to the icon of the category. An open category can be closed by a left-mouse click on the small − symbol. The opened *Time* category with its types is shown in the margin.

It contains the DO block (type *Do*), the CLOCK block (type *Clock*), the DOY block (type *Day of the year*) and so on.

There are three ways to insert INSEL blocks—or more precisely, entities of the block—into an INSEL model.

- A block[2] can be dragged from the palette into the work area by keeping the left mouse button pressed. INSEL displays a small + sign next to the mouse pointer once the mouse is moved into the work area. When the mouse button is released an entity of the block will be placed in the work area.
- When a block (i. e., type) is selected from the palette by a left-mouse click, then a click in the work area places a new entity of the selected block where you like. The selection of the block in the palette disappears. More than one entity of a marked block can be inserted in the work area as long as the *Shift* key on the keyboard is kept pressed. In this case, each mouse click in the work area will place one entity of the block in the work area.
- In order to create several copies of a block at once, select a type, the context menu of the palette can be opened by a right-mouse click in the palette area, and choose *Create Entities...*. In a dialog the number of required entities can be specified.





The new entities will be placed in the work area, arranged according to the number of entities per row.

Customizing the palette

[2] It would be more precise to say type, but in many cases it is more convenient to simply speak of blocks when the context is clear enough.

In INSEL 8 each user of a computer has an own copy of the palette. The location, where the user palette is stored can be on the local machine or on a remote computer, depending on the user-profile settings.

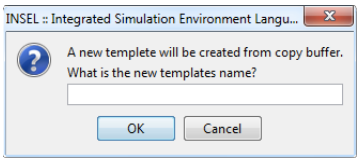
Palettes can be fully customized. Categories and types can be dragged with a left-mouse click to a different position within the palette. Keeping the *ctrl* key pressed (*cmd* on a Mac keyboard) creates a copy of the selected item instead of moving it.

New categories

The creation of new categories should be obvious. The position of the new category depends on the current selection within the palette. If nothing is selected, the new category will be created at the root of the palette.

New templates

All other palette operations can be made via the context menu, shown above. When one or more entities are selected in the work area, the *Create Template...* item is enabled. Choosing it, opens a dialog in which the name for the new template can be specified.



Please notice, that a template can contain more than just one block. Therefore, a template is not the same as a type.

New types

The creation of new types is a very advanced option and is explained in Module 11 :: *Programming INSEL blocks* of the Tutorial.

Etc.

All user-defined categories, templates, and types can have user-defined icons, they can be renamed and deleted at any time.

Rebuild palette

The factory setting of the palette can be rebuild. All user-made changes will be saved in a category named *SAVED ENTRIES*. The rebuild process is initiated only after a confirmation dialog.

1.4.2 Block entities

When INSEL blocks (entities) are created from the palette, they appear in the work area as minimised icons. The following picture shows a DO block and a SCREEN block, taken from the *Time* and the *Inputs and Outputs* category, respectively.



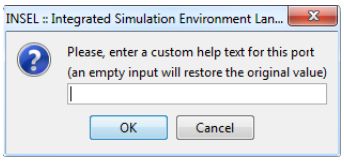
By default, each INSEL block is represented by a 32 times 32 pixel icon encapsulated in a frame. Block inputs are displayed at the left border, block outputs at the right side of the block's symbol.

Port tooltips

Block inputs and outputs can be accessed via ports. All ports in INSEL 8 have a tooltip

which displays helpful information to the meaning of the port. When the mouse pointer is moved to a port, the tooltip is shown after a short moment.

Default port tooltips can be overwritten. A double-click on a port opens a dialog where individual tooltips can be specified.



Connecting blocks

In order to connect an output port with an input port, click near one of the ports you wish to connect, keep the mouse button pressed and move the mouse pointer to the other port to be connected. Once you are close enough a small rectangle will show up, indicating that a release of the mouse button connects the selected ports.



Once connected, a connection line between input and output is shown. In general, an input port can only be connected with exactly one output port while output ports can be connected to an arbitrary number of different input ports.

Deleting connections

Block connections can be deleted one by one. A mouse click somewhere on the route of the connection and choosing *Delete* from the *Edit* menu or pressing the *Delete* key will dissolve the connection. When a block is deleted all its connections will be deleted automatically.

Trouble

If plenty of connection routes exist in a larger INSEL model, the routing algorithm – that is the part of VSEit which tries to find an ideal route for all connection lines – may not be able to find such a route and uses a fall back. In this case a short diagonal connection will be displayed which cannot be clicked on. If this happens, try to move the respective block and find a better routing for the block and then delete the connection.

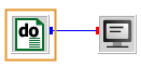
Moving blocks

Blocks can be moved to a different position in the work area by dragging them with pressed mouse button. The work area itself is infinite, which means, if you drag a block out of the visible part of the work area scroll bars will appear automatically.

More than one block can be moved at a time by selecting them first.

Selecting and deselecting blocks

A block can be selected by a mouse click on its icon. A frame will appear, indicating that a block is currently selected.



More than one block can be selected with the *Shift* key pressed. When more than one block is selected, a click any block deselects all other blocks, their frames disappear.

Alternatively, blocks can be caught with a rectangle that is created in the work area with the mouse, starting in the upper left corner and dragging the mouse pointer to a location in the lower right of the starting point.

All blocks can be selected using the *Select All* context menu.

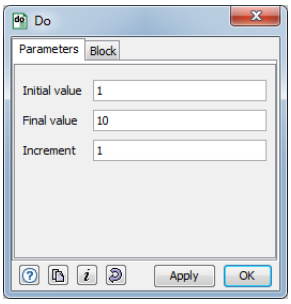
All selected blocks can be deselected by a mouse click on the white space of the work area. The selection of individual blocks can be inverted by pressing the *Ctrl* key (or *cmd* on Mac).

Deleting blocks

Selected blocks can be deleted by pressing the *Delete* key or via the *Edit – Delete* menu item or via a click on the Delete button in the tool bar. Please notice, that this action cannot be undone in the current version INSEL 8.3.

1.4.3 Entity editors

A double click on any INSEL block in the work area opens its entity editor. The range of entity editors varies from trivial to rather complex. This is the entity editor of the DO block:



Most INSEL blocks (better: entities) have two tabs in common: a *Parameters* tab and a *Block* tab.

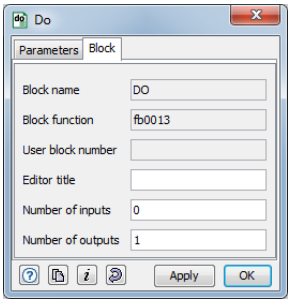
Parameters pane

The most important feature of the parameters pane is that you can access and modify all parameters of the coresponding INSEL block here. In case of the DO block these are the *Inital value*, *Final value*, and *Increment* parameters.

It is possible to use names of global variables (created with the DEFCON block) in any of the parameter fields. In the shown example, the DO block simply counts from one to ten.

Block pane

A click on the *Block* tab displays the block pane.



The block pane of an entity editor shows some insight information for the INSEL block, like the "real" *Block name* (DO, in this case), the name of the exported Fortran subroutine or C function implemented in a dynamic library, and the so-called *User block number* u, fixed by the inselEngine during model compilation.

Editor title

For each INSEL block entity a user-defined *Editor title* can be provided. The text will appear in the title bar of the entity editor and as a roll-over tooltip when the mouse pointer is moved across the minimized block in the block diagram. If no editor title is specified the roll-over tooltip displays the type name from the palette (e. g., Do), by default. After the first compilation of the model the default roll-over tooltip will change to type: u.

Input/output interface

The *Number of inputs* and the *Number of outputs* can be specified in the block pane of the entity editor – within block-specific limits for the number of allowed inputs and outputs.



The buttons in the bottom of the entity editor can be used for the following purposes (from left to right).

- The *Help* button gives direct access to the INSEL block reference page (Prerequisite: Adobe Reader must be installed).
- The *Clone* button opens another copy of the current editor. Changes made to any copy of an editor are synchronised whenever the *Apply* or the *OK* button is clicked.
- The *Info* button toggles the display of information to the block parameters, usually the physical units of the corresponding parameters.
- The *Reset* button rejects all unsaved changes made to the parameter settings and resets all attributes to the recently stored values.
- The *Apply* button saves the current values of all attributes without closing the entity editor window.
- The *OK* button saves the current values of all attributes and closes the entity editor window.

Execution

Well, when the model consisting of the DO block and the SCREEN block as used in the

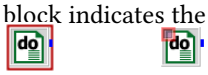
discussion so far, is executed the result is displayed in the output window.

```
Compiling new-1.vseit ...
No errors or warnings
Running insel 8.3.0 ...
   1.0000000
   2.0000000
   3.0000000
Normal end of run
```

Not very spectacular, but this example shows that everything seems to work.

1.4.4 Errors in networks

When a model has syntax errors like inconsistent number of inputs or parameters, for example, a red frame around the block indicates the block which causes the problem.

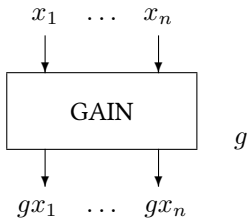




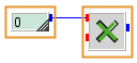
In addition, a toogle button in the *Types* pane can be used to indicate incomplete types and entities. If *on*, a small red square is shown in the upper left corner of entities which are causing problems.

1.5 Macros

As mentioned above, a collection of INSEL block entities can be saved in the palette as a template for further use. If, for example, we would like to rebuild the GAIN block



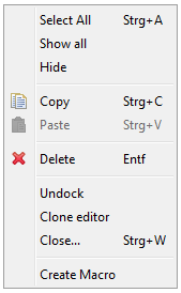
from a CONST block – type *Any constant*, and a MUL block – type *Multiplication*.



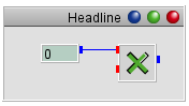
we'd select both blocks and create a template via the palette's context menu.

Creating and dissolving macros

However, it might be preferable, to combine both blocks into a macro and save the macro as template. We can do so by using the *Edit – Create Macro* menu or by using the context menu with a right-mouse click on the empty work area.



This will create a macro with a default title bar.



The title bar shows three buttons. From left to right, they can be used to open the entity editor of the macro, to maximize the macro to the full size of the current work area, and reduce its size again. The right button minimizes the macro. All other properties of the macro are inherited from ordinary entities.

Please observe that it is not necessary to add a macro input and a macro output port for the MUL block, because inner ports in the macro can be connected to other blocks across the border of the macro.



Only if you wish to be able to connect to the MUL block even when the macro is minimized the corresponding ports should be created via the macro's *Edit* function and its blocks pane. In this case, the minimized macro looks like any other INSEL block, as shown in the margin.

An option to create empty macros from scratch is to use the *Macro* type from the palette's *User blocks* category.

Macros can be created within macros. There is practically no limit for the depth of nested macros in INSEL.

One or more selected macros can be dissolved via the *Edit – Dissolve macros* menu or the macro's context menu.

Editing macros

In addition to using the buttons in the macro's title bar, the size of an opened macro can be modified by dragging its lower right corner with the mouse.

Macros can be moved by picking them up at the title bar or their frame, as indicated by a hand symbol of the mouse pointer.

Blocks can be moved from the work area into a macro by dragging them to the macro area. This is possible only if the target macro is opened. The drop option is indicated by a frame around the target macro. Please observe that any previous port connections will be conserved and that the ports are adapted accordingly. The same applies, when blocks are dragged out of a macro.

und sie bewegen sich doch (die Ports).

mit rev. 868 sollte nun das Ändern der Reihenfolge von Makro-Ports möglich sein. Dazu muss der Benutzer nur CMD/CTRL drücken, den Port anfassen und vertikal verschieben. Statt CMD/CTRL geht auch Klicken und Halten auf dem Port bis der Curser wechselt. Achtung: Beim Editieren der Makro-Hierarchie gibt es Situationen, in denen das Programm die Reihenfolge der Ports selbst wieder neu bestimmt.

Mit rev. 869 verschwinden Ports nicht mehr, wenn der letzte innere Link verschwindet. Achtung: This way, "orphan" ports may be created. These can only be deleted by shifting them to the bottom of the makro and, then, set the number of ports to a suited value.

2 :: INSEL programming concepts

The INSEL idea is based on a modular, block-oriented concept which adapts structured programming – a programming method which restricts algorithms to three basic programming structures, i. e., (i) sequence structures, (ii) if-then-else structures, and (iii) loop structures – to block diagrams. In computing science it has been shown, that all numerical problems can be solved with these three basic structures. Therefore, INSEL is a general-purpose programming language, which can – in principle – be adapted to any numerical task.

2.1 INSEL block groups

Although all INSEL blocks appear as named rectangles with inputs, outputs and parameters, each block belongs to a certain block group.

The following six block groups exist in INSEL:

- Constant blocks or short C-blocks
- Timer blocks or T-blocks
- Standard blocks or S-blocks
- Loop blocks or L-blocks
- Delay blocks or D-blocks
- If blocks or I-blocks

S-blocks

As the name indicates already, the group of S-blocks is the least specific. In Module 1 we have used the SIN block and the PLOT block, for instance. There is nothing special about them. They are typical S-blocks. When the SIN block gets an input value α, it calculates the corresponding sine value and connects the result with the block's output – finished. When the PLOT block gets a data point with coordinates x and y as inputs, it plots the data point – finished. But who delivers the inputs and how often – and who decides when the simulation run is through?

T-blocks

In the simple DO–SCREEN example of Module 1 we have specified by the parameters (initial value 1, final value 10, increment 1) the DO block "fires" 10 times: a 1 in the first step, a 2 in the second step, a 3 in the third step, and so on until the block outputs a 10 in the 10th step. Then there is nothing left to be fired – hence the DO block sends a signal to the inselEngine to end the run. Blocks having the ability to control a simulation model are called timers, or Timer blocks or just T-blocks.

It is not compulsory to include a Timer block in an INSEL application. The following example does not use a Timer block, for instance.



This simple application uses three blocks:

- The CONST block, which just delivers a constant output as specified by a parameter, 45 (which we humans interpret as $45°$) in this case.
- The SIN block, which calculates the sine of its input.
- The SCREEN block, which can be used to display alphanumerical information on the computer's screen.

Test it. You find the CONST and SIN block in the Mathematics category of the palette under *Constants* > *Any constant*, and under *Trigonometric functions* > *Sine*, respectively. The SCREEN block can be found in the Inputs and outputs category as Screen output.

When you run the model you will see the output 0.70710677 from the SCREEN block, which means that you have calculated $\sin(45°) = 0.70710677$.

Sorting

INSEL has executed every block one time: The CONST block which defines the output 45, the SIN block which calculates $\sin(45°)$ and the SCREEN block which displays the result – ready. Please observe that INSEL calls the blocks exactly in the order CONST, SIN, SCREEN, no matter in which order you have constructed the block diagram.

This means that there must be some mechanism in INSEL which converts a block diagram description into a calculation list – this mechanism is called sorting algorithm and is an integral part of the inselEngine. You will learn more about the inselEngine in due course.

C-blocks

No matter whether there is a T-block in a model or not the CONST block always needs to be executed only once and never again. Blocks with this property belong to the group of C-blocks.

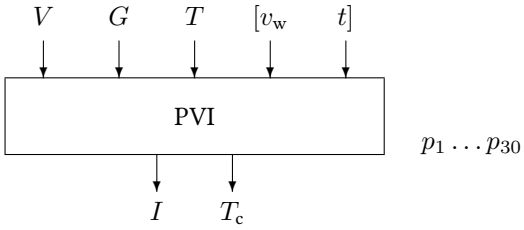
We can conclude that we have seen examples of a C-block (the CONST block), S-blocks (the SIN and SCREEN block), and a T-block (the DO block). This Module deals only with these three block types. Loop, Delay, If, and Macro blocks will be handled later.

2.2 Basic photovoltaics

So far, we have used only quite primitive blocks. One of the nice aspects of INSEL however, is that basically all blocks look alike and can be treated more or less in the same way, regardless of whether they are primitive like the CONST block or more complex like the PVI block, which we had used already in the previous Module.

The PVI block is located in the category Electricity under *Photovoltaics* > *Photovoltaic current (c-Si)*. This is the design of the block:





As indicated by the bitmap of the PVI block in the left margin, this block simulates the behavior of solar cells – PV is short for photovoltaics, i. e., the direct conversion of electromagnetic radiation into electricity. The sketch of the PVI block shows, that this block requires (up to) five inputs and – don't get shocked – 30 parameters.

With this information the block calculates two outputs, the PV current I in ampere (this I gives the block its name) and the cell temperature T_c in degrees Celsius. PVI is also a Standard block.

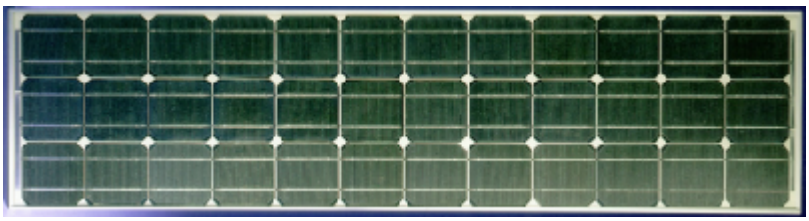
In order to use the block it is necessary to connect at least three inputs: the voltage V of the device in volt, the global radiation G in W/m^2. T stands for a temperature in degrees Celsius. This input has a specific role and will be discussed later.

Two-diode model

Concerning the parameters: The electric properties of the solar cell are modeled by a rather detailed physical model, well-known as the two-diode model. For the calculation of the thermal properties of the device an energy-balance differential equation is used. In addition, the block can be used for any particular electrical connection of cells and modules in series and in parallel. All in all this gives 30 parameters.

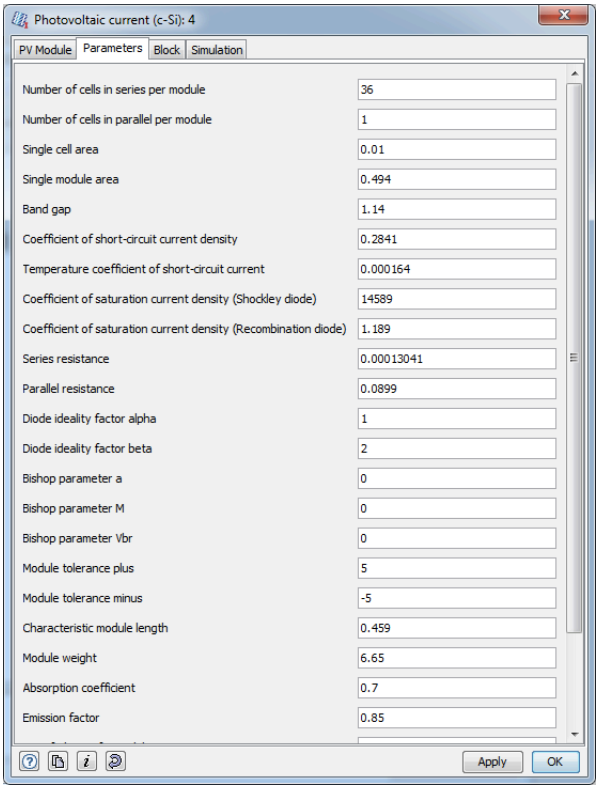
We will not go into the details here. More information about PV modeling in INSEL can be found in Module .

INSEL contains more than five-thousand parameter sets for practically all modules that are available on the market or have ever been produced. The photo shows one of these modules – the Siemens module SM 55.

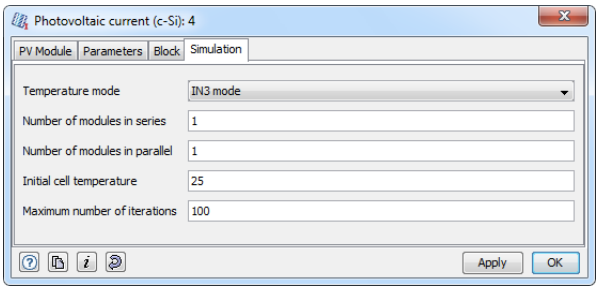


For historic reasons, the default parameter set used by the PVI block simulates exactly this module.

The "physical" parameters can be seen after a double-click on the PVI block on the parameters pane:



The “variable” parameters can be accessed via the Simulation pane:

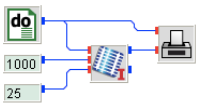


As we have seen, the PVI block calculates the PV current I as a function of the voltage V. Analogue, there is a block called PVV (Photovoltaic voltage (c-Si)) which calculates the PV voltage V as a function of the current I. It always depends on the actual problem, which one is better to use.

I-V curves

We have already used the PVI block in the previous Module for a plot of the voltage-current characteristics, the I-V curve under standard test conditions STC

(defined as global radiation equal to 1000 W/m^2 at a spectral distribution of AM 1.5 and a module temperature of 25 °C). We repeat the block diagram:



The DO block is used to vary the voltage in a range between 0 and 25 volt in steps of 10 millivolt. Two CONST blocks provide values for the global radiation and the module temperature. The PLOT block is used to display the I-V curve.

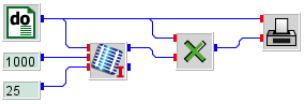
The parameter *Temperature mode* is set to IN3 mode, by default, which means that the module temperature is given by input number 3 – which comes from a CONST block with value 25. The other temperature modes will be discussed later.

Exercise

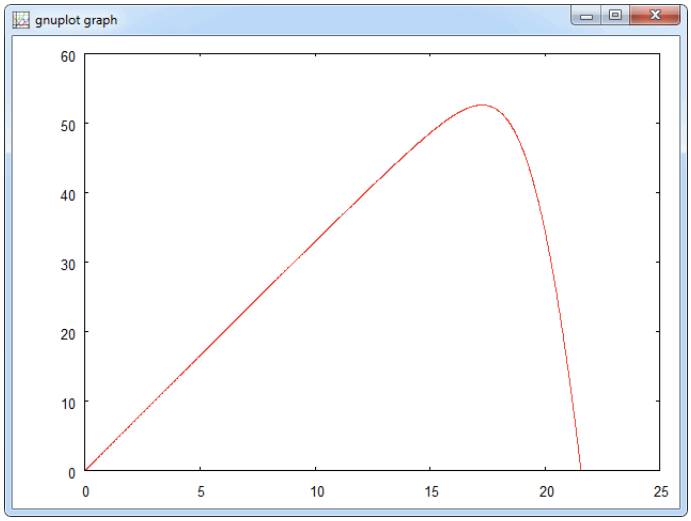
Now, please reconstruct the block diagram from scratch and run it until you see the I-V curve displayed by the PLOT block.

DC power

It is now easy to calculate and plot the DC (direct current) power output $P_{\mathrm{DC}} = I \cdot V$ of the module as a function of the operation voltage. All we need to do is to multiply the first output of the PVI block (the current I) with the output of the DO block (the voltage V). This can be done with the S-block MUL which we know already from the previous Module.



This block diagram solves the problem and shows the DC power as a function of the voltage.



We see from the graph that the output power depends very much on the operation voltage with a maximum of about 53 W_p (Watt peak) close to 17 volt. This point (V_m, I_m) is called the maximum-power point in photovoltaics and defines the peak power or nominal power of the module under standard test conditions. Under real operating conditions the maximum-power point varies, since it depends on global radiation and module temperature.

In most real PV generators there will be a device called maximum-power-point tracker which will always operate the generator close to this point. In a numerical simulation this operating point must be found by an iteration process. The INSEL block which performs this iteration is called MPP. This Loop block will be handled in Module .

There is another INSEL concept that can be learnt from the PVI block.

2.3 The INSEL concept of time

In a real-world PV generator the module temperature will depend on the weather conditions and will be a function of time. When you look at the temperature modes of the PVI block, you find the DEQ mode (differential equation). In this mode the module temperature is calculated as a function of voltage V, global radiation G, ambient temperature T_a, wind speed v_w, and time t.

So far, the PVI block we used had only three inputs. Hence, two more inputs for the wind speed v_w and time t are required. Remember, we can define the number of block inputs through the blocks pane.

Time in seconds

In the classical simulation environments like CSMP and SPICE, for example, and even in most modern ones like MATLAB and Simulink time plays an extra-ordinary role. In

INSEL time is just a variable among others. It is always an explicit block input, which has a time-dependent behavior. As a general rule, time in INSEL always runs in seconds.

Variable time steps

In temperature mode DEQ the PVI block must be supplied with a time input in seconds. If, for example, you want to run a PVI block in time steps of one hour, you must deliver values like 0, 3600, 7200, and so on to the PVI block.

Since the PVI block is able to remember the value of the time input of the latest call, the block itself can calculate the actual time difference between the previous call and the actual one, i. e., the time step. A consequence of this concept is that the time steps of a simulation run in INSEL do not at all need to be constant.

The PVI block can deal with any time step, no matter whether the time step is in the range of seconds or hours.

INSEL Lab

It's time for a concrete example. Let us observe how an SM 55 PV module heats up with time.

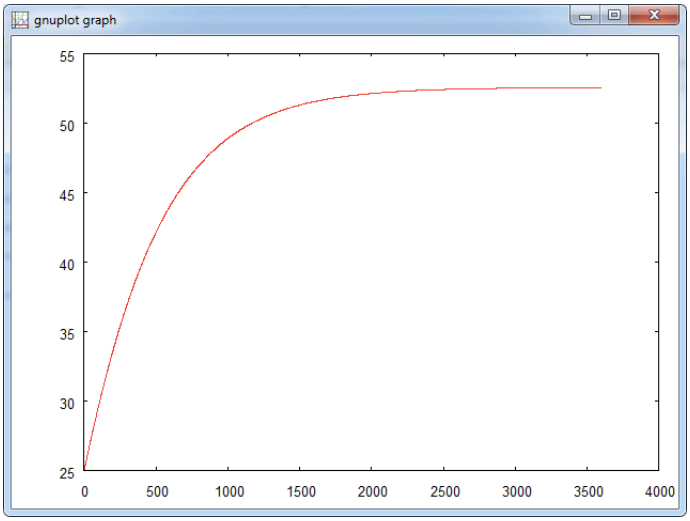
You will gain the highest benefit from INSEL when you do not think of writing simulation applications, but perform "close-to-real experiments" in a laboratory.

Let us place a Siemens SM 55 module in a laboratory environment, and wait until it is in equilibrium with ambient conditions, assumed to be $T_a = 25\,°C$, no air movement, i. e., $v_w = 0$ m/s and completely dark, i. e., $G = 0$ W/m^2. This is equivalent to setting the initial value for the cell temperature parameter of the PVI block to 25 degrees Celsius.

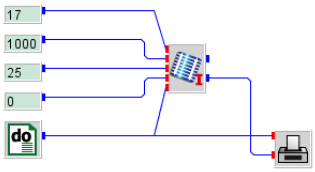
We then switch on a light source which illuminates the module with 1000 W/m^2, for example. In a real experiment we could use a PT-100 for the module temperature measurement, in INSEL the PVI block with the parameters of the Siemens SM 55 module provides the cell temperature T_c as a second output.

After about one hour we would expect equilibrium conditions for our experiment. Hence, let us run the simulation for one hour. Would you like to solve the problem yourself or just look at the solution?

Here is our solution:



The corresponding block diagram looks like this und moechte noch verschoenert werden:



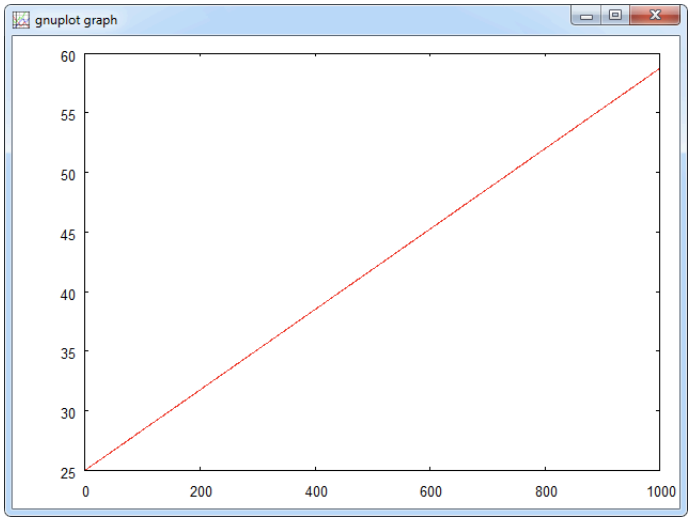
NOCT temperature

There is one last uncovered temperature mode of the PVI block, the NOCT mode. NOCT is short for nominal operating cell temperature. It is defined as the equilibrium module temperature under a global radiation $G_{\mathrm{NOCT}} = 800\,\mathrm{W/m^2}$, ambient temperature of 20 degrees Celsius and a wind speed of 1 m/s.

In NOCT mode the PVI block makes the linear interpolation

$$T_{\mathrm{c}} - T_{\mathrm{a}} = (T_{\mathrm{NOCT}} - 20\,^{\circ}\mathrm{C})\,\frac{G}{G_{\mathrm{NOCT}}}$$

You should quickly check the module temperature as a function of global radiation from 0 to 1000 $\mathrm{W/m^2}$. For the voltage you can use 17 volts – we have seen before that the voltage near the maximum power point of the module is of that order.

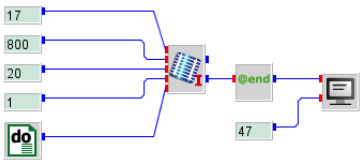


Exercise Adapt the DEQ mode example to NOCT conditions and compare the equilibrium temperature with the NOCT value.

Hint If you use the ATEND block with input T_c and connect its output to a SCREEN block, the SCREEN block displays only the last calculated temperature value. You find the ATEND block under the *Mathematics* > *Logics* category as At end. The ATEND block is already a first example of an I-block which will be discussed in more detail in Module .

Solution The result is 38.74 degrees Celsius compared to an NOCT of 47 degrees – frustrating or wrong?

This is the corresponding block diagram welches auch noch verschoenert werden moechte:

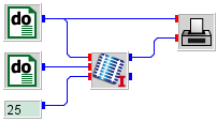


2.4 Nested Timer blocks

Timer block can be nested. This means that two or more T-blocks can be connected in series but not in parallel.

It's best to explain this with a concrete example: Assume that we want to use the PVI block to display not only one I-V characteristic but a set with the global radiation as curve parameter. In the first place, one would simply replace the CONST block for the

radiation by another DO block, and set the parameters of the new DO block to 200, 1000, with an increment of 200 W/m^2, for instance.



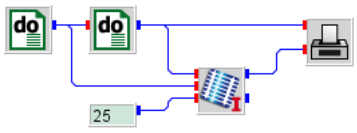
When you run this block diagram, INSEL will generate an error message which says "Too many timer blocks specified". Why?

The two DO blocks are not connected in series but in parallel. It is not clear how INSEL should handle the model. Shall INSEL first fix the voltage value to zero and then run through all radiation data, return to the voltage block, increment the voltage to 0.01 volt, run through all radiation data again, and so on? Or shall INSEL first fix the radiation value to 200 W/m^2 and then run through all voltage values, return to the radiation block, increment the radiation to 400 W/m^2, run through all voltage values again, and so on?

From a curve plotting point of view, the second option is clearly better. But how can we express that we prefer the second option?

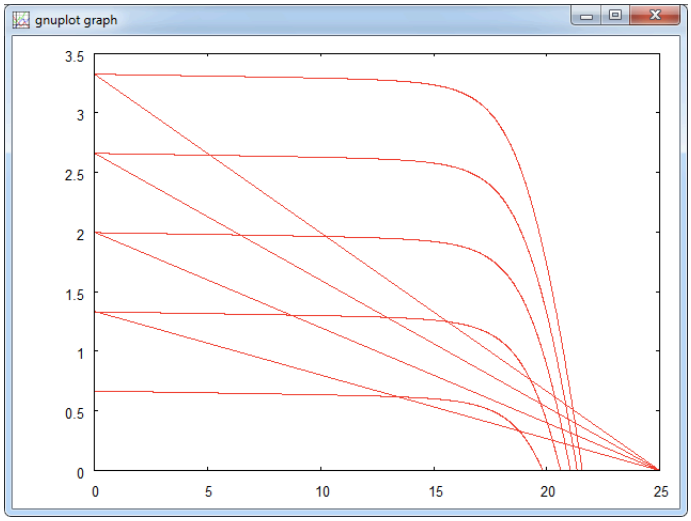
Key concept

One of the key concepts in INSEL is that blocks cannot be executed before all block inputs are "known," i. e., have values. So, if we add a new input port to the DO block which varies the voltage and connect it to the output of the DO block which varies the radiation, then the block for the voltage variation depends on the radiation block – they will be connected in series, nested!



Hence, the input of the DO block serves the purpose of arranging the two DO blocks in a fixed order.

When you run the model you will see some scrambled lines like this:



What happened? We have five blocks in the model:

- A CONST block which defines the module temperature
- A DO block which varies the global radiation
- A DO block which varies the voltage
- A PVI block which calculates the PV current
- A PLOT block which plots the data points

Calculation list

How does INSEL execute the blocks? The order in which the blocks in a model are executed is called calculation list in INSEL It can be displayed via the *Simulation* > *Show calculation list* menu.

```
Number    Block       Group    Jump
-----------------------------------
     5    CONST           C       1
     1    DO              T       1
     2    DO              T      -1
     3    PVI             S       1
     4    PLOT            S      -2
-----------------------------------
```

We see that at first the constant temperature value is set, then the first DO block with user block number 1 sets the radiation to 200 W/m^2, then the second DO block number 2 outputs a voltage of 0 V, then the PVI block calculates the PV current I, then the PLOT block plots the first data point ($x = 0, y = I_{sc}$), the short-circuit current. And then?

The last column in the calculation list is the so-called jump parameter. The value for the PLOT block is -2, i. e., INSEL returns control to the lower DO block number 2 in the

calculation list. The DO block 2 increments the voltage to 0.01 V, the PVI block calculates the corresponding current, and the PLOT block plots the second data point, while drawing a linear interpolation line between the first and the second point.

This process continues, until the PLOT block gets the last data point from DO block number 2, which is equal to 25 V, and plots it – again with a short linear interpolation line between the last value ($x = 24.99, y = 0$) and the actual point. And then?

INSEL again returns control to DO block 2. But this block has nothing left to do. So, it gives control to the next "upper" T-block, which is DO block number 1. This block increments the radiation to 400 W/m^2, and DO block 2 gets control again.

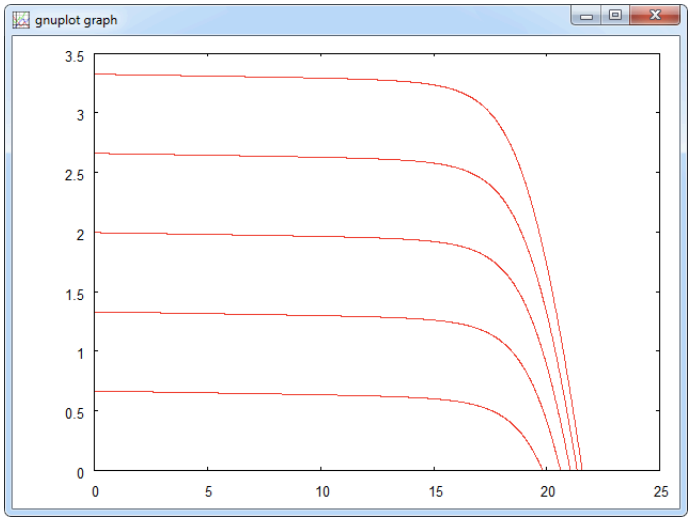
The DO block performs a reset since its input has a changed value and outputs its initial value again, PVI then calculates the short-circuit current I_{sc}(400 W/m^2), and the PLOT block gets the next data point ($x = 0, y = I_{sc}$(400 W/m^2)) and operates as always: draws a linear interpolation line between the last point and the actual point – et voilà!

Parametric plot

The PLOT block had no chance to recognize that the radiation has changed and that we wanted to see a new line, i. e., simulate a pen-up pen-down operation.

A way out is to "inform" the PLOT block about the curve parameter via the output of the DO block which varies the radiation. This means that besides the x- and y-coordinate the PLOT block requires a third input. The name of the block with such an extra input is PLOTP. Both, the PLOT and the PLOTP block can be found in the *Inputs and outputs* category of the palette as types *Gnuplot graph* and *Gnuplot graph (parametric)*, respectively.

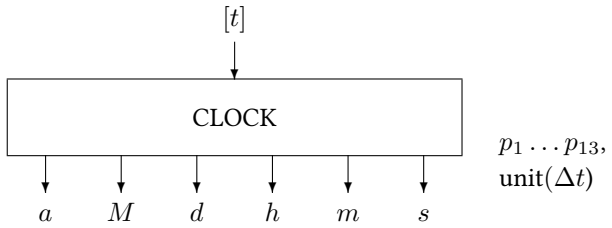
So, replace the standard PLOT block by the parametric PLOTP block. The first input is the curve parameter – the output of DO block number 1 – the second input is the voltage – the output of DO block number 2, the third input the corresponding current. Now the result looks as it should look like.



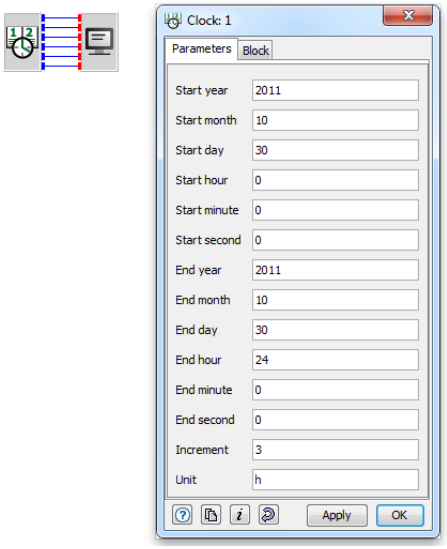
2.5 The Timer blocks CLOCK and FDIST

So far, the only timer block we have used is the DO block. A second example for an INSEL T-block is a block called CLOCK, found in the Time category as Clock. It behaves very much like the one you are probably wearing around your wrist: It runs through time in hours, minutes, and seconds, every day, month and year. The main difference is that a wrist-watch shows the time in which we humans are stuck. A simulation of a clock is much more flexible. We can let it run from any starting point to an end in any time step we like.





Start year, month, day, hour, minute, and second, end year, month, day, hour, minute, and second is the correct order of the parameters, followed by an increment Δt and a string for the unit of Δt. The unit must be one of these: a (for years), M (for months), d (for days), h (for hours), m (for minutes), or s (for seconds). If for example, we let the CLOCK run for one day from midnight to midnight in steps of 3 hours, for example, this object does the job:



We have added a SCREEN block so that we can observe what the CLOCK block does exactly.

```
Compiling clock.vseit ...
No errors or warnings
Running INSEL 8.3 ...
 2011.   10.   30.    0.    0.    0.
 2011.   10.   30.    3.    0.    0.
 2011.   10.   30.    6.    0.    0.
 2011.   10.   30.    9.    0.    0.
 2011.   10.   30.   12.    0.    0.
 2011.   10.   30.   15.    0.    0.
 2011.   10.   30.   18.    0.    0.
 2011.   10.   30.   21.    0.    0.
Normal end of run
```

Nothing spectacular happens. But you should take note of some details.

- We let the clock run exactly until 24:00:00 of 30 October 2011. This is absolutely equivalent to running the clock until exactly 00:00:00 of 31 October 2011.
- The CLOCK block runs in logical time steps and we are going to use it for exactly that purpose in most cases, for time step simulations. Logically we have defined a constant time step of three hours. The CLOCK block outputs the time information only once per time step and the time which is on output then is always the “left end” of the time interval, i. e., our first time interval is between 00:00:00 and 03:00:00 o’clock, but the CLOCK shows 00:00:00 “all the time.”

  As a consequence, the last output of the CLOCK is at 21:00:00 for another three hours. At midnight, the clock stops. When you count the lines that the SCREEN

block writes you find eight lines times three hours gives 24 hours – exactly what we wanted. Mathematically spoken, the CLOCK block runs through the interval with the left end closed, the right end open, i.e. [00:00:00,00:00:00).

Star format

:: We have used the Format string (6F6.0) in the SCREEN block's parameter. This is a Format string in Fortran language standard. Fortran formats will be discussed in more detail in the next Module "Reading and writing data files."

For the time being, you should note that there is a so-called star format, which you can use easily by just typing in an asterisk * in the parameter field. Please try it, the star format is really useful for output when you do not know the order of magnitude of your results in advance.

You may ask "What are applications of the CLOCK block?" Here comes one which opens a huge field of applications of the block: Solar energy applications. In INSEL almost all of them make use of the Gregorian calendar.

2.6 Solar radiation

Radiation outside atmosphere

A simple example for a solar energy application is the block GOH which calculates the extraterrestrial radiation – that is the solar radiation in Space outside the Earth's atmosphere.



Inputs to the block are basically the outputs of the CLOCK block – at least the first four inputs are necessary. Required parameters are latitude φ, longitude λ, and time zone Z of the observer's location.

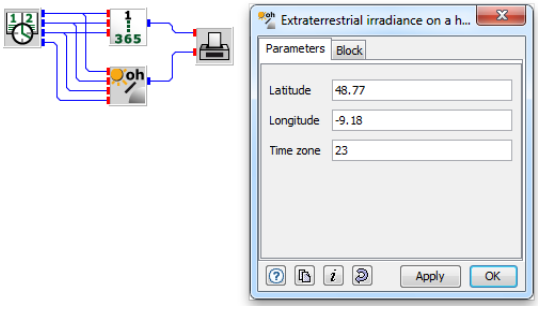
Latitude, longitue, time zone

The latitude of an observer is defined from the equator towards the poles, northern hemisphere positive, southern hemisphere negative – Stuttgart in Germany has a latitude of about $48.77°$ north, for example. The longitude is defined as west of Greenwich, a cosy suburb of London in the U.K. We have to go almost all the way around the globe to reach Stuttgart at its longitude of $350.82°$, but in INSEL we can also use $-9.18°$ as longitude value for Stuttgart.

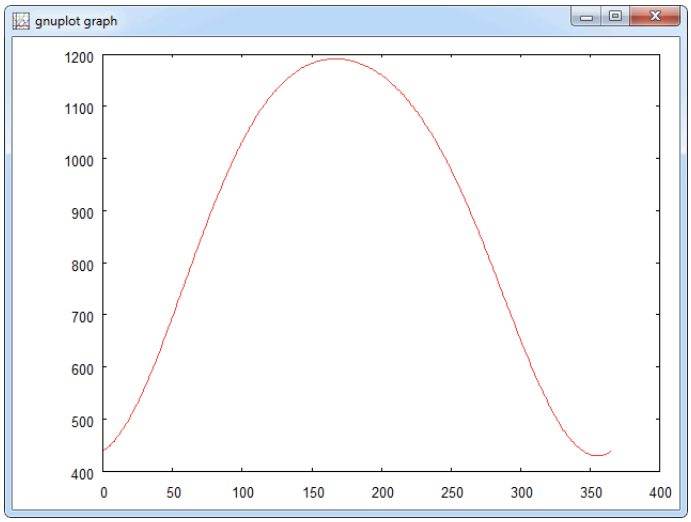
Time zones are also defined with respect to Greenwich defined as time zone zero – known as Greenwich Mean Time GMT – with approximately 15 degrees per time zone. The time on our German clocks shows Central European Time CET which corresponds to time zone 23. During summer we use daylight-saving time, which means we bring the

time on our watches one hour ahead in spring, and put it back in fall. The last input of the GOH block is 0 by default (i. e., does not consider daylight-saving time). If you connect a one with this input GOH interprets the given time as daylight-saving time.

Let us calculate the annual course of the extraterrestrial radiation on a horizontal surface at noon for our home location – in our case this is Stuttgart with the given geographical parameters.



You find the DOY block (Day of the year) in the Time category, block GOH (Extraterrestrial irradiance on a horizontal surface) in category Meteorology – Solar radiation. The result is shown in the next graph.



You should grasp two points from this example:

- A technical point: The CLOCK block is nice in time handling, it allows us to think of time like we are used to it. But for numerics it is rather bad. When we plot time

series, we need a continuous signal, not something strange like the Gregorian calendar with all its exceptions, like leap years etc.

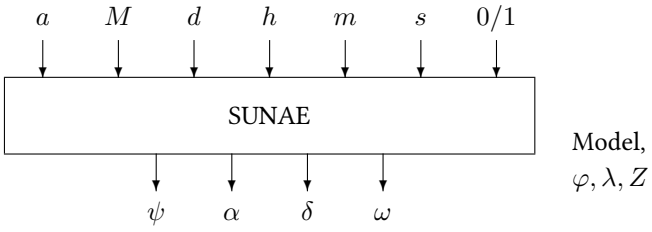
INSEL offers several routines (blocks, of course) that handle this aspect. In the above example we have used the day of the year block DOY which converts a Gregorian calendar date to a continuous signal. Similar blocks are the hour of the year block HOY, and minute of the year block MOY, for example – all found in the Time category.

:: A point of general interest: From the extraterrestrial radiation plot you can observe that in our place (Germany) the extraterrestrial radiation at the beginning of the year is much lower than in the middle of the year – there is a factor three between the values. The reason – which corresponds with our every-day-life experience – is due to the fact that in summer the Sun is much "higher" as compared to the winter case.

Solar position

The exact position of the Sun at any time can be calculated with the Standard block SUNAE (Meteorology – Geometry – Position of the Sun).





Horizontal system

The meaning of SUN in the block name is self-explaining, A stands for azimuth, in INSEL denoted by the Greek letter ψ, E stands for elevation, in INSEL denoted by α. Azimuth and elevation is one coordinate system which can be used to describe the solar position relative to a human observer. It is a very natural coordinate system, because it puts us as the observer into the center. The azimuth is the direction in which we see the Sun, rising in the east ($\psi \approx 90°$), moving via south ($\psi = 180°$) and setting in the west ($\psi \approx 270°$). Please notice that observers in the southern hemisphere have a different view.

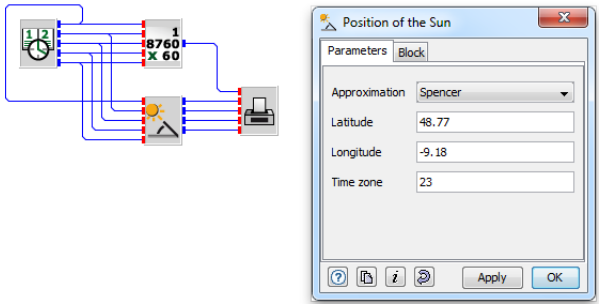
Equatorial system

The SUNAE block also outputs the solar position in a second coordinate system, which uses declination δ and hour angle ω as coordinates.

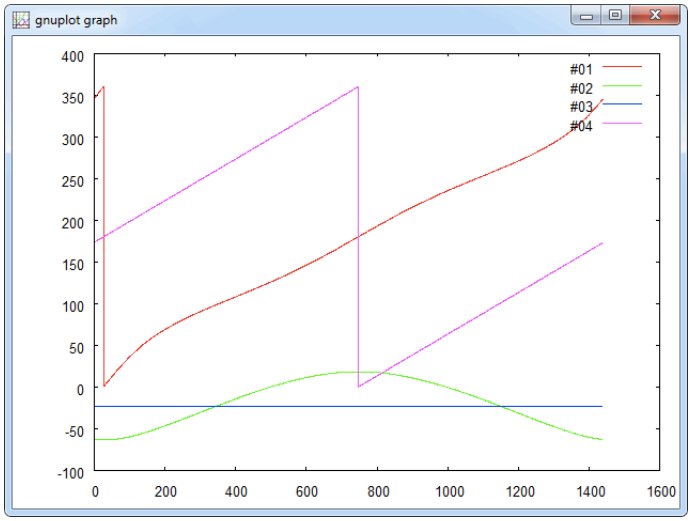
To understand this coordinate system is a bit less intuitive. But imagine to be located at the center of the Earth, with the Earth as a globe made of glass with a line grid for the latitudes and longitudes on its surface. Every day the Sun revolves once around this glass globe, following (almost) exactly a constant latitude. The angle between the equator and this latitude is called declination δ and is independent of any observer on the Earth's surface. We know that it varies between $+23.45°$ (our northern hemisphere summer) and $-23.45°$ (our winter).

The other angle, which describes the movement of the Sun around the Earth during a single day is the so-called hour angle ω. When a second observer is placed on the glas globe, his position and the moment when the Sun crosses this observer's longitude defines the hour angle $\omega = 0°$. Starting from here, the hour angle is counted positive as it follows the Sun on its way around Earth. The hour angle $\omega = 0°$ defines the true solar noon of the observer on the surface.

The following example shows the four coordinates for one day, 1 January 2012 at the location of Stuttgart, Germany.



This is the result.



You find the SUNAE block in the Meteorology category under *Geometry* > *Position of the Sun*.

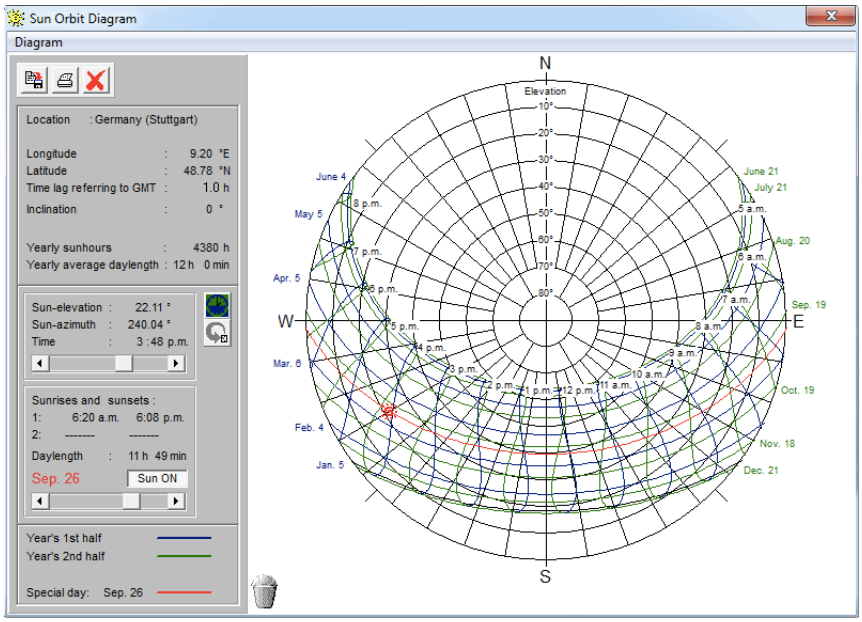
Three approximations for the calculation are currently implemented: The model of Spencer is the fastest in calculation time but the least accurate, the model of Holland and

Mayer is a good compromise between calculation time and accuracy. The model of Michalsky is rather high in accuracy, it is the algorithm which is used in the astronomical almanacs.

With INSEL we could use the SUNAE block for the operation of a computer-driven pyrheliometer (a device to measure the direct solar radiation, which requires accurate two-axis tracking of the solar position), but this is beyond the scope of this Module.

SunOrb

If you are further interested in an understanding of the movement of the Sun, there is a nice tool called SunOrb that has been programmed at the University of Bochum, Germany in the group of Prof. Dr.-Ing. H. Unger. The program can be used to calculate and draw solar diagrams like the following one for Stuttgart.



You find SunOrb under the Tools menu or you can start it directly from the tool bar with a click on its icon .

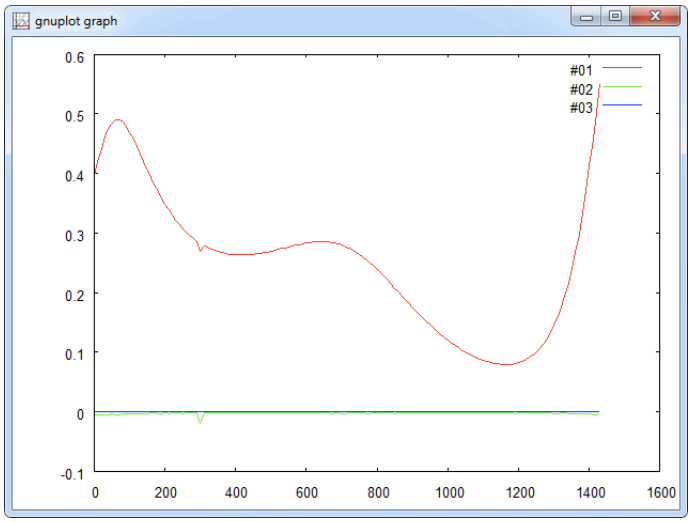
Exercise

As an exercise with the SUNAE block you can compare the accuracy of the three models. (Hint: Use three copies of the SUNAE block, each with a different model. Take the Michalsky model as reference and calculate differences to the coordinates of this model. If you follow this, you will need a Summation block SUM and a Change sign block CHS – you find both of them under the Math menu.)

Solutions

We provide four solutions in the `examples\tutorial\module2` directory in the files `sunae3a.vseit` (azimuth), `sunae3e.vseit` (elevation), `sunae3d.vseit` (declination), `sunae3o.seit` (hour angle).

As a result of the comparison for the azimuth angle, for example, we get this graph:



It shows the deviation of the azimuth angle calculation against the Michalsky model (red), the Spencer model (blue) overestimates the azimuth angle by up to 0.5 degrees on our reference day, 1 January 2002 in Stuttgart, whilst the deviation of the Holland/Mayer model (green) is almost not visible.

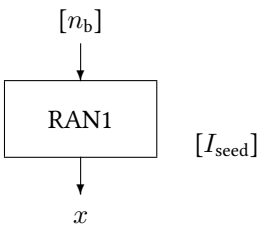
There are many more applications of the CLOCK block. We come back to some others in Part II.

Random numbers

Let us now turn our attention to the FDIST block, a Timer block which can be used to calculate the frequency distribution of any time series.

For the time series generation we use two random number generators, one that gives uniformly distributed, and one that gives normal or Gauss distributed random numbers. The first block is called RAN1, the second is called GASDEV, both are Standard blocks. You find these two blocks and the T-block FDIST in the Statistics category under Random numbers and Distributions, respectively.

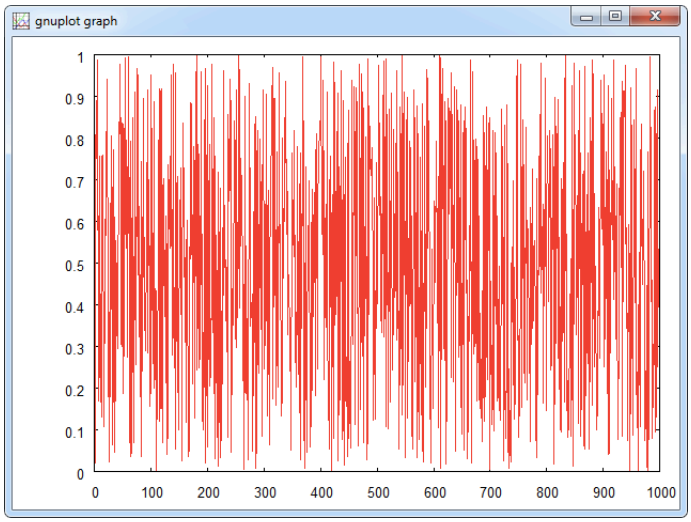
We start with the RAN1 block for the generation of uniformly distributed numbers.



The block has an optional input, and an optional parameter I_{seed}, which can be used to initialise the block – different instances of the block can be used with different I_{seed} values and generate different time series, but all will have a uniform distribution. We start with 1000 numbers.



We have chosen a value of 1536 for I_{seed}. Please notice that the generated time series depends only on the I_{seed} value, so that the "random" numbers can always be uniquely reconstructed. If a different time series is needed at each simulation run, I_{seed} can be set to 0: the generator is then initialised randomly. The time series plot looks really random:

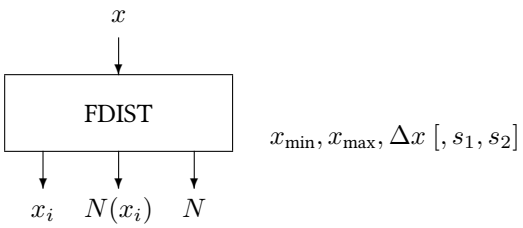


Difference between C-blocks and S-blocks

Maybe you have been wondering about the fact that we used the RAN1 block without an actual input. Since RAN1 is an S-block, it is automatically called by the inselEngine in every time step (of the main timer, which is the DO block in this case) of the simulation run. This is the main difference between C-blocks and S-blocks.

Now let's have a look at the FDIST block.

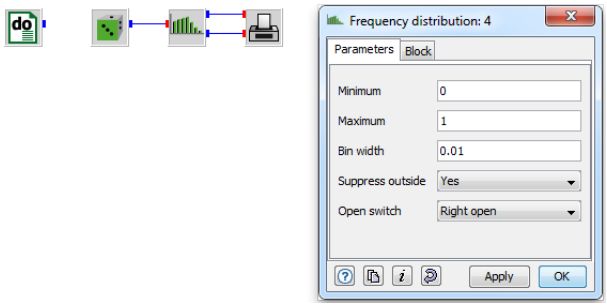




Frequency distributions

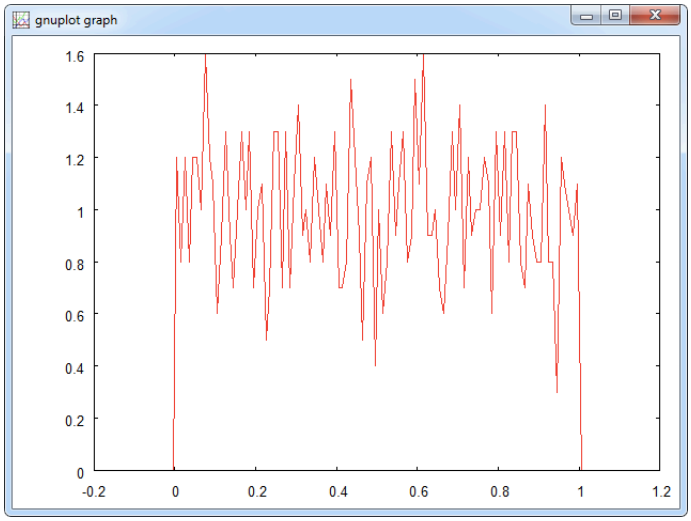
As expected, it has one input x for one value of the time series per call. Through the parameters we can fix the interval $[x_{\min}, x_{\max}]$ for the bins width Δx. Let us skip the optional parameters for the moment. Well, how do we expect the block to operate? We will deliver a time series (our RAN1 numbers) to the block's input. So far, so good. But when and how shall the block bring the results to the outputs?

Obviously, the block has to "wait" until the end of the time series to be then "informed" by the inselEngine that it is time to start the action, and that means: Play the role of a timer and deliver – one after the other – the x-coordinate x_i, starting with $i = 1$, the normalised number of data $N(x_i)$ in bin i. In addition, the total number of data N is an output which can be used for a the calculation of the absolute number data in a bin, for instance.

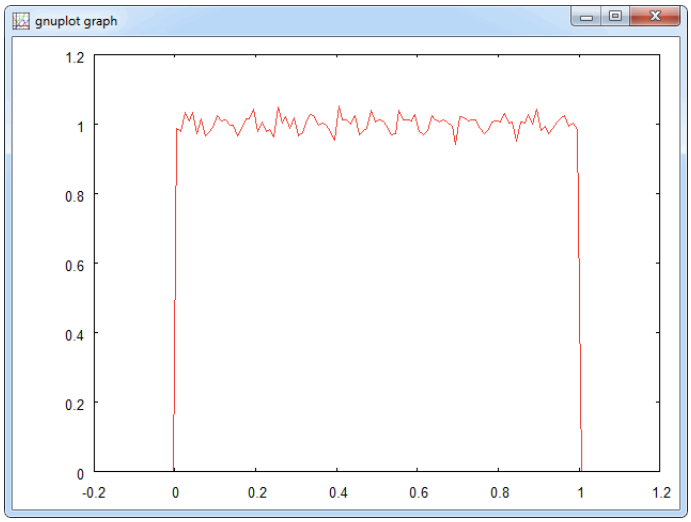


Due to the behavior of FDIST we can directly connect a PLOT block to the outputs of FDIST. Please notice that it is not necessary to connect the DO block with any other object. Due to its presence it generates a number of steps according to its actual parameter settings.

The result for 1000 numbers looks as follows.



For 100 000 numbers the distribution is already much smoother.



Exercise Plot the Gauss distribution on the basis of one thousand, ten thousand, one hundred thousand and one million normal distributed random numbers.

Solution See file `fdist.vseit` in the `examples` directory.

We could continue this Module with an extensive collection of examples for different blocks, but C-blocks are rather boring (and there aren't too many in INSEL), concerning T-blocks we have already looked at some important ones, and S-blocks? Well, there are

some hundred available in the different toolboxes. Perhaps, at this stage you should take your time and go through the block reference of INSEL. This will give you an overview on some of the basic blocks. When you do so, check the block group first and skip all non C-, T-, and S-blocks in the first run.

Questions

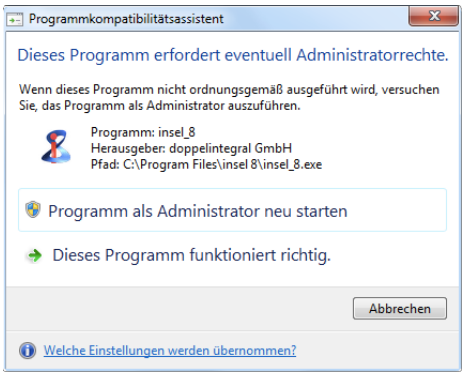
- There are seven different block groups in INSEL. Three of them have been discussed in detail: C-blocks, S-blocks and T-blocks. Can you explain in your words what the difference is?
- The PVI block has been used to calculate the characteristics of photovoltaic modules. Could you generate a graph which shows the I-V characteristic of a PV module for different module temperatures in one graph?
- When you create an INSEL block diagram INSEL sorts your model and creates a calculation list. Can you set up the calculation list for the example on page 35 that we used to show the four coordinates of the SUNAE block for one day, 1 January 2002 at the location of Stuttgart, Germany?

3 :: Reading and writing data files

In this module, you will learn how to read data from files and write data to files and how these data can be used in simulations. We will concentrate on data files which contain meteorological data since meteorology is one of the most widely-spread applications of INSEL.

A general warning

Reading files means that you enter explosive ground. When you do not exactly know, what kind of information is saved in a data file that you are going to read, it is quite probable that your program crashes with a message similar to this:



Luckily in computing this means that only your program smashes, or if the blunder is too bad, you have to restart your computer.

The story of writing data is even more dangerous. Writing data to files means that you are manipulating bits on the hard disk of your computer. You can imagine, that if – by accident – you change some of the bits in the Windows operating system, this may lead to a really serious crash. In the worst case, you can throw away your computer and buy a new one.

So, it's really worth to study this Module with the necessary care!

Let us start with the seemingly trivial question "What is a data file?"

Naming conventions

Whenever you work with a computer you work with files – it is impossible to do something with a computer that is not related to files. Under Windows the Explorer – which probably everybody has seen crash due to wrong file handling – is a tool which lets you organise millions of files. All these files have names following a specific naming convention.

In the "historic" age of the eighties and nineties of the 20th century the naming convention was: A file name may have a maximum of eight characters, followed by a dot and a file extension. The file extension was restricted to a maximum of three characters.

Under Windows a file could be saved under any path name with the same naming convention.

Today, everything is more comfortable – a file name can be quite lengthy and may even contain space characters, more than one dot and the length of path, name and extension is practically unrestricted. INSEL 8 can deal with any naming convention.

We are used to distinguish files by their extension: When we see a file with extension `.doc` we think "Aah, a Word document!," or a file with extension `.pdf` "Of course! This is an Acrobat file." Maybe one day people think "Yes, an INSEL file – what else!," when they see the extension `.insel`.

ASCII code

What makes up data files is that their content follows conventions, too. Like Enigma files are encoded in a specific "secret" code. One example for standardised code is the ASCII code: Every symbol – like letters and digits – is decoded by a series of seven bits, which can take either a value of zero, or a value of one – the dual system. Extended ASCII code uses eight bits, so that a larger set of symbols can be expressed. Eight bits are commonly called a byte. The trend goes to Unicode which uses sixteen bits, i. e., two bytes.

Hence, we can answer the question "What is a data file?" with the statement "A data file is nothing but a stream of bits. The meaning of this data stream needs to be known exactly – otherwise the data stream is completely useless."

`meteo82.dat`

The following lines show the head of a file that has been recorded in Oldenburg, North Germany, in the year 1982 – four years before INSEL 1.0 – named `meteo82.dat`.

```
 1 182 1   0.   0.   0.   0.   0.  4.5-40.  -40.0-40.0 265.  4.4
 1 182 2   0.   0.   0.   0.   0.  3.8-40.  -40.0-40.0 255.  4.3
 1 182 3   0.   0.   0.   0.   0.  3.4-40.  -40.0-40.0 255.  2.8
 1 182 4   0.   0.   0.   0.   0.  3.1-40.  -40.0-40.0 225.  2.3
 1 182 5   0.   0.   0.   0.   0.  3.0-40.  -40.0-40.0 225.  2.6
 1 182 6   0.   0.   0.   0.   0.  2.9-40.  -40.0-40.0 225.  3.0
 1 182 7   0.   0.   0.   0.   0.  3.2-40.  -40.0-40.0 225.  1.9
 1 182 8   0.   0.   0.   0.   0.  2.8-40.  -40.0-40.0 205.  2.5
 1 182 9   0.   0.   1.   0.   0.  2.3-40.  -40.0-40.0 205.  2.4
 1 18210  15.   6.   6.   3.   0.  2.7-40.  -40.0-40.0 195.  3.0
 1 18211  40.  28.  21.  19.   6.  3.4-40.  -40.0-40.0 195.  2.6
 1 18212  54.  25.  18.  15.   5.  3.8-40.  -40.0-40.0 195.  1.2
 1 18213  60.  24.  18.  15.   4.  4.3-40.  -40.0-40.0  85.  0.6
 1 18214  44.  17.  13.  10.   1.  3.9-40.  -40.0-40.0  75.  1.2
 1 18215  16.   9.   8.   5.   0.  3.9-40.  -40.0-40.0  85.  1.1
 1 18216   0.   3.   4.   1.   0.  4.0-40.  -40.0-40.0  85.  1.0
 1 18217   0.   0.   0.   0.   0.  4.1-40.  -40.0-40.0  85.  1.3
 1 18218   0.   0.   0.   0.   0.  4.1-40.  -40.0-40.0  85.  1.2
 1 18219   0.   0.   0.   0.   0.  4.3-40.  -40.0-40.0  65.  0.9
 1 18220   0.   0.   0.   0.   0.  4.2-40.  -40.0-40.0 165.  1.0
 1 18221   0.   0.   0.   0.   0.  4.4-40.  -40.0-40.0 175.  1.2
 1 18222   0.   0.   0.   0.   0.  4.9-40.  -40.0-40.0 195.  1.9
 1 18223   0.   0.   0.   0.   0.  5.2-40.  -40.0-40.0 225.  2.2
 1 18224   0.   0.   0.   0.   0.  5.3-40.  -40.0-40.0 225.  2.5
```

Without going into the details of this file for the moment some comments may be useful.

Records

- As you see, the file is organised in well-formatted "lines." Every line is usually called a record. So we'd better say: This example shows the first 24 records of the file `meteo82.dat`.
- Well-formatted lines means that all "columns" look alike, i. e., the decimal points are all in the same column, every line (more accurate: every record) has the same length. Take your time and count the number of columns. You should come to a value of 64 columns per record, including the blank or space characters.

Record length

Every column represents one alphanumeric symbol, represented by the corresponding symbol which can be a "0", a ".", or a space " ", for example. When each of the symbols is encoded in extended ASCII code, every column represents one byte. Hence we speak of bytes rather than columns. So we can conclude that when we want to describe the file we say: The file `meteo82.dat` is formatted and has a record length of 64 bytes.

Records end with a line break – otherwise we would see only one long, long line. Unfortunately, different operating systems use different conventions for line separators. Windows uses two bytes, i. e., a CR (carriage return) and LF (line feed), Mac OS only a CR, Linux only an LF. The line separator is usually not considered in the value for the record length. Again and again these different conventions are a source of trouble poor programmers have to live with. So again, be careful when you work with files in programming environments!

- Record 12 starts with the bytes `_1_18212__54` (for more clearness, we have replaced the invisible space characters by an underscore _). Does it mean that there is a number 18212 encoded in the file? Of course not. We humans conclude immediately that the first two bytes _1 stand for day one of the data recording, the next two bytes _1 stand for the month January, the next two bytes 82 are an abbreviation for the year 1982, the next two bytes 12 stand for the hour, and so on.

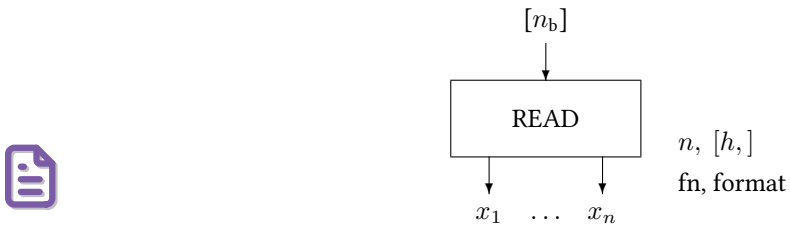
Fortran format

Computers don't conclude. They need to be told. This means that it is necessary to provide a "key" to any routine which shall interpret data files. Such a key is usually called a format. There are many conventions in computing that are used as formats. In INSEL the Fortran format conventions are used in order to describe the "keys" to data files.

3.1 Reading data

Reading data from files is a pre-requisite for many simulation runs, meteorological boundary conditions are required in practically all renewable energy simulations, for instance. Let us assume that some weather station has sent us a file with hourly ambient temperature data for one year, the file being named `temperature.dat`.

One INSEL block which can read data files sequentially is the READ block.

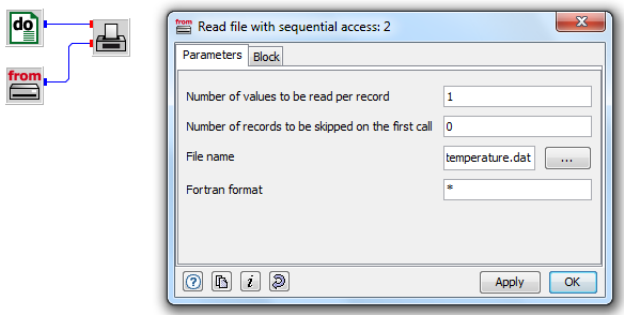


This block requires a parameter n for the number of values that is to be read per record, the file name fn, of course, and – very important – a parameter which describes the format of the file. In addition, there is an optional parameter h which allows us to start reading of the file not necessarily at the first record but at record number $h + 1$, i. e., if we set $h > 0$, the READ block skips reading the first h records. And there is an optional input n_b which can be connected to any output of a block to express the dependence of the READ block from this block.

`temperature.dat` The file `temperature.dat` is quite simple, it contains only one value per record. Here are the first ten records:

```
4.5
3.8
3.4
3.1
3.0
2.9
3.2
2.8
2.3
2.7
```

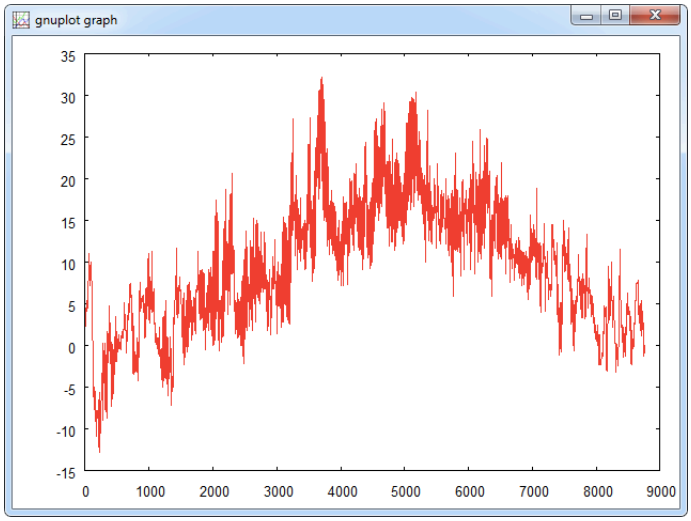
Star format In this case and in cases where all data in a record are numbers and separated by a blank character (space) the star format is very easy to apply to reading such data files. In order to read the file `temperature.dat` it is easiest to use this star format like we did in `examples\blocks\inputOutput\read.vseit`:



We can identify all the above discussed parameters: The file name `temperature.dat`, the Fortran format * (both are string parameters which are entered without enclosing quotes), and the parameter $n = 1$ for the number of outputs and the skip parameter h, set to the default value zero here.

Well, one year has 8760 hours (if it is not a leap year), so we use a DO block which counts from 1 to 8760, and a PLOT block because we would like to see the time series. Please observe again, that the Standard block READ must not necessarily be connected to the Timer. If this disturbs you, you can add a data input terminal to the READ block and connect the DO block output – it makes no difference in this case, but perhaps it would make the model structure clearer.

When you run the application the graph with the temperature time series will show up.



Current directory

You may wonder, how INSEL found the file `temperature.dat` although no path information is given in the file name. By default, INSEL searches for files in the directory of the model file. Of course, it is possible to include the full path to the file in the READ

object. Either slashes or backslashes can be used, like `c:\myData\temperature.dat` or `c:/myData/temperature.dat`, for instance. The total length of the string is restricted to 1024 bytes – all string parameters in INSEL 8 are restricted to this length.

There are no problems to be expected when a file contains more than one value per record like a standard file, provided by the Fraunhofer Institute for Solar Energy Systems ISE in Freiburg. The file `data\weather\iseyear.dat` provides data in 15 minutes resolution of some meteorological parameters for the location of Freiburg im Breisgau, which is in the very south of Germany, close to the Swiss border. For this example we use only the July fraction of the file saved under `data\weather\iseyear7.dat`. The first records of this file are shown here:

`iseyear7.dat`

```
7  1  0  7 30  17.5  95    0    0    0
7  1  0 22 30  17.3  94    0    0    0
7  1  0 37 30  17.1  92    0    0    0
7  1  0 52 30  16.8  91    0    0    0
7  1  1  7 30  16.6  90    0    0    0
7  1  1 22 30  16.4  89    0    0    0
7  1  1 37 30  16.2  88    0    0    0
7  1  1 52 30  16.0  87    0    0    0
7  1  2  7 30  15.8  86    0    0    0
7  1  2 22 30  15.8  84    0    0    0
7  1  2 37 30  15.7  83    0    0    0
7  1  2 52 30  15.6  82    0    0    0
7  1  3  7 30  15.6  81    0    0    0
7  1  3 22 30  15.4  80    0    0    0
7  1  3 37 30  15.3  79    0    0    0
7  1  3 52 30  15.3  78    0    0    0
7  1  4  7 30  15.3  76    0    0    0
7  1  4 22 30  15.0  75    0    0    0
7  1  4 37 30  14.7  74    5    0    5
7  1  4 52 30  14.7  73   14    0   14
7  1  5  7 30  14.6  72   31   52   27
7  1  5 22 30  14.5  71   57  160   40
7  1  5 37 30  14.7  69   83  228   50
7  1  5 52 30  15.1  68  116  301   61
7  1  6  7 30  15.6  67  150  366   70
7  1  6 22 30  16.2  66  187  420   79
7  1  6 37 30  16.9  65  226  466   88
7  1  6 52 30  17.4  63  267  507   91
7  1  7  7 30  18.1  60  309  543   99
7  1  7 22 30  19.1  57  351  580  103
7  1  7 37 30  20.2  53  395  620  107
7  1  7 52 30  21.5  52  435  643  112
7  1  8  7 30  21.2  52  473  658  117
7  1  8 22 30  21.7  54  516  683  122
7  1  8 37 30  21.6  55  557  700  129
7  1  8 52 30  22.0  55  592  709  134
7  1  9  7 30  21.6  55  629  713  145
7  1  9 22 30  21.5  55  666  724  153
7  1  9 37 30  22.1  55  701  736  158
```

```
 7  1  9 52 30  23.0  55  727  740  162
 7  1 10  7 30  23.3  52  758  751  166
 7  1 10 22 30  23.1  53  786  755  174
 7  1 10 37 30  23.0  52  811  758  180
 7  1 10 52 30  24.0  51  834  764  184
 7  1 11  7 30  24.5  49  851  762  191
 7  1 11 22 30  25.0  47  869  764  197
 7  1 11 37 30  25.0  45  882  763  202
 7  1 11 52 30  25.6  45  887  751  213
 7  1 12  7 30  25.7  45  894  748  217
 7  1 12 22 30  25.7  43  857  699  223
 7  1 12 37 30  26.0  41  906  768  209
 7  1 12 52 30  26.4  40  902  757  217
```

Interpretation

All data in the records are consequently separated by blanks so that we can use the star format to read this file. The file records contain

1 Month
2 Day
3 Hour
4 Minute
5 Second
6 Ambient temperature / °C
7 Relative humidity / %
8 Global horizontal irradiance / W m^{-2}
9 Direct normal irradiance / W m^{-2}
10 Diffuse horizontal irradiance / W m^{-2}

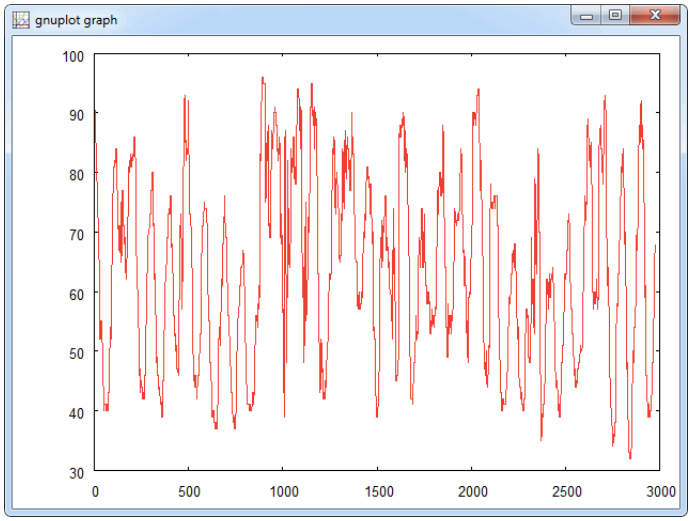
Exercise 3.1

The record length is 39 bytes. When you want to read the file, open a new VSEit network via *File* > *New* – or open the previously used file for the `temperature.dat` data – and save it under a new name, like `iseyear.vseit`, for example. The READ block so far has only one output, but we can add nine more via the block pane.

