

# IPAT-S Language Reference

Eric Kemp-Benedict

June 2005

© 2005 Eric Kemp-Benedict

The *IPAT-S Language Reference* may be freely reprinted photocopied, distributed, copied electronically and posted online.

This document was created using OpenOffice.org Writer. Body text is set in Palatino Linotype, and titles are in Trebuchet MS. Source code is set in ProggyFonts (ProggyCleanTT, ProggySmallTT and ProggyTinyTT).

For questions or comments, please contact the author:

*e-mail*

eric@kb-creative.net

*Mail*

Eric Kemp-Benedict  
8 Longfellow Road  
Cambridge, MA 02138  
USA

## Table of Contents

1 Introduction.....	1
Scenarios for Sustainable Development.....	1
Intended Audience.....	2
Features and Limitations.....	2
License Information.....	3
2 Overview.....	4
Overall Structure of an IPAT-S Script.....	4
Base year & scenario years.....	4
Dimensions.....	4
Body.....	5
Running the IPAT-S Interpreter.....	5
Executing IPAT-S from Other Programs.....	7
3 Data Types.....	8
Numerical Values.....	8
Logical Values.....	8
String Values.....	8
Data Types With Dimensions.....	8
Numbers With Dimensions.....	9
Automatic Dimension Matching.....	9
Unmatched Dimensions.....	10
Summable Variables.....	11
Variables.....	13
Ratio Variables.....	13
Number Variables.....	14
Logical Variables.....	14
4 Calculations.....	16
Chain, ::, and Related Expressions.....	16
Calculating Sums and Averages.....	18
Index Expressions.....	18
Numerical Expressions.....	20
Logical Expressions.....	20
.and, .or, .not.....	20
.neither. ... .nor.....	20
.gt, .lt, .eq, .neq, .ge, .le.....	21
String Concatenation.....	21
Functions.....	21
Procedures.....	23
The Standard Library.....	24
The Interpolate Procedure.....	25
Generating Random Numbers.....	25
The Loglet Procedure.....	26
Linear Programming – LP blocks.....	26

Introduction to Linear Programming in IPAT-S.....	27
Introduction to Linear Goal Programming.....	30
A side-note on “numbers with dimensions”.....	32
Using Weights in Linear Goal Programming.....	33
Beyond Non-Preemptive Minimax Linear Goal Programming.....	34
Implementing an I-O Problem Using an LP.....	38
Constraining Changes in Solution Variables.....	42
Getting around the limitations of LPs in IPAT-S.....	42
Return values from LPs.....	44
5 Output.....	45
Report Statements.....	45
Print blocks.....	45
Basic format of a print block.....	45
In-line calculations.....	46
Producing HTML.....	47
Redirecting Output.....	48
Running External Processes.....	48
6 Flow Control.....	50
The if ... then:... else ... :if Structure.....	50
The ForEach Loop.....	51
The Abort Statement.....	53
7 Using Subfiles.....	54
Readfiles.....	54
Scope.....	54
Global Blocks.....	54
8 Extending With External Libraries.....	56
IPAT-S API Calling Conventions.....	56
The IPATS_API.h Header File.....	57
Sample API Code.....	58
9 Miscellaneous.....	60
Enhancing the Legibility of Scripts.....	60
In-Line Comments using #.....	60
Comment Blocks.....	60
Line Continuation.....	60
Ditto.....	60
Marking Inputs with <>.....	61
Assertions.....	61
Errors.....	62
Special Cases.....	62
List of Error Messages.....	62
Reserved Words.....	66
Appendix: The GNU Lesser Public License.....	68

## **Index of Tables**

Table 1. Command-line switches and values.....	5
Table 2. Default behavior that switches override.....	6
Table 3. Standard mathematical functions.....	22
Table 4. Special-purpose functions for summable variable arguments.....	23
Table 5. Format specifiers for in-line calculations.....	46
Table 6. List of error messages.....	62

*IPAT-S Language Reference*

# 1 Introduction

---

IPAT-S is a computer scripting language for developing quantitative models for medium and long-term sustainability scenarios.<sup>1</sup> The S in IPAT-S stands for “Script,” while “IPAT” is shorthand for a formula, first proposed in the 1970s by Commoner, Ehrlich and Holdren, that has long been used in discussing environmental impacts of human action:

$$\text{Impact} = \text{Population} \times \text{Affluence} \times \text{Technology}$$

In the IPAT formula, the population sets the scale of the overall impact, modified by changing patterns of affluence and technology. The IPAT-S language generalizes the IPAT idea. Variables that determine the scale of impacts are given a special role. In IPAT-S, the IPAT formula is written:

```
Population >> Affluence * Technology -> Impact
```

The notation >>-> suggests an arrow, leading from `Population` to `Impact`, but whose influence is modified by `Affluence` and `Technology`. This formula can be read as, “changes in the scale of population lead to changes in the scale of an environmental impact, modified by changes in affluence and technology.”

IPAT-S is open-source software. It can be downloaded at no charge from <http://ipat-s.kb-creative.net/>, and can be used in personal and commercial applications. Sample scripts are available on the IPAT-S web site and in the IPAT-S distribution.

## ***Scenarios for Sustainable Development***

Sustainable development has been given many definitions. Perhaps the most commonly-cited is the one by the Brundtland Commission, which introduced the idea of sustainable development to the policy world in its report, *Our Common Future*: “[Sustainable development] meets the needs of the present without compromising the ability of future generations to meet their own needs.” The idea is that in the long run, humanity – as well as the rest of the living world – has only the world’s renewable resource base to support it. Yet humanity today is not particularly good at using renewable resources, and an enormous amount of global economic production, transport and even food production is dependent on non-renewable resources – especially in the wealthier countries – while at the same time these activities are destroying or degrading renewable stocks – often in poorer countries. At the same time, much of the world’s population does not enjoy the fruits of these activities. How will it be possible to eliminate the extreme poverty and inequality seen in the world today, without destroying the resource base on which future well-being will

---

1 Scripting languages require the use of an *interpreter*, a program that interprets that script on the fly and carries out its commands. The IPAT-S interpreter is available for free. It can be downloaded with the IPAT Studio IDE from <http://ipat-s.kb-creative.net/>.

depend? This is the central question of sustainable development. It has no agreed-upon answer, and in fact has no single answer. The challenge of sustainable development is profound.

The idea of sustainable development, as defined by the Brundtland Commission, implies that we must take action today while considering developments that may occur decades in the future. A good tool for examining alternative pathways into the future is scenario analysis. Scenarios are stories of possible futures, which help to guide decision-making by expanding the range of options under consideration. Although they rely on imaginative storytelling, they must be plausible and consistent if they are to be of use in developing policy or in guiding people to think about alternative ways of living. This goal is partly achieved by combining quantitative scenarios with narratives. IPAT-S is one of several tools available that support quantitative scenario development.

This document is a reference for the IPAT-S language, and is not a manual on scenario building. For information about sustainability scenarios, see the site <http://www.scenariosforsustainability.org/>. The site provides scenario-related resources, as well as a "toolkit" of free software to support a major scenario exercise.

## ***Intended Audience***

Scenario exercises come in many varieties. IPAT-S is designed primarily to support expert-input scenarios, in which a team of experts in the sectors of interest collaborate on the development of a scenario, which is subsequently released for review. However, it is not necessary to have an expert team in order to use IPAT-S: it is equally suited for quick studies and extensive model building.

IPAT-S is a programming language. If an IPAT-S user has some experience with programming, he or she should be able to understand the IPAT-S syntax and adapt one or more of the example scripts to suit the study. If not, the basic IPAT-S syntax is not difficult to learn, and there are many scripts available in the IPAT-S distribution. Users will also need scenario-building skills. These are not easy to define, but include the ability to translate a scenario narrative into a quantitative framework and the ability to focus only on the most relevant elements in either the narrative or the quantitative calculations. IPAT-S users may wish to seek advice from experienced scenario developers when building a scenario.

This manual is the official documentation of the IPAT-S language. If you are interested in using IPAT-S to develop scenarios, then you should find this manual a useful reference.

## ***Features and Limitations***

Broadly, IPAT calculations relate the scale of activity to the scale of an impact. IPAT-S is designed so that users can generate simple-looking scripts that implement the IPAT idea. The IPAT-S language is designed so that scripts can be "read" by people who are not programmers to give them an idea of the structure of scenario calculations. This is achieved in part through a compact syntax. An IPAT-S script that performs complex calculations may take only a few lines: please see the sample scripts on the web site (<http://ipat-s.kb-creative.net/>) and in the IPAT-S distribution.

In addition to the central question of relating scales that is at heart of the IPAT idea (and the IPAT-S language), IPAT-S also provides support for addressing the question of allocation of resources and technologies that emerges in many discussions of sustainability, and the imposition of constraints. It does this by allowing the user to implement linear programming (LP) problems in their scripts. Designing and solving LP problems is generally a difficult task. IPAT-S's compact syntax makes the job easier. A section of this manual is devoted to building LP calculations in IPAT-S.

The IPAT-S language is also extensible through the use of external libraries., and external programs can be executed from within an IPAT-S script. Under Windows, dynamic link libraries (DLLs) that conform to the IPAT-S application programming interface (API) can be loaded by an IPAT-S script. For users interested in this approach, a standard library of loadable procedures is provided in the IPAT-S distribution, along with the source code. The API and the use of external libraries are also discussed in this manual. External programs can be used to combine IPAT-S scripts with other models if the other models support text input and output. For example, if a Fortran model uses a text data input file, then the file can be produced as output from an IPAT-S script, the Fortran program can be run from within IPAT-S, and the output from the Fortran program can be processed and read into the IPAT-S script.

It is worth mentioning one important kind of problem for which IPAT-S is not designed – dynamic systems. (However, if a systems dynamics model were provided through an external library – a DLL – then it could be used in IPAT-S.) There are excellent software packages that offer a graphical user environment for analyzing systems dynamics problems. A well-known commercial package is Vensim, which also offers a limited (but still powerful) version that is free for non-commercial use. A less well-known text-based tool is Illium, which is available for free. Simulations of dynamic systems can take into account dynamical feedbacks between different parts of a complex system. Generally, in quantitative scenario development it is good to have a well-rounded toolkit of scenario-development software, which might include both systems dynamics software as well as tools like IPAT-S that do not use a systems-dynamics approach.

## ***License Information***

IPAT-S is being released under the GNU Lesser Public License, which is reproduced in the appendix. The software is copyright © 2002-2005 by Eric Kemp-Benedict. Under the GNU Lesser Public License, you may use the software for free. You may copy and modify the software, and may include it in commercial programs. However, you do not own the copyright, and cannot prevent anyone else from giving it away for free. For more information, see the full license. Note that the GNU Lesser Public License is intended primarily for software libraries, while the IPAT-S interpreter is a stand-alone program. However, I want to allow people to adapt the code and embed it in their own software. When it is used this way, it is being used as a software library.

## 2 Overview

---

### **Overall Structure of an IPAT-S Script**

An IPAT-S script consists of three parts: a specification of years, a list of dimensions (if any) and the body of the script. They must appear in this order in the script.

### **Base year & scenario years**

An IPAT-S scenario departs from a base year, a recent year for which most or all of the needed data are available.

To specify the base year, enter

```
base year year
```

or

```
baseyear year
```

The base year is followed by an optional declaration of scenario years. To specify one or more scenario years, enter

```
scenario year year
scenyear year
scenario years year1, year2
```

or other variations (such as `scenyears`). The commas are optional. Up to 999 scenario years can be declared. (The number of years is limited to 999 because year *values* are identified as four-digit numbers. To avoid confusing year values with year indices, years cannot have more than three digits.)

In some cases, there may be several scenario years at regular intervals. A special notation is offered by IPAT-S for this case. Each of these statements declares 2010, 2020, 2030, 2040 and 2050 as scenario years:

```
scenario years 2010 to 2050 by 10
scenyears 2010...2050 every 10
```

You can refer to year values in the script using a predefined variable `year` (or `y`, or `yr`). For example, `y. 0` will give the value of the base year, `y. 1` will give the first scenario year and `y. last` or `y. fin` will give the final scenario year.

### **Dimensions**

Dimensions offer a way of distinguishing different data. For example, to distinguish a variable `FuelUse` by fuel, define a dimension:

```
dimension fuel 'Petroleum', 'Coal', 'Natural Gas', 'Other'
```

The commas are optional. If this dimension is defined then variables later in the script can be distinguished from one another by fuel.

## Body

The body contains most of the script. It consists of data, variable declarations, chains, reporting statements, LP blocks and other script elements. The distinguishing feature of the body is that it cannot contain a specification of years or dimensions: those have to appear before the body.

## Running the IPAT-S Interpreter

The IPAT-S interpreter is a command-line (DOS) program that reads in an IPAT-S script, performs all the calculations and produces the output that the script calls for. There are different ways to run the interpreter. For each of these examples, suppose the script is in a file called “MyScript.ips” in the current working directory, and the IPAT-S interpreter (called “IPAT\_S.exe”) is in the directory C:\IPAT-S\. Output is being directed to a file called “MyScript.out.” These are alternative ways of running the interpreter:

- To run the script and capture all of the output, use:
 

```
C:\IPAT-S\IPAT_S.exe -o MyScript.out MyScript.ips
```
- Equivalently, use
 

```
C:\IPAT-S\IPAT_S.exe < MyScript.ips > MyScript.out
```

All of the available switches and their possible values are shown in Table 1. Most of the switches turn off some default behavior. The default behaviors are shown in Table 2.

Table 1. Command-line switches and values

Switch	Value	Result
-o	<i>filename</i>	Send output to file <i>filename</i>
-d	S	Space-delimited output for report statements
	T	Tab-delimited output for report statements
	<i>c</i>	Use character <i>c</i> as a delimiter
-D	N or n	Disable assert statements and set <code>_DEBUG_</code> to <code>false</code>
-l	N or n	Do not report LP errors
-p	N or n	Don't output print blocks
-q	N or n	Don't quote strings in delimited output
-r	N or n	Don't output report statements
-w	N or n	Use “long” format, not “wide” format, for report statements

Table 2. Default behavior that switches override

Switch	Overrides This Default
-o	Standard output
-d	Comma ( , )
-D	Asserts are active and <code>_DEBUG_</code> is true
-l	LP errors are reported to standard error
-p	Print blocks are output
-q	Strings are quoted in delimited output from report statements
-r	Report statements are output
-w	Report statements use “wide” format

By default, report statements produce standard comma-separated-variable (CSV) formatted data, in which separate data items are delimited by commas and strings are quoted. CSV-formatted data can be read by most spreadsheet and database programs, as well as by other kinds of programs. For standard tab-delimited output (in which strings are not quoted), use

```
C:\IPAT-S\IPAT_S.exe -d T -q n -o MyScript.out MyScript.ips
```

Sometimes it is desirable to produce output only from report statements, and other times only from print blocks. For example, if print blocks are used to produce HTML output, then report statements might be turned off so that HTML output can be sent to another program (e.g., to a CGI script). By default, output from both print blocks and report statements is produced. To specify that no report statement output should be produced, use the `-r` switch, and to specify that no print block output should be produced, use the `-p` switch:

- Turn off report statements:

```
C:\IPAT-S\IPAT_S.exe -r N -o MyScript.out MyScript.ips
```

- Turn off print blocks:

```
C:\IPAT-S\IPAT_S.exe -p N -o MyScript.out MyScript.ips
```

The `-w` switch turns off “wide” format for report statements. In wide format (the default), values for all years for an indicator are given on the same line. In long format, the value for each year is given on its own line.

Finally, the `-D` and `-l` switches control reporting of errors. The `-D` switch is typically used for production code, while the `-l` switch is used if return codes from linear programming models (LP blocks) is handled by the script.

## Executing IPAT-S from Other Programs

An IPAT-S script can be executed from any program written in a language that supports the execution of command-line programs. The Windows Application Programming Interface (API) defines the `CreateProcess()` function for running external programs. Many programming languages (including IPAT-S) provide support for executing external programs, either through `CreateProcess()` or with their own procedures. Programming languages that provide support include C, C++, Delphi, Visual Basic (VB), Visual Basic for Applications (VBA), various database scripting languages, and major scripting languages such as Perl, Tcl/Tk, Ruby, Python, etc. Some of these languages, such as Perl, Ruby, Python and Tcl/Tk, are free, well-developed and powerful. VBA is the scripting language for Microsoft Office, so it is available in Excel, Word, and other programs in the Microsoft Office suite. Finally, free and open-source C and C++ compilers are available for most platforms.