Now that your READ block has ten outputs you can give them understandable names, like M for month, d for day etc, by a double click on the respective ports.

Finally, choose some channels of your interest and analyze the file by plotting some time series portions from the file.

This graph shows the relative humidity, for example:



DWD data

Since many years monthly mean values of global radiation data are recorded by the DWD (Deutscher Wetterdienst – German Weather Service). We provide them for INSEL users in files named `dwdyyyy.dat` – `yyyy` is a place holder for the year 2011, for example. These lines show a part of the first records of file `dwd2011.dat`:

`dwd2011.dat`

```
Aachen          20   34   97  139  180  155
Augsburg        30   45  105  161  191  151
Berlin          18   41   92  132  185  181
Bonn            20   37   98  141  181  157
Braunschweig    19   34   90  140  174  177
Bremen          18   33   84  139  164  155
```

The file is well formatted in the above discussed sense, but – as a novelty – it does not only contain numerical data but also alphanumerical data, the name of the locations in this case. For such files, the Fortran star format can no longer be used.

Let us make a short excursion to some general Fortran format conventions.

3.1.1 Fortran format conventions

Edit descriptors

From the many so-called edit descriptors that exist in Fortran like `I`, `B`, `O`, `Z`, `F`, `E`, `EN`, `ES`, `D`, `G`, `A`, and `X` – to mention a few – INSEL makes use only of `F`, `E`, and `X`.

On the one hand this is a big advantage, because it is not necessary to read through pages and pages to understand all possible edit descriptors and their use. On the other hand this is a restriction, of course. But, you will see that almost all practical cases can be covered and in the (seldom) case that one of the other edit descriptors is required, the

advanced INSEL programmer can write and include his own code in INSEL to handle these cases.

Ergo, F, E, and X. What is their meaning? At first, F stands for floating point format and is used for real editing without exponents, E stands for exponential format, and X is used for positional editing.

Let us look at the definitions taken from the Microsoft Fortran documentation:

Syntax: Fw.d

The F edit descriptor tells Fortran to treat a number as a simple decimal floating-point value. On output, the I/O list item associated with an F edit descriptor must be a single- or double-precision real or complex number, otherwise a run-time error occurs. On input, the number entered may have any real or complex form as its value is within the range of the associated variable. The field is w characters wide, with a fractional part which is d decimal digits wide.

Syntax: Ew.d

An E edit descriptor means that there is an exponent in the syntax of the value. The I/O list item associated with the E edit descriptor for an output item must be a single- or double-precision real or complex number. A number input to a variable described with an E edit escriptor can have any real or complex form, as long as its value is within the range of the associated variable. The field is w characters wide. The input field for the E edit descriptor is identical to that described by an F edit decriptor with the same w and d.

Syntax: nX

The nX edit descriptor advances the file position by n characters. If n is absent, the X edit descriptor defaults to 1X.

So far the Microsoft text.

We are currently interested in the input cases, i. e., reading of files. Starting with Fw.d we have learnt that we can read floating-point numbers with a width of w bytes (including the decimal point by the way) and d bytes following the decimal point. Let's use the file meteo82.dat as an example. Recall the first four records of the file:

```
|...+...1...+...2...+...3...+...4...+...5...+...6...+...7.]
 1 182 1   0.   0.   0.   0.   0.  4.5-40.  -40.0-40.0 265.  4.4
 1 182 2   0.   0.   0.   0.   0.  3.8-40.  -40.0-40.0 255.  4.3
 1 182 3   0.   0.   0.   0.   0.  3.4-40.  -40.0-40.0 255.  2.8
 1 182 4   0.   0.   0.   0.   0.  3.1-40.  -40.0-40.0 225.  2.3
```

Here, in addition to the data a ruler is shown, which makes counting a bit easier.

We want the first eight bytes _1_182_1 (the underscore representing the invisible space again) of the first record to be read in as 1 for the day, 1 for the month, 82 for the year (1982), and 1 for the hour.

Floating-point numbers in INSEL are described by the F edit descriptor. Hence, in order to read the first two bytes as 1 we need an Fw.d format where the number of bytes w is equal to two and since we have no decimals after the non-existing decimal point d must be equal to zero. So F2.0 must be used – four times to read day, month, year, and hour, so that we can write F2.0,F2.0,F2.0,F2.0.

The edit descriptor `F` is a so-called repeatable edit descriptor, which means that if – like in our case – a specific format appears identically several times, a repeat factor can preceed the edit descriptor. This means writing `F2.0,F2.0,F2.0,F2.0` is equivalent to writing `4F2.0`.

The next column contains `___0`. So . . ., five bytes, no decimal fraction, `F5.0`. The next four values look alike, hence in total we have `5F5.0` – do you agree?

Then comes `__4.5-40.` – "What meaning of this?" as Peter Sellers said in the funny movie Murder by Death. In this special case the convention used in file `meteo82.dat` is, that whenever there is a `-40` in the file it means "lack of data." So we can guess that the `__4.5` is a datum and the concatenated "`-40.` is a datum which indicates "lack of data." This leads to an `F5.1` followed by an `F4.0` here.

The next two data seem to be missing too, so we interpret the bytes `__-40.0` as F7.1 and `-40.0` as F5.1. At the end of the record we see `_265.` followed by `__4.4` and have now understood that the edit descriptors are `F5.0` and `F5.1`.

To sum it up, the sequence of edit descriptors for the formatted file `meteo82.dat` is `4F2.0,5F5.0,F5.1,F4.0,F7.1,F5.1,F5.0,F5.1`, all separated by a comma.

Cross check

As we have already seen `meteo82.dat` has a record length of 64 bytes. We can cross-check this value with the sequence of edit descriptors: 4 times 2 bytes gives 8, plus 5 times 5 bytes gives 33, plus 5 bytes, gives 38, plus . . . gives 64. Okay?

Parentheses

In Fortran, format strings have to be parenthesised, i. e., the final Fortran format string to read all data in a record of file `meteo82.dat` is

`(4F2.0,5F5.0,F5.1,F4.0,F7.1,F5.1,F5.0,F5.1)`

Big numbers

When numbers get too big, the representation like 123 000 000 000 is no longer practical. In Fortran we can use the exponential representation for such cases. The number then would be shown as `0.1230E+12`, which reads as 0.1230×10^{12}. In order to describe this with the `E` edit descriptor we first have to count the number of bytes of `0.1230E+12`, which is equal to 10, including the decimal point, the `E` and the plus sign. The number of decimals is 4 bytes following the decimal point, so that the format is `E10.4`.

The `nX` edit descriptor is used to ignore `n` bytes during the reading of a record. In case of the above mentioned file `dwd2011.dat` we are not interested in a numerical evaluation of the bytes which contain the name of a location.

In printed form it is difficult to exactly count the number of bytes used for the location name due to the many spaces. In `dwdyyyy.dat` in total 20 bytes (including all spaces) are used for the location names. This means, when we want to read `dwdyyyy.dat` files we would like to always skip the first 20 bytes of each record. Hence, `n` is equal to twenty and we write `20X`.

In the files which contain data of complete years `12F6.0` values are saved for 12 monthly

means of the global radiation, followed by one `F7.0` value for the sum of the 12 months radiation values and an `F5.0` value which contains the percentaged deviation of the annual value from the long-term mean value.

When we sum up the number of bytes in the edit descriptors we find $20 + 72 + 7 + 5 = 104$ bytes.

This ends our short excursion to the Fortran format conventions which are important for INSEL programmers.

You should now apply your newly gained knowledge for an analysis of the two example data files `meteo82.dat` and `dwd2010.dat`. Before you can really start, some more information about the file contents is necessary.

Contents of `meteo82.dat`

Let us start with the file `meteo82.dat`. It contains the following data:

1 Day
2 Month
3 Year
4 Hour (1-24)
5 Global irradiance horizontal / $\mathrm{W\,m^{-2}}$
6 Diffuse irradiance horizontal / $\mathrm{W\,m^{-2}}$
7 Global irradiance tilt angle 70 degrees, facing South / $\mathrm{W\,m^{-2}}$
8 Global irradiance tilt angle 70 degrees, facing South-East / $\mathrm{W\,m^{-2}}$
9 Global irradiance tilt angle 70 degrees, facing South-West / $\mathrm{W\,m^{-2}}$
10 Ambient temperature / °C
11 Relative humidity / %
12 Air pressure / hPa
13 Precipitation / mm
14 Wind direction / degrees from north via east, south, etc.
15 Wind speed / $\mathrm{m\,s^{-1}}$

Since three of the data columns are completely lacking (all `-40`s) we can skip these columns.

Exercise 3.2 Can you construct the Fortran format when these bytes are skipped by `nX`?

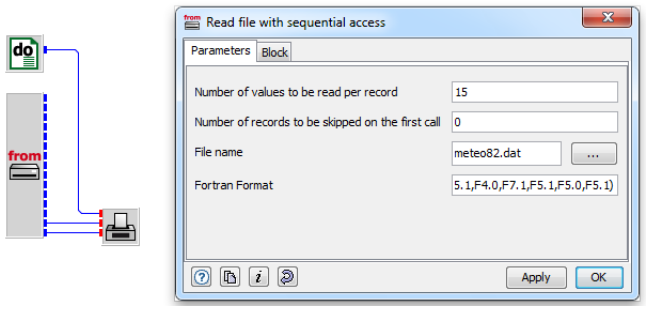
Example `meteo82.dat`

Now you have all the necessary information and can plot and analyse the time series saved in `meteo82.dat`.

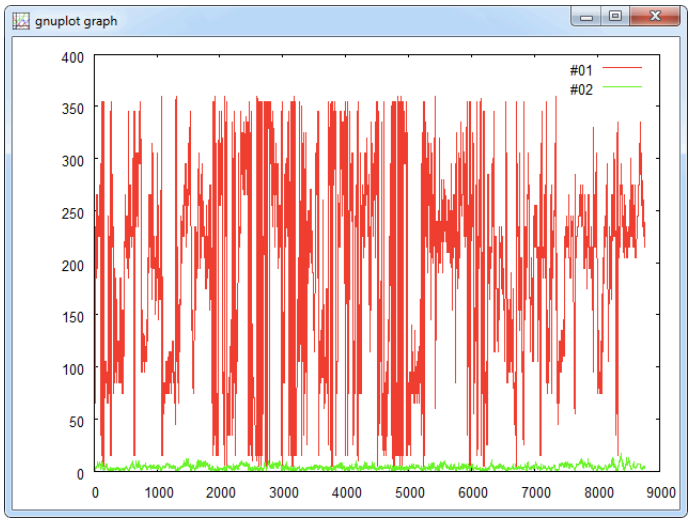
As usual, we provide example solutions. But you gain most from this Tutorial when you try and solve the problems on your own before you compare your solutions with ours.

Solution

We have constructed a READ block entity which fits the needs of `meteo82.dat` and plot the time series of wind direction and wind speed in one diagram.



Here comes the graph:



Rather than simply plotting the wind data it is much more interesting to analyse the radiation data on the different orientations, but we leave this task for you.

Example `dwd2002.dat`

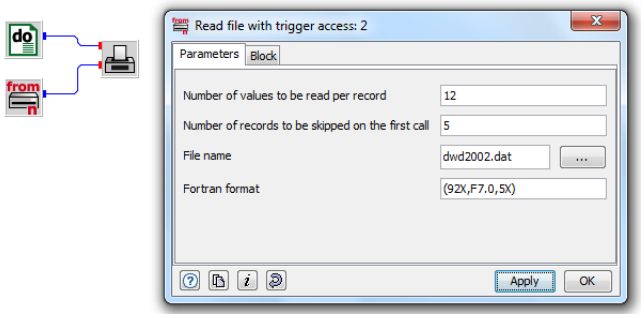
As another exercise use the file `dwd2002.dat` to compare the radiation distribution over Germany at the different locations available in `dwd2002.dat`. When you open the file – which resides in the `data\weather` directory – with a text editor you find out that `dwd2002.dat` contains data for 62 locations.

In the Fortran format conventions section we have seen that each record of the file contains a location name (20 bytes) followed by 12 monthly radiation values (`12F6.0`, in the unit kWh/m^2), the annual sum (`F7.0`, also in the unit kWh/m^2), and the deviation to long-term measurements (`F5.0`, in percent).

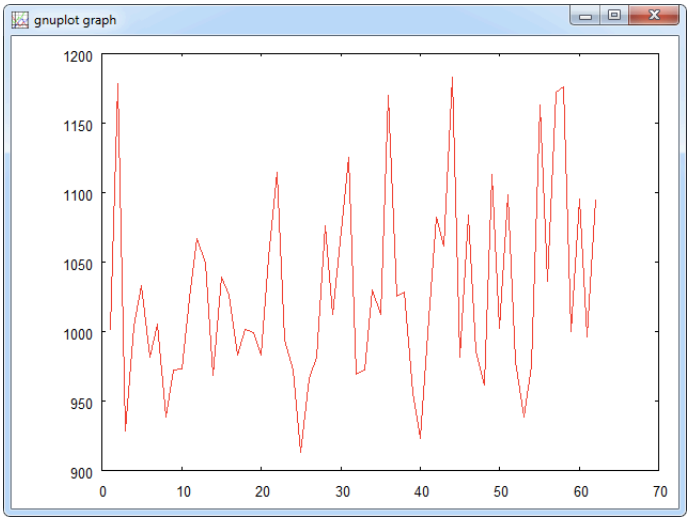
Exercise 3.3 Plot the annual radiation sums for all 62 locations.

Solution Since we are only interested in the annual radiation sums, we can skip to read all other

information in the file. Hence, we can simply use the format string (`92X,F7.0,5X`).



This is the output:



The locations in the file are ordered alphabetically, so the sequence in the plot does not make much sense. But we can observe, that the level of global radiation on a horizontal surface in Germany is in the range between 900 (the pitiable people in North-Germany) and 1200 kWh/m^2 in the South.

3.1.2 The READN block

When you look at the structure of the data in file `dwd2002.dat` and our only approach to file reading – sequential access – so far, you may find yourself confronted with the question "How can I use the READ block to access the monthly mean values saved in the file one after the other, i. e., plot the time series of monthly radiation data for a certain location?"

Exercise 3.4 Think about a solution for a few moments, please!

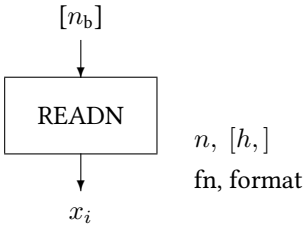
The problem is: One READ block execution reads one complete data record at a time. If we want to access all twelve data for a location, we can use a READ block with a Fortran format which reads all 12 values `(20X,12F6.0,12X)` with the consequence that we have all twelve values on output of the READ block at the same time. When this is what you want – no problem.

But if you want to plot the data for example, the PLOT block would require twelve inputs (all connected to the outputs of the READ block) plus one for the x-coordinate. What x-coordinate and what kind of a plot would this be?

So, what we really want is not to read a complete record in one step but only one datum like the radiation for January, plot it, read the next datum like the radiation for February and plot it and so on. It is obvious that the READ block as it is designed cannot solve this problem.

In such cases the READN block is one way out.[1] The READN block – like the READ block – reads one complete record as specified in the Fortran format parameter but outputs only one value at a time – it triggers the output values. The READN block expects a parameter which says how many calls of the block it has to wait until a next physical read access is to be performed on the file. The layout of the block is shown in the following graph:





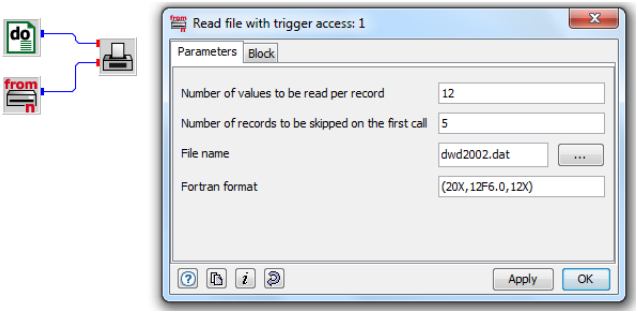
The layout of the block is exactly identical to the READ block but – no matter what the format parameter is – it outputs only one value per call.

Let us use the block for a plot of monthly mean values of global radiation for the location of Stuttgart on the basis of file `dwd2002.dat`. By opening the file with a text editor we find out that Stuttgart is record number 55. So, the READN block should skip the first 54 records. The Fortran format for reading is `(20X,12F6.0,12X)`. The file name is clear, so we can write the application.

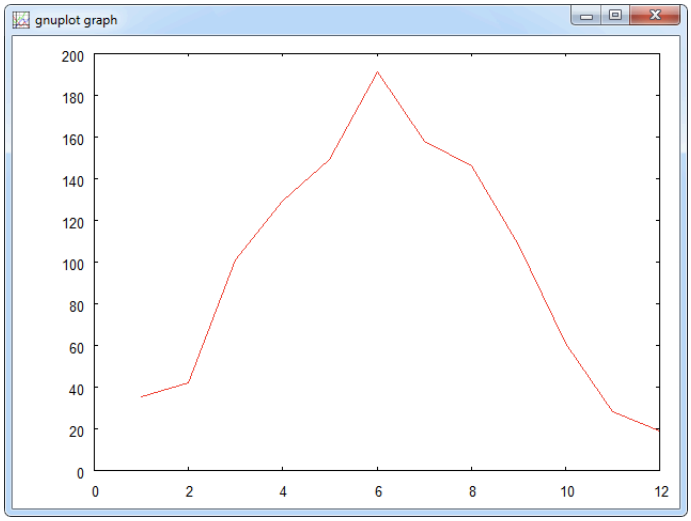
Exercise 3.5 Do it, now!

Solution The solution is

[1] Another possibility to solve this problem could be to use the multiplexer block MPLEX – see Block Reference Manual for details. But when the number of data per record gets large, the method is rather inconvenient since all outputs have to be connected.



and the time series plot is



3.1.3 The READD block

A third block, named READD for reading data files is available in INSEL which allows *direct access* to data files. In contrast to sequential access, where we can start reading a file at a given record and then the read operation returns the files records sequentially, i. e., one after the other, in direct access mode we can read records in any arbitrary order. In order to do that we have to deliver the record number of the record we want to access. How will the operation system perform the reading when we, for example, want to read the tenth record of a file?

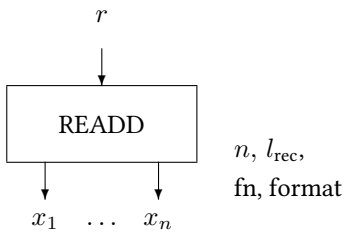
Well, as we have heard at the very beginning of this Module a file is nothing but a stream of bytes. Logically we have identified records as logical lines in the file. For direct access the operating system takes the record length – let us assume a record length of 80 bytes – of the file, multiplies it with the number of records to skip – nine in our example

– and then knows the displacement from the start of the file to the position where we want to start reading. In our example reading the tenth record means that $80 \times 9 = 720$ bytes must be skipped (plus the line separator bytes).

This method works only if all records have exactly the same length – otherwise the calculation would lead to some arbitrary byte in the file and makes an interpretation of the data stream impossible.

This is the design of the READD block:





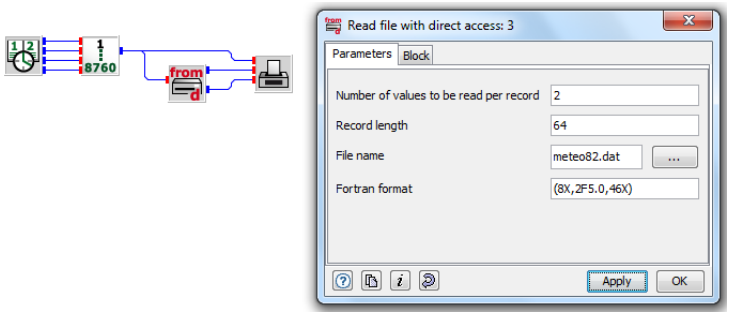
The parameters should be self explaining by now.

Exercise 3.6 As an example for direct access read and plot the time series of global and diffuse radiation on a horizontal surface for the month July as stored in file `meteo82.dat`.

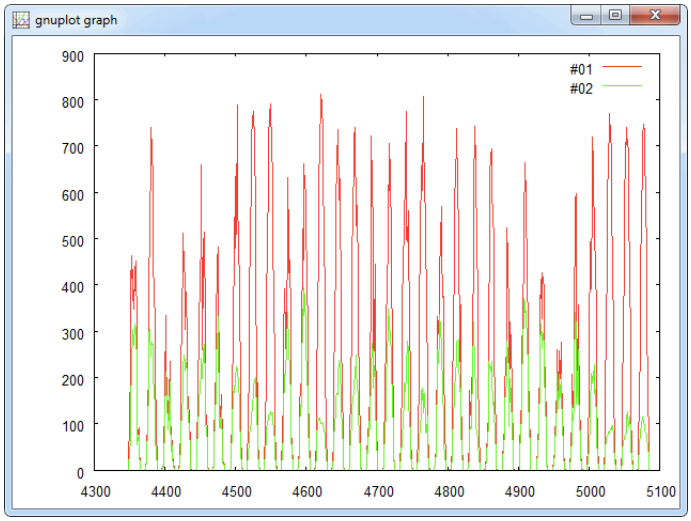
Solution 1 Yes, it is not really necessary to use the READD block. We can also solve the problem by opening the file `meteo82.dat`, find the record number where the first of July starts (record number 4345 since 1982 was not a leap year), quickly calculate 31 days times 24 hours gives 744 records, use a DO block which counts from one to 744, use the READ block and set the skip parameter to 4344, plot the two curves and ready.

But how would you use the READD block? Continue reading when you know the solution.

Solution 2 Use a CLOCK block and set the parameters from 01.07. any year 00:00:00 to 31.07. same year 24:00:00 in steps of one hour. Remember that 1982 was not a leap year. The hour of year block HOY can be used to convert the Gregorian calendar date into the hour of the year – this value is exactly the record number that we need, connect it to the READD block's input, and plot the curves. Now it is very easy to quickly change the parameters so that you can access the December data, for instance.



This is the plot of the radiation data:



3.1.4 File name qualifiers

File names in INSEL can be varied during a simulation run, when file name qualifiers (FNQs) are used in file name parameter strings. If the first character of a file name (without path) is a # character the file name is considered variable and is parsed by INSEL. Up to four qualifiers can be used as place holders for digits. The qualifiers have the format `%nX`, where `n` is a positive digit in the range of 0 to 9 and `X` is a character out of `YMDh` – reminding on their most frequent use of date and time information in data file names.

The numerical values for the `n` digits must be provided as block inputs. `Y` is used with the first input, `M` with the second, and so forth.

Example The file name `#myName%4Y.dat` results in `myName2011.dat`, assuming that the first input of the block containing the file name has a value of 2011.

Adding a second input which contains values for the months of a simulation, a qualified file name would be `#myName%4Yplus%2M.dat`. For a simulation running over a complete year the generated file names would be `myName2011plus01.dat`, `myName2011plus02.dat`, ... `myName2011plus11.dat`, `myName2011plus12.dat`.

Suppress leading zeros

The months from January to September are interpreted with leading zeros when the qualifier `#myName%4Yplus%2M.dat` is used. Leading zeros are suppressed when the qualifier `#myName%4Yplus%0M.dat` is used.

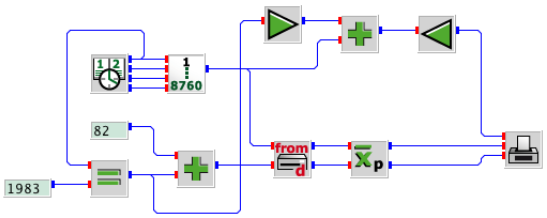
When a file name contains path information, the # sign is interpreted as qualifier only when it is the first character of the file name – not the path name. For example, running a CLOCK block over two years in steps of one month and connecting the first three outputs of the CLOCK block as inputs to an IO-block with the qualified file name `C:\Path\#prefix_%4Y_%0M_%2d_appendix.dat` results in something similar to

```
C:\Path\prefix_2000_1_01_appendix.dat
C:\Path\prefix_2000_2_01_appendix.dat
...
C:\Path\prefix_2000_9_01_appendix.dat
C:\Path\prefix_2000_10_01_appendix.dat
C:\Path\prefix_2000_11_01_appendix.dat
C:\Path\prefix_2000_12_01_appendix.dat
C:\Path\prefix_2001_1_01_appendix.dat
...
C:\Path\prefix_2001_12_01_appendix.dat
```

The file name parameter `#C:\Path\prefix_%4Y_%0M_%2d_appendix.dat` would not achieve the desired result but defines a constant file name `#C:\Path\prefix_%4Y_%0M_%2d_appendix.dat`.

`readd_8283.vseit`

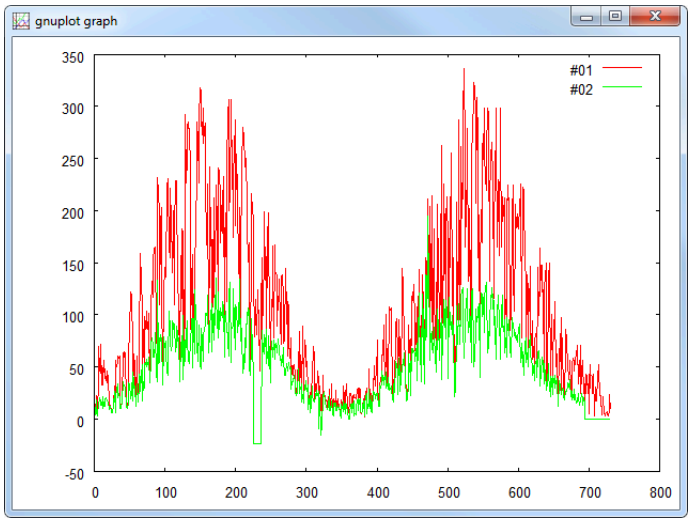
Assume, we want to read the data files `meteo82.dat` and `meteo83.dat` in a single simulation run. A file name qualifier which could be used for this purpose is `#meteo%2Y.dat`. As input to a READ or a READD block the two values 82 and 83 are required successively. One way to solve this problem is shown in the following model:



The CLOCK block varies date and time from 01.01.1982 00:00 to 31.12.1983 24:00 in steps of one hour. A CONST block with parameter 82 is used. As long as the CLOCK block runs through the year 1982 the logical result of the EQ block is false, i. e., 0. When the CLOCK block switches to the year 1983 the EQ block outputs the value 1. This value added to the CONST 82 gives the desired value 83.

Another solution could use a DO block with its output connected to the CLOCK block and the READ or READD block. The parameters of the DO block could be 82, 83, and 1. In this case the CLOCK block should run through any non-leap year, 2011, for instance.

We have used a AVEP block to calculate the daily means of the global and diffuse radiation data. Please observe, how we have used the EQ block to create an x-axis signal for two times 365 days. This is our result:



`readn_DWD.vseit`

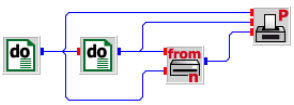
Another interesting application made possible though FNQs is the following: INSEL provides ground-measured DWD (German Weather Service) data for 62 German locations since the year 2000 in the `data\weather` directory. The files are organised as annual data files, named `dwd2000.dat`, `dwd2001.dat`, and so forth.

We have already used a READN block to read the Stuttgart data for one year earlier in this Module.

Exercise 3.7 Plot the monthly mean global radiation data for Stuttgart over the period from 2000 to 2010 with a single INSEL model.

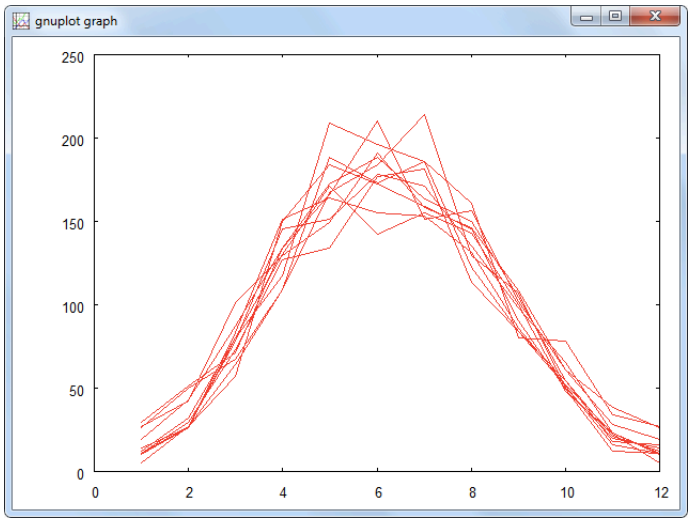
Solution Using file name qualifiers makes this task easy as pie:

`readn_fnqs.vseit`



We need two DO blocks, one for the variation of the year from 2000 to 2010 in steps of 1 and one for the variation of the month from 1 to 12 in steps of 1. The READN block

expects FNQs starting with input number 2. The qualified file name is `#dwd%4Y.dat` (plus path information) and, as usual, the PLOT block shows the result.



We are going to have one more example for the use of file name qualifiers at the end of the next section on writing data to files.

3.2 Writing data to files

With the WRITE block data can be written from an INSEL application to the computers hard disk, a USB flash memory disk, or any other media where you have write access.

There are basically two modes: You can

- Create a new file or overwrite an existing file
- Append data to a new or an already existing file



If the *Overwrite* mode is chosen, then after opening the file specified in the file name parameter the write pointer points to the beginning of the file and overwrites its content.

WARNING Be extremely careful with the option to overwrite files. When you choose this option,

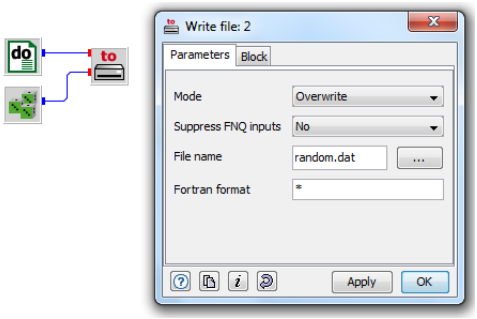
the block will definitively overwrite the file and the old file – if there was one – is definitively lost forever. There is no Undo. When it was a file which contained something very valuable before, now it is gone. So be careful. We deny any responsibility. So again, be careful!

If the *Append* option is chosen, the pointer is positioned at the end of the existing file before the first write operation.

If the *Generate error message* option is selected, the WRITE block will generate an error message if the file already exists and INSEL does not execute the simulation model.

As file name you can choose any name which is valid under the operating system version you are using. Of course, the file name can include path information. For example `c:\anyPath\myFile.dat` is a valid file name (assuming that the path `c:\anyPath` exists). If no path information is included in the file name INSEL writes the file into the current directory (usually the INSEL working directory `insel.work`).

`random.vseit` A simple application of the WRITE block is reformating of data files. `random.vseit` generates 100 Gauss distributed random numbers and writes the data to a file called `random.dat`.



`random.dat` The first ten records of the file look like this:

```
1.0000000      1.6216065
2.0000000     -0.39489648
3.0000000     -0.33821103
4.0000000      0.53852010
5.0000000     -0.42136794
6.0000000     -0.23904423
7.0000000      1.2835248
8.0000000      0.38314018
9.0000000      -1.4969951
10.000000      -1.0831203
```

Exercise 3.8 Please remember, that files like `random.dat` cannot be used for direct access with the READD block. Hence, write a small INSEL program which reformats the file to a file with format `(F12.7,1X,F5.1)`.

Solution Yes, you are right, we could have used a format string like `(F5.1,1X,F12.7)` in the `random.vseit` example already. But then you would have missed this one.

Three obvious blocks, DO, READ, WRITE and done. Please observe again, that the READ block does not need a connection to the DO block – because it is a Standard block and Standard blocks are always successors of the main Timer when no input is connected . . .

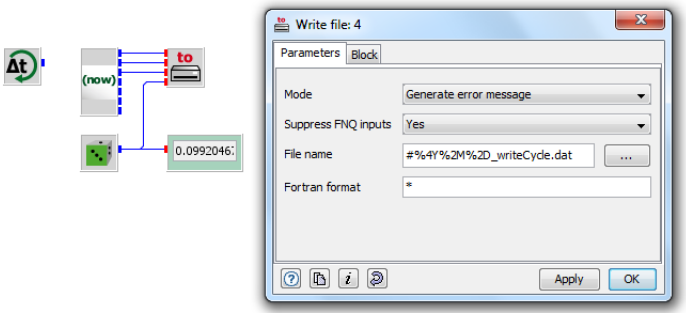


. . . and everything is well formatted now:

```
 1.0    1.6216065
 2.0   -0.3948965
 3.0   -0.3382110
 4.0    0.5385201
 5.0   -0.4213679
 6.0   -0.2390442
 7.0    1.2835248
 8.0    0.3831402
 9.0   -1.4969951
10.0   -1.0831203
```

3.2.1 Monitoring and simulation

INSEL can be used to monitor real-life systems, the most prominent ones being grid-connected PV generators like the Trade Fair Munich Generator, for example. Have a look at the following block diagram:



A CYCLE block is used to continuously run this INSEL model. The execution speed of the model – measured in real-time seconds – can be specified by the parameter of the CYCLE block. For every time step the NOW block returns the current date and time. Year, month, and day are used as file name qualifiers to write files with daily data of some monitored and/or simulated data – in this case only dummy random numbers.

Blocks from the palette's *GUI objects* category can be used to display any data over the course of the simulation run. Perspectives can be used to create complete GUIs

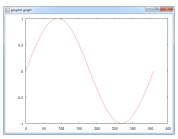
(graphical user interfaces) for the application. We will return to this topic in more detail in Module .

3.3 Plotting data

Another INSEL block of the data writing category – which you have used several times already – is the PLOT block. From the application point of view, x-y coordinates are connected to the block and the block generates a graphical output. Let us understand more deeply how the block operates.

At first, recall that every INSEL block works absolutely local. By this we mean that the block can perform only such operations, which depend on nothing but the actual values of the inputs, the parameters, and – in some block's cases – on the history.

Let us use the simple case of plotting the sine function as an example.





From the point of view of the PLOT block, on the first call the block receives an x-input zero and a y-input equal to zero, too. We know, that on the first call the first zero is an angle $\alpha = 0°$, and the second input is the $\sin(\alpha)$ – the PLOT block doesn't know anything about this.

The only point of interest from the PLOT block's point of view is that there is a data point (0,0) which I (the PLOT block) have to show as one of probably many data points in a graphical plot. What does the block do? It saves the data point in a data file and expects further actions – either it receives more data points, in which case the block will append them to the data file, or the instruction to display the complete graph.

An online plotter would show data points and their linear (or other) interpolation immediately. But the PLOT block is not an online plotter.

`insel.gpl` The data file always has the same name `insel.gpl` and is saved in the hidden-application-data directory. The instruction to display the graph comes from the inselEngine after the simulation run has been completed. The first ten records of file `insel.gpl` in this example are

```
0.0000000E+00  0.0000000E+00
0.1000000E+01  0.1745241E-01
0.2000000E+01  0.3489950E-01
0.3000000E+01  0.5233596E-01
0.4000000E+01  0.6975647E-01
0.5000000E+01  0.8715574E-01
0.6000000E+01  0.1045285E+00
0.7000000E+01  0.1218693E+00
0.8000000E+01  0.1391731E+00
```

```
 0.9000000E+01  0.1564345E+00
```

INSEL uses the maximum number of significant digits for Fortran four-byte REAL numbers (which is seven).

If you want to make further use of the file – maybe you like to post-process it with a presentation software of your choice – you can copy or rename the file to your needs. The only thing you need to document is what the meaning of the records, i. e., the x-coordinate and the y-coordinate(s) is.

The PLOT block, by default, generates a second data file `insel.gnu`, which contains some basic commands which enables Gnuplot to display the graph. In the case of our sine application the file looks like this.

```
set autoscale xy
set style data lines
set nolabel
plot "C:/Users/name/AppData/Roaming/doppelintegral/INSEL/tmp/insel.gpl" using 1:2 title ""
pause mouse
```

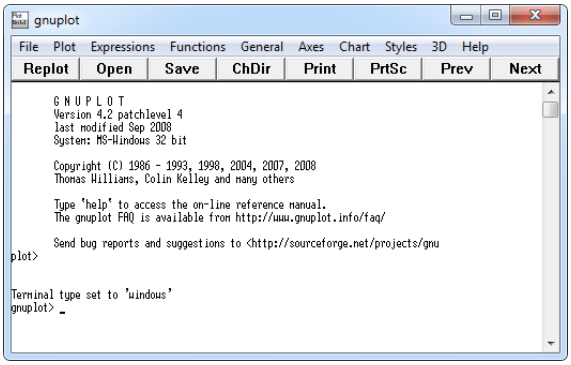
The first command `set autoscale xy` leaves it up the Gnuplot to find reasonable settings for range and increment of the x- and the y-axis. The next command `set data style lines` requests from Gnuplot to draw a connection line, i. e., a linear interpolation between the data points. `set nolabel` leaves the plot clean of any label names, and `plot "/path/insel.gpl" using 1:2 title ""` lets Gnuplot show the data plot based on the data in file `/path/insel.gpl` using the values in the first column as x-coordinate and the second as y-coordinate.

The value of `/path/` depends on the user's name and settings and is located on the lokal hard disk, by default. `title ""` suppresses any default legend of the plot, and finally `pause mouse` makes Gnuplot wait for a mouse click to close the window.

Such a default Gnuplot command file is always generated by the PLOT block when `insel.gnu` is given as PLOT block parameter. You can specify own Gnuplot command files for the PLOT block but this requires some knowledge about Gnuplot programming.

Interactive Gnuplot

One last hint to the PLOT block: You can start Gnuplot from the *Tools* menu or by a click on the icon in the tool bar. The Gnuplot window appears.



In the work area you see a prompt `gnuplot>` and a blinking cursor. Here you can enter Gnuplot commands. If, for example, you want to plot the last INSEL plot you made – this is file `insel.gpl` in the hidden application data directory – you can proceed as follows:

Type `pwd` at the gnuplot prompt (`pwd` is short for print working directory) and Gnuplot shows the actual directory name. You can use the change directory command `cd 'dirName`, where `dirName` stands for the target directory. Please notice the single quote in front of the directory name.

You can either specify a complete path – like `c:\myDirectory`, for example – or you can use relative directory names just like in a DOS box. Remember that for changing to a directory one level higher than the current directory the command is `ch ..` under DOS and `ch '..` under Gnuplot.

When the hidden application directory is the current directory you can use the command `load 'insel.gnu` (observe the quote again) and Gnuplot will display the last graph – just like INSEL when it performs these default steps for you automatically.

The difference is now, that after closing the graph window you can interactively use the menus and buttons of Gnuplot to make modifications to the plot. For example, if you want to add a label to the x-axis use the *Axis –X Label* menu item and enter the text for the label, skip the offset by simply clicking *OK* and then click the *Replot* button in Gnuplot's tool bar.

Many things should be self-explaining in the Gnuplot window. When you are interested in a deeper understanding of Gnuplot, the complete Gnuplot manual is available under the *Help* menu of Gnuplot. It is definitively worth to have a look, because Gnuplot is really powerful.

Summary

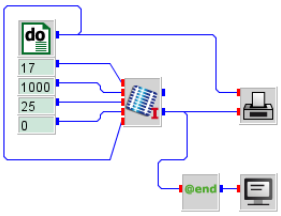
- Data files are streams of bytes which must be interpreted by encodings, like ASCII, for example.

- INSEL uses Fortran format conventions. You should know now how to work with the edit descriptors `F`, `E`, and `X`.
- Sequential access to data files is possible with the READ block. Optionally, we can start reading of formatted or unformatted (star format) files either from the first record or start reading the file at a well-defined offset.
- Direct and trigger access to read files is possible with the blocks READD and READN.
- The WRITE block can be used to write data to arbitrary data files.
- The PLOT block turned out as a block which writes two data files, i. e., `insel.gpl` and `insel.gnu`.

4 :: If blocks

A programming language is not a programming language if it does not provide at least one statement which enables the use of an if-then-else structure. In INSEL this structure element is represented by the concept of If blocks, or I-blocks, in short.

In Module we have already used an ATEND block as a first example for an I-block. Let us briefly recall its use. The block diagram which used the ATEND block was the following:



atendExample.vseit verschoenern

A DO block is used as timer which runs through one hour in steps of one second. For constant meteorological and operational conditions a PVI block calculates the warming up of a PV module from an initial temperature of 25 °C. The resulting temperature plot has been shown on page 25.

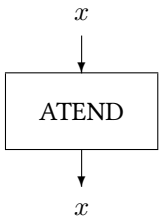
We saw from the graph that the module heats up to about 53 °C. What if we are only interested in the equilibrium temperature rather than the complete temperature profile? In this case we would like to let the module warm up, but display only the last, i. e., the equilibrium temperature value. This is a typical task for the ATEND block.

In the example, it uses the module temperature as input, whilst the output of the ATEND block – which is identical in value to its input – is connected to a SCREEN block. But the ATEND block ignores all input values until the simulation run is completed. Only then, the ATEND block lets the input signal pass.

Hence, from the point of view of the SCREEN block the SCREEN block is supplied by a value from the ATEND block only at the end of the simulation run, and thus displays only one value: the value at the end of the simulation run, which is the equilibrium temperature of the PV module in this case.

4.1 At end If blocks

Let us analyse the ATEND block in more detail.



@end

The block – like any other INSEL block – receives data depending on its input connection. But the ATEND block ignores all inputs until the end-of-run, i. e., until the condition whether the end of run is reached becomes true. Only in this case, the ATEND block lets the input signal pass through to its output and the inselEngine calls the blocks which are connected directly or indirectly to the ATEND block's output.

This is the typical behavior of an If block – it checks a specific condition. When the condition is true the blocks which make direct or indirect use of the I-block's output are executed, when the condition is not true the blocks which make direct or indirect use of the I-block's output are not executed.

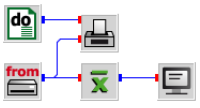
End of run

In case of the ATEND block the condition is the end of a simulation run. Other examples for blocks which use the end of simulation run condition are the blocks which calculate the average of an input signal over a complete run (block AVE), or cumulate an input signal over a complete run (block CUM), or find the absolute maximum (block MAXX) or minimum (block MINN) of a time series.

AVE block

We start with the average block AVE. In Module we had used the file `meteo82.dat` which contains hourly records of meteorological parameters for the location of Oldenburg in Germany for one year. One variable of the time series saved in this file is the global irradiance on a horizontal surface in W/m^2. How can we calculate its annual mean value?

The answer is straight forward: Use an average block, to be found under the *Mathematics Logics* category, connect it to the radiation output of a READ block which reads `meteo82.dat` and display the output of the AVE block with a SCREEN block. The block diagram is simple.

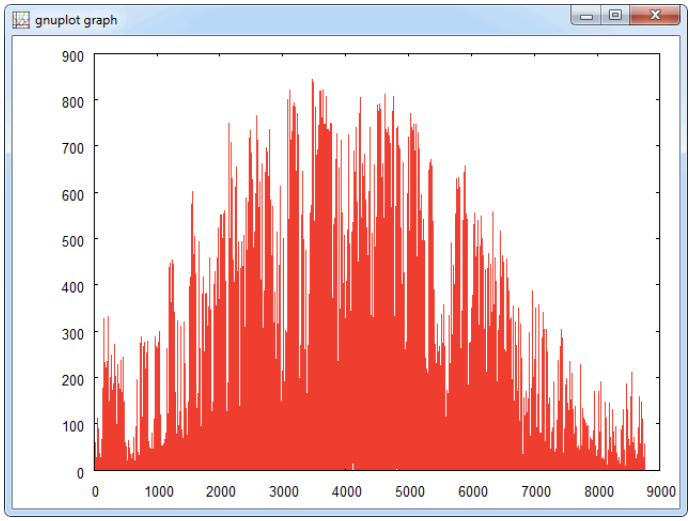


And the result is: The global irradiance on a horizontal surface in Oldenburg in the year 1982 has been 108.98 W/m^2 – did you find the same figure?

Please observe three details from our block diagram.

- We have added a PLOT block which displays a graph of the complete time series.

The resulting plot is useful for a plausibility check that we have really configured our READ block with the correct global irradiance data.



- Since we are interested in the global irradiance data on a horizontal surface only, we skip all other data of the input file by using the format `(8X,F5.0,51X)`. Hence, our READ block has only one output.
- Again, the READ block is not connected to the T-block DO, but executed in each of the 8760 time steps (since READ is a Standard block).

The unit of the hourly irradiance data – and hence of the annual average as well – is given in W/m^2. Physically spoken this is a power density, i. e., power in watt per area in square meter. There are some people in this world who seem to have slight problems with this kind of average calculation for solar irradiance, with somehow vague arguments like "But at night the Sun does not shine, so why shall I consider these hours in the calculation at all?" The answer is: The AVE block just calculates the global average $\bar{G}$ of the radiation time series $G(h), h = 1, \ldots 8760$ according to the standard definition of the average

$$\bar{G} = \frac{1}{8760} \sum_{h=1}^{8760} G(h)$$

Conversion of units

For those who prefer to think of global radiation as energy per square meter and time interval, it is easy to convert the annual mean value from W/m^2 to kWh m^{-2} a^{-1}. All we have to do is multiply $\bar{G}$ by the number of hours per year (which is 8760), and divide by one thousand for the conversion from Wh to kWh, i. e., multiply $\bar{G}$, given in W/m^2 by

8.76 to get $\bar{G}$ in kWh m^{-2} a^{-1}. The result is easily calculated:
$108.98 \times 8.76 = 954.66$ kWh per square meter and year.

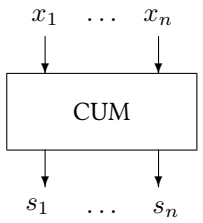
If you like to use INSEL for the calculation, connect a GAIN block with parameter 8.76 to the AVE block's output and display the output of the GAIN block rather than the output of the AVE block directly. This is a simple example for the conversion of units with INSEL. Please notice that instead of using a GAIN block, it would have also been possible to use a CONST block with parameter 8.76 and a MUL block which multiplies the AVE block's output with the CONST block's output. The result is exactly the same but using the GAIN block saves one INSEL block in the block diagram and is therefore preferred.

There are three more If blocks available which are very similar to the function of the AVE block:

CUM block

The CUM block calculates the cumulated sum of its input over a complete simulation run.

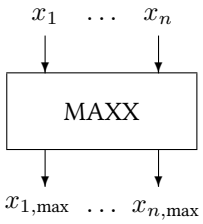




MAXX block

The MAXX block calculates the overall maximum of its input over a complete simulation run.

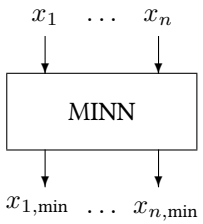




MINN block

The MINN block calculates the overall minimum of its input over a complete simulation run.



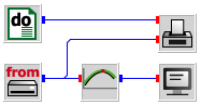


Exercise 4.1 Use the three blocks and apply them in order to calculate

- The cumulated value for the global irradiance on a horizontal surface in kWh/m^2
- The overall maximum value for the hourly global irradiance on a horizontal surface in W/m^2
- The overall maximum value for the ambient temperature in °C
- The overall minimum value for the ambient temperature in °C

as saved in file `meteo82.dat`.

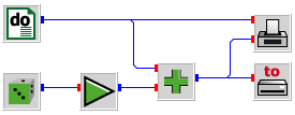
Solution 2



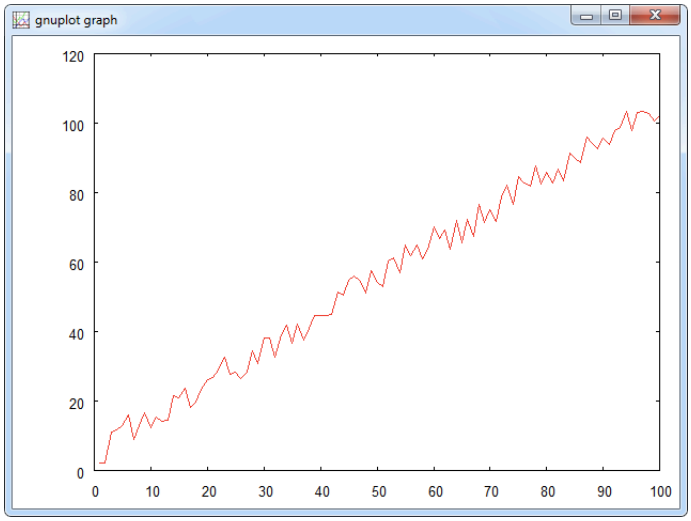
The maximum value of the hourly global irradiance on a horizontal surface is 843 W/m^2.

Fit blocks We turn our attention now to another set of If blocks which also use the condition end of simulation run and this is the set of fit blocks. What is a fit? A fit is a statistical method to approximate a set of data by an analytical equation of a given form. A very wide-spread and well-known fit uses the method called *linear regression*. In this case, the given (statistical) data set is approximated by a linear function. Before we discuss the FITLIN block let us create a data base for the function to be fitted.

`fitlin0.dat` Let a DO block deliver 100 steps and a RAN1 block generate one uniformly distributed random number for each step. When we multiply the random numbers by a factor ten with a GAIN block, for instance, and add the DO block's output and the output of the GAIN block, we defined a scattered variable which can serve as data base for the FITLIN block. We have saved the data in a file named `fitlin0.dat` in the `examples\tutorial` directory. It has been calculated with this block diagram, saved as `fitlin0.vseit` in the `examples\tutorial` directory as well:



The resulting data look at least a bit scattered and show an obvious trend.

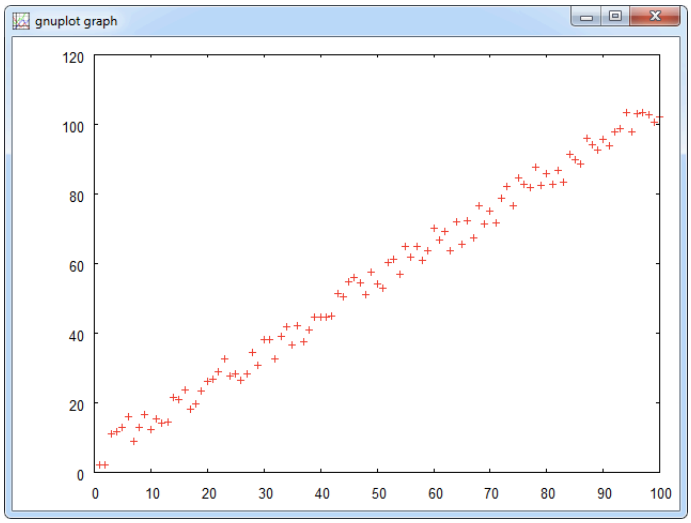


If you like, you can plot the data without the disturbing interpolation lines with Gnuplot.

Hint

Use Gnuplot in interactive mode, as briefly described in Module , page 65. Choose *Data Style Points* from Gnuplot's *Styles* menu, and click the *Replot* button.

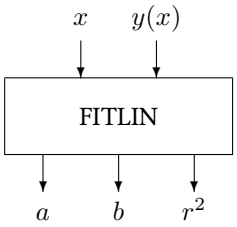
The result is this:



FITLIN block

Now we are ready to read the "statistical" data and perform a linear regression. The FITLIN block itself has the following layout:





Two inputs must be connected and fed with data: An independent variable x, and an x-dependent variable $y(x)$. The block requires no parameters. Outputs are the variables a, b, and r^2, where a and b are the best approximations to the equation

$$y(x) = a + bx$$

and r^2 is the regression parameter which describes the accuracy by which the equation $y = a + bx$ approximates the data. $r^2 = 0$ is the case of absolutely no correlation, $r^2 = 1$ stands for the case where all data points are either absolutely correlated or absolutely anti-correlated.

The block diagram is simple again.

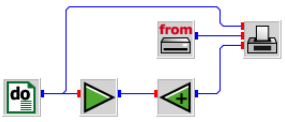


A READ block reads the scattered data from our file `fitlin0.dat`, just using the star format, for example. The FITLIN block finds the parameters a, b, and r^2. The SCREEN block displays the output with format (3F8.4). We get this result:

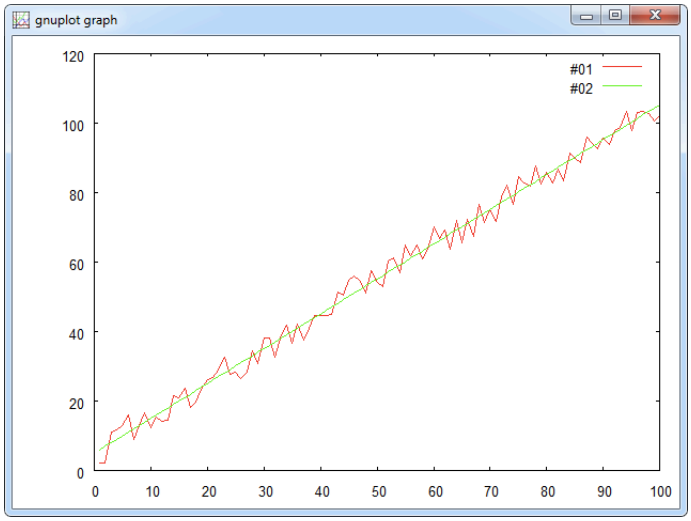
```
4.8319  1.0039  0.9903
```

Once the result is known ($a = 4.8319$ and $b = 1.0039$) we can plot the linear equation $y = a + bx$ and the scattered data in one diagram to see how good the fit is.

The block diagram which reads the scattered data requires only minor changes. A GAIN block is used to multiply the output of the DO block (the variable x) by the factor $b = 1.0039$, an OFFSET block with parameter $a = 4.8319$ and a SUM block builds the sum $y = a + bx$, which is displayed by the PLOT block.



This is the plot:



Standard fit routines

Some standard fit routines are available as blocks like block FITEXP, which fits data to the exponential function $y = a\exp(bx)$, block FITLN, which fits the logarithmic function $y = a + b\ln(x)$, and block FITPOW, which fits the power function $y = ax^b$.

There are some much more sophisticated fit blocks available in INSEL. For example, the PVFIT1 and PVFIT2 blocks are used to fit data which describe the performance of photovoltaic modules to equations known as the one-diode model and the two-diode model. For further details on these blocks please refer to the respective reference manuals.

An example for the PVFIT2 block will be presented in Module , page **??**.

4.2 If blocks with a parameter

AVEP block

During the discussion of the average block AVE we have seen a plot of the hourly time series of global radiation for Oldenburg, Germany. There was hardly something to distinguish, since the plot was basically a lot of red ink.

Annual radiation time series are much better visualized as series of daily data rather than hour by hour. An alternative would be a carpet plot – see block PLOTPMC for further details.

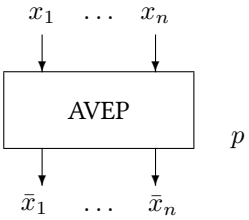
In order to calculate the daily means an average block would be useful which cumulates the radiation data over one day, i. e., 24 hours, divides the cumulated sum by 24, and outputs the result after every 24 hours. This is exactly what the AVEP block does – it

calculates an average over a number of steps as specified by a parameter p.

$$\bar{x} = \frac{1}{p}\sum_{i=1}^{p} x_i$$

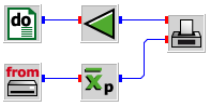
which is exactly the same definition as the formula used by the AVE block, the only difference being that p is a free block parameter.



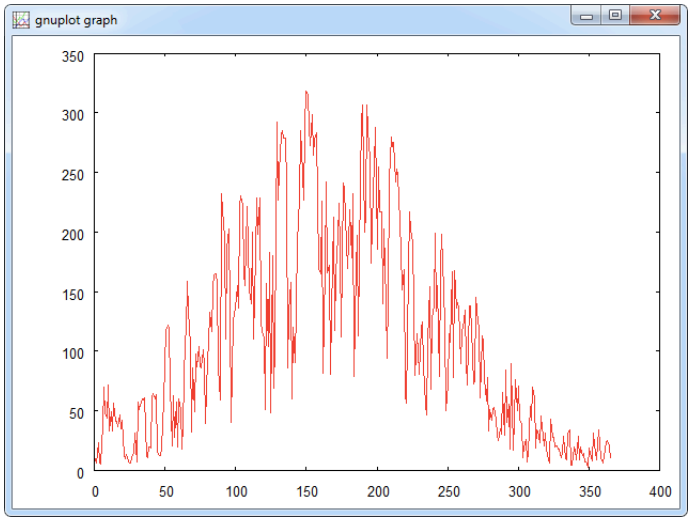


There are some very similar INSEL blocks named CUMP, MINNP, and MAXXP. You can probably guess what their functions are – check the Block Reference manual for details.

Let us construct a simple application for the AVEP block, and plot the time series of daily global irradiance data on a horizontal calculated from file `meteo82.dat`.



From a previous example we have simply replaced the AVE block with an AVEP block, used an ATT block for the division of the hours by 24 and plot the time series of daily data.



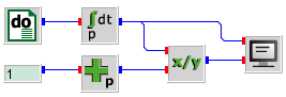
Please notice again, that the daily averages value are given in W/m^2. Since the time step of the data is one hour we can interpret the radiation data as Wh/m^2 as well. If you prefer to display the radiation data in $kWh\,m^{-2}\,d^{-1}$ you should multiply the values by 0.024. Since the maximum daily value is about 333 W/m^2 this corresponds nearly 8 $kWh\,m^{-2}\,d^{-1}$ in summer.

Hint Do not connect the outputs of the four different If blocks to one SCREEN block – INSEL will not accept this (try it) and display an error message that the SCREEN block depends on not enclosed If/Timer-blocks.

Bug or feature? This behavior has been newly introduced since version 6.0. Whether it is a feature or a bug is still not clear – most probably it must be considered as a bug.

The background is that different I-blocks can have different conditions. Hence, depending on the conditions some unwanted effects might occur if outputs of I-blocks with different conditions are brought together. But in cases like the one we are discussing, when the conditions are all the same – end of the simulation run – it should work. But it doesn't. The way out is to use four SCREEN blocks for the four outputs.

Workaround In case you wish to use an averaging block and a cumulation block and write the results to a data file, the above mentioned behavior does not allow this. However, a workaround to use the cumulation block and "simulate" the average block by SUMP block which cumulates constant 1 values with the appropriate parameter p and divide the cumulated signal by the output of the SUMP block:



Letting the DO block count from 1 to 10 and setting the parameters of CUMP and SUMP to 10 leads to the expected result:

```
55.000000       5.5000000
```

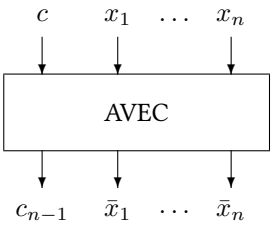
4.3 Conditional If blocks

What, if we want to calculate monthly means rather than daily? The complication is, that days always have 24 hours, but the number of days in a month is not constant. For example, January has 31 days, February 28, or – if it is a leap-year – in February the number of days is 29, March has 31 days and so forth.

AVEC block

A block which solves this problem conveniently is the AVEC block (average with condition). The layout of the block as follows.



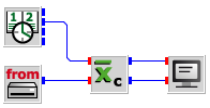


The block has two inputs: a condition input c and a signal input x_i. The idea of the block is to collect input data x_i as long as the condition input c remains constant. When the value of c changes, the block calculates the average over all x_i where c has been constant and outputs the average value. Let us look at an example first and then understand some more details about the block.

Monthly means

Assume, that we want to calculate the monthly mean ambient temperature values from the hourly data as stored in file `meteo82.dat`. Since the calculation of the average depends on the Gregorian calendar it is obvious that we use a CLOCK block as timer which runs through the hours of the year 1982.

For every time step of the CLOCK block INSEL reads one record from the file. The inputs to the AVEC block then are (i) the condition input month M as given by the CLOCK block, and (ii) the data input T_a from the READ block. The following block diagram does the job.



This is the result:

```
1.0000000     -0.59731185
2.0000000       1.3403274
3.0000000       4.9831991
4.0000000       7.1958332
5.0000000       11.912768
6.0000000       16.160418
7.0000000       18.441263
8.0000000       17.304436
9.0000000       15.336250
10.000000       10.365457
11.000000       6.6379166
12.000000       2.4293010
```

When you remember the file format of `meteo82.dat` as discussed in detail in Module 4, page 52 the ambient temperature is the tenth parameter of the file, so that we used the format `(33X,F5.1,26X)` and a READ block with one output to read the temperature time series.

A detail

Please observe that the SCREEN block uses the condition output of the AVEC block rather than the month output of the CLOCK block to display the monthly mean temperatures.

Why? Try, and figure out the reason by yourself for a moment.

Well, what happens? The CLOCK block starts with the first of January, zero hours, the READ block reads the corresponding data record, the AVEC block receives the data, and this sequence continues, continues . . . All the time the condition input of the AVEC block is equal to one, i. e., the AVEC block remains in data collection mode.

Then comes the last hour of January. The output of the CLOCK block is year 1982, month 1, day 31, hour 23 (not 24!). From the point of view of the AVEC block nothing special happens – the condition input is still equal to one, i. e., the block remains in data collection mode.

But then: in the next time step the CLOCK block changes its outputs to year 1982, month 2, day 1, hour 0 (not 1!). Now, from the point of view of the AVEC block, the condition input (month) has changed, i. e., the block has to perform some action.

The AVEC block calculates the average value, prepares the calculation of the average for the next condition (which is February, logically), outputs the monthly mean value for condition $c = 1$ (i. e., January) and request from the inselEngine to execute the successors – which is the SCREEN block only, in this case.

How shall the SCREEN block know that the value it gets is the January value? The output of the CLOCK in the actual time step says 2, i. e., February already. This is the reason why the AVEC block outputs the average value and the corresponding condition coordinate.

A second thought

Did you recognize that it is in fact a problem to display the last mean value?

The last time when the AVEC block is year 1982, month 12, day 31, hour 23. In this step, no change in the condition happens, and hence the AVEC block cannot know that the simulation run is finished.

Destructor call

For such cases INSEL has a mechanism that all blocks receive at least one additional so-called destructor call. From the AVEC block's point of view this implies a definitive condition change. This is the last chance for the AVEC block to calculate the last average value and put it on its output.

The same mechanism applies to the PLOT block. Maybe now you can have a better understanding of the details about the PLOT block discussed in Module , page 64.

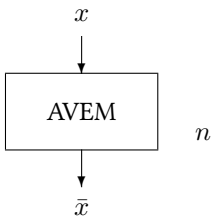
More conditional If blocks

There are some more I-blocks which use a condition input, like CUMC, MINNC, MAXXC. Please check the Block Reference manual for further details on these blocks.

AVEM block

Another block which is closely related to the mentioned ones is a block named AVEM which calculates a moving average of a given time series. The name might indicate that the AVEM block is another example for an If block, but actually the moving-average block is a Standard block.





The AVEM block calculates its output from a connected time series by the formula

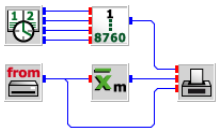
$$\bar{x}_j = \frac{1}{\min\{n, j\}} \sum_{i=\max\{1, j-n+1\}}^{j} x_i$$

which means that for any time step the AVEM block provides an average value over the previous time steps as defined by the block's parameter – let us neglect the initialization problem for the time being. This means, that the AVEM block outputs a value for each time step. But this is the behavior of a Standard block which always outputs a value, whenever it is called.

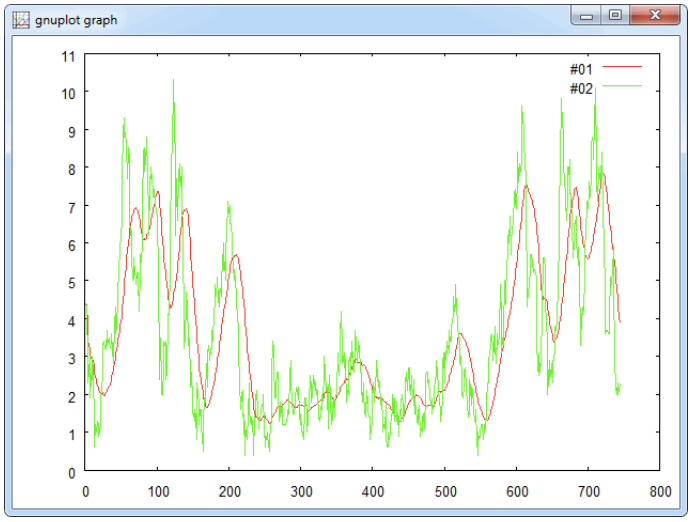
What makes the difference to If blocks is, that If blocks provide output values only under certain conditions and request from the inselEngine to execute the successors only now and then, depending on their condition.

As an example for the AVEM block we calculate the moving average of the wind speed data for January as saved in file `meteo82.dat`. As interval for the calculation of the moving average we use 24 hours. Remember that the wind speed is the last value in the records with format `F5.1`.

This is the block diagram



and this is the resulting plot:



Please observe that the AVEM block smoothens the high fluctuations in the hourly wind speed data – as expected.

Side remark about wind directions

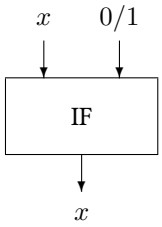
Allow us a last short remark on the AVEM block. Maybe you have the idea to look at the moving average of the wind direction time series. There is a little problem in doing so. Did you notice, that when you average wind directions from North that it may happen that the average of something like North-North-East and North-North-West must be calculated?

North-North-East direction corresponds to around 350 degrees and North-North-West to around 10 degrees. The average is 180 degrees, hence South direction, which is obvious nonsense. But we do not further look at this aspect here – one reason being that the AVEM block is not even an If block but a Standard block.

4.4 General if conditions

"The" If block

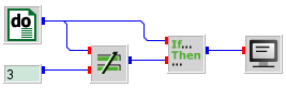
We turn our attention now to the more general cases of if conditions. The most natural candidate for a block of the group of If blocks is a block which gave the group of I-blocks its name: The block named IF.



if...
then

This block has two inputs, x and a logical input which can be either zero (false) or one (true), and one output – the signal that is connected to the x input. The IF block lets the input signal pass through, if the second input – the condition input – is true, otherwise it doesn't. "Otherwise it doesn't" means, the output is not available in the current step, and hence, the successors of the block, i. e., all blocks which make direct or indirect use of the IF block's output get no signal and are therefore not executed.

The best way to illustrate this behavior is a simple example. Let us construct a filter which lets all numbers pass except the number three.



The DO block counts from one to five, i. e., its parameters are set to 1 for the initial value, 5 for the final value, and 1 for the increment. The CONST block uses a value 3 as parameter. The block with the symbol $\neq$ is the NE block (not equal) and checks whether its two inputs are not equal (true) or equal (false). The NE block is a Standard block and can be found in the *Mathematics Logics* category.

Both, the output of the DO block and the output of the NE condition block are connected to the IF block. Finally, the IF block lets all values pass through, execept the value 3. So, from the point of view of the SCREEN block, which is connected to the IF block, the SCREEN block is served with data except when the output value of the DO block is equal to 3.

Test it! What do you expect to see as output? The values 1, 2, 4, and 5. Test it, please.

Logical conditions Like in any ordinary programming language the problem to set up an if structure is to formulate a condition which evaluates either to true (1) or false (0), and execute the if branch when the condition is true or to perform no operation if the condition is false.

INSEL provides blocks for the formulation of all standard logical conditions. These standard conditions are

- Equal (block EQ)
- Not equal (block NE)
- Greater than (block GT)

- Greater or equal (block GE)
- Less than (block LT)
- Less or equal (block LE)
- And (block AND)
- Inclusive or (block OR)
- Exclusive or (block XOR)

All of them are Standard blocks and with these blocks a lot of logical conditions can be constructed. The functions of the different blocks should be self explaining.

GE block

But let us look at the example of the GE block which checks for a greater-or-equal condition.



As expected the GE block has two inputs x_1 and x_2 and checks whether x_1 is greater of equal x_2. If yes, the block outputs a one, otherwise it outputs a zero. We have added an optional parameter p which weakens the hard equal condition.

Absolute equity?

What is the reason? With INSEL we are doing numerics mainly on the basis of Fortran REAL variables. These variables in the computer's memory have a rather limited accuracy of about seven to eight significant digits. Hence, comparing them to being absolutely equal might lead to unwanted results. Therefore, with the GE block the variables must not necessarily be absolutely equal but can differ by a tolerance p and are still considered equal by the GE block. When p is not specified, the GE block goes the hard way and compares for absolute equaltiy.

If some of the other condition blocks should be unclear, please refer to the Block Reference manual for the details.

4.4.1 Load profiles

In the next step let us practice to formulate if conditions for a realistic example. One of the most natural applications of the condition blocks like EQ, GE, GT, etc. is the formulation of load profiles in the widest sense.

Exercise 4.2 Let us assume we want to construct a condition for a public building – a library, for example – and we want to decide whether it is open (true) or not (false). First we have to define the hairy details.

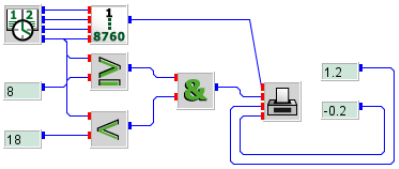
For reasons of simplicity, let us assume a not-too-complicated opening schedule. Let our library be open every day from 8 a.m. to 6 p.m. except the weekends, i. e., Saturday and Sunday, when our building is closed.

Try, and solve this problem as an exercise.

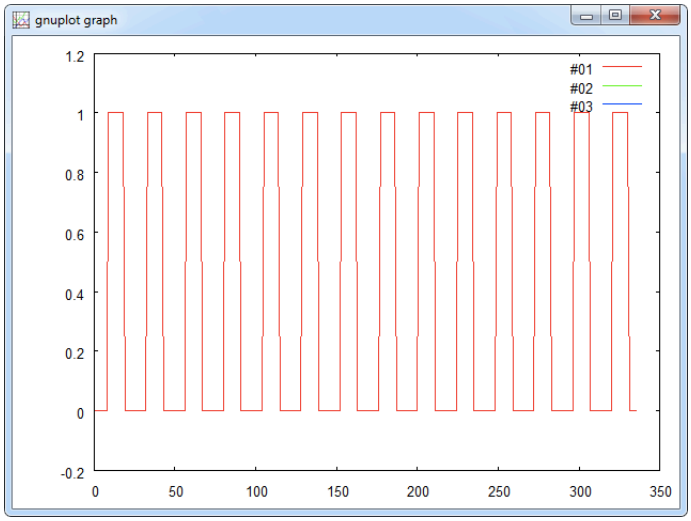
Solution Our solution process goes like this: At first, we ignore the complication of the weekend closure. Obviously, we will use a CLOCK block. The hour output of the CLOCK block will be used to decide whether it is already opening time or closing time. For sure, we need two constants for the opening time (8 o'clock) and the closing time (18 o'clock).

For the first step, we then need a GE block, a LT block, and an AND block to formulate our simplified condition. Please notice, that an LE block in combination with a constant 18 would keep the library open until 7 p.m. Do you copy?

Then we plot the opening condition in order to check whether our simplified solution works or not. If not, we go back and make changes until the simplified solution works. Our preliminary solution looks like this:

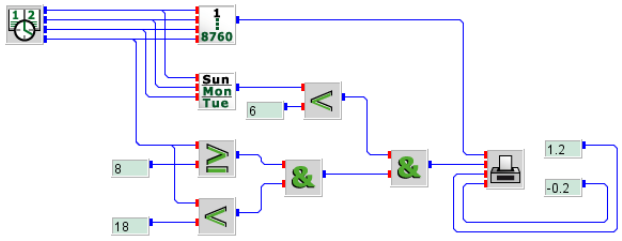


We have added two constants in order to make the plot a little nicer. The opening hours indicator for the first two weeks of January 2012 looks like this:

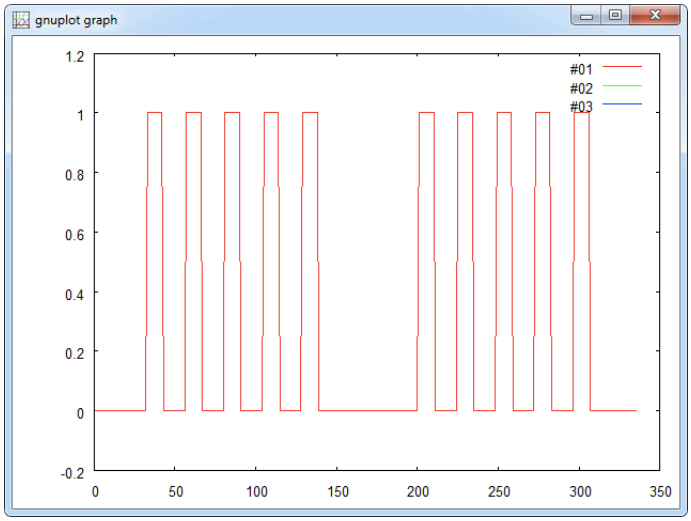


DOW block

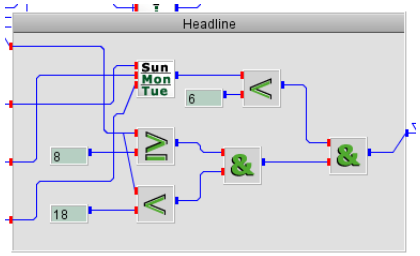
The last thing to complete our solution is to sort out the weekend case. In order to check for the day of the week we can use the DOW block which uses the a Gregorian date as input and returns a one for Monday, a two for Tuesday, and so forth. So, for our opening indicator we can check whether the DOW output is less than six – i. e., the library is open, or not. This makes a minor modification to our previous block diagram.



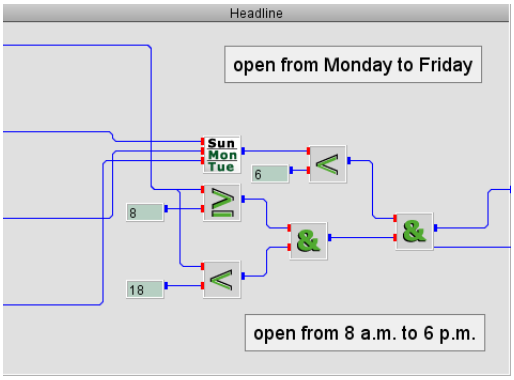
The opening scheme now looks as follows:



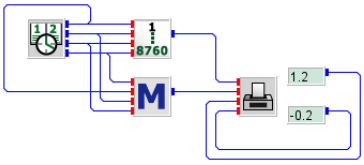
Macro This is the first place to use a macro an encapsulate the logic scheme. This makes the block diagram easier to read. Still a bit scrambled rooting.



In addition we can add some labels to our first macro which make it easier to understand what we did.



Finally, our block diagram reduces to



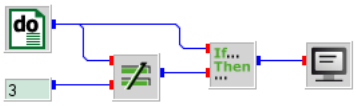
Exercise 4.3 As an exercise, add a modification to the opening schedule such that the library is closed during August an plot the annual opening scheme.

Blocks IFPOS and IFNEG

There are two If blocks named IFPOS and IFNEG which should perhaps be mentioned here, because they cover two rather common filters for (strictly) positive and (strictly) negative numbers. Their function and use are probably self explaining, if not, please check the Block Reference manual for further details.

4.5 Calculation list

Let us understand the concept of If blocks a little deeper by looking again at a previous example:



This model includes five INSEL blocks, namely the T-block DO, the C-block CONST, the S-blocks NE and SCREEN, and last but not least the I-block IF. Let us answer the question how exactly INSEL converts this block diagram into a calculation order.

As a general rule INSEL checks at first whether there are C-blocks included in the model. In this case INSEL finds exactly one C-block, namely the CONST block. INSEL "sorts" this block into the first place of the calculation list – we have already seen an example of a calculation list in Module , page 28.

Then INSEL looks for T-blocks, finds exactly one in the model, namely the DO block, and sorts the DO block into the second place of the calculation list. In the third step INSEL looks for S-blocks in the model. In this case there are two: the NE block and the SCREEN block.

Known inputs

As mentioned earlier, INSEL can execute blocks only, when their input signals are already "known", which means that they have an actual value. The "known" signals are all outputs of blocks in the calculation list so far, in our case this is the constant value of the CONST block and the output of the DO block. Hence, it is possible to sort the NE block into the third place of the calculation list. Please observe, that there is no way to sort the SCREEN block into the calculation list so far, since its input signal is not yet known, because the IF block does not yet appear in the calculation list.

There are no more S-blocks to consider in this example, so INSEL checks for blocks of other groups and finds the IF block. Since both its inputs are known already INSEL sorts the IF block into the fourth place of the calculation list. Now all blocks which make use of the IF block's output are analyzed – in this case the only left block is the SCREEN block, whose input is now known and can be sorted into the calculation list.

As a result, INSEL found the block order CONST, DO, NE, IF, and SCREEN.

It is now obvious that the CONST block is executed first. Due to the function of the block the constant parameter of the CONST block is connected with the blocks's output, that's all. The next block to call is the DO block, which connects its initial value with the block's output. Then the NE block compares its first and second output (not knowing where the values come from). If they are different, the NE blocks writes a 1 (logical true) to its output, otherwise a 0 (logical false).

Jump parameter

Next, the IF block is called. Two different things can happen: Either the second input is equal to 0, then the successors of the IF block are skipped, or the second input is equal to 1, then the successors of the IF block must be executed. During the sorting routine INSEL found that there is exactly one successor of the IF block, namely the SCREEN block. Usually INSEL jumps one step to the next block in the calculation list to find the next block to be executed, but after the IF block is executed INSEL needs to jump either one step to the SCREEN block, execute it, i. e., display the input on the monitor and then reach the end of the calculation list or jump two steps and skip the SCREEN block and reach the end of the calculation list.

The decision is made by the IF block, which is the only candidate who knows the meaning of its second input. The IF block informs INSEL what to do next, by setting the so-called Jump parameter either to 2 (skip the next block in the list) or 1 (execute the next block in the list)

When the end of the calculation list is reached, INSEL looks backward in the calculation list to find the next T-block and give control to it, which means that the DO block will increase its output by the increment defined as the block's third parameter and the next block is the DO blocks successor in the calculation list, i. e., the NE block. Please notice, that the CONST block will never be reached due to the calculation list rules.

The algorithm is executed until the DO block has "fired" all its values, then on the next call the DO block informs INSEL that nothing is left to do and INSEL ends the program.

We can summarize the discussion with a last look at the calculation list including the block names, block groups and Jump parameter values of each block:

```
Number    Block       Group    Jump
-----------------------------------
     4    CONST         C         1
     5    DO            T         1
     2    NE            S         1
     1    IF            I        -2
```

```
     3    SCREEN        S        -3
----------------------------------
```

Forward jumps

Please observe, that rather than pointing to the end of the calculation list the Jump parameters point to the block which has to be executed next. So – although the IF block has a negative parameter in this example – I-blocks are characterised by the property that they allow forward jumps in the calculation list.

Nested If blocks

In the discusion of timer blocks we have seen that T-blocks can be nested. It is also possible to nest I-blocks, but it is time for a break and we postpone this topic for the time being.

Exercise 4.4 Calculate the annual mean ambient temperature as stored in file `meteo82.dat`.

Exercise 4.5 Plot the daily mean ambient temperature as stored in file `meteo82.dat`.

Summary

- You have learnt that If blocks – or I-blocks, in short – can be used to skip execution of blocks which are directly or indirectly connected to I-blocks.
- Some typical examples for blocks of the I-group are blocks which calculate averages or cumulative sums, for example.
- There is a set of blocks which perform numerical fits to statistical data like the linear regression block FITLIN, for example.
- A block named IF allows for the definition of practically arbitrary conditions.
- There are blocks like EQ, NE, etc. which can be used to construct general conditions from very simple to very complex.

5 :: Delay and Loop blocks

The previous Module started with the statement "A programming language is not a programming language if it does not provide at least one statement which enables the use of an if-then-else structure." The same statement is valid for loop structures: A programming language is not a programming language if it does not provide at least one statement which enables the use of a loop structure.

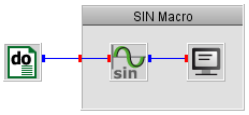
If you have studied the Tutorial from the beginning, you may intervene: We have used DO blocks and CLOCK blocks so often, aren't we through with loops in INSEL? No, we aren't. The blocks DO and CLOCK are T-blocks, not L-blocks. So, let us have a closer look at the difference between these two block groups.

No loop at all

Coming back to one of the most trivial examples of this Tutorial, where we have just calculated the sine of 45° on page 18. This example didn't use a timer at all. The CONST block, the SIN block, and the SCREEN were only called once. We could put the complete model into a macro with no inputs and no outputs.

One timer

If we wish that this macro (or model) depends on a variable input angle, we could add a DO block, delete the CONST block and connect the DO block's output with the sine block. This results in a loop.



Two timers

We could put this complete model into one macro gain, add an input to the DO block, and connect it to another DO block outside the macro, ending up in a nested DO block structure. There is no limit in nesting DO blocks and there is no limit in macro depth.

Three timers...

So we could continue in the same way, as long as we wish: Put the complete model into a macro again, add an input to the DO block, and so on.

What we can learn from this simple example is, that Timer blocks can be used to create nested loop structures. But these loops always run over complete models, i. e., over all blocks which are connected to the respective timer's output and those blocks' successors. With this concept it is impossible to create local loops.

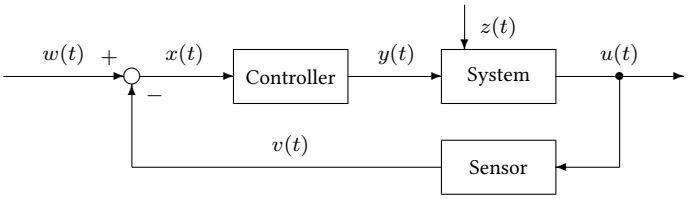
In other words, so far in this Tutorial we have treated "linear" simulation models only. Linear means here that any INSEL application we can write at this point follows basically a sequential structure, i. e., normally there is a Timer block which decides on the duration and time step of model execution and the rest of the model is executed more or less in a sequential order, except when there is an If block included, which allows to skip execution of some blocks depending of the conditions of the If blocks.

To express this fact in the language of structured programming, we have understood how we can handle sequential structures and if-then-else structures. The third required

concept in structured programming is the concept of loops, which exactly is the topic of this Module.

5.1 Handling control cycles

Let us look at a control cycle which is typical in measurement and control technology.



The task of a control cycle is to keep a controlled process variable u within in a narrow range close to a given set point w. The variable u usually depends on w and a disturbance variable z.

At this point, we are not really interested in control strategies. Instead, we want to analyze the control cycle from a structural point of view. So let us assume that the values of the command variable w and the disturbance variable z are known. How can we perform a calculation of the cycle states?

The sum $x = w - v$ cannot be calculated because the sum depends on the output of the control process, i. e., the value of the feedback variable v, which is not yet known. Since the sum is unknown, the controller cannot be executed and therefore the values of u and v cannot be calculated. But the value of v is necessary to know when we want to calculate the sum x. So, what?

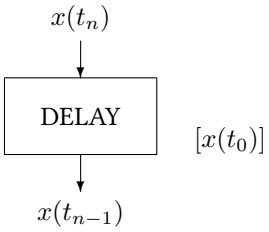
Algebraic loops

Closed loops like the one just described are called algebraic loops in computing. The solution of this problem is well known since the early times of analogue computing, i. e., when block diagram programming had its roots: Insert a delay element into the algebraic loop. So what is a delay element?

The characteristic properties of a delay element are that it delays its input signal for a specific time and, very important, that it is initialized with a value. This idea is the basis for a huge set of applications, ranging from numeric integration methods, numeric solutions of differential equations, and of course control cycles.

5.1.1 The DELAY block

In INSEL one delay element is a block from the group of Delay blocks named DELAY.



The DELAY block delays its input by one step. The optional parameter $x(t_0)$ is used as initial value. If not declared, $x(t_0)$ defaults to zero. The DELAY block can be found in the *Mathematics Loops* category.

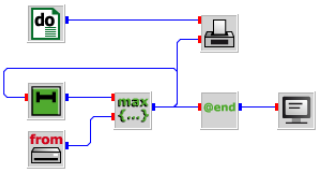
Controllers are typical Delay blocks in INSEL. And in fact, assuming that the controller starts with an initial value, let's say y_0, this simple measure solves our algebraic loop problem. Now that both inputs $y = y_0$ and z are known the system can deliver u and the sensor the required value v.

Exercise 5.1 In Module you used the I-block MAXX (Absolute maximum) from the *Statistics Maximum* category to find the overall maximum value for the hourly global irradiance on a horizontal surface in W/m^2 as saved in file `meteo82.dat`.

In the category *Mathematics Basics* you can find an S-block named MAX (Maximum) which outputs the maximum value of its connected inputs.

Can you use this block to find the overall maximum radiation value, too?

Solution The solution makes use of a DELAY block, of course.

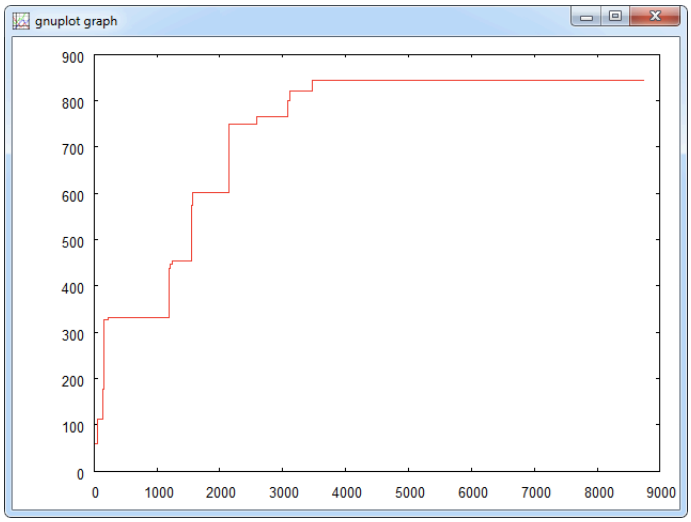


Since the time series starts at midnight, the radiation data are zero during the first calls and with an initial value zero of the DELAY block nothing happens. But when the first radiation value greater than zero occurs in `meteo82.dat`, the MAX block returns this value to the DELAY block which in return sets its output to this value.

In the next step the MAX block compares this output of the DELAY block with the next radiation value from the file. If the value from the READ block is greater than the output of the DELAY block, the MAX block returns this value to the DELAY block, otherwise the DELAY block receives its old value and nothing happens.

At the end of the simulation run the absolute maximum of the radiation time series is available at the MAX block's output.

In order to avoid too much SCREEN output, all data except the at-end value are filtered through the ATEND block. The next graph shows the evolution of the maximum with time.



It is interesting to have a look at the calculation list:

```
Number     Block        Group     Jump
-----------------------------------
     2     DO             T         1
     6     READ           S         1
     7     MAX            S         1
     1     ATEND          I         2
     3     SCREEN         S         1
     4     PLOT           S         1
     5     DELAY          D        -6
-----------------------------------
```

Please observe four details: (i) How the ATEND block jumps over its successor to the PLOT block, (ii) that the DELAY block points all the way up to the DO block as its successor, (iii) that we have used blocks from four different block groups in this simple exercise, and (iv) that the DELAY block is the last block in the calculation list, which is a typical property of all D-blocks.

Especially the last remark is worth a closer look. All blocks in INSEL depend on their inputs. This was one of the very first things we have learnt in Module . Now we learn, that Delay blocks are an exception to this rule. Why?

Constructor call

Delay blocks have an initial value at their output, before these blocks are called for the first time. Is this a miracle? Of course not. INSEL has a mechanism called constructor call – similar to the destructor call we became acquainted with in Module , page 80.

Before an INSEL model is executed by the inselEngine all blocks are called in this constructor call mode.

The constructor call is the time to check the plausibility of parameters fixed in the INSEL model. If for instance a value zero is provided as parameter of an attenuator block INSEL generates an error message and does not execute the model in order to avoid a division-by-zero exception. And this is the time to initialize the outputs of Delay blocks. But what happens when a model is executed?

Inputs as function of own outputs

The inselEngine must ensure, that all blocks which make direct use of the initial value of the DELAY block have a chance to access this value, and not the value after the DELAY block has been executed. So, in many cases all D-blocks appear at the end of the calculation list. In principle, it is possible to add a D-block to the calculation list, as soon as all blocks which make direct use of its output are already in the calculation list.

In the last example we have seen, that the output of the DELAY block is connected to a MAX block, which calculates the maximum on the basis of the DELAY block's output. The output of the MAX block is connected to the DELAY block as input. In consequence, this means that in case of the DELAY block its input depends on its actual output. This will become even clearer when we have a closer look at the group of L-Blocks later in this Module.

5.1.2 PID controller

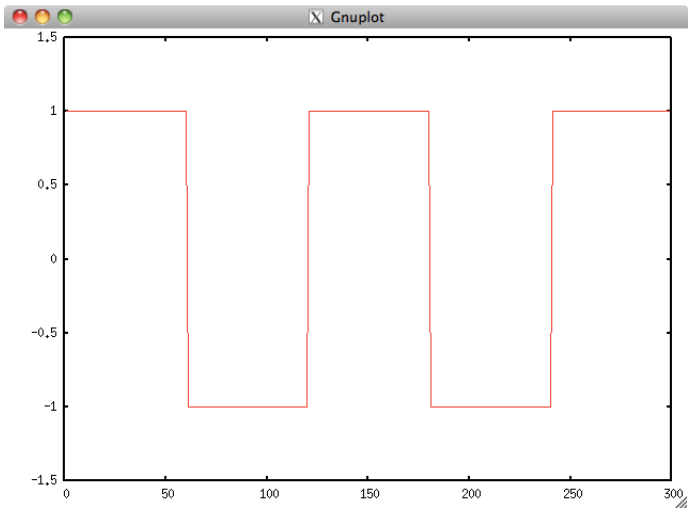
Coming back to control cycles, let us use a PID controller to follow a given signal, a step function, for instance.

What is a PID controller?

In order to prepare the solution, let us construct a demonstration signal.

Exercise 5.2 Construct an INSEL model for a step function which runs over five minutes in time steps of one second. The output signal shall vary between the values minus one and plus one with a sharp ramp, changing every 60 seconds.

Solution We have solved this problem by using two DO blocks, one ... see tutorial–ramp.vseit



idea 1: time axis

idea 2: plus one, minus one cahne via expg block

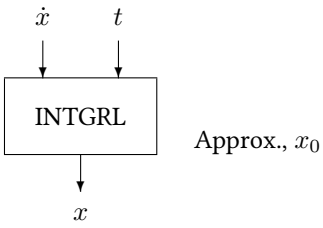
ides 3: introction of PID block

5.2 Solving differential equations

Sophisticated integration of differential equations. Long history before digital computing could overtake analogue simulation equipment

Maybe history, why digital block diagram simulation in the 60's practically had no chance against analogue computing - compared to today: extremely slow processors





Approx., x_0

$$\dot{x} = \cos(t) \Rightarrow x = \sin(t)$$

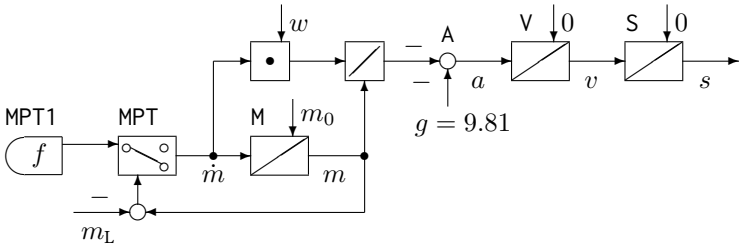
5.2.1 The Jentsch rocket

In his wonderful book "Digital simulation of continuos systems," published 1969, Jentsch [?] used the simple differential equation of a starting rocket to illustrate the principle of solving differential equations by the use of simulation languages. The equation is

$$a = -\frac{\dot{m}w}{m} - g$$

where a is the acceleration of the rocket, m the mass of the rocket (including gas), $\dot{m}$ the change in mass due to gas ejection, w is the velocity of the ejected gas relative to the rocket, and g the gravity of Earth, $g \approx 9.81\ \mathrm{m\,s^{-2}}$.

Since probably the younger readers of this Tutorial have never seen an "old-fashioned" block diagram description of a differential equation, here comes an adaption of Jentsch's example:



Starting from $\dot{m}$, the integrator `M` – a delay block – with initial value m_0 approximates the rocket mass m. The change in mass $\dot{m}$ is multiplied by w and divided by m by the two blocks marked with a dot and a division symbol. Finally, the acceleration a results from the summation block `A` – denoted by a small circle. By convention, the required minus signs are written close to the arrows pointing into the summation blocks. Velocity v and distance s are calculated by two more integrators named `V` and `S` with initial values zero.

Jentsch lets the example run through two blocks named `MPT1` and `MPT` over a time interval of twenty seconds.

Block `MPT1` determines the behavior of the ...

Block `MPT` represents a relay switches off ...

```
* --- Structure
      S    = I(0,V)
      V    = I(0,A)
      A    = -(MPT * W) / M - 9.81
      M    = I(M0,MPT)
      MPT  = REL(M - ML,MTP1)
      MPT1 = KUL(KLMPT1)
* --- Parameters
```

```
      W      = 3000
      M0     = 3300
      ML     = 300
      KLMPT1 = 0,-160, 20,-160
* --- Processing
      TIME   = (0/0.1,20)
      PRTIME = (0/0.1,20)
      PRINT(1,1) M,V,S / 1000
         FORMAT 1 (2(1) / 3)
         HEAD 1   (M,KG,V,M/S,S,KM)
      PLOT(2,1) M,V,S / 1000
         FORMAT 2 (0,4000,3/0,1.E+4,4,4/0,60,5)
      END
         KLMPT1 = 0,-320, 18.75,0
      END
      STOP
```

Exercise 5.3 Can you convert Jentsch's rocket example into an INSEL model?

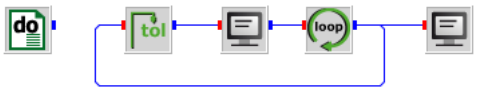
Solution ■

5.2.2 Solar collector equation

5.3 Loop block concept

Explain LOOP, NULL, and MPP.

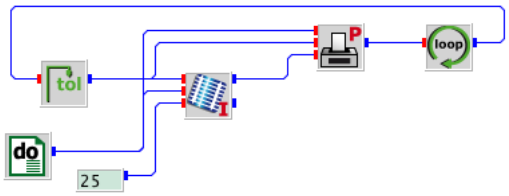
loop.vseit



A LOOP block and a TOL block are connected in a loop. The LOOP block uses 1 as initial value, 3 as final value and 1 as increment. Two SCREEN blocks display the outputs of the TOL and LOOP block, respectively.

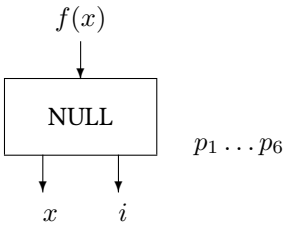
```
   1.0000000
   2.0000000
   3.0000000
Final output    3.0
   1.0000000
```

```
   2.0000000
   3.0000000
Final output    3.0
```



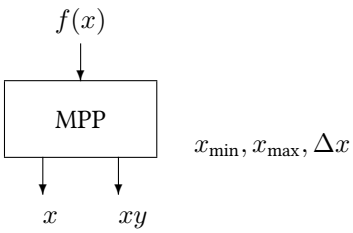
Example: NULL block - root of a function, involution algorithm, regula falsi algorithm.





More applied: maximum power point calculation.





Mention only: Even more applied: Battery charge regulator – see Module 7.2

Loop Blocks and Iterations

Iteration blocks are called Loop block or short L-Blocks. In INSEL the iteration blocks are the LOOP, MPP and NULL block.

- The LOOP block runs through a sequence of values defined by parameters, restricted to a part of the simulation model.

- The NULL block searches a root of a continuous function.

- The MPP block simulates an ideal maximum power point tracker. In general, the MPP block can be used to find the maximum of any unimodal function.

The output of an L-block must always be the input of a TOL (top of Loop) block.

PART II :: Applications and exercises

6 :: Solar meteorology

This is the first of three Modules which cover broad INSEL application fields. It concentrates on the aspects of meteorological data that are relevant in renewable energy applications, like solar electricity generation, solar thermal heating and cooling, desalination systems, biomass, wind turbine simulation, storages, hydrogen technology, building simulation, daylighting etc. INSEL fully covers all important meteorological parameters as there are solar radiation, ambient temperature, humidity, precipitation, and wind speed.

The Module does not cover the theoretical background for the calculations in detail. More information can be found in the block reference manual of INSEL. A full theoretical derivation of all the used methods can be found in the book Simulation of Solar Energy Systems by J. Schumacher.[1]

6.1 Global radiation

We start with the source of all life on Earth: the Sun. From a simulation point of view it can be considered as a black body at a temperature of 5777 K. As such it emits electromagnetic radiation with a theoretical spectrum following Planck's law

$$E(\lambda, T) = \frac{2\pi hc^2}{\lambda^5} \left(\exp\left(\frac{hc}{\lambda k T} \right) - 1 \right)^{-1} \tag{6.1}$$

where λ denotes the wave length, h is the Planck constant $6.6260755 \times 10^{-34}$ J s, c is the speed of light in vacuum 299 792 458 m s^{-1}, k is the Boltzmann constant 1.380658×10^{-23} J K^{-1} and T the temperature of the black body in kelvin.

The INSEL block PLANCK can be used to calculate the spectrum either as a function of the wavelength λ, or the frequency ν or the energy of the photons $h\nu$. You find it as type *Planck's radiation law* under the category *Meteorology* > *Solar radiation*.

Solar spectrum

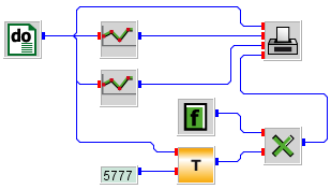
The real spectrum of the solar radiation which arrives at the surface of the Earth depends on many factors, like solar position, atmospheric conditions, for example. There is a standard which defines so-called AM 1.5 spectrum at a solar radiation of 1000 W m^{-2}, the 1.5 means that the rays of the Sun pass 1.5 times the shortest way through the atmosphere. This spectrum as well as the undisturbed spectrum outside atmosphere AM 0 is available in INSEL under the category *Meteorology* > *Spectrum*.

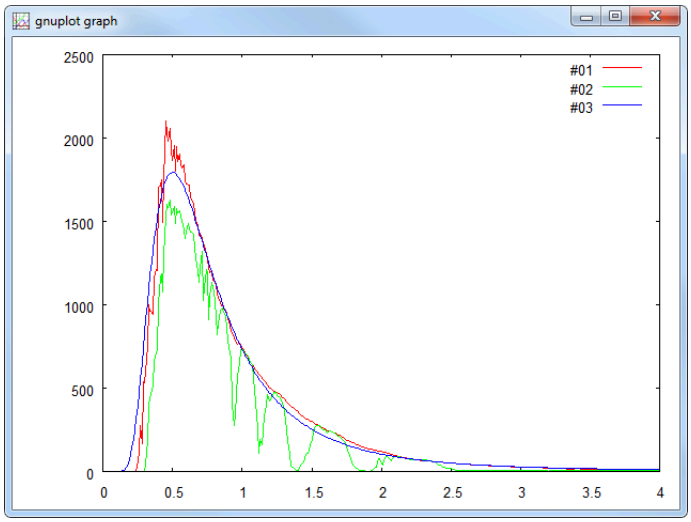
Exercise 6.1 Plot the theoretical Planck spectrum, the AM 1.5, and the AM 0 spectrum for a wavelength between 0 and 4 μm.

Hint

The Planck spectrum at 5777 kelvin is the spectrum at the solar surface. Before the radiation reaches the Earth it is diluted by a factor 2.1645×10^{-5}. You find the dilution factor under *Mathematics* > *Constants*.

Solution





The x-coordinate is the wavelength in micrometer, the y-coordinate shows the value of the electromagnetic terrestrial radiation in $\mathrm{W\,m^{-2}\,\mu m^{-1}}$.

Solar constant

The Stefan Boltzmann law is the result of the integral of an electromagnetic spectrum over all wavelengths. It says

$$E(T) = \int_0^\infty E(\lambda, T) d\lambda = \sigma T^4 \tag{6.2}$$

$\sigma = 5.6703 \times 10^{-8} \mathrm{W\,m^{-2}\,K^{-4}}$ is known as Boltzmann constant.

If the solar AM 1.5 spectrum $G(\lambda)$ is integrated the result is the solar constant $G_s = 1367\ \mathrm{W\,m^{-2}}$.

Exercise 6.2 Calculate the solar constant with an INSEL model.

Solution



[1] Not yet published.

We used the Global cumulation block CUM from the *Statistics* category and a GAIN block from *Mathematics* > *Basics*. Our choice of the wavelength interval [0,10] and increment 0.01 leads to the value 1368.2 W m^{-2}, which is close enough to the real value. Do not forget to set the parameter of the GAIN block to 0.01.

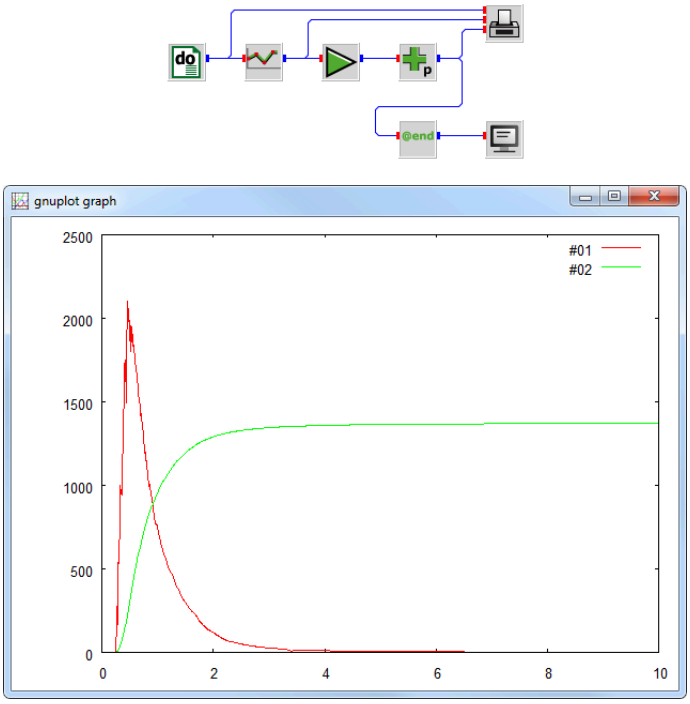
Exercise 6.3 Calculate the solar constant and plot the function

$$F(\lambda) = \int_0^{\lambda} G(\lambda) d\lambda \tag{6.3}$$

in one model.

Hint Replace the Global cumulation block CUM by a Summation with reset block SUMP and choose an appropriate parameter.

Solution



We have set the parameter of the SUMP block exactly to the number of steps the DO block performs, ie 1001. With tribute to laziness one could set it to any high value like 100000, for example, without having to figure out the correct number.

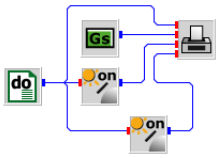
We have plotted the AM 1.5 spectrum in addition, and you can see that above 5 μm nearly nothing happens any more.

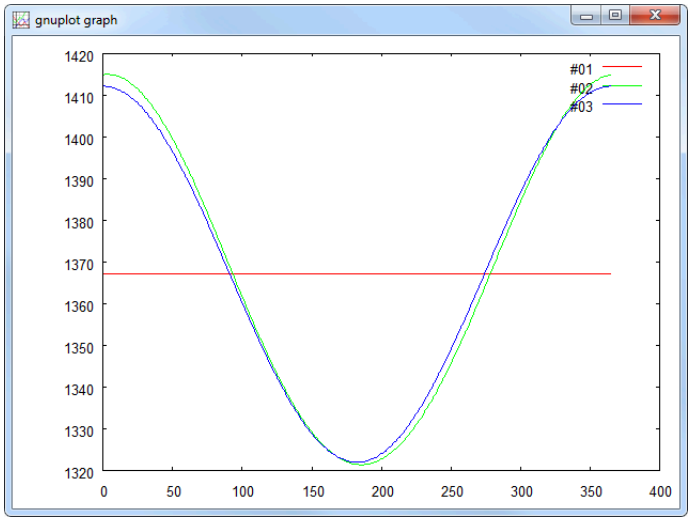
Distance Sun–Earth

The solar "constant" is not really constant, even if the spectrum is assumed constant AM 1.5. The reason lies in the dilution factor, which is a function of the distance between Sun and Earth. And this distance varies due to the elliptic shape of the Earth's orbit. Of course, INSEL has a block for the calculation of the direct normal extraterrestrial irradiance, i. e., in the direction towards the Sun, the GON block under *Meteorology* > *Solar radiation*.

Exercise 6.4 Plot the annual variation of the solar constant.

Solution





Extraterrestrial radiation

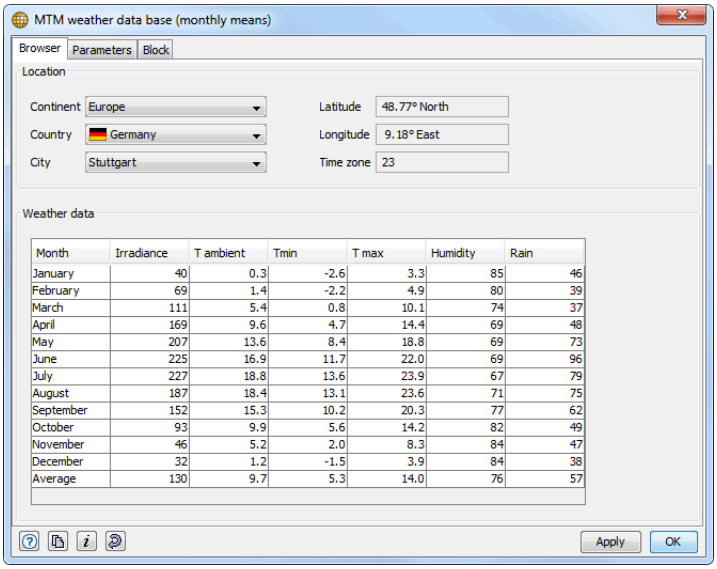
The radiation values we have calculated so far are of rather theoretical value – except for extraterrestrial applications. Terrestrial data always depend on the location, this is also true if we want to know the radiation outside atmosphere at a particular place and a particular time.

For the calculation of the extraterrestrial radiation on a horizontal plane at a given location INSEL provides the GOH block under *Meteo* > *Solar radiation*. The location is specified through latitude φ (north positve, south negative), longitude λ, (west of Greenwich positiv, east negative) and time zone Z (Greenwich mean time GMT = 0, Central European time CET = 23, counted positive in western direction).

Exercise 6.5 Plot the extraterrestrial radiation at an arbitrary location of your interest for every day's

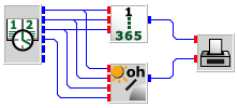
noon, i. e., 12:00 zone time.

Hint Whenever you are looking for local coordinates and time zone values, check the INSEL weather data base, which contains more than 2000 locations worldwide.

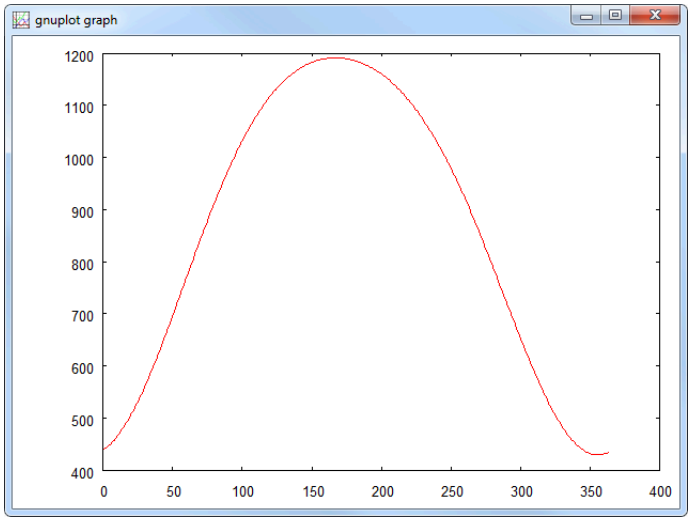


Month	Irradiance	T ambient	Tmin	T max	Humidity	Rain
January	40	0.3	-2.6	3.3	85	46
February	69	1.4	-2.2	4.9	80	39
March	111	5.4	0.8	10.1	74	37
April	169	9.6	4.7	14.4	69	48
May	207	13.6	8.4	18.8	69	73
June	225	16.9	11.7	22.0	69	96
July	227	18.8	13.6	23.9	67	79
August	187	18.4	13.1	23.6	71	75
September	152	15.3	10.2	20.3	77	62
October	93	9.9	5.6	14.2	82	49
November	46	5.2	2.0	8.3	84	47
December	32	1.2	-1.5	3.9	84	38
Average	130	9.7	5.3	14.0	76	57

Solution



We live in Stuttgart, Germany. So we used our coordinates. With the CLOCK block running in steps of one day for 12 o'clock this is our solar radiation – on top of the Stuttgart atmosphere.



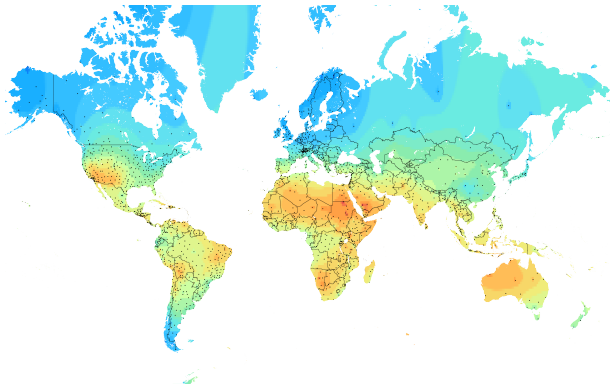
Terrestrial radiation

How much of this radiation does arrive at a terrestrial solar installation – let's say every hour of the year? The answer is difficult – of course we could say "Depends on the weather."

In case you have some recorded data, maybe in a resolution of one hour, then you can just read them in and use them. The procedure how to do this, the whole Format staff – all this has been discussed in Module . There is no need to discuss this issue here again.

Maybe you have given monthly means from a weather service, for example. Well, that is at least a starting point. Maybe you have no idea, what even the 12 monthly mean values are. Then the INSEL monthly mean weather data base can help. These are the locations available:

2000 locations in the INSEL weather data base



INSEL has access to the data via the MTM block under *Meteorology* > *Data*. The use of

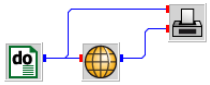
the block is very convenient: just browse to the location, ready. As an alternative you may wish to access locations directly via their coordinates. The *MTM weather data base (via lat/long)* type allows this.

Exercise 6.6 Plot the 12 monthly means values in the data base closest to your home place. The global radiation unit in INSEL is always $\mathrm{W\,m^{-2}}$, by default. Plot your data in $\mathrm{kWh\,m^{-2}\,d^{-1}}$.

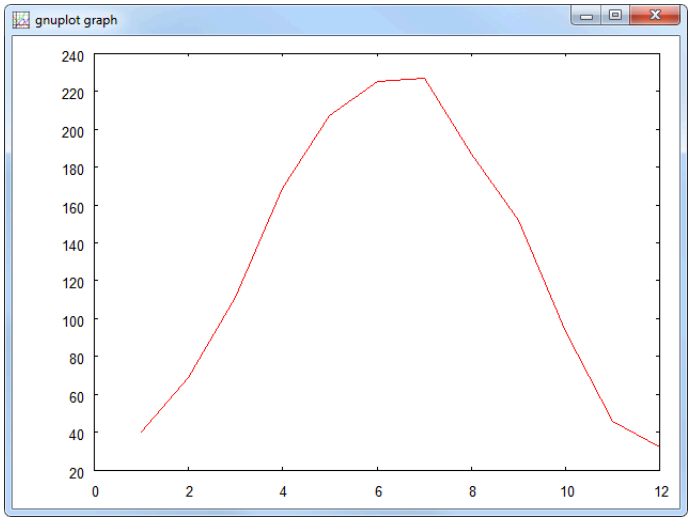
Solution

$$1\,\frac{\mathrm{W}}{\mathrm{m^2}} = \frac{1\,\mathrm{k}}{1000}\,\frac{24\,\mathrm{h}}{\mathrm{d}}\,\frac{\mathrm{W}}{\mathrm{m^2}} = 0.024\,\frac{\mathrm{kWh}}{\mathrm{m^2 d}} \tag{6.4}$$

Hence, we use a GAIN block with parameter 0.024.



This is our solar radiation – under the Stuttgart atmosphere:



The x-coordinate is the month, y-coordinate is the global irradiance in $\mathrm{kWh\,m^{-2}\,d^{-1}}$ on a horizontal plane in Stuttgart, Germany.

6.2 Radiation time series generation

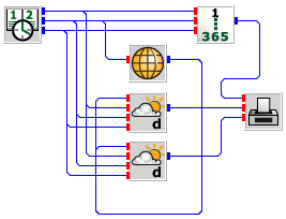
INSEL provides blocks which can generate synthetic time series of different meteorological data, solar radiation data in particular. The GENGD block delivers daily radiation data calculated from monthly means with excellent statistical properties over long periods like 20 years or more. It does not include a model for climate change, this is

a different playground. But from year to year the monthly means of global radiation differ significantly. Plenty of research has gone into the topic of weather data generation, the state-of-the-art is implemented in INSEL.

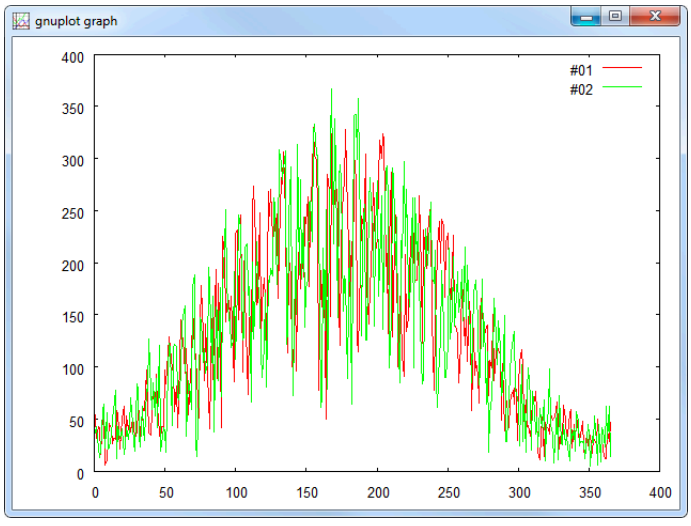
Exercise 6.7 Plot a time series of global radiation data on a horizontal plane over a period of one year in daily resolution for a location of your choice.

Hint Set the parameter for the year-to-year variability to zero, so that the given monthly mean from the MTM block is approximated as good as possible. Leave all other parameters as their defaults (except the location data, of course).

Solution



Since INSEL offers two models for the time series generation – the auto-regressive model of Gordon and Reddy and the Markov-matrix-based model of Aguilar and Collares-Pereira – we used both in order to compare their respective results. To make them comparable, the year-to-year variability parameter has been set to zero in both cases, otherwise the time series cannot be compared because it is not known in advance, how big the noise of the individual monthly means would be.



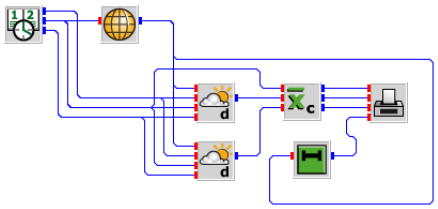
The x-axis shows the day of the year, the y-axis shows synthetic daily means of global

irradiance on a horizontal plane in W m^{-2}. The red line 01 results from the Gordon Reddy model, the green 02 curve is the result of the Aguiar Collares- Pereira model. At a first glance there is no significant difference between the two.

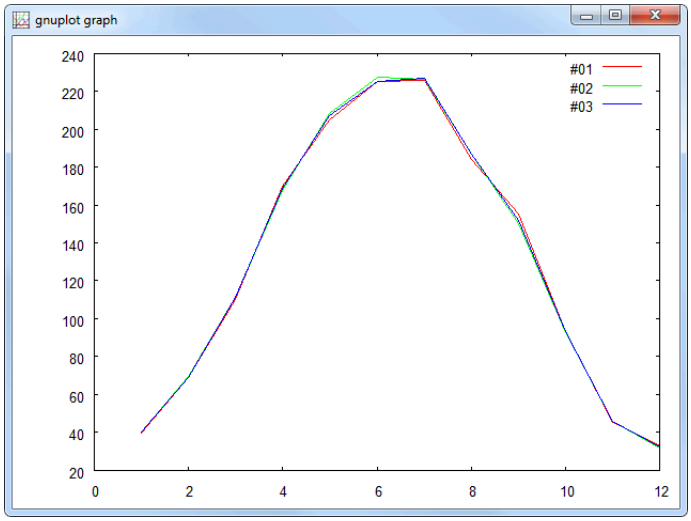
Exercise 6.8 Calculate the monthly means of the synthetic time series and compare it with the data that come from the MTM block.

Hint You will probably use the AVEC block for the calculation. Remember the earlier discussion – it is probably a good idea to think about a Delay block.

Solution



Since the successors of the AVEC block are executed only after the condition input has changed, the output of the MTM block needs to be delayed by one time step. Otherwise, the one-month-ahead value of the radiation would be plotted due to the functionality of the AVEC block.



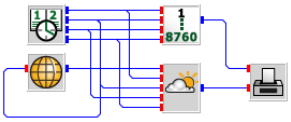
The x-axis shows the months, the y-axis shows monthly means of global irradiance on a horizontal plane in W m^{-2}. The 01 (red) line are the monthly means as recalculated from the Gordon Reddy model, 02 (green) is the result of the Aguiar Collares-Pereira model, 03 (blue) are the values taken from the INSEL weather data base.

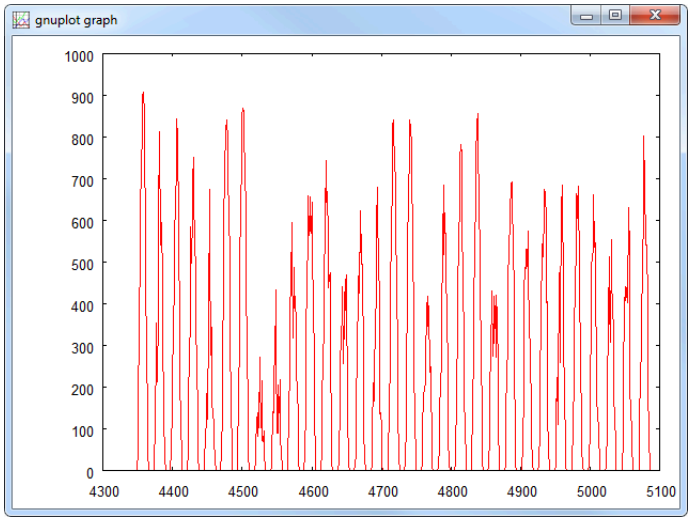
Hourly data

Once, daily radiation data are available a time series in hourly resolution can be generated for each day. The method is based on an autoregressive model developed by Aguiar and Collares-Pereira. In INSEL it is available as block GENGH. For convenience of the user, both blocks GENGD and GENGH are combined as block GENG. But the method and models are the same: at first, a daily time series is generated from the monthly mean value, then for each day time series of hourly data are generated from the daily means.

Exercise 6.9 Plot a time series of global radiation data on a horizontal plane over a period of one month in hourly resolution for a location of your choice.

Solution





The x-axis shows the hour of the year for the month July, the y-axis shows synthetic hourly means of global irradiance on a horizontal plane in $\mathrm{W\,m^{-2}}$.

Tilted surfaces

Only seldom a solar installation is exactly horizontal. In most cases the receivers are tilted by an angle $\beta \neq 0$ and orientated by an azimuth angle γ towards the equator, i. e., to the south ($\gamma = 180°$ in INSEL) on the northern hemisphere or to the north ($\gamma = 0°$ in INSEL) on the southern hemisphere, respectively.

There are plenty of models which can be used to convert horizontal data to tilted. Most of them use the same approach: in a first step the radiation data are split up into their beam and diffuse fractions by some statistical correlation, and in a second step both

components are converted to the tilted surface. Concerning the beam part G_{bh} the conversion can be done by pure geometry, in the case of the diffuse radiation some assumption about its distribution over the sky dome must be made.

Since tilted surfaces always "see" a part of the ground, this portion depends on the ground reflectance, or albedo ρ. Since it plays a minor role, the albedo is usually set constant to $\rho = 0.2$. In fact – like when the ground is covered by snow – much higher values can occur.

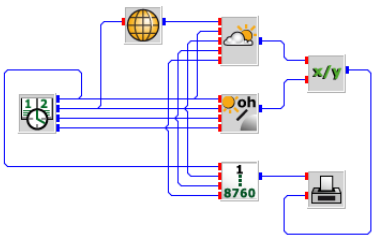
The correlations which calculate the diffuse fraction are based on the clearness index k_{t}. It is a good measure for the clearness of the atmosphere and is defined as the ratio between the global radiation that arrives at the Earth's surface on a horizontal plane G_{h} and its extraterrestrial pendant G_{oh}.

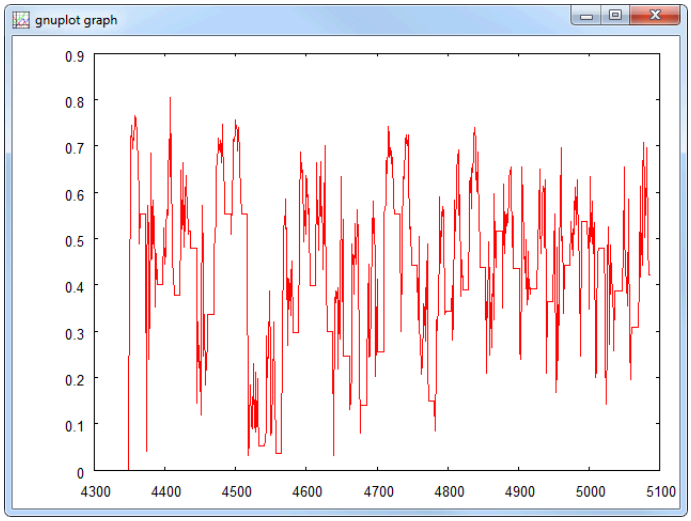
$$k_{\mathrm{t}} = \frac{G_{\mathrm{h}}}{G_{\mathrm{oh}}}$$

Due due its definition k_{t} can take theoretical values between zero and one. In practice, values outside the interval $[0.2, 0.8]$ are not very probable. Please observe, that at night, when $G_{\mathrm{oh}} = 0\ \mathrm{W\,m^{-2}}$, the clearness index is not defined. Although it is clear that in these cases the radiation to any orientated surface will also be $G_{\mathrm{t}} = 0\ \mathrm{W\,m^{-2}}$ this case can cause numerical inconvenience.

Exercise 6.10 Calculate the time series of clearness indices from the hourly radiation data of the last exercise. As an alternative you may wish to use measured radiation data from file `meteo82.dat` which has been introduced in Module , page 43.

Solution





The x-axis shows the hour of the year again, the y-axis shows synthetic hourly means of the clearness index.

Please observe that the problem of division by zero is handled by the DIV block by performing no operation, which leads to constant values of the clearness index during the night.

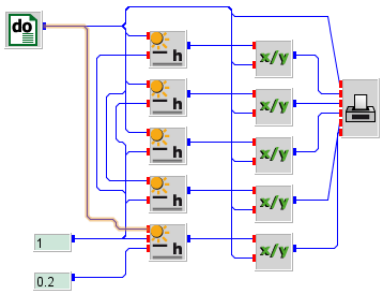
6.3 Diffuse radiation

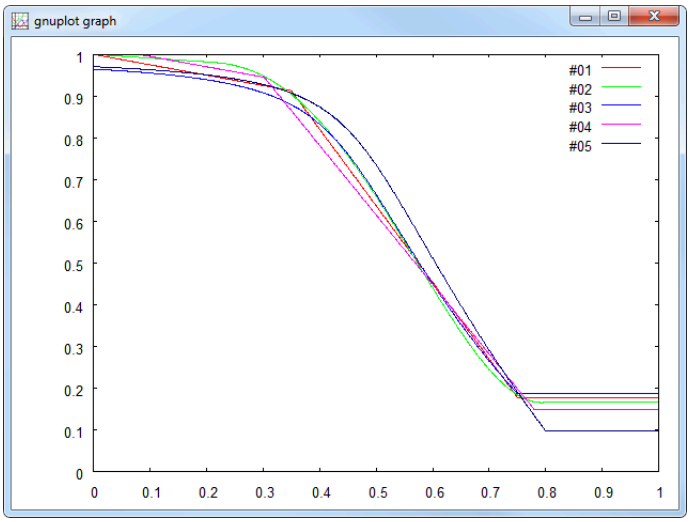
The blocks which calculate the diffuse fraction from global radiation are distiguished by the averaging interval of the radiation data, i. e., there are correlations for monthly means G2GDM (read: global to global diffuse monthly means), daily means G2GDD, and hourly means G2GDH. The blocks are found under the *Meteorology* > *Solar radiation* category. The reason, why these blocks require the inputs G_h and G_oh should be clear by now.

Exercise 6.11 Plot the diffuse fraction, i. e., the ratio $G_\mathrm{dh}/G_\mathrm{h}$ as a function of k_t for the first five correlations of the G2GDH block.

Hint This exercise is a bit tricky. Instead of real radiation data, use a DO block, which runs from zero to one and connect it with the G_h input. Remember, the correlations depend only on k_t.

Solution





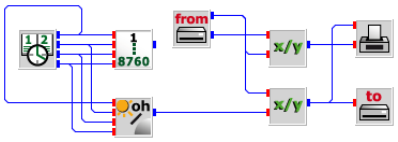
The x-axis shows the clearness index k_t, the y-axis shows the diffuse fraction $G_\mathrm{dh}/G_\mathrm{h}$. The correlations are: 01 (red) Orgill and Hollands correlation, 02 (green) Erbs, Klein and Duffie correlation, 03 (blue) Hollands correlation, 04 (magenta) Reindl, Beckman and Duffie, 05 (black) Hollands and Chra for an albedo of 0.2.

For tough guys

Would it not be interesting to compare the correlations with some real data? If you like, use any data file you have or the file `meteo82.dat` from Oldenburg, Germany (latitude 53.133 °N, longitude 8.217 °E, time zone 23) for the calculation of the diffuse fraction as function of k_t and bring the data and the correlations together in one plot.

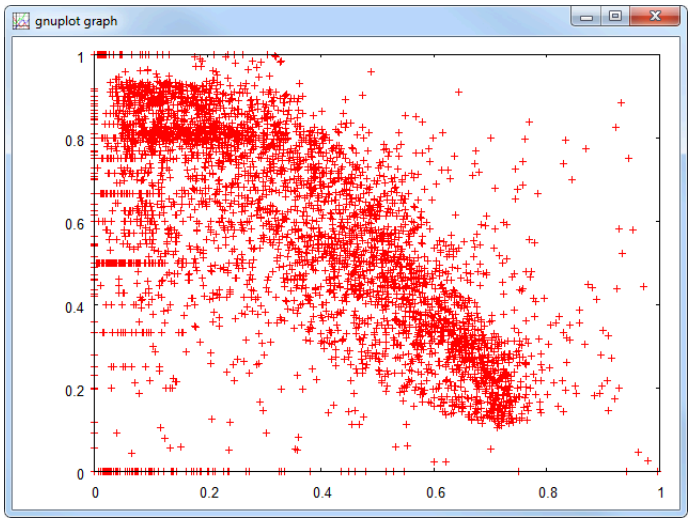
Solution

The main problem with this exercise is that both tasks have completely different independent variables. Therefore the solution is split up into subtasks. The first step is to calculate the hourly values from `meteo82.dat` and write the results to a file, let's say `ktDataOL.dat`.



In our example you see the HOY block as a relict. It is advisable to always check the data in files whether they show reasonable values. When you plot the diffuse data you will find that there is a lack of data for some hours indicated by values set to -40. So we restricted the values to the interval $x \in [0,1]$ and $y \in [0,1]$ with Gnuplot to check the data with the following steps

- Open Gnuplot from the tool bar.
- Change to the hidden application data directory, like `cd 'c:\Users\Myself\AppData\Roaming\doppelintegral\INSEL\tmp`.
- Type `load 'insel.gnu` (you will see a lot of unreasonable data).
- Change the x and y range to both [0,1] via Gnuplot's *Axis* menu.
- Set the *Data Style* to *Points* in the *Styles* menu.
- Press the *Replot* button.
- Et voilà.



The x-axis shows the clearness index k_t, the y-axis shows the diffuse fraction G_{dh}/G_h for the file `meteo82.dat`.

Second step Run the INSEL model which plotted the correlations again, this time keeping the

`insel.gpl` file (by renaming and copying it from the hidden application data directory to a destination of your choice – let's say `ktRelations.dat`).

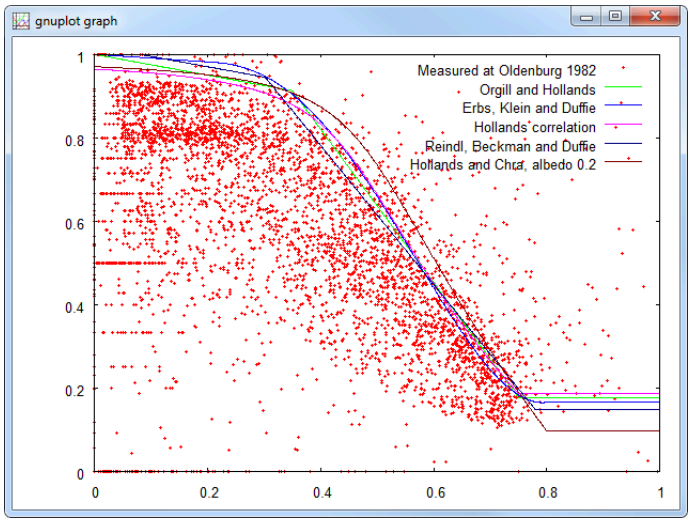
Third step

Now write a Gnuplot file. As starting point you can copy `insel.gnu` from the hidden application data directory. Rename the file and make some changes. For instance:

```
# Comment: We choose the filename ktExample.gnu
set autoscale xy
set data style lines
set nolabel
set data style lines
set pointsize 0.4
set xrange [0:1]
set yrange [0:1]
plot "C:/Temp/ktDataOL.dat"    using 1:02\
        title "Measured at Oldenburg 1982" with points,\
     "C:/Temp/ktRelations.dat" using 1:02\
        title "Orgill and Hollands",\
     "C:/Temp/ktRelations.dat" using 1:03\
        title "Erbs, Klein and Duffie",\
     "C:/Temp/ktRelations.dat" using 1:04\
        title "Hollands correlation",\
      "C:/Temp/ktRelations.dat" using 1:05\
         title "Reindl, Beckman and Duffie",\
      "C:/Temp/ktRelations.dat" using 1:06\
         title "Hollands and Chra, albedo 0.2"
```

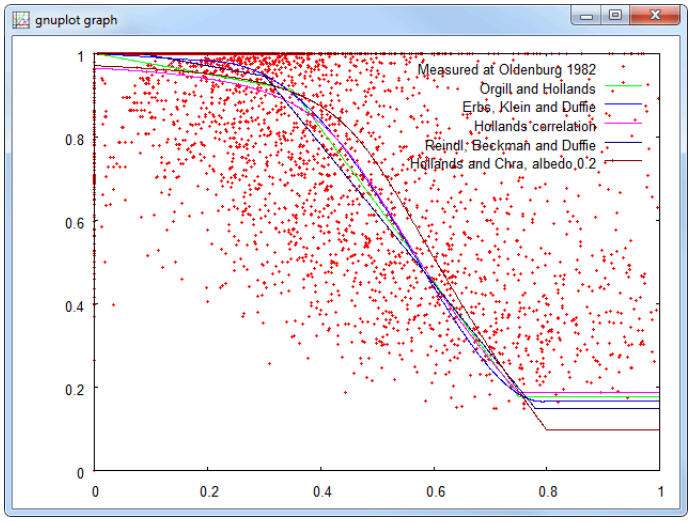
Last step

Type `load 'ktExample.gnu` (or how ever you named the file) at the Gnuplot prompt and you're done.



The x-axis shows the clearness index, the y-axis shows the diffuse fraction.

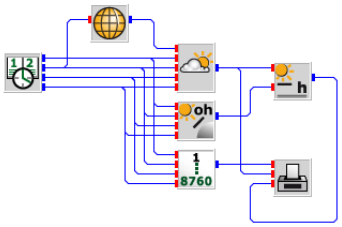
The correlations all seem to overestimate the measured Oldenburg data. But a look at a data set measured by the International Energy Agency in Uccle, Belgium 1960 shows that this is probably a problem of the Oldenburg data.

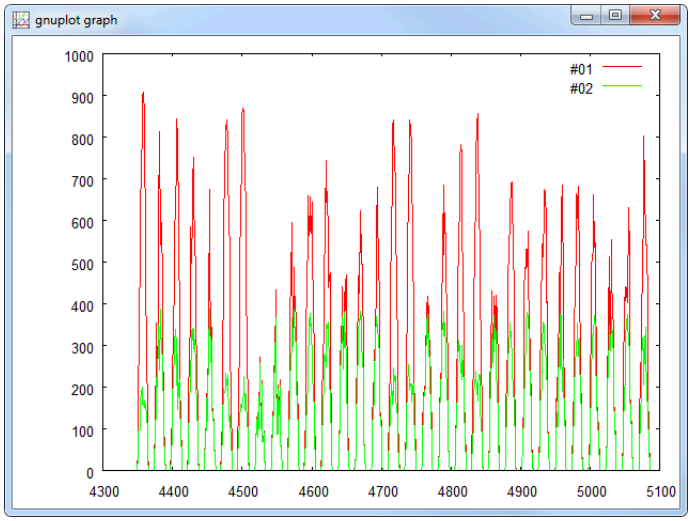


We are now ready to use the correlations to separate the global radiation time series of our earlier exercises.

Exercise 6.12 Plot a time series of global and diffuse radiation together in one graph.

Solution





The x-axis shows the hour of the year, the y-axis shows synthetic hourly means of global irradiance on a horizontal plane (red) and diffuse irradiance on a horizontal plane (green), both in $\mathrm{W\,m^{-2}}$.

6.4 Radiation on tilted surfaces

It is now a simple task to convert the horizontal data to tilted on a surface of any orientation, like vertical façades, for instance. With help of the SUNAE block, which can calculate the position of the Sun in two different coordinate systems (we had used it in Module , page 35f already) it is easily possible to calculate the global irradiance on one- or two-axis tracking systems.

The INSEL block which contains different algorithms for the conversion of horizontal data to tilted is named GH2GT (read: global radiation horizontal to global radiation tilted). As mentioned earlier, the difference between the model lies in the many ways how the diffuse fraction can be handled. The simple Liu and Jordan model assumes an isotropic distribution over the complete sky dome, others – like the Hay model – assume a brightening of the horizon band and the circumsolar region.

Exercise 6.13 Convert the time series of hourly global and diffuse radiation from your previous applications to

- a surface, facing the equator with a tilt angle equal to the locations latitude $\varphi \pm 15°$ if $|\lambda| > 30°$,
- a surface with the same tilt angle but with azimuth tracking,
- a surface with two-axis tracking.

Plot the daily means. In all three cases calculate the annual gain in kWh m^{-2} a^{-1} and in percent.

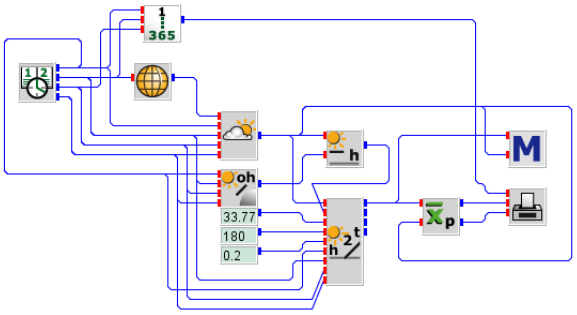
Solution

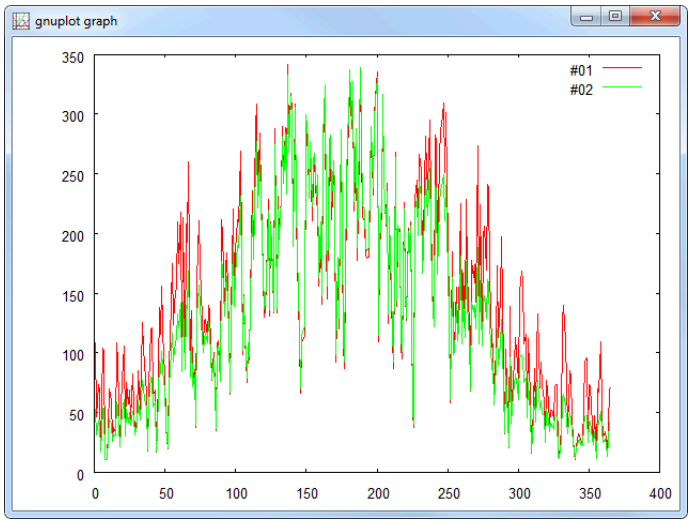
These are the results in the overview:

```
Gain:  10.28 %
Gain: 130.76 kWh   Total: 1272.36 kWh

Gain:  24.84 %
Gain: 377.35 kWh   Total: 1518.95 kWh

Gain:  28.66 %
Gain: 458.56 kWh   Total: 1600.16 kWh
```

Fixed tilted surface

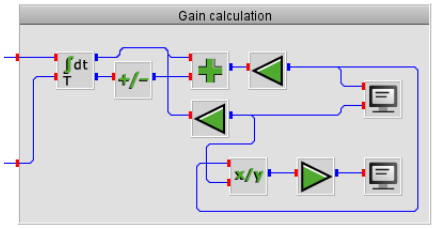
The calulation of the radiation data is straight-forward.





The x-coordinate is the hour of the year, the y-coordinate shows the global radiation horizontal (green) and tilted (blue) in W m^{-2}. The annual gain against the horizontal data is 131 kWh or 10 %, again for Stuttgart, Germany.

The calculation of the cumulated energies and the percental gain is in the macro:



Inputs are the global radiation tilted and horizontal. Both are cumulated over the whole year. The horizontal part is subtracted from the tilted radiation and divided by 1000 by the ATT block. Since the used time step is one hour we can interpret the radiation data in $\mathrm{W\,m^{-2}}$ now in $\mathrm{kWh\,m^{-2}}$.

Before we can calculate the percental gain the tilted radiation is divided by a factor of 1000 and then divided into the horizontal sum. The GAIN block converts the normalized value to per cent by multiplication of 100.

Finally two SCREEN blocks are used to display the figures. We used the format strings

```
(''Gain:'',F7.2,'' kWh   '',''Total: '',F7.2,'' kWh'')
(''Gain:'',F7.2,'' %'')
```

Please note again that the quotes in the Fortran format string are two single quotes and not one double-quote.

One-axis tracking

The only difference to the first case is the use of a SUNAE block for the calculation of the necessary surface azimuth orientation. The second output of SUNAE is used for the azimuth input of GH2GT (instead of the 180 degrees constant). The gain is significant, 377 kWh or 25 %.

Two-axis tracking

Again there is only one small modification compared to the previous case. Now the tilt angle is no longer constant but calculated from elevation α as

$$\beta = 90 - \alpha$$

The gain is not very much higher than in the one-axis-tracking case, 459 kWh or 28.7 %. ■

Division by $\cos\theta_z$

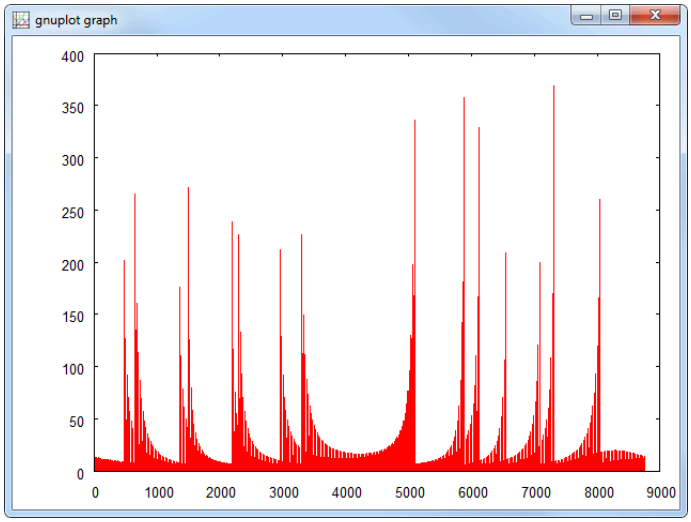
There is one point in the conversion of horizontal data to tilted which should be mentioned here. As was said before, the beam fraction of the horizontal radiation G_{bh} is converted to the tilted radiation G_{bt} by the pure geometric formula

$$G_{\mathrm{bt}} = G_{\mathrm{bh}} \frac{\cos\theta}{\cos\theta_z}$$

where θ denotes the incidence angle between the Sun ray and the receiver's normal direction and θ_z is the zenith angle. In the continuous time, $\cos\theta_z$ will be zero at sunrise

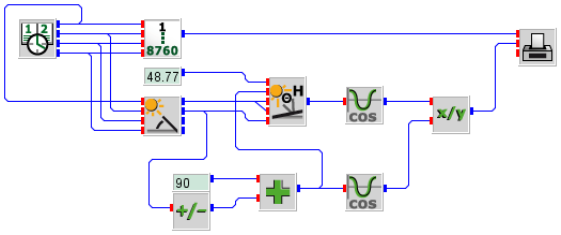
and sunset – this is a problem. INSEL solves it by not allowing the fraction $\cos\theta/\cos\theta_z$ to be greater than 20.

The following graph shows the strange behavior of $\cos\theta/\cos\theta_z$ for the calculation of a two-axis-tracking system in Stuttgart, Germany:



Rubbish

The time series has been calculated with `rubbish.vseit` in the `examples\tutorial\module6` directory.



Don't trust unchecked results!

The reason why we show this example is that the world of simulation is full of traps and dangers. Whatever you calculate, try to cross-check the plausibility of your results in as many ways as possible. Analyse intermediate results, check whether they make sense or not. Otherwise you are endangered to calculate series of street numbers rather than reasonable results. Hence, learn from this hint to be careful with the trust into your simulation results.

6.5 Ambient temperature time series generation

From former PV Module At our first attempt INSEL displayed the following error:

```
Compiling jakarta6_1.vseit ...
No errors or warnings
Running insel 8.1 ...
LOLP = 33.05 %(t) = 29.81 %(E)
E05246 Block 00057: Too many iterations in routine GENT
E02122 Error:  Unable to generate hourly ambient temperature data
LOLP =  8.88 %(t) =  8.77 %(E)
Normal end of run
```

The first LOLP shows that the result unchanged – cf. page 159. But then the error that the GENGT block is unable to generate hourly ambient temperature data leads to the termination of the simulation model.

Two remarks can be made to this problem.

Remark 1 First, sometimes unexpected things occur in the computer world. Swallow this bitter pill as a general experience – that's life. But what happened and how can we help us out?

Well, the generation of time series in INSEL is naturally based on a random number generator. It is possible to initialize this generator by a parameter – in most cases a value of 4711 is chosen as default in INSEL. So, one idea is to change this value to 4712, for instance. But in this case it doesn't help.

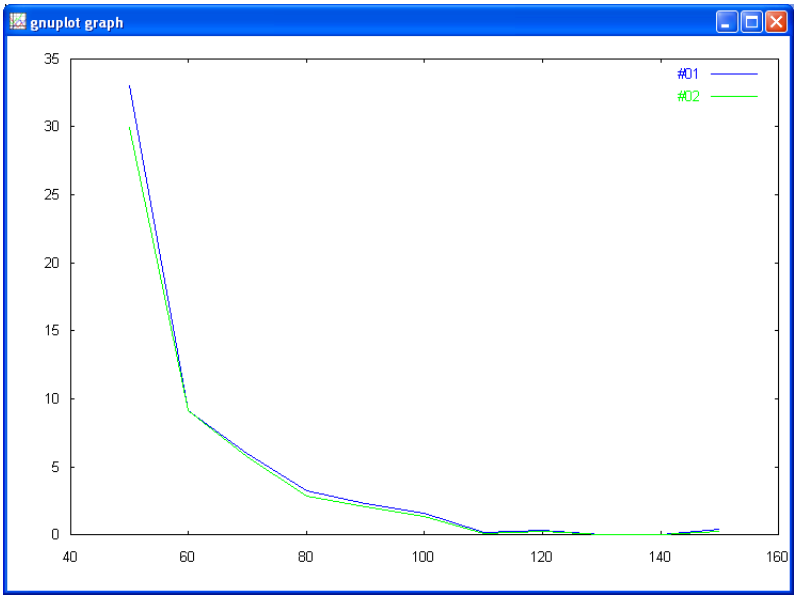
Remark 2 Whenever the GENGT block sees a new month and a new monthly mean temperature value on its input signals, the block starts the synthesis process of hourly data for the complete month. Then the monthly mean value which arises from the stochastic process is calculated. It will practically never be the same value as the monthly mean on the block's input. Hence, the block has to tolerate some deviation between the two.

In case of the GENGT block this value is 2 kelvin, by default. If the deviation is higher the block makes a completely new attempt to generate the temperature time series. If the deviation is still higher, a third attempt is made. And so on. In order to avoid an endless loop after a certain number (100 by default) of attempts the algorithm gives in, stops the stochastic process, and displays the error message which we have seen before. Let us try and increase the tolerance parameter from 2 to 3 kelvin.

And bingo!

Remark 2 Second, why does the GENGT block try to generate new ambient temperature data at all? As mentioned before the GENGT memorises only the month of hourly data, which is calculated when the month input changes. Since we run the simulation model with three different battery sizes the CLOCK block is executed three times and makes changes from month 12 to month 1 two times. Therefore, three completely different years are simulated.

Actually this is not what we want, since we would like to see the impact of the battery size on the load coverage. If we run the simulation model with an increment of 10, i. e., vary the battery sizes from 50 to 150 cells in series by ten instead of 50 we get this graph

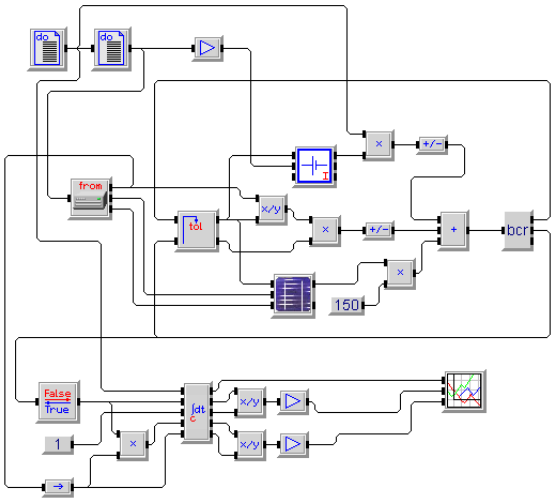


and see that the LOLP does not go down smoothly but in small ups and downs. At 120 and 150 battery cells in parallel the LOLP goes up even. The reason is that we calculate the LOLP under different meteorological conditions which does not make much sense.

Therefore, we should separate the time series generation process from the parameter variation. Hence, we run one year and save the meteorological data to a file. Then in the parameter variation process we read in this file instead of generating new weather data for each setting. In addition, it is practical to save the load data in this file, too. This will save some execution time and simplifies the model to some extent. It is best to use our former application `jakarta4.vee` and just paste in a WRITE block.

The model for the final parameter variation looks like this

jakarta7.vee


do
do
from
tol
x/y
+/-
dcr
150
False
True
1
idt
c

7 :: Photovoltaics

During the course of this Tutorial several aspects of photovoltaic simulation have been covered already. In Modules and the PVI block has been introduced and examples like plotting I-V curves and module temperature profiles were explained. The topic of maximum power point trackers has been touched in Module in the context of Loop blocks.

In this module some typical tasks will be presented which occur frequently in the project work of a photovoltaic engineer. Let's start with a simple simulation model of a grid-connected PV generator.

7.1 Grid-connected PV generators

The main components of a grid-connected PV system are

- PV generator
- Maximum power point tracker
- Inverter
- Weather, i. e., time series of global irradiance, ambient temperature and sometimes wind speed

Of course, weather is not really a component, but from INSEL's point of view there is no principal difference between components and other things which influence the performance of components – they are all just blocks. Concerning the weather, we will keep things simple for a start, because the topic has already been discussed in detail in Module . So let us assume the location of our first investigation is Oldenburg in Germany and our generator is orientated towards south with a tilt angle of 70 degrees. In this case, we may use the weather file `meteo82.dat` from Module and simply read in the required meteorological data.

The temporal resolution of the data is one hour which is a typical time step for PV simulations. The length of the data is one year which should be the minimum for the calculation of the energetic performance of PV systems. Since our aim is to analyze some system performance aspects in detail, we use a READD block so that we can directly access specific days, weeks or months comfortably – this method has been presented in Module , page 56 f.

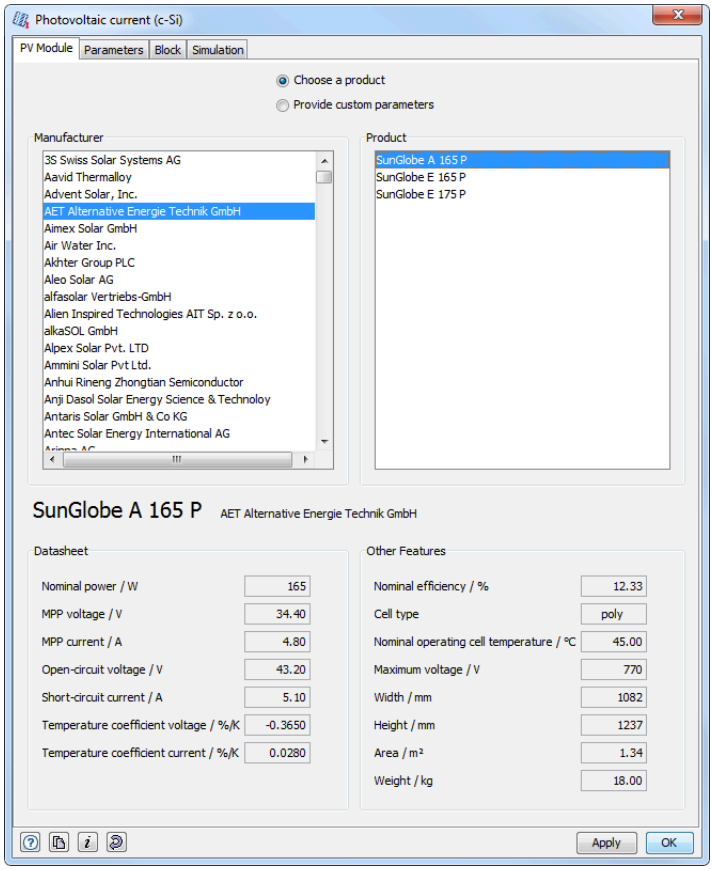
The next step is to decide which PV module we are going to use for our first example. Usually PV cells and modules are simulated in INSEL by using the PVI block with the underlying two-diode model. The main problem here is to find a set of parameters for a specific module.

INSEL provides different methods to get access to these parameters. The most convenient way is to use one of the modules in the INSEL data base which includes

about 20 000 parameter sets. If a module is not included in the module data base INSEL provides methods to determine its parameter set – we will come back to these methods later. For now, let's choose a module which is in the data base.

PV browser

A convenient way to choose a module from the data base is to use the PV module browser of the PVI block's *PV Module* tab.

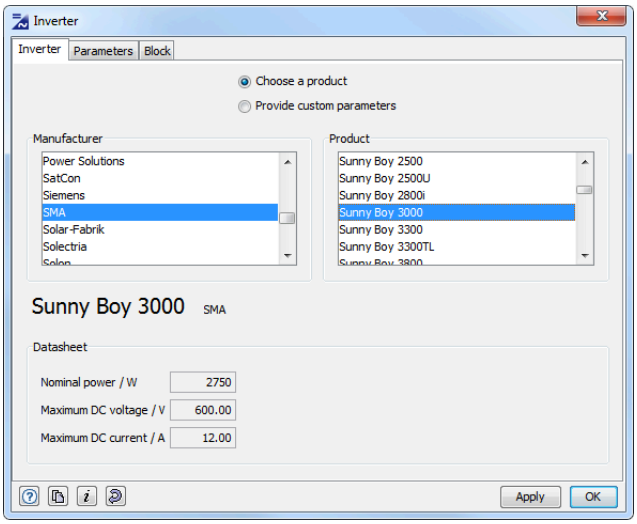


Click the radio button *Choose product*, choose a manufacturer from the *Manufacturer* list and a module from the *Product* list. The browser will show the main data sheet information of the selected PV module in the lower half.

The PV module browser gets its information from the file `pv_modules.fb.bin` in the `data/pv` directory of INSEL 8. It contains the PV module list of the California Energy Commission (CEC), with the model parameters fitted by NREL's System Advisor Model (SAM). The modules of earlier INSEL versions are still available for existing models, but they are not listed in the browser anymore.

Inverter data base

The second important component of a grid-connected PV generator is the inverter. Similar to the PV module data base there is a data base for more than 2 000 market-available inverters, from the same sources. The browser is integrated in the IVP block's *Inverter* pane.

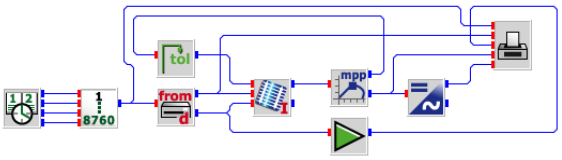


The inverter data are saved in the `data/pv` directory in file `pv_inverters.fb.bin`.

Exercise 7.1 Construct the block diagram for a grid-connected PV generator with LONGi LR5-54HPH-425M modules and the SMA inverter Sunny Boy 5.0 (SB50-1SP-US-41, 240 V), which are selected by default in new PVI and IVP blocks. Make sure that the voltage, current, and power levels fit reasonably. Analyze some details of the system performance.

Solution

The block diagram is rather simple:



The file `meteo82.dat` has been described in detail in Module , page 52. Since we are interested in the global irradiance in south direction at a tilt angle of 70° and the ambient temperature, we need access to the 7th and 10th data column in the file and the Fortran format is `(18X,F5.0,10X,F5.1,26X)` – do you copy?