An example of a program that runs the interpreter is the graphical user interface (GUI) that comes with the IPAT-S distribution. The GUI is a program called IPAT Studio, written in the popular scripting language Tcl/Tk. The source code for the program is available from the IPAT-S site, <http://ipat-s.kb-creative.net/>. The IPAT Studio program uses the commands above to process IPAT-S scripts. It is a wrapper program that sits on top of (“wraps”) the IPAT\_S interpreter. Here is the Tcl/Tk command in the IPAT Studio program code that processes the script:

```
exec $ipatInterpreter -p n -o $datafile $scriptFile
```

In this command, `$ipatInterpreter`, `$datafile` and `$scriptFile` are all Tcl/Tk variables that hold the file names for the interpreter (IPAT\_S.exe), the output data file and the script file. In this case, only the report commands are being processed, so print block output is turned off using `-p n`. The output is directed to `$datafile` by using the `-o` switch. By using this technique, it is possible to put an IPAT-S “engine” inside of another program. The IPAT Studio program does this in a very general way: any IPAT-S script can be run using IPAT Studio and the output can be retrieved. However, the same approach can be used in less-general cases. For example, once a script has been developed, it is possible to make a wrapper program that only allows the user to change a few key inputs, runs the pre-built IPAT-S script, and then retrieves the output for display. For a well-developed script, this is a way to make a user-friendly interface. The program IPAT Scenario Navigator uses this approach to allow users to browse an existing IPAT-S script.

## 3 Data Types

---

### *Numerical Values*

All numbers are represented internally as double-precision floating point values. If a number is a pure fraction (such as 0.15), then the decimal point must have a zero in front of it. If a number is followed by a percent sign (%), then it will be divided by 100. For example, 15% will be represented internally by 15/100 (= 0.15).

These are all valid ways to represent numbers:

```
23      # A double-precision value equal to 23.0
2.3e1   # Same as the previous example
0.15    # A double-precision value equal to 0.15
15%     # Same as the previous example
```

The pound sign (#) in this example indicates a comment (discussed later).

### *Logical Values*

There are two predefined logical values, `true` and `false`. These can also be referred to as `yes` and `no`. Different scripts might contain different mixes of “true and false” or “yes and no,” depending on how each script is meant to be read.

### *String Values*

Strings are sequences of letters, digits or other characters. In IPAT-S, strings are delimited by either single or double quotes. For example:

```
'This is a string'
"This is a string"
```

are both valid. While either single or double quotes are accepted, you cannot mix them. For example,

```
'This is NOT a string"
"This is NOT a string'
```

are both invalid.

String values appear many places in IPAT-S scripts: as labels for indicators in report statements, as values for dimensions and as file names. There are no user-defined string variables, but within a `foreach` loop, the IPAT-S interpreter makes the loop index (a string) available with a special syntax. Also, IPAT-S offers a string concatenation operator.

### *Data Types With Dimensions*

Certain types of variables – summable variables, variables and ratios – can be labeled with dimensions. For example, to represent passenger transport fuel use by transport mode and fuel, define

```
dim transp_mode 'Road', 'Rail', 'Air'
dim fuel 'Petrol', 'CNG', 'Biofuel', 'Coal', 'Electric'
```

```
summvar transpFuelUse{transp_mode fuel}
```

Having defined the dimensions, you can then refer to variables by dimension. For example, to set the base-year value for a variable `EnergyUse` for the fuel `Petrol` enter

```
EnergyUse.0 {fuel = 'Petrol'} = 37 # Energy in PJ
```

Specific dimension values can also be referred to inside chain expressions (discussed in a later section.) For example,

```
VA{sect = 'Ind'} >>PollIntens-> Pollution
```

drives pollutant emissions by changes in the value added from the industrial sector. Dimensions can also be specified at the end of a chain. For example,

```
VA{sect = 'Ind'} >>PollIntens-> Pollution{poll = 'SOx'}
```

Values can also be summed or averaged over one or more dimensions. For example,

```
VA{sect = sum} -> GDP
```

## Numbers With Dimensions

In some cases (especially when creating a linear program in an LP block), it is useful to specify dimensions for a numeric value. For example, the expression `5{sector}` will assign the value 5 to all sectors.

## Automatic Dimension Matching

In most cases, dimensions do not need to be expressed explicitly in calculations. Instead, the IPAT-S interpreter will attempt to match the dimensions of all variables in the calculation and the result, and make reasonable, automatic, guesses as to what is intended.

In the simplest example, the dimensions match for all terms. For example, in this script fragment,

```
summvar Population{region}, GDP{region}
ratio income{region}
...
:: Population >> income -> GDP
```

all of the variables (`Population`, `GDP` and `income`) have the same dimension, `region`. In this case, the GDP in each region is given by the regional population and income changes.

Suppose instead that `Population` and `GDP` have a regional dimension, but `income` (that is, the income growth rate) is the same for all regions. Then the script should be written:

```
summvar Population{region}, GDP{region}
ratio income
...
:: Population >> income -> GDP
```

In this case, the dimensions do not match for the different variables. Instead, the IPAT-S interpreter assumes that what is meant is that the same income values should be applied to all

regions. If that is not what is intended – for example, if the income variable should only apply to region 'Region 1', then the script should be written

```
summvar Population{region}, GDP{region}
ratio income
...
:: Population{region= 'Region 1'} >> income -> \
   GDP{region= 'Region 1'}
```

(Note that for this example, the IPAT-S line-continuation character, "\", has been used.)

Suppose now that GDP is meant to vary by scenario, as well as by region. The same population is used for all scenarios, but income growth varies by scenario (and is the same for all regions). In this case, the script can be written:

```
summvar Population{region}, GDP{region, scenario}
ratio income{scenario}
...
:: Population >> income -> GDP
```

For this script, the IPAT-S interpreter will create all possible `region-scenario` combinations, and apply the result to `GDP`. This is probably what was intended: the population in a given region is the same across scenarios, while for a given scenario the income growth is the same across regions. The combination of population and income growth drive changes in GDP for a given region and scenario.

Finally, suppose that GDP varies by region and scenario, while Population and income each vary only by region. That is, in this example regional GDP is the same in all scenarios. The script in this case becomes

```
summvar Population{region}, GDP{region, scenario}
ratio income{region}
...
:: Population >> income -> GDP
```

In this case, there is an unmatched dimension. The variable in which the result of this calculation is being stored (`GDP`) has a dimension (`scenario`) that the result itself does not have. In this case, the IPAT-S interpreter will assign the same regional GDP to all scenarios.

## ***Unmatched Dimensions***

In the final example in the previous section, the expression has an unmatched dimension, in that the variable being given the result from a calculation (`GDP`) has a dimension (`scenario`) that the result of the calculation itself does not have. There is another possible case of an unmatched dimension – when the result of a calculation has a dimension that the variable being assigned the result does not have. While the example in the previous section is not a problem, this second situation is (almost always) an error within a chain expression.

The second situation is an error because in most cases there is no obvious and sensible way to combine the variables. Rather than having the interpreter silently try to guess the meaning, when

this situation arises it issues a warning, “Warning: dimension not available in result of a chain.” It then goes ahead and tries to calculate the result, but the result is probably not what was intended. For example, consider this fragment of a script:

```
dimension practice 'rainfed' 'irrig'
dimension crop 'cereals' 'roots & tubers' 'oilcrops' 'other'
summar Area{crop, practice} Production{crop}
ratio yieldIncrease{crop, practice}

...

:: Area >> yieldIncrease -> Production
```

This looks like it ought to translate changes in area by practice into changes in total crop production. However, because production is not broken down by practice, there is actually insufficient information to determine how total production changes. The reason is that the relative yields on rainfed and irrigated land are required in order to do the calculation, and this cannot be calculated without knowing production separately for each practice, in addition to area. (The variable `yieldIncrease` is a ratio variable, so it is just an index, and carries no information about absolute area or production.) The interpreter will therefore complain with this expression. The solution is to write something like this:

```
dimension practice 'rainfed' 'irrig'
dimension crop 'cereals' 'roots & tubers' 'oilcrops' 'other'
summar Area{crop, practice} ProdByPract{crop, practice} \
    Production{crop}
ratio yieldIncrease{crop, practice}

...

:: Area >> yieldIncrease -> ProdByPractice
summarize ProdByPractice as Production
```

There is now sufficient information to calculate yields, and the result (`ProdByPractice`) can then be summed (summarized) to give production.

## Summable Variables

Summable variables are variables that it makes sense to add up, such as population, GDP (gross domestic product), energy use, total carbon emissions, water use, etc. They are distinguished from ratio variables, which in IPAT-S are indices of ratios such as income (GDP/capita), aggregate energy intensity (energy/GDP) and other ratios, as well as plain variables, which are neither summable variables nor ratios (such as fuel shares).

There are two reasons for separating summable variables from ratios. First, by assuming that summable variables can be added together, the syntax of IPAT-S can be simplified. Second, variables that can be summed generally set the scale of overall changes. Ratios express how a change in the scale of one variable (say, population) gets translated into the scale of another variable, such as total food requirements. They each provide different kinds of information. For example, China's GDP (a summable variable) is roughly one-half that of the U.S., when adjusted for purchasing power parity. However, its GDP per capita (a ratio) is much smaller, closer to one-tenth that of the U.S. GDP is a measure of overall economic weight, while GDP per capita is used as an indicator (a very imperfect one) of standard of living. Neither indicator is sufficient in itself to give a picture of the relative economic positions of China and the U.S.

In IPAT-S, summable variables automatically change along a time dimension: they can take on different values in the base year and in scenario years. Also, they can change along any other user-defined dimensions.

To declare summable variables, use the following syntax:

```
summable variable var1, var2, var3, etc.
summ var var1, var2, var3, etc.
summvar var1, var2, var3, etc.
```

or variations on these forms, such as `summ variable`. The commas are optional. When summable variables are declared, all of their values are set to zero by default.

To specify that a variable `var` changes along a dimension `dim`, enclose `dim` in braces after `var`:

```
summvar var{dim}
```

If a variable does not have any dimensions, then its scenario values can be specified when the variable is declared. For example,

```
summvar population = 300, 320, 350
```

For a scenario with one base year and two scenario years, this will set the variable `population` to have a base-year value of 300, a value of 320 in the first scenario year and a value of 350 in the last scenario year. For a scenario with three or more scenario years, the value of `population` will be 350 in all scenario years from the second year onward. In constructions like this one that involve numbers, the commas are required (unlike for year, dimension or variable definitions), because numerical expressions might be used to specify a value, and the expressions can have spaces embedded in them. For example,

```
# 3-year averages
summvar production = (298 + 295 + 302)/3, \
                    (320 + 318 + 322)/3, \
                    (335 + 337 + 332)/3
```

(Note that the backslash character "`\`" in this example is a line-continuation character.)

In addition to setting all scenario values at once, numerical expressions are used to specify values for scenario variables in particular years. This is most useful for setting initial, base-year values for variables. For example, here is a specification of population for all scenario years, as well as GDP and value added in the base year:

```
base year 1997
scenyears 2010 2020

dim sector 'Agric', 'Indust', 'Services'
summvar Pop, GDP, VA{sector}

Pop = 300, 320, 327

GDP.0 = 23527 * Pop.by # $ 23,527/cap is ave. income
VA.1997 {sector = 'Agric'} = 47% * GDP.first
VA.init {sector = 'Indust'} = 23% * GDP.BY
VA.by {sector = 'Services'} = (100% - 47% - 23%) * GDP.0
```

For purposes of demonstration, all the different ways of specifying the base year are shown: `GDP.0`, `GDP.by`, `GDP.BY`, `GDP.init`, `GDP.first` and `GDP.1997` are all equivalent ways of indicating the base-year value of GDP (for the case where 1997 is the base year). Similarly, `GDP.2` (or `GDP.2020`, in this example) indicates the value for the second scenario year, and `GDP.fin`, `GDP.final` or `GDP.last` indicates the value for the last scenario year.

## Variables

A variable is in most ways like a summable variable. They can be used in all contexts where summable variables are used. In fact, for the interpreter itself there is no distinction. However, to make scripts easier to read, summable variables should be used for quantities that can be summed, such as population or GDP. The variable data type should be used for numbers that are not summed, but that also are not implemented as IPAT-S ratio variables. One example of the use of an IPAT-S variable is for a fuel share. Another situation where a variable might be used is when a ratio is specified explicitly in a scenario, rather than as an index or growth rate, for example by specifying GDP/capita explicitly rather than leaving it implicit (see *ratio variables* in the next section).

Variables are declared in the same way as summable variables, but the word `summable` or `summ` is dropped. For example,

```
variable var1, var2, var3, etc.
var var1, var2, var3, etc.
```

As with summable variables, when variables are declared, all of their values are set to zero by default.

A special variable that provides the value of each year (e.g., the numbers 2000, 2010 and 2020) is created automatically in IPAT-S. To use the value of the current year in an expression, include the variable `year` in an expression. The variable name is not case-sensitive, and can also be written `yr`, `y`, `YR`, `Year`, etc. This can be useful for making time-dependent formulas other than the simple growth rate that is a basic feature of the IPAT-S syntax. For example, the following expression performs a linear interpolation over time between a base-year value and a long-term convergence value:

```
:: byVal + (convVal-byVal)*(y - y0)/(convYear - y0) -> Val
```

## Ratio Variables

A ratio variable is an index relative to the previous year. For example, suppose there are two summable variables, `Pop` and `GDP`:

```
summar Pop GDP
```

Then the ratio `GDP/Pop` represents average income. In most scenarios it is important to keep track of all three variables – population, GDP and average income. However, while it is possible to calculate and store the ratio, the information is redundant, if population and GDP are already known. Instead, the IPAT-S syntax allows for a specification of how the ratio changes over time, for example as an income growth rate, rather than keeping track of the ratio value explicitly. In the IPAT-S language, such changes are specified using ratio variables.

To declare one or more ratio variables, use the following syntax:

```
ratio var1, var2, var3, etc.
```

The commas are optional.

Ratio variables, like summable variables, automatically change along a time dimension: they can take on different values in different scenario years. (The value for a given year is the index with respect to the previous year. Ratio variable values are not defined for the base year.) Also, ratio variables can change along any user-defined dimensions. In this case, the dimensions must be specified when the variable is declared, similar to the way that summable variables are specified. For example,

```
ratio var1, var2{dim1, dim2}
```

If a ratio variable does not have any dimensions associated with it, then the value can be specified when the variable is declared. For example,

```
ratio income = growth(2.1%, 2.5%)
```

This line says that the ratio variable `income` is an index relative to the base year that increases by 2.1% per year on average between the base year and first scenario year, and by 2.5% per year between the first and second scenario years (and for all subsequent scenario years). This is an example of an index expression. Index expressions are discussed in detail below.

## Number Variables

A number variable holds a single numerical value. It does not change in time or across any dimensions.

One case where number variables might be used is for conversion factors. To convert from CO<sub>2</sub> to carbon equivalent, for example, the mass of CO<sub>2</sub> should be multiplied by 12 and divided by 44 (the ratio of the atomic weight of carbon to the molecular weight of carbon dioxide). If this must be done frequently in a script, then it might be convenient to declare a number variable:

```
number CO2toC = 12/44
```

Another situation where number variables are often necessary is in complex index expressions. Index expressions are discussed in detail below.

As with summable variables and ratios, number variables can be declared before they are given a value. For example:

```
number var1, var2, var3, etc.
num var1, var2, var3, etc.
```

The commas are again optional.

In addition to number variables, when the year and all dimensions for a summable variable or variable are fully specified, then it is treated as a number. For example, if `VA` varies by the dimension `sector`, then `VA.last{sector = 'Agriculture'}` (the value for `VA` for the 'Agriculture' sector in the final scenario year) is treated as though it were a number. It can appear in any expression where a number can appear.

## Logical Variables

A logical variable holds a single logical value. It does not change in time or across any dimensions. Logical variables are declared using the keyword `logical`:

```
logical isDecreasing
```

A logical variable can be initialized when it is declared:

```
logical isDecreasing = true
```

One use for logical values is to “turn on” or “turn off” portions of a script. For example,

```
logical printHTML = false
...
if printHTML then
  read "Reports/HTML.ips"
endif
```

## 4 Calculations

---

### *Chain, ::, and Related Expressions*

Chain expressions are the basic IPAT-S expression. They specify how summable variables change, and in doing so compact a great deal of information into a small space. Chains must be declared by putting `chain` or `::` at the start of the line. They have the basic form:

```
chain summvar >> index_expr -> summvar
```

or

```
:: summvar >> index_expr -> summvar
```

although there are many variations on this basic form.

Here is an example of a chain:

```
:: Pop >> income -> GDP
```

The syntax is meant to suggest an arrow, `>>->`, leading from `Pop` to `GDP`, but modified by `income`. This line is translated into the following sequence of instructions by the interpreter:

1. Take the scenario values for the summable variable `Pop`
2. Let the summable variable `GDP` increase in the scenario in the same ratio as `population`, multiplied by the ratio variable `income`.

That is,

$$\text{GDP.N} = \text{GDP.N-1} * (\text{Pop.N} / \text{Pop.N-1}) * \text{income.N}$$

for all years `N`.

The usefulness of the chain notation is hopefully clear in this simple example. The calculations carried out by the interpreter, if they were written in a general-purpose programming language, would be difficult to interpret in a glance. Also, they must be implemented separately for each year by looping over years. When using the chain notation, all of this is taken care of internally by the IPAT-S interpreter.

The chain notation has a straightforward interpretation. For example, the chain above can be read, "GDP grows with population, subject to changes in average income." Similarly, the IPAT formula itself can be written,

```
:: Population >> Affluence * Technology -> Impact
```

which has the interpretation that the environmental impact grows like population, but modified by changes in affluence and technology.

Sometimes there is no intermediate ratio between one summable variable and the next, because the final summable variable should simply grow in proportion to the one driving the change. For example, suppose total solid waste generation is expected to grow directly in proportion to GDP. Then introducing variables `SolidWaste` and `GDP`, the chain can be written

```
:: GDP >>-> SolidWaste
```

There is another situation in which a ratio can be omitted. This is the case for `summarize...as` expressions. For example, the variable `GDP` may be expressed as the sum of `Value_Added` for each sector, where `Value_Added` is given by `Employment`, modified by changes in `Productivity`, and where `sector` is a dimension. In this case, the expressions

```
:: Employment >> Productivity -> Value_Added
summarize Value_Added as GDP
```

or the combined expression

```
summarize Employment >> Productivity -> Value_Added as GDP
```

will sum up the `Value_Added` figures to give total `GDP`.

A third situation in which ratios might be omitted is when one summable variable or variable is directly a function of other summable variables or variables (`summarize...as` statements are a special case of this). For example, total income may be available by income quintile (a summable variable). Then average income by quintile can be calculated in the following way:

```
dim quintile 'Q1' 'Q2' 'Q3' 'Q4' 'Q5'
summyar totIncome{quintile}
var incomePerCap{quintile}
...
# Average income by quintile (each quintile has 20% of the population)
:: totIncome / (20% * Pop) -> incomePerCap
```

In this case, the chain has no `>>` in it. This tells the interpreter that, rather than `incomePerCap` growing *like* the expression on the left, it is exactly equal to it, after first adjusting for dimensions. One of the adjustments that will be made is that any dimensions that appear before `->` and that do not appear after it will be summed over. This is also what `summarize-as` statements do. So, the following two expressions are equivalent:

```
summarize Value_Added as GDP
```

and

```
Value_Added -> GDP
```

Either one can be used, depending on how the script is meant to be read.

Another type of situation is where a summable variable simply grows by some growth rate, without being driven by another variable. The chain notation can be used in this case as well, but with the driving variable omitted:

```
:: >> 100% - delta + S_K_ratio * s/k -> K
```

A very special case is when a summable variable should remain at its base-year value throughout the scenario. In this case there is no driving variable, and the growth rate is zero (so the index is one). Using the syntax above, this can be written:

```
:: >>-> byX
```

for a variable `byX` that remains at its base-year value. This is a legal expression. However, this is not easy to interpret, so IPAT-S provides a shorthand:

```
byv byX
```

or

```
byvalue byX
```

will set a summable variable or variable to its base-year value.

Chains can be connected in sequence. They can be also be added and subtracted. For example,

```
:: Pop >>Income-> GDP >>EnergyIntensity-> EnergyUse \
    >>CarbonIntensity-> CarbonEmiss
```

will calculate GDP, EnergyUse and CarbonEmiss in the same expression, while

```
:: (Pop >> DomI -> UrbBOD + IndVA >> IndI -> IndBOD) >>-> BOD
```

will calculate UrbBOD, IndBOD and then add them to drive total BOD. In fact, this is where the term “chain” comes from, since separate chains can be linked together to give a larger chain.

## Calculating Sums and Averages

To calculate the sum or average of an expression involving summable variables or variables over a dimension, set the dimension equal to `sum` or `ave`. For example,

```
:: (Pop * GDPperCap){region = sum}/Pop{region = sum} -> AveIncome
```

will calculate the average income (GDP per capita) over all regions, weighted by population.

## Index Expressions

Chains link summable variables to one another via index expressions, which are indices with respect to a previous year. Ratio variables are given their values using index expressions. That is, a ratio variable stores the value of an index expression, the same way a number variable stores the value of a number. There are several special forms for writing index expressions.

These are equivalent ways of creating an index that grows by 2% each year across the scenario:

```
index(1.02)
ndx(102%)
growth(2%)
gr(0.02)
incr(2%)
```

Note that `index` and `ndx` are synonyms. Also, `growth`, `gr`, `incr` and `increment` are all synonymous to each other. These variations are offered for convenience and to increase the clarity of scripts.

To apply an annual increase of 2% over the first period and 3% over the second (and later) periods, these are all equivalent:

```
index(1.02, 1.03)
ndx(102%, 1.03)
growth(2%, 3%)
gr(0.02, 3%)
incr(2%, 0.03)
```

The parentheses indicate that the values should be applied on an annual basis. However, sometimes it is more convenient to specify a change between one scenario year and another. In this case, use square brackets. For example, these are all equivalent ways of specifying an increase

of 20% between the base year and the first scenario year, and of 30% between the first and second scenario years (and all subsequent scenario years):

```
index[120%, 130%]
increment[20%, 30%]
growth[0.2, 0.3]
```

If a summable variable or variable is introduced in an index expression, then it is first converted to an index. For example, if GDP is a summable variable, and PollIntensity is a ratio, then

```
PollIntensity = GDP ^ 0.14
```

would say that PollIntensity should grow like the index of GDP, raised to the 0.14 power (i.e., in this example, PollIntensity is assumed to vary with a GDP elasticity of 0.14). This is also an example of a complex index expression, in which indices appear in formulas. For example,

```
income ^ (-0.3)
GDP ^ (-0.1)
Affluence * Technology
diet = dietAsymp * income^0.6 / (income^0.6 + dietAsymp - 1)
```

The last example implements a sigmoid (s-shaped) curve. As the income ratio variable grows very large, diet approaches the value dietAsymp (which is assumed to be a number variable in this example). Index expressions can combine variables by multiplication (\*), division (/), addition (+), subtraction (-) and exponentiation (^), and by using functions.

The convention of converting all summable variables (and variables) in index expressions into indices is usually convenient, but it can sometimes lead to unexpected results. For example, suppose you wish to represent the accumulation of capital K in a scenario by the accumulation of savings (expressed as a fraction s of total output), less depreciation, delta, modified by the capital-to-output ratio k. Then you might be tempted to write something like

```
number delta = 9%
var s = 15%, 20%
var k = 1.1, 0.97
:: >> 100% - delta + s/k -> K
```

However, s and k in the index expression embedded in the chain will first be converted to indices, so within the complex index expression, only relative values for s and k will be used, which was not what was intended. The solution to this problem is to introduce a number variable that is equal to the base-year value of the ratio s/k. Number variables are represented verbatim in index expressions, so that unlike summable variables and variables, they retain their values. Here is a complete script that implements this simple capital-accumulation model:

```
comment:
This script implements a simple model of capital accumulation.
Given the growth in the capital-output ratio, the savings
rate and the depreciation rate, the level of investment
emerges from a simple accounting relationship. This relationship
is used to determine the level of capital. Combined with
the capital-output ratio, this then determines the level
of output.
:comment

baseyear 2000
sceneyears 2001 to 2015 by 1

summvar K X S # capital, output, saving

var s = <15%> # Saving rate over time (here, constant)
number delta = <9%> # depreciation rate (assumed constant, so a num)
```

```

K.0 = 10.0 # trillion US$
X.0 = 9.0 # trillion US$/year
S.0 = s.0 * X.0

number S_K_ratio = S.0 / K.0 # Base year saving/capital-output ratio

ratio k = gr(<1.0%>) # growth in capital/output ratio

# Capital declines with depreciation, increases with savings rate,
# depending on capital productivity
:: >> 100% - delta + S_K_ratio * s/k -> K

:: K >> 1/k -> X

report X as "GDP"
report K as "Capital"
report X/K as "Capital productivity"

```

## Numerical Expressions

Number expressions are numbers, number variables, the results of functions acting on numbers and the results of calculations involving numbers that use multiplication (`*`), division (`/`), addition (`+`), subtraction (`-`) and exponentiation (`^`), and grouping with parentheses. Some examples:

```

3.01 # The number 3.01
x # The value of the number variable x
(x^2 + 3)^(-1) # A numerical calculation
sqrt(x) # The square root of a number variable
Pop.fin/Pop.init # The ratio of the final to the initial
# value of the population

```

## Logical Expressions

Logical values can be combined in logical expressions to give a logical result. In addition to grouping using parentheses, logical expressions can use the operators `.and.`, `.or.`, `.not.` and the special expression `.neither. ... .nor.` Also, numerical and string values can be compared using the relational operators `.gt.`, `.lt.`, `.eq.`, `.neq.`, `.ge.` and `.le.`

### **.and., .or., .not.**

Logical values can be combined using the conventional binary operators `.and.` and `.or.`, and by the negation operator `.not.` For example, if `logical1` and `logical2` are logical variables, then

```

logical1 .and. logical2
logical1 .or. logical2
logical1 .and. (.not. logical2)

```

are all legal expressions. There are synonyms for each of these operators:

```

.and., .&., .* # Equivalent forms for .and.
.or., .|., .+ # Equivalent forms for .or.
.not., .!., .~ # Equivalent forms for .not.

```

### **.neither. ... .nor.**

There is a special syntax for logical expressions in IPAT-S that can sometimes be used to make scripts easier to read, the `.neither. ... .nor.` syntax. Here is an example:

```

if .neither. logical1 .nor. logical2 .nor. logical3 then
  statements
endif

```

This says that if none of logical1, logical2 or logical3 are true, then the statements should be executed. Otherwise, they should be skipped.

## **.gt., .lt., .eq., .neq., .ge., .le.**

Numbers and strings can be compared using the `.gt.` (greater than), `.lt.` (less than), `.eq.` (equal), `.neq.` (not equal), `.ge.` (greater than or equal to) and `.le.` (less than or equal to) relational operators. For example,

```

if Pop.fin .gt. Pop.init then:
print:
The population increased over the course of the scenario
:print
else
print:
The population did not increase over the course of the scenario
:print
:if

```

For an example using strings:

```

# Identify inputs and outputs with the same name
foreach input, output:
  if $input .= $output then:
    ident{input=? output=?} = 1
  :if
:foreach

```

There are synonyms for each of these relational operators:

```

.gt., .>      # Equivalent forms for .gt.
.lt., .<      # Equivalent forms for .lt.
.eq., .=      # Equivalent forms for .eq.
.neq., .ne., .<>, .!=. # Equivalent forms for .neq.
.ge., .>=    # Equivalent forms for .ge.
.le., .<=    # Equivalent forms for .le.

```

## **String Concatenation**

IPAT-S offers a string concatenation operator, “+”. This is most useful within a `foreach` loop, where the loop index is available as a string. For example,

```

foreach dim:
  if $dim .ne. "Northeast" then:
    report x{dim = ?} as "x(" + $dim + ")"
  :if
:foreach

```

Will print the value of `x` for all values for `dim` except when `dim` equals “Northeast.” Also, it formats the output so that the label appears as `x(dimname)`.

## **Functions**

Several standard mathematical functions are available in IPAT-S. All functions take a single argument. For most functions, the argument can either be a number (or number variable), a summable variable or a variable. For certain special functions, the argument must be a summable variable or variable. When the standard mathematical functions are applied to either summable

variables or variables, the function is applied to all values, for all years and all dimensions. There is also a special value, pi, that cannot be changed by the user. It is equal to the constant  $\pi = 3.14159\dots$ , the ratio of the circumference of a circle to its diameter. This constant shows up in diverse situations, including statistics and geography.

The available standard mathematical functions are shown in Table 3.

Table 3. Standard mathematical functions

<i>Function</i>	<i>Description</i>
<code>abs(x)</code>	Absolute value of $x$
<code>cos(x)</code>	Cosine of $x$ , where $x$ is in radians
<code>exp(x)</code>	$e$ (= 2.7183...) to the power of $x$
<code>ln(x)</code>	Natural logarithm of $x$ (logarithm with a base of $e = 2.7183\dots$ )
<code>ln0(x)</code>	Natural logarithm, returning zero for zero values for $x$
<code>log(x)</code>	Logarithm base 10 of $x$
<code>log0(x)</code>	Logarithm base 10, returning zero for zero values for $x$
<code>N(x)</code>	Cumulative standard normal distribution
<code>N_inv(x)</code>	Inverse of the cumulative standard normal distribution
<code>neg(x)</code>	Equal to $x$ if $x < 0$ , equal to 0 otherwise
<code>pos(x)</code>	Equal to $x$ if $x > 0$ , equal to 0 otherwise
<code>sign(x)</code>	Sign function, equal to 1 if $x > 0$ , -1 if $x < 0$ and 0 if $x = 0$
<code>sin(x)</code>	Sine of $x$ , where $x$ is in radians
<code>sqrt(x)</code>	Square root of $x$
<code>step(x)</code>	Step function, equal to 1 if $x \geq 0$ , and equal to 0 if $x < 0$
<code>step0(x)</code>	Step function, equal to 1 if $x > 0$ , and equal to 0 if $x \leq 0$

For a normally distributed random variable with mean `mean` and standard deviation `sd`, the probability that the value is less than  $x$  can be calculated in IPAT-S using the standard normal distribution this way:

$$N((x - \text{mean})/\text{sd})$$

See the sample scripts for examples of using these functions. The normal distribution, logarithm and inverse normal distribution can be used together to implement a lognormal income distribution, which is demonstrated in some of the scripts.

In addition to the standard mathematical functions are the special-purpose functions listed in Table 4 that can only be applied to summable variables or variables. They each return a summable variable or variable with the same dimensions as their argument.

Table 4. Special-purpose functions for summable variable arguments

<i>Function</i>	<i>Description</i>
<code>accum(v)</code>	Integral over straight-line path defined by the values of $v$
<code>accumulate(v)</code>	Same as <code>accum(v)</code>
<code>byv(v)</code>	Summable variable containing the base year value of $v$
<code>byvalue(v)</code>	Same as <code>byv(v)</code>
<code>lag(v)</code>	Values for $v$ for lagged by one scenario year
<code>rate(v)</code>	Rate of change over time of $v$ *

\* Estimated by fitting a polynomial to scenario values.

## Procedures

In addition to the standard functions, IPAT-S supports external procedure libraries through the use of dynamic link libraries (DLLs). The procedures and libraries are external because they are written in a programming language other than IPAT-S, such as C, C++, Fortran, Delphi, Visual Basic or another language. With this feature, external models can be dynamically loaded as IPAT-S procedures, allowing the functionality of the model and the IPAT-S scripting language to be combined.

The procedure must first be loaded from the DLL before it can be used (called) in calculations. The IPAT-S syntax for loading a procedure from a DLL is:

```
load ExternalProcName from 'Library' as ProcName
```

The `ExternalProcName` is the name of the function in the library. `Library` is the filename of the DLL (the “.dll” extension is optional). The `ProcName` is the procedure name as it will be called in the IPAT-S script. The procedure name in the script can be assigned by the script writer, following the same rules as for variable names.

To use an external procedure, use the following syntax:

```
call ProcName using Argument1 Argument2 ... ArgumentN
```

The `ProcName` is the same as in the `load` statement above. `Argument1`, `Argument2`, etc. are the arguments required by the procedure.

The use of external procedures is somewhat restrictive. Arguments cannot be ratio variables. Also, if arguments are summable variables or variables, then they cannot have dimensions. While this may seem like a severe limitation, it is possible to accommodate variables with dimensions using a `foreach` loop, and this restriction greatly simplifies the application programming interface (API), making it much easier to develop and use external libraries than would otherwise be the case. The API is discussed in chapter 8.

There is a standard set of external procedures in the IPAT-S distribution. These are in a DLL file, the library “IPATS\_standard.dll”. In addition to using the procedures themselves, the source code for the standard library can be used as an example for using the IPAT-S API.

**Note**


---

By default, Windows Explorer hides DLL files from view. To see the file, you may need to change the settings on Windows Explorer.

To use the library in a script, the DLL file has to be in: 1) the same folder as the script, 2) the IPAT-S folder, or 3) the Windows System or System32 directory. If you get a `DLL failed to load` error, it may be because the file is not in one of these three places. Please see your Windows documentation for more information.

## The Standard Library

The following procedures are defined in the standard library:<sup>2</sup>

`accumulate` or `_accumulate`

*Usage:* call `accumulate` using `stock flow`

Assuming flow changes linearly between years, integrate flow and add to stock.

*Note:* This is now superseded by the built-in `accumulate()` and `accum()` functions

`accumulateExp` or `_accumulateExp`

*Usage:* call `accumulateExp` using `stock flow`

Assuming flow grows exponentially between years, integrate flow and add to stock.

`interpolate` or `_interpolate`

*Usage:* call `interpolate` using `var x x1 val1 x2 val2 ...`

Perform a linear interpolation between  $x_i, val_i$  pairs and store the result in `var`. Note that it is possible to interpolate over years by setting `x` to the `year` variable.

`uniform_rand` or `_uniform_rand`

*Usage:* call `uniform_rand` using `var min max [seed]`

Generate a uniformly-distributed random number between `min` and `max` and store the result in `var`. The `seed` is optional. If it is specified, then the random sequence will be initialized with `seed`.