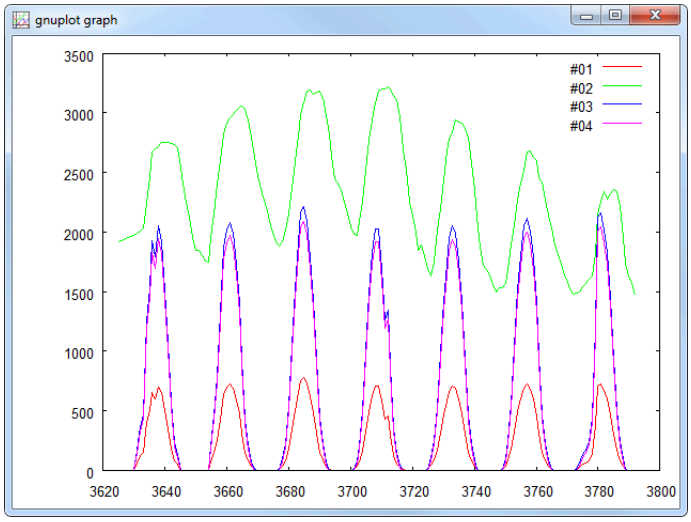
A reasonable PV generator for the Sunny Boy 3000 inverter could have about 3 kW_p and since one module has a nominal power of 165 W a total of 20 modules would result in a

3.3 kW_p generator. The nominal voltage of the module is 34.4 V so that two strings of 10 modules in series each would fit with the voltage range requirement of the inverter.

Do not forget to set the temperature mode of the PVI block to NOCT mode. If you would use IN3 mode, the yield of the generator would be overestimated because of unrealistic low module temperatures.

Since the open-circuit voltage of the module is about 40 V, ten modules in series should never reach 500 V on the DC side and we can set the search interval for the mpp tracker to [0,500]. An accuracy of 0.1 V should be sufficient.

As always, it is recommended to check first if the data from the file are read correctly. The next plot shows one week, starting from the first of June 1982 – nice weather in Oldenburg, quelle surprise!



Since the irradiance and power values are of order 1000, we have multiplied the ambient temperature by a factor of 100 so that the order of the numbers is comparable. We see that the temperature varies between 20 and 30 °C. The radiation data reach values of about 800 $W\,m^{-2}$. This is comparatively low for June. The large tilt angle $\beta = 70°$ is the reason for it.

The reason for the large tilt angle is, that the data were recorded for a self-sufficient laboratory building, the *Energielabor* of the Oldenburg University – very innovative at that time to construct a building without grid connection. When you analyze the performance data over a whole year you will find that the tilt angle is optimized with respect to winter operation and low elevations of the Sun. We will come back to some aspects of the Energielabor building and its technologies later.

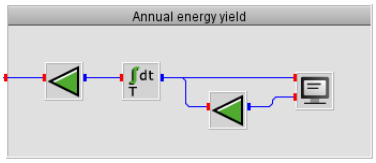
To come back to the plot, the blue and magenta lines are the DC and AC power outputs.

They reach a maximum of about 2000 W, so maybe the inverter is oversized or the PV generator is undersized. A quick view at the maximum DC power during the whole year shows that the highest value is 2917 W. Did you find the same value with the MAXX block? ::

We are now going to investigate some more details of the grid-connected PV generator model.

Exercise 7.2 Calculate the annual energy yield of the system in kWh and in kWh/kW_p.

Solution We have put the calculation into a small macro which uses the AC power output of the IVP block as input.

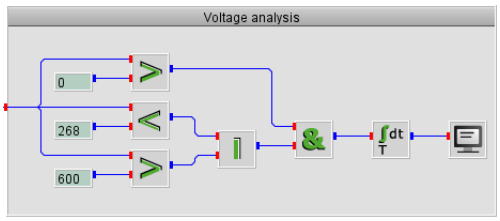


The connection of an attenuator block with a factor 1000 for the conversion of watt to kilowatt hours, a global cumulation block and a screen block shows that the annual energy yield is 2868 kWh. Another attenuator with parameter 3.3 for the nominal power of the generator shows 869 kWh/kWp.

Please observe again that we can simply use the power output of the inverter and think of the watt unit as watt hours, since the simulation time step is one hour. For a different time step an additional attenuator block would be required. For example, if the time step is 15 minutes, we have to divide the power by a factor four in order to have the correct value for the watt hours. In practice, you would probably use the value 4000 for the parameter of the attenuator which converts watt to kilowatt hours. ::

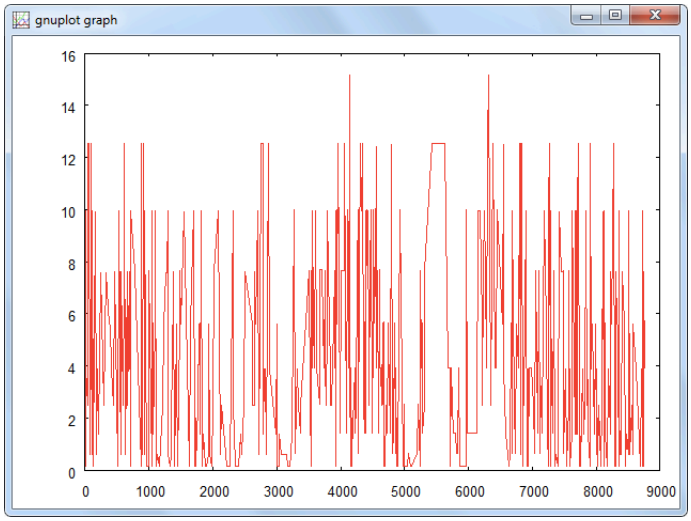
Exercise 7.3 Check how often the maximum power point voltage is outside the inverter's DC voltage range of 268 to 600 volt and how much energy is lost if the corresponding DC power is neglected in the energy cumulation. Do not count the night hours when the MPP voltage is zero.

Solution Again, we present the counter in a little macro.



The required logics is quite straight forward. We need three comparisons for the zero voltage case, the lower voltage limit of 268 V and the upper limit of 600 V (which is never reached), one OR block to distinguish between the upper and lower limit cases and, in addition, one AND block for the night values. A global cumulater block CUM is used to count the outside range cases and find a value of 464 occurrences during daytime and 4751 with the night cases.

In order to check that the model calculates the DC power in the low voltage range correctly, we have analyzed the DC power output for the cases where the voltage is less than 268 volt. The graph shows the result filtered through an IF block:



The highest values are in the 10 W range and therefore less than the self consumption of the inverter. They sum up to 2.1 kWh which is absolutely negligible.

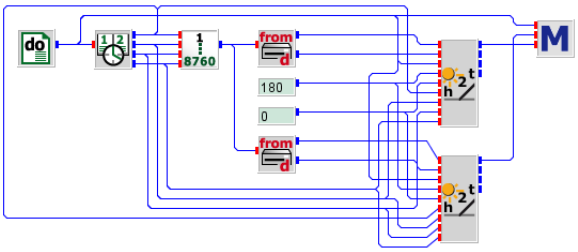
Conclusion The restriction to the voltage level of an inverter in the simulation of a grid- connected PV generator can be neglected, provided that the system is properly dimensioned. However, if a system is designed at the voltage limit of the inverter it is possible to quantify the losses as a function of the PV generator size easily.

7.2 Optimum tilt angle

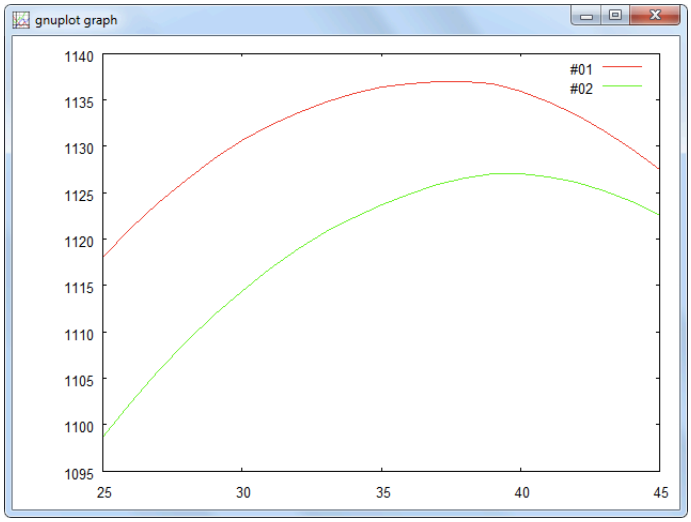
We have started the discussion about grid-connected PV with the data set measured in Oldenburg at a tilt angle of 70 degrees towards south. This is far from the optimum regarding annual electricity production.

Exercise 7.4 Calculate the optimum tilt angle for this location in Germany (latitude 53.133 degrees north, longitude 8.217 degrees east, time zone 23).

Solution We need to read the global and diffuse horizontal radiation data from `meteo82.dat` (and `meteo83.dat` for comparison). The Fortran format string to read these data is `(8X,2F5.0,46X)`. We assume that the maximum lies between 25 and 45 degrees. So, we let a DO block vary the tilt angle in this range in steps of one degree. The rest should be clear.



The routing is a bit fancy. The macro in the upper right corner contains blocks for some output, i. e., this graph:

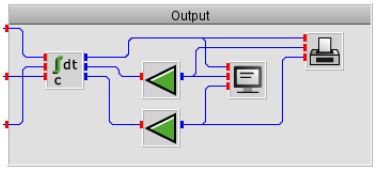


and this table:

```
25.000    1118.070    1098.687
26.000    1121.154    1102.355
27.000    1123.917    1105.800
28.000    1126.400    1108.936
29.000    1128.609    1111.791
30.000    1130.576    1114.401
31.000    1132.248    1116.796
32.000    1133.634    1118.946
33.000    1134.796    1120.762
34.000    1135.715    1122.342
```

```
35.000    1136.364    1123.676
36.000    1136.786    1124.862
37.000    1137.013    1125.833
38.000    1136.959    1126.593
39.000    1136.681    1127.062
40.000    1135.881    1127.030
41.000    1134.784    1126.710
42.000    1133.391    1126.105
43.000    1131.705    1125.215
44.000    1129.727    1124.042
45.000    1127.454    1122.584
```

This is the macro:



The result for the optimum tilt angle is 37° with 1137 kWh/m^2 and 39° with 1127 kWh/m^2, respectively. It can be observed that the maxima are rather flat, i. e., there are only about four kilowatt hours less in a range of ±5°, which corresponds to approximately 0.35 % less energy yield.

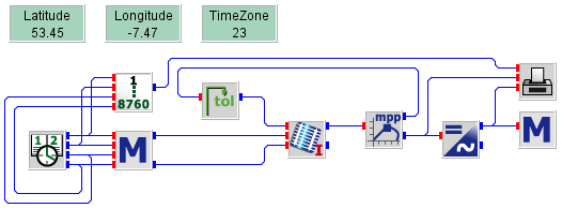
If you have studied Module on solar radiation, you are now ready to simulate the performance of grid-connected PV generators "from scratch," i. e., without the necessity of measured data time series in hourly resolution. The only requirement is monthly means of radiation and temperature data, which are available for all locations worldwide – in the INSEL weather data base, for instance.

Exercise 7.5 In the previous example we have seen the optimum tilt angle for a PV generator in Oldenburg is about 38 degrees. Calculate the annual AC energy yield for the generator which we had used several times in this module, i. e., 20 SunGlobe A 165 P modules and a Sunny Boy 3000 inverter from SMA.

Two hints Since we have constructed all parts which are necessary to solve this task several times already, you may wish to start from the example `nurnberg.vseit` in the `examples\electricity\griConnectedPV` directory.

When you use the MTM browser you will find that the location Oldenburg itself is not available. Aurich is quite close to Oldenburg.

Solution Only a few things need to be modified when you start from `nurnberg.vseit`.

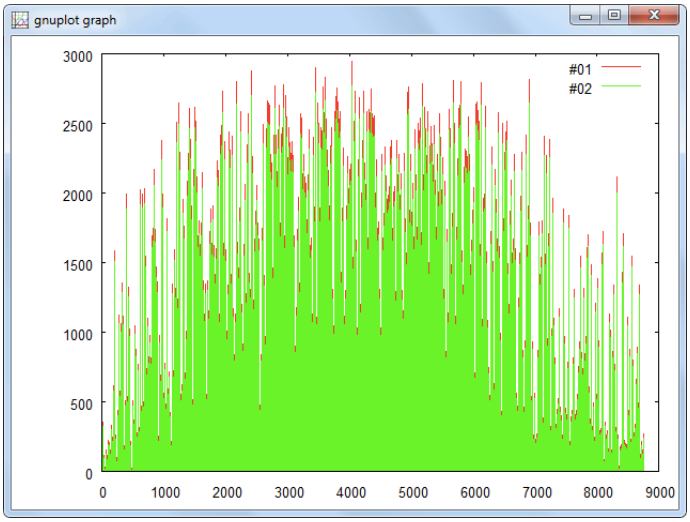


Radiation and temperature data: Change the location of the MTM browser from Nurnberg to Aurich. Since the latitude, longitude and time zone parameters are required three times by the blocks GOH, GENGT, and GH2GT, we have used three global constants from the *Mathematics* > *Constants* category to define *Latitude*, *Longitude* and *TimeZone* and set the values to the data displayed by the MTM browser – remember that eastern longitudes require a minus sign.

PV generator settings: Browse to the SunGlobe A 165 P module and set the number of modules in series and in parallel in the PVI block's *Simulation* pane.

Inverter: Browse to the SMA Sunny Boy 3000 inverter.

Energy cumulation: We can use the macro from page 128.



The result is a total AC energy yield of 3451 kWh or 1046 kWh/kWp. Compared to the values 2868 kWh or 869 kWh/kWp that we had calculated for the 70 degrees tilt angle case, this is a gain of twenty per cent.

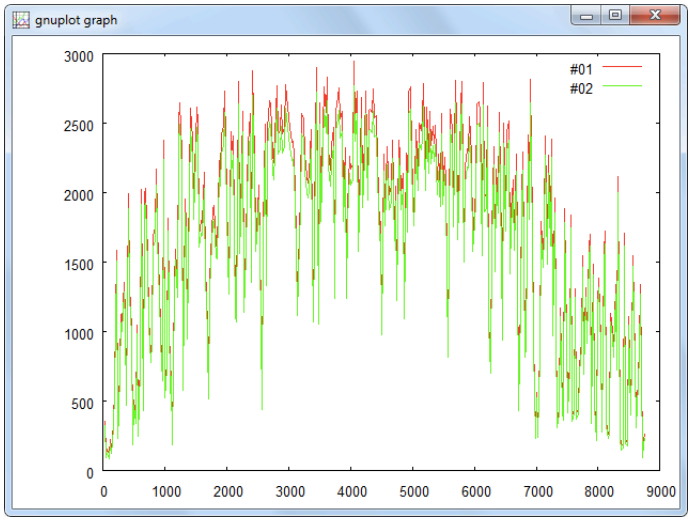
Plenty of ink, not much information

The last graph, which displayed the hourly time series of DC and AC energy production of a PV generator near Oldenburg, Germany, wastes a lot of green ink. Due to the nearly

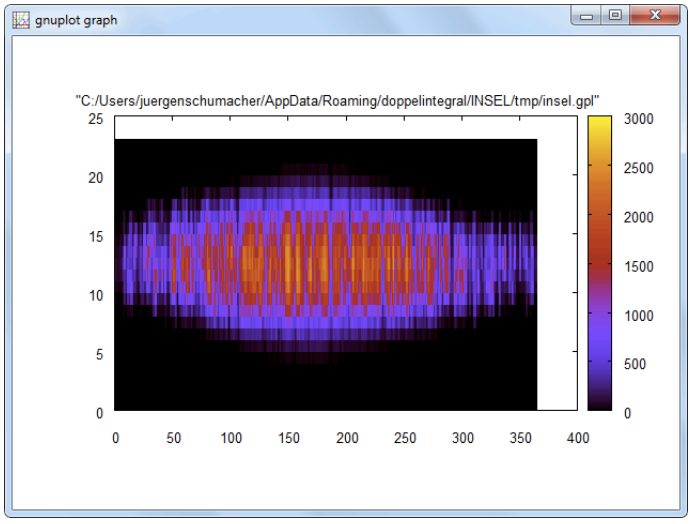
twenty thousand data points (two times 8760 hours of the year 1982) it is even hard to distinguish between the DC and the AC data.

Exercise 7.6 Replot the graph with the daily DC and AC power peaks only.

Solution A MAXXP block with $p = 24$ does the job.



Another option to plot a large amount of data is a carpet plot, as implemented in block PLOTPMC:



Explain and mprove appearance.

7.3 Parameter identification methods for PV modules

7.4 Module mismatch and shading problems

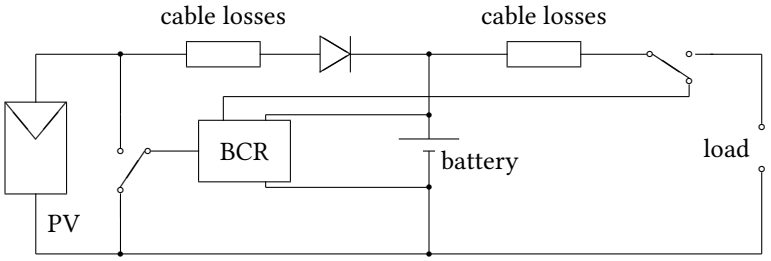
7.5 Thin-film modules

7.6 Stand-alone PV systems

Einleitung überarbeiten. In the concrete application in Oldenburg there is a PV installation with a tilt angle of 70 degrees towards south. The reason behind this tilt angle is that the building at the University of Oldenburg (the so-called Energielabor) used to be an autonomous system at the time. In this case, the optimum tilt angle is not defined through maximum energy output but through the storage system, a lead-acid battery in this case. The goal is to minimise the time when the battery is empty.

Let us have a closer look at the simulation methods required to calculate the performance of systems, which have no grid access. Readers not interested in stand-alone PV systems can proceed to the next section starting on page **??**, although some of the topics presented may be of interest nevertheless. The most typical autonomous PV system consists of a PV generator, usually directly coupled to a battery via a battery charge regulator and several loads, of course. The following sketch depicts the usual circuit.

Bild verschönern



As can be seen, the components PV generator, battery charge regulator, battery and load are connected in parallel and therefore – according to Kirchhoff's rules and neglecting cable losses and the blocking diode – must share the same voltage, or, in other words, the sum of all currents must be zero at all times. This forms the basic idea of the INSEL simulation model of stand-alone PV-battery systems.

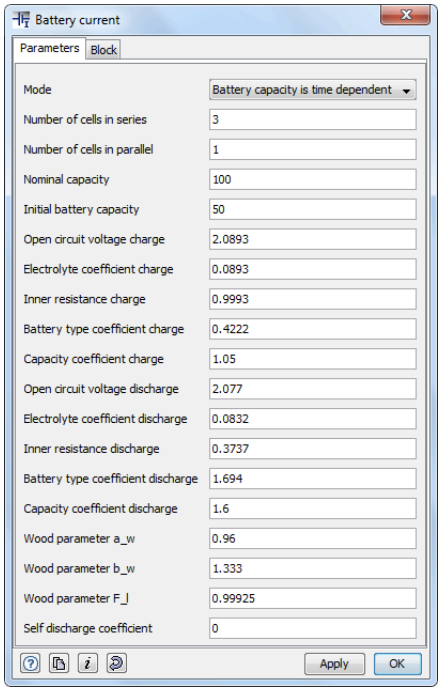
We will look at the components one by one, let's begin with the battery.

7.6.1 Batteries in INSEL

The beginning of such a section is usually something like: Batteries are electrochemical devices which can be charged and discharged. They work as follows. . . We would like to suggest a different approach.

Hyman model

In INSEL there is a simulation model for batteries, the so-called Hyman model. It is implemented in two blocks named BTI and BTV and allows for the calculation of battery current and battery voltage, respectively. This is the BTI entity editor:



A short look into the block reference – via a click on the *Help* button – shows that the block requires two inputs, i. e., voltage and time – the latter as always in INSEL as increasing time measured in seconds – and an optional input which depends on the block's capacity mode. The outputs are the battery current I and the actual battery capacity Q. The third output is the charge efficiency η which has a meaningful value only in case of charging, of course. We come back to the other capacity mode shortly.

The performance of the block depends on the actual values of its parameters, 19 in this case. You can see that two times five parameters have to be known, one set for the charging case and one set for the discharge case. For instance, the open circuit voltage parameter is of order 2 V, which is a very typical value for a single lead-acid battery cell. At some later stage, you may want to know how the values can be calculated for a real

battery which is not part of the – so far tiny – battery parameter data in INSEL. For the time being just accept that the displayed values have been determined for a VARTA block battery Vb 624 with a nominal capacity of 100 Ah.

Please observe that the nominal capacity is just one parameter in the general BTI block and can be adapted freely, as can the initial capacity value. The block can be used to simulate series and parallel connections of individual cells, which lead to a multiplication of the cell voltage in the series case and a multiplication of the current in case of parallel connections. From a numerical point of view there is no problem in sight. However, in reality connecting batteries in parallel must be treated with care, since defects in one or more cells can lead to unwanted discharging of the complete battery bank – but that is a different story.

Three parameters are connected with the name Wood to be used with Wood's charge efficiency model. Some further details to Wood's model can be found in the block reference manual. The same applies to the self-discharge parameter which ends our short excursion to the BTI block's parameter set.

Now let us play a bit with our new toy to see and understand in more detail how it functions and where the traps and dangers are – and there are quite some.

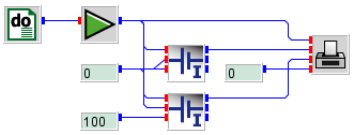
I-V curve

How does the *I-V* characteristic of our battery look like if it is fully charged (capacity 100 Ah) or fully discharged (capacity 0 Ah)? In order to answer these two questions the Capacity input 3 mode is useful because it allows us (similar to the PV module temperature case) to define a value for the actual battery capacity.

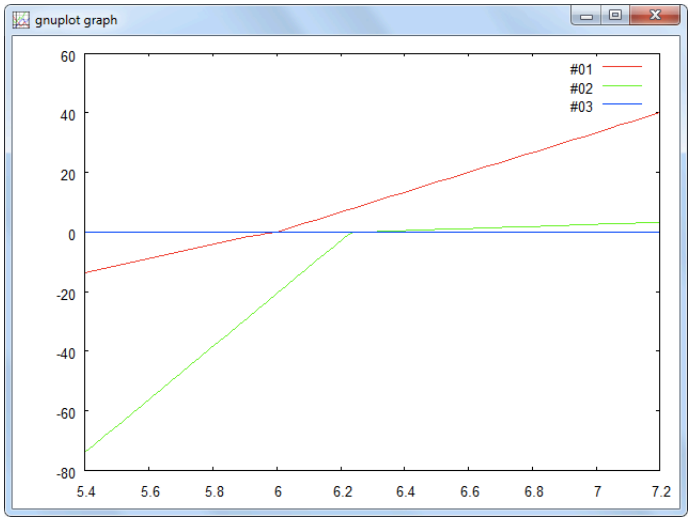
Exercise 7.7

Concerning the voltage range, it is useful to know that a single lead acid battery cell should never be discharged under 1.8 V and battery gassing starts at 2.4 V per cell. Now you!

Solution



Okay, our solution is a bit quick-and-dirty, but it does all we wanted. Did you remember to multiply the voltage range by the number of cells in series? Here comes the graph as it was meant.



Sign convention

The first observation is that positive and negative current values occur. The convention is that for charging positive values and for discharging negative current values are used.

Empty or not?

Second, an empty Vb 624 battery (red curve) can be charged at a maximum current of about 40 A within the recommended voltage range. On the other hand, it can be further discharged at a current of 15 A. How can this be? Is the battery not empty when the capacity is 0 Ah? In fact, it is not and the reason for this is the somewhat strange definition of the actual capacity which can even take negative values down to −60 Ah in case of the Vb 624.

The electrochemical background for this behavior is provided by the so-called Peukert law, which describes how the amount of charge which can be taken from a battery depends on the magnitude of the discharge current. Since this module is not meant as a lecture on electrochemistry we skip any further discussion of this topic, except two remarks.

Two remarks

The first (not too serious) is that you should take negative capacities as an example from real life to the joke about the empty room (Mathematicians like the guy who wrote this module, have a strange kind of humor – so say some): If two people move into an empty room and three come out, one guy has to enter it later, so that the room will be empty again.

The second (more relevant) is that, as a general rule, batteries should never be discharged by more than about 70 % (some say 60, others 80 %) of its nominal capacity. That means, we will construct our simulation models in such a way that negative capacities do not occur – if possible.

Fully charged case

Coming back to the resulting graph: We can see that the fully charged battery (green curve) still accepts small charging currents according to the Hyman model, which would

lead to capacity values higher than nominal capacity. In the BTI and BTV blocks this is suppressed since the Wood model assumes that for a battery at 100 per cent of its nominal capacity the charge efficiency is down to zero.

The last observation from this example is that a fully charged battery can obviously be discharged by a bit less than 80 A, which means that nearly 80 per cent of its capacity can be discharged within one hour. This statement is true only, if we neglect that during this hour the capacity will change continuously.

Discharging batteries

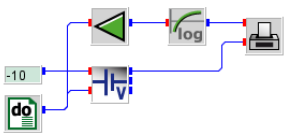
Nominal capacity

But this remark leads us to the definition of the nominal capacity of a battery. It is defined as the amount of charge, measured in ampere hours, which can be extracted from a fully charged battery with a nominal current I_n so that after a nominal time t_n a nominal final discharge voltage $V_\mathrm{d,n}$ is reached – it is not a trivial experiment in real life to determine the nominal capacity of a real battery.

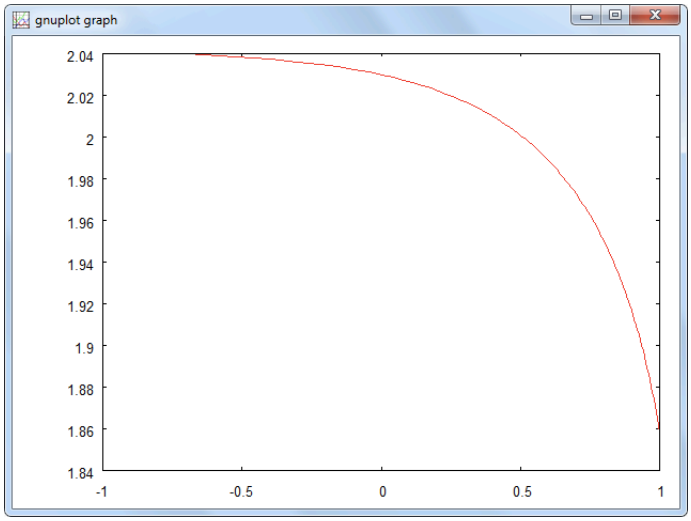
The "usual" definition of the nominal time is $t_\mathrm{n} = 10$ hours, the most common definition of the final discharge voltage is $V_\mathrm{d,n} = 1.85$ V per cell. Hence, a fully charged 100 Ah battery would be empty after 10 hours, if continuously discharged at 10 amps.

Exercise 7.8 Let us do the experiment in INSEL. It is common practice to plot the time axis on a logarithmic scale in such applications.

Solution



The INSEL model is straight forward, maybe except the detail that we have started "recording" time after 6 minutes (360 seconds) rather than confusing the LOG10 block by having to deal with $\log 0$.



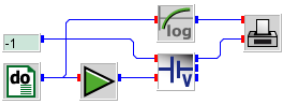
Our result presents a final discharge voltage slightly higher than 1.85 V – we have already talked about error tolerances in simulation models several times.

Serious business?

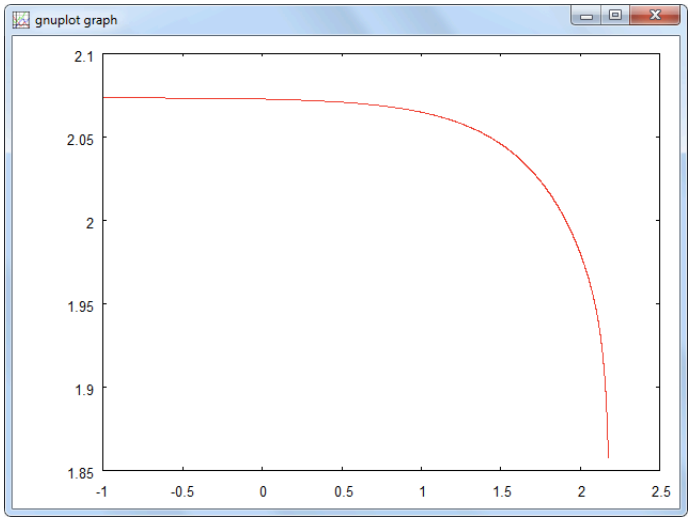
Some battery manufacturers use a nominal time of 100 h for the determination of the nominal capacity of their batteries. Consequently, the nominal discharge current for a 100 Ah battery goes down to 1 A. It follows from the previous discussion that the battery voltage at the end will be higher than 1.85 V after 100 hours of discharging. What does this mean?

Exercise 7.9 Repeat the discharge experiment and use a discharge current of one ampere instead of ten.

Solution



In this case, we have changed the time step to 0.1 hours (instead of 360 seconds) and used a GAIN block for the simulation time in seconds as required by the BTV block.



The result is that after 100 hours the voltage is down to 1.95 V (remember $\log 100 = 2$) and it is possible to further discharge the battery for another 50 hours before the final discharge voltage limit of 1.85 V is reached. ::

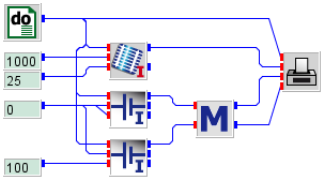
Nominally, yes, but a real battery would be dead after such a deep discharge.
We believe that plenty of batteries in stand-alone systems with charge regulators based only on voltage measurements had a poor lifetime, because small discharge currents have lead to deep discharges of the batteries.

Charging batteries with PV modules

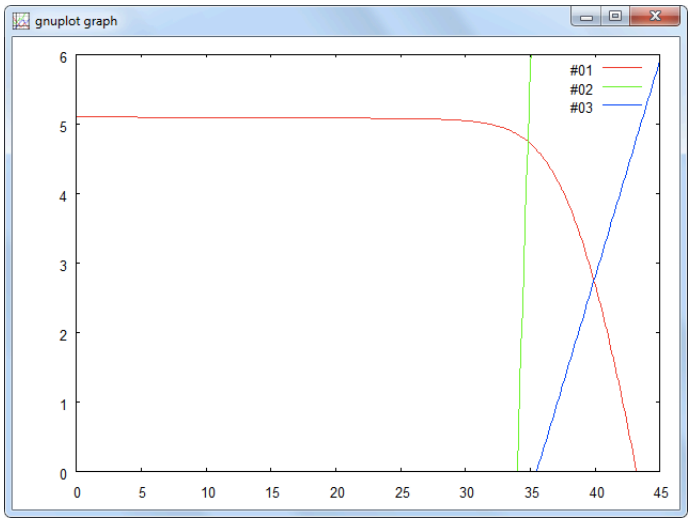
When we want to charge a battery with a PV module or a PV generator, it must be assured that the voltage levels of both battery and PV fit together. For instance, the module which we have used earlier – the SunGlobe A 165 P – has an open-circuit voltage of 43.2 V with its MPP at 34.4 V. Hence, an ideal battery for this module would have a nominal voltage of approximately 34 V, which can simply be reached by connecting 17 cells in series (numerically easy, in practice not always possible to find on the market).

Exercise 7.10 How do the I-V curves of such a battery (empty, i. e., capacity 0 Ah and/or full i. e., capacity 100 Ah) and one SunGlobe module under standard test conditions (1000 W/m^2, 25 °C, AM 1.5) look like? Plot it!

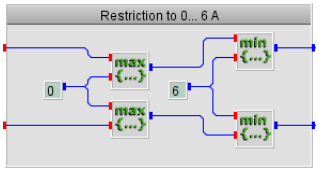
Solution This is what we did to demonstrate the idea:



We used an empty and a fully charged battery with identical parameters and suppressed unwanted data via a small macro. This is the result:



and this is the macro which simply restricts the battery current values to the interval between zero and six ampere.

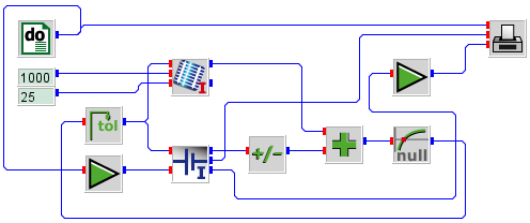


In conclusion, the battery will be charged in a voltage range approximately between 35 and 40 V with currents in the range up to 5 A. This means that in one hour approximately 200 Wh can be charged into the battery. Since the battery has a nominal capacity of 100 Ah and a nominal voltage of $17 \times 2 = 34$ V the nominal energy content of the fully charged battery is 3.4 kWh. This means that it will take approximately $3.4/0.2 = 17$ h to charge the battery with one module illuminated under STC. ::

The numerical problem to solve is to find the intersection between the two actual I-V curves of the PV module and the battery. Thereby, in each time step the battery capacity will increase as a function of the irradiance.

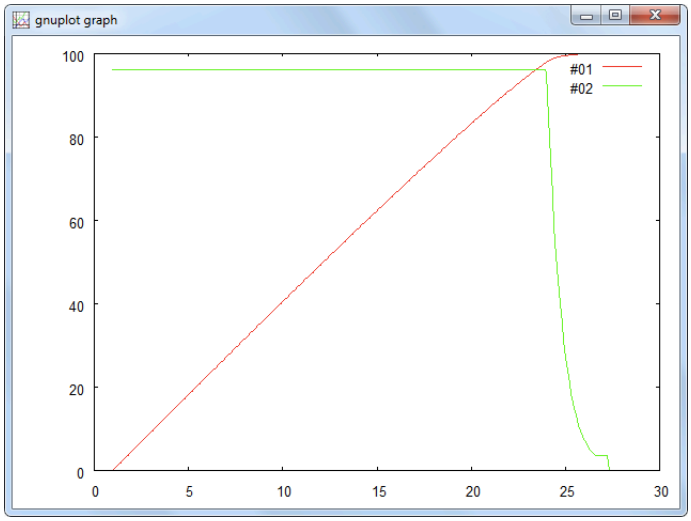
Exercise 7.11 For a first experiment assume that we have the PV module in a laboratory under a solar simulator which radiates 1000 W/m^2 constantly and charges our empty battery, connected in parallel to the PV module. Assume that the module temperature is kept constant at 25 degrees Celsius. In Module the NULL block has been introduced. Can you use it for the battery charge experiment?

Solution



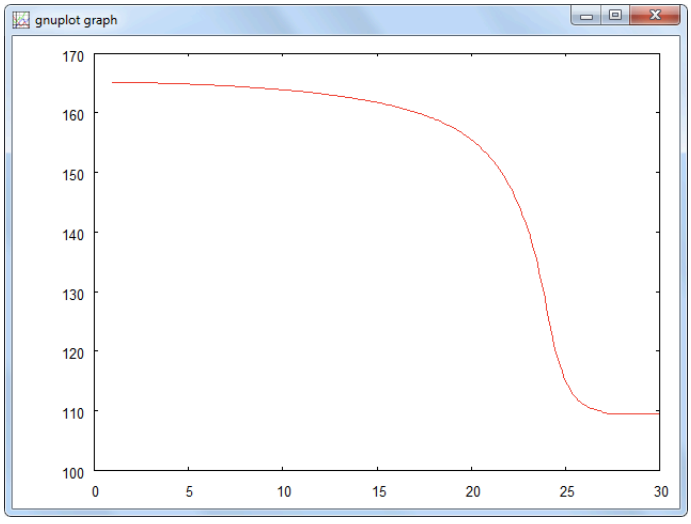
We have set the parameters of the DO block to 1, 30, 0.1 to run through 30 hours. The conversion to the second signal required by the BTI block is done by a GAIN block with parameter 3600.

The example demonstrates the idea that the PV current and the battery current must be equal, or their difference must be zero. This is exactly what the NULL block does, it iterates the output signal, ergo the voltage, in the range specified by its parameters (we have chosen 0 to 50 with a tolerance of 0.001) so that its input becomes zero (or null, in German).



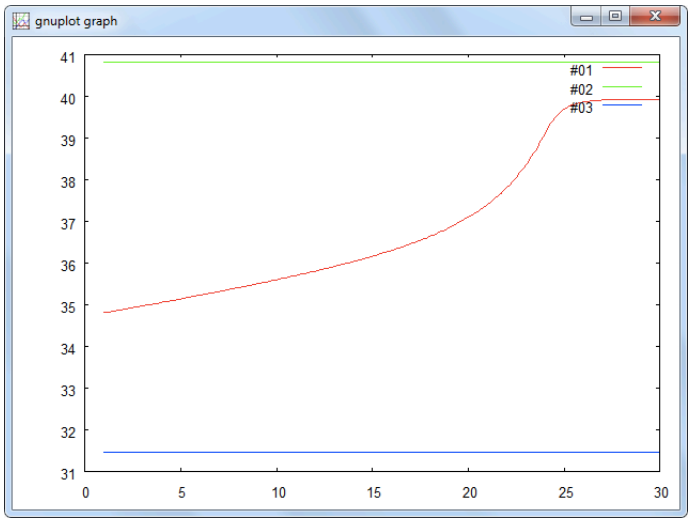
In addition, we have plotted the charge efficiency in per cent. As can be seen from the graph the efficiency is constant in a wide range according to Wood's model, and therefore the increase in capacity is linear. Only when the battery reaches its nominal capacity, the efficiency goes down rapidly, and drops to zero, when the battery is full.

The next graph shows, how the module output power changes with time.



The module power decreases with time, not down to zero but it stays at a value slightly less than 110 W. This means that we simulate this power as charging the battery, but dissipate the energy in the battery in a gassing process.

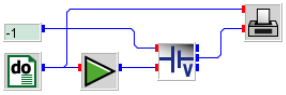
Another interesting quantity to observe is the battery voltage.



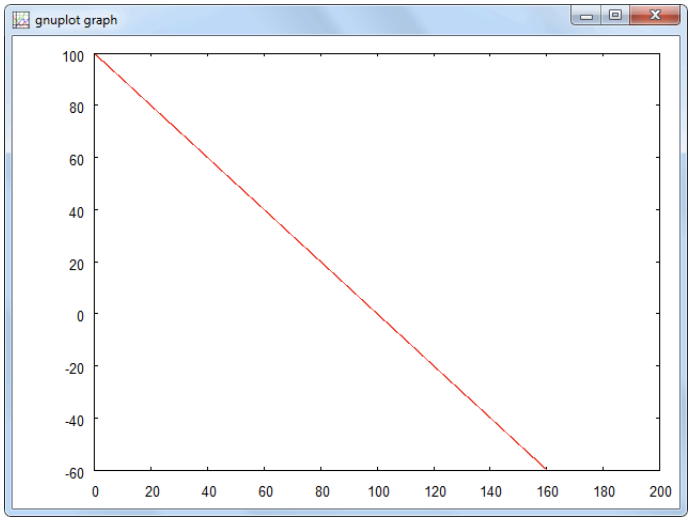
The green and the blue constant lines show the recommended lower (1.85 times 17) and upper (2.4 times 17) voltage limits. In our experiment the voltage is always within these limits, which is good for the battery.

Deep discharging

Let us come back to the discharge process and use a former model again for a small experiment. What happens if we try to discharge the battery longer?



For two hundred hours with one ampere trying to extract 200 Ah out of our 100 Ah battery, for instance.



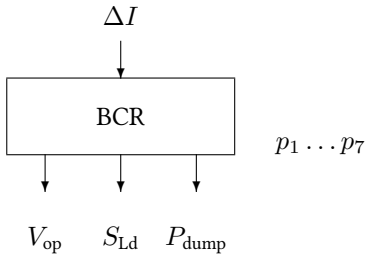
The model in the BTI block lets the capacity go down to minus 60 Ah and then keeps the capacity constant – the level of this value is one of the Hyman parameters, by the way. So, the numerical model is stable, whatever nonsense is done to it.

Unfortunately, real batteries do not have the same behavior, they just flush off. This is the reason why some electronic guys come in and construct controllers like battery charge regulators which try to avoid unregular operations of batteries in real life.

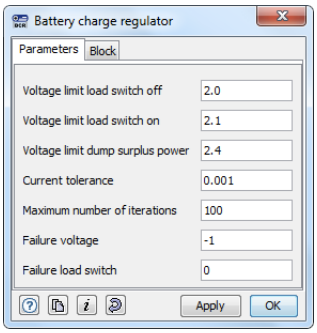
In INSEL there is a block which simulates a battery charge regulator with fixed voltage limits – we have mentioned already that this is not the optimal solution for a BCR in real life. The block, named BCR, is basically a NULL block with additional restrictions. It delivers an indicator, when the load should be switched off in order to avoid deep dischgarge of the battery and it returns a value how much energy is lost in case of gassing.

Have a closer look at the construct (don't worry, we are not going to speak about The Matrix).

The BCR block



The BCR entity comes with a careful default parameter set for a 2 V cell.



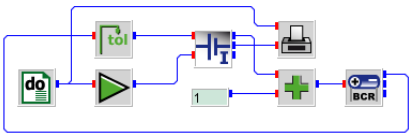
Since BCR is similar to the NULL block it should be clear that it is an L-block and requires a corresponding TOL. Like the NULL block it iterates its output signal V_{op} in such a way that the input ΔI becomes zero, plus minus the value of parameter *Current tolerance*.

In case this cannot be accomplished within the limits specified by the parameters it suggests to switch off the load by setting $S_{\mathrm{Ld}} = 0$ or calculates the amount of power P_{dump} which would better have been dumped before it leads to gassing inside the battery.

When the load should be switched off it is not wanted that the load is switched on again immediately but that the battery is given some time to recover. Therefore, there is a hysteresis before the load switch is set to one again, defined through the parameter *Voltage limit load switch on*. The dump voltage parameter defines the maximum allowed operation voltage for the battery. The other parameters are not so important for the time being. Of course, the maximum number of allowed iterations should be greater than zero.

Now let us rebuild the former example where the battery was discharged constantly with 1 A, but now controlled by a BCR in order to avoid deep discharge.

This is a first attempt to solve the problem.

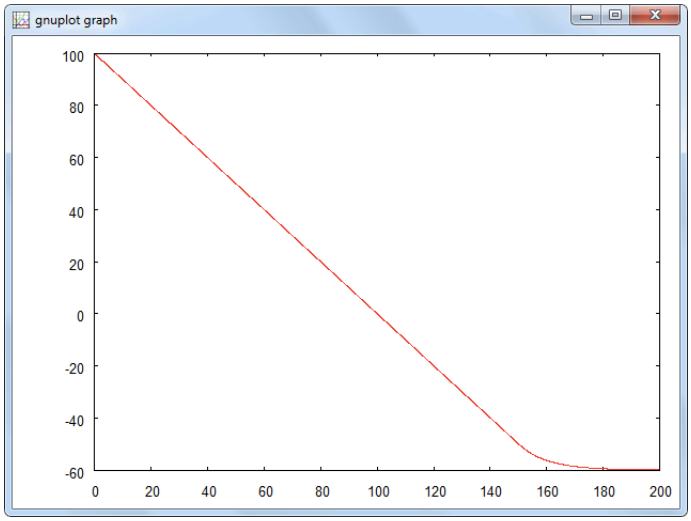


We have changed the default parameter set of the BCR block to 1.85 V for the switch-off voltage, and 1.9 V for the switch-on voltage.

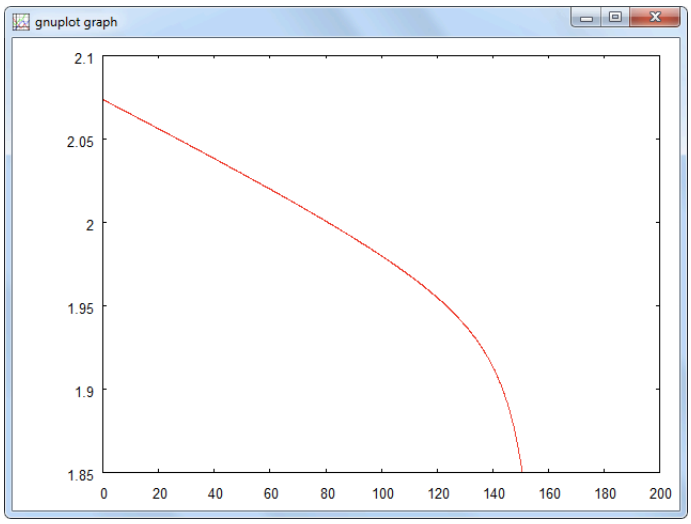
Instead of the connection of a constant one ampere discharge current we are now looking for the correct operating voltage of the battery coupled with a load which constantly tries to discharge one ampere from the battery. Since the battery discharge

current is negative by convention, we have used a positive value for the discharge current of one ampere instead of using −1 A and a CHS block.

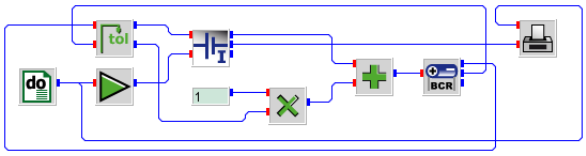
The somehow frustrating result is not much different from our previous example.



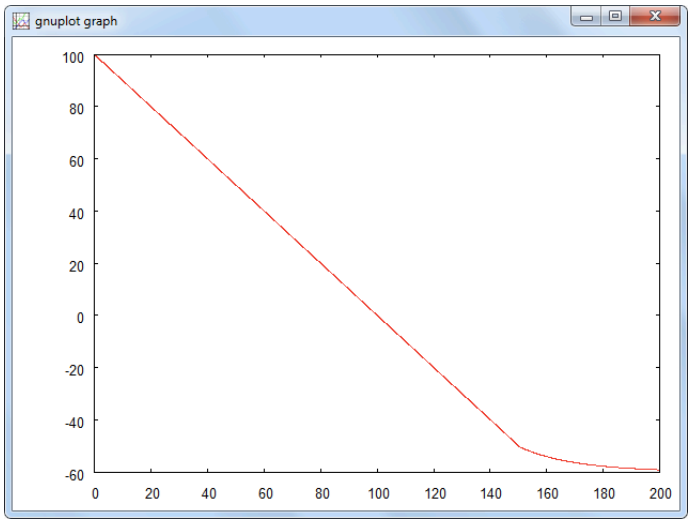
A small difference occurs only at the end of the discharge process when we reach the −60 Ah capacity. The reason for the difference is that – in contrast to the previous application – we now contol the lower voltage limit and have restricted it to 1.85 V, demonstrated by the next graph.



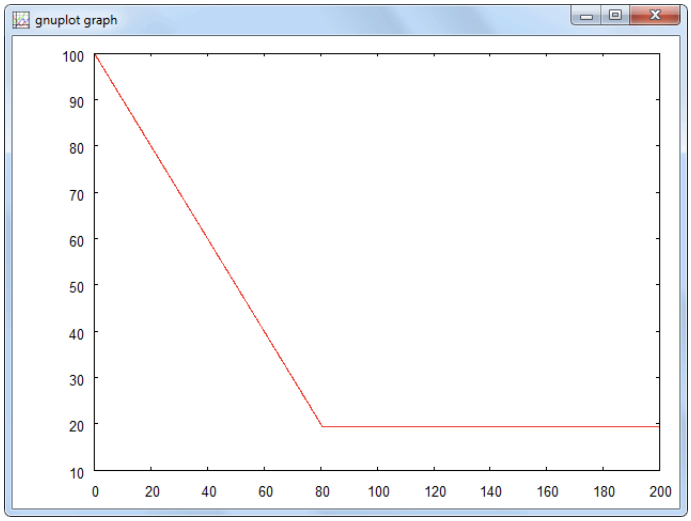
However, the battery is still deep discharged in the model. Did you already find out, why? Yes, the reason is that we have calculated that we should switch off the load, but we have not done it although the BCR block informed us that it would be wise to do so (by setting the second output of the BCR and TOL blocks to zero). Hence, we should multiply this information into the current balance before the BCR decides how to handle the situation.



But, what a frustration! The result again remains nearly unchanged.



The simple reason is that very small discharge currents can lead to deep discharging of batteries, if only controlled by the battery voltage. In consequence, this means that a switch-off voltage of 1.85 V is too high. So, what about a 2 V minimum?



Now everything seems fine. The battery is not discharged below 80 % of its nominal capacity. But what a birth! The nearness of reality – nothing is simple. But now we are – almost – prepared for a realistic simulation of a stand-alone PV system.

7.6.2 Implementation of load profiles

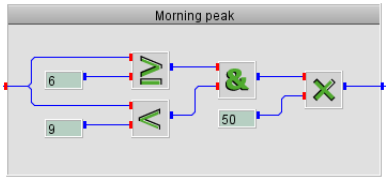
The main influence on the performance of any stand-alone PV system is how users of the system use the system. Huge efforts are done world-wide to find out, how the energy requirements of users or customers are.

In this part of the Tutorial we will discuss some of the basic techniques how load profiles can be implemented in INSEL. Of course, these profiles depend extremely on the location. For a small case study, let's assume we want to design a PV system for a – let's say – community somewhere near the equator. They have a lot of available solar irradiance during the day, that's for sure. And they have some basic load during the day for equipment like computers and stuff and definitely some consumers like lighting in the evening hours – this is where the batteries come in at latest.

Exercise 7.12 Let's assume the following electricity demand profile: a basic load of 10 kW, a morning peak of 50 kW between 6 and 9 o'clock, a noon peak of 30 kW between 12 and 14 o'clock, and an evening peak of 100 kW between 18 and 23 o'clock. At the weekend – say Saturdays and Sundays – we give half the people off to stay on the beach or anywhere. This means, we divide the peaks by two but leave the basic load unchanged. We compensate this with no vacations in the course of the year.

The block diagram is not complicated, but maybe you try it on your own before you carry on reading.

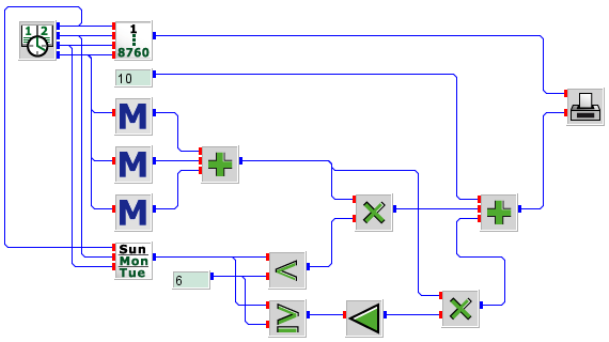
Solution Let us start with one of the peak loads – which are candidates for macros. For example, the morning peak macro



delivers the desired result, i. e., 50 kW from six to nine, else zero.

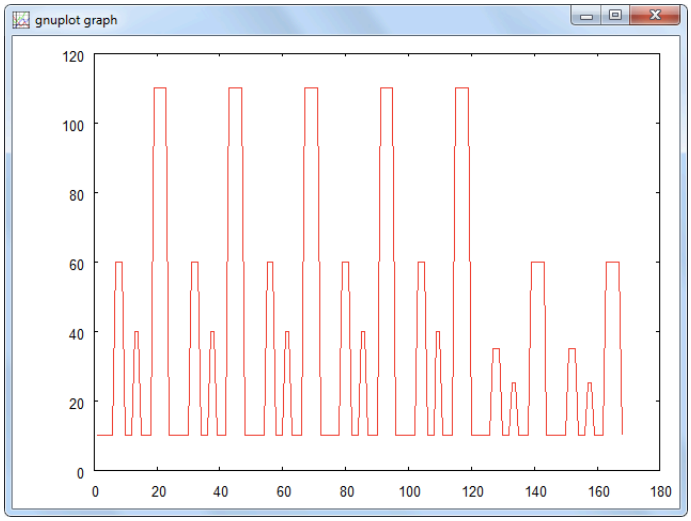
The trick with the weekend case should also be clear now: create disjoint cases, like day of the week less than six and day of the week greater or equal six, handle both cases as required and sum up the results. This is a very old block diagram idea.

The complete block diagram is this:



As you have certainly noticed, our load profile definition was independent of the calendar week, so that it can be described as a function of weekday and time without further complications. But it should be clear how the principle idea can be extended to vacation periods and whatsoever exceptions – just more work, or Gschäft, like the Swabians say.

We used the first week from January 01, 2007, since it practically started with a Monday. We ended up with this graph:



It is easy to find that the mean load is 35.4 kW via an AVE block, for instance. This corresponds to an average daily energy demand of 849 kWh.

In order to design a stand-alone system properly it is necessary to dimension the PV generator and the battery. From the user's point of view both components should be as large as possible since this implies the highest usability. But when economic aspects come in it is clear that reasonable sizes of the components are required. Hence, a compromise between user comfort and costs should be found. It is clear, how costs can be quantified but how can user comfort in a stand-alone PV system be measured?

7.6.3 System sizing

Loss of load probability

One way to quantify the performance of a stand-alone PV system makes use of the *loss of load probability* LOLP. Two different definitions of this parameter exist, one in terms of time intervals

$$\text{LOLP} = \frac{\sum_{T_{\text{LOL}}} \Delta t}{\sum_{T} \Delta t}$$

where the sum in the numerator is taken over all time steps when the load cannot be satisfied and the sum in the denominator is taken over all time steps.

The other definition uses the energy relation

$$\text{LOLP} = \frac{\sum_{T_{\text{LOL}}} P_{\text{L}} \Delta t}{\sum_{T} P_{\text{L}} \Delta t}$$

where P_{L} denotes the required load power during the particular time step.

The complementary value of the loss of load probability is the *load coverage*, defined as

$$\text{LC} = 1 - \text{LOLP}$$

The LOLP can obviously only be calculated through a simulation model.

Let us briefly discuss reasonable starting values for the sizes of PV generator and battery in a stand-alone system.

Battery sizing

Battery sizes are sometimes measured in days during which the load can be fully covered by the battery alone. In our case, a 1-day battery would have a nominal energy content of approximately 850 kWh. As a rule of thumb, a battery size of about 1.5 days is a reasonable starting point. Hence, in our case the battery size should be approximately 1200 kWh.

The next thing to decide is the voltage level at which the system shall be operated. Sine we are trying to design a rather large stand-alone system, let us fix the DC voltage to approximately 240 volt. For simplicity, let us assume that we can upgrade our Vb 624 battery to arbitrary dimensions. One cell has a nominal voltage of 2 V, which implies that we have to connect 120 cells in series. This means that one such block has an energy content of 240 V $\times$ 100 Ah or 24 kWh so that 50 of these blocks are required in parallel.

PV sizing

The question of the PV generator size cannot be answered without fixing a location first. It was already mentioned that we have something close to the equator in mind, why not Jakarta in Indonesia. From the INSEL weather data base we find that Jakarta is located 6.18 degrees south at a longitude of 106.83 degrees east (i. e., -106.83 degrees in INSEL). The time zone is 17. The annual average global radiation on a horizontal plane is 236 W/m^2. Assuming a PV module efficiency of about 15 per cent we will get approximately $236 \times 0.15 \times 24 = 850$ watt hours electricity per square meter and day.

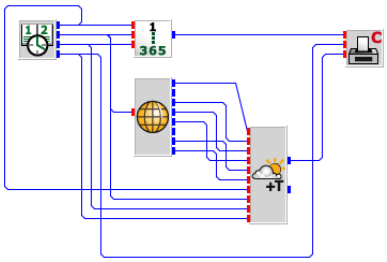
If we further assume that the PV generator should be able to charge the complete battery within one day a PV area of $1200/0.85 \approx 1400$ m^2 is required. When we decide to use the SunGlobe module from the beginning of this Tutorial's Module with a module area of about 1.3 m^2 that gives us round about 1000 modules, or a generator with a peak power of 165 kW.

In a first step, let us assume that all previously defined loads are DC loads, directly coupled to PV generator and battery as depicted in the electric circuit diagram on page 134, neglecting cable losses and blocking diode.

You are now ready to simulate the defined stand-alone system over one year and calculate the two LOLPs with respect to time and energy.

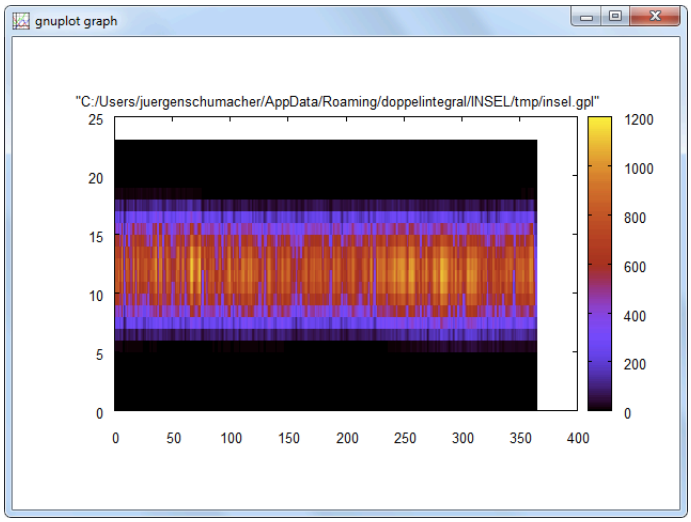
Exercise 7.13 The first step is to provide meteorological data for Jakarta, i. e., global radiation horizontal and ambient temperature. This procedure has been shown in section for the radiation data. What is new is the generation of temperature data.

Solution



The only difficulty might be the question how to access the inputs for the annual mean ambient temperature, annual temperature amplitude etc. These required values are provided by the MTM block. Since they are not default outputs of the MTM entity they must be added manually by modifying the number of block outputs in the MTM block's entity editor.

This example is a good opportunity to use a carpet plot for the representation of the annual time series.



A very smooth annual distribution of the radiation data can be observed – excellent conditions for stand-alone systems with high load coverage.

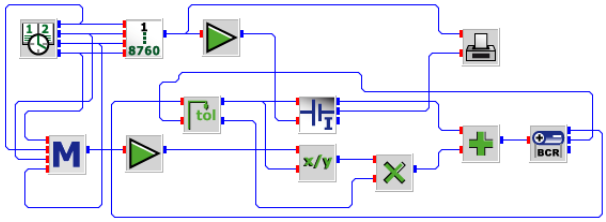
Exercise 7.14 In the second step we try to check, whether our battery definition really fits the load requirements.

Solution We have done most of the modeling work earlier in this module. So, we opened the model with the load profile `prof00.vseit`, saved it as `jakarta2.vseit`, opened the

`bcr2.vseit` model, selected all the relevant objects, copied them to the clipboard, opened the `jakarta2.vseit` file and pasted the battery staff.

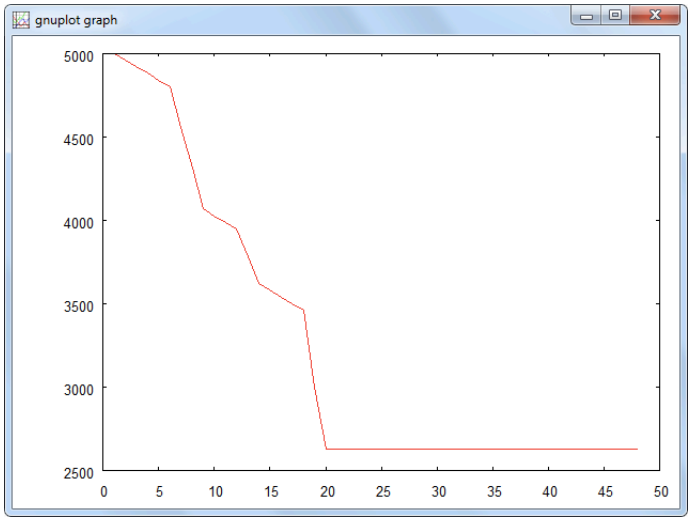
A few further modifications let us test the discharge behavior of the battery under our load profile and leads to this block diagram:

`jakarta2.vseit`



From the earlier discussion of the `bcr02.vseit` model we have learnt that reasonable values for the BCR are 2 V for the switch-off voltage, 2.1 V for the switch-on voltage and 2.4 V for the dump voltage. Since we want to use 120 cells in series and 50 cells in parallel, all voltage values have to be multiplied by 120. But what about the nominal and initial capacity? The parameters are valid for one cell of the lead-acid battery, so they are independent of the size defined through the parameters for series and parallel connections. Hence, we start with a full battery by setting the initial value of the capacity to 100 Ah.

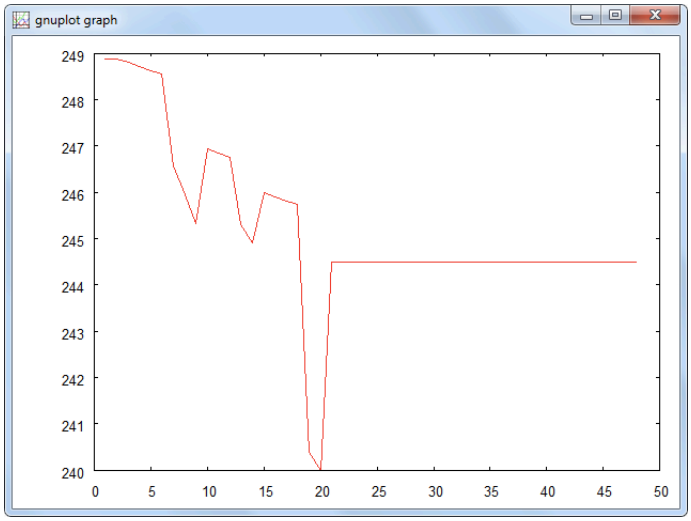
Please recall the sign convention for the load as discussed earlier. When everything is fine, a two day simulation gives this graph:



We see that only 50 per cent of the battery capacity (in total 5000 Ah, since we have

connected 50 cells in parallel) are used before the load is disconnected. Maybe later we can reduce the switch-off voltage a little but for the time being 50 per cent is just right.

Before we look at the charging process let us briefly check to voltage levels.

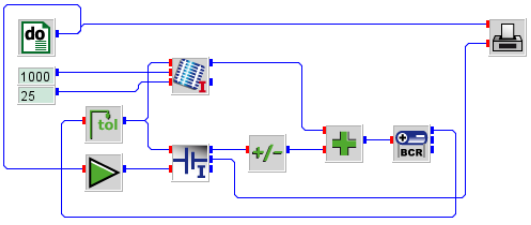


The discharge voltage varies between 250 and 240 volts. Please observe the voltage increase to 245 V after the load is disconnected.

The next question is, how to connect the PV generator? The nominal voltage of one module is 34.4 V. Since we expect the battery to be charged somewhere between 250 V and 288 V a number of $260/34.4 \approx 7.6$, seven or eight modules should be connected in series. Let's try seven first. Since we have planned to use 1000 modules we need $1000/7 \approx 142$ modules in parallel, make it 150.

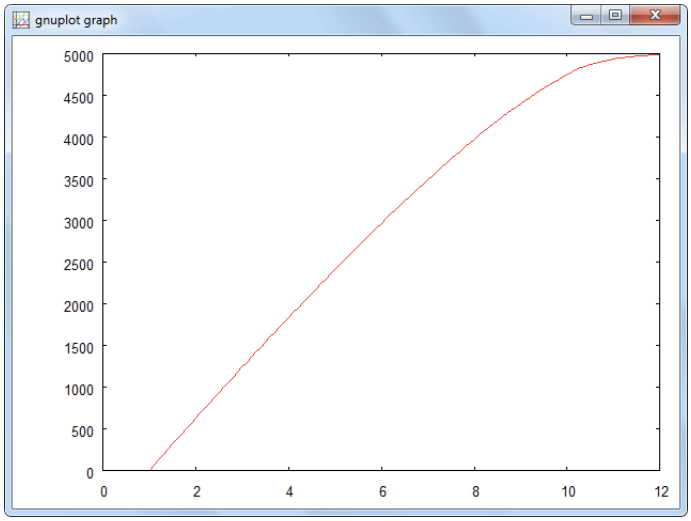
jakarta3.vseit

Let us charge an empty battery under standard test conditions. We did a similar experiment on page 142 already using a NULL block. Now we use a BCR.

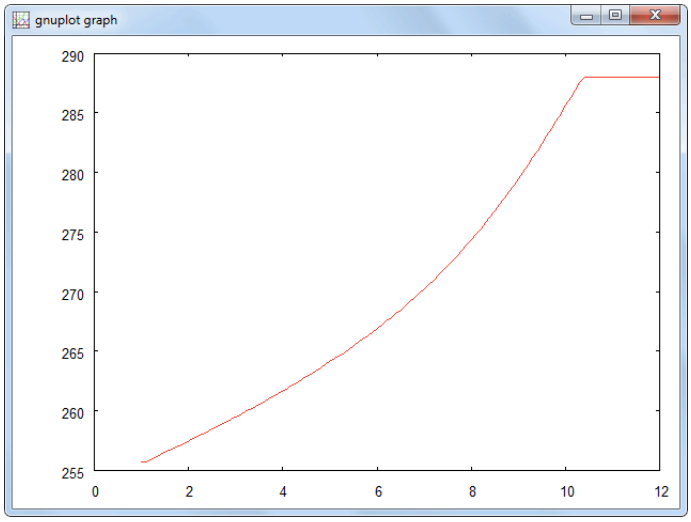


Please observe again the sign convention: The PV generator delivers a positive current and since the battery is charged and the charge current is also positive by definition, the

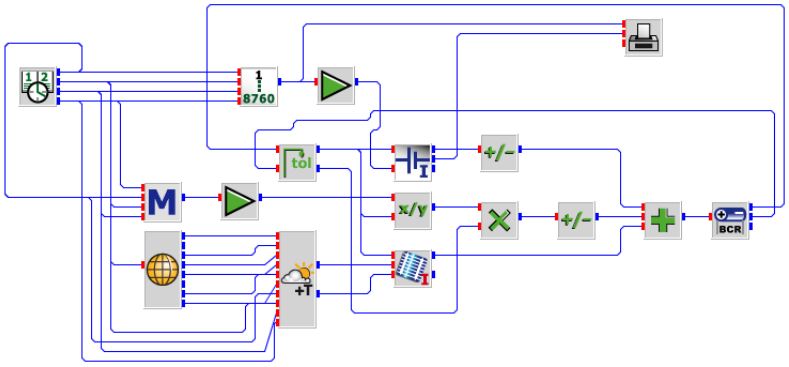
battery current contributes negative to the BCR balance. In other words, the battery is considered as a load.



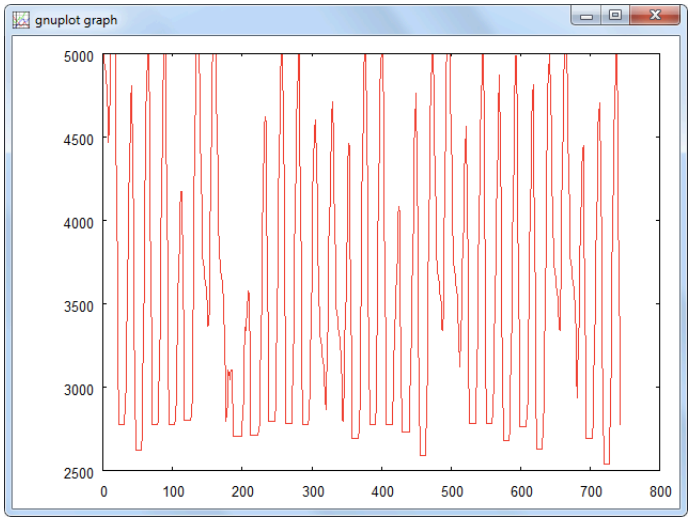
In fact, after 12 hours the battery is completely full. Observing the voltage again, shows that the charge voltage increases from 255 V and is limited by the BCR to a value of 288 V.



`jakarta4.vseit` Now we put everything together and run a one year simulation. It should be no problem to construct the block diagram.

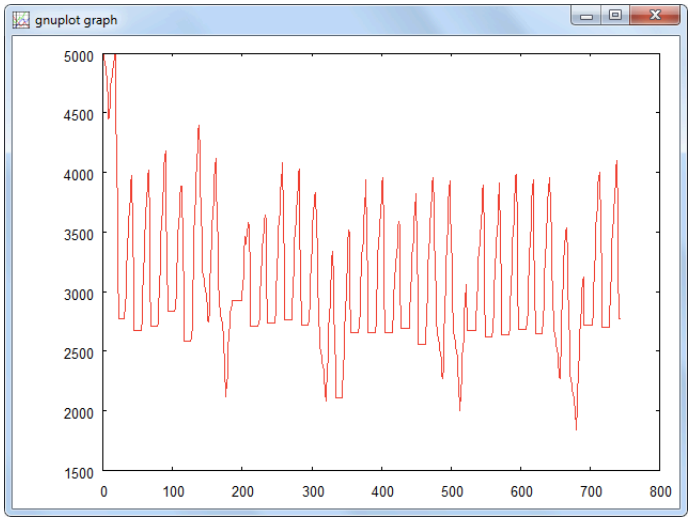


We were careful and started the simulation for January with a fully charged battery and observed the battery capacity.



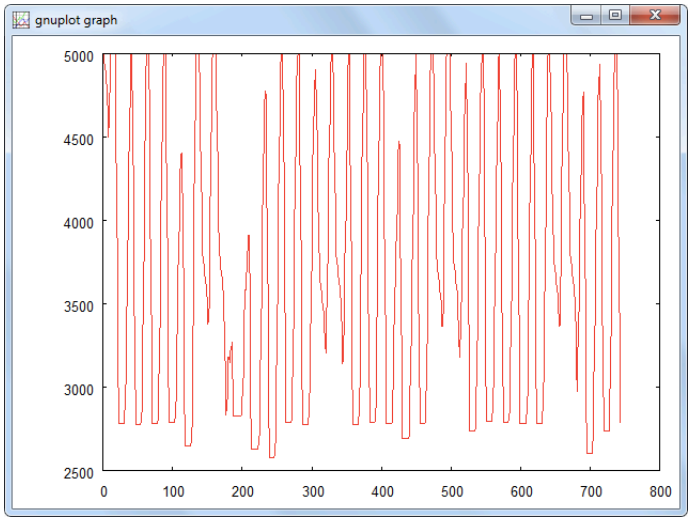
Does your result look like this? Then you are probably happy with the system layout. But you have forgotten – like we did at first – that now under real operating conditions the temperature mode of the PVI block should be set to NOCT mode rather than assuming that the module temperature is given by input number three.

The result for January should be this:

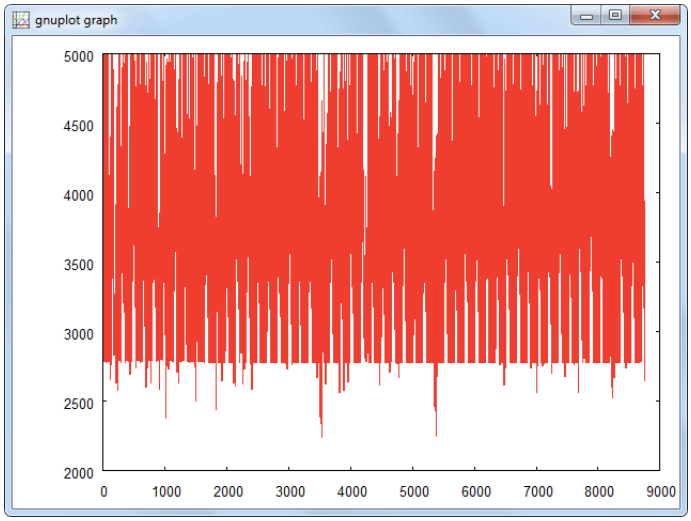


It is amazing how big the impact of the module temperature is. Our generator is not able to fully charge the battery at all.

When you remember how we have dimensioned the PV generator we considered the nominal conditions of the PV modules. In reality the module temperature is usually much higher than 25°C, and therefore, the voltage much lower. Let's try to compensate this effect by enlarging the number of modules in series from seven to eight and plot the graph again.



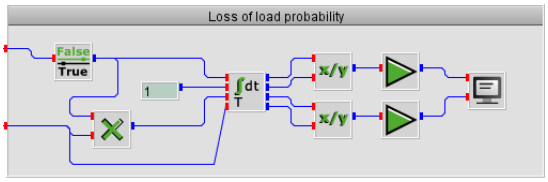
Everything seems fine. Now the same for the whole year.



Nice.

Exercise 7.15 Let's calculate the LOLP. The solution is easy. Try it for yourself first.

Solution This is the macro we used for the calculation of the loss of load probability in terms of time and energy:



Obviously, we should use a CUM block for the different integrations. Cumulating a constant value of one in each time step gives us the total number of time steps, which we divide into the total number of time steps in which the load switch (2nd output of the BCR block) is equal to zero – or its logical inverse is equal to one, calculated by the INV block. A GAIN block with a factor of 100 gives us the LOLP with respect to time in per cent.

The second input of the macro is connected to the total load in kW, multiplied with the number of time steps when the load is off. The SCREEN block outputs

```
LOLP = 33.05 %(t) = 29.81 %(E)
```

if the format parameter is set to

```
''LOLP = '',F5.2,'' %(t) = '',F5.2,'' %(E)''
```

This means that we have a load coverage of approximately 70 %. Now you may want to modify the sizes of the PV generator and/or the battery. For instance, increasing the size of the PV by 20 % (180 modules in parallel instead of 150) gives us

```
LOLP = 31.99 %(t) = 27.91 %(E)
```

which is not much of a gain. Going back to 150 modules in parallel and increasing the battery capacity by 20 % (60 cells in parallel instead of 50) has a much higher impact and we find

```
LOLP =  9.58 %(t) =  9.82 %(E)
```

Please observe that a modification of the sizes has an immediate influence on the system's operating voltage due to the slanted I-V characteristic of the battery. Changing parallel connections is relatively harmless in this respect.

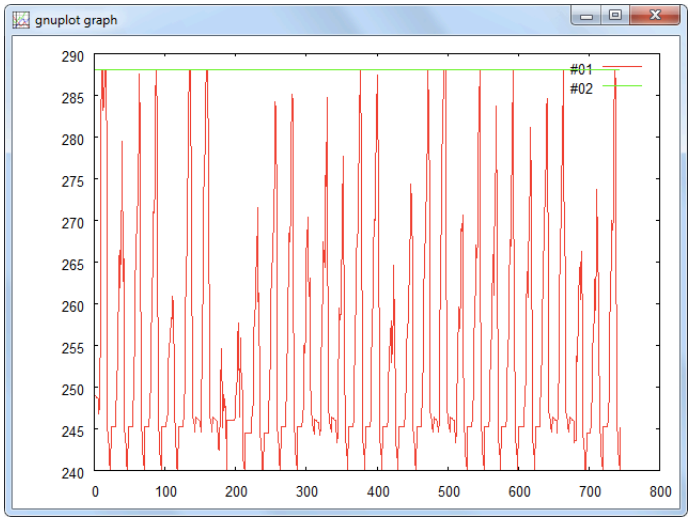
A modification of the number of either PV or battery cells in series implies that the settings of the BCR must be adapted. Otherwise the operation points may no longer be reasonable.

7.6.4 System studies

Now that we have a working simulation model of a stand-alone PV system let us do a few investigations into the system behavior, because it is very easy to access all variables of interest. We try to answer the question, whether the electric performance of the system is reasonable or not.

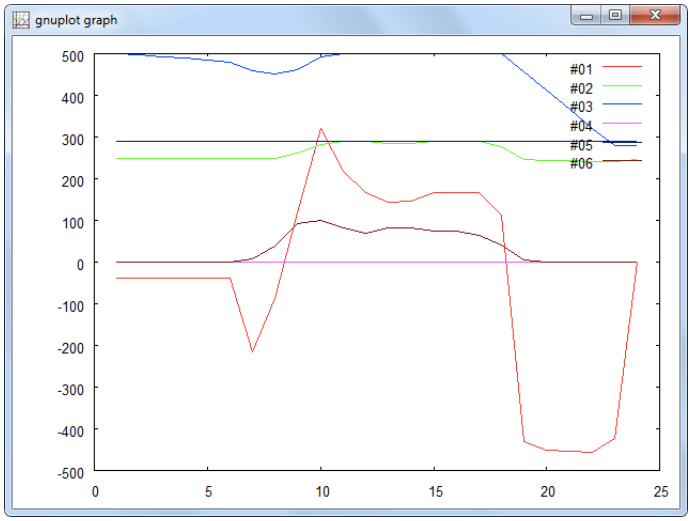
For our analysis we start from the example `jakarta4.vseit` and save is as `detail1.vseit` before we make any modifications. For the system sizes we choose 150 PV modules in parallel, 8 in series, and 50 battery cells in parallel although – from the LOLP point of view one would probably choose something like 60 batteries in parallel since this brings down the LOLP from 30 to 10 per cent.

The first thing to look at is the system operating voltage. Just connect the voltage output of the BCR to the PLOT block and see what happens in January.



We can observe that the battery uses the full voltage range which we have specified in the BCR and that we reach both limits from time to time. Another thing we can see immediately is that when the battery reaches the lower voltage limit of 240 V switching off the load results in an immediate increase of the battery voltage and it remains at open-circuit voltage until the battery is charged again by the PV generator.

Look at the first of January, for instance, and plot battery capacity and voltage, charge/discharge current of the battery and output power of the PV generator before the BCR.

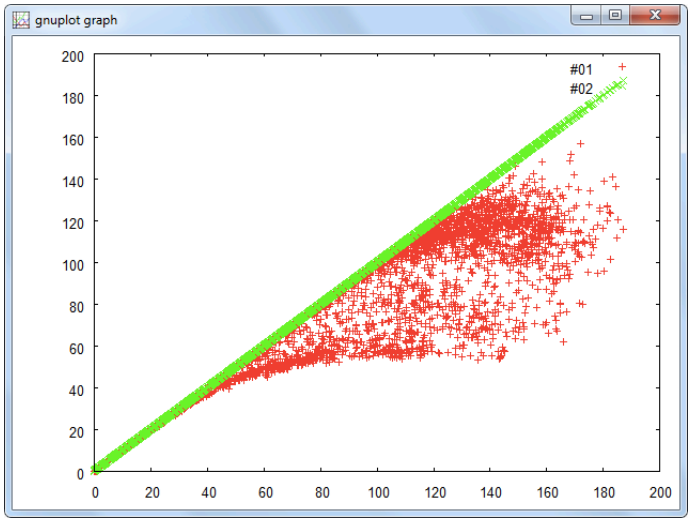


In order to have all curves in one reasonable plot, we have downsized the capacity (blue curve) by a factor 10, plotted the PV power in kW (brown curve). The green curve shows the system voltage, the red curve is the battery charge/discharge current. Please observe, how the BCR limits the charge current even before the battery is fully charged.

Feel free to do more investigations into the model on your own.

Exercise 7.16 Our last example will check how much energy is lost due to the fact that we operate the PV generator coupled directly to the battery rather than in its MPP. Plot the PV power against the MPP power and see how much energy we loose.

Solution



It is assumed that you know by now how to come to this graph. If you have any problem with the block diagram, peep into `detail2.vseit` and check section , page 64.

The figures are 303 MWh compared to 364 MWh, hence we loose approximately 20 per cent. But please notice that not all the MPP power could be used by the system due to the limited battery capacity.

Hint Maybe one hint is useful: The PVI block calculates the PV current only as a function of the battery voltage. It does not care, how much of this PV power can actually be stored in the battery. If you want to evaluate this fraction you must balance the PVI block with the P_{dump} output of the BCR block. We will not go into the details here because this Tutorial (un)fortunately (depending on the point of view) cannot be endless.

We will close the section about stand-alone systems with a first look at how parameter variations can be implemented in INSEL models. In Modules and **??** we will come back to the topic more systematically.

7.6.5 Parameter variations

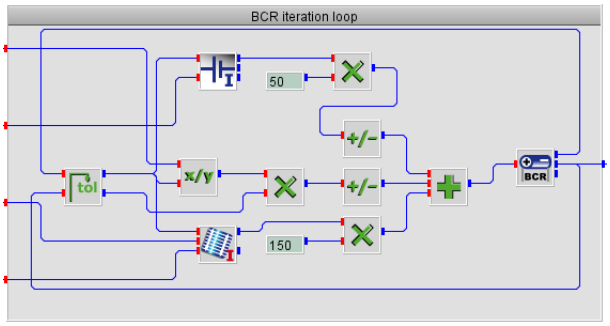
In INSEL we distinguish very scrupulous between block *inputs* and block *parameters*. Inputs are considered to change practically in each time step while block parameters are usually constant during a whole simulation run. Therefore, it is not trivial to modify block parameters during a simulation run. Nevertheless, in some cases tricks and workarounds enable parameter variations.

Looking at the PVI block and its two-diode-model equation shows that very large and very small numbers like Boltzmann constant or electronic charge and comparatively large numbers like temperature in kelvin have to be evaluated in exponential functions. This is numerically very sensitive and one reason, why INSEL evaluates the two-diode-model equation in current densities rather than current.

Single cell simulation

In fact, the PV generator dimensions as defined through the PVI block's parameter settings are internally reduced to a single cell. Then the equations are evaluated and finally the cell voltage is simply multiplied by the total number of cells in series and the cell current by the total number of cells in parallel. This means that there is absolutely no difference in the calculation of a PV generator with 150 modules in parallel and the calculation of a PV generator with only one module in parallel and the current multiplied by 150, for instance. The same argument holds for the number of cells in series of the BTI block.

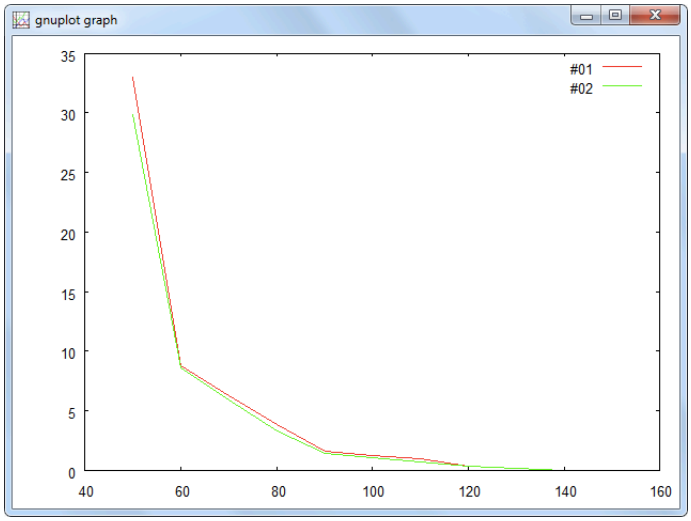
This macro demonstrates the idea:



In consequence, when we want to make a parameter study with variable PV and battery dimensions, we may set the corresponding parameters to one in the PVI and BTI blocks, use two DO blocks for the respective values and multiply their outputs into the currents.

Exercise 7.17 Vary the battery size between approximately one and three days, i. e., in our example between 50 and 150 batteries in parallel. Start from `jakarta5.vseit`, make the modifications and save the model as `jakarta6.vseit`, for example.

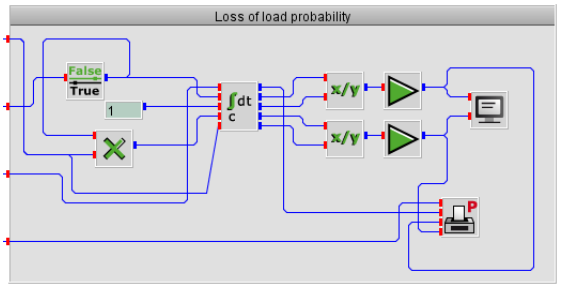
Solution



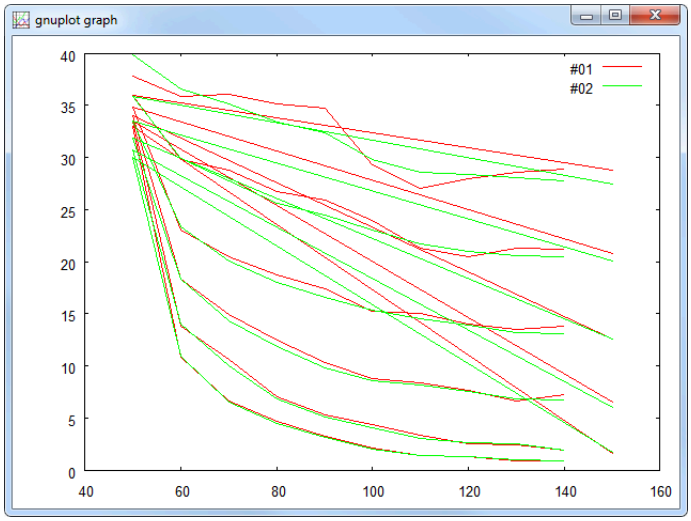
The result looks reasonable. ::

For freaks! What, if we would like to vary the PV size in the same diagram? Only a new DO block is required to vary the number of PV modules in parallel, let's say from 100 to 150 in steps of ten.

Solution We have to modify the loss-of-load-probability macro a little bit.



The output is a bit frustrating, however.



Well, although we used a parametric plot block with the number of PV modules from the DO block as curve parameter this is not really the solution. Can you find out what happened?

We can approach the problem from two sides. The Gnuplot view or the INSEL view. Let's start with the Gnuplot view.

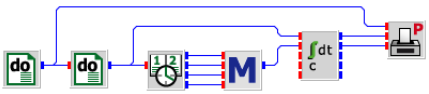
`insel.gpl` As you have learnt earlier, the PLOTP block writes its data by default to `insel.gpl` in the hidden application data directory. These are the first records:

```
0.5000000E+02  0.3785388E+02  0.3982829E+02
0.6000000E+02  0.3584475E+02  0.3658619E+02
0.7000000E+02  0.3603881E+02  0.3515315E+02
0.8000000E+02  0.3517123E+02  0.3341349E+02
0.9000000E+02  0.3469178E+02  0.3236130E+02
0.1000000E+03  0.2929224E+02  0.2980828E+02
0.1100000E+03  0.2708904E+02  0.2853823E+02
0.1200000E+03  0.2800228E+02  0.2841235E+02
0.1300000E+03  0.2860731E+02  0.2802343E+02
0.1400000E+03  0.2885845E+02  0.2775716E+02

0.1500000E+03  0.2875571E+02  0.2746183E+02
0.5000000E+02  0.3592466E+02  0.3582610E+02
0.6000000E+02  0.2985160E+02  0.2989381E+02
```

You must know that Gnuplot plots a new line every time it finds an empty record in the data files. Now it becomes obvious, what the problem is: The blank line comes too early. A down-to-earth approach would be to use a text editor and move all the wrong blank lines one down. You could then use the interactive Gnuplot window, type in `load 'insel.gnu` and get the pretty printed graph.

Model structure

If you wish to understand what happens in INSEL take a look at the principal structure of the block diagram.



Three Timer blocks and one If block control the PLOTP block. Since the PLOTP block is a successor of the CUMC block the PLOTP block is called only when the CUMC block decides to let its successors be executed. Hence, what does the CUMC block see on its condition input? As long as the battery capacity values are equal to 10 the CUMC block continues cumulating its inputs.

The first time the number of battery cells in parallel is not equal to ten but twenty, the CUMC block calculates the outputs and the PLOTP block is called. Please observe again what we have seen in Module already, that the CUMC outputs a value of 10 rather than its input 20 because the CUMC block delays this value (which is very practical).

It should be clear how the calls of the PLOTP block continue for 20, 30 battery cells in parallel and so forth. What if this parameter reaches 150? Again the CUMC block cumulates its inputs. When does it stop?

The answer is, when the condition input changes its value back to 50. How can this happen? Only when the DO block for the battery block is reseted by the DO block for the PV size variation. What is the PV size in this case? 110 modules rather than 100. So? The PLOTP block is called with the condition 110 modules in parallel and starts a new line – too early from a logical point of view – but for human logics only, not the logics of the block diagram.

Is there no way out then? What about delaying the PV size parameter with a DELAY block (initialized by zero – or 100 which leads to a similar result)? Unfortunately, we get the same plot as before. Why? Let's look at the calculation list via the *File* > *Show calculation list* menu.

Not what we wanted

```
Number    Block       Group    Jump
-----------------------------------
    17    CONST         C         1
...
    30    CONST         C         1
    57    DO            T         1
    58    DO            T        47  --!
    56    CLOCK         T        -1    1
    14    HOY           S         1    2
     9    GAIN          S         1    3
...
    15    MTM           S         1   25
    60    GENGT         S         1   26
    35    TOL          -L         1   27
```

```
    52    DIV           S         1   28
...
    13    SUM           S         1   36
    55    BCR           L       -10   37
    65    INV           S         1   38
    45    MUL           S         1   39
    64    CUMC          I       -39   40
    53    DIV           S         1   41
     7    GAIN          S         1   42
    54    DIV           S         1   43
     8    GAIN          S         1   44
    63    SCREEN        S         1   45
    68    PLOTP         S       -45   46
    69    DELAY         D       -48  --!
---------------------------------
```

What do we see? Our DELAY block becomes the last block in the calculation list. But when is it executed? The Jump parameter of the CUMC block points to the CLOCK block. Once the CLOCK block is finished it returns to the DO block number 58 due to its Jump value of −1. The DO block changes the number of batteries from 10 to 20 (at the beginning) and returns control to the CLOCK block. The next time the CUMC block is called it sees a changed battery parameter and therefore its successors are executed – down to the PLOTP block and the PLOTP block returns to the CLOCK block.

When the CLOCK block finishes, it returns to the DO block number 58. When this DO block is finished – i. e., has reached 150 battery cells – it jumps 47 steps forward to the DEALAY block which changes its output in return. The successor of the DELAY block is the DO block 57 which changes the number of PV modules from 100 to 110.

Hence, when the CUMC block is called the next time it sees 50 cells on the condition input. The PLOTP block is called – but with 110 modules. Ergo, wrong. In other words, INSEL has sorted the DELAY block between the outer DO block and the DO block for the variation of the battery size.

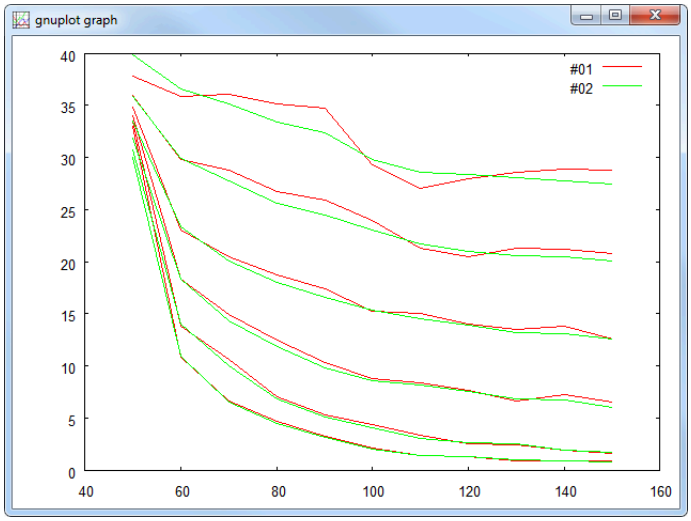
Can we force INSEL to sort it into a different position? Remember, the position of a block in the calculation list depends on the input signals connected to it. In our attempt the only input to the DELAY block came from the outer DO block and therefore the position of the DELAY block in the calculation list is rather logical. If we want to make the DELAY block depend on the inner DO loop, we must add a second kind of dummy input to the DELAY block – the output from the inner DO block, for instance. Please check the modified calculation list.

What we wanted

```
    57    DO            T         1
    58    DO            T        -1
    56    CLOCK         T        46 --!
    14    HOY           S         1   1
     9    GAIN          S         1   2
...
```

```
    15    MTM           S         1  24
    60    GENGT         S         1  25
    35    TOL          -L         1  26
    52    DIV           S         1  27
...
    13    SUM           S         1  35
    55    BCR           L       -10  36
    65    INV           S         1  37
    45    MUL           S         1  38
    64    CUMC          I       -39  39
    53    DIV           S         1  40
     7    GAIN          S         1  41
    54    DIV           S         1  42
     8    GAIN          S         1  43
    63    SCREEN        S         1  44
    68    PLOTP         S       -45  45
    69    DELAY         D       -47 --!
-----------------------------------
```

and the result.



Now we may lean back and be happy that this works. ::

7.7 The hybrid system Energielabor

8 :: Solar heating and cooling

This Module is related to the simulation of solar-driven heating and cooling systems. Starting from the behavior of single components like water- and air-based solar collectors, heat exchangers, storage tanks, sorption wheels, evaporative humidifiers and closed absorption cooling machines, more and more complex systems are developed. These systems reach from simple water-based solar heating systems to complete solar-driven open or closed desiccant cooling systems.

The solar collector data base (static models) includes over 300 market-available water based flat plate and evacuated solar collectors, some market available absorption cooling machines are included in the data base, too.

8.1 Solar collectors

Air collectors

We start with the utilization and behavior analysis of two types of solar air collector models, implemented in the SCAIRC and SCAIRCD blocks. The model of the SCAIRC block is a quite simple static model without consideration of the collector heat capacity and a very fast algorithm. The more complex dynamic model of the SCAIRCD block considers the thermal mass of the solar air collector.

The decision which one of the two models should be used depends on the purpose of the simulation. If only the annual energy production is calculated in hourly time steps, the simple static model is normally sufficient. For detailed analysis of measurement data with sampling times in the range of several minutes or even seconds, for example, the static model won't deliver sufficient results. In this case the utilization of the dynamic model is the right decision.

Exercise 8.1

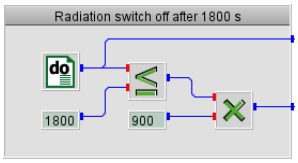
Plot the outlet air temperature of the static and the dynamic solar air collector for a time period of 3600 s with a time step of 5 s for an inlet and ambient temperature of 20 °C, an air flow rate of 0.3 $m^3\,s^{-1}$, 900 $W\,m^{-2}$ solar irradiation on the collector plane, which is switched off after 1800 s and a wind speed close to the collector of 1 $m\,s^{-1}$. The collector parameters are given in Table for both collector types.

Hint

You'll find the solar air collector blocks in the category *Thermal energy* > *Collectors*. Use the DO block for the time period and logical blocks from *Mathematics* > *Logics* to switch off the solar irradiation after 1800 s.

Solution

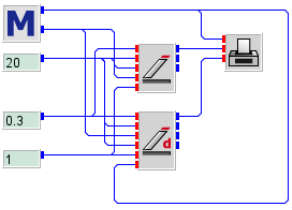
At first, we create a macro for the time and radiation data.



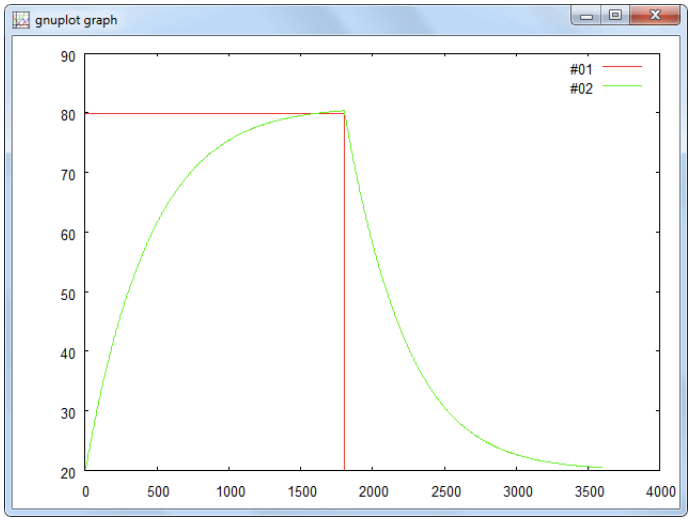
The rest of the model is trivial then.

Parameters	
Numbers of collectors in series	3
Numbers of collectors in series	1
Collector tilt angle / °	34
Collector length / m	12.5
Collector width / m	0.96
Channel height / m	0.095
Channel width / m	0.060
Number of channels	16
Plate thickness / m	0.0014
Optical efficiency ($\tau\alpha$)	0.80
Insulation thickness / m	0.06
Insulation heat conductivity / $W\,m^{-1}K^{-1}$	0.04
Absorber heat conductivity / $W\,m^{-1}K^{-1}$	238
Emissivity glas cover	0.88
Emissivity absorber front	0.16
Emissivity absorber back	0.085
Emissivity channels	0.085
Distance absorber/ front cover	0.025
Thickness back cover material / m	0.002
Density back cover material / $kg\,m^{-3}$	3500
Density absorber material / $kg\,m^{-3}$	2702
Specific heat back cover material / $J\,kg^{-1}\,K^{-1}$	500
Specific heat absorber material / $J\,kg^{-1}\,K^{-1}$	500

Table 8.1: Parameters for the static and dynamic solar air collector simulation.



This is the result:



The x-axis shows the time in s, the y-axis shows the air temperature at the collector outlet in °C. As expected, the curves show that the static model reacts immediately to the change in radiation while the thermal mass of the collector leads to a steady decrease in collector outlet temperature.

Water collectors

For water-based collectors so far only a static model SCETA stimmt nicht mehr! is included in the inselST library, which is used for both flat plate and evacuated collectors. The collector performance and efficiency is described by the four parameters, absorber area, maximum efficiency, linear and quadratic heat loss coefficient. This information is normally given on the data sheet of each market available solar collector.

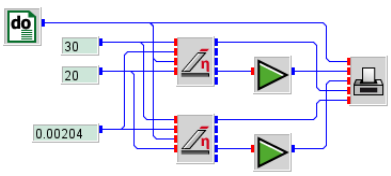
Exercise 8.2 Plot the collector efficiency in per cent and the outlet temperature in °C of a flat plate and an evacuated collector as a function of the solar irradiation on the collector plane, starting from 100 W m^{-2} to 1000 W m^{-2}, with a temperature of 20 °C for both, water inlet and ambient air temperature. The collector parameters are given in Table for both collector types.

Hint

Use the DO block for the variation of the solar irradiation with an increment of 1 W m^{-2}. Use GAIN blocks from the *Mathematics* > *Basics* category to multiply the collector efficiency with 100 to get the value in per cent.

Solution

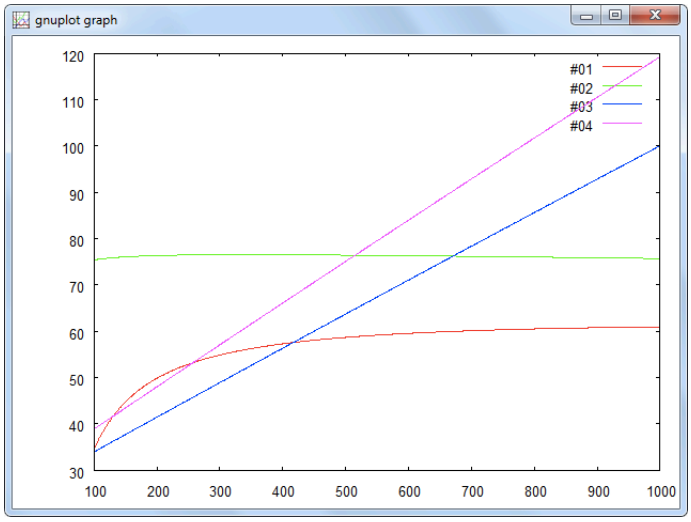
The solution is not difficult.



Parameters	Flat plate	Vacuum tube
Absorber area / m^2	0.9141	0.9141
Maximum efficiency η_0	0.806	0.794
Linear heat loss coefficient /m^2 K W^{-1}	3.666	0.132
Quadratic heat loss coefficient / m^2 K^2 W^{-1}	0.0155	0.010
Sp. heat capacity of collector fluid / J kg^{-1} K^{-1}	3900	3900
Number of collectors in series	1	1
Number of collectors in paralell	1	1
Temperature mode	inlet temp.	inlet temp.

Table 8.2: Collector parameters.

This is the result:



The x-axis shows the solar radiation in the collector plane in W m^{-2}, the y-axis shows the collector efficiency in % and the temperature in °C. Line 1 and 2 refer to the collector efficiency and line 3 and 4 show the outlet temperature of the flat plate and the evacuated collector.

8.2 Storage tanks

Two types of storage tanks are available in the *Thermal energy* > *Tanks* category, a fully mixed (block TANKFM) and a stratified storage tank (block TANKST). Both tank models can be used for heat and cold storage as well.

Full mixed or stratified

The characteristics of the storage tanks with respect to their size and heat losses are

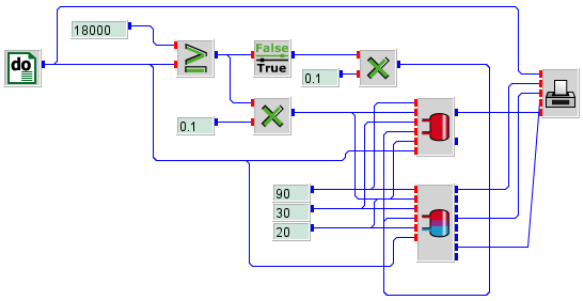
described by six parameters in case of the fully mixed tank. In case of the stratified storage tank two more parameters are necessary to define the effective vertical heat conductivity and the number of nodes N, in which the storage volume shall be divided. The maximum number of nodes is limited to 200. However, in normal cases, depending on the storage size, 5 to 20 nodes are sufficient for nearly all calculations.

The temperature of all N nodes (tank sections) are connected to an output. Output number $N + 1$ is connected to the energy content of the storage tank Q in joule . Since the default number of outputs is set to 5, the number of outputs has to be adapted to the number of nodes increased by one for the energy content output.

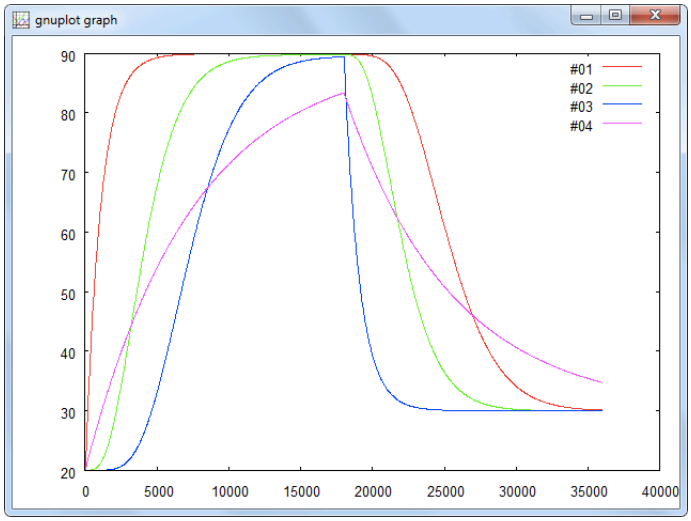
Exercise 8.3 To analyze how the two storage tank types perform, use both blocks with the default parameters and load them by a heat source with a constant temperature of 90 °C and a mass flow rate of 0.1 kg m^{-2} over a time period of 18 000 s. Afterwards, both tanks shall be unloaded with an inlet temperature of 30 °C and a mass flow rate of 0.1 kg m^{-2} for another 18 000 s. Divide the stratified storage tank into 7 layers, plot the output temperature of outputs 1, 4 and 7 and the output temperature of the fully mixed tank in one graph over the whole time period of 36 000 s.

Hint Use a DO block for the time period of 36 000 s with an increment of 20 s.

Solution The solution is straight forward again.



And the graph is:



The x-axis shows the time in s, the y-axis shows the output temperature in °C. Line 1, 2 and 3 refer to the output temperature of the stratified storage tank on node 1, 4 and 7. Line 4 shows the output temperature of the fully-mixed storage tank.

As clearly visible from the graph, the temperature of the stratified storage tank increases and decreases much faster and reaches a much higher maximum temperature as in case of the fully mixed storage tank. The reason for this difference in the loading behavior is obvious.

If the fact is considered, that in the stratified storage tank always the medium with the lowest/hottest temperature is exchanged by the hot/cold medium of the heating source/load, whereas in case of the fully mixed storage tank always the higher/lower temperature of the mixed medium is exchanged. Due to the greater temperature difference between heating source/load and the return temperature from the stratified storage tank more energy is stored/extracted in the same time period as in the fully mixed tank.

8.3 Heat exchangers

All together three different types of heat exchangers are currently available in the the thermal tool box. In the simple heat exchanger models for parallel, counter and cross flow heat exchangers the heat transfer efficiency is simply calculated from the overall heat-transfer coefficient UA in W K^{-1}, the specific heat of the two fluids c_p in J kg^{-1}K^{-1} and their mass flows rates in kg s^{-1}. These heat exchanger models can be used for all kind of fluids like air, water, oil, etc.

The constant efficiency heat exchanger is a very simple model simulating a heat exchanger with constant heat recovery efficiency, which can be defined by the user as

parameter. However, the effect of changing mass flow rates is not considered in this block. Additionally, the nominal electricity can be defined for the calculation of the electricity consumption of rotating heat exchangers. The input of the rotation speed has no influence on the calculated results as long as the rotation speed is grater than zero. This function can be used to consider operation modes without heat exchanging function, e.g. when the heat exchanger is bypassed or in case of a rotation heat exchanger, if there is no rotation.

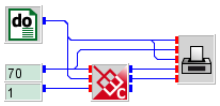
The third available heat exchanger is a real physical model of an air based cross-flow heat exchanger, which considers despite of different mass flow rates of the two air streams also condensation and icing effects within the heat exchanger, including the transfered condensation and latent enthalpy. With its 20 parameters the block can be adapted to the size of each cross-flow heat exchanger, even if heat transfer ribs are included between the heat exchanger plates. However, the construction details of the heat exchanger have to be known.

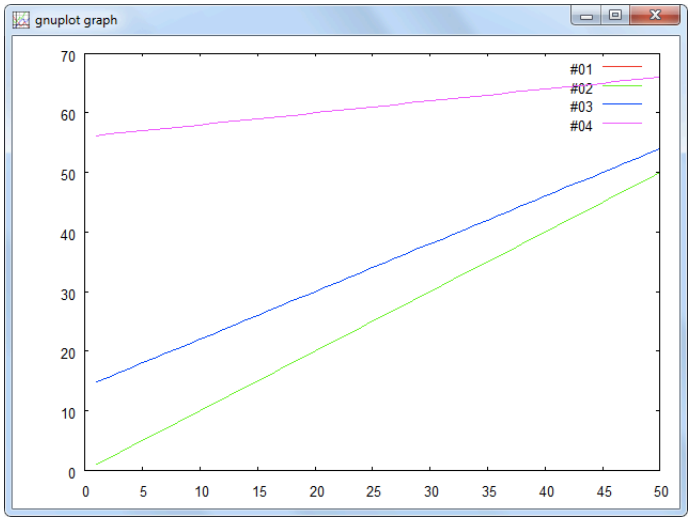
Further heat exchanger models for earth heat exchangers and water sprayed cross flow heat exchangers for evaporative cooling are currently under development.

Exercise 8.4 Use the constant efficiency heat exchanger with a heat recovery efficiency of 0.8, an air inlet temperature on the hot side of 70 °C and an air inlet temperature on the cold side which is increased from 1 °C to 50 °C. Set the rotations speed to a value grater than zero and print the two inlet and the two outlet temperatures.

Hint Use the DO block to increase the temperature of the air at the cold side inlet.

Solution





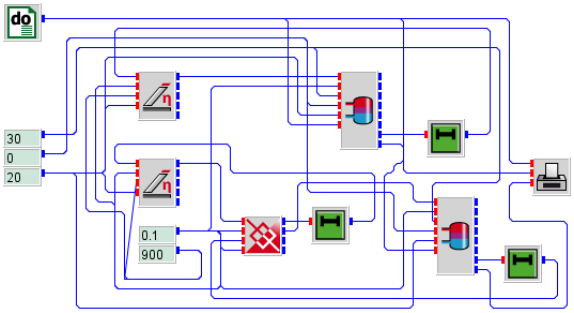
The x-axis shows the calculation steps, the y-axis shows the inlet and output temperature in °C. Line 1 and 2 refer to the input temperatures on the hot and cold side, line 3 and 4 show the temperatures at the heat exchanger outlet.

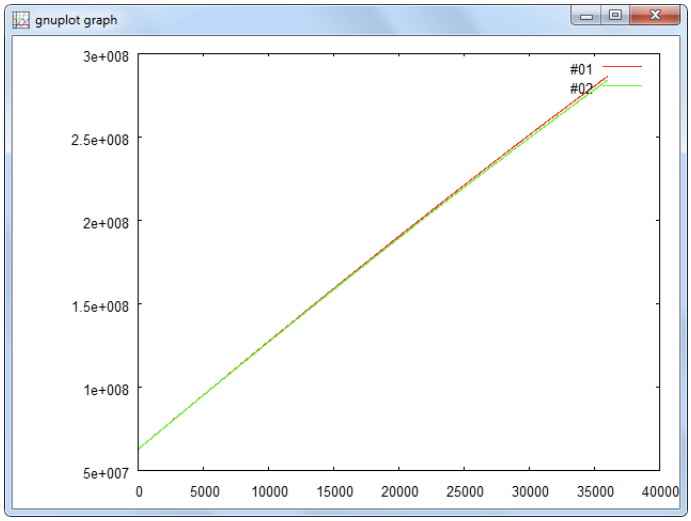
To demonstrate how a heat exchanger block can be integrated in a model for heat transfer between two systems, a complete model of a solar based heating system will be developed in the next exercise, consisting of evacuated solar collectors and a stratified heat storage tank .

Exercise 8.5 Use the evacuated solar collector and the stratified storage tank as described in the collector and storage tank exercise. Analyze two cases, in the first one, connect the evacuated solar collector directly to the storage tank. In the second case, integrate a counter flow heat exchanger from *Thermal* > *Heat exchangers* > *Simple heat exchangers* with the default parameters between the evacuated collector and the stratified storage tank. Print the energy content of the storage tank for both cases in one graph for a time period of 36 000 s with a time step of 20 s, a solar irradiation on the collector plane of 900 W m^{-2} and an ambient temperature of 20 °C. Since the tanks are not unloaded, set the mass flow of the load to zero and the load temperature e.g. to 30 °C.

Hint For this example it is necessary to connect the output temperature of the storage tank and/or the heat exchanger to the input temperature of the collector/heat exchanger. However, at calculation start the output temperatures of the components are not known, therefore DELAY blocks form *Math* > *Loops* have to be integrated between the outputs and the inputs. A DELAY block always outputs the calculated value of the previous time step. Therefore, an initial value has to be defined as parameter, to provide an output value for the first calculation time step. In case of the storage tank and heat exchanger the initial value can be set to the initial temperature of the storage tank of 20 °C.

Solution



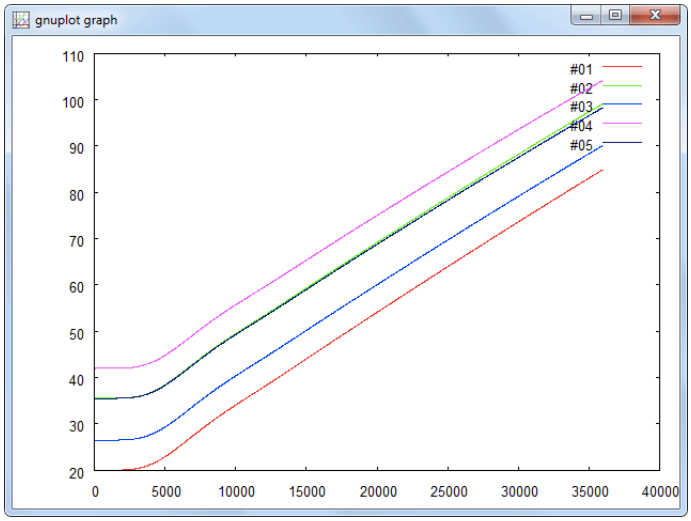


The x-axis shows the time in s, the y-axis shows the energy content of the storage tank in joule. Line 1 refers to the energy content of the storage tank which is directly connected to the collector and line 2 shows the energy content of the storage tank, which is connected to the heat exchanger.

As visible from the graph, there is only a marginal difference between the energy content of the two storage tanks. This means, it doesn't matter if a heat exchanger is included in the system or not, although the heat exchanger has only a heat transfer efficiency of about 70 %.

Exercise 8.6 To analyze this effect, print for the same boundary conditions the input and output temperatures of the two collectors and the input temperature of the storage tank, which is connected to the output of the heat exchanger.

Solution



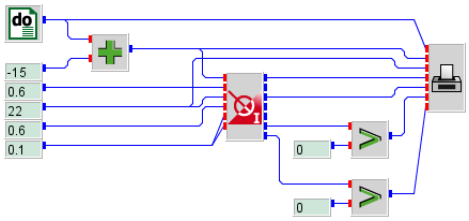
The x-axis shows the time in s and the y-axis shows the temperature in °C. Line 1 and 2 refer to the input and output temperature of the collector without heat exchanger. Line 3 and 4 show the input and output temperature of the collector with heat exchanger and line 5 the input temperature of the storage tank, which is connected to the output of the heat exchanger.

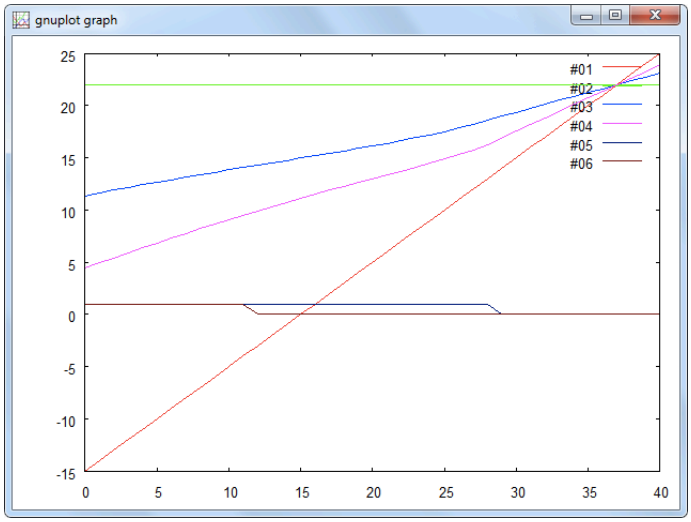
As visible from the graph, the input and output temperatures of the collector with heat exchanger are much higher than the input and output temperatures of the collector, which is directly connected to the storage tank. However, the input temperature of the storage tank, which is connected to the heat exchanger (line 5), is nearly equal to the input temperature of the storage tank, which is directly connected to the collector. This means, that the integration of a heat exchanger just leads to a temperature lift in the collector circuit. This causes some higher collector losses, which are visible in the marginal differences between the storage tank input temperatures and the stored energies.

Exercise 8.7 Use the cross-flow heat exchanger with condensation and icing with the default parameters. Plot the input and output temperatures of the heat exchanger for a cold air inlet temperature, which is increased from $-15\,°C$ to $25\,°C$ with a constant relative humidity of 60 %, a warm air inlet temperature of constant $22\,°C$ with a relative humidity of 60 % and an air volume flow of 0.1 $m^3\,s^{-1}$ on both sides. Print in the same graph indicators showing if condensation and icing occurs within the heat exchanger.

Hint Use the DO block to increase the cold air input temperature, but be aware that negative values are not allowed in this block. To built indicators for condensation and icing use logical blocks from the Math menu, which output a 1 if the condensed water or ice mass is greater than zero and a 0 if no condensation or no icing occurs.

Solution





The x-axis shows the calculation step, the y-axis shows the temperature in °C. Line 1 and 2 show the input temperature of the cold and warm air stream, line 3 and 4 show the output temperatures of the two air streams. Line 5 and 6 are the indicators for condensation and icing. As clearly visible from the graph, the output temperatures do not increase linear with the increase of the cold air inlet temperature, as long as icing and/or condensation occurres.

Exercise 8.8 To analyze how the heat transfer efficiency of the heat exchanger is influenced by condensation and icing effects, calculate the heat transfer efficiency for the warm and the cold air stream using the following equations:

Cold air stream

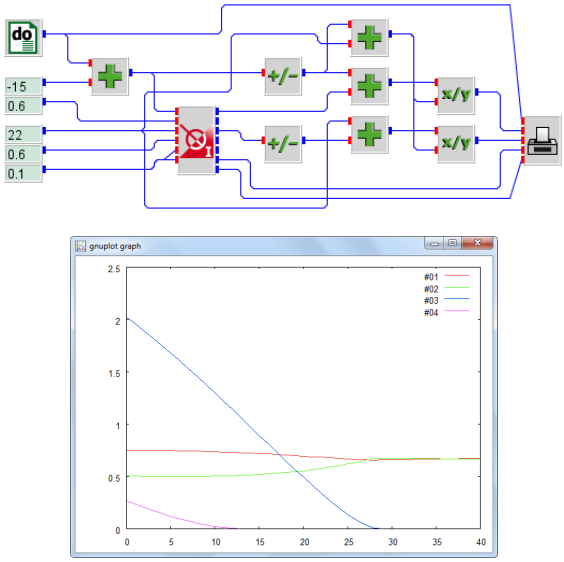
$$\phi = \frac{\mathrm{T}_{\mathrm{cold,out}} - \mathrm{T}_{\mathrm{cold,in}}}{\mathrm{T}_{\mathrm{warm,in}} - \mathrm{T}_{\mathrm{cold,in}}}$$

Warm air stream

$$\phi = \frac{\mathrm{T}_{\mathrm{warm,in}} - \mathrm{T}_{\mathrm{warm,out}}}{\mathrm{T}_{\mathrm{warm,in}} - \mathrm{T}_{\mathrm{cold,in}}}$$

Print the calculated heat exchanger efficiencies together with the condensed water mass and the ice mass in one graph for the same boundary conditions as described above.

Solution



The x-axis shows the calculation step, the y-axis shows the heat exchanger efficiency and the condensed water and ice mass in $\mathrm{kg\,h^{-1}}$. Line 1 and 2 show the calculated heat transfer efficiency calculated for the cold and warm air stream. Line 3 and 4 refer to the condensed water mass and the ice mass.

As visible from the graph, the heat exchanger efficiency of the cold air stream decreases with decreasing condensed water mass from about 70 to 63 % and remains constant for conditions without condensation. The increase in the heat exchanger efficiency with increasing condensed water mass results from the condensing and latent enthalpy, which is set free during the condensing and icing process. However, the heat exchanger efficiency calculated form the warm air stream increases with decreasing condensed water mass from about 48 to 63 %. This antithetic behavior results from the fact, that a part of the condensing enthalpy is also transferred to the warm air stream which leads to higher outlet temperatures and therefore to a lower heat exchanger efficiency.

... to be continued

9 :: INSEL GUI's with VSEit

Sorry, this Module is not yet available.

10 :: INSEL in MATLAB

The INSEL Renewable Energy blockset for Simulink, including the S-function `SinselBlock`, has been removed from INSEL, since it had not been used for years. INSEL models can still be run from MATLAB through the `insel` executable, as described at the end of this module.

10.1 MATLAB

MATLAB is a high-level computer language, developed by the company The Mathworks, Inc. during the 1980's. The acronym MATLAB stands for matrix laboratory. With its interactive environment the software can be used for algorithm development, data visualisation, data analysis, and for numeric computation. It contains mathematical, statistical, and engineering functions, like

- Matrix manipulation and linear algebra
- Polynomials and interpolation
- Fourier analysis and filtering
- Data analysis and statistics
- Optimization and numerical integration
- Ordinary differential equations (ODEs)
- Partial differential equations (PDEs)
- Sparse matrix operations

Additional toolboxes provide specialized mathematical computing functions for areas including signal processing, optimization, statistics, symbolic math, partial differential equation solving, and curve fitting.

When the program is started the MATLAB default desktop opens, as shown in Figure .

Below a standard menu and toolbar four windows are shown:

- The Current Folder window at the left side displays the content of the assigned current folder. The current folder can be changed anytime from the pull-down menu in the toolbar.
- The window displayed in the center is the Command Window. It shows a prompt >> where MATLAB commands and functions can be executed interactively.
- The Workspace and Command History windows are displayed at the right side of the MATLAB desktop.

Command window

There are several ways how to communicate with MATLAB. The most direct access is to

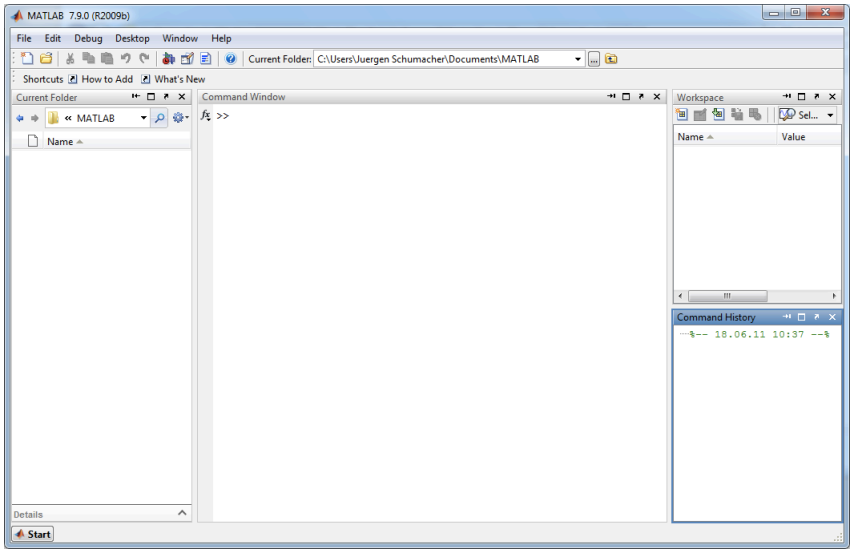


Figure 10.1: Default MATLAB desktop layout.

type in commands at the command promt. For example, typing `magic(4)` makes MATLAB answer with

```
>> magic(4)
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

If we wish to know how `magic` works, we can ask MATLAB for help:

```
>> help magic
 MAGIC  Magic square.
    MAGIC(N) is an N-by-N matrix constructed from the integers
    1 through N^2 with equal row, column, and diagonal sums.
    Produces valid magic squares for all N > 0 except N = 2.

    Reference page in Help browser
       doc magic
```

The link `doc magic` leads directly to the online help browser for more information.

We could ask MATLAB to calculate the sums of the four colums:

```
>> sum(magic(4))
ans =
    34    34    34    34
```

M-files

Beside the interactive computational environment MATLAB provides a powerful programming language. Files that contain code in the MATLAB language are called M-files. Two kinds of M-files can be written:

- Scripts operate on data in the workspace. They do not accept input arguments, nor do they return output arguments.
- Functions do not access data in the workspace but internal variables. Data exchange with the workspace is possible through input arguments and through return output arguments.

hello.m

A simple script which displays the hello-world string has only one line of code:

```
'Hello World!'
```

When the script is saved in a file named `hello.m` to the current folder, it can be executed by just typing the name of the script at the command prompt:

```
>> hello
ans =
Hello World!
```

sum12N.m

A simple function which sums up integers from one to a variable n and returns the result in a variable named y is

```
function y = sum1toN(n)
y = sum(1:n)
```

The result is

```
>> sum1toN(10);
y =
    55
```

It is possible to write functions for MATLAB in C or Fortran, too. The minimal C and Fortran prototypes are

helloC.c

```
#include "mex.h"
void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[]) {
   mexPrintf("Hello C!\n");
}
```

and

helloF.f

```
      INCLUDE "FINTRF.H"
      SUBROUTINE MEXFUNCTION(NLHS,PLHS,NRHS,PRHS)
      IMPLICIT   NONE
      INTEGER    NLHS,NRHS
      MWPOINTER  PLHS(*),PRHS(*)
      MEXPRINTF('Hello Fortran!')
```

```
RETURN
END
```

10.2 Running INSEL models from MATLAB

The MATLAB function `system` runs the `insel` executable like any other shell command and returns its exit status and its standard output:

```
>> [status, output] = system('insel hello.insel');
```

The text in `output` can then be parsed, e. g., with `str2num` for numerical output of SCREEN blocks. See Chapter for the syntax of `.insel` files.

PART III :: Advanced concepts

11 :: INSEL without GUI

11.1 Running `.insel` files

We hope, that it has been wonderful to see how the completely graphical approach to programming with graphical programming elements like INSEL blocks and their interconnections works. When you "look behind the stages" it doesn't require much information to interpret a user-written block diagram simulation application. By generating a graphical block diagram, let us ask and answer the question, what actually happens and which kind of information is provided to the simulation environment.

In block diagram based simulation environments like INSEL the information consists basically only of two information types – the block diagram structure and the used parameters.

As an alternative to graphical programming in VSEit, you can also write INSEL models in a text editor. In order to write an INSEL simulation program in its ASCII representation you must use a text editor like Windows Notepad or Kedit, for example. You then need to know the syntax of the INSEL programming language, which is very simple and consists of only few keywords.

Hello, world!

In a book of Kerninghan and Ritchie on programming in C the now famous Hello, world example was given, a C program which displays the string "Hello, world!" on the computer's display. If you want to solve this task with INSEL, you need a block which is able to display information on the screen. One of these blocks is the SCREEN block. It has an optional parameter for the format of the displayed information.

Please notice that two different types of information have to be provided for the SCREEN block and – more general – each INSEL application:

Structure and parameters

First, the model structure fixes which blocks are used in a certain application and how they are interconnected. Second, the model parameters fix what the current values of the block parameters are.

`hello.insel`

In this example, model structure and parameters are extremely simple, because all we need is one single block. These two statements do the job:

```
s 1 screen
p 1 '(''Hello, world!'')'
```

S or s statement

The first record starts with an `s` which is an INSEL keyword (short for structure). It is followed by an arbitrary block number (which has to be unique for every block that is used in a given INSEL program) and the block's name, SCREEN in this case. Please observe that the entries are separated by a delimiter, one blank (space character) in this case. Usually, a list of block inputs follows after the block name, but in this example no inputs need to be connected.

P or p statement

The second record provides the necessary parameter information starting with the

keyword p (short for parameter). In order to enable INSEL to uniquely identify the given values with a certain block, the above mentioned user-defined block number follows the p-keyword.

The parameter list comes next, in this case the `'(''Hello, world!'')'` string. Because the format parameter is a string, it has to be embedded in quotes. Concerning the string value pay attention to use two single quotes `''` and not one combined ". The parentheses in the string follow the Fortran format conventions.

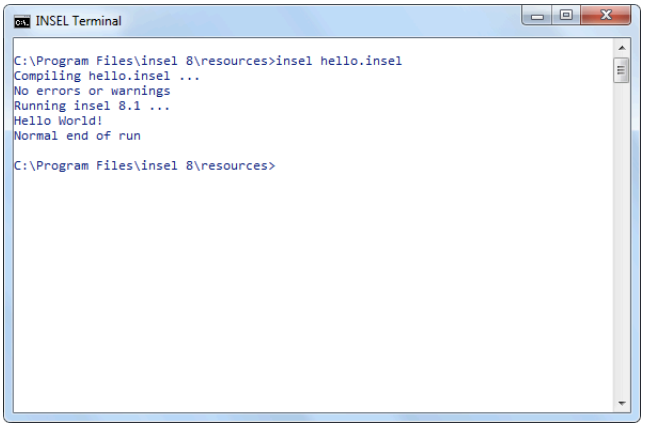
Now you are ready to save the information under a file name like `hello.insel`, for example. It is necessary to use the `.insel` extension for INSEL source code files. The next step is to tell INSEL that you like to execute the `hello.insel` application.

Execute `hello.insel`

There are two options how the model can be executed: either from the VSEit interface via File > Open .insel File... and the Run button, or from a DOS box via `insel hello.insel`. The second option requires that `insel.exe` is in the current `%PATH%` and that `hello.insel` is available in the current directory.

Exercise 11.1 Please open an INSEL Terminal from the tool bar and try it.

Solution



Photovoltaics

As a second more applied example with inputs and outputs we now write a `.insel` model which calculates the power output of a photovoltaic module as a function of the voltage. We start with the timer block DO, which outputs the voltage from 0 to 40 V in steps of 0.01 V to the PVI block.

```
s 10 do
p 10  0       % Initial value
     40       % Final value
      0.01    % Increment
```

Comments

Please observe that comments can be used in `.insel` files: everything starting with a % symbol to the end of the record is gobbled by the INSEL compiler.

The PVI block uses the first output from the DO block as an input, i. e., the output from block number 10. This first output is written as `10.1`, the second output would be `10.2` etc. The PVI block also needs the irradiance and module temperature as inputs. To keep the model simple, irradiance and temperature are set constant. Hence, we define two different constant blocks.

```
s 11 const
p 11 1000.0   % Irradiance in W/m2
s 12 const
p 12   25.0   % Module temperature in degrees celsius

s 20 pvi 10.1 11.1 12.1
```

The next block is the multiplication block MUL, where the voltage (output from the DO block number 10) and the current (output from the PVI block number 20) are multiplied.

```
s 30 mul 10.1 20.1
```

Finally, the result of the MUL block – the DC power of the PV module – is plotted against the voltage.

```
s 40 plot 10.1 30.1
```

What is still missing, are the parameters for the PVI block. INSEL provides a data base for about 20 000 modules of the PV module list of the California Energy Commission (CEC), with the model parameters fitted by NREL's System Advisor Model (SAM). It is saved in the binary file `pv_modules.fb.bin` in the `data/pv` directory, so the easiest way to find a module is the PV browser of VSEit (see module 7). Each module is identified by its manufacturer and model name, e. g., `SunPower SPR-X21-350-BLK`.

Module ID

If only the first parameter of the PVI block (the mode) is given, all other parameters are read from the data base, for the module ID given as string parameter:

```
s 20 pvi 10.1 11.1 12.1
p 20 0        % Mode
     'SunPower SPR-X21-350-BLK'
```

The modules of earlier INSEL versions are still in the data base, with IDs like `pv005405`, so that existing models keep working.

`.bp files`

Parameters of modules which are not in the data base, e. g., determined with the PVDET1 block from the data sheet, are saved in files with the extension `.bp`, which is short for block parameters. This is a `.bp` file for the Advent 210 module of Advent Solar, Inc.:

`pv005405.bp`

```
% File name      pv005405.bp
% Module         Advent 210
% Manufacturer   Advent Solar, Inc.
% Cell type      poly
```

```
              % Mode must be set externally
 60           % Number of cells in series N_s per module
  1           % Number of cells in parallel N_p per module
  1           % Number of modules in series M_s
  1           % Number of modules in parallel M_p
  0.0275      % Cell area A_c (m^2)
  1.663       % Module area A_m (m^2)

              % Electrical parameters
  1.12        % Band gap (eV)
  0.2542      % Short-circuit current parameter C_0 (V^-1)
  0.153E-03   % Isc temperature coefficient  C_1 (V^-1 K-^1)
  0.169663E+05 % Shockley saturation parameter C_01 (A m-2 K^-3)
  0           % Recombination saturation parameter C_02 (A m-2 K^-5/2)
  0.00012389  % Series resistance r_s (Ohm m^2)
  0.03129369  % Parallel resistance r_p (Ohm m^2)
  1.0165366   % Shockley diode ideality factor alpha
  2           % Recombination diode quality beta
  0           % Bishop parameter-1
  0           % Bishop parameter-2
  0           % Bishop parameter-3
  3.0         % Module tolerance plus
 -3.0         % Module tolerance minus

              % Thermal parameters
  1.680       % Characteristic module length l_m (m)
 22.700       % Module mass m_m (kg)
  0.70        % Default absorption coefficient a
  0.85        % Default emission factor epsilon
900.0         % Default specific heat of a module C_mod (J kg^-1 K^-1)
 47.0         % Nominal operating cell temperature NOCT (degrees C)
 25           % Intial value of cell temperature (degrees C)

              % Numerical parameters (optional)
  1E-5        % Error tolerance of voltage of single cell (V)
100           % Maximal number of iterations to solve I/V-equation
```

When you look at the file and into the documentation of the PVI block, you see that the temperature mode is not part of the .bp file but must be set as an extra parameter.

I or i statement

Rather than copying the whole file into your .insel file the include statement can be used. Its syntax is simply

```
i 'file name'
```

When the INSEL compiler finds this statement in a .insel file it replaces the statement with a verbatim copy of the file content. A .bp file is included in the parameters of the PVI block like this:

```
s 20 pvi 10.1 11.1 12.1
p 20 0       % Mode
     i 'pv005405.bp'
```

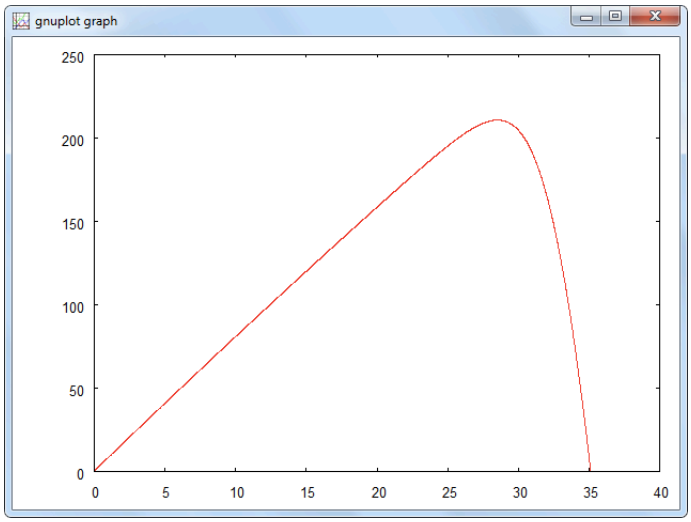
In conclusion, the complete program for calculating the DC power of a PV module of the data base as a function of voltage looks like this:

```
s 10 do
p 10  0       % Initial value
     70       % Final value
      0.01    % Increment
s 11 const
p 11 1000.0   % Irradiance in W/m2
s 12 const
p 12 25.0     % Module temperature in degree celsius

s 20 pvi 10.1 11.1 12.1
p 20 0        % Mode
     'SunPower SPR-X21-350-BLK'
s 30 mul 10 20.1

s 40 plot 10 30
```

Please observe the syntax used by the PLOT block: when no output number is specified this defaults to output number one.

Exercise 11.2 Save the file under any name, for example `pv.insel`, and run it.

Solution



Arbitrary order of statements

One special feature of graphical programming languages like INSEL is that the order of statements in the source code is completely free. We could shuffle the model into any arbitrary order, like

```
s 40 plot 10 30
p 10  0       % Initial value
     70       % Final value
```

```
      0.01    % Increment
s 11 const
s 12 const

s 20 pvi 10.1 11.1 12.1
p 20 0        % Mode
     'SunPower SPR-X21-350-BLK'
s 30 mul 10 20.1

s 10 do
p 11 1000.0   % Irradiance in W/m2
p 12 25.0     % Module temperature in degree celsius
```

In a conventional programming language it would be impossible to use variables like the PLOT block's inputs from blocks 10 and 30 before the values are defined. This makes it possible – and you probably used this feature without notice – to construct the VSEit applications in any order, delete VSEit objects, add new ones etc. By the way, the VSEit objects appear in the `.vseit` file in the order in which you placed them on the screen.

C or c statement

One last statement completes the set of only four statements in total – INSEL is perhaps the simplest programming language in the world with only four statements (the earlier versions had even only two: `s` and `p`). The `c` statement can be used to define constants by name and value. The syntax is

```
c name value
```

The variable `name` (no enclosing quotes) defines the name of the constant, `value` specifies its value, which can be either a valid numerical or string parameter with the usual INSEL conventions (i. e., strings enclosed by quotes, numerical values not enclosed by quotes). Variable names can be constructed from the characters `[A-Z][a-z][0-9]` but have to start with an alphabetic character.

In addition, the special character # is allowed in variable names. Its use should however be restricted to developers of "`.include/.insel`" applications. What is this?

11.2 .include/.insel applications

Program development (not only) in the field of renewable energy simulation can be classified into two different aspects: (i) the calculation model formulation and (ii) program parts which provide convenient user interfaces. In many cases both program parts are combined into one software project.

The `c`- and `i`-statements enable a concept which we call "`.include/.insel`" applications. With this method INSEL provides a programming environment for the experienced INSEL user and C/C++ programmer (or any other high-end programming language software developer), where both calculation kernel and user interface can be written completely independent. The results are applications which look like common

Windows applications, but which give the experienced user of such a program access to the modeling level, without having to recompile the user interface code.

To understand this in more detail let us use the previous example, in which the DC power of a PV module was calculated. In this example we had defined two constants for the irradiance and module temperature, blocks 11 and 20, respectively. The values 1000 W/m^2 and 25 °C were inspired by the standard test conditions for PV modules. Additional parameters were the voltage range and increment and the ID of a specific PV module.

PV module flasher

We have modified the example so that it can be used as a "laboratory flasher" which can be used with a convenient user interface for any real (simulated) PV module.

`flasher.include`

The model is split up into two files. The first one contains only c-statements:

```
% Include file for the definition of the free parameters
c #PVModule        'SunPower SPR-X21-350-BLK'
c #InitialValue     0
c #FinalValue      70
c #Increment        0.01
```

`flasher.insel`

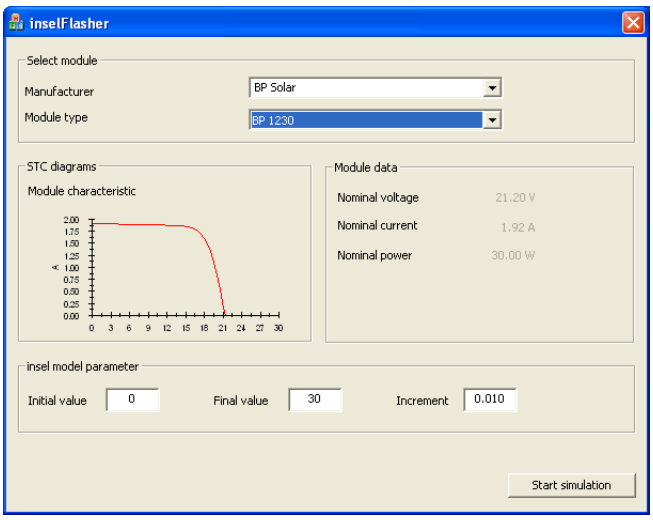
The second file contains the model which makes use of the variables defined in the include file.

```
% INSEL file to plot the STC I-V curve
i 'flasher.include'
s 10 do
p 10  #InitialValue
      #FinalValue
      #Increment
s 11 const
p 11 1000.0   % Irradiance in W/m2
s 12 const
p 12   25.0   % Module temperature in degree celsius

s 20 pvi 10.1 11.1 12.1
p 20 0        % Mode
     #PVModule

s 40 plot 10 20
```

In order to adapt the model to any given PV module – or in other words, to flash a certain module in the laboratory and create an I-V curve protocol – only the values in the include file must be changed. There is no need to touch the `.insel` file.

In consequence this means that the user interface needs to manipulate the `.include` file only. Once a user of such an interface has entered the parameters the interface tool can execute the complete model by calling the inselEngine. This is what was meant when we said that interface and calculation model are completely disjoint.

The details for programming of C++ interfaces is beyond the scope of this Tutorial. This screenshot shows a possible implementaion of the flasher example.



11.3 Optimization with GenOpt

11.4 Direct calls of INSEL blocks

For most applications outside the GUI, INSEL blocks are best called through the `insel` executable, as described at the beginning of this module, or through the Python wrapper:

```
import insel
insel.block('sum', 2, 3)                          # 5.0
insel.block('gain', 2, 5, 7, parameters=[3], outputs=3)
insel.template('a_times_b', a=7, b=3)             # 21.0
insel.run('/path/to/model.vseit')
```

The wrapper writes a small model, runs it with the `insel` executable and returns the outputs as Python numbers.

Block interface

It is still possible to call a block function directly from a C/C++ or Fortran program, independently of the inselEngine. All blocks have the same interface

```
SUBROUTINE name(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
```

in Fortran, or

```
#include "insel_block.h"
extern "C" void name(int *ctrl, double *ins, double *outs, int *ints,
   double *reals, double *params, char strings[][1024]);
```

in C/C++. All reals are double precision, each string has 1024 characters. The named indices of `CTRL` (`RETCODE`, `CALLMODE`, `OPMODE`, `N_INS`, ...) are defined in `insel_block.h` and `insel_block.inc`, which are installed in the `include` directory of INSEL; C indices are the Fortran indices minus one.

Calling sequence

The calling program takes over the tasks of the inselEngine:

- Load the block library (`libInselFB.so`, for instance) and get the block function, e. g., with `dlopen` and `dlsym`. The function `insel_library()` exported by each library returns the API version and the table of its block functions. The blocks call `insel_identify` and `msg` from the INSEL tools library, so it must be loadable, too.
- Call the block with `CTRL(CALLMODE) = IDENTIFY` (−1). The block writes its blank-separated block names to `STRINGS(1)` and the sizes (`MIN_INS`, `MAX_INS`, `N_OUTS`, `N_INTS`, `N_REALS`, `MIN_PARAMS`, ..., `GROUP`) to `CTRL`, see `ID_MIN_INS` etc. in `insel_block.h`. The calling program allocates the arrays accordingly.
- Set `CTRL(OPMODE)` to the 1-based index of the block name if the function contains more than one block, set `CTRL(N_INS)` to the number of connected inputs, and fill `PARAMS` and `STRINGS`.
- Call the block once with `CONSTRUCT` (1), as often as needed with `STANDARD` (0), and finally with `DESTRUCT` (2). After each call, a non-zero `CTRL(RETCODE)` indicates a warning or an error.

The block state lives only in INTS and REALS, hence one set of arrays per block instance allows for any number of instances.

12 :: Programming INSEL blocks

The Module “Programming INSEL blocks” of the INSEL 7 Tutorial started with the statement “One thing is for sure: this is a heavy Module. What we are trying to show here is how INSEL can be tailored to your particular needs on a source code level, i. e., this Module demonstrates how you can interfere with INSEL with your own code. Not many programs allow this at all. INSEL does.” Well, INSEL 8 still does, but doing so in INSEL 8 is now “easy as pie.”

Of course, it is still necessary to have basic skills in a programming language like Fortran or C, for example – therefore, the Module offers a Fortran crash course for the novice-programmer student. And, with roundabout one hundred pages, this is the by far longest and toughest Module of this Tutorial and you will need a lot of patience to work through it – patience with yourself, patience with the software concepts, and patience with the here-and-there nerving author.

However, the rest is mainly taken over by wizards, compiler tools, LaTeX documentation routines, and so forth. But, see for yourself, and be prepared...

Installation requirements

User blocks are compiled with the GNU compilers into a library which INSEL loads at runtime. In order to use the programming support in INSEL 8 the following tools must be installed and found via the `PATH` variable:

- The GNU C++ compiler (g++) and the GNU Fortran compiler (gfortran). Under Windows they are available, for example, with MinGW-w64. Under macOS, INSEL uses clang++ for C++ and the gfortran of the Homebrew package gcc.
- GNU make (under Windows `make` or `mingw32-make`)
- A PDF LaTeX compiler (`pdflatex`), only needed for the User Block Reference Manual
- Optionally, you may wish to use the integrated development environment Eclipse IDE. A detailed description of how-to-install-and-use Eclipse can be found in Module of this Tutorial.

Before you start to work with the tools it is recommended to check that the following commands are available in a terminal (a command prompt under Windows):

- `gfortran --version`
- `g++ --version`
- `make --version`

Fortran, C/C++, etc.

A few words to programming languages and recommendations. Most of the blocks which build the calculation kernel of INSEL are written in Fortran 77, while the inselEngine is written in C++ and based on the compiler-compiler tools Flex and Bison.

The graphical user interface is written in Java. User blocks can be written in Fortran 77, Fortran 90, or C++; the Block Wizard generates C++ by default.

Nevertheless, we recommend, that you write your blocks in Fortran 77.

Why Fortran?

Because the good-old-fashioned Fortran dinosaur is still a very powerful and easy-to-learn programming language for numerical calculations in our view. Large libraries written in Fortran exist – one of the most important and well known is the IMSL library, a collection of Fortran subroutines and functions useful in research and mathematical analysis. With a little experience in computing you can understand and apply this language in a single day if you concentrate on the essentials.

If you ask, "Why Fortran 77 – I have heard that there is a new Standard called Fortran 95, sounds like there was some progress in language development?" our answer is:

"Keep it simple. For programming of numerics very few statements are required. The rest makes Fortran more and more look like C++. If you need a powerful language like C++ for Windows interface programming, for example, then learn C++ and not Fortran."

If you do not yet know how to write Fortran programs here comes a quick introduction. When you are already familiar with Fortran programming or intend to use INSEL with a different language like C or C++ anyway, you can directly proceed to the section "Programming INSEL blocks (cont.)" on page 235.

12.1 A Fortran crash course

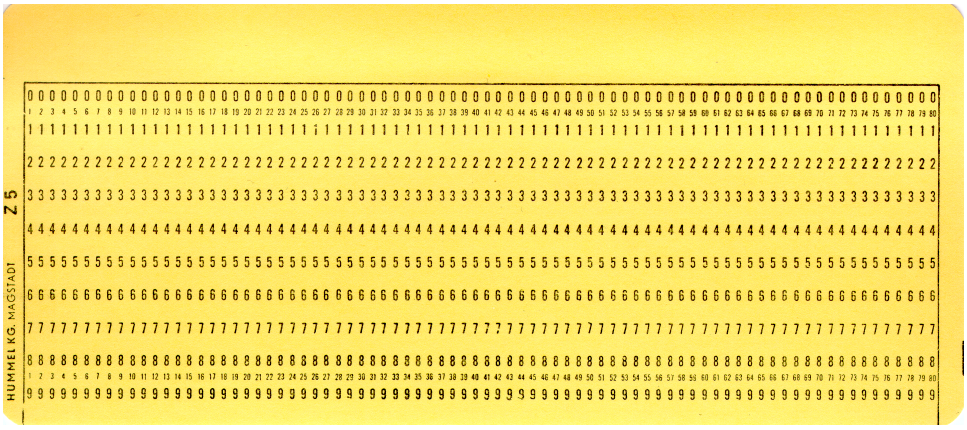
Fortran has been one of the first programming languages which made the step from Assembler programming to a high-level programming language. The name Fortran (formerly FORTRAN) stands for “formula translator” which means that the language was designed for the solution of mathematical problems from the very beginning. In Fortran 77 the concept of structured programming (which is the basic concept of INSEL) was introduced. Like writing an essay or a book in a specific language, it depends on the style of the author, whether the text is readable or not. So from the very beginning you should put some effort into the development of your programming style. Fortran allows you to write structured programs or to write some spaghetti code. We try to guide you to writing structured source code. So let’s go ahead.

Fortran character set

The Fortran character set contains the capital letters `A` to `Z`, the digits `0` to `9` and the special characters `= + - * / ( ) , . \ $ ’ :` and the space character. In comments any other character may be used. However, crashes of software have been observed only due to the use of some German special characters like ü, for instance. So, be careful and avoid non-ASCII characters wherever possible. Please notice also that the lower case letters `a` to `z` are not included in the Fortran 77 standard. But every newer compiler accepts these letters. A purist however would not use them.

Fortran source code underlies strict conventions which are based on the layout of punch cards, which were used in “historic” ages till the seventies and eighties of the last century. Probably the younger readers have never had the opportunity to see a punch card, so here is a picture of one of them.

Punch card



As you can see a punch card has 80 columns (numbered from 1 to 80 and not – like in C from 0 to 79). Columns 1 to 72 are used for code, while columns 73 to 80 are ignored by a Fortran 77 compiler. These columns have been used in former times for a systematic

numbering of the cards. Today they are no longer used. The use of columns 1 to 72 is not free but also underlies certain rules.

- Column 1 to 5 are reserved for labels, column one plays a special role: when a literal constant C is placed in column 1 the Fortran compiler interprets the corresponding line as comment, i. e., ignores everything written in that record.
- Column 6 is reserved for markers of continuation lines.
- Columns 7 to 72 are used for Fortran statements.

Labels can have up to five digits, at least one digit must be different from zero. You can think of labels as statement numbers.

Completely blank lines are treated like comment lines and are ignored by the Fortran compiler.

When the length of a statement exceeds the available space (column 7 to 72) the statement can be continued in the next record. In this case a continuation symbol must be used in column 6 of the continuation line. The symbol can be any symbol of the Fortran character set, except `0` and blank. We recommend to use the sign `&` which is not a Fortran 77 character but is accepted by all compilers as continuation marker.

Between elements of a statement one or more blank characters can be added for better readability. They are ignored (except in strings or literal constants) by the Fortran compiler.

12.1.1 The principle form of a Fortran program

In Fortran there is no special end of statement symbol (like ; in C, for example).

A Fortran program always begins with a statement which describes the type of the code. A main program for example starts with the (optional) statement

```
PROGRAM name
```

where `name` can be any allowed Fortran name, like `TEST`, for instance. A variable declaration part follows and then a set of executable statements. The end of a Fortran program, i. e., the last statement of a program must always be the

```
END
```

statement.

Now you know the key elements of a Fortran program – we will soon talk about different statements. Let us make a first example.

```
PROGRAM HELLO
PRINT*,"Hello, world!"
END
```

This program prints the string "Hello, world!" on the computer screen. It is not a Windows program but expects to run in a DOS box or a Terminal where it can write to.

Source code

Before you continue, you should copy the program – the text is called source code – and write a file named `hello.f`, for example, by using a text editor.

You can use any text editor which is available on the market – and there are plenty. If you have no text editor at hand, you can use Notepad, which is a simple text editor that is part of Windows, or TextEdit, which is a similar editor for Mac OS. Please notice that Microsoft's Word or Apple's Pages are programs for typesetting but not for source code development. In the end you should learn how to use a professional text editor.

For many years, our prefered editor has been Kedit of Mansfield Software Group, Inc. (www.kedit.com). Unfortunately, it is available for the Windows platform only and in times of Mac OS, Linux etc...

A candidate now proposed by the INSEL developers is jEdit (www.jedit.org), a platform-independent Java-based editor.

Whatever editor you use, in the end you should get a file named `hello.f` (or similar).

Compiler and linker

The next step is to compile and link the program. What you need to do this is software: a Fortran compiler and a linker. Like in the case of a text editor, plenty of compilers for all kinds of languages are available on the market.

It is probably a good idea to use the gcc (gcc.gnu.org) and gfortran (www.gfortran.org) open-source compilers, which are most supported by INSEL. Please find out which Fortran compiler you wish to use and compile and link `hello.f`.

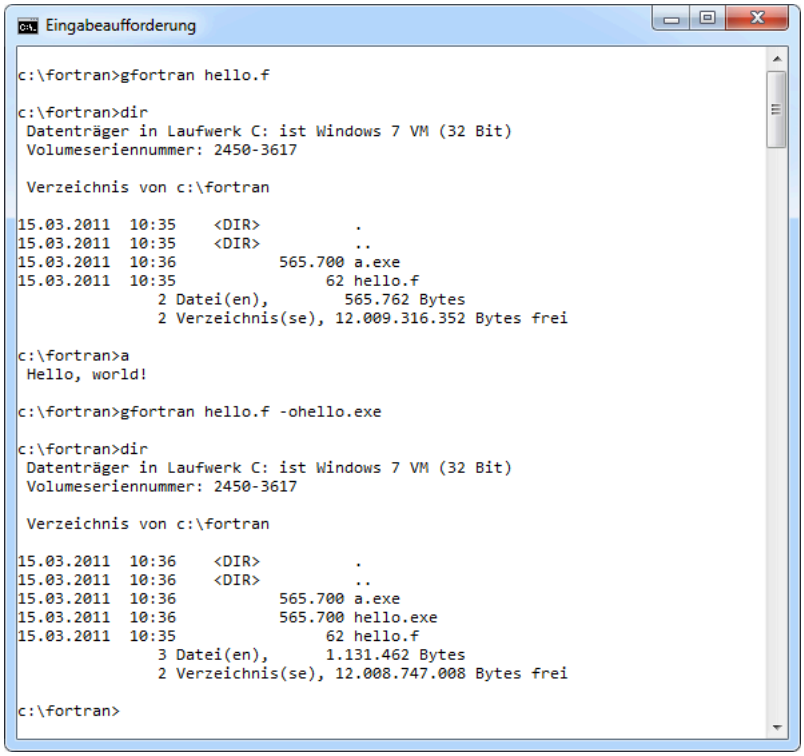
Object code and executable code

The compiler will then generate object code in a file called `hello.obj` or `hello.o` and the linker will generate executable code and write this to a file `hello.exe` (most typical under Windows) or just `hello` or similar. When everything works, you can type `hello` at the DOS or terminal prompt, the program `HELLO` will execute and display `Hello, world!` and terminate.

The program does not do much, but once it works you can be sure to have a working environment.

If you are new to programming, we recommend NOT to continue with the crash course before you have seen the hello world example running in real.

If you followed our recommendation and have installed gfortran you will end up with something similar to the following screen:



You can see that by default gfortran names the executable `a.exe`. If you prefer to give it a different name you can use the option `-ohello.exe`, for example (o for output). You can also see that there is no object code in the directory. If you wish to only compile and not link the source code you can call the compiler by `gfortran -c hello.f` and you will find the object code in the directory.

Before we start with the syntax of the Fortran language, let us recall a few remarks that have already been made about structured programming earlier in this Tutorial.

Structured programming

In the seventies and eighties programmers started to understand that structured programming techniques needed to be developed in order to keep software maintainable. Programmers were challenged to write more transparent programs.

Structured programming was an attempt to restrict software developers to make use only of three different program structures:

- Straight sequences of statements which are executed in linear order. Statements can either be simple statements or encapsulated collections of statements which follow the rules of structured programming.
- If-then-else statements which allow branching in the code with a definite target where the two branches come together again, so that the structure can be regarded as a single kind-of-macro operation.
- Iteration loops which allow for the programmatical execution of code in well-defined repeat structures.

Single-entry, single-exit

As a consequence of these rules structured program parts have a well-defined and unique entry point and one – and only one – unique exit point. Sometimes this rule is referred to as the single-entry single-exit principle. This rule restricts very much the use of a sort of crude statement which allows jumps into any part of the code, the `GO TO` statement. In structured programming it is not forbidden to use a `GO TO` but its use is restricted to very special local operations – by agreement.

Encapsulation of code is a key idea of structured programming: develop your programs from simple statements to bigger structures. Some Fortran concepts like `FUNCTIONS`, `SUBROUTINES` help you to follow this concept.

But now it's time to dive into some of the most the important syntax rules of the Fortran 77 programming language.

12.1.2 Fortran data types

Like any programming language Fortran has its own set of supported data types. The most important data types are `INTEGER`, `REAL`, `DOUBLE PRECISION`, `LOGICAL`, and `CHARACTER`.

All variables that are used by a program should be declared in the declaration part of the program. Fortran has some rules which implicitly relate variables to specific data types by default. This may seem practical because it releases you from having to declare all variables that you use. But experience shows that this property of the Fortran language can give you a real tough time sometimes, in particular in error detection. We recommend to just forget this property and add to ALL your Fortran programs the statement

```
IMPLICIT NONE
```

following either the `PROGRAM` statement or any other code type statement, like `FUNCTION...` or `SUBROUTINE...` When you use an `IMPLICIT NONE` statement the Fortran compiler does not accept any variables that are not declared and this makes code development much safer.

So now the question is "What is the difference between the several data types and how can I declare them?" Let us first look at the principle difference.

- `INTEGER` variables are used for the set of integers, i. e., 0, ± 1, ± 2 etc. The minimum and maximum value an `INTEGER` variable depends on the size of the variable in the computer's memory. Usually four bytes represent an `INTEGER` variable. In this case the possible range is from -2147483648 to $+2147483647$.
- `REAL` variables can represent any real number. The accuracy of a `REAL` number depends on the number of bytes which represent the `REAL` number in the computer's memory – four bytes, i. e., 32 bits is today's most common size, the trend is towards 64 bits, i. e., 8 bytes. A four byte `REAL` covers the range from $-3.402823\text{E}{+}38$ to $+3.402823\text{E}{+}38$. Please notice that the number of significant digits is about 7, which is not very accurate and can lead to numerical problems in sensitive iterations. INSEL therefore compiles all Fortran sources with the options `-fdefault-real-8 -fdefault-double-8`, so that in INSEL blocks every `REAL` has 8 bytes, like `DOUBLE PRECISION` and the C/C++ type `double`.
- `DOUBLE PRECISION` variables are similar to the `REAL` type but use twice the number of bytes of `REAL` variables for the representation of the current value. Their significant number of digits is of order 15, the numbers cover a range from -10^{308} to $+10^{308}$.
- `LOGICAL` is a boolean data type which can have one of two values only, either it is `TRUE` (1) or `FALSE` (0).
- The `CHARACTER` data type is used for alphanumerical strings.

Literal constants

We have used the string "Hello, world!" already in our first Fortran program. But we used it as a constant value – a so-called literal constant – not as a variable `CHARACTER` data type.

There are more data types available in the Fortran 77 standard, but we will concentrate on the mentioned ones – and you will see that you can probably solve more than 99.9 percent of your numerical problems with these data types – this "statement" depends a little on your level of computing experience, of course.

Now that we know which data types are important, let's have a look at how can they be declared and used. We have seen that Fortran is a very formal language in the sense that it has clear rules about the format of every source code line. Since statements must be written in the range of column 7 to 72, the declaration of variables also starts in column 7. The mentioned data types can be declared by statements – starting in column 7 – like

```
INTEGER          I1, I2
REAL             R1, R2
DOUBLE PRECISION D1, D2
LOGICAL          L1, L2
CHARACTER*80     C1, C2
```

INTEGER, REAL, DOUBLE PRECISION, LOGICAL, and CHARACTER are key words since they form basic elements of the programming language Fortran. I1, I2, R1, R2, D1, D2, L1, L2, C1, and C2 are variable names and are all valid. There are rules for names: A name (in strict Fortran 77) cannot have more than 6 characters (all newer Fortran compilers accept longer names) and the first character may not be a digit, i. e., 1TEST would not be a valid name for a Fortran variable name but TEST1 would be valid.

Initial values

You cannot rely on the idea that all declared variables automatically have a reasonable initial value. Some (most) compilers initialise declared variables with a value zero or blank (in the CHARACTER case) but in the end it is up to you to ensure that the variables have reasonable values at any time.

One simple method to initialise variables is to include an initial value in the statement which declares a variable. If for example you modify the above example and write

```
INTEGER          I1 /0/,       I2 /0/
REAL             R1 /0.0/,     R2 /0.0/
DOUBLE PRECISION D1 /0.0/,     D2 /0.0/
LOGICAL          L1 /.FALSE./, L2 /.FALSE./
CHARACTER*80     C1 /" "/,     C2 /" "/
```

you can be sure – independent of any compiler's behavior – that your numerical variables are initialised by zero, the logical variables are initialised with 0 (false – please observe that Fortran uses dots for logicals like .TRUE. and .FALSE.) and the CHARACTER strings are initialised with a blank.

INSEL does the opposite of a zero initialisation on purpose: Fortran blocks are compiled with `-finit-real=snan -finit-integer=-999999999`, so that a variable which is used before it got a value is NaN (not a number) or -999999999, and the mistake becomes visible. Please note also that a variable with an initial value in its declaration is initialised only once, when the program starts, and keeps its value between calls of a subroutine. In INSEL blocks this is rarely what you want – see Remark 4 at the end of this crash course.

All the data declared in the example can be used as variables, i. e., their values can be changed during the execution of the program where they are used. If you want to define constants which cannot be changed – by accident for example – during program execution you can declare them as PARAMETER. The corresponding statement is

```
PARAMETER (name1=value1, name2=value2)
```

In this case the variables name1 and name2 are initialised by value1 and value2, respectively, but it is impossible to change the values of name1 or name2 during program execution. It is necessary to declare the variables name1 and name2 before the PARAMETER statement. For example,

```
REAL       PI, KELVIN
PARAMETER (PI = 3.14159265, KELVIN = 273.15)
```

Vectors

All declared variables can be either scalar (like we have declared all of them) or have a dimension greater than zero. In order to declare one-dimensional vectors with ten elements, for example, we can simply write

```
INTEGER          I(10)
REAL             R(10)
DOUBLE PRECISION D(10)
LOGICAL          L(10)
CHARACTER*80     C(10)
```

We can then access the variables via their index like `I(1)`, `I(2)`, ... `I(10)`, for example.

Two dimensional variables could be declared as

```
INTEGER          I(10,10)
REAL             R(10,10)
DOUBLE PRECISION D(10,10)
LOGICAL          L(10,10)
CHARACTER*80     C(10,10)
```

definining 10×10 matrices of `INTEGER`s, `REAL`s, and so forth. It should be obvious how the elements can be accessed in this case.

Set operations

The next question is "How can we change the values of a variable programmatically?." The answer is, by using a set operation. Set operations look like equations: on the left side of the equation we get the result, on the right side of the equation we write the operation.

A statement like

```
I1 = I1 + 1
```

will perhaps surprise you when you have never seen such code before. Mathematically spoken, the equation is complete nonsense: How can `I1` be equal to `I1 + 1`? – impossible. In Fortran (and most other programming languages) `I1 = I1 + 1` means: Well, before the operation `I1` has a value, let's say zero. The statement `I1 = I1 + 1` then means, take the value of `I1`, add `1` to it and save it under the name `I1` again. This means, that if `I1` had a value zero before the execution of the statement `I1 = I1 + 1` then `I1` will have a value of one after excution of the statement. Some programming languages express this as `I1 <- I1 + 1`, or `I1++` or similar, but they all mean the same operation: Add one to the current value of the `INTEGER` variable `I1`.

Basic operations

Now we know, how we can define variables, how to initialise them, and how to perform basic operations with them like add

```
I1 = I1 + I2
```

or subtract

```
I1 = 1
I2 = 2
I1 = I1 - I2
```

multiply

```
R1 = 1.5
R2 = 2.5
R1 = R1 * R2
```

or divide

```
R1 = 1.5
R2 = 2.5
R1 = R1 / R2
```

or exponentiate

```
R1 = 1.5
R2 = 2.5
R1 = R1 ** R2
```

numerical Fortran variables. With variables of type `LOGICAL` or `CHARACTER` these numerical operations are not possible.

Division by zero

The division example shows the first danger already: What happens for example if `R2` is equal to zero? It is most probable that the operating system generates and error message like “Exception error: Divison by zero” and terminates the execution of the program. We will in a few moments see how this behavior can be avoided.

What if we mix the basic operations in one expression like

```
D1 = I1 + I2 * R1 ** R2
```

Then the sequence of operations – and hence the result `D1` – depends on the “order” of the operator. This order in Fortran is the natural order: `**` highest, then `*` and `/`, then `+` and `-`, then from left to right. As in school mathematics the order can be changed by using parentheses
`( ... )`.

Hence, the above statement `D1 = I1 + I2 * R1 ** R2` is equivalent to
`D1 = I1 + (I2 * (R1 ** R2))`.

The `+` and the `-` symbols have two meanings: they represent operations of adding and subtracting and they can be used as signs. In this case too, the higher order operator is executed first, i. e., `D1 = -R1 ** R2` is equivalent to `D1 = -(R1 ** R2)`.

Generally spoken, all the above statements have the form

```
variableName = expression
```

If `variableName` on the left side is a numerical data type like `INTEGER`, `REAL`, or `DOUBLE PRECISION` then `expression` must evaluate to a numerical value. For variable names with type `LOGICAL` the statement must result in either `.TRUE.` or `.FALSE.`, and if `variableName` is a `CHARACTER` then `expression` must evaluate to a `CHARACTER`.

The expression can mix different numerical data type variables. In this case before the operating system performs an operation the variable of lower order is converted to the data type of the other operand with higher order. For the statement
D1 = I1 + I2 * R1 ** R2 this means that at first R1 ** R2 is calculated – both have the same data type, no problem – then I2 is multiplied by the result of R1 ** R2 – since I1 is an INTEGER and the result R1 ** R2 is a REAL before the operation I1 is converted to a REAL variable – and so on. The result will be a REAL variable but shall be stored in the DOUBLE PRECISION variable D1, so that the result is converted to DOUBLE PRECISION automatically. One exception to this rule is that if an exponent is of type INTEGER and the operand is of any other numerical type then no conversion of the INTEGER will be made.

CONTINUE, PAUSE, STOP

Okay. So far we can define some variables, perform some basic operations on them and show some values of the computer's screen. That means we can make sequences of statements. To complete the sequence structure there are three more statements which do not allow any branching in the program. The first is the

```
CONTINUE
```

statement. It does nothing but continue, which means that execution will continue with the next statement in the sequence. It is good programming style to use a CONTINUE statement together with labels (see later). The second statement is the

```
PAUSE x
```

statement. In this case program execution will pause and x will be displayed on screen – x can either be an integer in the range of 0 to 99999 or x can be a string like "We have paused the program xy"

When program execution is paused the program can only be continued by a user, who can press any key of the keyboard to continue. The last of the three statements is the

```
STOP x
```

statement, which acts like the PAUSE statement in displaying but which terminates the program execution like the END statement we have already used earlier.

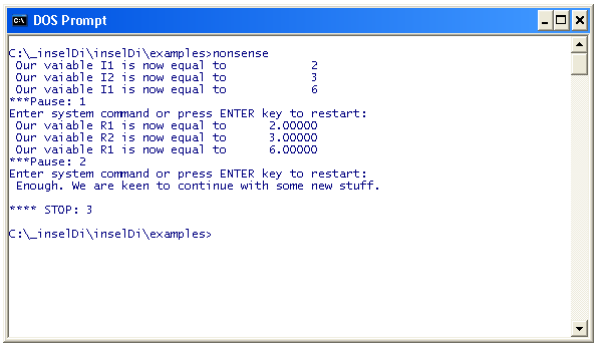
Exercise 12.1 Write some code which declares and initialise some variables, perform some operations with the data, display some results via the PRINT statement (PRINT *,list accepts a list of data which can have any data type – the elements in the list must be separated by a comma).

We used this code to test what we wrote:

```
      PROGRAM NONSENSE
C     We follow our teachers and use
      IMPLICIT NONE

C     Now we declare some variables
```

```
      INTEGER      I1 /0/, I2 /0/
      REAL         R1 /0.0/, R2 /0.0/
      CHARACTER*80 STRING /' '/

C     And now we do some nonsense
      I1 = 2
      PRINT *,'Our vaiable I1 is now equal to ',I1
      I2 = 3
      PRINT *,'Our vaiable I2 is now equal to ',I2
      I1 = I1 * I2
      PRINT *,'Our vaiable I1 is now equal to ',I1

      PAUSE 1

C     And now the same with reals
      R1 = 2.0
      PRINT *,'Our vaiable R1 is now equal to ',R1
      R2 = 3.0
      PRINT *,'Our vaiable R2 is now equal to ',R2
      R1 = R1 * R2
      PRINT *,'Our vaiable R1 is now equal to ',R1

      PAUSE 2

C     Enough
      STRING = 'Enough. We are keen to continue with some new stuff.'
      PRINT *,STRING

      STOP 3
      END
```

Result This is the screen shot after completing the program.



12.1.3 If-Then-Else structures

As the name of this section points out already the most typical statement of the if-then-else structure is the if-then-else statement. Its most important form is

```
IF (condition) THEN
```

```
   expression1
ELSE
   expression2
END IF
```

Here, `condition` must evaluate to a `LOGICAL` which is either `.TRUE.` or `.FALSE.` and `expression1` and `expression2` can be any set of structured statements, i. e., both expressions can use sequence structures, if-then-else structures and the later discussed loop structures. We start with the condition first.

Conditions

Conditions are always based on logical comparisons, like equal, not equal, greater than, less than, and so on. Logical comparisons can be combined with logical operators like and, or, and so forth.

In Fortran there are six comparison operators

```
.EQ.
.NE.
.GT.
.GE.
.LT.
.LE.
```

with the meanings `.EQ.` = equal, `.NE.` = not equal, `.GT.` = greater than, `.GE.` = greater or equal, `.LT.` = less than, and `.LE.` = less or equal.

Comparisons have the general form

```
expressionA operator expressionB
```

The expressions can be either both numerical or both textual.

As a first example let us come back to the above mentioned problem of division by zero.

```
R1 = 1.5
R2 = 2.5
IF (R2 .NE. 0.0) THEN
   R1 = R1 / R2
ELSE
   PRINT *,"Hoppla, divion by zero."
   PRINT *,"Our way out: we do nothing"
END IF
```

Here of course, we know that `R2` cannot be equal to zero, but – the example should be self-explaining.

Please notice, how we "pretty-print" our source code: whenever a branching statement like `IF` occurs the following text is indented by three bytes – some programmers prefer two or four bytes. When `expression1` is complete we move back three bytes to the previous position, the next expression is completely indented three bytes again, and so forth. I have seen many people write the same code in such a shape:

```
R1=1.5
R2=2.5
IF(R2.NE.0.0)THEN
R1=R1/R2
ELSE
PRINT*,"Hoppla, divion by zero."
PRINT*,"Our way out: we do nothing"
ENDIF
```

It's right, the program does the same, but I personally get a goose skin when I see code like this. To me it appears like the type setter hands out a wonderful book like "The Master and Margarita" of Michail Bulgakov without using spaces.

At the end of this crash course we have collected some guidelines which we suggest to follow in pretty-printing.

There are three more types of the `IF` statement which are used frequently. All of them do not provide anything really new but could also be expressed with the above introduced if-then-else structure. We mention them briefly, anyway.

The first is useful in very simple cases, i. e., when only one statement depends on the condition. The general form is then

```
IF (condition) statement
```

which means if the `condition` is `.TRUE.` the statement will be executed, otherwise not. Please notice that this statement is equivalent to writing

```
IF (condition) THEN
   statement
END IF
```

which we prefer.

The second type of `IF` statement is useful when one out of several options is to be chosen. The general form is

```
IF (condition1) THEN
   expression1
ELSE IF (condition2) THEN
   expression2
ELSE IF (condition3) THEN
   expression3
ELSE
   expression4
END IF
```

Meaning and use of the statement should be self-explaining. This form is preferable to the equivalent form

```
IF (condition1) THEN
   expression1
```

```
ELSE
   IF (condition2) THEN
      expression2
   ELSE
      IF (condition3) THEN
         expression3
      ELSE
         expression4
      END IF
   END IF
END IF
```

Please notice the different use of the `END IF`s. The third form is something rather different, it defines a so-called arithmetic `GO TO` statement. The simple `GO TO` statement – you remember, a statement which should be used very carefully – is

```
GO TO label
```

When such a statement appears in the source code the program continues execution at the specified label.

Labels

Labels have been mentioned previously, but now let us look at their meaning. A label, as we have heard already, by definition is a number in the first five columns of a source code record with a value between `1` and `99999` – a label number must be unique in a program unit, so it can be used only once. As we have also heard before, a labeled record should always have only a `CONTINUE` statement. Hence, a labeled record has the form

```
123   CONTINUE
```

where `123` is an example for a label.

GO TO

In this example, when the program executes the statement

```
GO TO 123
```

it continues execution at the statement labeled `123`. It should be clear, that by making extensive use of such jumps it is possible to write absolutely unreadable spaghetti code. Hence try to avoid `GO TO` statements wherever you can.

To return to the third type of branching `IF` statements, the definition of the arithmetic `GO TO` statement is

```
GO TO (label1, label2, ... labelN) I
```

Here `I` is an integer greater than zero (it is possible that `I` is an expression which evaluates to an integer greater than zero). When `I` is or evaluates to a value one, program execution branches to `label1`, when `I` is or evaluates to a value two, program execution branches to `label2`, and so forth.

You may ask, what happens if I is greater than N or less than one. In the documentation of your Fortran compiler – I'm sure you'll find the answer – but we recommend that you simply don't let this happen, i. e., write your own code like

```
      IF (I .GE. 1 .AND. I .LE. 5) THEN
         GO TO (1,2,3,4,5) I
      ELSE
         STOP "BLUNDER in program part..."
      END IF
1     CONTINUE
C     ...
2     CONTINUE
C     ...
C     et cetera
```

In Fortran there are some even more adventure-like statements, for example one where the adress to be jumped to can be calculated. Please, ignore them, do no even look at them, concentrate on valuable things. Write program branching statements in your code with the explained options – there is absolutely no need for more `IF` or `GO TO` statements.

Loops

But one last group of statements is definitively important, that of loops. This, after having discussed sequence and if-then-else structures is the last required concept in programming. Once you are fond of making use of sequential structures, if-then-else structures, and loop structures, you can solve any numerical problem – from the programming language point of view – that will ever exist. This has been proven in a very general way and published in a paper by the father of the structured programming concept Edsger W. Dijkstra.

Imagine, you want to write a program which counts from 1 to 10 and displays the numbers on screen, lets call it the 1-to-10 example. We could write a trivial program like

```
PROGRAM ONE2TEN
IMPLICIT NONE
INTEGER I
I = 1
PRINT *,I
I = 2
PRINT *,I
I = 3
PRINT *,I
I = 4
PRINT *,I
I = 5
PRINT *,I
I = 6
PRINT *,I
I = 7
PRINT *,I
I = 8
PRINT *,I
```

```
I = 9
PRINT *,I
I = 10
PRINT *,I
STOP
END
```

DO statement

There is a statement which automises this idea – the DO statement. For the example we could use

```
DO I = 1,10,1
   PRINT *,I
END DO
```

In general form the statement is

```
DO I = startValue, finalValue [, increment]
   expression
END DO
```

where `startValue` is a numerical value for the first execution of the loop, on every call the loop variable – `I` in this case – is incremented by the given `increment` (the `increment` is optional – when it is omitted it defaults to 1) and set to the current value. The loop runs for the last time when the final value is reached and then stops. Please find out what the value of `I` is after the loop finished (`PRINT` statement after the `END DO`) and try to imagine how the `DO` statement works internally.

Notice again, that we use the `DO` keyword to indent the affected records by three bytes in our pretty-printing, too, and that the `END DO` returns back to the last used column.

Of course, `startValue`, `finalValue`, and `increment` can be of any reasonable numerical data type, like `INTEGER`, `REAL`, or `DOUBLE PRECISION`.

DO WHILE statement

Fortran 77 does not support other `DO` constructions, but the `DO WHILE` structure is supported by almost all Fortran compilers and can be included in structured program code. The Fortran language specifications (later than 1977) say that a `DO WHILE` statement has the general form

```
DO WHILE (condition)
   expression
END DO
```

Here, `condition` can be any logical condition which evaluates to a `LOGICAL`. As long as `condition` is `.TRUE.` the `DO` loop is executed, when condition is `.FALSE.` the `DO` loop is terminated and program execution continues with the next executable statement following the `END DO`. Please notice, that the `condition` is evaluated before a `DO` loop cycle is initiated.

Exercise 12.2 As an example for the `DO WHILE` statement let us formulate the 1-to-10 example with it. Maybe you'd like to try on your own, before you look at our code.

Solution

```
PROGRAM  DOWHILE
IMPLICIT NONE
INTEGER  I
LOGICAL  COND
I    = 1
COND = .TRUE.
DO WHILE (COND)
   PRINT *,I
   IF (I .LT. 10) THEN
      I = I + 1
   ELSE
      COND = .FALSE.
   END IF
END DO
STOP
END
```

EXIT and CYCLE

Two statements related to `DO WHILE` loops are importrant to know: the `EXIT` and the `CYCLE` statement. When you want to immediately jump completely out of a `DO` loop for some reason, you can use the `EXIT` statement. The `CYCLE` statement allows you to jump to the `END DO` statement and next incremented value (if any is left) without terminating the loop.

Now you know enough about Fortran programming that you can – in principle – solve all numerical computer problems. Maybe you will be surprised, but in our view Fortran is that simple. It is a language which can be learnt in one day. But so far we were mostly concerned with reading Fortran. What about writing Fortran? When you really want to write Fortran you need to practise and solve more complex problems than just counting from one to ten – this is nice for a teaching course, but is toy ground.

Practise!

The best way to become familiar with all the formal stuff is to solve a more complex problem from scratch. Maybe you are currently working on a new component model of some I-don't-know machine. Then feel lucky to develop and implement your ideas in a Fortran program – best practise.

When you have no such problem, but are in the lucky situation of having a spare day or two days extra time, maybe you try to solve the "eight-queens-problem" with a Fortran program – this was actually the way the INSEL author learnt programming at the University of Frankfurt in Germany in the late seventies.

Anecdote

Can you imagine what programming at a German university meant in the late nineteen-seventies?

Students got a pack of punch cards and went to a punch card machine. Via a keyboard the machine punched some rectangular holes into the punch card – one set of holes in the current column per key stroke. At the end of the writing-a-program process the student took his pack of punch cards, put it into a mailbox (hardware!) of the computer center and the next day! found some endless paper print out in a book shelf with

whatever was programmed in the punch card pack – a printout of the source code and what was thought might be the solution to the eight-queens-problem. And when there was (at least) one little mistake? A lost day, find the error and try again for tomorrow, that was the "debugging experience" - tough.

You don't know what the eight-queens-problem is?

Eight-queens problem

Think of a chess board which has eight-by-eight squares. A chess queen can move any number of vacant squares, horizontally, vertically, or diagonally. Whenever there is some other chess piece on one of the reachable squares it can be captured. The eight-queens-problem is to place eight chess queens on a chess board so that no queen can capture any other queen. As far as I remember there are 92 solutions.

By the way, finding even one single solution on a real chess board without the help of a computer program is rather difficult – try it!

We are now almost through with our crash course through Fortran and its programming concepts – fair enough, we should say with what we think are the essential programming concepts. Now those of you who know more about programming will probably protest:

- What? We learnt nothing about input/output handling but the simple `PRINT` statement.
- Our programs will be much more complex than this simple stuff and will consist of a collection of units. We want to exchange data between these units and Fortran has some data exchange concepts like `COMMON` blocks and `BLOCK DATA`. We have heard nothing about them.
- This space is intentionally left free . . .

Yes, you are all right. But here comes our but:

The aim of this Fortran course is not to let you develop applications from scratch but tries to guide you through the concepts of the simulation environment INSEL.

Input and output

INSEL provides blocks for input/output operations. There is no need for an INSEL user, nor for an INSEL programmer who works on the level of source code development to implement basic input/output operations. Input is handled completely by INSEL, i. e., INSEL is responsible for data input. For the advanced INSEL programmer there is some information about input/output and file handling in the section Advanced INSEL block programming.

The only thing with which you will be confronted as someone who wants to write INSEL extensions or new blocks is the question how output data like error messages can be generated.

Error handling

INSEL as an integrated simulation environment provides a concept for error handling – the INSEL message system. In this Module you will learn how you can communicate

with this message system. We hope that this concept satisfies all your needs for input/output communication.

The second point is that we want you to work in the INSEL environment, i. e., you shall not feel left alone but guided. In the ideal case, you shall just not have to write complex solutions but just some fixes for situations where you feel that INSEL as it is is not perfectly tailored to your requirements.

INSEL provides a well-defined interface which is based on Fortran's `SUBROUTINE` concept. In a few moments we will explain how Fortran subroutines can be used and how the interface looks like.

12.1.4 Structuring program projects

One of the key concepts in structured programming is the top-down development of code. It is a method where one complex problem is split up into several smaller problems. These part problems then are split again so that at the end of the chain code can be written for those relatively lower level problems. All structured programming languages provide concepts for top-down development and conventions for smaller code units which can be used as solutions to partial problems. The units are usually called procedures. In the end a complex problem solution will consist of many procedures which altogether form the final program.

Fortran knows four types of procedures:

- Intrinsic functions
- Statement functions
- External functions
- Subroutines

Intrinsic functions

Intrinsic functions are already included in the Fortran language standard. They provide functions for type conversion, arithmetic functions, mathematical functions, and `CHARACTER` functions.

All functions must have a specific type like `INTEGER`, `REAL`, `DOUBLE PRECISION`, `LOGICAL`, or `CHARACTER`, for example, and they will return a value of that data type. We list some of the most important intrinsic functions now – for a complete overview please refer to your Fortran compiler's reference manual.

`INT(x)` Converts a `REAL` or `DOUBLE PRECISION` variable `x` into an `INTEGER` without rounding, i. e., the decimal fraction is cut off.

`REAL(x)` Converts an `INTEGER` or `DOUBLE PRECISION` variable `x` into a `REAL`.

`FLOAT(x)` Converts an `INTEGER` variable `x` into a `REAL`.

`DBLE(x)` Converts an `INTEGER` or `REAL` variable `x` into a `DOUBLE PRECISION`.

`ABS(x)` Returns the absolute value of an `INTEGER` or `REAL` variable.

`ANINT(x)` Returns the rounded `INTEGER` value of a `REAL` variable `x`.

`SQRT(x)` Returns the sqaure root of a `REAL` variable `x`.

`ACOS(x)` Returns the arc cosine of a `REAL` variable `x`.

`ASIN(x)` Returns the arc sine of a `REAL` variable `x`.

`ATAN(x)` Returns the arc tangent of a `REAL` variable `x`.

`COS(x)` Returns the cosine of a `REAL` variable `x`.

`COSH(x)` Returns the hyperbolic cosine of a `REAL` variable `x`.

`EXP(x)` Returns the exponential function $\exp(x)$ of a `REAL` variable `x`.

`LOG(x)` Returns the natural (base e) logarithm of a `REAL` variable `x`.

`LOG10(x)` Returns the decade (base 10) logarithm of a `REAL` variable `x`.

`SIN(x)` Returns the sine of a `REAL` variable `x`.

`SINH(x)` Returns the hyperbolic sine of a `REAL` variable `x`.

`TAN(x)` Returns the tangent of a `REAL` variable `x`.

`TANH(x)` Returns the hyperbolic tangent of a `REAL` variable `x`.

Statement functions

Statement functions have the general form

```
name(parameter) = formula
```

where `parameter` is a list of one or more variable names, separated by commas. These variables can then be used in a formula to calculate the value of the variable `name`, which must be defined in the decalartion section of the unit before the statement function. The formula may use only one statement.

An example for the theorem of Pythagoras would be

```
REAL A,B,HYPO
HYPO(A,B) = SQRT(A**2 + B**2)
```

where `A` and `B` would be the side lenths of a right triangle, and `HYPO` would be the length of the hypotenuse.

Statement function may only be used in the program unit where they are defined.

External functions

External functions are formulated as separate program units and can be compiled individually. They can be called by any other program unit. The first statment of an external function specifies the type of the program. So far, we have only worked with program units of type `PROGRAM`.

The first statement of an external function has the general form

```
type FUNCTION name(parameterList)
```

where type specifies the type of the return value of the function, INTEGER or REAL, for instance, name defines the name under which the function can be called then. At the same time a variable of type type named name is defined. A value must be assigned to this variable in the function, this value is the return value of the function then. Hence, a function returns exactly one value – the function value.

The parameterList in parentheses consists of variable names, separated by commas. The list can be empty, but the parentheses may not be omitted.

The code of the FUNCTION is equivalent to that of a PROGRAM unit – it has a declaration section (again, the first statement should be the IMPLICIT NONE statement), followed by a number of exectutable statements. The last statements must be a RETURN and an END statement. The RETURN statement causes the program to return to the calling program, while the END statement signals the end of the program unit to the compiler.

Subroutines

Subroutines can return more than one value. Actually, they can exchange an arbitrary number of variables between the calling program and the called program, i. e., the subroutine. Similar to a FUNCTION, a SUBROUTINE starts with a statement which declares the type of the program unit. The general form of a subroutine type statement is

```
SUBROUTINE name(parameterList)
```

The variable name fixes the name of the routine, parameterList stands for the list of parameters, again separated by commas. The term (parameterList) is usually called the subroutines interface. While a FUNCTION could be used in a calling programm just by the FUNCTION name, calling a SUBROUTINE requires a CALL statement of the form

```
CALL name(parameterList)
```

where name is the name of the SUBROUINE and parameterList is a list of variables as expected by the subroutine.

The structure of a subroutine is as usual: unit declaration, IMPLICIT NONE statement, variable declarations, executable statements, and end with a RETURN statement and an END statement.

Fortran distiguishes strictly between functions and subroutines. However, nowadays it is common practise to think of a function as a function which returns a value, while a subroutine is a function which does not return a value – all matter of taste.

Puhh! After this little tough bit of theory it's time for some explaining examples.

Example intrinsic function

The use of the intrinsic functions should be ovious. If, for example you want to use the square root of a variable X in a statement, the code looks like

```
LEFT = BLABLA + SQRT(X) - DINGS
```

Example statement function

So, we start with a second statement function for the conversion of temperature data from degrees Celsius to Kelvin. If we assume a temperature value TC is given in degrees Celsius then we would like to calculate the corresponding value in kelvin. The formula is simple

```
TK = TC + 273.15
```

First, we need a name for the function – how about TK? The only required parameter is the temperature in degrees Celsius TC. So, the statement function is expressed as

```
TK(TC) = TC + 273.15
```

Exercise 12.3 Write a small program which makes use of this statement function.

Solution We wrote this one:

```
PROGRAM TC2TK
REAL      TC,TK
TK(TC) = TC + 273.15
DO TC = 0.0,100.0,10.0
   PRINT *,TC,TK(TC)
END DO
STOP
END
```

resulting in the obvious output

Statement functions are not often used, because their main drawback is that they can only be used locally in a program unit. If another program unit wants to make use of the same statement function, a new definition of the statement function is required.

FUNCTIONs

A more general way is to use a FUNCTION – as we have learnt, a FUNCTION can be used from all other program units without redefinition (assuming that it is "linked" to the executable – we come back to this point in a minute).

Exercise 12.4 Implement the above example statement function in a separate Fortran function and compile it.

Solution Our solution is:

```
REAL FUNCTION FUNTC2TK(TC)
IMPLICIT NONE
REAL TC
FUNTC2TK = TC + 273.15
RETURN
END
```

We saved our code in a file named `funTc2Tk.f`. And now let's see how we can use our function from another program unit.

Writing the calling code is straight forward.

```
PROGRAM USETC2TK
IMPLICIT NONE
REAL TC,FUNTC2TK
DO TC = 0.0,100.0,10.0
   PRINT *,FUNTC2TK(TC)
END DO
STOP
END
```

Again, we gave the file the same name that we used for the function, hence called it `useTc2Tk.f`. This can be regarded as a general recommendation to follow, since it makes it very clear that the file named `useTc2Tk.f` contains a program named `USETC2TK`. But please keep in mind that the two names are completely independent – one is the file name and the other one is the program name.

If we try to compile and link `useTc2Tk` in the usual way, we get an error message that the symbol `TC2TK` is missing. What happens?

When you study the Fortran code in file `useTc2Tk.f` and compile it only (e. g., using the command `gfortran -c useTC2TK.f`) you will recognize that the compiler is completely satisfied with the code, i. e., understands that we want to use a `FUNCTION` named `TC2TK` and leaves it up to the linker to locate the function. But how shall the linker find the function? We have to "tell" it, where the function resides – namely in the file `funTc2Tk.o`. So the command line which generates the executable `useTC2TK.exe` looks like

```
gfortran useTc2Tk.f funtc2tk.o -ouseTc2Tk.exe
```

assuming that you have already compiled `funTc2Tk.f`.

Case sensitive vs. not case sensitive

Maybe you have recognized that we were not very straight with the use of lower-case and upper-case letters. In our case it does not matter very much, since Fortran and Windows both are not case sensitive. But it is a bad habit to mix cases – like we do sometimes – because other operating systems like Linux, for example, are case-sensitive so that `useTc2Tk.f` and `useTC2TK.f` are really two different file names.

Re-writing our conversion function `FUNTC2TK` in a `SUBROUTINE` is not difficult now.

```
SUBROUTINE SUBTC2TK(TC,TK)
IMPLICIT NONE
REAL TC,TK
TK = TC + 273.15
RETURN
END
```

As calling program we used

```
PROGRAM CALLTC2TK
IMPLICIT NONE
REAL TC,TK
DO TC = 0.0,100.0,10.0
```

```
      CALL SUBTC2TK(TC,TK)
      PRINT *,TC,TK
END DO
STOP
END
```

and compiled and linked it analogue to `funTc2Tk`.

Remarks

We like to conclude this Fortran crash course with a few remarks on the use of subroutines and a set of exercises that you may or may not work out. Solutions will be provides in a separate section.

Remark 1 : CALL and SUBROUTINE interface must fit

Let us come back to the interface of subroutine `SUBTC2TK`.

```
SUBROUTINE SUBTC2TK(TC,TK)
```

The interface consists of two variables named `TC` and `TK`. In the declaration part `TC` and `TK` are both declared as `REAL` variables.

We have used the subroutine by a `CALL` statement from a calling program named `CALLTC2TK` with the statement

```
CALL SUBTC2TK(TC,TK)
```

In the declaration part of the calling program `TC` and `TK` were both declared as `REAL` variables.

So, both for the calling program and the called subroutine we used exactly the same data types and even exactly the same names. As we have seen, we could compile the calling program independent of the subroutine without errors. But how does the Fortran compiler know how the interface to the subroutine is defined?

Dangerous

The answer is: The Fortran compiler doesn't know anything about the interface of the subroutine. It compiles the call as implemented. It is up to you to ensure that the call and the subroutine fit together. And this means, that both in the call and in the subroutine the parameters must exactly be of the same number, order, and type. This can be a terrible source for errors. So – be careful, very careful!

Concerning the names, they can be different. So you may call subroutine `SUBTC2TK` with the `CALL` statement

```
CALL SUBTC2TK(MYTC,MYTK)
```

for example, presumed you have declared the variables `MYTC` and `MYTK` as `REAL`, of course.

Remark 2 : Call by reference

When we used the program CALLTC2TK we could observe that we have set TC to a value, "handed the value over" to the subroutine SUBTC2TK, which could use the actual value, perform an operation with it and returned the result in another variable named TK, which the calling program could access and display via a PRINT statement. Does this mean that there are four REAL values in total: TC in the calling program, TC in the called program, TK in the calling program and TK in the called program? The answer is: No, both variables exist only once in the computer's memory.

So we should better not say "handed the value over," but say "handed the variable over" to the subroutine SUBTC2TK, or – even better – "handed a pointer to the variable over" to the subroutine SUBTC2TK. You ask yourself what a pointer is? A pointer is a memory address, i. e., the address where a specific variable or an array of variables resides. As a result, if you want to hand over a complete array of variables to a subroutine, lets say a `REAL X(10)` vector, it is suffucient to (i) either use the name of the vector like `CALL ...(X)`, or – which is absolutely equivalent (ii) use the first element of the vector and write `CALL...(X(1))`. Both calls are equivalent because both the vector X and the element X(1) start at the same address in memory.

This calling method is named "call by reference."

Remark 3 : Dangerous consequence

Let us look at an example which demonstrates one of the dangerous traps of Fortran's consequent call by reference.

```
      PROGRAM DANGER
      IMPLICIT NONE
      REAL NULL /0.0/
      PRINT *,'NULL is now',NULL
      CALL SETFIVE(0.0)
      NULL = 0.0
      PRINT *,'NULL is now',NULL
      STOP
      END

      SUBROUTINE SETFIVE(X)
C     This Subroutine sets the variable X to a value of 5.
C     Nothing else.
      IMPLICIT NONE
      REAL X
      X = 5.0
      RETURN
      END
```

danger.f This program – we named it danger.f – shows two new aspects of Fortran subroutines. (i) Main program unit and subroutine may reside in the same file, so that the both program and subroutine can be compiled and linked via

`gfortran danger.f -odanger.exe`, and (ii) it is possible to hand over "constants" to subroutines.

The program itself is primitive. The main program `DANGER` defines a `REAL` variable `NULL` and initialises it with zero. From the next statement we expect to see an output line similar to `Null is now 0.0`.

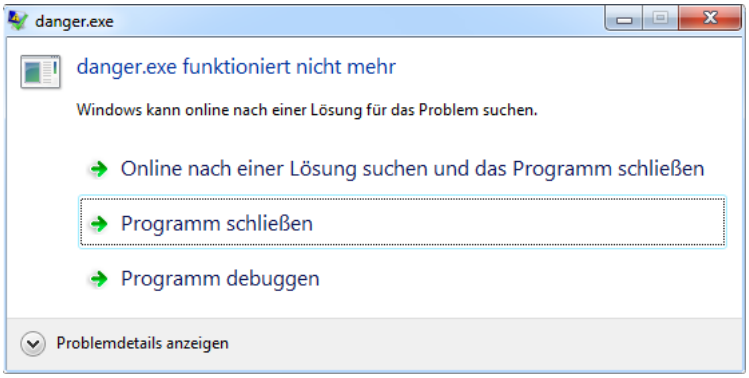
Then we call the subroutine `SETFIVE` with the constant `0.0` as parameter. The subroutine `SETFIVE(X)` sets `X` equal to five (remember: `X` is a pointer to the constant `0.0` – so this operation is really stupid). Returning back to the main program, we set `NULL` equal to `0.0` again and display the value of `NULL` before our program stops.

What do you expect to happen?

When you compile the program with the Salford compiler the ouput is

This seems to be what we expected on the first sight.

When you compile the program with the GNU Fortran compiler the ouput is a well-known window under Windows:



When I compiled this program on an IBM mainframe computer (many years ago) the result was

```
NULL is now 0.0
NULL is now 5.0
```

Do you believe it? What happens?

The blunder is obviously the fact that we hand over a constant `0.0` and change its value in a subroutine. What does call-by-reference mean consequently? Hand over the address of the variable – i. e., the address which points to the constant `0.0`, the subroutine changes the contents at this address to a value `5.0`, the next statement `NULL = 0.0` copies the content of the address where the constant `0.0` is stored (which has a value `5.0` now), and so the second print comes out as indicated.

You can consider this example as an intelligence test for your compiler, but what is much more important:

AVOID under any circumstances to hand over constants to subroutines or functions!!! It is seducing sometimes, but we recommend to avoid it.

Remark 4 : Local variables are not memorised from one call to the next

In programming languages like C/C++ the programmer has an infinite number of options to make programs unreadable for the less-experienced source code reader. For instance, the parameters of functions can be handed over by value or by reference, variables can be static or not, there are pointers, pointers to pointers, references, address operators and many, many, many other things. In Fortran things are much easier – of course, with the drawback of less flexibility but the huge advantage of the option to learn the language like you did, just in a few hours. And for numerical simulations the things you learnt are really sufficient – promised.

What happens, when you hand over a variable to a Fortran subroutine or function? The routine gets the start address of the object, whatever the object is. That's it.

But what happens to a local variable of a subroutine, i. e., a variable which is declared in the subroutine but is not one of its formal parameters, when the subroutine returns? According to the Fortran standard its value is lost, unless the variable has the `SAVE` attribute. With the next call it starts with an undefined value. Many compilers used to keep all local variables in static memory anyway, and older INSEL versions were compiled with the options `-fno-automatic -finit-local-zero`, which made every local variable keep its value and start with zero. This is no longer the case: INSEL compiles Fortran without these options, so local variables live only during one call. In addition, INSEL uses the options `-finit-real=snan -finit-integer=-999999999`, so that a local variable which is used before it gets a value is NaN or -999999999 – a forgotten initialisation shows up in the results instead of being hidden by a zero.

Implicit SAVE

There is one trap: a variable which is initialised in its declaration, like `INTEGER COUNT /0/`, or with a `DATA` statement, is implicitly `SAVE`d. It is initialised only once when the program starts and keeps its value from one call to the next. But there is only one copy of it, which is shared by all instances of the block in an INSEL model. Its value would therefore mix the calls of all these instances.

INTS and REALS

This is why every value that an INSEL block must remember from one call to the next belongs into the `INTS` and `REALS` arrays of the block, which are explained later in this Module. The inselEngine provides them separately for each block instance, initialises them with zero, and keeps their values between calls. Use local variables only for intermediate results within one call, and give them a value before you use them.

Exercise 12.5

As promised, we finish with a couple of exercises (some adapted from an old book by W. E. Spiess and F. G. Rheingans on Fortran programming).

1 Which strict Fortran 77 names are wrong?

```
A-1     % Wrong, means A minus 1
X1328   % Correct
Y*      % Wrong, contains invalid symbol *
2Z00    % Wrong, does not start with a letter
TEST    % Correct
CONTENT % Wrong, more than 6 bytes
REAL    % Wrong, REAL is a statement
```

2 What are the results of the following operations?

```
INTEGER I
REAL    R

R = 7.3
I = R + 4.5   % Result: I = 11

R = 4.0
I = R / 3.0   % Result: I = 1

I = 2
R = I / 3     % Result: R = 0.0

R = 4.0
R = R / 3.0   % Result: R = 1.333333
```

3 What is wrong in the Fortran codes for the given mathematical operations?

$x = (a+b)^2$	`X = A + B ** 2`
$x = \frac{b^{K-L+1}}{b^{K+L-1}+a}$	`X = B ** (K-L+1) / B ** (K+L-1) + A`
$5 = (B^7+2)\cdot C$	`5 = (B**7 + 2.) * C`
$y = \sqrt[4]{a^3}$	`Y = A ** (3/4)`
$z^2 = \frac{a\cdot b}{b+3}$	`Z**2 = A · B / (B + 3.)`

4 Where are the mistakes in this program?

```
      IMPLICIT NONE
      INTEGER I
      REAL A(10),B,C,D
      DO I = 1,10
         A(I) = I
      END DO
      I = 0
1     CONTINUE
      I = I + 1
      B = A(I-1)**2
      C = A(I/2)**3
      D = B + C
```

```
PRINT *,D
IF (I .LE. 11) GO TO 1
STOP
END
```

Mistake 1: On the first call of statement `B = A(I-1)**2` the variable `I` is equal to 1, hence `A(I-1)` evaluates to `A(0)` which is undefined.

Mistake 2: On last call of statement `B = A(I-1)**2` `I` is equal to 12, hence `A(I-1)` evaluates to `A(11)` which is undefined.

Mistake 3: On the first call of the statement `C = A(I/2)**3` the variable `I` is equal to 1, so the integer division `A(1/2)` evaluates to `A(0)`, which is undefined.

5 What is wrong with the following function?

```
REAL FUNCTION SUM
IMPLICIT NONE
INTEGER I,N
SUM = 0.0
DO I = 1,N
   SUM = SUM + A(I)
END DO
RETURN
END
```

Mistake 1: Functions must have formal parameters in parentheses.

Mistake 2: The variable `N` has no value.

Mistake 3: The vector `A` is not declared and thus has no values.

Correct would be

```
C     This routine can sum up to 100 values
      REAL FUNCTION SUM(A,N)
      IMPLICIT NONE
      INTEGER I,N,NMAX
      PARAMETER (NMAX = 100)
      REAL A(NMAX)
      IF (N .GT. NMAX) THEN
         PRINT*,"Sorry, too many elements for SUM Function"
         STOP
      END IF
      SUM = 0.0
      DO I = 1,N
         SUM = SUM + A(I)
      END DO
      RETURN
      END
```

5 What is wrong with the following subroutine statements?

```
SUBROUTINE ALPHA(A,B)
```

```
      IMPLICIT NONE
      INTEGER N
      N = 10
      REAL A(N),B(N,N)
C     ....
```

Mistake 1: Declarations like `A(N),B(N,N)` may not be preceeded by executable statements like `N = 10`.

Mistake 2: Dynamical dimensioning of arrays is not allowed – unless the dimension is one of the formal parameters.

Consequently, the subroutine fragment

```
      SUBROUTINE ALPHA(A,B,N)
      IMPLICIT NONE
      INTEGER  N
      REAL     A(N),B(N,N)
C     ....
```

is formally correct.

6 The equation of time is a term used in solar meteorology which describes the deviation from mean solar time to true solar time. One formula for the calculation is given by Spencer:

$$\begin{aligned} e_t &= (0.000075 + 0.001868\cos(d) - 0.032077\sin(d) \\ &\quad -0.014615\cos(2d) - 0.04089\sin(2d))(180 \cdot 4/\pi) \end{aligned}$$

Here, d denotes the day angle defined by

$$d = \frac{2\pi(d_n - 1)}{365}$$

Write a function which returns e_t as a function of the day number $d_n \in [1, 365]$.

Solution

```
      REAL FUNCTION ET(DN)
      IMPLICIT NONE
      REAL DN,GAMMA
      REAL R2PI /6.2831853/
      GAMMA = R2PI * (DN - 1.0) / 365.0
      ET    = (0.000075
     &         + 0.001868 * COS(GAMMA)
     &         - 0.032077 * SIN(GAMMA)
     &         - 0.014615 * COS(2.0 * GAMMA)
     &         - 0.04089 * SIN(2.0 * GAMMA)) * 229.18
```

12.1.5 Guidelines for writing INSEL Fortran code

- Fortran code lines should not exceed column 72 – even not comments starting with a `!`. (Reason: Printability – and INSEL compiles Fortran with `-Werror=line-truncation`, so a statement beyond column 72 is a compiler error)
- Values which must be kept from one call of a block to the next are stored in `INTS` and `REALS`, never in local variables, `SAVE` or `DATA` statements, or `COMMON` blocks. (Reason: Each block instance needs its own memory, see Remark 4 above)
- All Fortran variables and statements should be written in uppercase letters. (Reason: Compatibility – and tribute to old FORTRAN programming style)
- Use the ampersand & for continuation lines. (Reason: There is no reason, just the INSEL author's habit)
- The only allowed statement which starts with a label should be the `CONTINUE` statement (Reason: Accepted programming style since decades)
- Keywords like `GO TO`, `END DO`, `END IF` should be separated by a blank. (Reason: Just a matter of INSEL taste)
- Use the `DO ... END DO` construct in connection with `EXIT` and `CYCLE` instead of `DO <label> ... <label> CONTINUE` (which we did not even explain, not to say recommend).
- The general number of indention bytes should be three. (Reason: Looks like the INSEL convention)

Example

```
IF (VAR1 .LT. VAR2) THEN
   X1 = 7.1
   DO I = 1,10
      Y(I) = I * X1
   END DO
ELSE
   X1 = -7.1
END IF
```

Please, don't confuse yourself with constructions like

```
IF(VAR1.LT.VAR2)THEN
X1=7.1
DO I=1,10
Y(I)=I*X1
ENDDO
ELSE
X1=-7.1
ENDIF
```

- All Fortran sources should use the `IMPLICIT NONE`. statement. (Reason: Avoid unnecessary error sources)

- In the CALL to a subroutine there should be no space between the name of the subroutine and the opening bracket. (Reason: Better search options)
- Empty lines should be used sparsely. (Reason: When you read the source code, have as much as possible on your screen)
- In the representation of exponential numbers no blank should be added between the base and the exponent, i. e., `1.60201E-19` is preferred to `1.60201 E - 19`. (Reason: Better search options)
- In the definition of variable types no blank should be used, for example `CHARACTER*80` is preferred to `CHARACTER * 80`. (Reason: Matter of taste, but be consequent)
- With any mathematical binary operator like =, +, -, *, /, ** (at least) one blank should be added preceeding and following the operator. If more than one statement belonging together follow in subsequent records, the number of blanks to be used should clearify the structure of the sequence of statements. (Reason: Better readability)

Example

```
RUZ   = 0.0
RJPH  = (RCPH + RC1 *  RT)    * RG
RJD1  = RCD1  * (RT ** 3)     * EXP(PARAMS(4) / RT)
RJD2  = RCD2  * (RT ** 2.5)   * EXP(PARAMS(5) / RT)
R8    = 1.60201E-19 / (RALPHA * 1.38054E-23)
R9    = 1.60201E-19 / (RBETA  * 1.38054E-23)
RUHAT = RUZ + RJ * RS
```

is preferred to

```
RUZ = 0.0
RJPH = (RCPH + RC1 * RT) * RG
RJD1 = RCD1 * (RT ** 3) * EXP(PARAMS(4) / RT)
RJD2 = RCD2 * (RT ** 2.5) * EXP(PARAMS(5) / RT)
R8 = 1.60201E-19 / (RALPHA * 1.38054E-23)
R9 = 1.60201E-19 / (RBETA * 1.38054E-23)
RUHAT = RUZ + RJ * RS
```

or even worse

```
RUZ=0.0
RJPH=(RCPH+RC1*RT)*RG
RJD1=RCD1*(RT**3)*EXP(PARAMS(4)/RT)
RJD2=RCD2*(RT**2.5)*EXP(PARAMS(5)/RT)
R8=1.60201E-19/(RALPHA*1.38054E-23)
R9=1.60201E-19/(RBETA*1.38054E-23)
RUHAT=RUZ+RJ*RS
```

- Comment should be introduced with a capital C in column one.

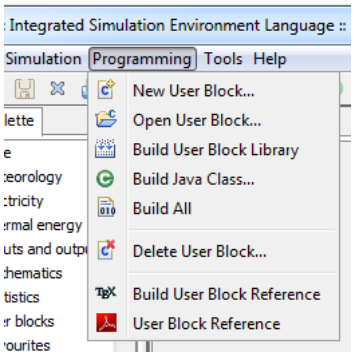
- Other types of comment, for example starting with an exclamation mark anywhere in the Fortran code area, are not recommended.
- Comment is written in uppercase and lowercase letters.
- Comments should start with a capital letter. If the first character of a comment corresponds to a variable of another context the variable name should be used as typed.
- Comment follows the same guidelines as usual code. If a comment follows a conditional statement like `DO` or `IF` it is indented in the same way as the following code.
- All INSEL block sources should start with the documentation header which the Block Wizard generates. For a detailed description of the format of INSEL header files refer to the `src2tex` utility as described later in this Module.
- Names of local variables should be short. (The strict Fortran 77 limit of six characters is not enforced by gfortran, and the names of the INSEL block interface, like `BLOCK_NAMES`, are longer.)
- `REAL` variables should use an `R` as the first character, in general.
- `INTEGER` variables should use an `I` as the first character, in general.
- `CHARACTER` variables should use an `S` as the first character, in general.
- `LOGICAL` variables should use an `L` as the first character, in general.

12.2 Programming INSEL blocks (cont.)

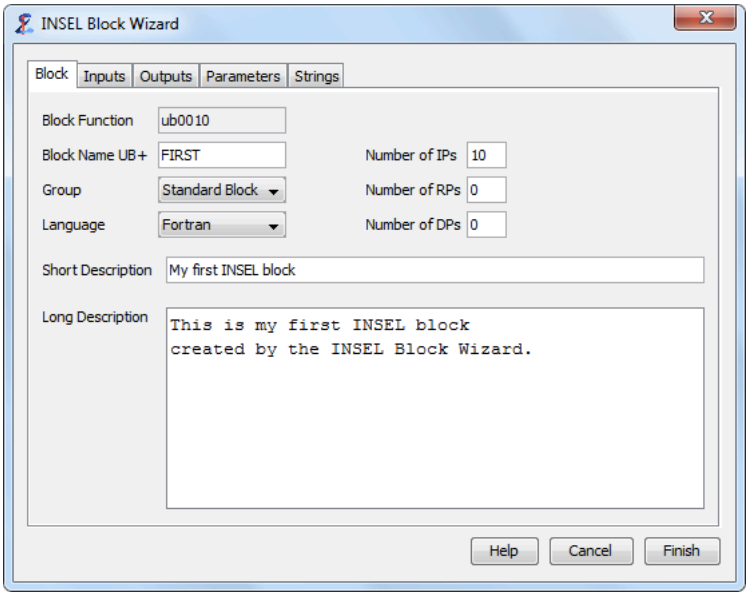
Programming INSEL blocks means to write Fortran subroutines or C++ functions. Both languages use the same interface, the INSEL block API (version 2), which will be explained in this section. Readers of our Fortran crash course will probably prefer to write Fortran subroutines rather than C++ functions.

12.2.1 Block wizard

In INSEL 8 a Block Wizard and all procedures required to integrate new INSEL blocks are now part of the graphical user interface of INSEL.



The Block Wizard can be opened via the *New User Block . . .* dialog, found under *Programming* or as icon in the tool bar.



Five tabs are available which INSEL users are very fond of:

Block tab

As has been discussed frequently, each INSEL block must have a unique name. A new block name can be entered in the *Block Name UB+* text field. Any given block name will be preceded by "UB" to indicate that the new block is going to be a "User Block." Hence, if you enter FIRST for instance, the internal INSEL block name will be UBFIRST. The block name may only contain letters, digits, and the underscore, and may have at most 31 characters including "UB". The Block Wizard refuses the name of a user block which already exists.

Block function

As can be seen from the grayed *Block Function* field the block will be implemented in a function named `ubfirst`, i. e., the block name in lower case. The source file gets the same name: `ubfirst.cpp`, `ubfirst.f`, or `ubfirst.f90`, depending on the language. The block function name is always generated automatically by the Block Wizard and can not be edited.

This naming convention is used when the user block library is built: every source file whose name starts with `ub` (in lower case) must define the block function of the same name, e. g., the Fortran subroutine `UBFIRST` in `ubfirst.f`. Other source files in the same directory may contain helper subroutines and functions. Older INSEL versions used numbered function names like `ub0010`; they are no longer used.

The *Group* can be chosen from a pull-down menu – groups have been introduced in Module , page 18.

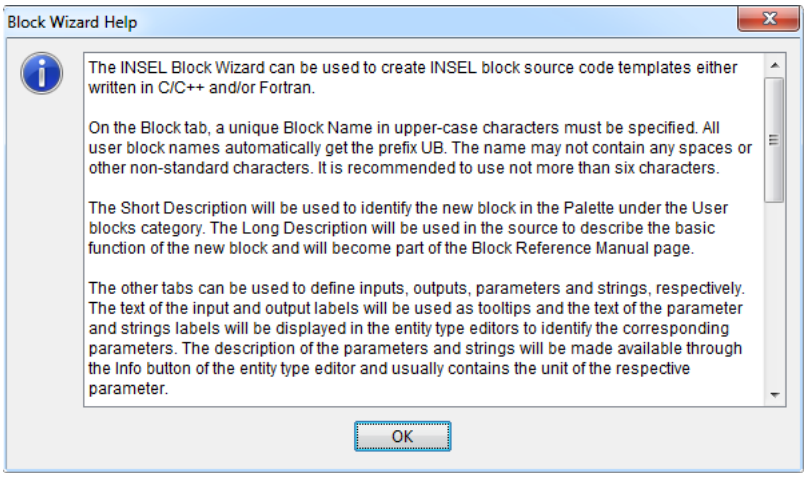
The *Language* pull-down menu allows to choose between generated C++ (the default),

Fortran 77, or Fortran 90 template source code.

The fields *Number of integers (INTS)* and *Number of reals (REALS)* define the size of the integer and real memory of each block instance. This concept is probably new to you and will be discussed soon. For your first block, both can be left at zero.

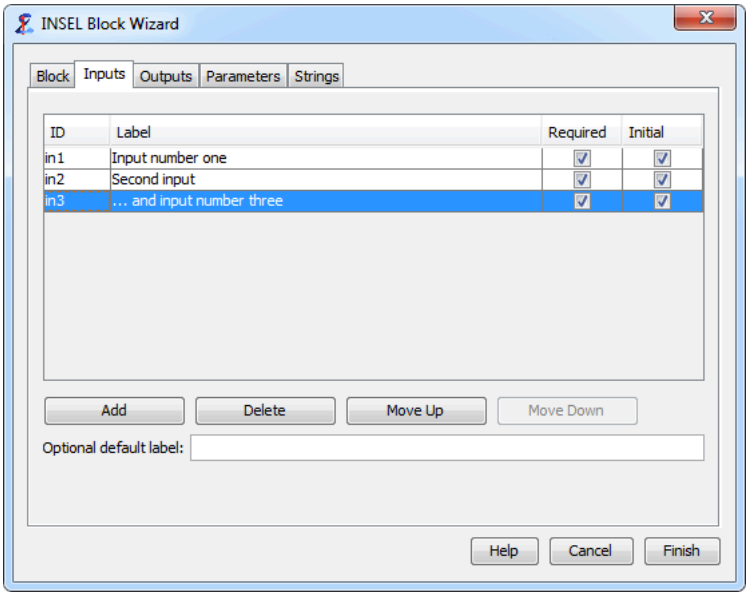
Finally, the *Block* tab displays two text input fields: (i) The *Short Description* will be displayed verbatim in the VSEit Palette, (ii) the *Long Description* will be used as text describing the key idea of what the new block shall do. This text will be typeset into the Block Reference Manual.

The *Help* button can be used to see a summary of the Block Wizard's functionality at any time.



Inputs tab

The *Inputs* tab can be used to define the number of inputs to the new block by using the *Add* button as often as required. The IDs of the input names are generated automatically.



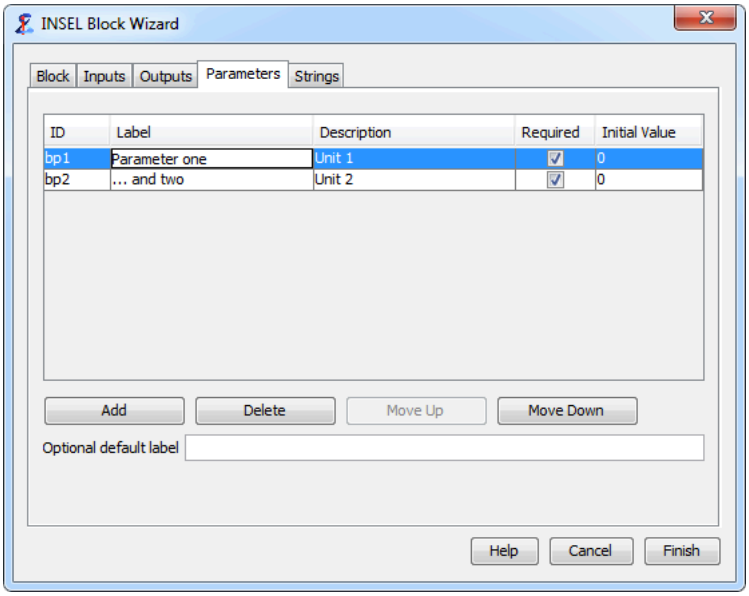
Each input can be given a textual *Label* by double-click into the input field and entering text. The text will be used as tooltip in the VSEit entity as well as in the documentation of the new block in the Block Reference Manual. Of course, all text is open to later editing outside the Block Wizard.

Two types of checkboxes are available for optional inputs and the initial number of inputs displayed on the VSEit entity. It is only possible to uncheck both bottom up. WARNING: Unchecking a number of required inputs can be a dangerous source of errors if not handled properly in the source code.

As long as the Block Wizard is opened, the order of the input variables can be changed by the *Move Up* and *Move Down* buttons where applicable.

Outputs tab — This tab is very similar to the *Inputs* tab and should be self-explaining.

Parameters tab — The *Parameters* tab can be used to define a number of (numerical) block parameters.



The labels will be displayed in the open view of the VSEit entities. The description is used by the VSEit entity in connection with the *Info* button and usually contains the unit of the individual parameters. For each parameter an individual initial value can be set.

Strings tab

This tab is very similar to the *Parameters* tab and can be used to define string parameters for the new block. Only very few INSEL blocks (like file handling blocks) make use of string parameters.

Finish button

When the design of the new INSEL block is ready the Block Wizard can be closed via the *Finish* button. A couple of things will happen in the background then.

12.2.2 Templates

(i) Depending on the choice of language you made, a C++ or Fortran source code template for the block will be generated.

(ii) A Java template for the VSEit entity will be created and compiled into a Java class file. The labels are written to a properties file, see below.

(iii) The new VSEit entity will be added to the Palette category *User blocks*.

(iv) The C++ or Fortran template will be opened in your favorite text editor and you can start to implement your block idea.

Ad (i)

Under Windows each user has an individual directory where the user documents are located. In the newer versions of Windows the name of this directory is `Documents` (independent of the installed language package) and resides in a directory under `Users`

followed by the user's name. This directory is the location for the working directory of INSEL, named `insel.work` as we had seen earlier. Under Linux and macOS the working directory is `~/Documents/insel.work`.

Here you can find the directory `inselUB` which will contain all your files belonging to your user blocks. All source codes will be written to the `src` directory. Hence, the full qualified name of the file created by the Block Wizard is similar to

```
C:\Users\Myself\Documents\insel.work\inselUB\src\ubfirst.f
```

Ad (ii) The Java file which belongs to the INSEL block you just created goes to the same directory. The block name in capitals will be used as Java file and class name, for example

```
C:\Users\Myself\Documents\insel.work\inselUB\src\UBFIRST.java
```

As long as you do not want to make any changes to your new block design outside the Block Wizard you must not know anything about the Java file. But if you wish to modify the block's design later, you will need to modify the Java code manually. So, this is a verbatim copy of the generated file:

UBFIRST.java

```
package eu.insel.userblock;

import de.vseit.network.Attribute;
import de.vseit.network.schema.Icon;
import de.vseit.network.schema.StringType;
import eu.insel.block.Block;
import eu.insel.block.BlockInfo;

@Icon(path="icons/ubfor.png")

@BlockInfo(inMin  = 3,
   inMax  = 3,
   inIni  = 3,
   outMin = 1,
   outMax = 1,
   outIni = 1,
   bpMin  = 2,
   bpMax  = 2,
   spMin  = 0,
   spMax  = 0)

public final class UBFIRST extends Block <UBFIRST>
{
   public @StringType(init="0") Attribute<String> bp1;
   public @StringType(init="0") Attribute<String> bp2;

   public UBFIRST(){}
}
```

Try to bring together the information you provided about your new block and the `BlockInfo` part – get a grasp, at least. Please note that the name of the block function is

not part of the Java class: the inselEngine finds a block by its name.

The Java class file will be stored in a – usually hidden – directory named `AppData` under newer Windows versions. It resides in parallel to the `Documents` directory. The method how this directory can be made visible in Windows Explorer depends on your Windows version. Once you can "see"

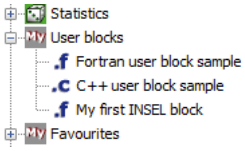
```
C:\Users\Myself\AppData
```

you can make your long way down to the subdirectory

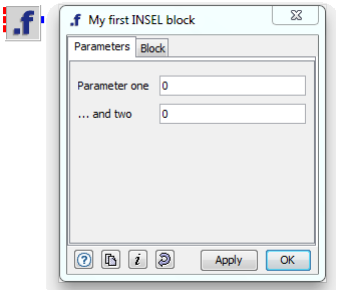
```
Roaming\doppelintegral\INSEL\customTypes\eu\insel\userblock
```

and find the Java class file. Usually you do not want to know all this, but in some emergency cases it might be useful to know, anyway.

Ad (iii) The Wizard should make your new INSEL block visible in the *Palette* immediately and the *User Block* category should look like this:



You can drag your new block (or more precisely your new Type) into the work area and open it.



But if you try to run a model which contains the new block you will get something like

```
Compiling new-1.vseit ...
E04012 Line     1: Unknown blockname: UBFIRST
W04015 Parameters for undefined blocknumber     1 specified
  1 error(s),   1 warning(s)
```

Why's that? So far, no functionality of the new block has been defined. The source code has not even yet been compiled. So how should the inselEngine be able to do something with a block, which is not even known yet?

We have to implement something, compile and link code into a library, which of course has to be found by INSEL before the code can be executed.

Ad (iv) So, let us approach the generated (and opened in the text editor) code slowly and in small portions. We have chosen Fortran 77 as language. If you have closed the editor in the meantime, you can reopen the file via the *Programming* > *Open User Block...* menu, for example.

Header The first records are these:

```
C-----------------------------------------------------------------------
C #Begin
C #Block UBFIRST
C #Description
C    This is my first INSEL block
C    created by the INSEL Block Wizard.
C #Layout
C  #Inputs      3
C  #Outputs     1
C  #Parameters  2
C  #Strings     0
C  #Group       S
C #Details
C  #Inputs
C     #INS(1) Input number one
C     #INS(2) Second input
C     #INS(3) ... and input number three
C  #Outputs
C     #OUTS(1) Only one output specified
C  #Parameters
C     #PARAMS(1) Parameter one
C     #PARAMS(2) ... and two
C  #Strings
C     #None
```

In the header of the code we basically find the information entered in the Block Wizard: the block name UBFIRST, the Description, the number of inputs, outputs etc in a Layout table, and the Labels entered in the text input fields. All this information is only comment, as we can see from the capital C in column one (Fortran convention).

In addition we see some keywords like `#Block`, `#Description`, `#Layout` etc. These keywords will be interpreted by a small program named `src2tex` which will transform the information contained in the comment lines into TEX format (we will come back to this point soon).

Internals What follows is the `#Internals` part of the header.

```
C #Internals
C  #Integers
C     #None
C  #Reals
C     #None
```

```
C #Dependencies
C   #Subroutine INSEL_IDENTIFY
C #Authors
C   INSEL Block Wizard
C #End
C-----------------------------------------------------------------------
```

Here, the elements of the INTS and REALS arrays of the block are documented, as soon as your block uses them, e. g., `#INTS(1) Counter for the number of calls`.

Declaration section

The third important part is the declaration section.

```
      SUBROUTINE UBFIRST(
     &          CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      IMPLICIT NONE
      INCLUDE 'insel_block.inc'
      CHARACTER*1024 BLOCK_NAMES
      INTEGER MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &          MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP
      PARAMETER (BLOCK_NAMES = 'UBFIRST',
     &          MIN_INS     = 3,
     &          MAX_INS     = 3,
     &          N_OUTS      = 1,
     &          N_INTS      = 0,
     &          N_REALS     = 0,
     &          MIN_PARAMS  = 2,
     &          MAX_PARAMS  = 2,
     &          MIN_STRINGS = 0,
     &          MAX_STRINGS = 0,
     &          GROUP       = 3)
      CHARACTER*1024 STRINGS(MAX_STRINGS+1)
      INTEGER CTRL(NCTRL),INTS(N_INTS+1)
      REAL INS(MAX_INS+1),OUTS(N_OUTS+1),REALS(N_REALS+1),
     &          PARAMS(MAX_PARAMS+1)
```

As you can see, INSEL blocks written in Fortran are subroutines – with exactly the same formal parameter set for each block, independent of all other block properties. The formal parameter set consists of seven pointers to the arrays CTRL (control array shared with the inselEngine), INS (inputs), OUTS (outputs), INTS and REALS (integer and real memory of the block instance), PARAMS (numerical parameters), and STRINGS (string parameters, 1024 characters each). The file `insel_block.inc` defines the size NCTRL of the CTRL array, names for its elements like CALLMODE, and the names of the call modes. All real values are 8-byte reals: INSEL compiles Fortran with `-fdefault-real-8`, so REAL here is the same as DOUBLE PRECISION and as double in C++.

When you look at the defined parameters and compare them with the input to the Block Wizard, the values should be familiar – `GROUP = 3` is the internal representation of Standard Blocks in INSEL. All parameters will be explained in detail on page 260.

Please observe, that the dimensions of the arrays are all oversized by one. The only reason for this is to avoid compiler warnings or errors when an array (like REALS in this

case, for example) has dimension zero. Hence, do NEVER access these additional values because their memory is undefined and accessing undefined memory can lead to unpredictable errors during execution time.

Code section

Finally, we arrive at the code section where you can let your phantasy completely free to implement new Nobel-price ideas or whatever you think is missing in INSEL but useful for your simulations.

```
C-----------------------------------------------------------------------
      IF (CTRL(CALLMODE) .NE. STANDARD) THEN
         IF (CTRL(CALLMODE) .EQ. IDENTIFY) THEN
C           Identification call
            CALL INSEL_IDENTIFY(CTRL,STRINGS,BLOCK_NAMES,
     &          MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &          MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP)
            RETURN
         ELSE IF (CTRL(CALLMODE) .EQ. CONSTRUCT) THEN
C           Constructor call
         ELSE
C           Destructor call
         END IF
         RETURN
      END IF
C---- Standard call ------------------------------------------------
      RETURN
      END
C-----------------------------------------------------------------------
```

C++

Had you chosen C++, the Block Wizard would have generated the file `ubfirst.cpp` with the same documentation header in a `/* ... */` comment (with C indices, e. g., `#INS[0]`) and the following code (line breaks adapted to this page):

```
#include "insel_block.h"

extern "C" void ubfirst(int *ctrl, double *ins, double *outs,
      int *ints, double *reals, double *params, char strings[][1024]) {
   char BLOCK_NAMES[1024] = "UBFIRST";
   int MIN_INS     = 3;
   int MAX_INS     = 3;
   int N_OUTS      = 1;
   int N_INTS      = 0;
   int N_REALS     = 0;
   int MIN_PARAMS  = 2;
   int MAX_PARAMS  = 2;
   int MIN_STRINGS = 0;
   int MAX_STRINGS = 0;
   int GROUP       = 3;

   if (ctrl[insel::CALLMODE] != insel::STANDARD) {
      if (ctrl[insel::CALLMODE] == insel::IDENTIFY) {
         insel_identify(ctrl, strings, BLOCK_NAMES, &MIN_INS, &MAX_INS,
               &N_OUTS, &N_INTS, &N_REALS, &MIN_PARAMS, &MAX_PARAMS,
               &MIN_STRINGS, &MAX_STRINGS, &GROUP);
```

```
      } else if (ctrl[insel::CALLMODE] == insel::CONSTRUCT) {
         // Constructor call
      } else {
         // Destructor call
      }
      return;
   }
   // Standard call ------------------------------------------
}
```

The header file `insel_block.h` defines the names of the `ctrl` indices and of the call modes in the namespace `insel`, and declares `insel_identify` and `msg`. The function must be declared `extern "C"`, and all arrays start with index zero, so the first input is `ins[0]`. The strings of `strings` are padded with blanks and not terminated by a null character.

12.2.3 Call modes

As mentioned before, you can see now, how the different Call modes in an INSEL block are organized. Again, the Call modes are

`CTRL(CALLMODE) = -1` Identification call (`IDENTIFY`)
This Call mode is executed by the inselEngine (or any other call method when `CTRL(CALLMODE)` is equal to `IDENTIFY`, i. e., `-1`). The meaning of this call is to find out the values specified in the parameter statement. This is always the very first mode organised by the inselEngine. In this way the inselEngine receives all information about the general layout of the block(s) which is/are defined in the subroutine and can handle the memory requirements for a specific block instance.

There is no secret in the `INSEL_IDENTIFY` subroutine. It copies `BLOCK_NAMES` to `STRINGS(1)` and the other values from the parameter statement to elements of the `CTRL` array, which are reserved for this purpose. The block must return right after the call.

If you wonder how the linker will later find the routine: it is compiled into the inselTools library which is linked to all libraries containing INSEL blocks. `INSEL_IDENTIFY` is written in C++, so that Fortran and C++ blocks can call it alike.

It is not necessary that you care for all these details, but we thought maybe you'd like to know.

`CTRL(CALLMODE) = 1` Constructor call (`CONSTRUCT`)
Before an INSEL model is executed INSEL provides the option for INSEL block programmers to write some statements which are executed before the INSEL model itself starts execution. Here you can check for the reasonability of the parameters as given by the user or perform some preparatory step for your block.

The idea of the Constructor call is very close to the constructor concept of C++ classes.

`CTRL(CALLMODE) = 2` Destructor call (DESTRUCT)
Before an INSEL model terminates INSEL provides the option for INSEL block programmers to write some last statements.

The idea of the Destructor call is very close to the destructor concept of C++ classes.

`CTRL(CALLMODE) = 0` Standard call (STANDARD)
This mode is used in every simulation time step as defined by an INSEL T-block. In most cases, these statements will contain the most vital part of your block (and all other INSEL blocks).

Example Having said all this let us add some code to your first INSEL block. What shall we do? Remember, we have defined a block which requires three inputs, two numerical parameters and which provides one output. How about the formula

$$o = \sin(i_1) + i_2 * i_3/p_2$$

where we use p_1 (parameter number one) to decide whether i_1 (input number one) is given in degrees or in radians. Okay, we could easily implement this in VSEit and make a macro out of it. But the task is just complex enough to show some INSEL block programming techniques.

First of all, let us summarize the ideas for the block in the header of the source code:

```
C     #INS(1) Any angle $i_1$ either in degrees or in radians
C     #INS(2) Just another input named $i_2$
C     #INS(3) And input number three $i_3$
C  #Outputs
C     #OUTS(1) The result of $\sin(i_1) + i_2 * i_3 / p_2$
C  #Parameters
C     #PARAMS(1) Switch to decide whether $i_1$ is in degrees
C            $(p_1 = 0)$ or radians $(p_1 \ne 0)$
C     #PARAMS(2) The second parameter $p_2$
```

Here we have used some TEX conventions, like everything between two Dollar signs is Math mode, everything else standard text. Maybe it is worth that you consider learning some basic TEX.

Now, let's write a first code section:

```
C---- Standard call ------------------------------------------------------
      IF (ANINT(PARAMS(1)) .EQ. 0) THEN
C        Angle is in degrees
         OUTS(1) = SIN(INS(1) * ASIN(1.0) / 90.0)
     &           + INS(2) * INS(3) / PARAMS(2)
      ELSE
C        Angle is in radians
         OUTS(1) = SIN(INS(1)) + INS(2) * INS(3) / PARAMS(2)
      END IF
      RETURN
      END
C---------------------------------------------------------------------------
```

Please note the continuation line: a Fortran statement must end in column 72, and INSEL compiles Fortran with `-Werror=line-truncation`, so a longer line is an error.

Before we go into details, let us see whether the compiler accepts our code and use the *Programming > Build User Block Library* menu item. It runs `make` (or `mingw32-make` under Windows) with the makefile of the INSEL installation in the directory `insel.work\inselUB\resources`. The makefile compiles all `.f`, `.f90`, and `.cpp` files of the `src` directory, generates the table of the block functions from the names of the `ub*` files, and links everything into the library `libInselUB.dll` (`libInselUB.so` under Linux, `libInselUB.dylib` under macOS) in the `resources` directory. The same can be done in a terminal, e. g., under Linux with

```
cd ~/Documents/insel.work/inselUB/resources
make -f /usr/local/insel/makefile
```

No errors

If you are lucky and made no mistakes you should get an output similar to this in the INSEL output window:

```
Starting Build User Block Library thread ...
Compile libInselUB for Linux
Fortran user blocks compiled.
libInselUB completed.
Library successfully created
Build User Block Library thread finished
```

Error

Otherwise the compiler will report some error messages like:

```
Compile libInselUB for Linux
../src/ubfirst.f:41:39:

   41 |          OUTS(1) = SIN(INS(1) * ASIN(1.0)) / 90.0)
      |                                       1
Error: Invalid character in name at (1)
```

Do you see where the mistake is? Getting code compiled without errors can give you a hard time sometimes. In addition, error messages are not always clear from the beginning but must be interpreted with a lot of phantasy. Never give up!

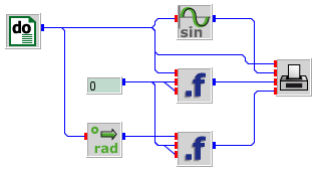
Loading

The inselEngine loads all block libraries of the INSEL installation and of the directory `insel.work\inselUB\resources`; only files whose names contain “insel” are considered. Each library exports a function `insel_library()` which returns the version of the block API and the table of its block functions. A library which was built for an older version of the block API is ignored with the warning

```
W04095 ... was built for another version of the INSEL block API
       and is ignored. Please recompile it.
```

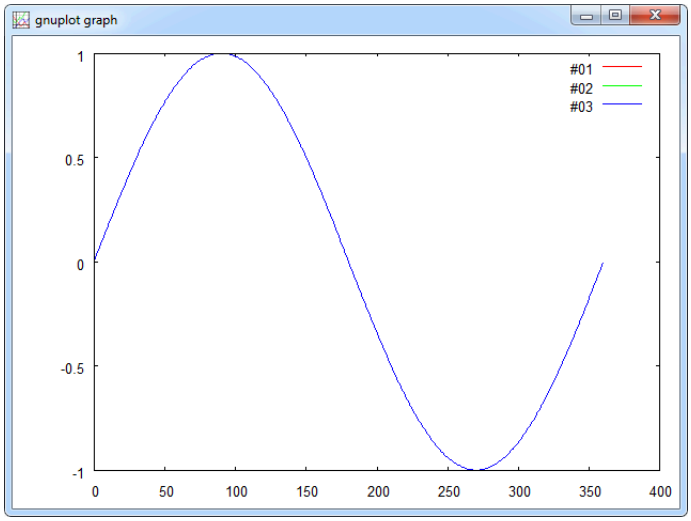
In this case, user blocks written for an older INSEL version must be adapted to the interface described here and the library must be built again.

Okay, once you get your code compiled (and linked) let us test it before we come back to some details.



As a first test we have used two UBFIRST blocks, the upper one with first parameter set to zero (degrees case) and the lower one with value one (radians). The second parameter is set to one in both cases. Since the second and third inputs are all equal to zero, the UBFIRST block just reduces to the function $\sin(x)$.

Since the DO block varies the input angle between zero and 360 degrees the PLOT block should display three identical sine curves. And indeed,



from the legend we see that three curves are plotted, all of them exactly equal.

12.2.4 Properties

When you open the UBFIRST entity you will see that the parameter labels are still the ones defined in the Block Wizard and not – as you perhaps might have expected – the modified parameter names in the Fortran source's header. One reason is that TEX code can be used in the header but not in the VSEit entities.

When you check out your `inselUB\src` directory you will find a file named `i18nUserBlock.properties`. Open it with your text editor and you will see the content.

```
#
#Tue Mar 15 15:37:05 CET 2011
UBFIRST=My first INSEL block
UBFIRST.bp1=Parameter one
UBFIRST.bp1-DESCR=Unit 1
UBFIRST.bp2=... and two
UBFIRST.bp2-DESCR=Unit 2
UBFIRST.in1=Input number one
UBFIRST.in2=Second input
UBFIRST.in3=... and input number three
UBFIRST.out1=Only one output specified
```

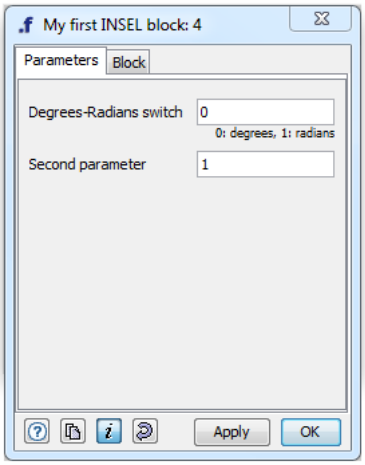
As you can see, all labels from the Block Wizard are appended. Feel free to edit the labels to your needs, for example

```
UBFIRST.bp1=Degrees-Radians switch
UBFIRST.bp1-DESCR=0: degrees, 1: radians
UBFIRST.bp2=Second parameter
UBFIRST.bp2-DESCR=
```

and save it. Three things are left to say to the `.properties` file:

- The modified file has to be copied to the directory where the class files reside. One way to accomplish this is by running *Build All* from the Programming menu or from the tool bar.
- The bad news is that INSEL has to be restarted before the changes are applied.
- The good news is that the `i18n` in the file name stands for internationalisation (with 18 letters nternationalisatio – those computing guys). That means you can have a file named `i18nUserBlock_de.properties` with German labels for the German versions of your blocks.

Following the recommendation to build all and restart INSEL we finally get



12.2.5 Documentation

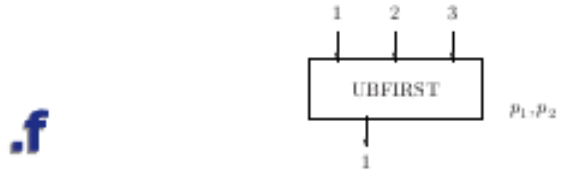
The *Build All* function also runs *Build User Block Reference*, which compiles the User Block Reference Manual with `pdflatex` in the directory

```
C:\Users\Myself\Documents\insel.work\inselUB\doc\blockReference\english
```

The manual is opened with *User Block Reference* from the *Programming* menu. A block page looks like this:

1.3 Block UBFIRST

This is my first INSEL block created by the INSEL Block Wizard.



Name	UBFIRST
Function	ub0010
Inputs	3
Outputs	1
Parameters	2
Strings	0
Group	S

Inputs

1 Any angle i_1 either in degrees or in radians
2 Just another input named i_2
3 And input number three i_3

Outputs

1 The result of $\sin(i_1) + i_2 * i_3/p_2$

Parameters

1 Switch to decide whether i_1 is in degrees ($p_1 = 0$) or radians ($p_1 \neq 0$)
2 The second parameter p_2

Strings

None

You can open the manual page also from the *Help* button in the Entity editor of your user block.

The LaTeX code of a block's manual page is generated from the documentation header of its source file by `src2tex`, a small Python program of INSEL which is called as `src2tex <source-file> [output-dir]`. It writes one `.tex` file per block name. The same program generates the pages of the INSEL Block Reference Manual.

.des files

If a file named after your block with extension `.des`, for example `ubfirst.des`, exists in the directory `../inselBlockReference/des` relative to the directory in which `src2tex` runs, `src2tex` adds an `\input` of this file at the end of the block's page. You can use

everything TEX and LATEX provides in it – and that is a lot!

Source to TEX syntax

Fortran and C++ code written for INSEL can be documented within the framework of the source code itself. INSEL provides a converter called `src2tex` which generates documentation files written in LATEX from special statements in Fortran or C++ source code comment records. The file extension must be `.f` or `.cpp`.

#-commands

The src2tex commands are introduced by a # symbol. src2tex ignores the first two characters of every line, so the #-commands may not start before column three. In Fortran, these two characters are `C` plus one space character, in C++ the header is a `/* ... */` comment and its lines start with two blanks.

The #-commands are case sensitive and must be written as shown below, i. e., the first letter is an uppercase letter, while all the other letters are lowercase.

The following #-commands are known to the src2tex converter (Please note that the sequence of the statements is crucial, i. e., the #-commands may only be used in the given order due to the sequential structure of the src2tex converter):

Begin

`#Begin` marks the beginning of a section which is interpreted by the src2tex converter.

Block

`#Block <Namelist>` marks the beginning of an INSEL block section. The `<Namelist>` is a list of block names, which are defined in the source code under consideration. Usually, `<Namelist>` consists of only one unique block name. If `<Namelist>` has more than one entry, the names have to be separated by blanks, optionally preceded by commas. `#Block <Namelist>` is used as a name for the LATEX section of the INSEL Block Reference Manual.

Description

`#Description <Name>` is followed by a short description of the function of block `<Name>` written in LATEX in the following lines. When the description is not unique for all blocks in the `<Namelist>` of the `#Block` command a particular block description may be specified by `<Name>`. A `#Description` without `<Name>` applies to all other blocks.

Layout

`#Layout` is used by src2tex to define a list of the most important properties of an INSEL block, such as

`#Inputs <NumberOfInputs>` where `<NumberOfInputs>` is either an INTEGER constant or a range of allowed `<InputValues>` of the form `<InputMin> ... [<InputMax>]` with `<InputMin>` and `<InputMax>` being INTEGER constants such that `<InputMin>` is less than `<InputMax>`,

`#Outputs <NumberOfOutputs>` where `<NumberOfOutputs>` is a constant INTEGER value,

`#Parameters <NumberOfParameters>` where `<NumberOfParameters>` is either an INTEGER constant or a range of allowed `<ParameterValues>` of the form

<ParametersMin> ... [<ParametersMax>] with <ParametersMin> and <ParametersMax> being INTEGER constants such that <ParametersMin> is less than <ParametersMax>,

#Strings <NumberOfStrings> where <NumberOfStrings> is either an INTEGER constant or a range of allowed <StringValues> of the form <StringsMin> ... [<StringsMax>] with <StringsMin> and <StringsMax> being INTEGER constants such that <StringsMin> is less than <StringsMax>,

#Group <GroupInformation> where <GroupInformation> may either be a C (for Constant blocks), T (for Timer blocks), S (for Standard blocks), L (for Loop blocks), D (for Delay blocks) or I (for the If block group of INSEL).

The text after these commands is copied to the manual as it is. Like #Description, #Layout <Name> can be given for a particular block.

Details

#Details is – like #Layout – a sectioning src2tex command, i. e., there are some subcommands to #Details, namely

#Inputs <Block> starts a list of all available inputs as used by the INSEL block <Block>. When the inputs are unique for all the blocks in a section which have not been specified by an #Inputs <Block> command in one of the preceding records then <Block> may be omitted and the #Inputs command is applied to all other blocks within this section.

```
#INS(1) <TeX-Text-1>
#INS(2) <TeX-Text-2>
#INS(n) <TeX-Text-n>
```

where <TeX-Text-i> may be any description of the ith input written in LaTeX. When the number n is not constant but variable n should be written as n to produce a MathFont representation of n in the LaTeX code. In case of n equal to zero, i. e., the #Inputs description of a block with no inputs, a #None statement should be provided. Lines without # continue the text of the previous item.

#Outputs <Block> starts a list of all available outputs as used by the INSEL block <Block>.

```
#OUTS(1) <TeX-Text-1>
#OUTS(2) <TeX-Text-2>
#OUTS(n) <TeX-Text-n>
```

See #Inputs for further details.

#Parameters <Block> starts a list of all available numerical parameters as used by the INSEL block <Block>.

```
#PARAMS(1) <TeX-Text-1>
#PARAMS(2) <TeX-Text-2>
#PARAMS(n) <TeX-Text-n>
```

See #Inputs for further details.

#Strings <Block> starts a list of all available string parameters as used by the INSEL

block <Block>.
#STRINGS(1) <TeX-Text-1>
#STRINGS(2) <TeX-Text-2>
#STRINGS(n) <TeX-Text-n>
See #Inputs for further details.

Remarks

#Remarks <Block> allows for the inclusion of some remarkable text corresponding to the <Block> block. The use of #Remarks is optional.

This concludes the list of #-commands which are used by src2tex to generate *.tex files from the source code file *.f or *.cpp. The following #-commands are ignored by src2tex. They document the internals of the block for the readers of the source code.

Internals

#Internals is another sectioning command. Its subcommands are

#Integers which introduces a list of the elements of the INTS array
#INTS(1) <TeX-Text-1>
#INTS(2) <TeX-Text-2>
#INTS(n) <TeX-Text-n>
See #Inputs for further details.

#Reals which introduces a list of the elements of the REALS array
#REALS(1) <TeX-Text-1>
#REALS(2) <TeX-Text-2>
#REALS(n) <TeX-Text-n>
See #Inputs for further details.

Dependencies

#Dependencies is a command which allows for the description of subroutines or functions that are used by the source code. It follows a list of those subroutines and functions which are used, for example #Subroutine INSEL_IDENTIFY.

Authors

The #Authors statement can be used to document the name of the block's code author(s).

End

The #End statement marks the end of the source code documentation.

C conventions

Blocks written in C++ use C syntax rules in their documentation header, i. e., Fortran's #PARAMS(1) corresponds to C's #PARAMS[0] etc. src2tex adds one to numerical indices, so that the manual always counts from one.

12.3 Text output from INSEL

This section is about handling of text output from INSEL. In our Fortran course we have used simple PRINT statements. The precondition for the PRINT statement is that the program runs in a DOS box or a text terminal. Windows applications cannot use PRINT statements, since there is no defined "receiver" for text messages, in the first place. How

does INSEL handle this problem?

12.3.1 Message files

Language? A second problem which has to do with text output is, that text is always written in a certain language – like English, German, Spanish, or any other language that a software supports. If we would include the text in the source code like

```
PRINT *,'This is an error message in English'
```

adaption to a new language would mean, that we have to go through all source files of the program, translate all messages into the new language, recompile and link all sources. As long as the project has only a few subroutines, this procedure seems acceptable. But in an application like INSEL, which has more than thousand source code files, this method drops out.

`insel.msg` The second problem is solved in INSEL with a file called `insel.msg` in the `resources` directory of the INSEL installation. This file contains all textual messages that can occur. It is an ASCII file, so you can open it with your text editor and peep in. Here are the first ten records:

```
00000 Error #1I6.6# (no detailed error message found)
00001 ... General messages
00002 File not found
00003 Path not found
00004 Too many open files
00005 File access denied
00006 Invalid file handle
00012 Invalid file access mode
00015 Invalid drive number
00016 Cannot remove current directory
```

To adapt INSEL to a new language now simply means a new translation of the file `insel.msg`.

user.msg INSEL uses message numbers in the range of 0 to 89999. Message numbers of 90000 or higher are reserved for user-defined messages. The text for user-defined messages is expected in a file named `user.msg` located in the `inselUB\resources` directory of the INSEL working directory `insel.work`.

`os0ltxt` The first mentioned problem – i. e., the question where and how text is displayed – is solved in the C++ function `os0ltxt` of the inselTools library. When INSEL runs on the command line, `os0ltxt` writes the text to the terminal. When INSEL runs in the graphical user interface, the inselBridge redirects all text to the INSEL output window. The calling routines are therefore independent of the place where the text appears.

Usually, you will not make direct calls to `os0ltxt` but use the INSEL message system, which is presented next.

12.3.2 The INSEL message system

All INSEL messages are distributed via the MSG subroutine (written in C++, part of the inselTools library) which formats the message and passes it to os0ltxt. Blocks, functions, and subroutines can call MSG via

```
CALL MSG(I,R,S)
```

where I is an integer array of size ten I(10), R is a real array of size ten R(10) and S is an array of ten CHARACTER*1024 strings S(10).

I(1) must include a unique message-type and message-text identifier. The format of this identifier is

```
xyyyyy
```

Message types

where x indicates a one-digit message-type following the convention

0 General message with message number supressed

1 General message

2 Warning message

3 Error message

4 Fatal error message

and yyyyy indicates a five-digit message number as defined in a .msg file. As mentioned before, insel.msg in the resources directory of the INSEL installation is used for yyyyy less than 90000. If yyyyy is greater or equal 90000 the file user.msg in the user's working directory inselUB\resources is used.

Hence, independent on the installation and on further updates of insel.msg, you can use your own numbers and write your own user.msg file with your personal error messages. Each record consists of the five-digit message number followed by the text.

Block messages

INSEL blocks usually call MSG with their own CTRL, REALS, and STRINGS arrays:

```
CTRL(RETCODE) = 205002
CALL MSG(CTRL,REALS,STRINGS)
```

In this case CTRL(RETCODE), i. e., CTRL(1), contains the message identifier, CTRL(BLOCKNR), i. e., CTRL(4), the user-defined block number, and CTRL(8) to CTRL(10) are free for integer arguments of the message. Since CTRL(RETCODE) is also the return code of the block, a warning (2yyyyy) or an error (3yyyyy) is counted by the inselEngine. MSG only accesses the array elements which are referenced in the message text, so make sure that they exist – a block with N_REALS = 0 must not use a message with a real argument together with its REALS array.

In C++, msg is declared in insel_block.h:

```
void msg(int *ctrl, double *reals, char (*strings)[1024]);
```

and a block calls it like this:

```
ctrl[insel::RETCODE] = 205002;
msg(ctrl, reals, strings);
```

Please note that due to C conventions `ctrl[0]` contains the message-type and message-text identifier in this case.

The standard output of `MSG` is a message of type

```
 syyyyy text
```

where `s` is a one letter message-type indicator following the convention:

`M` An `M` stands for a general message.

`W` `W` is a warning message.

`E` `E` is an error message.

`F` `F` is a fatal error message.

For message type 0, neither the indicator nor the message number are displayed, only the text.

`yyyyy` `yyyyy` is the above-mentioned message number.

`text` `text` is a concatenated text message which may include variable numerical and textual information.

For text, an INSEL specific syntax has been developed. Non-constant entries are format statements of the form `#n*#` where `n` is the index of the corresponding array element ranging from 1 to 10 and `*` may be one of the following formats:

For use with the `I` array the format is `Im.m` where `I` is short for integer and `m` is the number of characters to be displayed in text (including leading zeros). `I` and `Im` display the number without leading zeros or blanks.

For use with the `R` array the format is `F`, followed by an optional width and precision like `F8.2`. In the current version, width and precision are ignored and the number is displayed in a default format.

For use with the `S` array the format is `A` where `A` is short for alphanumeric. Trailing blanks are removed from the text.

Example In the file `insel.msg` you will find the record

```
05002 Block #4I5.5#: Number of divisions by zero: #9I8#
```

This message is used by the DIV block, which divides its first input by the second input `OUTS(1) = INS(1) / INS(2)`. Whenever DIV is called with `INS(2) = 0` the block does not perform the division but counts `INTS(1) = INTS(1) + 1` instead of causing a runtime error. At the end of a simulation run `INTS(1)` is equivalent to the number of calls with `INS(2) = 0`.

In the destructor call `INTS(1)` is copied to `CTRL(9)` and `CTRL(RETCODE)` is set to `205002`. It follows, that the format string `#9I8#` is replaced by the corresponding number of divisions by zero. `CTRL(4)` contains the user-defined block number, hence, `#4I5.5#` provides this information. If we assume the number of divisions by zero was 17 and the user defined block number 4711, the code

```
      ...
C           Destructor call
            IF (INTS(1) .GT. 0) THEN
C              Display number of divisions by zero
               CTRL(9) = INTS(1)
               CTRL(RETCODE) = 205002
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
      ...
```

will result in the warning message

```
W05002 Block 04711: Number of divisions by zero: 17
```

Exercise 12.6 Write a block which uses `MSG` to display the call modes. Hint: Message number 4030 displays the first string handed over via `MSG`.

Solution Do you have your code ready? Here is our solution, saved as `ubcallmode.f` in the `src` directory of `inselUB`. It uses its own local arrays `I`, `R`, and `S` for `MSG`, since the block has no string parameters. The documentation header is left out.

```
      SUBROUTINE UBCALLMODE(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      IMPLICIT         NONE
      INCLUDE          'insel_block.inc'
      CHARACTER*1024   BLOCK_NAMES
      INTEGER          MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &                 MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP
      PARAMETER       (BLOCK_NAMES = 'UBCALLMODE'
     &,                MIN_INS     = 1
     &,                MAX_INS     = 1
     &,                N_OUTS      = 0
     &,                N_INTS      = 0
     &,                N_REALS     = 0
     &,                MIN_PARAMS  = 0
     &,                MAX_PARAMS  = 0
     &,                MIN_STRINGS = 0
     &,                MAX_STRINGS = 0
     &,                GROUP       = 3)
      CHARACTER*1024   STRINGS(MAX_STRINGS+1)
      INTEGER          CTRL(NCTRL),INTS(N_INTS+1)
```

```
      REAL             INS(MAX_INS+1),OUTS(N_OUTS+1),REALS(N_REALS+1),
     &                 PARAMS(MAX_PARAMS+1)
      INTEGER          I(10)
      REAL             R(10)
      CHARACTER*1024   S(10)
C-----------------------------------------------------------------------
      I(1) = 4030
      IF (CTRL(CALLMODE) .NE. STANDARD) THEN
         IF (CTRL(CALLMODE) .EQ. IDENTIFY) THEN
C           Identification call
            S(1) = 'Identification call'
C           CALL MSG(I,R,S)
            CALL INSEL_IDENTIFY(CTRL,STRINGS,BLOCK_NAMES,
     &         MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &         MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP)
            RETURN
         ELSE IF (CTRL(CALLMODE) .EQ. CONSTRUCT) THEN
C           Constructor call
            S(1) = 'Constructor call'
            CALL MSG(I,R,S)
         ELSE
C           Destructor call
            S(1) = 'Destructor call'
            CALL MSG(I,R,S)
         END IF
         RETURN
      END IF
C---- Standard call ----------------------------------------------------
      S(1) = 'Standard call'
      CALL MSG(I,R,S)
      RETURN
      END
```

If you have studied section about the INSEL message system the code should be self explaining. In order to test the block, build the user block library and write a small program like

```
s 1 DO
p 1 1 3 1
s 2 UBCALLMODE 1.1
```

and run it. When everything works fine the result should be similar to

```
Compiling callmode.insel ...
Constructor call
0 errors, 0 warnings
Running INSEL 8.3 ...
Standard call
Standard call
Standard call
Destructor call
Normal end of run
```

Please observe that the Constructor call is made before the start of the simulation run.

Hint The MSG call in the Identification call is commented out on purpose. The Identification call is made when the inselEngine loads the block libraries, even before it compiles the model file – so the text would be displayed in all your simulation runs, whether they use UBCALLMODE or not.

12.4 INSEL block source code examples

We are now ready to look at some INSEL blocks. Before we start, let us have a summary view on the variables which are in common to all INSEL blocks.

Declaration section The first statement declares the type of the program unit and uses the already discussed subroutine interface `(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)`. The next statement declares `IMPLICIT NONE`, which means that all variables which are used in the subroutine have to be declared in the declaration section of the code. As pointed out in our Fortran crash course, we strongly recommend to use an `IMPLICIT NONE` statement in order to avoid the implicit variable type settings of Fortran. The statement `INCLUDE 'insel_block.inc'` defines the size `NCTRL` of the `CTRL` array, the names of the `CTRL` indices, and the names of the call modes.

A sequence of declarations of some vital INSEL variables follows. Take your time to understand them properly.

`BLOCK_NAMES` must be a `CHARACTER*1024` variable, followed by the declaration of ten `INTEGER` variables which are essential for INSEL. Their names are `MIN_INS`, `MAX_INS`, `N_OUTS`, `N_INTS`, `N_REALS`, `MIN_PARAMS`, `MAX_PARAMS`, `MIN_STRINGS`, `MAX_STRINGS`, and `GROUP`. They are handed over to `INSEL_IDENTIFY` during the Identification call.

Let us look at their meaning one by one.

`BLOCK_NAMES` `BLOCK_NAMES` is a `CHARACTER` constant which defines the INSEL block name(s) implemented in a particular subroutine. Like all literal constants in Fortran, it must be embedded in quotes. Some well-known INSEL block names are CONST, DO, CLOCK, PVI, PLOT etc. Make sure that you don't use an already existing INSEL block name in `BLOCK_NAMES`, because INSEL block names must be unique. Block names consist of letters, digits, and underscores, and start with a letter; the Block Wizard accepts at most 31 characters. If you want to define more than one block in your subroutine, the names have to be separated by at least one blank (space character). The inselEngine counts the names: each name corresponds to one operation mode, see `CTRL(OPMODE)` below. Older INSEL versions required an additional parameter `OPM` for the number of operation modes; it no longer exists.

`MIN_INS` The `MIN_INS` parameter defines the minimum number of inputs that a user of your block has to connect to the block(s), when it is used in an INSEL model. If a user of your block

connects less than MIN_INS inputs to one of the blocks listed in BLOCK_NAMES then the INSEL compiler generates an error message and does not execute the model.

MAX_INS The MAX_INS parameter defines the maximum number of inputs that a user is allowed to connect to the block (and defines the size of the INS array). In most cases, the number of block inputs that have to be connected by a user of your block will be a constant, i. e., MIN_INS and MAX_INS are set to the same value.

N_OUTS The N_OUTS parameter defines the size of the output array OUTS, hence is the (maximum) number of block outputs. From a user's point of view the number of block outputs must not necessarily be a constant but from a programmer's point of view it has to be a constant, because the size of the OUTS array is fixed.

N_INTS The N_INTS parameter defines the size of the INTEGER array INTS, and

N_REALS the N_REALS parameter defines the size of the REAL array REALS. These two arrays hold the state of a block instance. The inselEngine allocates them separately for each block instance in a model, initialises them with zero, and keeps their values from one call to the next. Hence, you must use INTS and REALS whenever your block has to memorise values from one call to another, because you do not know the number of block instances in advance – local variables are not kept between calls, see Remark 4 of the Fortran crash course. In older INSEL versions these arrays were called IP (with ten reserved elements at its beginning), RP, and DP.

MIN_PARAMS The MIN_PARAMS parameter defines the minimum number of numerical parameters that a user has to specify if s/he wants to use your block. If a user of your block provides less than MIN_PARAMS numerical parameters to one of the blocks listed in BLOCK_NAMES then the INSEL compiler generates an error message and does not execute the INSEL model.

MAX_PARAMS The MAX_PARAMS parameter defines the maximum number of numerical parameters that a user is allowed to specify for the use of your block (and defines the size of the PARAMS array). In most cases, the number of numerical block parameters will be a constant, i. e., MIN_PARAMS and MAX_PARAMS are set to the same value.

MIN_STRINGS The MIN_STRINGS parameter defines the minimum number of string parameters that a user has to specify for the use of your block. If a user of your block specifies less than MIN_STRINGS string parameters then the INSEL compiler generates an error message and does not execute the INSEL model.

MAX_STRINGS The MAX_STRINGS parameter defines the maximum number of string parameters that a user is allowed to specify for the use of your block (and defines the size of the STRINGS array). In most cases, the number of string parameters will be a constant (zero in most cases). Each string has 1024 characters and is padded with blanks.

GROUP The GROUP parameter defines the belonging of a block to an INSEL group. As shown in earlier Modules of this Tutorial within the framework of INSEL six block groups are defined.

- GROUP = 1: Constant block (C-block)
- GROUP = 2: Timer block (T-block)
- GROUP = 3: Standard block (S-block)
- GROUP = 4: Loop block (L-block)
- GROUP = 5: Delay block (D-block)
- GROUP = 6: If block (I-block)

The use of the INSEL group requires a rather deep understanding of the INSEL concepts and for your first INSEL blocks it is not recommended to go into the details, hence GROUP should be set to 3 – i. e., the Standard block group. The advanced INSEL programmer finds additional information later in this section.

When you want to design a block, you must provide values for all discussed variables in the PARAMETER statement. Because these values are constants it follows that all blocks which are defined in one subroutine must have the same layout. The information you provide in the PARAMETER statement is used by the inselEngine for memory allocation.

CTRL The control array CTRL is shared between the inselEngine and the block. Its elements are addressed by the names defined in `insel_block.inc`:

CTRL(RETCODE) CTRL(RETCODE), i. e., CTRL(1), contains the return code. It is set to zero by the inselEngine before each call. A block reports a warning or an error by setting it to the number of a message, see the section on the INSEL message system: numbers of the form 2xxxxx are warnings, numbers of the form 3xxxxx are errors. When a block returns an error in the Constructor call, the model is not executed.

CTRL(CALLMODE) CTRL(CALLMODE), i. e., CTRL(2), contains the call mode: IDENTIFY (−1), STANDARD (0), CONSTRUCT (1), or DESTRUCT (2).

CTRL(OPMODE) CTRL(OPMODE), i. e., CTRL(3), contains the operation mode, i. e., the position of the block's name in BLOCK_NAMES, starting with one. A negative value indicates the second variant of a block (see the NULL block below), so use ABS(CTRL(OPMODE)) if your block does not distinguish variants.

CTRL(BLOCKNR) CTRL(BLOCKNR), i. e., CTRL(4), contains the user-defined block number. In an INSEL model every block has a unique number.

CTRL(N_INS) CTRL(N_INS), i. e., CTRL(5), always contains the number of currently connected block inputs.

CTRL(JUMP) CTRL(JUMP), i. e., CTRL(6), is the Jump parameter as discussed in Module , page 88, for instance.

CTRL(DEBUG) CTRL(DEBUG), i. e., CTRL(7), contains the Debug level of a simulation run. When it is

zero, no debug information is generated. The Debug level can be set with the `-d` option when the inselEngine is called.

`CTRL(8) ... CTRL(10)`

`CTRL(8)` to `CTRL(10)` are the integer arguments of messages (`#8I#` to `#10I#`). In the Constructor call, timer blocks return their number of calls in `CTRL(8)`.

The remaining elements of `CTRL`, up to `NCTRL = 24`, are written by `INSEL_IDENTIFY` or reserved. Don't use them.

In C++, the same names are defined in the namespace `insel` of `insel_block.h`, with indices starting at zero, e. g., `ctrl[insel::RETCODE]` is `ctrl[0]`.

The declaration statements following the `PARAMETER` statement make use of the above mentioned INSEL parameters and SHOULD UNDER NO CIRCUMSTANCES BE CHANGED.
The line

```
C-----------------------------------------------------------------------
```

separates the non-executable statements from the first executable statement.

Maybe you have recognized that the dimension of the INSEL arrays exceeds the dimensions defined in the `PARAMETER` statement by one. The reason is very pragmatic: to avoid unnecessary compiler errors for the case where one of the parameters is equal to zero. Since all variables are handed over to the subroutine as pointers it doesn't make any difference. But the programmer must make sure that not more than `MAX_PARAMS` elements are used of the `PARAMS` array, for instance. Otherwise unforeseen computer crashes will result.

The following listings are taken from the current INSEL sources. The documentation headers are left out, and lines which are identical in all blocks are sometimes replaced by `....`

12.4.1 The CONST block

Let us start our block journey with one the most primitive INSEL blocks, the CONST block. It has one parameter p and one output y. During execution the CONST block performs the operation $y = p$, that's it. As the name indicates and as you know, the CONST block is a C-block. By definition, it is called only once in a simulation run, independent of any T-block settings. This is the code:

```
      SUBROUTINE FB0001(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      IMPLICIT          NONE
      INCLUDE           'insel_block.inc'
      CHARACTER*1024    BLOCK_NAMES
      INTEGER           MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &                  MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP
      PARAMETER        (BLOCK_NAMES = 'CONST'
     &,                 MIN_INS     = 0
     &,                 MAX_INS     = 0
```

```
     &,                N_OUTS      = 1
     &,                N_INTS      = 0
     &,                N_REALS     = 0
     &,                MIN_PARAMS  = 1
     &,                MAX_PARAMS  = 1
     &,                MIN_STRINGS = 0
     &,                MAX_STRINGS = 0
     &,                GROUP       = 1)
      CHARACTER*1024   STRINGS(MAX_STRINGS+1)
      INTEGER          CTRL(NCTRL),INTS(N_INTS+1)
      REAL             INS(MAX_INS+1),OUTS(N_OUTS+1),REALS(N_REALS+1),
     &                 PARAMS(MAX_PARAMS+1)
C-----------------------------------------------------------------------
      IF (CTRL(CALLMODE) .NE. STANDARD) THEN
         IF (CTRL(CALLMODE) .EQ. IDENTIFY) THEN
C           Identification call
            CALL INSEL_IDENTIFY(CTRL,STRINGS,BLOCK_NAMES,
     &         MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &         MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP)
            RETURN
         ELSE IF (CTRL(CALLMODE) .EQ. CONSTRUCT) THEN
C           Constructor call
         ELSE
C           Destructor call
         END IF
         RETURN
      END IF
C---- Standard call ----------------------------------------------------
      OUTS(1) = PARAMS(1)
      RETURN
      END
C-----------------------------------------------------------------------
```

The code should be completely clear by now. From the name of the subroutine we see that the CONST block is a member of the inselFB library, i. e., included in the Fundamental blocks library.

12.4.2 The SUM, MUL, MAX, and MIN blocks

The next example shows the use of operation modes.

```
      SUBROUTINE FB0002(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      IMPLICIT         NONE
      INCLUDE          'insel_block.inc'
      CHARACTER*1024   BLOCK_NAMES
      INTEGER          MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &                 MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP
      PARAMETER       (BLOCK_NAMES = 'SUM MUL MAX MIN'
     &,                MIN_INS     = 1
     &,                MAX_INS     = 999
     &,                N_OUTS      = 1
     &,                N_INTS      = 0
     &,                N_REALS     = 0
```

```
     &,                MIN_PARAMS  = 0
     &,                MAX_PARAMS  = 0
     &,                MIN_STRINGS = 0
     &,                MAX_STRINGS = 0
     &,                GROUP       = 3)
      CHARACTER*1024   STRINGS(MAX_STRINGS+1)
      INTEGER          CTRL(NCTRL),INTS(N_INTS+1)
      REAL             INS(MAX_INS+1),OUTS(N_OUTS+1),REALS(N_REALS+1),
     &                 PARAMS(MAX_PARAMS+1)
      INTEGER          I
C-----------------------------------------------------------------------
      IF (CTRL(CALLMODE) .NE. STANDARD) THEN
         IF (CTRL(CALLMODE) .EQ. IDENTIFY) THEN
C           Identification call
            CALL INSEL_IDENTIFY(CTRL,STRINGS,BLOCK_NAMES,
     &         MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &         MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP)
            RETURN
         ELSE IF (CTRL(CALLMODE) .EQ. CONSTRUCT) THEN
C           Constructor call
         ELSE
C           Destructor call
         END IF
         RETURN
      END IF
C---- Standard call ----------------------------------------------------
      GO TO (1,2,3,4) ABS(CTRL(OPMODE))
1     CONTINUE
      OUTS(1) = 0.0
      DO I = 1,CTRL(N_INS)
         OUTS(1) = OUTS(1) + INS(I)
      END DO
      RETURN
2     CONTINUE
      OUTS(1) = 1.0
      DO I = 1,CTRL(N_INS)
         OUTS(1) = OUTS(1) * INS(I)
      END DO
      RETURN
3     CONTINUE
      OUTS(1) = INS(1)
      DO I = 2,CTRL(N_INS)
         IF (INS(I) .GT. OUTS(1)) OUTS(1) = INS(I)
      END DO
      RETURN
4     CONTINUE
      OUTS(1) = INS(1)
      DO I = 2,CTRL(N_INS)
         IF (INS(I) .LT. OUTS(1)) OUTS(1) = INS(I)
      END DO
      RETURN
      END
C-----------------------------------------------------------------------
```

Four names

The value of BLOCK_NAMES shows us that the INSEL blocks SUM, MUL, MAX, and MIN are implemented in this subroutine – four blocks, i. e., four different operation modes. The number of block inputs for each block can vary from one to a maximum of 999. The four blocks have one output each, ergo N_OUTS = 1. Let's continue with the standard call.

The standard call starts with a computed GO TO statement. From our earlier discussion of the CTRL array, maybe you remember that CTRL(OPMODE) contains the current operation mode. Consequently, when ABS(CTRL(OPMODE)) has a value one, two, three, or four the GO TO branches to label 1, 2, 3, or 4 and continues execution there.

Label 1

At label 1 we find the code for the SUM block (the first mentioned in BLOCK_NAMES, i. e., operation mode 1). When you do remember that CTRL(N_INS) always contains the number of currently connected inputs, the code should be clear – including the RETURN statement.

Label 2

At label 2 we find the code for the MUL block (the second mentioned in BLOCK_NAMES, i. e., operation mode 2). It is very similar to the SUM block and the code should be clear again.

Labels 3 and 4

At labels 3 and 4 you find the code of the MAX (operation mode 3) and the MIN block (operation mode 4), respectively.

All four blocks perform rather similar operations, they all have the same flexible number of inputs, one output, no parameter. That is the reason why we have put the four of them into one file.

There is one further interesting point which you can learn from this example, and that is the fact that this block uses only one local variable, the loop index I, which gets its value in each call. All other variables come via the interface. In all four modes we use the same variable names, OUTS(1), for example, but they all represent completely different variables and values. The management of the variables is done by the inselEngine. Each of the four blocks can be used an arbitrary number of times in one INSEL model without causing any conflict with the variables. Isn't that great?!

12.4.3 The DIV block

The next example demonstrates the practical use of the different call modes and makes use of the message system.

```
      SUBROUTINE FB0004(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      IMPLICIT          NONE
      INCLUDE           'insel_block.inc'
      CHARACTER*1024    BLOCK_NAMES
      INTEGER           MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &                  MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP
      !TODO: define MOD & INTDIV
      PARAMETER        (BLOCK_NAMES = 'DIV MOD'
     &,                 MIN_INS     = 2
```

```
     &,                MAX_INS     = 2
     &,                N_OUTS      = 1
     &,                N_INTS      = 1
     &,                N_REALS     = 0
     &,                MIN_PARAMS  = 0
     &,                MAX_PARAMS  = 0
     &,                MIN_STRINGS = 0
     &,                MAX_STRINGS = 0
     &,                GROUP       = 3)
      CHARACTER*1024   STRINGS(MAX_STRINGS+1)
      INTEGER          CTRL(NCTRL),INTS(N_INTS+1)
      REAL             INS(MAX_INS+1),OUTS(N_OUTS+1),REALS(N_REALS+1),
     &                 PARAMS(MAX_PARAMS+1)
C-----------------------------------------------------------------------
      IF (CTRL(CALLMODE) .NE. STANDARD) THEN
         IF (CTRL(CALLMODE) .EQ. IDENTIFY) THEN
C           Identification call
            CALL INSEL_IDENTIFY(CTRL,STRINGS,BLOCK_NAMES,
     &         MIN_INS,MAX_INS,N_OUTS,N_INTS,N_REALS,MIN_PARAMS,
     &         MAX_PARAMS,MIN_STRINGS,MAX_STRINGS,GROUP)
            RETURN
         ELSE IF (CTRL(CALLMODE) .EQ. CONSTRUCT) THEN
C           Constructor call
         ELSE
C           Destructor call
            IF (INTS(1) .GT. 0) THEN
C              Display number of divisions by zero
               CTRL(9) = INTS(1)
               CTRL(RETCODE) = 205002
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
         END IF
         RETURN
      END IF
C---- Standard call -----------------------------------------------------
      IF (INS(2) .NE. 0.0) THEN
         IF (CTRL(OPMODE) .EQ. 1) THEN
            OUTS(1) = INS(1) / INS(2)
         ELSE
            OUTS(1) = MOD(INS(1), INS(2))
         END IF
      ELSE
         ! Returns NaN in case of division by 0
         OUTS(1) = 2
         OUTS(1) = ACOS(OUTS(1))
         IF (INTS(1) .EQ. 0) THEN
C           First division by zero
            INTS(1) = 1
            CTRL(RETCODE) = 205001
            CALL MSG(CTRL,REALS,STRINGS)
         ELSE
            INTS(1) = INTS(1) + 1
         END IF
      END IF
```

```
      RETURN
      END
C-------------------------------------------------------------------------
```

This is our first example with some code in the destructor call. The `PARAMETER` statement shows that the subroutine implements two blocks, DIV and MOD, which both require exactly two inputs (let's say x_1 and x_2), one output ($y = x_1/x_2$ for DIV), and one `INTEGER` state variable `INTS(1)`. We take care of the fact that a division by zero normally ends in a run-time error and completely removes the program which caused the run-time error from memory and thus stops it.

The standard call tests whether the second input x_2 – or more precisely `INS(2)` – is different from zero. In this case the DIV block performs the division x_1/x_2 – or, more precisely again `INS(1) / INS(2)` – and outputs the result on output one, i. e., `OUTS(1) = INS(1) / INS(2)`.

`INS(2) = 0`

What if `INS(2)` is equal to zero? In this case the block returns NaN (not a number, computed as `ACOS(2.0)`), displays a warning message, and continues program execution. But one warning message should be sufficient. If the DIV block is called many times with an invalid second input equal to zero, it is sufficient to display the warning message once, count the number of occurances and – at the very end of the model execution – inform the user how often this exception has occurred. This is exactly what has been programmed in the if-then-else statement.

When the second input is equal to zero we first check the value of `INTS(1)` – all elements of `INTS` and `REALS` are initialised with a value zero by the inselEngine, you can rely on this. When `INTS(1)` is equal to zero, it means that this is the first time that the DIV block shall divide by zero. We set `INTS(1)` equal to one, and set `CTRL(RETCODE)` to a value which stands for the severity of the message (a 2 as first digit indicates that this is a warning only – see the section on the INSEL message system) and the message number, in file `insel.msg` in this case. When we look up the file `insel.msg` we find the record

```
05001 Block #4I5.5#: Division by zero
```

so that the next statement `CALL MSG(CTRL,REALS,STRINGS)` results in a displayed warning message

```
W05001 Block xxxxx: Division by zero
```

where `xxxxx` indicates the block number `CTRL(BLOCKNR)` (with leading zeros) of the corresponding DIV block in the INSEL model.

When on a further call of the same DIV block the second input is equal to zero, the `ELSE` part of the `IF` statement finds that `INTS(1)` is not equal to zero and the block simply increases the value of `INTS(1)` by one – until the end of the execution of the INSEL simulation model. And then?

And then the same DIV block is called again by the inselEngine in the destructor call. Here we check, whether `INTS(1)` – our counter for the occurrences when `INS(2)` is equal to zero – is greater than zero, i. e., if during the execution of the INSEL model a divide-by-zero situation has occurred or not. If yes, we copy the counter to `CTRL(9)` and display it with message number 5002 of the `insel.msg` file.

Flexible memory

Please notice again: when an INSEL model uses more than one DIV block, each instance gets its own memory for `INTS(1)`. This means that the different DIV blocks all have their private counters. It is not possible to solve this problem with a local variable, let's say `INTEGER COUNT0`. Such a variable does not keep its value from one call to the next. And if we declared it as `INTEGER COUNT0 /0/` (which implies `SAVE`), there would be a conflict between the different blocks, because there would be only one counter. The result would be the total number of calls with `INS(2) = 0` of all DIV blocks in the INSEL model. It would also be a nice result, but not what was intended.

The last argumentation has shown what the `INTS` and `REALS` arrays are good for in INSEL: they hold the memory of each block instance from call to call.

12.4.4 The ROOT, GAIN, ATT, and OFFSET blocks

Check out the details of `fb0006.f` as a first example for some code in the Constructor call by yourself. The subroutine also implements the DELAYS block, which uses `REALS` to memorise its inputs until the next call.

```
      SUBROUTINE FB0006(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      ...
      PARAMETER       (BLOCK_NAMES = 'ROOT GAIN ATT OFFSET DELAYS'
     &,                MIN_INS     = 1
     &,                MAX_INS     = 10
     &,                N_OUTS      = 10
     &,                N_INTS      = 1
     &,                N_REALS     = 10
     &,                MIN_PARAMS  = 1
     &,                MAX_PARAMS  = 1
     &,                MIN_STRINGS = 0
     &,                MAX_STRINGS = 0
     &,                GROUP       = 3)
      ...
         ELSE IF (CTRL(CALLMODE) .EQ. CONSTRUCT) THEN
C           Constructor call
            IF (ABS(CTRL(OPMODE)) .LT. 1 .OR. ABS(CTRL(OPMODE))
     &         .GT. 5) THEN
C              Invalid operation mode
               CTRL(RETCODE) = 305126
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
            IF (ABS(CTRL(OPMODE)) .EQ. 1 .AND. ABS(PARAMS(1))
     &         .LE. 1.0) THEN
C              Invalid root exponent
               CTRL(RETCODE) = 305005
```

```
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
            IF (ABS(CTRL(OPMODE)) .EQ. 3 .AND. PARAMS(1) .EQ. 0.0) THEN
C              Zero is an invalid attenuator parameter
               CTRL(RETCODE) = 305023
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
            IF (ABS(CTRL(OPMODE)) .EQ. 5) REALS(1) = PARAMS(1)
         ELSE
C           Destructor call
            IF (INTS(1) .GT. 0) THEN
C              Display number of calls with negative input
               CTRL(9)  = INTS(1)
               CTRL(RETCODE) = 205004
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
         END IF
         RETURN
      END IF
C---- Standard call -------------------------------------------------
      GO TO (1,2,3,4,5) ABS(CTRL(OPMODE))
1     CONTINUE
C     ROOT
      DO I = 1,CTRL(N_INS)
          IF (INS(I) .GE. 0.0) THEN
             OUTS(I) = INS(I) ** (1.0 / PARAMS(1))
          ELSE
             IF (INTS(1) .EQ. 0) THEN
C               First call with negative input
                INTS(1) = 1
                CTRL(RETCODE) = 205003
                CALL MSG(CTRL,REALS,STRINGS)
             ELSE
                INTS(1) = INTS(1) + 1
             END IF
          END IF
      END DO
      RETURN
      ...
5     CONTINUE
C     DELAYS
      DO I = 1,CTRL(N_INS)
          OUTS(I) = REALS(I)
          REALS(I)  = INS(I)
      END DO
      RETURN
      END
C-----------------------------------------------------------------------
```

12.4.5 The T-block DO

So far, we have examined a C-block and some simple S-blocks. In order to understand how INSEL Timer blocks can be created let's have a closer look at one of the most

frequently used INSEL blocks: the DO block. This is its source code:

```
      SUBROUTINE FB0013(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      ...
      PARAMETER       (BLOCK_NAMES = 'DO'
     &,                MIN_INS     = 0
     &,                MAX_INS     = 1
     &,                N_OUTS      = 1
     &,                N_INTS      = 3
     &,                N_REALS     = 2
     &,                MIN_PARAMS  = 3
     &,                MAX_PARAMS  = 3
     &,                MIN_STRINGS = 0
     &,                MAX_STRINGS = 0
     &,                GROUP       = 2)
      ...
         ELSE IF (CTRL(CALLMODE) .EQ. CONSTRUCT) THEN
C           Constructor call
            IF (ABS(PARAMS(3)) .EQ. 0.0) THEN
C              Invalid increment
               CTRL(RETCODE) = 305019
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
            IF (PARAMS(3) .GT. 0.0) THEN
               IF (PARAMS(2) .LT. PARAMS(1)) THEN
C                 Invalid initial / final value
                  CTRL(RETCODE) = 305020
                  CALL MSG(CTRL,REALS,STRINGS)
               END IF
            END IF
            IF (PARAMS(3) .LT. 0.0) THEN
               IF (PARAMS(1) .LT. PARAMS(2)) THEN
C                 Invalid initial / final value
                  CTRL(RETCODE) = 305020
                  CALL MSG(CTRL,REALS,STRINGS)
               END IF
            END IF
            IF (CTRL(RETCODE) .NE. 0) RETURN
            INTS(2) = INT((PARAMS(2) - PARAMS(1) + PARAMS(3)) /
     &         PARAMS(3))
            IF (INT((PARAMS(2) - PARAMS(1) + PARAMS(3)) / PARAMS(3) +
     &         1.E-5)
     &         .GT. INTS(2)) THEN
               INTS(2) = INTS(2) + 1
            END IF
            CTRL(8) = INTS(2)
         ELSE
C           Destructor call
         END IF
         RETURN
      END IF
C---- Standard call -------------------------------------------------
      IF (INTS(3) .EQ. 0) THEN
C        First call in DO loop
```

```
         INTS(1) = 1
         INTS(3) = 1
         REALS(1) = DBLE(PARAMS(1))
         REALS(2) = DBLE(PARAMS(3))
      END IF
      IF (INTS(1) .LE. INTS(2)) THEN
C        NOTE: Using double precision temporarily, in hope to avoid 0.3999999
         OUTS(1) = SNGL(REALS(2) * (INTS(1)-1) + REALS(1))
         INTS(1) = INTS(1) + 1
         IF (CTRL(N_INS) .EQ. 1) THEN
C           There is an input connected
C           Hence, DO is used as a subtimer, ie set jump to one
            CTRL(JUMP) = 1
         END IF
      ELSE
         IF (CTRL(N_INS) .EQ. 0) THEN
C           There is no input connected
C           Hence, DO is used as a timer, ie set LEND true
            CTRL(CALLMODE) = 2
         END IF
C        Prepare next DO loop
         INTS(3) = 0
      END IF
      RETURN
      END
C-----------------------------------------------------------------------
```

Let us start with the Constructor call. At first, a few plausibility checks are made: Is the increment different from zero? Does the sign of the increment fit the order of initial and final value? If something is wrong, the block returns and inselEngine will reject model execution.

If everything is okay so far, the constructor call calculates the number of required calls to the DO block, saves it in `INTS(2)`, and returns it to the inselEngine in `CTRL(8)`.

The first thing the DO block checks in standard call is, whether the block is called in standard call for the first time. More precisely, the block checks whether `INTS(3)` is equal to zero. Remember, that the DO block can be nested. This means, that there can be many “first calls” to the block. Consequently, `INTS(1)` is used as counter for the number of calls and `INTS(3)` is used as reset memory. The initial value and the increment are kept in `REALS(1)` and `REALS(2)`.

As long as `INTS(1)` has not reached the number of `INTS(2)` calls output one is computed from the initial value, the increment and the number of calls, and `INTS(1)` keeps track of the number of calls. The next statement

```
         IF (CTRL(N_INS) .EQ. 1) THEN
C           There is an input connected
C           Hence, DO is used as a subtimer, ie set jump to one
            CTRL(JUMP) = 1
         END IF
```

requires your full concentration.

Remember, CTRL(N_INS) contains the actual number of block inputs. The DO block has one optional input. If it has an input, the DO block is used as a subtimer, if not, it is the main timer in the simulation model. If the DO block is a subtimer, it has a negative jump parameter, pointing to the preceding timer in the calculation list. In this case, the DO block has to give control to the preceding timer, when the DO block itself has reached its final call.

But as long as the DO loop is running the successor of the DO block in the calculation list has to be called next. The DO block informs the calling inselEngine by setting the jump parameter CTRL(JUMP) to a value of one.

The last ELSE branch handles the last call case. When the DO block has no input – i. e., is the main timer – it sets the logical end condition to true, i. e., sets CTRL(CALLMODE) to a value of two, which means that the inselEngine has to switch to Destructor call mode. In any case the DO block prepares for a new first call by setting INTS(3) to zero.

Two INSEL block mechanisms

In conclusion, we have learnt two basic mechanisms in INSEL. First, non-Standard blocks can inform the inselEngine not to use the jump parameter from the calculation list, but to call the block's successor in the list. Second, blocks can inform the inselEngine to end a simulation run and switch to Destructor call mode.

12.4.6 The I-block IF

It is easy now to understand the fundamental I-block IF. Here comes the code:

```
      SUBROUTINE FB0022(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      ...
      PARAMETER       (BLOCK_NAMES = 'IF'
     &,                MIN_INS     = 2
     &,                MAX_INS     = 2
     &,                N_OUTS      = 1
     &,                N_INTS      = 0
     &,                N_REALS     = 0
     &,                MIN_PARAMS  = 0
     &,                MAX_PARAMS  = 0
     &,                MIN_STRINGS = 0
     &,                MAX_STRINGS = 0
     &,                GROUP       = 6)
      ...
C---- Standard call -------------------------------------------------
      ! 1, 2, 0.5, -0.5, INF are truthy
      ! 0 and NaN are falsy
      IF (INS(2) .LE. -0.5 .OR. INS(2) .GE. 0.5) THEN
         CTRL(JUMP)  = 1
         OUTS(1) = INS(1)
      END IF
      RETURN
      END
C--------------------------------------------------------------------
```

The Constructor and Destructor call sections are empty. Remember that the IF block jumps over its successors (positive jump parameter) as long as the condition input two is false, i. e., zero. When the condition input is true (in the current version: ≤ -0.5 or ≥ 0.5), the IF block must give control to its successors by setting the jump parameter CTRL(JUMP) to one and by passing input one to output one – that's it.

12.4.7 The D-block DELAY

The fundamental D-block DELAY is easy to understand, too.

```
      SUBROUTINE FB0015(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      ...
      PARAMETER       (BLOCK_NAMES = 'DELAY'
     &,                MIN_INS     = 1
     &,                MAX_INS     = 1000
     &,                N_OUTS      = 1000
     &,                N_INTS      = 0
     &,                N_REALS     = 0
     &,                MIN_PARAMS  = 0
     &,                MAX_PARAMS  = 1000
     &,                MIN_STRINGS = 0
     &,                MAX_STRINGS = 0
     &,                GROUP       = 5)
      ...
         ELSE IF (CTRL(CALLMODE) .EQ. CONSTRUCT) THEN
C           Constructor call
C           Initialize the output signal
            DO I = 1,CTRL(N_INS)
               OUTS(I) = PARAMS(I)
            END DO
         ELSE
C           Destructor call
         END IF
         RETURN
      END IF
C---- Standard call -------------------------------------------------
      DO I = 1,CTRL(N_INS)
         OUTS(I) = INS(I)
      END DO
      RETURN
      END
C---------------------------------------------------------------------
```

The Constructor call initializes all outputs by the corresponding parameters. In Standard call the CTRL(N_INS) block inputs are just written to the outputs. Remember that the delay effect is based on the fact that D-blocks are always sorted to the end of the calculation list.

12.4.8 The L-block NULL

Our last INSEL block source code example is the slightly more complex NULL block. Let us look a the code portion by portion. Since the block uses several PARAMS, INTS, and REALS, we provide the documentation header first:

```
C #Details
C    #Inputs
C       #INS(1)  Signal $y = f(x)$, which corresponds to the output $x$
C    #Outputs
C       #OUTS(1) Signal $x$ which is varied iteratively until $y
C               = 0 \pm \Delta y_{\rm max}$. This output has to be
C               connected to a corresponding TOL block.
C       #OUTS(2) Indicator $i$ for iteration failure
C               \begin{detaillist}
C                  \item[0] Solution found
C                  \item[1] Too many iterations
C                  \item[2] Both function values positive at boundaries
C                  \item[3] Both function values negative at boundaries
C                  \item[4] Found trivial solution
C               \end{detaillist}
C    #Parameters
C       #PARAMS(1) Mode
C              \begin{detaillist}
C                 \item[0] Involution algorithm (in the current version
C                          this is the only option)
C              \end{detaillist}
C       #PARAMS(2) Lower limit $x_{\rm min}$ of the iteration interval
C       #PARAMS(3) Upper limit $x_{\rm max}$ of the iteration intervall
C       #PARAMS(4) Tolerance $\Delta y_{\rm max}$ for the accuracy of the
C              calculated root
C       #PARAMS(5) Maximum number $N_{\rm max}$ of iterations
C       #PARAMS(6) Output value for $x$ which is returned when the number
C              $N_{\rm max}$ of iterations is reached or no solution was
C              found within the iteration interval
C    #Strings
C       #None
C #Internals
C    #Integers
C       #INTS(1) Integer representation of mode PARAMS(1)
C       #INTS(2) Counter for the number of calls
C       #INTS(3) Memory for unsuccessful iteration
C       #INTS(4) Memory for position of TOL block (no longer used)
C       #INTS(5) Set to 1 when NULL found a postive function value in the
C               iteration interval, otherwise 0
C       #INTS(6) Set to 1 when NULL found a negative function value in
C               the iteration interval, otherwise 0
C       #INTS(7) Counter for the number of no solution in the iteration
C               interval
C       #INTS(8) Counter for the number of unsuccessful iterations
C       #INTS(9) Counter for the number of trivial solutions
C    #Reals
C       #REALS(1)  Left  interval limit
C       #REALS(2)  Function value at left limit
C       #REALS(3)  Right interval limit
C       #REALS(4)  Function value at right limit
```

As can be seen from PARAMS(1) an involution algorithm is the only implemented option in the current version. All parameters including the indicator OUTS(2) should be clear from the description. So let us inspect the declaration and Constructor and Destructor call sections.

```
      SUBROUTINE FB0054(CTRL,INS,OUTS,INTS,REALS,PARAMS,STRINGS)
      ...
      PARAMETER       (BLOCK_NAMES = 'NULL'
     &,                MIN_INS     = 1
     &,                MAX_INS     = 1
     &,                N_OUTS      = 2
     &,                N_INTS      = 9
     &,                N_REALS     = 4
     &,                MIN_PARAMS  = 6
     &,                MAX_PARAMS  = 6
     &,                MIN_STRINGS = 0
     &,                MAX_STRINGS = 0
     &,                GROUP       = 4)
      ...
         ELSE IF (CTRL(CALLMODE) .EQ. CONSTRUCT) THEN
C           Constructor call
            INTS(1) = INT(ANINT(PARAMS(1)))
            IF (CTRL(OPMODE) .LT. 0) INTS(1) = INTS(1) - 1
            IF (INTS(1) .LT. 0 .OR. INTS(1) .GT. 0) THEN
C              Invalid mode
               CTRL(RETCODE) = 305011
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
            INTS(1) = INTS(1) + 1
            IF (PARAMS(3) .LT. PARAMS(2)) THEN
C              Invalid iteration interval
               CTRL(RETCODE) = 305092
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
            IF (PARAMS(4) .LE. 0.0) THEN
C              Invalid error tolerance
               CTRL(RETCODE) = 305051
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
            IF (ANINT(PARAMS(5)) .LT. 1) THEN
C              Invalid number of maximal iterations
               CTRL(RETCODE) = 305093
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
            INTS(2) = 1
            OUTS(1) = PARAMS(2)
         ELSE
C           Destructor call
            IF (INTS(7) + INTS(8) .GT. 0) THEN
C              Display number of unsuccessful iterations
               CTRL(9) = INTS(7) + INTS(8)
               CTRL(RETCODE) = 205097
               CALL MSG(CTRL,REALS,STRINGS)
```

```
            END IF
            IF (INTS(9) .GT. 0) THEN
C              Display number of trivial solutions
               CTRL(9)  = INTS(9)
               CTRL(RETCODE) = 205151
               CALL MSG(CTRL,REALS,STRINGS)
            END IF
         END IF
         RETURN
      END IF
```

In the declaration section `GROUP = 4` makes NULL to an L-block.

Mode 0 vs. 1

The Constructor call checks the mode parameter first. INSEL 7, HP VEE, and INSEL 8 with VSEit follow the convention that the mode starts counting with zero. In graphical user interfaces pop-up menus usually use zero to indicate the first item. INSEL 8 started to support MATLAB and Simulink with the Renewable Energy blockset. Simulink uses index 1 for the the first item in pop-up menus. So, we distinguish both cases by positive or negative operation modes in `CTRL(OPMODE)`. If positive, the first item has index zero, if negative, the first item is considered as one. What follows are three simple plausibility checks for parameters two to five. Finally, the counter `INTS(2)` is set to one and the left interval boundary `PARAMS(2)` is put on output one, so that the first function value $f(x_{\min})$ arrives at the first Standard call.

We had seen from the DIV block already, that a typical use of Destructor calls are summaries of errors and warnings. The same applies here.

Next we check how the NULL block starts the iteration.

```
C---- Standard call -------------------------------------------------
      OUTS(2) = 0.0
      GO TO (1) INTS(1)

1     CONTINUE
C     Method of nested intervals
      INTS(2) = INTS(2) + 1
      IF (INTS(2) .EQ. 2) THEN
C        Evaluate first function value
         IF (ABS(INS(1)) .LE. PARAMS(4)) THEN
C           Found trivial solution
            INTS(3) = 1
            OUTS(1) = PARAMS(6)
            OUTS(2) = 4.0
            IF (INTS(9) .EQ. 0) THEN
               CTRL(RETCODE) = 205152
               CALL MSG(CTRL,REALS,STRINGS)
               INTS(9) = 1
            ELSE
               INTS(9) = INTS(9) + 1
            END IF
            RETURN
         END IF
```

```
      REALS(1)  = PARAMS(2)
      REALS(2)  = INS(1)
      OUTS(1) = PARAMS(3)
      RETURN
   END IF
```

On its first Standard call (`INTS(2) = 2`) the NULL block receives the function value `INS(1)` which corresponds to `PARAMS(2)`. Should this value be less than the accuracy tolerance defined as `PARAMS(4)` then there is a trivial solution. When this happens for the first time, a warning message is displayed. Apart from this exception, `PARAMS(2)` and its corresponding function value are stored in `REALS(1)` and `REALS(2)`, respectively, and the right interval boundary `PARAMS(3)` is put on output one.

The next call (`INTS(2) = 3`) again checks for the trivial case first. If false, `PARAMS(3)` and `INS(1)` are stored in `REALS(3)` and `REALS(4)` and the algorithm starts. In order to have a solution in the interval `[REALS(1),REALS(3)]`, one of the function values `REALS(2)` and `REALS(4)` must be positive, while the other one must be negative. If this is the case, it follows from the continuity of the function to analyze that there must be a root in the iteration interval and the NULL block puts the center of the iteration interval on output one. Otherwise a warning message is generated.

The core algorithm loop is this:

```
      IF (ABS(INS(1)) .LE. PARAMS(4)) THEN
C        Found the solution
         CTRL(JUMP)  = 1
         INTS(2) = 1
         INTS(3) = 0
         RETURN
      END IF
      IF (INTS(3) .EQ. 1) THEN
C        Iteration has already been done unsuccessfully
         CTRL(JUMP)  = 1
         INTS(2) = 1
         INTS(3) = 0
         RETURN
      END IF
      IF (INTS(2) .GE. PARAMS(5)) THEN
C        Maximum number of iterations exceeded
         IF (INTS(8) .EQ. 0) THEN
            CTRL(RETCODE) = 205098
            CALL MSG(CTRL,REALS,STRINGS)
            INTS(8) = 1
         ELSE
            INTS(8) = INTS(8) + 1
         END IF
         INTS(3) = 1
         OUTS(1) = PARAMS(6)
         OUTS(2) = 1.0
         RETURN
      END IF
```

```
C     Determine next x
      IF (INS(1) .GT. 0.0) THEN
         IF (REALS(2) .GT. 0.0) THEN
            REALS(1) = OUTS(1)
            REALS(2) = INS(1)
         ELSE
            REALS(3) = OUTS(1)
            REALS(4) = INS(1)
         END IF
      ELSE
         IF (REALS(2) .GT. 0.0) THEN
            REALS(3) = OUTS(1)
            REALS(4) = INS(1)
         ELSE
            REALS(1) = OUTS(1)
            REALS(2) = INS(1)
         END IF
      END IF
      OUTS(1) = (REALS(1) + REALS(3)) / 2.0
      RETURN
      END
C-----------------------------------------------------------------------
```

At first, three conditions are tested. When the solution is found, the block resets `INTS(2)` to one and `INTS(3)` to zero and – as we have seen several times already – sets the jump parameter `CTRL(JUMP)` to one indicating the end of the loop and the successor of the NULL block (and not the TOL) is called by inselEngine. The other two cases are that `INTS(3)` has a value of one, i. e., the iteration has already failed (for example because of a trivial solution), or that the maximum number of iterations has been reached.

Exercise 12.7 As long as all three conditions are false a new interval center will be determined and written to output one. Please read and understand, how the `REALS` are updated in every iteration step and how the case-by-case analysis is made in which interval the search for the root continues.

12.5 Interfacing INSEL with Python

The Python package `insel` (`pip install insel`) runs INSEL models from Python. It calls the `insel` command line program of the INSEL installation and parses its text output into a float, a list of floats, or a list of lists of floats. The package is also a convenient way to write unit tests for INSEL blocks and models – including your own user blocks. Its public functions are `insel.block`, `insel.template`, and `insel.run`.

`insel.block` `insel.block(name, *inputs, parameters=[], outputs=1)` runs a single block with the given inputs and parameters and returns its outputs.

```
>>> import insel
>>> insel.block('sum', 2, 3)
5.0
>>> insel.block('do', parameters=[1, 10, 1])
```

```
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
>>> insel.block('gain', 2, 5, 7, parameters=[3], outputs=3)
[6.0, 15.0, 21.0]
```

`insel.template` `insel.template(path, **parameters)` runs a template, i. e., an `.insel` or `.vseit` model which contains placeholders like `$a || 3$` (name and default value), after substituting the given parameters. A relative path is searched in the folder `templates` of the current working directory, and `.insel` is appended if the path has no extension.

```
>>> insel.template('a_times_b', a=7, b=3)
21.0
```

`insel.run` `insel.run(path)` runs an existing model without any substitution.

```
>>> insel.run('templates/one_to_ten.insel')
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

User blocks User blocks are loaded like all other INSEL blocks, so they can be tested in the same way. With the UBFIRST block of this Module in radians mode ($p_1 = 1$) and $p_2 = 2$, $\sin(0) + 2 \cdot 3/2$ is expected:

```
>>> insel.block('ubfirst', 0, 2, 3, parameters=[1, 2])
3.0
```

13 :: Programming INSEL extensions in Eclipse

When you have made your way down to this section of the Tutorial then you have probably already written several INSEL blocks on your own. At some stage you will certainly wish to have more support in programming and debugging than just the INSEL Block Wizard, your text editor, and the restricted debugging features in INSEL itself.

Programmers worldwide use integrated development environments (IDEs) in their daily work. Many IDEs are available, like Microsoft's Visual Studio, Sun's Netbeans, or IBM's Eclipse, to mention just a few. Visual Studio is commercial software, Netbeans and Eclipse are open-source projects.



Eclipse IDE. It was developed at IBM and first released in the year 2001.

some history: Text editors in general

SUN: Stanford University Network Netbeans (Sun Microsystems) vs. Eclipse (IBM) Name gemein!

NetBeans started as a student project in 1996. When Oracle acquired Sun in 2010, NetBeans became part of Oracle, which sees NetBeans as the official IDE for the Java Platform.

first version November 7, 2001

creation of the independent Eclipse Foundation in 2004

Sehr gute Seite fuer Eclipse basics:

`https://www.ics.uci.edu/~pattis/common/handouts/introtopythonineclipse/`

This section is meant as a short introduction into the installation of Eclipse and some compiler tools for Java, C/C++, Fortran, Ruby, to the novice Eclipse user.

The following software tools/plugins and their installation will be described:

- Java Development Kit (JDK)
- Eclipse
- Xcode command line tools
- C/C++ Development Tools (CDT)
- Fortran Development Tools (Photran)
- Window Builder

Version control of INSEL sources is done with Git, see the "INSEL Git Repositories" chapter of the INSEL Programmer's Guide — Subversion (SVN), formerly used at doppelintegral, has been retired.

Since Eclipse is written in Java the basic installation of Eclipse needs a working Java Runtime Environment. Therefor, we start with the installation of a Java package.

13.1 Java Development Kit

Java is cross-platform JVM



Java was developed by Sun Microsystems. On November 13, 2006, Sun Microsystems made the bulk of its implementation of Java available under the GNU General Public License (GPL).

later acquired by the Oracle Corporation.

Several Java packages, recommendation JDK (Java SE Development Kit) for Java Developers which includes a complete Java Runtime Environment plus tools for developing, debugging, and monitoring Java applications.



Download the JDK from `www.oracle.com`. The recommended version (November 2017) is Version 8 Update 152.

Java SE Development Kit 8u152

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement Decline License Agreement

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.94 MB	jdk-8u152-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.88 MB	jdk-8u152-linux-arm64-vfp-hflt.tar.gz
Linux x86	168.99 MB	jdk-8u152-linux-i586.rpm
Linux x86	183.77 MB	jdk-8u152-linux-i586.tar.gz
Linux x64	166.12 MB	jdk-8u152-linux-x64.rpm
Linux x64	180.99 MB	jdk-8u152-linux-x64.tar.gz
macOS	247.13 MB	jdk-8u152-macosx-x64.dmg
Solaris SPARC 64-bit	140.15 MB	jdk-8u152-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.29 MB	jdk-8u152-solaris-sparcv9.tar.gz
Solaris x64	140.6 MB	jdk-8u152-solaris-x64.tar.Z
Solaris x64	97.04 MB	jdk-8u152-solaris-x64.tar.gz
Windows x86	198.46 MB	jdk-8u152-windows-i586.exe
Windows x64	206.42 MB	jdk-8u152-windows-x64.exe

For Mac OS X download the .dmg file and install the JDK by a double-click. When you open a Terminal, `which java` tells you that the `java` command has been installed at `/usr/bin/java`. Typing `java -version` returns the version number, `1.8.0_152` in this case. You are now ready to install Eclipse.

13.2 Eclipse

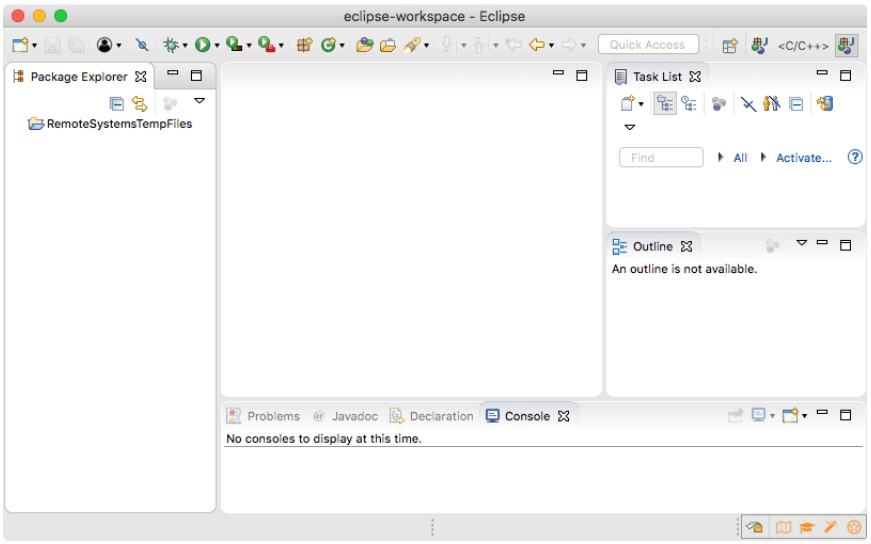
Download Eclipse from `www.eclipse.org`. Screenshots below are from an old Eclipse release (“Oxygen”, 2017); the general installation flow described here is still valid for current Eclipse releases, but the screenshots should be refreshed. A double-click on file `eclipse-inst-mac64.tar.gz` (48.1 MB) will extract the `Eclipse installer.app`. Another double-click on the installer .app let’s you choose between different Eclipse IDE versions, like *Eclipse IDE for Java Developers*, or *Eclipse IDE for C/C++ Developers*, for instance. Which version you choose is a matter of personal taste. Language support and other features can later be combined into any of the default packages.

Next, the installer asks for an *Installation Folder*. Again, the destination is your personal choice. By default, the installer suggests to create a directory named `eclipse` your home directory, `Eclipse.app` will then be installed in a subdirectory named `cpp-oxygen`, `java-oxygen`, or similar.



When you launch the application, Eclipse will ask you for a workspace directory. This is the directory where you usually do your Eclipse work. If you wish, select the *Use this as the default and do not ask again* check box. The workspace directory can be changed at any time via *Edit* > *Switch Workspace*.

Eclipse should welcome you (with the Java perspective, for instance).



The Eclipse window shows a menu and tool bar, the Package Explorer will be discussed in a minute, the central pane is a text area, at the right side a Task List and an Outline Perspective are displayed, at the bottom several Views are shown.

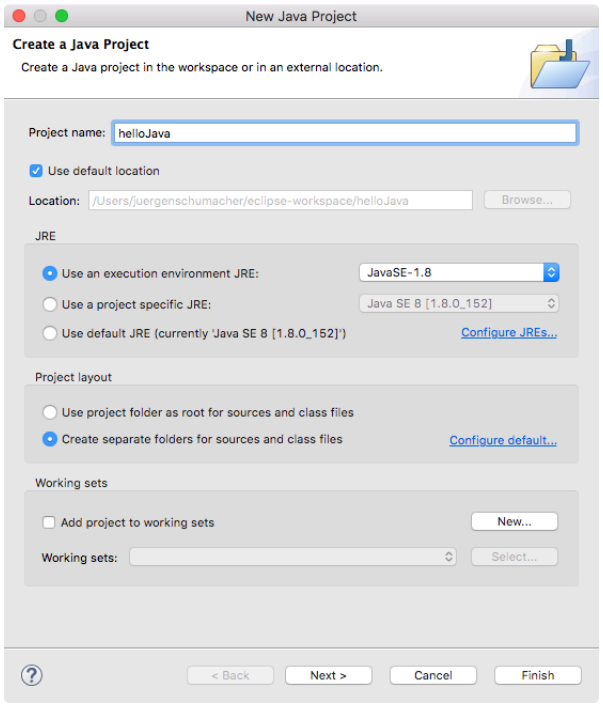
Some Eclipse terminology

Internet connection

If you are connected to the Internet via a Proxy server open the dialog *Eclipse* > *Preferences...* > *General* > *Network Connections* and set the proxy configuration.

13.2.1 A first Java project

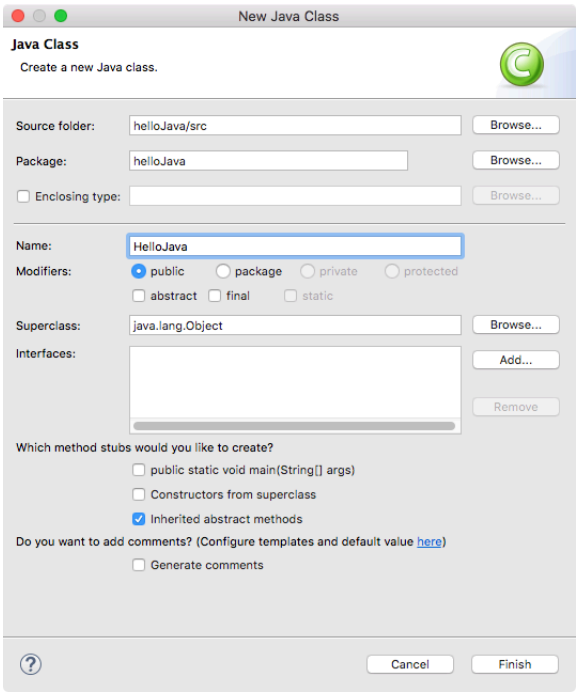
In order to test the Eclipse installation, start eclipse – if not already started. Create a new Java project via *File* > *New* > *Project* > *Java Project*.



The only thing to do here is to give the project a name, `helloJava`, for instance, and click the *Finish* button. Please observe that you can specify the JRE version you wish to use here.

The Package Explorer window shows that Eclipse has created a project with the desired name and a directory named `src` for the `.java` source file.

The next step is to create a new Java class via *File* > *New* > *Class*. At first, the hint that class names in Java should always start with an uppercase letter. So, the natural name for our first Java class is `HelloJava`.



When you have a closer look at the New Java Class window you'll observe that Eclipse suggested to create the new Java class in a package named `helloJava`. So, what is a package in Java?

Java package

The Java Tutorial says: "A package is a namespace that organizes a set of related classes or interfaces. Conceptually you can think of packages as being similar to different folders on your computer. . . . Because software written in Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages."

Naming conventions

In order to avoid duplication of names with programmers writing Java classes and interfaces worldwide package names should be unique. For instance, most companies use their reversed Internet domain name to begin their package names. If the domain name contains a hyphen or any other special character, or starts with a digit, or contains a reserved Java keyword like `int` the suggested convention is to add an underscore. For example `1ofmy-domains.int` would turn into `int_._1ofmy_domains`.

Since we talk about INSEL, we suggest to use the `insel.eu` domain to begin the package name.

A second convention is that package names should be written in all lowercase to avoid conflicts with class or interface names which should always start with an uppercase letter.

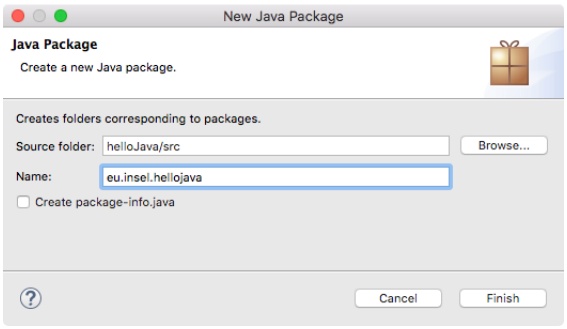
In the context of INSEL development, we have reserved the following package names for us:

- `eu.insel.vseit`
- `eu.insel.block`
- `eu.insel.userblock`
- `eu.insel.opensource`

If you intend to write a package for use with INSEL, please contact us, so that we can register your package name. If you intend to write a proprietary package, please name it using your research centers or companies Internet domain, e. g., `de.dlr.csp` or `com.firm.ourpackage`.

New Package

Next, create a new package in the source directory `src` via the *File* > *New* > *Package* dialog.



Give the package a name and click Finish.

New class

Coming back to the creation of the class you will see that Eclipse suggests the package name. Remember that class names start with an uppercase letter, like `HelloJava` for instance.

If you mark the *public static void main(String[] args)* check box, Eclipse will automatically create the `main` method which is required by any Java class. This is the code you get:

```
package helloJava;

public class HelloJava {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }

}
```

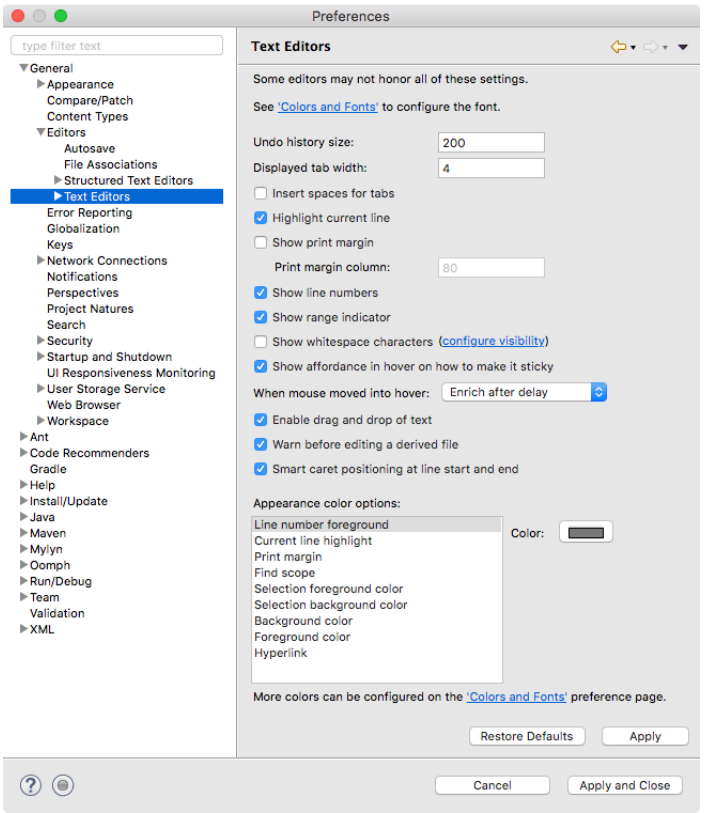
You may now wish to add something like

```
System.out.println(”Hello Java”);
```

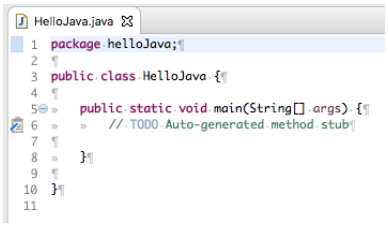
to the main method, save the file and press the *Run* button (the round green one with the white triangle) and observe how the string *Hello Java* is displayed in the Console pane at the bottom of the Eclipse window.

Eclipse Preferences

A last remark before we close the Java topic in Eclipse. In Eclipse nearly everything can be tailored to user-specific needs and wishes. For example, the behavior of text editors can be settled in the Text Editors pane, which can be opened via the *Eclipse* > *Preferences* dialog.



When you examine the generated code in more detail you will see, that the indention of the code lines is four characters by default. However, there aren't four blank characters in the code, but tab characters with a displayed width of four spaces – as defined in the Text Editors pane. This can be made visible by marking the *Show whitespace characters* check box.



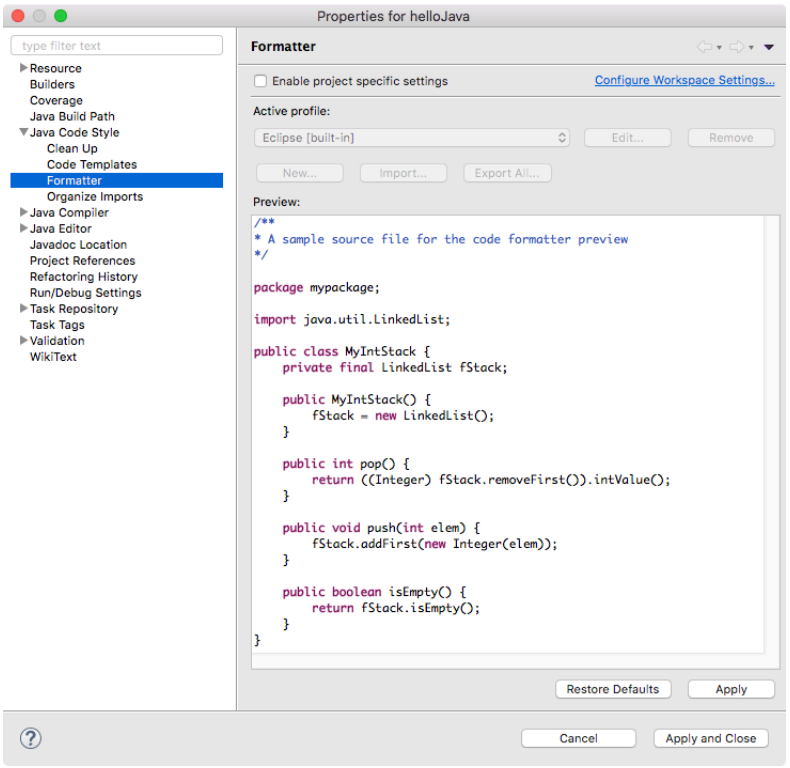
Another detail can be observed in this view: Mac OS X uses a one-byte line ending character LF (line feed) whereas older versions of Mac OS used CR (carriage return), Unix and Linux use LF, all Windows versions use CR LF since the beginning. For these reasons you might wish to always have whitespace characters visible in your text editor.

Everybody wishes and has his or her own style. Our style, the INSEL-developer style, is to use three bytes and space characters instead of tabs. One reason being, that the complete INSEL documentation is written in LaTeX, and LaTeX doesn't like tabs that much. So we don't either and recommend to change the *Displayed tab width* value to 3 and set the *Insert spaces for tabs* check box.

Line numbers are programmer's friends. They often help to understand compiler messages better. So you might like to set the *Show line numbers* check box.

InselJavaFormatter

If you wish to write your Java code one hundred percent INSEL compatible you can use the `InselJavaFormat.xml` formatter wo liegt diese Datei? for "pretty-printing." You can import the INSEL Java Formatter in Eclipse via the *File* > *Properties* dialog (if your current project is a Java Project) or via the *Eclipse* > *Preferences* dialog by using *Configure Workspace Settings...*.



A last hint at this point is that Eclipse provides a check box *Save automatically before build* which can be found under *Eclipse* > *Preferences* on the *General* > *Workspace* pane. This feature is very practical.

13.2.2 Installing Eclipse plugins

different mechanisms (siehe auch Eclipse Buch 3.2 Seite 16 f.)

Auslieferung mit einem Installationsmanager

Auslieferung in Form einer URL

Auslieferung als ZIP Datei

Extension Sites

13.3 C/C++ Development Tools (CDT)

Eclipse provides much more than just an IDE for Java programmers. Hundreds of plugins are available for all kind of Eclipse extension. For C/C++ programmers, the most important Eclipse plugin is CDT (C/C++ Development Tools), a plugin for the development of C or C++ code. CDT does not include a C or C++ compiler. Therefore, an installation of a C/C++ compiler is a necessary prerequisite if you want to use CDT.

Compiler installation

There are several ways how to install compilers in Mac OS X. Many installations come with an EXPLAIN Xcode 9.0.1

Xcode

Check if the full Xcode package is installed:

type xcode-select -

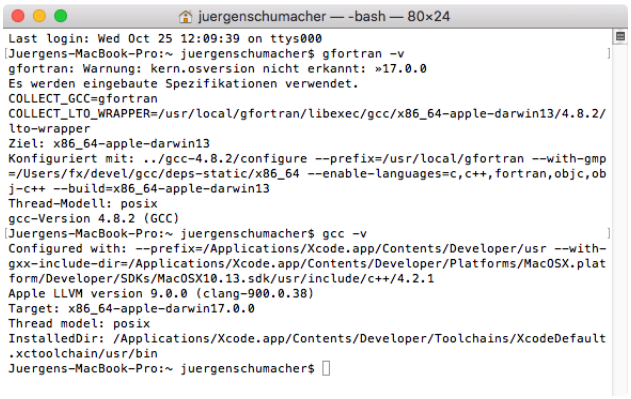
If the answer is

/Applications/Xcode.app/Contents/Developer

then the full Xcode package is already installed. Otherwise, if an error like

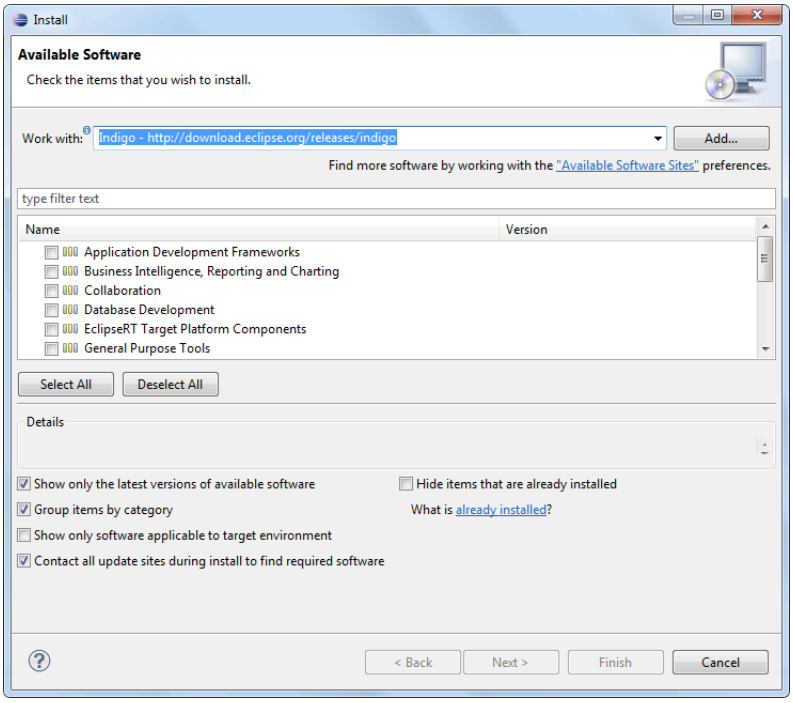
error: unable to get active developer directory

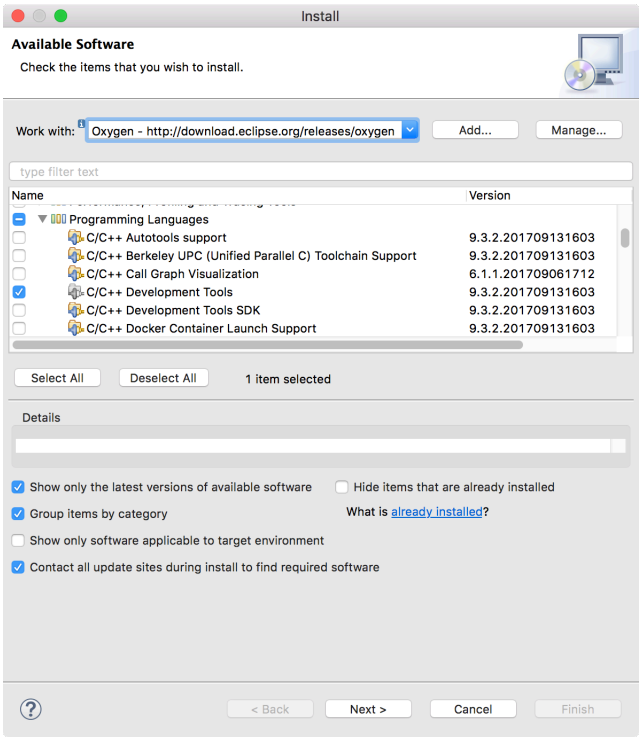
is returned you will need to update Xcode to the newest version (Check for updates in the App Store).



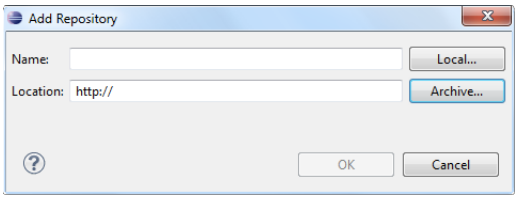
New Software mechanism in Eclipse

Assuming that you have a C/C++ compiler installed on your computer, open the *Help* > *Install New Software...* dialog in Eclipse. This is the main mechanism of how to install new software into Eclipse. When you open the dialog a window similar to the following one should open:

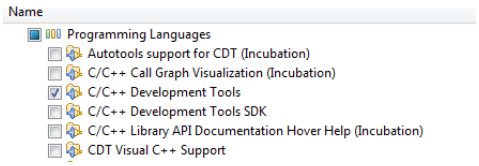

Install
Available Software
Check the items that you wish to install.
Work with:
Indigo - http://download.eclipse.org/releases/indigo
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
Application Development Frameworks
Business Intelligence, Reporting and Charting
Collaboration
Database Development
EclipseRT Target Platform Components
General Purpose Tools
Select All
Deselect All
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Finish
Cancel



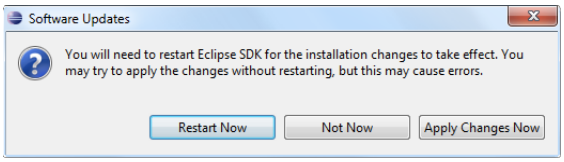
You can check your Eclipse installation for available sites by opening the *Work with* pull-down menu. If no sites are available, open the Add Repository dialog by a click on the *Add...* button.



You can check your eclipse installation for available sites by opening the *Work with* pull-down menu. If no sites are available, click on the *Add...* button open and enter the update site matching your installed Eclipse release (see `https://download.eclipse.org/releases/` for the list) in *Location* text field of the Add Repository dialog. Now browse to the *Programming Languages* item and mark the *C/C++ Development Tools* check box.



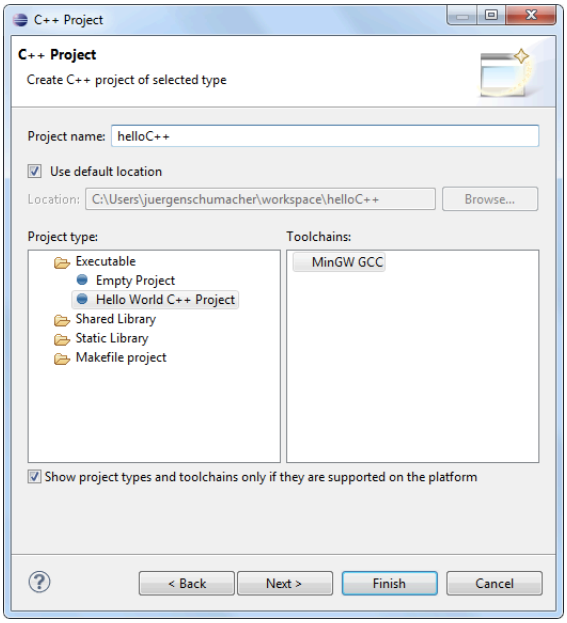
Click *Next* to continue. During installation, license terms have to be accepted. When the installation is finished, a restart of Eclipse is necessary and Eclipse welcomes you with CDT available.



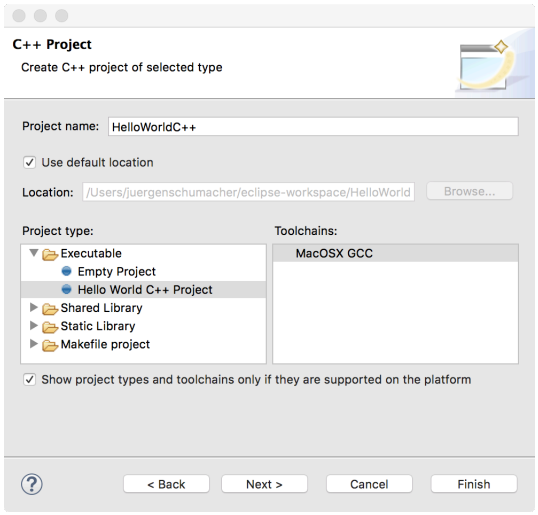
Hello C++ example

Now, when you choose *File* > *New* > *Project...* eclispe offers C/C++ projects.

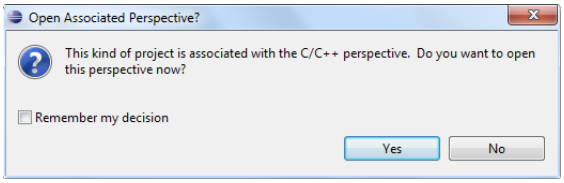
As a first test, we choose the default Executable project Hello World C++ Project and click Finish.



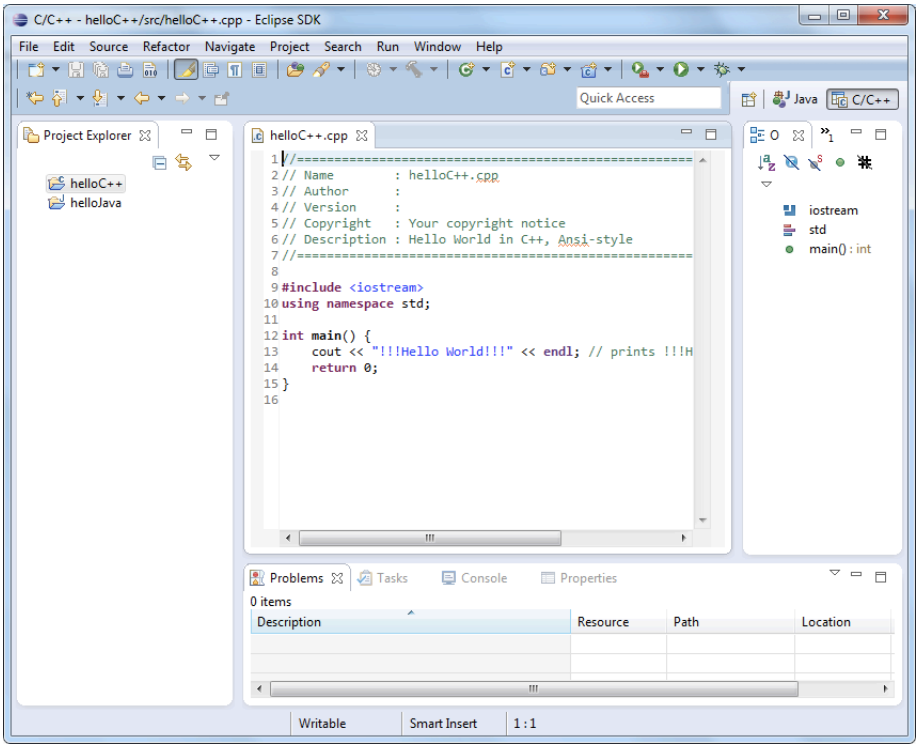
As a first test, choose a C++ Managed Build project and the default Executable project *Hello World C++ Project.*



Please observe that Eclipse automatically suggests to use the MacOSX GCC *Toolchain*, i. e., the installed gcc compiler.



Perspectives erläutern.



Not much has happened. The Package Explorer has turned into a Project Explorer, a couple of new views, like Console appear at the bottom window and a new C/C++ Perspective shows up in the upper right corner of the Eclipse window.

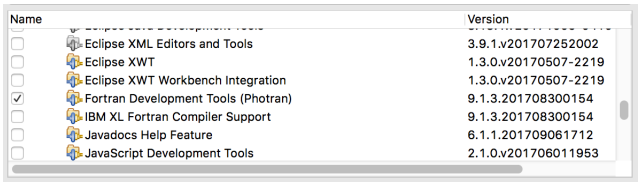
Simply running the project file does not work – since we have not yet compiled and linked the executable. We can do so by choosing the Project > Build Project dialog or the "hammer" shortcut. Running the executable bravely shows us the default `!!!Hello World!!!` string in the Console window.

13.4 Fortran Development Tools (Photran)

Photran is an integrated development environment and refactoring tool for Fortran. Photran is based on Eclipse and CDT. It supports all Fortran standards from Fortran 77 (our recommended standard) to Fortran 2008.

The installation of Photran into Eclipse is straight forward and very similar to the CDT installation. Hence, open *Help > Install New Software...*, browse to *Programming Languages* and select the *Fortran Development Tools (Photran)* check box.





Click *Next* to continue. During the installation, license terms have to be accepted. When the installation is finished, a restart of Eclipse is necessary.

Workaround

Ist das noch aktuell? In case, your Eclipse version does not offer a Photran package you might try this: Start Eclipse, then download the latest Photran zip file from `http://wiki.eclipse.org/PTP/photran/builds` click Help > Install New Software..., click the Add... button, click the Archive... button, choose the zip file you downloaded, click OK to close the Add Site dialog. This will return you to the Install dialog. Expand Photran (Fortran Development Tools) and check the box next to Photran End-User Runtime. If you are running Linux and have the Intel Fortran Compiler installed, or if you are on a Macintosh and have the IBM XL Fortran compiler installed, expand Fortran Compiler Support and select the appropriate compiler. Click the Next button If you get an error message, see below for troubleshooting information. Click the Finish button and agree to the license to complete the installation.

A hint for Mac OS X users

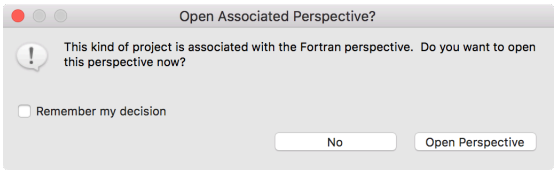
If you are using gfortran the compiler is installed in `/usr/local/bin` which is not on the PATH by default. If you are launching Eclipse from a Terminal, the PATH can be set by modifying `/etc/paths`. However, if you are launching Eclipse from the Finder or the Dock, then the PATH is not obtained from the shell or `/etc/paths`. Instead, it is obtained from `~/.MacOSX/environment.plist`.

Seems like this mechanismn was removed with OSX Lion. Use Eclipse ... Preferences ... C/C++ ... Build ... Environment and add Variable PATH with Value (e. g., /usr/local/bin) and check the Append variables to native environment check box. An alternative may be to update /etc/paths.

The format of the `environment.plist` file is as follows (change the path appropriately). If you create or edit this file, you will need to log out (or reboot) before the changes will take effect.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
   "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>PATH</key>
```

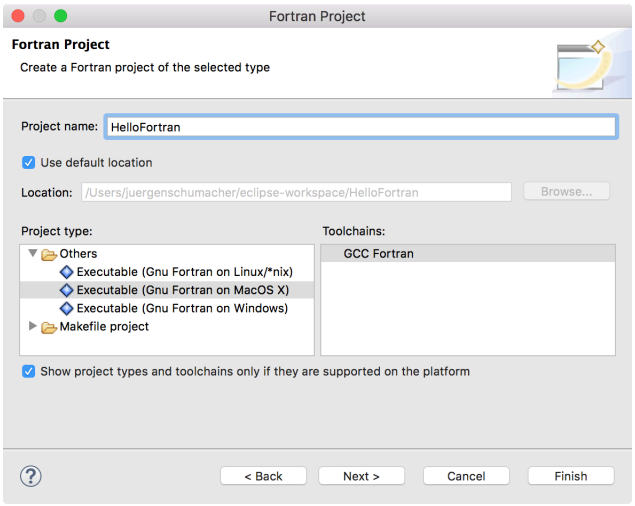
```
<string>/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
</string>
</dict>
</plist>
```



Eclipse suggests to change to the Fortran perspective, i. e., a view which is adapted to Fortran programmers.

Hello Fortran example

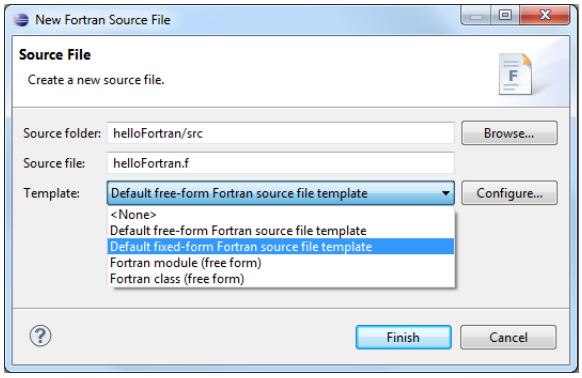
Now, when we choose the File > New > Project... menu item Eclispe offers Fortran projects. As a first test, we choose the Executable (Gnu Fortran on Windows) and click Finish. Please observe that Eclipse automatically suggests to use the GCC Fortran toolchain.



Now Eclipse suggests to change to the Fortran perspective again, we agree. We find the new project helloFortran in the Fortran Projects tree > remembering that helloC++ and helloJava are not Fortran projects > confusing.

In order to be compatible with the CDT and Java project structure, we create a `src` folder for the Fortran sources via File > New > Source Folder and type in the Folder name `src`.

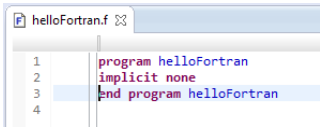
The next step is to create a new Fortran source file, named `helloFortran.f`, for example, via File > New > Source File.



We choose the Default fixed-form Fortran source file template and click Finish. When we try to build the executable – remember the hammer – we get a list of 7 errors:

```
../src/helloFortran.f:1.1: Error: Non-numeric character in statement label at (1)
../src/helloFortran.f:1.1: Error: Unclassifiable statement at (1)
../src/helloFortran.f:2.5: Error: Non-numeric character in statement label at (1)
../src/helloFortran.f:2.5: Error: Unclassifiable statement at (1)
../src/helloFortran.f:3.1: Error: Non-numeric character in statement label at (1)
../src/helloFortran.f:3.1: Error: Unclassifiable statement at (1)
make: *** [src/helloFortran.o] Error 1
```

Not bad, for a start. What happened? Since we have decided to use the Fortran 77 standard in two places, i. e., (i) by using the `.f` extension and (ii) by choosing the default fixed-form Fortran source file template, the `gfortran` compiler parses for Fortran 77 compatible statements. And these start in column 7, as indicated by the marked sixth column – the column for continuation lines, as you may already know or remember from our Fortran crash course. Hence, we indent the code correspondingly with space characters, recompile and see the errors vanishing.



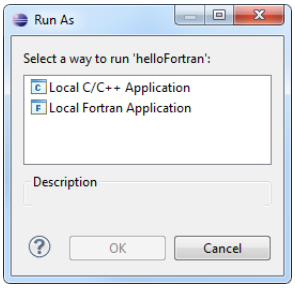
The program is correct but does nothing. So we add the statement

```
print*,"Hello Fortran!"
```

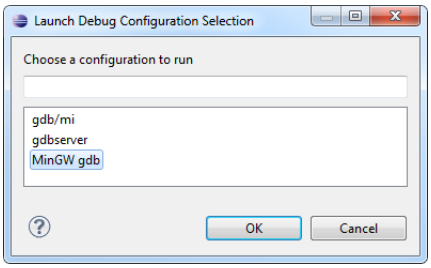
recompile and admire the result in the Console window.

Troubleshooting

Depending on you got here, you might be surprised to see the `!!!Hello World!!!` string from our helloC++ example. In this case highlight the helloFortran project, and click the Run button again. Now Eclipse will display a dialog and asks you to select a Run configuration.



Choose Local Fortran Application and click OK. Depending on your installation Eclipse offers several configurations.



A double-click on MinGW gdb finally achieves the desired result and we see `Hello Fortran!` in the Console pane.

Debugging

By the way, gdb stands for the Gnu Project Debugger. More about information about debugging can be found at the project's web page `www.gnu.org/s/gdb`, for example. If you wish to try debugging on the fly, just click the Debug button in Eclipe's toolbar (the little six-leg bug, next to the Run button).

There is not much to debug in our helloFortran example. In general, debugging code is extremely helpful in, yeah, debugging code and locating bugs.

In order to see the debugger work, you must set at least one Breakpoint where you wish the debugger to pause execution. You can do so by a double-click on the corresponding line margin. A small blue circle appears, indicating that there is a breakpoint. As usual, Eclipse will ask you to confirm a switch of perspective.

And indeed, the execution pauses at the breakpoint and waits for your input, which means that you are "in" the program. When your Fortran program contains variables, you can observe their current values and many things more.

It is really worthwhile to learn more about debugging – but not here.

13.5 WindowBuilder

The graphical user interface of INSEL 8 (inselVSEitUI) is mostly written using Swing, still with a few JavaFX parts. At some stage you might wish to add some graphical support to your own INSEL applications. A tool which is very useful for that purpose is WindowBuilder, which integrates into Eclipse and generates Java code for Swing, SWT (Standard Widget Toolkit), and RCP (Rich Client Platform) user interfaces from a drag-and-drop design view. It can be installed via *Help* > *Install New Software...* from the Eclipse Marketplace, or from the update site listed on WindowBuilder's Eclipse project page (search for its current release, rather than relying on any URL pinned to a specific old Eclipse version).

When you create a new WindowBuilder project via File > New > Other... WindowBuilder offers several wizards. INSEL 8 is mainly based on Swing components. Hence, let us choose a Swing Designer > Application Window.

Automatically the Swing Application wizard suggests to use our so far only Java project and its source directory. As package we choose our already existing package `eu.insel.hellojava`, the natural application name is `HelloSwing` – with capital H. WindowBuilder creates Java code for us:

```
package eu.insel.hellojava;

import java.awt.EventQueue;
import javax.swing.JFrame;

public class HelloSwing
{
   private JFrame frame;
   /**
    * Launch the application.
    */
   public static void main(String[] args) {
      EventQueue.invokeLater(new Runnable() {
         public void run() {
            try {
               HelloSwing window = new HelloSwing();
               window.frame.setVisible(true);
            } catch (Exception e) {
               e.printStackTrace();
            }
         }
      });
   }

   /**
    * Create the application.
    */
   public HelloSwing() {
      initialize();
   }

   /**
```

```
    * Initialize the contents of the frame.
    */
   private void initialize() {
      frame = new JFrame();
      frame.setBounds(100, 100, 450, 300);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

We can immediately launch the application via the Run button and see the empty application window.

We will definitively not go into an attempt to explain the basics of Swing here, but at least, we want to see a Hello Swing! in the window. Hence, we enter the WindowBuilder's Design window.

New world

At the bottom of the pane containing the Java code you see two tabs, Source and Design. Most probably, you are currently in the Source pane. A click on the Design tab takes you into a new world.

The full window features a Structure view with its Components and Properties, the full Palette with plenty of Swing components and – that is the best – a preview of your new Swing application.

Before you can start to drag and drop components into the preview window, Swing requires a Layout Manager. We choose a FlowLayout. The tooltip of the FlowLayout says "A flow layout arranges components in a left-to-right flow, much like lines of text in a paragraph. Flow layouts are typically used to arrange buttons in a panel. It will arrange buttons left to right until no more buttons fit on the same line."

When you select FlowLayout in the Components palette and move the mouse pointer to the preview window, the preview window displays a green frame and a + sign, indicating the ContentPanel as taget for the layout. Just drop the layout there.

You may then select the JLabel component and drop it in the preview window, too. Enter some nice text like "Hello Swing!" and you are done. Saving and running the application displays what we wanted.

Programming can be so easy and wonderful – sometimes!

The generated code is easy to read:

```
frame = new JFrame();
frame.setBounds(100, 100, 450, 300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

JLabel lblNewLabel = new JLabel("Hello Swing!");
lblNewLabel.setHorizontalAlignment(SwingConstants.LEFT);
frame.getContentPane().add(lblNewLabel);
```

We stop our excursion to the WindowBuilder tool here.

We have gone a long path to reach this point, used three different programming languages (Java, C/C++, and Fortran) and could start working on new software. One part, however, is still missing and that is *Version Control*.

Version control for all INSEL sources (engine, blocks, tools, GUI, and this documentation) is done with Git, hosted on Gitea at Concordia University. Eclipse includes Git support out of the box (menu *Window* > *Perspective* > *Open Perspective* > *Other...* > *Git*), so no additional plugin is required. See the "INSEL Git Repositories" chapter of the INSEL Programmer's Guide for the list of repositories, clone URLs, and access instructions.

13.6 Eclipse as INSEL block IDE

13.6.1 A makefile project for user block development

User blocks are compiled with the makefile which is installed as `makefile` in the INSEL installation directory. It compiles all Fortran and C++ files in the `src` directory of `insel.work/inselUB` into the library `libInselUB` (`.so`, `.dll`, or `.dylib`). Block files must start with `ub` in lower case, and each one defines the block function of the same name, e. g., `SUBROUTINE UBFOR` in `ubfor.f` or `ubcpp()` in `ubcpp.cpp`. The makefile collects these names into the block table returned by `insel_library()`, so no other registration is needed. In an Eclipse makefile project the build command is

```
make -f <INSEL_HOME>/makefile
```

run from `insel.work/inselUB/resources`, where `<INSEL_HOME>` is the INSEL installation directory. The compiler flags are

```
FCFLAGS = -g -Og -c -Wall -Wno-intrinsic-shadow -Wno-unused-dummy-argument
          -Wno-integer-division -fno-underscoring -fdefault-real-8
          -fdefault-double-8 -fmessage-length=0 -Werror=line-truncation
          -finit-real=snan -finit-integer=-999999999
CCFLAGS = -std=c++17 -g -Og -c -Wall -fmessage-length=0
```

plus `-fPIC` on Linux. All reals are double precision. Local variables are not static: any state of a block instance which must survive between two calls belongs in `INTS` and `REALS`. Uninitialised locals are NaN or -999999999, which makes their use easy to detect.

13.6.2 Debugging user blocks in Eclipse

1. download the most recent GDB from https://www.sourceware.org/gdb/download/
2. expand the gdb-7.12.1.tar.xz file: `tar xopf gdb-7.12.1.tar.xz`
3. `cd gdb-7.12.1` in terminal to open the gdb folder
4. then follow the instructions in the README file in the gdb folder, or simply follow the following steps:
5. `./configure`, wait for the terminal
6. `make` and wait again (which can take some time)
7. `sudo make install`

Now gdb is installed at /usr/local/bin/

Bugs, bugs, bugs. Programming in general, or writing INSEL blocks in particular implies the search for errors and bugs in the source code before a program can be executed without failure.

The only way of finding errors and bugs before a program can be executed is to study error messages generated by the compiler and correct the mistakes in the sources. However, compiler messages are not always easy to interpret. Sometimes error messages

relate to statements which have nothing to do with the error's source. But there is no other way. Gaining experience with compiler messages is a tough experience every programmer has to undergo.

Once code can be executed, usually a test phase starts where programmers check whether their code executes in the expected way, which is not always the case for new code. Strange and unexpected results may appear or the program which executes the new code even crashes. What then?

The old-fashioned way to find such bugs is to modify the source code, add output of some intermediate results via print statements or dialog boxes – in INSEL the standard way to output intermediate results is provided by the INSEL message system, as described earlier in this Module on page 256 ff.

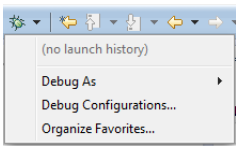
The process of searching, finding and removing bugs from source code is called debugging. Fortunately, helpful tools exist which make it easier for programmers to find bugs, so-called source-code debuggers. Such debuggers allow programmers to inspect source code during step-by-step program execution and observe variables and their current values without code changes.

In order to use debugging features it is necessary to compile the code to debug with a debug flag. The INSEL makefile printed on page 303 already compiles user blocks with `-g -Og`, i. e., with debug information and only optimisations which do not interfere with debugging.

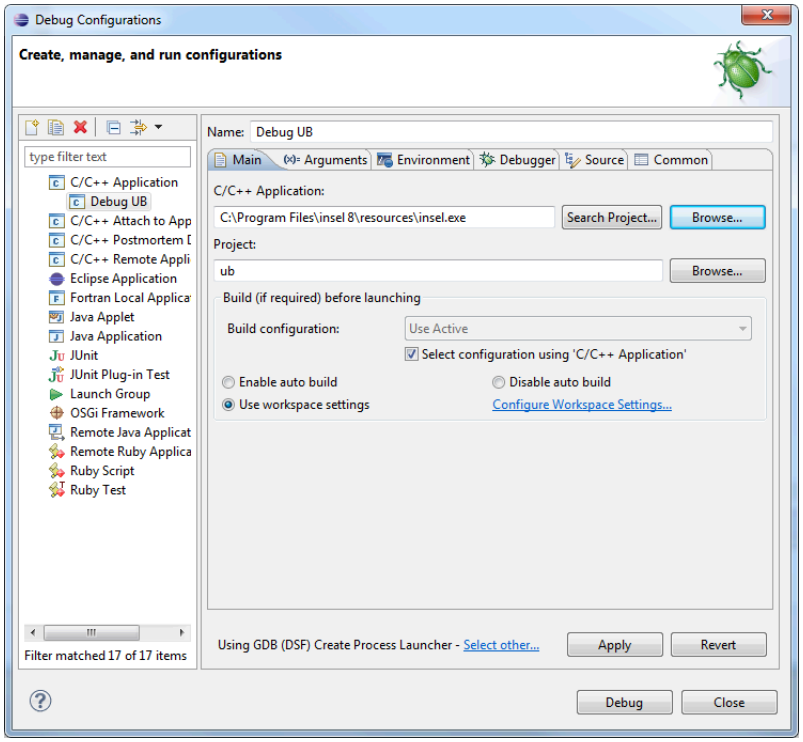
GDB debugger

The CDT plugin of Eclipse uses GDB, the GNU project debugger to translate user interface actions into GDB commands in the background. Let us have a look at GDB at work, using a practical, but simple example and use the libInselUB library with the two sample blocks UBCPP and UBFOR.

Before Eclipse can start a program in debug mode a debug configuration is required.



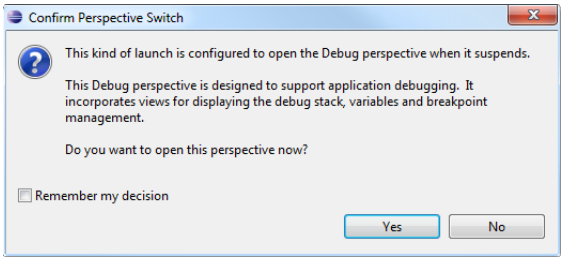
A new debug configuration can be created from the tool bar's Debug pull-down menu Debug Configurations... which opens the following dialog:



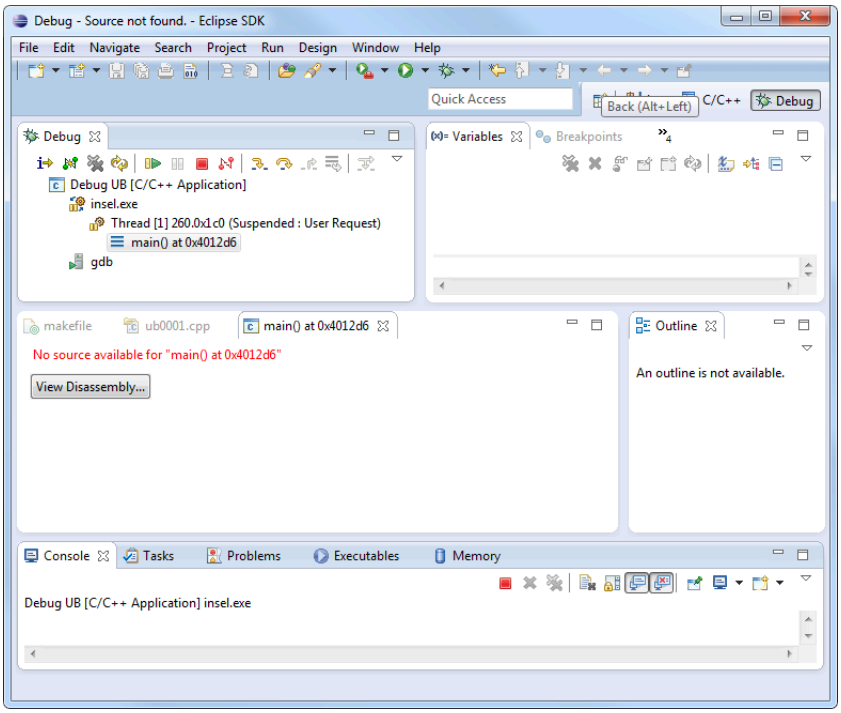
Select C/C++ Application in the tree at the left edge of the window and use the New button in the upper left corner to create a new configuration. Specify a name for the configuration, e. g., Debug UB and browse to the project directory you wish to use for debugging, e. g., ub.

Since `libInselUB` is a dynamic library it cannot directly be executed but must be wrapped with an executable. In INSEL two candidates are available: `insel_8.exe` of the installation directory and `insel.exe` of the `resources` directory. The first one starts the graphical interface of INSEL, while `insel.exe` starts the inselEngine in a terminal. As a start, let us choose the second option and browse to the file with the C/C++ application.

Click the Debug button to save your changes and to start the debugger immediately. By default, Eclipse starts to rebuild the project files and compiles all sources. Next, Eclipse suggests to switch to the Debug perspective.



The Debug perspective is shown in the next figure:





Several views become visible, the most important being the Debug view with several buttons to control execution. The red square button is used to terminate debugging, the two yellow arrows step into and step over a statement, respectively – please check out the buttons tool tips for further information. When debugging starts, the debugger "stands" on the main() statement of `insel.cpp`

```
int main (int argc, char* argv[])
{
```

awaiting instructions to step into or step over the main function, for example. Since INSEL users do not have access to the INSEL source code Eclipse displays some

warnings in the windows' title bar and in the text editor view. A click on the View Disassemby... button shows some assembler code, like

```
004012d3:   sub $0x28,%esp
004012d6:   and $0xfffffff0,%esp
004012d9:   mov $0x0,%eax
004012de:   add $0xf,%eax
004012e1:   add $0xf,%eax
004012e4:   shr $0x4,%eax
```

which is not of much use for most of us. So let's step into the main function and see what happens. This is the console output:

```
(no debugging symbols found)
No source file named ubcpp.cpp in loaded symbols.
No source file named ubcpp.cpp in loaded symbols.
[New thread 3060.0xe8c]
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
Single stepping until exit from function main,
which has no line number information.
```

Not too interesting. A second click on the Step Into button brings us to the end of the main function and ends the debugger. `insel.exe` displays some text and informs us about its usage and that a filename is missing:

```
This is insel 8.3 (c) 1986-2017 doppelintegral GmbH
Missing filename
Usage: insel filename [options]
   filename
      Any .insel model
   [options]
      -d  Debug mode
      -j  insel called from Java
      -l  Show calculation list
      -m  Show .insel file
      -s  Syntax check only
```

`cpp.insel` Now, you are going to write your first real INSEL model – without the help of VSEit. Enter the following lines in a text editor

```
s 1 const
p 1 17
s 2 ubcpp 1
p 2 3.14
s 3 screen 1 2
```

and save the file as `cpp.insel` for example. The name is arbitrary, but the file extension has to be `.insel`. What does the text mean?

`.insel` syntax

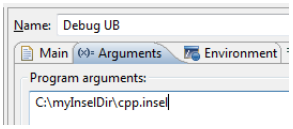
As an experienced INSEL user you see three block names: CONST, UBCPP, and SCREEN – not case sensitive – and three (arbitrary but unique) user-block numbers 1, 2, 3 for the three blocks. The blocks are defined through a leading s which is short for structure.

A one follows the name of the UBCPP block which means that the UBCPP block uses the first output of block number one as an input. The one and two following the name of the SCREEN block mean that the SCREEN block gets inputs from block number one (the CONST block) and from block number two (the UBCPP block).

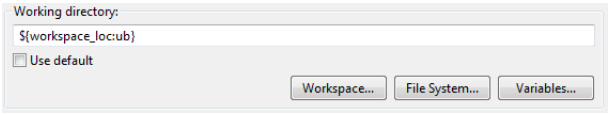
Parameters are assigned to two blocks via p statements (short for parameter). The connection between the block and the parameter values is associated through the unique user-block number.

Program arguments

Connect the new INSEL model `cpp.insel` with the debug configuration of your user block project. To do so, open the debug configuration click the Arguments tab and enter the full path to `cpp.insel` in the Program arguments: text field.



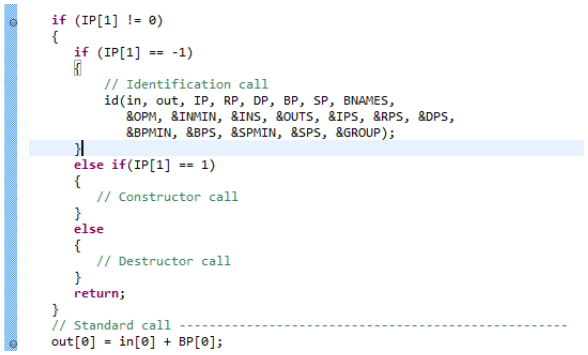
In addition you must specify the path to the INSEL libraries in the same pane.



Do do so, uncheck the Use default checkbox and use the File System... button to browse to the `resources` or `Contents` directory, respectively. Then click the Apply button and close the window.

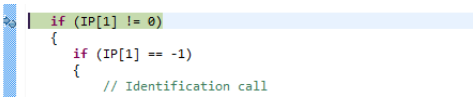
Breakpoints

Our next aim is to run this INSEL model in debug mode and to pause execution in the code of the UBCPP block. The source code of this block is available in file `ubcpp.cpp` of the `\insel.work\inselUB\src` directory located in your home directory. Please, open `ubcpp.cpp` now in the Debug perspective. Pausing execution is caused by so-called breakpoints. Breakpoints can be added and/or removed by a double-click in the left margin of a source code line, visually indicated by a small blue bullet:



The screenshot shows two breakpoints in `ubcpp.cpp`, one at the if statement which checks for non-Standard calls, and one breakpoint at the statement which sets the first output as sum of first input and first block parameter – in C notation as `outs[0]` etc.

When you run the debug configuration now and click the step-into arrow, the debugger does not run through the complete program but pauses at the first breakpoint that you've set at the if statement, indicated by highlighting the code line.



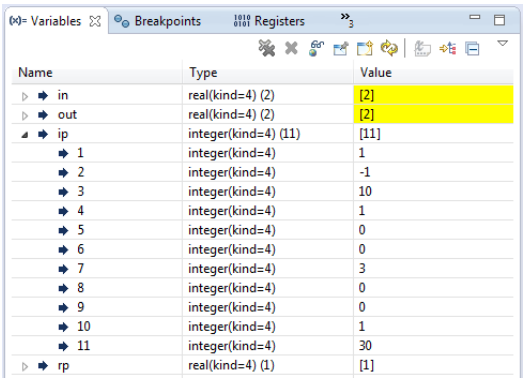
Now you are free to wander through the code while it is executed – step into statements, step out of statements, and so on. When you remember, how INSEL blocks work, the first stop at `if (ctrl[insel::CALLMODE] != insel::STANDARD)` should result in an identification call. You can observe this now by stepping into the if statement.

Program flow

In conclusion, the first huge advantage of using a debugger is that you can observe how your code "really" executes – sometimes you will see that there is a big difference compared to what you "thought" how your code executes.

Current values of variables

The second advantage is that you can observe the current values of all variables your code uses at any time. The Variables view shows all relevant variables in the current program status. Have a look at the status of the variables at the very first breakpoint stop in `ubcpp.cpp`:



Remember, at this early stage of executing `cpp.insel` inselEngine calls block CPP (and all other blocks in the model) in Identification Call, indicated by `ctrl[insel::CALLMODE]` = `insel::IDENTIFY` (−1). GDB shows the plain index, i. e., `ctrl[1]`.

The pointers `ins`, `outs`, `ints`, `reals`, and `params` are null in this call. The inselEngine allocates these arrays with the sizes reported by `insel_identify` after all identification calls have been completed.

One dimension too much

The Fortran template declares its arrays with one element more than needed, e. g., `INTS(N_INTS+1)`, in order to avoid zero-sized arrays.

DO NEVER ACCESS THE EXCESS BLOCK VARIABLES BECAUSE THEIR CONTENT IS UNDEFINED.

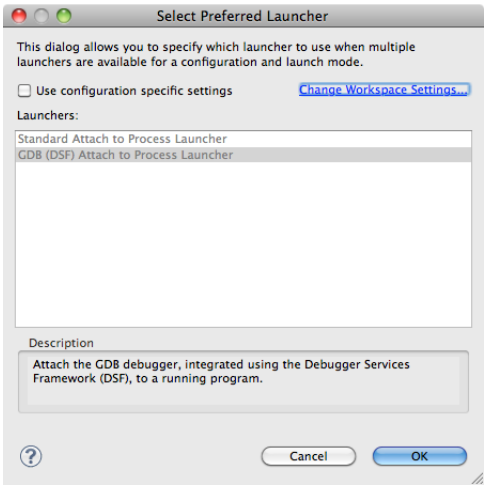
Further reading

This remark on quick-and-dirty INSEL programming ends our short excursion to debugging user-block libraries with Eclipse. More information about CDT debugging in Eclipse can be found in the online documentation and plenty of books about the topic.

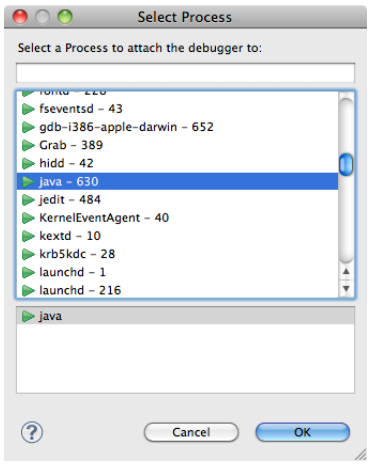
A hint for Mac Users

As seen on page 305 the creation of a debug configuration requires an executable like `insel.exe`, for example. On a Mac computer INSEL is installed as an application bundle named `insel 8.app` but application bundles are not accepted as executables in debug configurations.

A way out of this dilemma is to create a C/C++ Attach to Application debug configuration and connect it to the GDB (DSF) Process Launcher which can be selected in the Debug Configuration window.



Start `insel 8.app` and run the debug configuration of your library project in Eclipse. The *Select Process* dialog opens.



Choose the *java* process and you're done. Happy debugging.

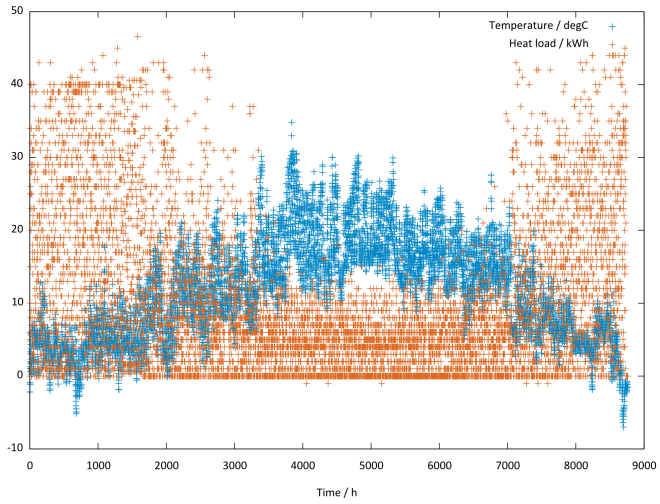
PART IV :: Workshops

14 :: PV Heat Pump Storage System

In this workshop you will be guided step-by-step in the construction of a template for the simulation of a simple system based on PV, a heat pump, and a thermal storage. A given annual heat demand profile in hourly resolution will be used as input. The task of the template is, to provide an easy method for the calculation of the component's performance and the required backup heat from additional devices. The level of autonomy will be quantified as solar fraction, i. e., the percentage of energy self-sufficiency.

Load profile

In the `data` folder of this workshop's root folder you can find a file named `ws1_load_profile.dat`. This file contains two values per record, i. e., the ambient temperature / °C and the heat demand / kWh. The total number of records is 8,760, i. e., one record for each hour of the year. The plotted data are as follows:



Exercise 14.1 Reproduce the plot above and calculate the monthly means of ambient temperature and heat demand. as stored in file `ws1_load_profile.dat`.

```
 1.    3.32   17.10
 2.    4.86   16.79
 3.    7.10   13.33
 4.   11.73    8.65
 5.   13.78    6.55
 6.   19.21    4.70
 7.   18.88    4.19
 8.   17.81    3.92
 9.   15.66    4.53
10.   13.04    7.44
11.    7.51   12.59
12.    3.67   16.72
```

Heat pumps

Heat pumps use electric power and a usually relatively low-temperature heat source as input. Most heat pumps are either air(A) or water(W) based on the input side as well as on the heat output side (IO). Hence, the four main types of heat pumps are (i) WW, (ii) WA, (iii) AW, or (iv) AA.

In this workshop, we are going to use the WPL 18 E heat pump of the German manufacturer Stiebel Eltron as a typical example. WPL 18 E is an air-in/water-out (AW) heat pump.

Heat is extracted from the outside air via the air-side heat exchanger (evaporator). The refrigerant evaporates and is compressed by a compressor. This requires electrical energy. The refrigerant is now at a higher temperature level and releases the heat from the air via another heat exchanger (condenser) to the heating system. Then the refrigerant relaxes and the process starts again. At air temperatures below approximately $+7\ ^\circ\mathrm{C}$, the humidity is reflected as frost on the evaporator fins. This ripening is automatically defrosted. The resulting water is collected in the defrosting trough and drained via a hose. In the defrosting phase, the fan switches off and the heat pump cycle is reversed. The heat needed for the defrost is removed from the buffer. At the end of the defrosting phase, the heat pump automatically switches back to heating mode.

How can a heat pump be characterized? A typical ratio between electric input and thermal output COP (coefficient of performance) is about 3, i. e., for one kWh of electrical input, the heat output is three kWh. This corresponds approximately to the efficiency of electric power plants, which is about 0.33.

Technical data sheets of heat pumps usually provide the electrical power demand and the COP as a function of the temperature of the medium *entering* the heat pump, i. e., ambient air temperature T_a in our case. What can be a little confusing is that, the temperature of the medium *leaving* the heat pump is called *supply* temperature. For the time being, let us agree to use the term *inlet temperature* T_in for the temperature of the air entering the heat pump and *outlet temperature* T_out for the temperature of the water exiting the heat pump (and supplying some heat load demands).

The level-of-detail at which manufacturers provide technical data about their heat pumps varies a lot. In case of WPL 18 E, Stiebel Eltron provides a printed heating power diagram, showing the electrical demand, the COPs, and the resulting heating power as functions of the ambient temperature, i. e., the *inlet* temperature, using the *outlet temperature* as a curve parameter. The values have been manually transfered from the power diagram to numerical data and these are summarized in the follwing table:

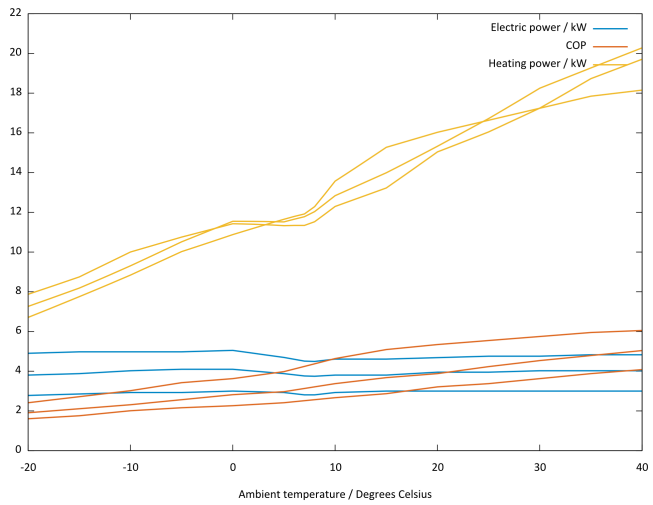
```
Pel / kW                        COPs
      ---- Tout / degC ----            ---- Tout / degC ----
Tin     35     50     60        Tin     35     50     60
-20    2.780  3.805  4.902      -20    2.414  1.909  1.606
-15    2.854  3.878  4.976      -15    2.717  2.111  1.758
-10    2.927  4.024  4.976      -10    3.020  2.313  2.010
 -5    2.927  4.098  4.976       -5    3.424  2.566  2.161
  0    3.000  4.098  5.049        0    3.626  2.818  2.263
  5    2.927  3.878  4.693        5    3.980  2.970  2.414
  7.5  2.780  3.732  4.463       10    4.636  3.374  2.667
 10    2.927  3.805  4.610       15    5.091  3.677  2.869
 15    3.000  3.805  4.610       20    5.343  3.879  3.212
 20    3.000  3.951  4.683       25    5.545  4.232  3.374
 25    3.000  3.951  4.756       30    5.747  4.535  3.626
 30    3.000  4.024  4.756       35    5.949  4.789  3.879
 35    3.000  4.024  4.829       40    6.051  5.040  4.081
 40    3.000  4.024  4.829
```

Please remember, that the relation between electrical power input P_{el} and heat power output P_{heat} is given by

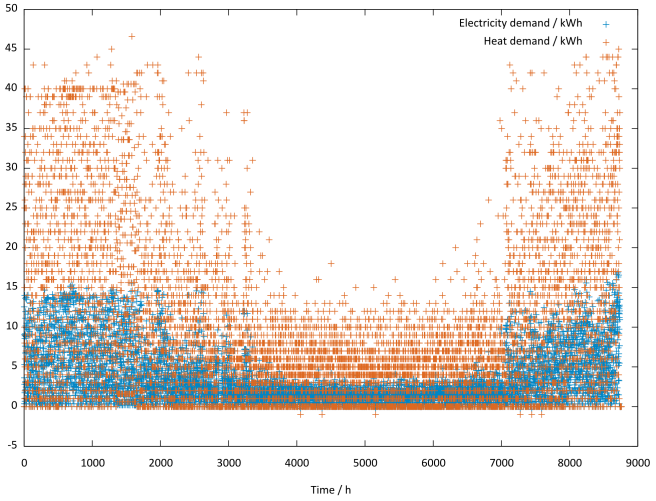
$$P_{\text{heat}} = \text{COP} \cdot P_{\text{el}}$$

Exercise 14.2 Plot the table data with the electrical power, COP, and heating power as functions of ambient temperature.

Hint: Use two nested DO blocks and two POLYG2 blocks with the outlet temperatures as curve parameter.

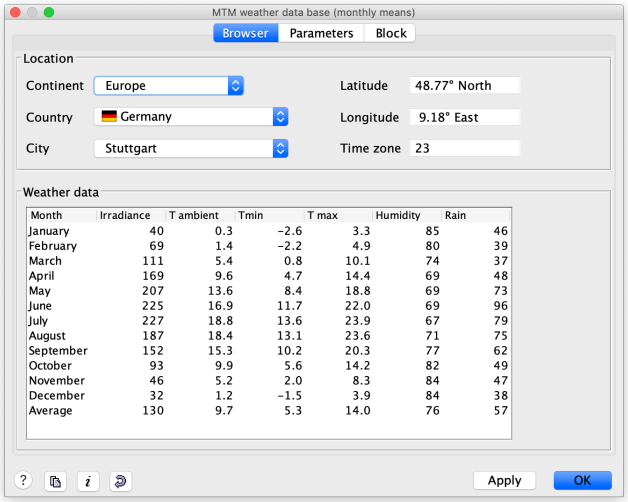


Exercise 14.3 Now that we know the hourly ambient temperature, the heat demand, and "our" heat pump characteristics, plot the time series of the heat and electricity demand, and calculate the peak heat and electricity demand, and the annual average COP.




```
Pq,max = 46.60 kWh  Pel,max = 16.83 kWh  COP,ave = 2.77
```

PV The heat demand data stored in file `ws1_load_profile.dat` are measured data from a multi-family building near Stuttgart, Germany. The long-term average irradiance data for Stuttgart can be found in the MTM data base of INSEL.



Month	Irradiance	T ambient	Tmin	T max	Humidity	Rain
January	40	0.3	-2.6	3.3	85	46
February	69	1.4	-2.2	4.9	80	39
March	111	5.4	0.8	10.1	74	37
April	169	9.6	4.7	14.4	69	48
May	207	13.6	8.4	18.8	69	73
June	225	16.9	11.7	22.0	69	96
July	227	18.8	13.6	23.9	67	79
August	187	18.4	13.1	23.6	71	75
September	152	15.3	10.2	20.3	77	62
October	93	9.9	5.6	14.2	82	49
November	46	5.2	2.0	8.3	84	47
December	32	1.2	-1.5	3.9	84	38
Average	130	9.7	5.3	14.0	76	57

Exercise 14.4 Assuming a PV-module efficiency of 15 per cent on average, calculate the heat demand coverage as a function of the installed PV peak power.

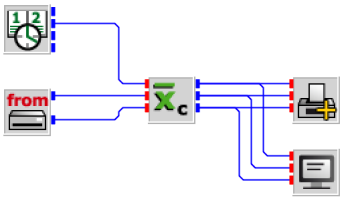
Summary

- You have learnt ..
- Some typical examples ..

Solutions

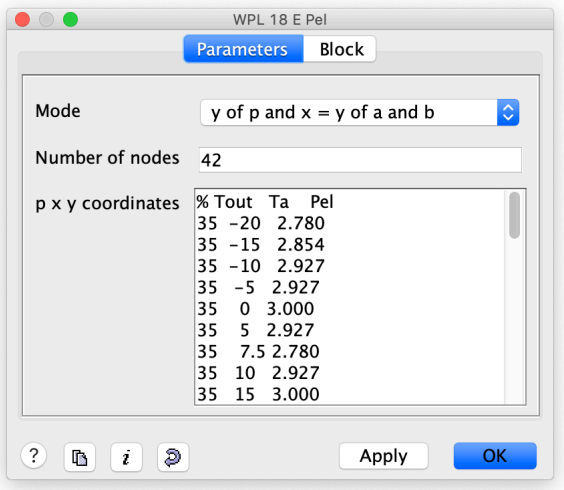
Exercise 14.1 The task has been to calculate monthly means of ambient temperature and heat demand in a given file. A CLOCK block with time step one hour is a conventient way to handle the fact that different months have a different number of days. A READ block to read the data can be used. An AVEC block can calculate the conditional averages.

Finally, a SCREEN block can display the results in tabular form. The format used in the text was (F8.0,2F8.2).



Exercise 14.2 The task was basically to plot table data from a given table. The first problem that arises is, how to transfer the data to INSEL blocks? The suggested solution is, to use two POLYG2 (p,x,y) blocks, one for the electric power and one for the COPs. The parameter p is the outlet temperature, x should be the ambient temperature, and y the electric power and the COPs, respectively.

The POLYG2 entity editor for the electric power should look like this:



One way to get there is, to copy and paste the table from 314 into your favorite text editor, rearrange the data and then copy/paste the formatted data into the two POLYG2 blocks. Please remember that entity editors accept only keyboard shortcuts, i. e., `ctrl c` / `ctrl v` on Windows and `cmd c` / `cmd v` on macOS.

Since the task was to plot three curve sets with three graphs per set as a function of ambient temperature, two DO blocks can be used. One for counting the three curve sets, i. e., initial value 1, final value 3, increment one, and the second one for the variation of the ambient temperature, i. e., initial value -20, final value 40, increment one, for example.

The DO block is a Timer block (T-block). INSEL accepts an arbitrary number of T-blocks in one model, but only one, unique main timer. If the two DO blocks would be in the model without any dependencies on other blocks, i. e., both DO blocks without input, the inselEngine could not determine which of them is meant as main timer, they would be equitable or flat, non-hierarchical.

If we were to plot the three curves for a fixed outlet temperature, we would vary the ambient temperature and plot the three graphs. Consider this as an INSEL model that you coud put in a macro, for instance. Now, we want to execute this “macro” three times. Ergo, we would use a DO block and connect its output as an input to the macro. We

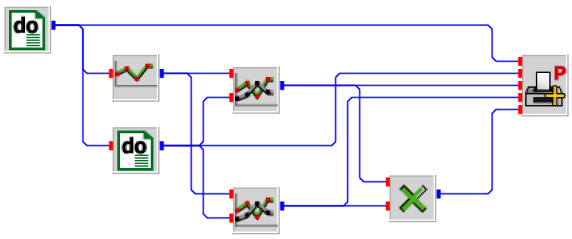
could do this by adding an input to the ambient-temperature DO block and connect the "outer" DO block to this input. Now it is unique, the 1-2-3 DO block is the main timer, and the second DO block is what we call a sub-timer.

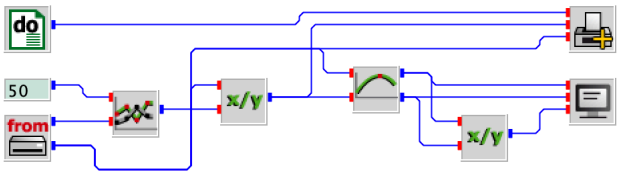
The last problem is, that we do not have 1, 2, 3 as outlet temperature, but 35, 50, and 60 (which cannot be parametrized by the DO block). The trick is, to use a POLYG block with three nodes, use

```
1   35
2   50
3   60
```

as parameters and connect the input to the 1-2-3-DO block.

The rest should be obvious now. A MUL block multiplies the electric power by the COP to get the heat power. A parametric PLOTP is used to finally plot the desired result with the outlet temperature as curve parameter, the ambient temperature as x-coordinate, and the three y inputs for the electrical power, the COPs, and finally the heating power.




do
50
from
x/y
x/y

15 :: TRNSYS Restaurant

The idea of this workshop is to provide a step-by-step introduction into the basic ideas of building simulation in INSEL. So far, there are three different approaches to building simulation in INSEL 8: (i) a very simple model based on a German Standard (DIN 18599) for monthly heating and cooling demand calculations (INSEL block D18599), (ii) a R-C model based simple dynamic model (INSEL block FDBS), and (iii) a as-much-as-possible dynamic simulation model, based on blocks for internal and external walls (WALL and WALLX), radiation exchange (RADI), zone temperature calculations (TROOM), etc.

In a future release of INSEL (version 9.0) it is planned to have a block (EPLUS) which allows to incorporate Energy Plus models and exchange data between Energy Plus simulations and INSEL models.

So, what is the TRNSYS Restaurant?

The TRNSYS Restaurant is a virtual building, which is used (in the TRNSYS documentation and software distribution) as a "generic" example for the dynamic simulation of a building in TRNSYS. It has three zones, a well-defined occupancy schedule and all details of "building physics", etc. And, as TRNSYS is one of the most respected building simulation softwares woldwide, the TRNSYS Resuatrant can serve as a "blue print" of a building simulation example.

Construction and schedules

The TRNSYS Restaurant consists of a dining room, a kitchen, and a storage area. The floor plan of the restaurant is shown in Figure 15.1.

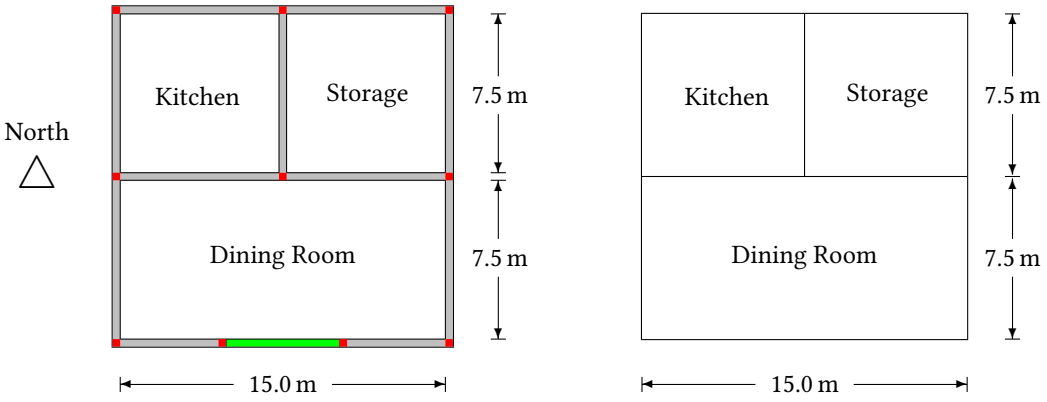


Figure 15.1: Floor plan with window (green), heat bridges (red) and lattice model of the TRNSYS Restaurant.

As can be seen from the figure, the dining room faces directly south. It has a large double-glazed window with a total area of 10 m^2. The exact geometry of the window is

not defined in the original example. Let us assume that the window is placed in the center of the south wall with a width of ten meters and a height of one meter. The dimensions of the three rooms as used in the TRNSYS model are shown in Table 15.1.

Zone	Name	W–E / m	N–S / m	Area / m^2	Surface / m^2	Volume / m^3
1	Dining Room	15.0	7.5	112.5	360.0	337.5
2	Kitchen	7.5	7.5	56.25	202.5	168.75
3	Storage	7.5	7.5	56.25	202.5	168.75

Table 15.1: Dimensions of the building. The height of all rooms is 3.0 meters.

It can be noticed that the geometry is not hundred percent correct, i. e., the fact, that the area of the dining room is exactly the sum of the areas of the kitchen and storage shows that the thickness of the internal wall between kitchen and storage is neglected in the geometric model. The same applies to the heat bridges in the corners.

Constructions

The building consists of two types of walls (outside and inside), the floor, and a flat roof. The details of their constructions are compiled in Table 15.2.

Name	Layer	Thickness m	Conductivity $W\,m^{-1}\,K^{-1}$	Density $kg\,m^{-3}$	Spec. heat $J\,kg^{-1}\,K^{-1}$	U-value $W\,m^{-2}\,K^{-1}$
Outside	Gypsum	0.019	0.728	1,601	750	
	Insulation	0.076	0.0431	32	750	0.501
	Stucco	0.025	0.692	1,858	750	
Inside	Gypsum	0.019	0.728	1,601	750	
	Wood	0.058	0.0116	592	2,250	0.191
	Gypsum	0.019	0.728	1,601	750	
Floor	Stone	0.025	1.436	881	1,500	
	Insulation	0.076	0.0431	32	750	0.498
	Concrete	0.102	1.731	2,242	750	
Roof	Plastboard	0.016	0.528	1,200	840	
	Airspace	0	n.a.	n.a.	n.a.	
	Insulation	0.076	0.0431	32	750	0.452
	Concrete	0.102	1.731	2,242	750	
	Roofing	0.006	0.694	2,100	1,000	

Table 15.2: Layer structures and thermophysical properties of walls, floor, and roof (all inside to outside). TRNSYS simulates the airspace of the roof as a thermal resistance of 0.18 $m^2\,K\,W^{-1}$. The U-values are calculated with the EN ISO 6946 values for the heat resistances, i. e., $R_i = 0.13$ and $R_o = 0.04\,m^2\,K\,W^{-1}$.

The heat transfer coefficient at the outside of the exterior walls and roof is assumed to vary with the wind speed. The heat transfer coefficient of the floor is set to a very small value ($10^{-5}\,kJ\,h^{-1}\,m^{-1}\,K^{-1} = 0.278 \times 10^{-5}\,W\,m^{-1}\,K^{-1}$) which imposes the surface temperature to be equal to the ground temperature.

Schedules

By definition, the building has a people-occupancy schedule: It is assumed that the restaurant has an occupancy from 7 a.m. to 10 p.m. every day. The number of people in the building varies as given in Table 15.3.

Time	Weekdays	Weekends
0 – 8	0	0
8 – 10	5	10
10 – 12	2	5
12 – 14	10	10
14 – 17	2	4
17 – 22	10	10
22 – 24	0	0

Table 15.3: Daily number of people in the restaurant.

Gains

The model assumes that each occupant of the building delivers a convective heat gain of $150\ \mathrm{kJ\,h^{-1}} = 41.7$ W, a radiative heat gain of $70\ \mathrm{kJ\,h^{-1}} = 19.5$ W, and an absolute humidity of $0.058\ \mathrm{kg\,h^{-1}}$. Gains from people are assumed for both, dining room (scale factor: five times number of customers) and kitchen (0.5 times number of customers).

Other gains come from the lights with $300\ \mathrm{kJ\,h^{-1}} = 83.47$ W convective and $1{,}500\ \mathrm{kJ\,h^{-1}} = 417$ W radiative, respectively (no humidity). The lights are on whenever the building is occupied, i. e., from 7 a.m. to 10 p.m. Their power is scaled by a factor two for the dining room and one for the kitchen.

The kitchen has also gains associated with the stoves, i. e., $10{,}000\ \mathrm{kJ\,h^{-1}}$ = 2,778 W convective and $5{,}000\ \mathrm{kJ\,h^{-1}}$ = 1,389 W radiative, and a humidity of $0.1\ \mathrm{kg\,h^{-1}}$, respectively. The stoves are on during occupancy time, i. e., from 7 a.m. to 10 p.m.

The storage room has fixed gains from a freezer with only convective heat ejection of $1{,}500\ \mathrm{kJ\,h^{-1}} = 417$ W, running 24 hours every day.

Air flows

The infiltration rate is fixed at half an air change per hour. An additional infiltration of the dining room is given as 0.03 times the number of customers. The kitchen is ventilated with ambient air from 7 a.m. to 10 p.m. at a rate of 0.75 air changes per hour.

Heating and cooling

The dining room and kitchen are maintained at 20 °C during occupied hours and at 15 °C other times. The maximum power of the kitchen and dining room heaters is $50{,}000\ \mathrm{kJ\,h^{-1}} = 13.9$ kW each. It is assumed that the heat provided is purely convective, i. e., no radiative and no humidification gains. The storage area is unheated.

A cooling unit with a nominal power of $50{,}000\ \mathrm{kJ\,h^{-1}} = 13.9$ kW is located in the kitchen. It turns on if the temperature rises above 26 °C. Dining room and storage don't have cooling units installed. TRNSYS Type 56 assumes that all cooling power is purely convective but can have a dehumidification fraction. In case of the TRNSYS Restaurant, dehumidification is switched off.

The initial conditions for all three zones are 20 °C air temperature and 50 percent relative humidity.

Exercise 15.1 Calculate the heating and cooling demand (assuming that the building is located in

Stuttgart, Germany) on the basis of the German DIN 18599 Standard, implemented in the INSEL block D18599.

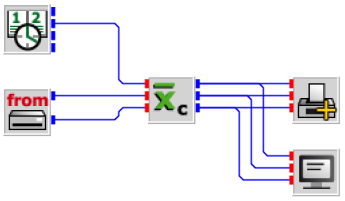
Summary

- You have learnt ..
- Some typical examples ..

Solutions

Exercise 15.1 The task has been to calculate monthly means of ambient temperature and heat demand in a given file. A CLOCK block with time step one hour is a conventient way to handle the fact that different months have a different number of days. A READ block to read the data can be used. An AVEC block can calculate the conditional averages.

Finally, a SCREEN block can display the results in tabular form. The format used in the text was (F8.0,2F8.2).



Résumé

One way or another – whether you worked your way through the complete Tutorial or have just peeped into some of the Modules, we hope that you could benefit from the Modules you have studied.

INSEL offers several levels of detail. It is not necessary to go all the way down to the deepest programming techniques if you wish to just want to find an answer to one of your actual problems. Nevertheless, we hope that youfound some interesting solutions in this Tutorial.



A Appendix

A.1 Directory structure, dependencies, and paths

INSEL 8 makes use of the following directories

- Installation directory
- Path directory
- Hidden application data directory
- Working directory

Installation directory

The main directory of INSEL is the *installation directory*, i. e., the directory where INSEL is installed. INSEL 8.3 is 64-bit only; the default names and paths are

- `C:\Program Files\INSEL_8.3`
  for Windows
- `/Applications/INSEL.app`
  for macOS
- `/opt/INSEL`
  for Linux

However, the user may wish to install into a different directory – which is possible in all versions. The question then is "How can other applications find out, if and where INSEL is installed on a computer?"

Path directory

INSEL installs a file named `inselroot.ini` in the *Path* directory. The file has two records only: a *section name* `[InstallDir]` and a *key* `inselroot` containing the path to the INSEL installation, e. g.,

```
[InstallDir]
inselroot=C:\Program Files\INSEL_8.3
```

The location where `inselroot.ini` resides depends on the operating system used.

- `C:\ProgramData\INSEL`
  for Windows
- `/opt/INSEL`
  for macOS and Linux

The `inselTools` library provides a method to retrieve the installation directory. The C++ interface of this function is

```
extern "C" void inselroot(char* path, int* iLen, int* irc);
```

The Fortran call

```
CALL INSELROOT(PATH,ILEN,IRC)
```

returns the path to the installation directory in `PATH`, which is a `CHARACTER*1024` variable, `ILEN` and `IRC` are of type `INTEGER` containing the length of the path string (without backslash zero) and the return code, i. e., zero when everything is okay.

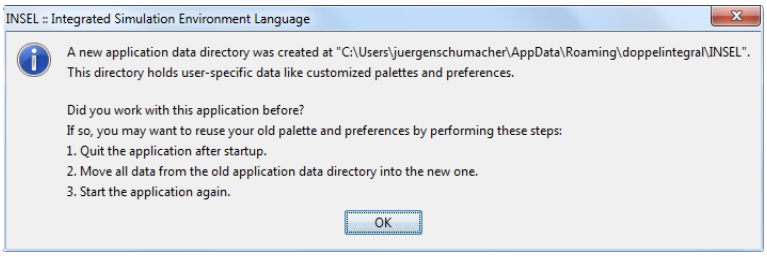
Hidden application data directory

INSEL writes user-specific data and temporary files to the *hidden application data directory*. Amongst other files, this directory contains a user-specific version of the palette, custom types (if any), and the user-specific preferences in a file named `Preferences.vseit`.

The location of the hidden application data directory is operating-system dependent.

- `C:\Users\USERNAME\AppData\Roaming\INSEL`
  for Windows
- `/Users/USERNAME/.insel`
  for macOS and Linux

Location and name cannot be configured, except within the options to redirect the user's home directory as such. When the directory does not exist for a new INSEL user it is created automatically and hint similar to the following is displayed.



The `inselTools` library provides a method to retrieve the hidden application data directory. The C++ interface of this function is

```
extern "C" void getinseldir(int* I, char* path, int* iLen, int* irc);
```

A Fortran call with `I = 2`

```
CALL GETINSELDIR(I,PATH,ILEN,IRC)
```

returns the path to the hidden application data directory in `PATH`, which is a `CHARACTER*1024` variable, `I`, `ILEN` and `IRC` are of type `INTEGER`.

The other variables return the length of the path string (without backslash zero) and the return code, i. e., zero when everything is okay.

Working directory

Every INSEL user has an own working directory, named `insel.work`, by default.

The location of the working directory is operating-system dependent.

- `C:\Users\USERNAME\Documents\insel.work`
  for Windows
- `/Users/USERNAME/Documents/insel.work`
  for macOS and Linux

It is possible to redirect the working directory via the *File* > *Preferences* dialog (Windows) or the *INSEL* > *Preferences* dialog (Mac OS), respectively. Linux? The path can be specified absolute or relative to the user's documents directory.

Subdirectories

User-block programming is supported in a subdirectory of the working directory, named `inselUB` – the name and the substructure of the `inselUB` directory is fixed.

Temporary files and files derived from user-block programming like Java `.class` files are maintained in the `tmp` and `customTypes` subdirectories of the hidden application data directory.

`getinseldir`

This is a list of full-path directory names which can be retrieved via `getinseldir` when the index `I` is set correspondingly:

1 Working directory

2 Hidden application data directory

3 Custom types directory

4 Temporary files directory

5 INSEL model directory

6 User block support directory

RedirectStdErrToErrorLog=false fuer Eclipse Entwicklung

RedirectStdErrToErrorLog=true fuer Auslieferung

A.1.1 File Handling

In insel 8 all internal files get a fixed Fortran unit number. User unit numbers should start at 50.

Wieder auf GETUNI umstellen!!!

10 insel.msg
11 report.tmp
12 FREI (ex: mtmup input file)
13 BLOCKDOC TEMPORAER (ex: sedes spetrabs.dat)
14 FREI (ex: sedes ccmneu.dat)
15 FREI (ex: inselWeather.ind)
16 FREI (ex: inselWeather.dat)
17 FREI (ex: inselWeather.loc)
18 TSOIL
19 insel.gpl
20 insel.gnu
21 pvibp-file
22 pvdet*-file
23 inselroot.ini

c	Hex	Oct	Char		Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	000	NUL	(null)	32	20	040	Space	64	40	100	@	96	60	140	‘
1	1	001	SOH	(start of heading)	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	(start of text)	34	22	042	”	66	42	102	B	98	62	142	b
3	3	003	ETX	(end of text)	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	(end of transmission)	36	24	044	&	68	44	104	D	100	64	144	d
5	5	005	ENQ	(enquiry)	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	(acknowledge)	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	(bell)	39	27	047	’	71	47	107	G	103	67	147	g
8	8	010	BS	(backspace)	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	73	49	111	I	105	69	151	i
0	A	012	LF	(line feed)	42	2A	052	*	74	4A	112	J	106	6A	152	j
1	B	013	VT	(vertical tab)	43	2B	053	+	75	4B	113	K	107	6B	153	k
2	C	014	FF	(form feed)	44	2C	054	,	76	4C	114	L	108	6C	154	l
3	D	015	CR	(carriage return)	45	2D	055	-	77	4D	115	M	109	6D	155	m
4	E	016	SO	(shift out)	46	2E	056	.	78	4E	116	N	110	6E	156	n
5	F	017	SI	(shift in)	47	2F	057	/	79	4F	117	O	111	6F	157	o
6	10	020	DLE	(data link escape)	48	30	060	0	80	50	120	P	112	70	160	p
7	11	021	DC1	(device control 1)	49	31	061	1	81	51	121	Q	113	71	161	q
8	12	022	DC2	(device control 1)	50	32	062	2	82	52	122	R	114	72	162	r
9	13	023	DC3	(device control 1)	51	33	063	3	83	53	123	S	115	73	163	s
0	14	024	DC4	(device control 1)	52	34	064	4	84	54	124	T	116	74	164	t
1	15	025	NAK	(negative acknowledge)	53	35	065	5	85	55	125	U	117	75	165	u
2	16	026	SYN	(synchronous idle)	54	36	066	6	86	56	126	V	118	76	166	v
3	17	027	ETB	(end of trans. block)	55	37	067	7	87	57	127	W	119	77	167	w
4	18	030	CAN	(cancel)	56	38	070	8	88	58	130	X	120	78	170	x
5	19	031	EM	(end of medium)	57	39	071	9	89	59	131	Y	121	79	171	y
6	1A	032	SUB	(substitute)	58	3A	072	:	90	5A	132	Z	122	7A	172	z
7	1B	033	ESC	(escape)	59	3B	073	;	91	5B	133	[	123	7B	173	{
8	1C	034	FS	(file separator)	60	3C	074	<	92	5C	134	\	124	7C	174	|
9	1D	035	GS	(group separator)	61	3D	075	=	93	5D	135	]	125	7D	175	)
0	1E	036	RS	(record separator)	62	3E	076	>	94	5E	136	^	126	7E	176	~
1	1F	037	US	(unit separator)	63	3F	077	?	95	5F	137	_	127	7F	177	DEL