`gauss_rand` or `_gauss_rand`

*Usage:* call `gauss_rand` using `var mean sd [seed]`

Generate a random number that follows a gaussian (normal) distribution with mean `mean` and standard deviation `sd` and store the result in `var`. The `seed` is optional. If it is specified, then the random sequence will be initialized with `seed`.

---

<sup>2</sup> Up to version 4.9.6, all of the procedure names in the standard library were preceded by an underscore (`_`). From version 4.9.6 forward, procedure names are no longer required to have an underscore. In any case, procedures can be given any name.

Logelet or \_Loglet

*Usage:* call Loglet using var sat1 dur1 mid1 ...

Perform a “loglet” composition using an arbitrary number of logistic functions, where `sat` is the saturation level, `dur` is the duration in years and `mid` is the midpoint year, and store the result in `var`.

Sample scripts in the IPAT-S distribution demonstrate the use of external libraries and the standard library. However, four procedures deserve special mention – `interpolate`, `uniform_rand`, `gauss_rand` and `Loglet`.

## The Interpolate Procedure

The `interpolate` procedure can be used to implement a lookup table. For example, suppose that industrial pollution intensity changes with average income (GDP per capita). Then a script could call the `interpolate` procedure as in this example:

```
base year 2000
scenario years 2010 2020
load interpolate from 'IPATS_standard' as Interp
summvar pop, GDP
pop = 300, 320, 340
GDP.0 = 22100 * pop.0
:: pop >> gr(1.2%) -> GDP
summvar pollIntens
call Interp using pollIntens GDP/pop \
                5000    1.2 \
                10000   1.1 \
                30000   1.0
report pollIntens as "Pollution Intensity"
```

## Generating Random Numbers

The `uniform_rand` and `gauss_rand` procedures generate pseudo-random numbers. The `uniform_rand` procedure generates a uniformly-distributed pseudo-random number between specified limits, while `gauss_rand` generates a pseudo-random number with a gaussian distribution (a normal, or “bell-shaped” distribution) with specified mean and standard deviation. To guarantee a consistent sequence of pseudo-random numbers, a seed can be specified. If the same seed is used on different runs of the program (or on different calls to the procedure), then identical sequences of pseudo-random numbers will be generated. If it is omitted, then a seed will be generated automatically.

The following script generates a set of five scenarios for a scenario variable, with values normally distributed around a central trend with a standard deviation of 5:

```
baseyear 2000
scenario years 2010 to 2040 by 10
dimension scenario 'run 1' 'run 2' 'run 3' 'run 4' 'run 5'
load _gauss_rand from 'IPATS_standard' as Random
summvar x, xtrend
summvar result{scenario}
xtrend.0 = 100
:: >> gr(1.0%) -> xtrend
```

```

for each scenario:
  call Random using x xtrend 5
  :: x -> result{scenario = ?}
:for

report result as "Scenario Variable"

```

Note that this example gets around the limitation of not allowing dimensions in procedure arguments by calling the procedure multiple times (using a `foreach` loop – see chapter 5) and assigning the return value (stored in variable `x`) to the result variable for different scenarios.

## The Loglet Procedure

The `loglet` procedure constructs a “loglet” composition – a superposition of logistic curves. The idea of loglets has been developed by Jesse Ausubel and Paul Waggoner.<sup>3</sup> Here is a sample script that uses the IPAT-S `loglet` procedure:

```

base year 2000
scenario years 2010 2020 2030
load _Loglet from 'IPATS_standard.dll' as Loglet
summv ar x
#
# To use the Loglet procedure, specify for each term:
#
#   max value, characteristic duration (in years), midpoint (year)
#
# An arbitrary number of terms can be introduced
#
call Loglet using x \
                100 25 2015 \
                150 15 2025 \
                30  5  2020
report x as "x"

```

## Linear Programming - LP blocks

Linear programming (LP) problems appear in many contexts, and there are many good books and web sites about linear programming (as well as quadratic programming, nonlinear optimization and other related topics). Only a minimal introduction to linear programming will be offered here. Nevertheless, after reading this section it should be possible to use the linear programming feature in IPAT-S for simple problems without further references. The kinds of problems for which the technique is appropriate should also be reasonably clear. Anyone wishing to read further will find a large literature.

Linear programming problems feature a goal of minimizing or maximizing some quantity (called the *objective function*) subject to constraints, where the objective function and constraints are linear expressions. For example, the problem of finding the least-cost food budget (the goal) that meets certain nutritional requirements (the constraints) can be cast in the form of an LP. In this example, the objective function is the cost of a particular combination of foods, and the goal is to minimize the cost.

Although conventional LP problems allow for only one objective function, there is a multi-objective extension of linear programming called *linear goal programming* (LGP). LGP is a technique for *multi-criteria analysis* (MCA), in which several incommensurable criteria or

<sup>3</sup> For information and software for performing loglet analyses, visit <http://phe.rockefeller.edu/>.

objectives must be taken into account. Since multi-criteria problems are common when developing policies for sustainable development, LGP is given a special focus in this section.

## Introduction to Linear Programming in IPAT-S

Linear programming calculations are built into IPAT-S using Michel Berkelaar's widely-used LP\_Solve library (ver 3.2). Like IPAT-S, LP\_Solve has been released under the GNU Lesser Public License.

In IPAT-S, a linear programming problem (LP) has the following syntax (where the use of square brackets indicates an optional element, and LP-specific commands are in bold>):

```
LP: [type]
  solve for var1, var2, etc.
  minimize or maximize Objective Function

  Constraint1 [dual: dualvar] [slack: slackvar]
  Constraint2 [dual: dualvar] [slack: slackvar]
  ...
:LP
```

Several elements of this must be explained, and they will be introduced through the use of an example. First, the LP problem is completely enclosed within `LP: [type] ... :LP`, also called an *LP block*. The `type` specifies whether the LP is to be solved only for scenario years, in which case it is `scenario`, or for all years, including the base year, in which case it is `all`. The word `scenario` can be abbreviated as `scen`, and `all` can be omitted. For example, to solve an LP year-by-year for all scenario years, but not the base year, you can use:

```
LP: scen
  ...
:LP
```

while to solve an LP year-by-year for all years, including the base year, you can use

```
LP:
  ...
:LP
```

Next, the line beginning `solve for` (or `solvefor`) lists the variables (summable variables or variables) that are to be solved by the LP. In the example of the low-cost food budget, the variable to solve for might be called `Diet`, with a dimension of `food`:

```
var Diet{food}
...
LP:
  solve for Diet
  ...
:LP
```

Any number of variables can be listed, optionally separated by commas.

Following the `solve for` line, the objective function is specified, as well as whether it is to be minimized or maximized. The keywords `minimize` and `maximize` can be abbreviated as `min` and `max`. In the example of the food budget, suppose that there is a variable `price`, with a dimension of `food`. Then the objective function could be added to the LP this way:

```
var Diet{food} price{food}
...
LP:
  solve for Diet
  minimize price * Diet
...
:LP
```

There are several points to note in this example:

- The variable `price` appears as a coefficient multiplying the variable `Diet`, which is to be solved for. In such expressions, the coefficient must always appear before the variable being solved for. Coefficients can be arbitrary expressions involving numbers, variables or summable variables.
- Each variable to be solved for can only appear in one term per constraint.
- In the objective function, the interpreter calculates total price, for each year. Although both `price` and `Diet` have `food` as a dimension, the result does not have a dimension, except for time, because there is only one objective function for each year. To reduce the dimensionality of the expression, after multiplying `price` and `Diet`, they are added up by the IPAT-S interpreter to give the expression for the objective function.

The last point is an example of a general feature of how IPAT-S handles dimensions within LPs, discussed further below.

The constraints follow the objective function. Constraints can include any of the variables to solve for, as well as *changes* in those variables (as discussed later in this section). In the case of the food budget, the constraints state that certain nutritional goals are to be met, such as a minimum level of caloric intake, minimum requirements for protein, micronutrients, etc. To introduce the constraint, a dimension `nutrient` is added to the script, along with two additional variables: `MinRequired`, which has a dimension of `nutrient`, and represents the minimum levels of intake for each nutrient; and `NutrContent`, which has dimensions of `food` and `nutrient`, and represents the nutritional content of each food for each nutrient. The LP might then appear as:

```
var Diet{food} price{food}
var MinRequired{nutrient} NutrContent{food, nutrient}
...
LP:
  solve for Diet
  minimize price * Diet
  NutrContent * Diet >= MinRequired
:LP
```

In this expression for the constraint, note that `MinRequired` has `nutrient` as a dimension, `Diet` has `food` as a dimension and `NutrContent` has both. In evaluating the expression, `NutrContent` is

first multiplied by `Diet` for each `food` and `nutrient`, the result is summed across `food` and finally is compared to `MinRequired`. This can be expressed mathematically by

$$\sum_{i \in \text{food}} (\text{NutrContent}_{ij} \times \text{Diet}_i) \geq \text{MinRequired}_j, \quad \forall j \in \text{nutrient}$$

Compared to the equivalent IPAT-S expression, it can be seen that without much effort on the user's part, IPAT-S has set up the calculations as desired: it calculates total nutritional supply across all foods before comparing to requirements, nutrient by nutrient.

Sometimes it is useful to calculate the *shadow price* of a constraint. This is the change in the objective function due to a change in the constraint. When the objective function calculates total cost, as it does here, the shadow price actually is a price – it is the implicit price of imposing the constraint, per unit value of the constraint. It is a standard theorem in the theory of linear programming that the shadow price is the value (or the negative of the value) of the dual variable of the constraint. To calculate the shadow prices of the nutrition constraints, add a variable to hold the information, and add the command `dual: varname` on the line with the constraint:

```
var Diet{food} price{food}
var MinRequired{nutrient} NutrContent{food, nutrient}
var ShadowPrice{nutrient}

...

LP:
  solve for Diet
  minimize price * Diet
  NutrContent * Diet >= MinRequired  dual: ShadowPrice
:LP
```

The shadow price in this case gives the implicit cost of maintaining a healthy diet, given current food prices. The cost is estimated for each nutrient, and could be used to determine subsidies for particular foods or for nutritional supplements.

A similar syntax can be used to calculate the slack variables in the LP problem, or the difference between the calculated value of the left-hand side of a constraint and the target value on the right-hand side. This is accomplished with the `slack:` keyword:

```
var Diet{food} price{food}
var MinRequired{nutrient} NutrContent{food, nutrient}
var ShadowPrice{nutrient} slack{nutrient}

...

LP:
  solve for Diet
  minimize price * Diet
  NutrContent * Diet >= MinRequired  dual: ShadowPrice slack: slack
:LP
```

The slack variable can be used to find out which are the most constraining of the constraints when exploring the implications of an LP calculation.

The text within `LP: ... :LP`, above, is a complete specification of an LP problem in IPAT-S. If a key variable, such as `price`, were to change between scenario years, then the dietary composition of a minimal food budget, and its total cost, will also change, generating a quantitative scenario for the cost of a minimally nutritious food budget under a condition of changing food prices.

This example shows how compact the IPAT-S notation can be for LP problems. Although only one variable name (`Diet`) appears in the objective function, there is actually one variable per `food`, since `Diet` has `food` as a dimension. Also, although only three lines of text express the constraints, there are as many constraints as there are `nutrients`. Dimension matching and summation are taken care of by the IPAT-S interpreter.

While there are some limitations to building LPs in IPAT-S, the limitations can be overcome with modest effort on the part of the user, and a large variety of LP problems can be set up and solved in IPAT-S. Some examples will be given here, while others can be found in the IPAT-S distribution. Before discussing how to get around these limitations, a special extension of the LP technique is discussed – linear goal programming.

## Introduction to Linear Goal Programming

The example above demonstrated use of an LP to solve an allocation problem – how to allocate food in diets so that minimal nutritional requirements are met. There are most likely many possible diets that can meet the nutritional requirements, so some principle must be invoked in order to distinguish between the possibilities. The principle used above is an economic one: choose the least-cost food budget that meets the requirements. However, while this is an interesting question, and has some policy relevance – for example, for setting poverty lines – it can be problematic when creating scenarios, and for addressing sustainability problems. The difficulty with scenario development is that people take many things into account when buying food besides price, such as familiarity or taste, and often do not consider some nutritional components. In principle, these considerations can also be introduced as “costs” or “benefits,” but in practice this approach is problematic. A further difficulty that arises in sustainability analysis is that many competing criteria might reasonably apply when making a decision. What alternative principle might be invoked?

An approach that can be used in many instances in quantitative scenario work and especially for sustainability issues is to look at departures from some standard pattern, using a linear goal programming (LGP) approach.<sup>4</sup> In the example of the food budget, suppose that a nutritional education campaign is introduced that is expected to lead people to pay more attention to certain nutrients in their diet. They will adjust their diets so that they get minimum levels of those nutrients. However, they do not want to change what they are currently doing significantly, so there will be some resistance to change. This is an example of a scenario narrative, which can be analyzed quantitatively using an LGP approach. The approach will be illustrated with an example.

The example will be the problem outlined above: after the base year, people are expected to adjust their diet to meet nutritional goals, but will try to remain as close to their current diet as

---

4 The method presented in this section can be described as a *non-preemptive minimax* linear goal programming approach. It is just one variation on linear goal programming. Readers interested in more general LGP approaches can refer to the brief introduction to general LGP, “Beyond Non-Preemptive Minimax Linear Goal Programming,” later in this section.

possible. To implement this approach, a variable `CurrDiet` is introduced, with dimension `food`, in addition to the variable `Diet` that was introduced above:

```
var Diet{food} CurrDiet{food}
...
LP: scen
...
:LP
```

It is assumed that `CurrDiet` is assigned values so that it is equal to the base year values for `Diet` in all scenario years. That is, `CurrDiet` in the scenario is what the diet would be if it remained at base year levels for all scenario years. The LP is indicated as being type `scen`, in this case, because the LP should only be applied in scenario years.

Next, two variables are introduced, `deltaPlus` and `deltaMinus`, which (for this application) have no dimensions.

```
var Diet{food} CurrDiet{food}
var deltaPlus deltaMinus
...
LP: scen
...
:LP
```

The variables to solve for are `Diet`, `deltaPlus` and `deltaMinus`, and the goal is to minimize the total of `deltaPlus` and `deltaMinus`:

```
var Diet{food} CurrDiet{food}
var deltaPlus deltaMinus
...
LP: scen
  solve for Diet, deltaPlus, deltaMinus
  min deltaPlus + deltaMinus
...
:LP
```

Next, the constraints are introduced. One constraint is the same as in the example above – it uses the variables `NutrContent` and `MinRequired`, and demands that in scenarios the nutritional goals are met:

```
var Diet{food} CurrDiet{food}
var deltaPlus deltaMinus
var MinRequired{nutrient} NutrContent{food, nutrient}
...
LP: scen
  solve for Diet, deltaPlus, deltaMinus
  min deltaPlus + deltaMinus
  NutrContent * Diet >= MinRequired
...
:LP
```

In addition, a new pair of constraints implement the LGP approach:

```

var Diet{food} CurrDiet{food}
var deltaPlus deltaMinus
var MinRequired{nutrient} NutrContent{food, nutrient}

...

LP: scen
  solve for Diet, deltaPlus, deltaMinus
  min deltaPlus + deltaMinus
  NutrContent * Diet >= MinRequired

  # LGP constraints
  Diet + deltaPlus >= CurrDiet
  Diet - deltaMinus <= CurrDiet
:LP

```

This is the full specification of the LP for the LGP problem. What does it do? First, suppose that the base year diet meets all nutritional requirements. Then `deltaPlus` and `deltaMinus` can be set to as small a value as required; that is, they can be zero. When `deltaPlus` and `deltaMinus` are zero, the LGP constraints become

```

Diet >= CurrDiet
Diet <= CurrDiet

```

But there is only one way that `Diet` can be both greater than or equal to `CurrDiet` and less than or equal to `CurrDiet`, and that is if `Diet` is equal to `CurrDiet`. This gives the correct result: if the base year diet is nutritionally sufficient, then the scenario diet that is as close to the base year diet as possible is the base year diet itself.

Next, suppose the base year diet does not meet nutritional requirements. In this case, `Diet` cannot equal `CurrDiet`, and either one or both of `deltaPlus` and `deltaMinus` will be greater than zero. The two LGP constraints then open up a window of possible values for the difference between `Diet` and `CurrDiet`. The goal of minimizing the total of `deltaPlus` and `deltaMinus` ensures that the window will be as small as possible – that is, that the diet in the scenario will be as close to the current diet as possible while meeting the nutritional requirements. This was the situation described in the scenario narrative, so this LP is a quantitative implementation of that narrative.

## A side-note on “numbers with dimensions”

In the example above, `deltaPlus` and `deltaMinus` have no dimensions, while `Diet` and `CurrDiet` have the dimension “food.” In order to match up the two sides of the LGP constraints, `deltaPlus` is added to each food in `Diet`, and `deltaMinus` is subtracted from each food in `Diet` before comparing to the current diet.

Another way to do this is to use a special notation available in LP blocks:

```

Diet + 1{food} * deltaPlus >= CurrDiet
Diet - 1{food} * deltaMinus <= CurrDiet

```

The notation `1{food}` creates a number with a dimension. It says that in all scenario years there should be a coefficient of 1 for each `food`.

While the `number{dimension}` notation is not required in the example above, because of IPAT-S’s built-in dimension-matching rules, there are situations where it might be required. For example, suppose that there is a variable `FuelUse`, with dimensions of sector and fuel, which should be summed to give total fuel use by fuel in the variable `TFD`, representing total final fuel demand.

Outside of an LP, this would be accomplished using a summarize-as statement. Inside an LP it is accomplished using a constraint:

```
var FuelUse{sector, fuel} TFD{fuel}
LP:
...
FuelUse - TFD = 0{fuel}
...
:LP
```

This constraint says that for each fuel, the sum of `FuelUse` over sectors minus `TFD` for that fuel should be zero: that is, `TFD` should be the sum of `FuelUse` across sectors. If the constraint were written with 0 (zero) on the right-hand side, rather than `0{fuel}`, then total `FuelUse` across all fuels and sectors would be equal to the total of `TFD` over all fuels, but they would not be equal for each fuel.

## Using Weights in Linear Goal Programming

In the diet example above, the window of values opened up by the LGP constraints is the same for all foods. This suggests that all foods are equally likely to be substituted in order to meet the nutritional goals. However, that might not be the case. A particularly nutritious food might be universally disliked or difficult to prepare. People will substitute it if they have to, but will preferentially substitute other foods first. Another possibility is that some foods might not be substitutable. For example, consumption of a staple grain may vary only over a very small range. In order to meet the nutritional requirements, people will add or remove other foods to or from their total diet, but the level of consumption of the staple grain – as bread, fermented beverage, etc. – is not negotiable. This situation can also be accommodated within the LGP approach. In the dietary example being used here, the problem can be addressed by adding a variable `weight`, with dimension `food`. The weight variable is then multiplied by the `deltaPlus` and `deltaMinus` variables:

```
var Diet{food} CurrDiet{food}
var deltaPlus deltaMinus weight{food}
var MinRequired{nutrient} NutrContent{food, nutrient}
...
LP: scen
  solve for Diet, deltaPlus, deltaMinus
  min deltaPlus + deltaMinus
  NutrContent * Diet >= MinRequired
  # LGP constraints
  Diet + weight * deltaPlus >= CurrDiet
  Diet - weight * deltaMinus <= CurrDiet
:LP
```

Suppose, for example, that there is a food called `grain`, and the weight for `grain` is zero. Then in the implementation of the two LGP constraints, when the IPAT-S interpreter expands the constraints in the LP block for all dimensions, it will create one pair of inequalities of the form:

```
Diet{food = 'grain'} + weight{food = 'grain'} * \
  deltaPlus{food = 'grain'} >= CurrDiet{food = 'grain'}
Diet{food = 'grain'} - weight{food = 'grain'} * \
```

```
deltaMinus{food = 'grain'} <= CurrDiet{food = 'grain'}
```

but because the weight is zero in this case, these inequalities become

```
Diet{food = 'grain '} >= CurrDiet{food = 'grain '}
Diet{food = 'grain '} <= CurrDiet{food = 'grain '}
```

The only way these can both be satisfied is if Diet for grain is equal to CurrDiet for grain. That is, because the weight is zero, there is no flexibility in changing the consumption of grain. The lower the weight, the less flexibility in introducing that food, and the higher the weight, the more flexibility.

A special case of a weight is when the weight is proportional to some quantity. Suppose that instead of having no flexibility in the use of the staple grain, as in the example above, that the flexibility of adjusting dietary composition is proportional to the consumption of that food: if a food is not eaten at all, then it is not going to be introduced now, even to meet nutritional requirements; if a food is a major component in diets, then people are willing to add to or subtract from their consumption of that food significantly in order to meet their nutritional needs. In this case, rather than introducing a new variable weight, the variable CurrDiet can be used as the weight:

```
var Diet{food} CurrDiet{food}
var deltaPlus deltaMinus
var MinRequired{nutrient} NutrContent{food, nutrient}
...
LP: scen
  solve for Diet, deltaPlus, deltaMinus
  min deltaPlus + deltaMinus
  NutrContent * Diet >= MinRequired

  # LGP constraints
  Diet + CurrDiet * deltaPlus >= CurrDiet
  Diet - CurrDiet * deltaMinus <= CurrDiet
:LP
```

With this structure for the LP, the window of possible values is proportional to the CurrDiet. In the absence of other information, this can often be a useful assumption for default weights.

Examples of LGP problems are provided in the sample scripts that are part of the IPAT-S distribution.

## Beyond Non-Preemptive Minimax Linear Goal Programming

This section contains a brief overview of general LGP problems for interested readers. The presentation is somewhat more compressed and technical than in the discussion above.

A linear goal program can be expressed (somewhat schematically) in the following way:

$$\begin{aligned} &\text{first min } \sum_i \mu_i^{(1)} \delta_i^+ + \nu_i^{(1)} \delta_i^- \\ &\text{then min } \sum_i \mu_i^{(2)} \delta_i^+ + \nu_i^{(2)} \delta_i^- \\ &\dots \\ &\text{finally min } \sum_i \mu_i^{(N)} \delta_i^+ + \nu_i^{(N)} \delta_i^- \end{aligned}$$

subject to conventional LP constraints

$$\sum_j a_{ij} x_j \leq = > C_i$$

and goal constraints

$$\text{Type I: } \sum_j b_{ij} x_j + \delta_i^+ - \delta_i^- = S_i$$

$$\text{Type II: } \sum_j b'_{ij} x_j + \delta_i^+ = S'_i$$

$$\text{Type III: } \sum_j b''_{ij} x_j - \delta_i^- = S''_i$$

The variables  $\delta_i^+$  and  $\delta_i^-$  are called *deviational variables*. The goal is to minimize a weighted sum of deviations from a set of *aspiration levels*  $S_i, S'_i, S''_i$  – this is called a *minsum* problem. As shown above, it is possible for there to be several objective functions that are solved one after the other. The earlier the goals are applied, the higher the *priority* they are said to have. (After each is minimized, it is constrained to equal its minimum value as later constraints are solved.) If there is more than one objective function, the problem is said to be *preemptive* (since earlier goals are given absolute preference to later goals, and are said to “preempt” them), and is said to have *lexicographic minsum* goals, since goals are satisfied in order, analogous to the way that words are ordered lexicographically by comparing one letter at a time.

There are three types of goal constraint – Type I, in which both the  $\delta_i^+$  and  $\delta_i^-$  appear, Type II, in which only the  $\delta_i^+$  appear, and Type III, in which only the  $\delta_i^-$  appear. In Type I constraints, it is desired that the expression involving the problem variables (the  $x_j$ ) be exactly equal to the aspiration level  $S_i$ , but if the value has to be higher or lower than the aspiration level it is still accepted as a solution. In Type II constraints, the expression involving the problem variables must not exceed the aspiration value  $S'_i$ , while in Type III constraints the expression involving the problem variables must not be less than the aspiration value  $S''_i$ .<sup>5</sup>

In addition to the goal constraints, conventional LP constraints can also be introduced. (In the usual exposition of the LGP technique, conventional LP constraints are *not* introduced, because they can in principle always be expressed as goal constraints for which the deviational variables do not enter the objective function. However, in practice it is convenient to use the mix of conventions.)

To demonstrate how a preemptive linear goal program can be implemented in IPAT-S, an example from James P. Ignizio’s *Introduction to Linear Goal Programming* will be presented.<sup>6</sup> The

5 In the LGP models introduced earlier in this section, the deviational variables had fewer dimensions than did the aspiration levels. Minimizing the deviational variables then minimized the maximum deviation away from any given aspiration level. For this reason the models were of the *minimax* type, rather than minsum.

6 Ignizio, James P. 1985. *Introduction to Linear Goal Programming*. Newbury Park: Sage Publications. A Sage University Paper, No. 56 in the series Quantitative Applications in the Social Sciences.

problem itself will not be motivated here. Instead, its form will be given and then the IPAT-S script that solves it presented. The problem has the following goals:

first min  $\delta_1^- + \delta_2^-$   
 then min  $\delta_3^+$   
 then min  $\delta_4^-$   
 finally min  $\delta_1^+ + 1.5 \delta_2^+$

subject to the constraints

$$x_1 + \delta_1^+ - \delta_1^- = 30$$

$$x_2 + \delta_2^+ - \delta_2^- = 15$$

$$8x_1 + 12x_2 + \delta_3^+ - \delta_3^- = 1000$$

$$x_1 + 2x_2 + \delta_4^+ - \delta_4^- = 40$$

and

$$x_i, \delta_i^{\pm} \geq 0$$

The strategy in IPAT-S is to loop over constraint priorities using a `foreach` loop. As each priority is introduced in turn, the corresponding goal is "turned on," by setting a filter `currPriority` for the priority to 1. Also, after each goal is satisfied, it is constrained to equal the value that was found by introducing a filter `wasApplied` that is set to 1 for every goal that is solved. The IPAT-S script is shown here without discussion. It is more involved than the earlier examples, but the connection between the problem statement above and the LP block should be reasonably clear.

```
base year 2000
dim item "1" "2"
dim constraint "1" "2" "3" "4"
dim priority "1" "2" "3" "4"
dim direction "+" "-"

sumvvar Production{item}

var GoalCoeff{priority, constraint, direction}
var AspLevel{constraint} ProdCoeff{constraint, item}
var wasApplied{priority} currPriority{priority} goalValue{priority}

var delta{constraint, direction} deviationDirection{direction}
deviationDirection{direction = "+"} = 1
deviationDirection{direction = "-"} = -1

##
## Goals
##
# Priority 1 goal
ditto priority = "1":
GoalCoeff{',', constraint = "1", direction = "-"} = 1
GoalCoeff{',', constraint = "2", direction = "-"} = 1
# Priority 2 goal
ditto priority = "2":
GoalCoeff{',', constraint = "3", direction = "+"} = 1
# Priority 3 goal
ditto priority = "3":
GoalCoeff{',', constraint = "4", direction = "-"} = 1
# Priority 4 goal
ditto priority = "4":
```

```

GoalCoeff{'', constraint = "1", direction = "+"} = 1
GoalCoeff{'', constraint = "2", direction = "+"} = 1.5

##
## Constraints & Aspiration Levels
##
ditto constraint = "1":
AspLevel{''} = 30
ProdCoeff{'', item = "1"} = 1
ditto constraint = "2":
AspLevel{''} = 15
ProdCoeff{'', item = "2"} = 1
ditto constraint = "3":
AspLevel{''} = 1000
ProdCoeff{'', item = "1"} = 8
ProdCoeff{'', item = "2"} = 12
ditto constraint = "4":
AspLevel{''} = 40
ProdCoeff{'', item = "1"} = 1
ProdCoeff{'', item = "2"} = 2

foreach priority:
  # Turn on current priority
  currPriority{priority = ?} = 1
  LP:
    solve for Production delta
    min currPriority * GoalCoeff * delta
    # Already solved - maintain values
    wasApplied * GoalCoeff * delta = wasApplied * goalValue
    # Constraints
    ProdCoeff * Production + deviationDirection * delta = AspLevel
  :LP
  :: GoalCoeff(priority = ?) * delta -> goalValue(priority = ?)
  # Turn off current priority
  currPriority{priority = ?} = 0
  # Yes, this one has been applied
  wasApplied{priority = ?} = 1
:foreach

report Production as "Production (units/day)"
report -(deviationDirection * delta){direction = sum} \
  as "Difference from aspiration level"

```

When this script is run, the result is equal to the result given in Ignizio's book. The following table was created by copying the values from the Indicator tab in the IPAT Studio IDE:

	<i>item</i>	<i>constraint</i>	2000
Production (units/day)	1		30
Production (units/day)	2		15
Difference from aspiration level		1	0
Difference from aspiration level		2	0
Difference from aspiration level		3	-580
Difference from aspiration level		4	20

## Implementing an I-O Problem Using an LP

Linear programming models are linear systems of constraints with a linear objective function. Input-Output (I-O) problems are linear systems of equations without an objective function. In contrast to linear programming problems, where the solution to the linear equations is not unique (so an objective function is required), the solution to an I-O problem is unique, and can be solved using standard matrix calculations.

IPAT-S does not offer a syntax for doing matrix calculations directly. However, it is possible to carry out I-O or other matrix calculations using an LP block. The idea is that in this case it does not matter what the objective function is. Because the solution is unique, there is no flexibility, so the objective function – whatever it is – can only have one, unique value.

As with LP problems, there are many good sources already available on I-O, so the I-O methodology will not be discussed here. Instead, a very simple I-O - like model will be presented, implemented using an LP block. Although the model presented here is simplified, users who want to implement an I-O calculation in IPAT-S can use this example as a starting point.

The script to be presented below models a three-sector economy – agriculture, industry and services – with an added labor sector to represent the supply of labor to the economic sectors. Total final demand for goods from each sector drives overall economic production, based on inter-sector demand for intermediate goods. Total production levels in each sector drive sectoral employment, modified by changes in sector productivity.

Two scenarios are presented – Business as Usual (BaU) and Greening Industry. The Greening Industry scenario is almost identical to BaU. However, it features increased demand from industry for agricultural products, reflecting a shift away from non-renewable resources as industrial feedstocks toward biomass-based feedstocks.

The basic I-O equation expresses a simple idea: that before economic production sectors can provide for final demand, they must first supply other economic sectors with intermediate goods and services. This is expressed in the following way. Output from each sector is represented by a vector  $X$ , with elements  $X_i$ , where  $i$  represents an economic sector. For each dollar of output, sector  $j$  pays a total of  $A_{ij}$  to sector  $i$ . Total final demand for products from sector  $i$  is represented by the vector  $Y_i$ . So, of total output  $X_i$  from sector  $i$ , part is consumed by other sectors, and the remainder is consumed as final demand. That is,

$$X_i - \sum_j A_{ij} X_j = Y_i \quad .$$

Using matrix notation, this equation can be written

$$X - AX = Y$$

or

$$(I - A)X = Y \quad ,$$

where  $I$  is the identity matrix. Given an exogenous final demand vector  $Y$  and a matrix  $A$  (the “technical coefficients”), this equation can be solved for  $X$  by calculating an inverse matrix:

$$X = (I - A)^{-1} Y .$$

This is the conventional I-O calculation. In the IPAT-S script presented here, the LP block takes the following form, with line numbers added for reference:

```

1) LP:
2) solve for Output, IntermedOutput, IntermedDemand, dummy
3) min dummy # Not really needed
4)
5) Output - IntermedDemand = TFD
6) Output - ident * IntermedOutput = 0{supplySector, scenario}
7) IntermedDemand - a * IntermedOutput = 0{supplySector, scenario}
8) :LP

```

In this script fragment, the basic I-O calculation can be seen in line 5: Output less purchases for intermediate goods (called `IntermedDemand` here) is equal to total final demand (TFD). In line 7, `IntermedDemand` is set equal to `a` multiplied by `IntermedOutput` – that is, it is equal to  $AX$  in the standard I-O formulation. This is all part of the conventional I-O calculation. The new elements are in lines 3 and 6.

In line 3, the objective of the LP is set to minimize a new variable, `dummy`. As the name implies, `dummy` is just a “dummy variable” – it plays no real role in the calculations. All it does is provide a variable to put in the objective function. Since the equations have a unique solution, it does not matter what the objective function is.

In line 6, `Output` is set equal to `ident * IntermedOutput`. This line gets around a limitation of the IPAT-S syntax. In IPAT-S, dimensions (which play the role of indices for matrices) can appear in any order. However, in general, matrix indices cannot appear in any order:  $A_{ij}$  is not the same as  $A_{ji}$ . Ordinarily this is not a limitation in IPAT-S – it is a feature! But in this case it complicates the calculations. To get around this problem, two identical dimensions are defined, using IPAT-S’s `ditto` keyword for convenience:

```

ditto 'Agriculture', 'Industry', 'Services', 'Labor':
dimension supplySector ''
dimension demandSector ''

```

Then the dimensions are identified using the `ident` variable:

```

var ident{supplySector, demandSector}
foreach supplySector demandSector:
  if $supplySector .eq. $demandSector then:
    ident{demandSector = ?, supplySector = ?} = 1
  :if
:foreach

```

Line 6 in the LP block then enforces the condition that the two indices are referring to the same sectors, and that `Output` is the same as `IntermedOutput` for a given sector.

Here is the complete script:

```

comment:
This IPAT-S script demonstrates the implementation of an I-O problem
using an IPAT-S LP block.

Two scenarios are presented, "Business-as-Usual" (BaU) and
"Greening Industry." The only difference between them is that

```

```

under "Greening Industry" there is more demand per dollar for
agricultural goods from industry than in BaU. For example, more
industrial feedstocks might be provided from agriculture than
from mining & manufacturing.
:comment

baseyear 2000
scenario years 2010 2020

dimension scenario 'BaU', 'Greening Industry'

# In IPAT-S, there is no notion of an "order" to array indices. So when
# there is an array like the I-O "A" array, it is necessary to define
# two copies of the indices. Later in the script these are identified
# using the "ident" variable.
ditto 'Agriculture', 'Industry', 'Services', 'Labor':
dimension supplySector ''
dimension demandSector ''

# Exogenous variables
var a{supplySector, demandSector, scenario}
ratio laborProductivity{demandSector, scenario}
ratio DemandGrowth{supplySector, scenario}

# Endogenous variables
sumvar TFD{supplySector, scenario}
sumvar Employment{demandSector, scenario}
sumvar IntermedOutput{demandSector, scenario}, \
    IntermedDemand{supplySector, scenario}, \
    Output{supplySector, scenario}

##
## Technical coefficients
##
# Agriculture
ditto demandSector = 'Agriculture', scenario = 'BaU':
a{'', supplySector = 'Agriculture'} = <0.05>
a{'', supplySector = 'Industry'} = <0.15>
a{'', supplySector = 'Services'} = <0.30>
a{'', supplySector = 'Labor'} = <0.35>

ditto demandSector = 'Agriculture', scenario = 'Greening Industry':
a{'', supplySector = 'Agriculture'} = <0.05>
a{'', supplySector = 'Industry'} = <0.15>
a{'', supplySector = 'Services'} = <0.30>
a{'', supplySector = 'Labor'} = <0.35>

# Industry
ditto demandSector = 'Industry', scenario = 'BaU':
a{'', supplySector = 'Agriculture'} = <0.02>
a{'', supplySector = 'Industry'} = <0.60>
a{'', supplySector = 'Services'} = <0.15>
a{'', supplySector = 'Labor'} = <0.07>

ditto demandSector = 'Industry', scenario = 'Greening Industry':
a{'', supplySector = 'Agriculture'} = <0.02, 0.05, 0.10>
a{'', supplySector = 'Industry'} = <0.60, 0.57, 0.52>
a{'', supplySector = 'Services'} = <0.15>
a{'', supplySector = 'Labor'} = <0.07>

# Services
ditto demandSector = 'Services', scenario = 'BaU':
a{'', supplySector = 'Agriculture'} = <0.05>
a{'', supplySector = 'Industry'} = <0.10>
a{'', supplySector = 'Services'} = <0.15>
a{'', supplySector = 'Labor'} = <0.50>

ditto demandSector = 'Services', scenario = 'Greening Industry':
a{'', supplySector = 'Agriculture'} = <0.05>
a{'', supplySector = 'Industry'} = <0.10>
a{'', supplySector = 'Services'} = <0.15>
a{'', supplySector = 'Labor'} = <0.50>

##
## Demand vector
##
ditto supplySector = 'Agriculture':
TFD.0{''} = 150

```

```

DemandGrowth('', scenario = 'BaU') = growth(<1.0%>)
DemandGrowth('', scenario = 'Greening Industry') = growth(<1.0%>)

ditto supplySector = 'Industry':
TFD.0{''} = 2045
DemandGrowth('', scenario = 'BaU') = growth(<1.5%>)
DemandGrowth('', scenario = 'Greening Industry') = growth(<1.5%>)

ditto supplySector = 'Services':
TFD.0{''} = 2100
DemandGrowth('', scenario = 'BaU') = growth(<2.0%>)
DemandGrowth('', scenario = 'Greening Industry') = growth(<2.0%>)

# There is no final demand for 'Labor'
:: >> DemandGrowth -> TFD

##
## Identify supply & demand sectors
##
var ident(supplySector, demandSector)
foreach supplySector demandSector:
  if $supplySector .eq. $demandSector then:
    ident(demandSector = ?, supplySector = ?) = 1
  :if
:foreach

##
## Implement IO as an LP
##
var dummy

LP:
solve for Output, IntermedOutput, IntermedDemand, dummy
min dummy # Not really needed

Output - IntermedDemand = TFD
Output - ident * IntermedOutput = 0{supplySector, scenario}
IntermedDemand - a * IntermedOutput = 0{supplySector, scenario}
:LP

##
## Employment
##
ditto demandSector = 'Agriculture':
Employment.0{''} = 500
laborProductivity{'', scenario = 'BaU'} = growth(<0.5%>)
laborProductivity{'', scenario = 'Greening Industry'} = growth(<0.5%>)

ditto demandSector = 'Industry':
Employment.0{''} = 3000
laborProductivity{'', scenario = 'BaU'} = growth(<1.5%>)
laborProductivity{'', scenario = 'Greening Industry'} = growth(<1.5%>)

ditto demandSector = 'Services':
Employment.0{''} = 5000
laborProductivity{'', scenario = 'BaU'} = growth(<0.75%>)
laborProductivity{'', scenario = 'Greening Industry'} = growth(<0.75%>)

:: a{supplySector = 'Labor'} * IntermedOutput >> \
    1/laborProductivity -> Employment

##
## Report results
##
report TFD as "Total Final Demand"
report Employment as "Employment"
report 100 * Output/Output.0 as "Output Index"

```

There are some advantages to putting an I-O problem into an LP format. It is easy to introduce constraints into this basic problem, which would then lead to an LP problem, rather than a matrix calculation. For example, part of total final demand might be flexible – e.g., traded goods. In this case, exports and imports could appear as variables to solve for, and constraints and goals could

be added. Alternatively, domestic consumption might be flexible, and might adjust to meet certain social or environmental goals. In this case, an LGP approach might be appropriate.

## Constraining Changes in Solution Variables

In some cases it is useful to constrain the change in a variable from one scenario year to the next, rather than its value in a particular year. For example, in some scenario calculations it might be desirable to limit the increase in an environmental impact, substitution of renewable energy, etc. IPAT-S offers a syntax for such constraints for when an LP is calculated for scenario years, change (variable). For example, the constraint

```
LP: scenario
    ...
    change(x) > 0
    ...
:LP
```

will ensure that the variable  $x$  always increases in the scenario.

## Getting around the limitations of LPs in IPAT-S

There are two major limitations of the LP implementation in IPAT-S. The first of these is that variables cannot be negative. This is a standard limitation in linear programming theory and techniques have been developed to compensate for it. The second is that in some cases it might be desirable to restrict the action of an LP only to specific dimensions, and that is generally not possible within the LP syntax itself. Both of these problems can be circumvented in IPAT-S.

First, as an example of a situation where a variable might become negative, consider the case of a trade balance, implemented as part of an LP. The key constraint might be expressed as

```
LP:
    solve for DomProduction, NetImports, ...
    ...
    DomProduction + NetImports = DomRequirements
    ...
:LP
```

In this expression it is assumed that domestic requirements (`DomRequirements`) are specified before the LP is calculated. The difficulty is that `NetImports` can realistically be either positive (if the region is an importer) or negative (if the region is an exporter). However, as a solution variable, `NetImports` cannot be negative. If other constraints (for example, a minimum level of exports) lead `DomProduction` to be greater than `DomRequirements` then the LP will become infeasible.

The solution to this problem is to introduce two variables in place of `NetImports`: `Exports` and `Imports`. Each of the new variables is always positive. The LP should then be rewritten

```
LP:
    solve for DomProduction, Exports, Imports, ...
```

```

...
DomProduction + Imports - Exports = DomRequirements
...
:LP

```

This example will behave as expected.

The second situation is one where the LP calculations should only be applied to a subset of dimensions. For example, suppose that the LP is solving for energy production, but production from hydroelectric plants is constrained by the available resources, and cannot change. In this case, the solution is to apply a filter to the energy production variable, and other relevant variables as needed. Suppose that energy production is represented by the variable `EnergyProd`, with dimension `fuel`. Then define a variable `exclude`, also with dimension `fuel`, and introduce it in the LP problem in the following way:

```

var EnergyProd{fuel} FixedProd{fuel}
var exclude{fuel}
exclude{fuel = 'hydro'} = 1
LP:
solve for EnergyProd, ...
...
exclude * EnergyProd = FixedProd
(1{fuel} - exclude) * EnergyProd ...
...
:LP

```

Most of the details have been omitted from this example, as indicated by the prolific use of ellipses (...). However, the technique is extremely general, so it is appropriate that the example should not include too many irrelevant details. Taking the script fragment line by line, in the first two lines the variable `EnergyProd` and `exclude` are both introduced, each with dimension `fuel`. Also, a variable `FixedProd` is introduced, which will be zero for all fuels except `hydro`, and equal to the desired value for hydropower. Then in the next line, the fuel `hydro` is marked as being excluded from the calculations, by setting the `exclude` variable for that dimension to 1. All other values for `exclude` will be zero by default. The next two lines initiate the LP calculations, and show that `EnergyProd` is a variable to solve for. After the ellipses, the line `exclude * EnergyProd = FixedProd` ensures that the production of hydropower will be set at its fixed value. The next line, `(1{fuel} - exclude) * EnergyProd ...`, is a fragment of a constraint. The coefficient `(1{fuel} - exclude)` is 0 for the fuel `hydro` and 1 for everything else, because it is equal to 1 minus each of the entries in `exclude`. Because of this, the entry for `fuel = 'hydro'` for `EnergyProd` will not enter into the calculations, as desired.

One caution with this approach to restricting dimensions: it is easy to accidentally make an infeasible LP, even when the intended LP problem would be feasible. For example, suppose in the example above the full constraint expression is

```
(1{fuel} - exclude) * EnergyProd >= MinEnergyProd
```

where `MinEnergyProd` has `fuel` as a dimension. Then the inequality for `fuel = 'hydro'` will be

```
(1- exclude{fuel = 'hydro'}) * EnergyProd{fuel = 'hydro'} >= \
MinEnergyProd{fuel = 'hydro'}
```

Because `exclude{fuel = 'hydro'}` is equal to 1, the expression on the left-hand side of this inequality is zero – that is why hydropower production is effectively excluded from the LP calculations. The inequality becomes

```
0 >= MinEnergyProd{fuel = 'hydro'}
```

This is fine as long as `MinEnergyProd{fuel = 'hydro'}` is less than or equal to zero. However, if it is positive, then the LP will be infeasible. If dimensions are being excluded in an LP using a filter, as in the example above, then it is important to make sure that the inequalities for the excluded dimension are all consistent.

## Return values from LPs

By default, if the LP solver encounters a problem, the IPAT-S interpreter reports it, sending it to standard error output. However, in some cases it is desirable to catch the return conditions from the LP within the script. For example, if an LP is infeasible, it *may* mean that the script writer made an error when setting up the problem, but it might also mean that the user's goals really cannot be met. In the first problem reporting an infeasible LP as an error is appropriate, while in the second the script writer may want to have the script branch, and do one set of calculations if the LP is feasible, and another set if it is infeasible. (Note that duals, slacks and values are set before returning from the LP block even if there is an error, so the values for these variables can be used to diagnose problems.)

To handle LP return values within the script, do one of the following:

- Add the command-line switch “-l n” when invoking the interpreter.
- Set the internal flag `_LP_REPORT_` to `false`.

After an LP block, the values for two internal constants, `_LP_STATUS_` and `_LP_YEAR_`, will be set. (They are “constant” because they cannot be set within the script – only by the interpreter.)

`_LP_STATUS_` is set equal to one of `_LP_OK_`, `_LP_INFEASIBLE_`, `_LP_UNBOUNDED_`, or `_LP_UNKNOWN_ERROR_`. Then `_LP_YEAR_` is set to the year in which the error occurred, or to the final year, if the LP finished with `_LP_OK_`.

## 5 Output

---

### *Report Statements*

The simplest way to report the results from a script is to use a statement like

```
report expression as "label"
```

For example,

```
report 1000 * GDP/Pop as "Income ($PPP/capita)"
```

Summable variables, variables, number variables, ratio variables, and numbers can appear in the expression. The label can contain any text other than a quotation mark, as long as it is all on one line. Like index expressions, report expressions can combine variables by multiplication ( \* ), division ( / ), addition ( + ), subtraction ( - ) and exponentiation ( ^ ), grouping with parentheses, and by using functions.

A special feature of report expressions is that the .0, .by, .2 and other ways of specifying the base year and scenario years can be applied to whole groups of calculations. For example, the following expression can be used to calculate a productivity index by sector:

```
report 100 * (VA/Empl) / (VA/Empl).2000 as "Productivity (2000=100)"
```

Another special feature of report statements is that ratio variables can be inserted into expressions. (This is not the case generally with variable or summable variable expressions.) Ratio variables are given the value 1.0 in the base year, and are otherwise reported as indices. For example, suppose `income` is a ratio variable. Then,

```
ratio income = gr(2.00%, 1.50%, 1.25%)
...
report 100 * income as "Income (2000 = 100)"
```

will report `income` as an index.

### *Print blocks*

Report statements are very useful, but limited. They are most useful for quickly reviewing the output from a scenario, or for “dumping” data to be formatted in another program, such as a spreadsheet program. In contrast, print blocks take more time to set up, but are much more flexible. A wide variety of outputs can be produced using print blocks.

### **Basic format of a print block**

Print blocks are set off from the rest of the script using the syntax `print: ... ;print`. Everything within the block is printed verbatim, except for in-line calculations. For example,

```
print:
Please print this as it is.
;print
```

will produce the output “Please print this as it is.”

A variant on `print:` is to use the `print line:` modifier. It can be written either `printline` or `print line`. When `print line:` is used, carriage returns are stripped from the text. This can make the script look more clean, while producing the desired output. For example, to make a tab-delimited line of output, use code like

```
print line:
    First item
    Second item
    Third item
:print
# This will put a carriage return at the end of the line
print:

:print
```

This will produce the following output:

```
First item    Second item    Third item
```

The same result could be accomplished with a single `print` block,

```
print: First item    Second item    Third item
:print
```

but it is not as clear in this case that the items are separated by tabs.

## In-line calculations

In-line calculations allow scenario results to be embedded in printed output. In-line calculations are set off using square brackets. For example,

```
base year 1997
scen years 2005 2010 2015
...
print:
The scenario extends from [y.start] to [y.fin],
a span of [y.fin-y.start] years.
:print
```

will produce “The scenario extends from 1997 to 2015, a span of 17 years.” with a line break after “2015,”. Any numerical expression can be placed inside the square brackets.

The results of in-line calculations can also be formatted, using C-style formatting statements. The most useful such statement is “%.nf”, where *n* indicates the number of decimal places. For example, an expression like the following will produce a percentage to one decimal place:

```
print:
The urban population in the final scenario year
is ["%.1f", 100 * UrbPop.fin/Pop.fin]%.
:print
```

which will produce as output something like “The urban population in the final scenario year is 65.2%,” with a line break after the word “year.” Other useful format specifiers and characters are shown in Table 5.

Table 5. Format specifiers for in-line calculations

Format Command	Meaning	Example
%.nf	Use <i>n</i> decimal places	%.2f

<i>Format Command</i>	<i>Meaning</i>	<i>Example</i>
<code>%e</code>	Scientific notation (lower-case e)	<code>%e</code>
<code>%E</code>	Scientific notation (capital E)	<code>%E</code>
<code>%nf</code>	Minimum field-width <i>n</i> (right-justified)	<code>%12f</code>
<code>%0nf</code>	Field-width of <i>n</i> , padded with zeros	<code>%012f</code>
<code>%-nf</code>	Minimum field-width <i>n</i> (left-justified)	<code>%-12f</code>

Here are examples of each of these format specifiers, for a value of 148.886. The result is put between vertical bars "| |" to show the effect of the field width specifier:

```
Using format |result|
"% .2f"      |148.89|
"% .2e"      |1.49e+02|
"% .2E"      |1.49E+02|
"%7.2f"      | 148.89|
"%07.2f"     |0148.89|
"%-7.2f"     |148.89 |
```

Square brackets can be placed inside a print block using double brackets. For example,

```
print:
The following text is [[inside a bracket]].
:print
```

will produce "The following text is [inside a bracket]."

The print block allows considerable flexibility in formatting output. Many programs will take formatted text as input – for example, some GIS programs, databases, graphing programs and special-purpose programs such as the Dashboard of Sustainability of the International Institute for Sustainable Development. In cases where the output format of report statements does not match the input requirements for these programs, use print blocks.

Print blocks can also be used to provide mainly text output, with a few embedded numerical results. In this case, HTML codes can produce nicely formatted output. Combining HTML with IPAT-S calculations is described in the next section.

## Producing HTML

In order to produce nicely formatted output, it is useful to embed HTML inside print blocks. The formatted text can then be read by most popular word processors, and can be published on the web. Also, the IPAT Studio IDE has a simple HTML previewer. For example,

```
print:
<H1>Carbon Emissions: An <i>IPAT-S</i> Example Script</H1>
<H2>Introduction</H2>
<p>In [year.0], the population was [Pop.0] million. By [year.last], it
has risen to [Pop.last] million.</p>
<H2>Carbon Emissions</H2>
<p>
Driven by rising population and affluence, but offset by
technological improvements, carbon emissions rise between
[year.first] and [year.last],
from ["%.2f", 1000 * CO2.by/Pop.by] ktC per person in the
base year to
["%.2f", 1000 * CO2.last/Pop.last] ktC per person
by the end of the scenario,
a ["%.1f", 100 * ((CO2.last/CO2.by)/(Pop.last/Pop.by) - 1)]% increase.
Added to the the ["%.1f", 100 * ((Pop.last/Pop.by) - 1)]%
```

```

increase in population, this leads to a rise in total carbon
emissions by ["%.1f", 100 * ((CO2.last/CO2.by)-1)]%
over the course of the scenario, an average rate of
["%.1f", 100 * ((CO2.last/CO2.by)^(1/(year.last-year.first))-1)]%
per year.
</p>
:print

```

will produce output like

```

<H1>Carbon Emissions: An <i>IPAT-S</i> Example Script</H1>
<H2>Introduction</H2>
<p>In 1997, the population was 300 million. By 2020, it
has risen to 370 million.</p>
<H2>Carbon Emissions</H2>
<p>
Driven by rising population and affluence, but offset by
technological improvements, carbon emissions rise between
1997 and 2020,
from 5.67 ktC per person in the
base year to
6.43 ktC per person
by the end of the scenario,
a 13.5% increase.
Added to the the 23.3%
increase in population, this leads to a rise in total carbon
emissions by 40.0%
over the course of the scenario, an average rate of
1.5%
per year.
</p>

```

This output is HTML code that can be viewed in an HTML browser. The line breaks do not matter to the HTML browser, which formats text based on the location of “tags,” such as the paragraph tags `<p>` and `</p>`.

## Redirecting Output

Ordinarily, the destination of any output from an IPAT-S script is set on the interpreter command line. However, it can also be redirected inside a script. Generally, the sequence of commands is the following:

```

# Delete the file if it exists
clear filename
# open file for appending
set output filename

...commands, including print or report statements...

# Close the output file: return to default output
reset output

```

Once the output file is set with `set output`, output from any print blocks or report statements is appended to that file. To overwrite an existing file, use `clear` to delete the file before any output. Finally, to return to the default output, use the `reset output` statement.

## Running External Processes

Other programs can be started from within an IPAT-S script. External programs can be started using three commands: `start`, `run`, and `wait`. These commands have the same structure: the command, followed by a program name, in quotes, and an optional command line (arguments to send to the program), also in quotes. The different commands have the following effects (where the square brackets indicate optional text):

`start "progname" ["commandline"]`

1. Open the program in a new window
2. Don't wait for it to finish

`run "progname" ["commandline"]`

1. Run the program in a hidden window (for example, for console applications)
2. Don't wait for it to finish

`wait "progname" ["commandline"]`

1. Run the program in a hidden window
2. Wait for it to finish (for example, if the output from some program will be used as input to IPAT-S or to some later command)

The IPAT-S interpreter will search the `PATH` environment variable for the program. Note that in program and file names, either forward or backward slashes can be used to separate directories.

## 6 Flow Control

---

There are three flow control structures in IPAT-S, the conditional `if ... then:... else ... :if` structure, the `foreach` loop and the `abort` statement.

### *The if ... then:... else ... :if Structure*

The `if ... then:... else ... :if` structure has the general form

```
if logical_expr then:
    ... commands ...
else
    ... commands ...
:if
```

If `logical_expr` is true, then the first set of commands are executed and the second set are not. If the expression is false, then the second set of commands are executed and the first set are not. If statements can be nested, up to 100 deep.

The colon after `then` is optional, and `endif` can be used instead of `:if` to close the block:

```
if logical_expr then
    ... commands ...
else
    ... commands ...
endif
```

The `else` keyword is not required. The following is a legal statement:

```
if logical_expr then:
    ... commands ...
:if
```

In this case, if `logical_expr` is true, then the commands are executed, and are not executed if it is false. For example,

```
if printHTML then:
    read 'reports/HTML.ips'
:if
```

might optionally print or not print HTML-formatted output from a scenario, while

```
if Pop.fin .gt. Pop.init then:
print:
The population increased over the scenario.
:print
else
print:
The population did not increase over the scenario.
:print
:if
```

would optionally print that the population increased or did not increase, depending on the scenario values of the `Pop` summable variable.

## The ForEach Loop

The foreach loop provides for explicitly looping over one or more dimensions. Ordinarily this is not necessary, because IPAT-S chain, index, reporting and other expressions automatically take care of iteration over dimensions. However, in some circumstances automatic looping is not available – for example, when using an external library or producing formatted output. In these cases, the foreach loop can be used.

The foreach loop has the general form

```
for each dim1, dim2, ... :
...
:for
```

where one of the dimensions dimN can be set to year, yr or y, indicating a loop over years.

Alternative forms are:

```
foreach dim1, dim2, ... :
...
:for
```

and

```
foreach dim1, dim2, ... :
...
:foreach
```

Within the foreach loop, dimension values can be referred to in two ways. First, in a calculation with explicit dimensions, dimension values can be inserted using "dimname = ?". For example,

```
foreach fuel:
:: var1 >>-> var2{fuel = ?}
:for
```

This calculation will drive the variable var2 with the values for var1 for each value of the dimension fuel. (However, for this example, var1 >>-> var2 will do the same thing, without the need for a foreach loop.)

Second, the text for a dimension value can be inserted using the special notation \$dimname. For example,

```
foreach fuel:
print:
The value for the fuel [$fuel] in [y.fin] is [myvar.fin{fuel = ?}].
:print
:for
```

Values for specific years can be referred to using the notation .?. For example, a script fragment like

```
foreach year, region, scenario:
assert GDP.?(region=?, scenario=?) .=. GDPinit.?(region=?, scenario=?) \
"GDP has changed"
:foreach
```

can be used to find out if a value that should not have been changed by part of a script has changed, for some region, scenario or year.

Note that the foreach loop is limited, and should only be used when no other option is available. The most common situations where it should be used are when using external procedures (because the arguments to procedures cannot have dimensions), when using logical expressions, and when producing formatted output using print blocks. An example of a foreach loop in combination with an external procedure is given in the section *Procedures*, in chapter 4.

The main reason to avoid foreach loops when possible is that foreach loops are time-consuming, while the built-in loops are very fast. For example, the following (standard) script takes about 50 ms on a Pentium laptop:

```
baseyear 2000
scenario years 2025 2050

ditto 'val1' 'val2' 'val3' 'val4' 'val5', 'val6', 'val7', 'val8':
dim dim1 ''
dim dim2 ''
dim dim3 ''

summvvar a{dim1, dim2, dim3} b{dim1, dim2, dim3}
a.0 = 10
b.0 = 10

:: >> gr(1.0%)-> a -> b

report a as "a"
report b as "b"
```

The equivalent script using a foreach loop takes about 1500 ms (1.5 seconds) – 30 times longer:

```
baseyear 2000
scenario years 2025 2050

ditto 'val1' 'val2' 'val3' 'val4' 'val5', 'val6', 'val7', 'val8':
dim dim1 ''
dim dim2 ''
dim dim3 ''

summvvar a{dim1, dim2, dim3} b{dim1, dim2, dim3}
a.0 = 10
b.0 = 10

ditto dim1=?,dim2=?,dim3=:

foreach dim1,dim2,dim3:
:: >> gr(1.0%)-> a{''} -> b{''}
:for

report a as "a"
report b as "b"
```

For this reason, built-in loops should be used whenever possible. When it is not possible, foreach loops are an option.

Because foreach loops are intended for limited circumstances, there are some restrictions on their use. First, foreach loops cannot be nested.<sup>7</sup> Second, no `read` statements can be placed inside a foreach loop – all of the commands inside the loop must be in the script or subfile where the foreach loop appears.

---

<sup>7</sup> This is expected to change in a future version of IPAT-S.

## ***The Abort Statement***

To abruptly exit from a script (for example, in case of an error that is caught by the script, rather than by the interpreter), use the `abort` statement. It has the following syntax:

```
abort n
```

where  $n$  is an integer code. The code will be returned as an exit code by the interpreter, which can then be read and used by a batch file or other calling program. It is best to use code values between 64 and 127, since codes in that range will be different from the C compiler exit codes and the codes from the IPAT-S interpreter.

## 7 Using Subfiles

---

### Readfiles

It is easy for scripts to get very large. Even a moderately-sized script can be difficult to read. Also, the bulk of a script is frequently devoted to loading the base-year data, which is not the most interesting information for someone reviewing the script. To help with this situation, IPAT-S allows scripts to be broken into separate files, which are then brought into the main part of the script using a `read` statement. For example, if the base-year values for a feed allocation table are placed in a file “FeedTable.txt,” then the lines

```
summvvar FeedTable{livestock feed} # Feed supply in 1000 LSU
read "FeedTable.txt"
```

could be appear in the main script. This makes the script easier to read.

Sample scripts using the `read` command are included in the IPAT-S distribution.

### Scope

Unless they are placed in a global block (see below), the scope of variables is limited to the script in which they are declared, and any scripts read in by the script in which they are declared. This means that variables declared in a script that is read in are only available to that script (and to any scripts that it reads in), but are not available to the calling script.

The benefit of restricting the scope of some variables is that while major variables can be accessed from anywhere in the script, any temporary variables introduced in detailed calculations can be hidden from the rest of the script. Also, the names of temporary variables can be reused in other parts of the script.

### Global Blocks

Variables declared inside a global block are available anywhere in the script. This can be used to make scripts easier to read. For example, suppose that there is a script with a large number of variables declared:

```
base year 2000
scenario years 2010 to 2050 by 10
summvvar pop, GDP, PrimaryEnergy, Cropland, Grazing_Land, \
        WaterUse, CO2, NOx, SOx, CH4, ElectricityUse
...

```

To clean up the script, these can be placed in an external file:

```
base year 2000
scenario years 2010 to 2050 by 10
read 'init/Variables.ips'
...

```

where the file “Variables.ips,” in the “init” folder, contains the variables, declared inside a global block:

```
Global:
summar pop, GDP, PrimaryEnergy, Cropland, Grazing_Land, \
:Global WaterUse, CO2, NOx, SOx, CH4, ElectricityUse
```

This ensures that the variables are accessible anywhere in the script.

## 8 Extending With External Libraries

---

The IPAT-S application programming interface (API) allows external libraries (dynamic link libraries, or DLLs) to be loaded in an IPAT-S script. The syntax for loading and calling procedures is described in chapter 4. In this chapter the procedure for writing an external library is outlined.

External libraries must be written in a programming language that supports compilation, such as C, C++, Delphi, Fortran, Visual Basic, etc. An interpreted language like IPAT-S is not suitable. Also, the compiler should be capable of creating DLLs under Windows. Any major Windows compiler should be able to do that, such as Borland C++, Delphi, Microsoft Visual C++, MetroWerks Code Warrior, etc. The standard IPAT-S library is built using the free mingw compiler.

Many applications today offer APIs to allow users to extend their functionality. As with other APIs, an IPAT-S library must adhere to a fairly rigid set of rules. However, if you already have a model that you wish to make available in an IPAT-S script, you do not need to rewrite the model in order to make it available. In general, it should be possible to make a “wrapper” function that meets the requirements of the IPAT-S API. The wrapper function would then call one or more of the functions in the model code. Only the wrapper function is visible to the IPAT-S script (that is, only the wrapper function is exported in the DLL). The details of the model are hidden.

### *IPAT-S API Calling Conventions*

When IPAT-S calls an external procedure, it passes it four arguments, in this order:

<code>param_list</code>	an array of parameters of type <code>IPATS_param</code>
<code>num_params</code>	a count of the total number of parameters
<code>years</code>	an array of year values, of size <code>num_years</code>
<code>num_years</code>	the number of years (base year and scenario years)

All of the parameters are integers, or arrays of integers, except for the first one. The `param_list` is an array of type `IPATS_param`, which is a special data type defined for the IPAT-S API. `IPATS_param` is a structure (struct), and in C it has this definition:

```
typedef struct {
    int type;
    double numval;
    double *numptr;
} IPATS_param;
```

This says that the structure holds, in order, an integer `type`, a double value `numval` and an address (pointer) of a double value, `numptr`.

The IPAT-S interpreter is compiled using the mingw 32-bit compiler. In this environment, an integer declaration `int x` in C should correspond to `Dim x As Long` in Visual Basic and either `x:`

Integer or `x: Longint` in Delphi. A declaration of `double x` in C should correspond to `Dim x As Double` in Visual Basic and either `x: Double` or `x: Real` in Delphi. The asterisk `*` in C indicates a pointer – the address of a variable. So, `double *x` in C should correspond to `x: ^Double` in Delphi. In Visual Basic, when referring to a pointer, refer to it as though it holds the `AddressOf` a variable.

The `IPATS_param` structure is defined in a C header file, “`IPATS_API.h`,” that is part of the IPAT-S distribution. The header file also defines these constants:

```
#define IPATS_NUMVAL 0 // Numeric value
#define IPATS_NUMREF 1 // Numeric variable
#define IPATS_VARREF 2 // Array variable, indexed by year
#define IPATS_TMPVAR 3 // Temporary array, indexed by year
```

The type argument in the `IPATS_param` structure contains one of these values. If the library is written in C, then the “`IPATS_API.h`” file can be included in the code, and the constants `IPATS_NUMVAL`, etc. can be used to refer to the types. Otherwise, the numerical values can be used.

For type `IPATS_NUMVAL`, the parameter value is passed in the `numval` variable in the `IPATS_param` structure. For all other types, the parameter value is an address, or pointer, which is passed in the `numptr` variable. Parameters of type `IPATS_NUMREF` or `IPATS_VARREF` can be modified by an external library procedure, and the changes will be reflected in the IPAT-S script. Parameters of type `IPATS_NUMVAL` are simply numbers, and cannot be changed. If parameters of type `IPATS_TMPVAR` are changed in the external library, the changes will be discarded in IPAT-S.

The IPAT-S interpreter does very little checking of parameter types before calling a procedure. The interpreter only checks to see if variables have dimensions, and issues an error message if any do. Otherwise, the external procedure (that is, the library code that you write) is responsible for checking arguments. This requires caution on the part of the library programmer. However, it also leaves a lot of flexibility. One example of how that flexibility can be used is in the standard library procedure `interpolate`. The `interpolate` procedure can take any number of arguments, and most of the arguments can be of any of the four defined types. Because the arguments are not checked by IPAT-S, it is easier to write such a procedure. The code for the `interpolate` procedure is given below.

Information about errors is passed back to the IPAT-S interpreter using numeric codes. The C header file defines labels for the codes:

```
#define IPATS_EXIT_OK 0 // Program ran without error
#define IPATS_ERR 1 // Generic "error" condition
#define IPATS_ERR_NUMPARMS 2 // Wrong number of parameters
#define IPATS_ERR_PARMTYPE 3 // Wrong parameter type
#define IPATS_ERR_NOTEMP 4 // Improper use of temp variable
```

For libraries written in C or C++, the labels can be used if the “`IPATS_API.h`” file is included in the code. For libraries written in other languages, the numeric codes can be used.

## The `IPATS_API.h` Header File

Here is the complete `IPATS_API.h` C header file:

```
#ifndef IPATS_API
#define IPATS_API
#include <windows.h>
```

```

#define MAXPARAMS 100

#define IPATS_NUMVAL          0      // Numeric value
#define IPATS_NUMREF         1      // Numeric variable
#define IPATS_VARREF         2      // Array variable, indexed by year
#define IPATS_TMPVAR         3      // Temporary array, indexed by year

// Use these codes to return error information to the interpreter
#define IPATS_EXIT_OK        0      // Program ran without error
#define IPATS_ERR            1      // Generic "error" condition
#define IPATS_ERR_NUMPARMS   2      // Wrong number of parameters
#define IPATS_ERR_PARMTYPE   3      // Wrong parameter type
#define IPATS_ERR_NOTEMP     4      // Return value assigned to
                                   // temporary variable

typedef int (*IPATS_PROCEDURE) ();

typedef struct {
    int type;                          // One of the IPATS_types given
                                        // above
    double numval;                     // The value, if IPATS_NUMVAL
    double *numptr;                   // The pointer, if any other type
} IPATS_param;

#endif

```

## Sample API Code

An example of an external library, "IPATS\_standard.dll," with its source code, is included in the IPAT-S distribution. The source file for the IPATS standard library and the "IPATS\_API.h" header file should be useful resources for users wishing to develop their own external libraries.

Here is the complete C code for an API procedure, the `interpolate` procedure in the standard library:

```

#include "IPATS_API.h"

//
// interpolate:
// usage: call interpolate using var x x1 val1 x2 val2 ...
// Perform a linear interpolation between xi, vali pairs
//

__declspec(dllexport) int interpolate (IPATS_param*, int, int*, int);

//
// isNumber is a helper function
//
int isNumber(IPATS_param param) {
    return (param.type == IPATS_NUMVAL || param.type == IPATS_NUMREF) ? 1 : 0;
}

int interpolate (IPATS_param *param_list, int num_params,
                int *years, int num_years) {
    int i, j, numpairs, offset;
    double *val, x, x1, x2, y1, y2;

    // Must be even number of parameters
    if (num_params % 2) return IPATS_ERR_NUMPARMS;
    if (num_params < 4) return IPATS_ERR_NUMPARMS;

    numpairs = num_params/2 - 1;

    // First argument must be a var ref: others are arbitrary
    if (param_list[0].type != IPATS_VARREF) return IPATS_ERR_PARMTYPE;

    val = param_list[0].numptr;

    for (i = 0; i < num_years; i++) {
        if (isNumber(param_list[1])) {
            x = param_list[1].numval;

```

```

    } else {
        x = param_list[i].numptr[i];
    }
    for (j = 0; j < numpairs - 1; j++) {
        offset = 2 * j;
        if (isNumber(param_list[offset + 2])) {
            x1 = param_list[offset + 2].numval;
        } else {
            x1 = param_list[offset + 2].numptr[i];
        }
        if (isNumber(param_list[offset + 3])) {
            y1 = param_list[offset + 3].numval;
        } else {
            y1 = param_list[offset + 3].numptr[i];
        }
        if (isNumber(param_list[offset + 4])) {
            x2 = param_list[offset + 4].numval;
        } else {
            x2 = param_list[offset + 4].numptr[i];
        }
        if (isNumber(param_list[offset + 5])) {
            y2 = param_list[offset + 5].numval;
        } else {
            y2 = param_list[offset + 5].numptr[i];
        }
        if (x < x1) {val[i] = y1; break;}
        if (x1 == x2) continue;
        if (x <= x2) {val[i] = y1 + (y2 - y1) * (x - x1)/(x2 - x1); break;}
    }
    if (j == numpairs - 1) {
        offset = 2 * j;
        if (isNumber(param_list[offset + 3])) {
            y1 = param_list[offset + 3].numval;
        } else {
            y1 = param_list[offset + 3].numptr[i];
        }
        val[i] = y1;
    }
}
return IPATS_EXIT_OK;
}

```

## 9 Miscellaneous

---

### *Enhancing the Legibility of Scripts*

#### In-Line Comments using #

In-line comments are indicated in scripts using a pound sign (#). Anything from a pound sign to the end of a line will be ignored by the interpreter.

#### Comment Blocks

In addition to in-line comments, long blocks of comment text can be set apart using comment blocks:

```
# This is an in-line comment
summar Population{region, scenario} # Another in-line comment

comment:
This is a long block of commented text that goes on for
more than one line. In this case it is more convenient to
set the text apart using a comment block.
:comment
```

Anything within comment: and :comment is ignored by the interpreter.

#### Line Continuation

Lines can be continued if they are very long (or for added clarity), by adding a backslash (\) at the end of the line. For example,

```
summar Population {region} UrbanPop {region} \
FuelUse {fuel, region} Emissions {fuel, region}
```

#### Ditto

Frequently in a script the same text appears repeatedly, and close together in a block, especially when entering base-year data. For example, suppose there is a scenario for a multi-region area, and use region as a dimension. It is then necessary to enter data for each region:

```
Population.by{region = 'Southeast'} = ...
GDP.by{region = 'Southeast'} = ...
FuelUse.by{region = 'Southeast', fuel = 'coal'} = ...
FuelUse.by{region = 'Southeast', fuel = 'petroleum'} = ...
FuelUse.by{region = 'Southeast', fuel = 'natural gas'} = ...

...
```

If there are several regions, and several values for each region, this can quickly get tedious, as well as being visually noisy. For this situation the IPAT-S interpreter offers the ditto statement. Ditto is a kind of “instant macro.” It has this syntax:

```
ditto text :
... '' ...
```

Anywhere the ditto marks (two single quotation marks: ' ') appears, the text between ditto and the colon ( : ) will be copied in. (Any spaces following ditto and before : are removed first.)

Using the ditto feature, the script fragment above could be written

```
ditto region = 'Southeast' :
Population.by{''} = ...
GDP.by{''} = ...
FuelUse.by{'', fuel = 'coal'} = ...
FuelUse.by{'', fuel = 'petroleum'} = ...
FuelUse.by{'', fuel = 'natural gas'} = ...
...
```

or even

```
ditto .by{region = 'Southeast' :
Population''} = ...
GDP''} = ...
FuelUse'', fuel = 'coal'} = ...
FuelUse'', fuel = 'petroleum'} = ...
FuelUse'', fuel = 'natural gas'} = ...
...
```

but this second option is not recommended, because it is difficult to understand when reading the script.

## Marking Inputs with <>

Some numbers are special, in that they are key scenario inputs. Scenario inputs are values that are likely to change from one scenario to another, or that represent a crucial assumption regarding a particular scenario. For example, income growth might change significantly between scenarios. In a script of reasonable size, it can be difficult for someone to find where the key scenario inputs are hidden. For this reason, IPAT-S offers a special syntax for marking scenario inputs.

To indicate that one or more numbers is an input, enclose them in angle brackets (except in LP blocks, where the notation is not available):

```
ratio income = growth(< 2.1%, 2.5% >)
```

Note that the IPAT-S interpreter ignores the angle brackets. They are only offered as a convenience for the user. However, the IPAT Studio IDE, which is distributed with the IPAT-S interpreter, takes advantage of the angle bracket notation in several ways: by color-coding everything between angle brackets, by offering shortcuts to jump to numbers marked off by angle brackets, and by allowing script writers to extend the graphical user interface so that users can modify key inputs interactively.

## Assertions

Assertions are an important tool for preventive programming – making sure that a script does what it is supposed to do and minimizing the danger of inadvertently breaking a script when modifying it. An assertion in IPAT-S has the following format:

```
assert logical_expression string
```

If `logical_expression` is false, then `string` is reported as an error. For example,

```
assert BYPop.fin .eq. BYPop.0 "BYPop different in base year and final year"
```

will report an error “BYPop different in base year and final year” if the assertion that they are the same fails.

Note that assertions are never supposed to appear, and should not be used as a routine way of communicating with users. Instead, they are insurance against changes to the code that make the assertion false. For example, if a script is being developed by a team, one team member might assume that the GDP values she develops in one module are unchanged when she uses them again in a downstream module. If she then adds an assertion that *says* they are the same, she can ensure the integrity of the code.

Assertions can be turned off either with the `-D` command-line switch when invoking the IPAT-S interpreter, or by setting the built-in `_DEBUG_` variable to `false`. By default, the IPAT Studio IDE checks assertions, while IPAT Scenario Navigator turns them off.

## Errors

IPAT-S produces error messages whenever it encounters incorrect syntax, an error with an LP calculation, and in other cases. Any error message will return the line number where the error occurred as well as the text where the interpreter found a problem. For example, “Line #1: parse error at 'dim'.” This can help narrow down the location of the error. If the line is very long, one strategy is to break the line up by inserting line-continuation characters (a backslash, `\`) to locate the error.

## Special Cases

Divide-by-zero errors are almost never reported. In most cases, the interpreter will simply return a zero as the result.

Many syntax errors are indicated generically as “parse error.” If a parse error arises and no base year has been defined, then the interpreter will ask, “Did you forget to define a base year?” because that is the most likely cause. There are several additional IPAT-S-specific error messages.

## List of Error Messages

Table 6. List of error messages

Message	Cause	Solution
'\$dimname' outside of print block or foreach loop	Loop reference to a dimension name outside of a loop or outside of a print block	<ul style="list-style-type: none"> <li>• Make sure the reference is enclosed in both a print block and a foreach loop</li> <li>• Delete the reference</li> <li>• Replace the <code>\$dimname</code> reference with a fixed reference</li> </ul>
Ambiguous expression: not a number	A variable with dimensions is being illegally assigned a single numerical value	Completely specify dimensions and years on the left-hand-side of the equals sign

<i>Message</i>	<i>Cause</i>	<i>Solution</i>
Can't construct index from variable: zero value	Divide by zero error in an index expression	Don't construct an index for that variable, or catch the error condition (for example, using a <code>step0()</code> function)
Can't nest foreach loops	Putting one foreach loop inside another ("nesting")	<ul style="list-style-type: none"> <li>• Delete one of the foreach loops</li> <li>• Combine the foreach loops (for example, <code>foreach dim1 dim2</code>)</li> </ul>
Can't open 'read' file	Error trying to open an external readfile	<ul style="list-style-type: none"> <li>• Check the file name (including the path from the main script)</li> <li>• Make sure the file exists</li> </ul>
Can't put a 'read' file in foreach loop	Trying to open an external readfile in foreach loop	<ul style="list-style-type: none"> <li>• Delete the read statement</li> <li>• Copy the contents of the readfile into the calling script</li> </ul>
Can't use 'dim=sum' or 'dim=ave' for index expressions	Trying to sum or average over a dimension in an index expression. Not allowed, since result is ambiguous.	<ul style="list-style-type: none"> <li>• Remove <code>dim=sum</code> or <code>dim=ave</code></li> <li>• Define a new variable with proper dimensions to hold the result</li> </ul>
Can't use 'dim=sum' or 'dim=ave' when assigning value	Trying to sum or average over the dimensions in the LHS of an assignment or the result of a chain. Not allowed, since result is ambiguous.	<ul style="list-style-type: none"> <li>• Remove <code>dim=sum</code> or <code>dim=ave</code></li> <li>• Define a new variable with proper dimensions to hold the result</li> </ul>
Dimension not defined	Referring to a dimension name that is not defined	<ul style="list-style-type: none"> <li>• Check the spelling of the dimension name</li> <li>• Add the dimension if necessary</li> </ul>
Dimension value not defined	Referring to a dimension value that is not defined	<ul style="list-style-type: none"> <li>• Check the spelling of the dimension value</li> <li>• Add the dimension value if necessary</li> </ul>
Dimension value without dimension name	Encountered <code>dim 'value'</code> before dimension name	Add the dimension name after the keyword <code>dim</code> or <code>dimension</code>
DLL failed to load	Could not load dynamic link library (DLL)	<ul style="list-style-type: none"> <li>• Check the spelling of the file name</li> <li>• Check that the file exists</li> <li>• Check that the file is in either: 1) the same folder as the script; 2) the IPAT-S folder; 3) the Windows System or System32 folder</li> </ul>
Duplicate dimension name	Defining a dimension name that already exists	<ul style="list-style-type: none"> <li>• Choose a different name</li> <li>• Delete the duplicate reference</li> </ul>
Duplicate dimension value	Defining a dimension value that already exists	<ul style="list-style-type: none"> <li>• Choose a different name</li> <li>• Delete the duplicate entry</li> </ul>
Error loading procedure	Could not load procedure from a DLL	Check the spelling of the procedure name

<i>Message</i>	<i>Cause</i>	<i>Solution</i>
Exceeded maximum number of years (999): y discarded	Too many years defined: year y will not be available	If possible, reduce the number of scenario years in your script – otherwise, contact: support@ipat-s.kb-creative.net
Foreach loop reference outside foreach loop	Loop reference to a dimension (dim = ?) outside loop	<ul style="list-style-type: none"> <li>• Enclose the reference in a foreach loop</li> <li>• Delete the reference</li> <li>• Replace the dim = ? reference with a fixed reference</li> </ul>
Foreach loop reference to non-looping dimension	Referring to a dimension as though it were a loop index	<ul style="list-style-type: none"> <li>• Add the dimension to the foreach loop</li> <li>• Delete the reference</li> <li>• Replace the reference with a fixed reference</li> </ul>
Illegal character in declaration	Unexpected character when defining a variable, dimension name or other identifier	Replace the offending character with a valid character (a letter, digit or underscore)
Illegal year reference	Expression yN refers to an invalid year	<ul style="list-style-type: none"> <li>• Change the reference to a valid year</li> <li>• Change the list of years</li> </ul>
Invalid year index	Year reference by index (e.g., var.z) to a year that does not exist	<ul style="list-style-type: none"> <li>• Change the reference to a valid year</li> <li>• Change the list of years</li> </ul>
IPAT-S calculation error	Error in mathematical function (e.g., sqrt(-1))	Check the calculations – this probably indicates an error in the scenario logic. In cases where it doesn't, consider using an if statement or step(), ln() or log() function to handle the special condition
LP: Cannot use 'change' when solving for all years	Trying to bound the change in a solution variable when solving for the base year	Solve only for scenario years, or reorganize the LP calculations
LP: Dual variable must have same dimensionality as RHS	Dual variables must have the same dimensions as the right hand side of the constraint	Change the definition of the dual variable so its dimensions match the constraint value
LP: Infeasible	The LP is inconsistent	Check the logic of the LP and all of the coefficient values, and modify the LP – the logic may be flawed, or there may be no solution for that set of coefficient values
LP: Slack variable must have same dimensionality as RHS	Slack variables must have the same dimensions as the right hand side of the constraint	Change the definition of the slack variable so its dimensions match the constraint value

<i>Message</i>	<i>Cause</i>	<i>Solution</i>
LP: Solution variable appears in more than one term	Violation of LP syntax	Combine all coefficients of a solution variable into one term:  <i>Incorrect:</i> $C1 * SolVar + C2 * SolVar = \theta$  <i>Correct:</i> $(C1 + C2) * SolVar = \theta$
LP: Unbounded	The LP does not have a finite solution	Check the logic of the LP and all coefficient values
LP: Unknown problem with solution	The LP could not be solved, for unknown reason	Check the logic of the LP and all coefficient values
LP: Variable to solve for in constraint value	Solution variable where general variable should be	Move the expression onto the left-hand side of the constraint
parse error	Generic syntax error	Check the syntax of the line where the error occurred or the line immediately before it – most likely it is a simple error
Procedure: must be variable reference	Cannot use number or temporary variable in external procedure	Refer to the documentation for the procedure you are using for a list of valid parameters
Procedure: unknown error	Generic error in external procedure	The procedure documentation (or source code) may help – otherwise, check parameter values for a possible source of error
Procedure: wrong number of parameters	Wrong number of parameters in external procedure	Refer to the documentation for the procedure you are using for a list of valid parameters
Procedure: wrong parameter type	Wrong type of parameter in external procedure	Refer to the documentation for the procedure you are using for a list of valid parameters
'Read' files nested too deeply	Too many nested files using read command	If possible, rearrange the script to use fewer readfiles – otherwise, contact: support@ipat-s.kb-creative.net
Reference to year in foreach loop, but not looping over year	There is a reference to a year as though in a foreach loop (var. ?) but not looping over years	Add year to the list of looping dimensions or remove the year reference
This name already used for ...	Using a variable name that has already been used, either earlier in the script or internally by the IPAT-S interpreter	<ul style="list-style-type: none"> <li>• Choose a different name</li> <li>• Delete the duplicate definition</li> </ul>
Too many parameters in procedure call	Exceeded maximum number of parameters in procedure call (a limit of the IPAT-S interpreter)	If possible, rearrange the calculations to use a smaller number of parameters – otherwise, contact: support@ipat-s.kb-creative.net

<i>Message</i>	<i>Cause</i>	<i>Solution</i>
Unexpected character in dimension reference	Illegal character within { } braces	Check the line where the error occurred – this is most likely a simple error
Unmatched 'If'	: if or endif found with no matching if	Delete the : if or endif, or add the if
Variable cannot have dimensions in procedure call	External procedures cannot use dimensioned vars	Define a variable without any dimensions, and assign the result to the dimensioned variable after the procedure call
Variable variable not defined	Referring to a variable that is not defined	Add the variable definition earlier in the script
Warning: dimension not available in result of a chain	In a chain expression, a dimension that appears in the left-hand side of the expression does not appear on the right-hand side	Create an intermediate result variable with the missing dimension, then assign it to the desired variable
Year not defined	Year reference that is not the base year or scenario year	<ul style="list-style-type: none"> <li>• Change the reference to a valid year</li> <li>• Change the list of years</li> </ul>

## Reserved Words

There are several words that would be valid variable names except that they have special meaning for the IPAT-S language. These are reserved words, and are not available as variable names. Also, unlike variable names, they are not case-sensitive: the interpreter will recognize them whether they are capitalized or not.

This is a complete list of reserved words (as well as some multi-word reserved phrases):

as	from	scenario year (or scenario years, scen year, etc.)
base year (or baseyear)	growth (or gr)	set output
by	if	start
call	increment (or incr)	summable variable (or summ var, summvar, etc.)
chain	index (or ndx)	summarize
clear	load	then
dimension (or dim)	logical	to
ditto	no	true
else	number (or num)	using
every	ratio	variable (or var)
false	report	wait
for	reset output	
foreach	run	

y	year (or yr)
y0, y1, y2, etc.	yes

The following words are only reserved in an LP block:

all	minimize (or min)	solve for (or solvefor)
maximize (or max)	scenario (or scen)	

Function names and some built-in variables and constants are also reserved. These names are case-sensitive.

_DEBUG_	accum	N_inv
_LP_INFEASIBLE_	cos	pi
_LP_OK_	exp	rate
_LP_REPORT_	lag	sign
_LP_STATUS_	ln	sin
_LP_UNBOUNDED_	ln0	sqrt
_LP_UNKNOWN_ERROR_	log	step
_LP_YEAR_	log0	step0
abs	N	

# Appendix: The GNU Lesser Public License

---

GNU Lesser Public License  
Version 2.1, February 1999  
Copyright (C) 1991, 1999 Free Software Foundation, Inc.  
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. [This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

## Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

#### TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not

derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made

generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

#### NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS