
AIMMS Language Reference - External Procedures and Functions

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit www.aimms.com or order your hard-copy at www.lulu.com/aimms.

Copyright © 1993–2011 by Paragon Decision Technology B.V. All rights reserved.

Paragon Decision Technology B.V.	Paragon Decision Technology Inc.	Paragon Decision Technology Pte.
Schipholweg 1	500 108th Avenue NE	Ltd.
2034 LS Haarlem	Ste. # 1085	80 Raffles Place
The Netherlands	Bellevue, WA 98004	UOB Plaza 1, Level 36-01
Tel.: +31 23 5511512	USA	Singapore 048624
Fax: +31 23 5511517	Tel.: +1 425 458 4024	Tel.: +65 9640 4182
	Fax: +1 425 458 4025	

Email: info@aimms.com
WWW: www.aimms.com

AIMMS is a registered trademark of Paragon Decision Technology B.V. IBM ILOG CPLEX and sc CPLEX is a registered trademark of IBM Corporation. GUROBI is a registered trademark of Gurobi Optimization, Inc. KNITRO is a registered trademark of Ziena Optimization, Inc. XPRESS-MP is a registered trademark of FICO Fair Isaac Corporation. MOSEK is a registered trademark of Mosek ApS. WINDOWS and EXCEL are registered trademarks of Microsoft Corporation. $\text{T}_{\text{E}}\text{X}$, $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$, and $\text{A}_{\text{M}}\text{S}_{\text{L}}\text{A}_{\text{T}}\text{E}_{\text{X}}$ are trademarks of the American Mathematical Society. LUCIDA is a registered trademark of Bigelow & Holmes Inc. ACROBAT is a registered trademark of Adobe Systems Inc. Other brands and their products are trademarks of their respective holders.

Information in this document is subject to change without notice and does not represent a commitment on the part of Paragon Decision Technology B.V. The software described in this document is furnished under a license agreement and may only be used and copied in accordance with the terms of the agreement. The documentation may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Paragon Decision Technology B.V.

Paragon Decision Technology B.V. makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Paragon Decision Technology B.V., its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claims for lost profits, fees or expenses of any nature or kind.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors shall not be responsible under any circumstances for providing information or corrections to errors and omissions discovered at any time in this book or the software it describes, whether or not they are aware of the errors or omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors do not recommend the use of the software described in this book for applications in which errors or omissions could threaten life, injury or significant loss.

This documentation was typeset by Paragon Decision Technology B.V. using $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ and the LUCIDA font family.

Chapter 11

External Procedures and Functions

Even though AIMMS offers easy-to-use multidimensional data structures combined with a powerful programming language, there are often good reasons to relay parts of the execution of your model to external procedures and functions written in e.g. C/C++ or FORTRAN. The capability to call external procedures and functions in your AIMMS application allows you

Why call external procedures

- to re-use existing software (e.g. a library of financial functions, or a collection of accurate, nonlinear process models),
- to speed up selected computations by making use of dedicated data structures which are difficult to implement in AIMMS itself, and
- to provide links to external data sources (e.g. on-line data feeds or proprietary databases).

This chapter describes the steps you have to follow for linking libraries of external procedures and functions to AIMMS. Such procedures and functions can be used to manipulate AIMMS data during the execution of a model. In addition, external libraries may contain functions that can be used inside the constraints of a nonlinear mathematical program.

This chapter

11.1 Introduction

The aim of this section is to give you a quick feel for the effort required to make a link to an external function or procedure through a short illustrative example linking a C implementation of the Cobb-Douglas function (discussed in Section 10.2) into an AIMMS application. Section 32.1 contains a more elaborate example of an external procedure which uses AIMMS API functions to obtain additional information about the passed arguments.

Getting started

The interface to external procedures and functions is arranged through special EXTERNAL PROCEDURE and EXTERNAL FUNCTION declarations which behave just like internal procedures and functions. Instead of specifying a body to initiate internal AIMMS computations, the execution of external procedures and functions is relayed to the indicated procedures and functions inside one or more DLL's.

External procedures and functions

Consider the Cobb-Douglas function discussed in Section 10.2. Given the cardinality n of the set `InputFactors` and two arrays `a` and `c` of doubles representing the one-dimensional input arguments of the Cobb-Douglas function (both defined over `InputFactors`), the following simple C function computes its value.

*The
Cobb-Douglas
function*

```
double Cobb_Douglas( int n, double *a, double *c ) {
    int i;
    double CD = 1.0 ;

    for ( i = 0; i < n; i++ )
        CD = CD * pow(c[i],a[i]) ;

    return CD;
}
```

In the sequel it is assumed that this function is contained in a DLL named "Userfunc.dll".

In order to make the function available in AIMMS you have to declare an EXTERNAL FUNCTION `CobbDouglasExternal`, which just relays its execution to the C implementation of the Cobb-Douglas function discussed above. The declaration of `CobbDouglasExternal` looks as follows.

*Linking to
AIMMS*

```
EXTERNAL FUNCTION
    identifier   : CobbDouglasExternal
    arguments    : (a,c)
    range        : nonnegative
    DLL name     : "Userfunc.dll"
    return type  : double
    body call    : Cobb_Douglas( card : InputFactors, array: a, array: c )
```

The arguments `a` and `c` must be declared in the same way as for the internal `CobbDouglas` function discussed on page 142, with the exception that for the external implementation we will also compute the Jacobian with respect to the argument `c(f)`. For this reason, the argument `c(f)` is declared as a VARIABLE.

```
SET:
    identifier   : InputFactors
    index        : f ;
PARAMETER:
    identifier   : a
    index domain : f
VARIABLE:
    identifier   : c
    index domain : f ;
```

The translation type “card” of the set argument `InputFactors` causes AIMMS to pass the cardinality of the set as an integer value to the external function `Cobb_Douglas`. The translation type “array” of the arguments `a` and `c` are instructions to AIMMS to pass these arguments as full arrays of double precision values. As function arguments are always of type `Input`, AIMMS will disregard any changes made to the arguments by the external function. The double return value of the C function `Cobb_Douglas` will become the result of the function `CobbDouglasExternal`.

Explanation

After the declaration of an external function or procedure you can use it as if it were an internal function or procedure. Thus, to call the external function `CobbDouglasExternal` in the body of a procedure the following statement suffices.

Calling external functions

```
CobbDouglasValue := CobbDouglasExternal(a,c) ;
```

Of course, any two (possibly sliced) identifiers with single common index domain could have been used as arguments. AIMMS will determine this common index domain, and pass its cardinality to the external function.

Unlike internal functions, external functions can be called inside constraints. To accomplish this, the declaration has to be extended with a `DERIVATIVE CALL` attribute. For this attribute you specify the external call that has to be made when AIMMS also needs the partial derivatives of all variable arguments inside constraints of mathematical programs. In the absence of a `DERIVATIVE CALL` attribute, AIMMS will use a differencing scheme to estimate these derivatives. The details of using external functions in constraints, as well as the obvious extension to compute the derivative of the Cobb-Douglas function directly, are given in Section 11.4.

Use in constraints

Once you have developed a collection of external functions and procedures, it may be a good idea to make this available in the form of a library for use in AIMMS applications. In this way, the users of your library do not have to spend any time translating their AIMMS arguments into external arguments of the appropriate type in the external procedure and function declarations.

Setting up external libraries

To provide a library as an entity on its own, you can store all the external procedures and functions in a separate model section, and save this section as a source file. The functions and procedures in the library can then be made available by simply including this source file into a model.

Save library as include file

When you want to protect the interface to your external library, you can accomplish this by encrypting the include file containing the function library (see also the AIMMS User's Guide). Thus, the interface to the external library becomes invisible, effectively preventing misuse of the library outside AIMMS.

Hiding the interface

11.2 Declaration of external procedures and functions

External procedures and functions are special types of nodes in the model tree. They have the same attributes as internal procedures and functions with the exception of the BODY and DERIVATIVE attributes, which are replaced by the attributes in Table 11.1.

External procedures and functions

Attribute	Value-type	See also page
DLL NAME	<i>string, file-identifier</i>	
RETURN TYPE	integer, double	
PROPERTY	FortranConventions, UndoSafe	141
BODY CALL	<i>external-call</i>	
DERIVATIVE CALL	<i>external-call</i>	

Table 11.1: Additional attributes of external procedures and functions

With the mandatory DLL NAME attribute you can specify the name of the DLL which contains the external procedure or function to which you want to make a link in your AIMMS application. The value of the attribute must be a string, a string parameter, or a FILE identifier, representing the path to the external DLL. If you do not specify an absolute path, AIMMS will search for the DLL in all directories in the AIMMSUSERDLL and PATH environment variables, respectively.

The DLL NAME attribute

When you use a FILE identifier to specify an external DLL name, AIMMS will use the CONVENTION attribute of that FILE identifier (if specified) to pass numeric values to any procedure or function in that DLL according to the specified unit convention (see also Section 30.8). When the DLL name has not been specified through a FILE identifier, or when its CONVENTION attribute is left empty, AIMMS will use the unit convention specified for the main model.

FILE identifier and unit conventions

Without any such convention, AIMMS will use the default convention, i.e. arguments will be scaled according to the unit specified for each argument, and AIMMS will assume that the result of an external function is scaled according to the unit specified in its UNIT attribute. Unit analysis for functions and procedures is discussed in full detail in Section 30.4.1.

Default argument scaling

The RETURN TYPE indicates the type of any *scalar* numerical value returned by the DLL function. The possible values are integer and double. AIMMS will use the value returned by the DLL function either as the return value of the EXTERNAL PROCEDURE, or as the (numerical) function value of the EXTERNAL FUNCTION, whichever is applicable. If you do not specify the RETURN TYPE attribute, AIMMS will discard any value returned by the function.

The RETURN TYPE attribute

You cannot directly use the returned value of a DLL function as the function value of an EXTERNAL FUNCTION when its return value is either an indexed parameter, a set, a set element or a string. In such cases you must pass the function name as an additional external argument to the DLL function, and specify how the function value must be dealt with.

Restricted use

Consider a C function Cobb_Douglas_Arg with prototype

Example

```
void Cobb_Douglas_Arg( int n, double *a, double *c, double *CDValue );
```

which passes the Cobb-Douglas function value through the argument CDValue instead of as the return value. In this example CDValue is a scalar, which could have been passed as the result of the DLL function as well. The following EXTERNAL FUNCTION declaration provides a link with Cobb_Douglas_Arg and obtains its function value via the argument list.

```
EXTERNAL FUNCTION
  identifier   : CobbDouglasArgument
  arguments   : (a,c)
  range       : nonnegative
  DLL name    : "Userfunc.dll"
  body call   : Cobb_Douglas_Arg( card : InputFactors, array: a, array: c,
                                scalar: CobbDouglasArgument )
```

With the PROPERTY attribute you can specify through the FortranConventions property whether the external function is based on FORTRAN calling conventions. By default, AIMMS will assume that the DLL function is written in a C-like languages such as C, C++ or PASCAL. The precise differences between both calling conventions are explained in full detail in Section 11.5. In addition, for external procedures, you can specify the UndoSafe property. The semantics of the UndoSafe property is discussed in Section 10.1.

The PROPERTY attribute

As with internal procedures and functions, all formal arguments of an external procedure or function must be declared as local identifiers. AIMMS supports the following identifier types for formal arguments of external procedures and functions:

Formal argument types

- simple and compound SETS,
- scalar and indexed PARAMETERS,

- scalar and indexed VARIABLES (external functions only), and
- HANDLES (external procedures only).

Many details regarding the handling of arguments of internal procedures and functions also apply to external procedures and functions. Thus, arguments of external procedures and functions can be defined over global and local sets, and their associated units of measurement can be specified in terms of either global units or locally defined unit parameters, completely similar to internal procedures and functions (see Section 10.1).

Argument handling

The HANDLE identifier type is only supported for formal arguments of external procedures, i.e. it is not possible to declare global identifiers of type HANDLE. The following rules apply:

HANDLE arguments

- HANDLE arguments are always declared as scalar local identifiers,
- HANDLE arguments can only be passed to the DLL function as an integer handle (see below), and
- the actual argument in a call to the external procedure corresponding to a formal HANDLE argument can be a (sliced) reference to an identifier in your model *of any type and of any dimension*.

HANDLE arguments allow you to completely circumvent any type checking on actual arguments with respect to the dimension and the respective index domains of the corresponding formal arguments in the call to an external procedure. As a result of this, however, the actual data transfer of HANDLE arguments to the DLL function must completely take place via the AIMMS API (see also Chapter 32).

In the mandatory BODY CALL attribute you must specify the call to the DLL procedure or function, to which the execution of the EXTERNAL PROCEDURE or FUNCTION must be relayed. Such an external call specifies:

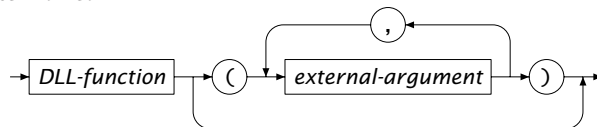
The BODY CALL attribute

- the name of the DLL function or procedure that must be called, and
- how the actual AIMMS arguments must be translated into arguments suitable for the DLL function or procedure.

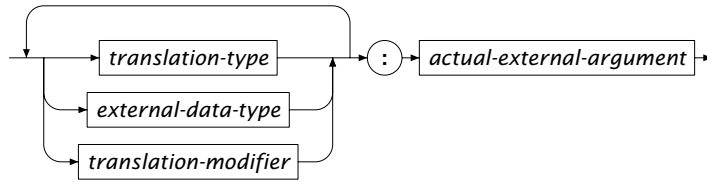
Any external call must be specified according to the syntax below. In the Model Explorer, you can specify all components of the BODY CALL attribute using a wizard which will guide you through most of the necessary detail.

external-call :

Syntax



external-argument :



The mandatory translation type indicates the type of the external argument into which the actual argument must be translated before being passed to the external procedure. The following translation types are supported.

*Mandatory
translation type*

- **scalar**: the actual scalar AIMMS argument is passed on as a scalar of the indicated external data type.
- **literal**: the literal specified in the external call is passed on as a scalar of the indicated external data type, i.e. a literal argument does *never* correspond to an actual AIMMS argument, but is specified directly in the BODY CALL attribute.
- **array**: the AIMMS argument is passed on as an array of values according to the indicated translation type and external data type. The precise manner in which the translation takes place is discussed below.
- **card**: the cardinality of a set argument is passed on as an integer value. The set argument can be either a set passed as an actual AIMMS argument or the domain set of a multi-dimensional parameter passed as an actual argument.
- **handle**: an integer handle to a (sliced) set or parameter argument is passed on. Within the external procedure you must use functions from the AIMMS API (see also Chapter 32) to obtain the dimension, domain and range associated with the handle, or to retrieve or change its data values.
- **work**: an array of the indicated type is passed as a temporary workspace to the external procedure. The actual argument must be an integer expression and is interpreted as the size of the array to be passed on. This translation type is useful for programmers of languages such as standard F77 FORTRAN which lack facilities for dynamic memory allocation.

The actual external argument specified in an external argument of the BODY CALL attribute can be

*Actual external
argument*

- a reference to a formal argument of the EXTERNAL PROCEDURE at hand (for the scalar, array, card, handle and work translation types),
- a reference to a domain set of a formal multi-dimensional argument of the EXTERNAL PROCEDURE at hand (for the card translation type), or
- an integer, double or string literal (such as 12345, 123.45 or "This is a string") directly specified within the BODY CALL attribute (for the literal translation type).

For every formal argument of an `EXTERNAL PROCEDURE`, you can specify its associated *input-output* type through the `Input`, `InOut` (default) or `Output` properties in the `PROPERTY` attribute of the local argument declaration. With it, you indicate whether or not AIMMS should consider any changes made to the argument by the DLL function. For each input-output type, AIMMS performs the following actions:

Input-output type

- **Input:** AIMMS initializes the external argument, but discards all changes made to it by the DLL function,
- **InOut:** AIMMS initializes the external argument, and passes back to the model the values returned by the DLL function, or
- **Output:** AIMMS allocates memory for the external argument, but does not initialize it; the values returned by the DLL function are passed back to the model.

As with internal functions, all `EXTERNAL FUNCTION` arguments are `Input` by definition. The return value of an `EXTERNAL PROCEDURE` and the function value of an `EXTERNAL FUNCTION` are considered as an (implicit) `Output` argument when passed to the DLL function as an external argument.

In translating AIMMS arguments into values (or arrays of values) suitable as arguments for an external procedure or function, AIMMS supports the external data types listed in Table 11.2.

External data type

External data type	Passed as
<code>integer</code>	4-byte (signed) integer
<code>double</code>	8-byte double precision floating number
<code>string</code>	C-style string
<code>integer8</code>	1-byte (signed) integer
<code>integer16</code>	2-byte (signed) integer
<code>integer32</code>	4-byte (signed) integer

Table 11.2: External data types

Not all combinations of input-output types, translation types and external data types are supported (or even useful). Table 11.3 describes all allowed combinations, as well as the resulting argument type that is passed on to the external procedure. The external data types printed in bold are the default, and can be omitted if appropriate. Throughout the table, the data type `integer` can be replaced by any of the other integer types `integer8`, `integer16` or `integer32`.

Allowed combinations

Allowed types			AIMMS argument	Passed as
translation	input-output	data		
scalar	input	integer double string	scalar expression	integer double string
	inout output	integer double string	scalar reference	integer pointer double pointer string
literal	—	integer double string	—	integer double string
card	—	—	set, parameter	integer
array	input inout output	integer double	parameter	integer array double array
		integer string	element parameter set	integer array string array
		string	string/unit parameter	string array
handle	input inout output	—	set, parameter, handle	integer
work	—	integer double	integer expression	integer array double array

Table 11.3: Allowed combinations of translation, input-output and data types

When you are passing a multidimensional AIMMS identifier to an external procedure or function as a array argument, AIMMS passes a one-dimensional buffer in which all values are stored in a manner that is compatible with the storage of multidimensional arrays in the language which you have specified through the PROPERTY attribute. The precise array numbering conventions for both C-like and FORTRAN arrays are explained in Section 11.5.

Passing array arguments

When you pass a scalar or multidimensional output string argument, AIMMS will pass a single char buffer of length 256, or an array of such buffers. You must use the C function strcpy or a similar function to copy the string data in your DLL to the appropriate char buffer associated with the output string argument.

Output string arguments

When considering your options on how to pass a high-dimensional parameter to an external procedure, you will find that passing it as an array is often not the best solution. Not only will the memory requirements grow rapidly for increasing dimension, but also running over all elements in the array inside

Full versus sparse data transfer

your DLL function may turn out to be a very time-consuming process. In such a case, it is much better practice to pass the argument as an integer handle, and use the AIMMS API functions discussed in Section 32.4 to retrieve only the nondefault values associated with the handle. You can then set up your own sparse data structures to deal with high-dimensional parameters efficiently.

In addition to the translation types, input-output types and external data types you can specify one or more translation modifiers for each external argument. Translation modifiers allow you to slightly modify the manner in which AIMMS will pass the arguments to the DLL function. AIMMS supports translation modifiers for specifying the precise manner in which

Translation modifiers ...

- special values,
- the data associated with handles, and
- set elements,

are passed.

When a parameter or variable that you want to pass to an external DLL contains special values like ZERO or INF, AIMMS will, by default, pass ZERO as 0.0, INF and -INF as $\pm 1.0e150$, and will not pass any of the values NA and UNDF. When you specify the translation modifier `retainspecials`, AIMMS will pass all special numbers by their internal representation as a double precision floating point number. You can use the AIMMS API functions discussed in Section 32.4 to obtain the `MapVal` value (see also Table 6.1) associated with each number. The translation modifier `retainspecials` can be specified for numeric arguments that are passed either as a full array or as an integer handle.

... for special values

When passing a multidimensional identifier handle to an external DLL, AIMMS can provide several methods of access to the data associated with the handle by specifying one of the following translation modifiers:

... for handles

- `ordered`: the data retrieval functions will pass the data values according to the particular ordering imposed any of the domain sets of the identifier associated with the handle. By default, AIMMS will use the natural ordering determined by the data entry order of all domain sets.
- `raw`: the data retrieval functions will also pass inactive data (see also Section 23.3). By default, AIMMS will not pass inactive data.

The details of ordered versus unordered and raw data transfer are discussed in full detail in Section 32.4.

AIMMS can pass set elements (in the context of element parameters and sets) to external procedures in various manners. More specifically, set elements can be translated into:

... for set elements

- an integer external data type, or

- a string external data type.

When the external data type is string, AIMMS will pass the element name for each set element. Transfer of element names is always input only. In general, when the external data type is integer, AIMMS can pass either

- the ordinal number with respect to its associated subset domain (`ordinalnumber` modifier), or
- the element number with respect to its associated root set (`elementnumber` modifier).

Alternatively, when set elements are passed in the context of a set you can specify the indicator modifier in combination with the integer external data type. This will result in the transfer of a multidimensional binary parameter which indicates whether a particular tuple is or is not contained in the set.

When you pass an element parameter as an integer scalar or array argument, AIMMS will assume the `ordinalnumber` modifier by default. When passed as integer, element parameters can be input, output or inout arguments. When element parameters are passed as string arguments, they can be input only.

Passing element parameters

Element numbers and ordinal numbers each can have their use within an DLL function. Element numbers remain identical throughout a modeling session using a single data set, regardless of addition and deletion of set elements, or any change in set ordering. For this reason, it is best to use element numbers when the set elements need to be used in multiple calls of the DLL function. Ordinal numbers, on the other hand, are the most convenient means for passing permutations that are used within the current external call only. With it, you can directly access a permuted reference in other array arguments.

When to use

Sets can be passed as array arguments to an external DLL function. When passing set arguments, you have to make a distinction between one-dimensional root sets, one-dimensional subsets (both either simple or compound), and multidimensional subsets and indexed sets. The following rules apply.

Passing set arguments

One-dimensional root sets and subsets can be passed as a one-dimensional array of length equal to the cardinality of the set. To accomplish this, you can must pass such a set as

Pass as one-dimensional array

- an array of integer numbers, representing either the ordinal or element numbers of each element in the set (using the `ordinalnumber` or `elementnumber` modifier), or
- a string array, representing the names of all elements in the set.

One-dimensional set arguments passed in this manner can only be input arguments. As a specific consequence, you cannot modify the contents of root sets passed as array arguments.

You can pass any subset (whether it is simple, compound or indexed) as a multidimensional integer indicator array defined over its respective domain sets, indicating whether a particular tuple of domain set elements is contained in the subset (value equals 1) or not (value equals 0). The dimension of such indicator parameters is given by the following set of rules:

*Pass as
indicator
parameter*

- the dimension for a *simple subset* is 1,
- the dimension for a *compound subset* (i.e. a multidimensional relation) is the dimension of the Cartesian product of which the set is a subset,
- the dimension for a *subset of a compound set* is 1,
- the dimension of an *indexed set* is the dimension of the index domain of the set plus 1.

Set arguments passed as an indicator argument can be of input, output, or in-out type. In the latter two cases modifications to the 0-1 values of the indicator parameter are translated back into the corresponding element memberships of the subset.

When you pass set arguments to an external DLL, AIMMS will assume no default translation methods when the set is passed as an integer array, as each type of set does not allow every translation method. For integer set arguments you should therefore always specify one of the translation modifiers `ordinalnumber`, `elementnumber` or `indicator`.

*Set argument
defaults*

Sets can also be passed by an integer handle. AIMMS offers various API functions (see also Section 32.2) to obtain information about the domain of the set, its cardinality and elements, and to add or remove elements to the set.

*Passing set
handles*

11.3 WIN32 calling conventions

The 32-bit Windows environment (WIN32) supports several calling conventions that influence the precise manner in which arguments are passed to a function, and how the return value must be retrieved. When calling an external function or procedure in this environment, AIMMS will *always* assume the WINAPI calling convention. The following macro in C makes sure that the WINAPI calling convention is used. That same macro also makes sure that the function or procedure is automatically exported from the DLL.

*WIN32 calling
conventions*

```
#include <windows.h>
#define DLL_EXPORT(type) __declspec(dllexport) type WINAPI
```

You can add this macro to the implementation of any function that you want to call from within AIMMS, as illustrated below.

```
DLL_EXPORT(double) Cobb_Douglas( int n, double *a, double *c )
{
```

```

    /* Implementation of Cobb_Douglas goes here */
}

```

By default, C++ compilers will perform a process referred to as *name mangling*, modifying each function name in your source code according to its prototype. By doing this, C++ is able to deal with the same function name defined for different argument types. If you want to export a DLL function to AIMMS, however, you must prevent name mangling to take place, ensuring that AIMMS can find the exported function name within the DLL. You can do this by declaring the prototype of the function using the following macro, which accounts for both C and C++.

*Prevent C++
name mangling*

```

#ifdef __cplusplus
#define DLL_EXPORT_PROTO(type) extern "C" __declspec(dllexport) type WINAPI
#else
#define DLL_EXPORT_PROTO(type) extern __declspec(dllexport) type WINAPI
#endif

```

Thus, to make sure that a C++ implementation of `Cobb_Douglas` is exported without name mangling, declare its prototype as follows before providing the function implementation.

```

DLL_EXPORT_PROTO(double) Cobb_Douglas( int n, double *a, double *c );

```

Function declarations like this are usually stored in a separate header file. Note that along with this prototype declaration, you must still use the `DLL_EXPORT` macro in the implementation of `Cobb_Douglas`.

When your external DLL requires initialization statements to be executed when the DLL is loaded, or requires the execution of some cleanup statements when the DLL is closed, you can accomplish this by adding a function `DllMain` to your DLL. When the linker finds a function named `DllMain` in your DLL, it will execute this function when opening and closing the DLL. The following example provides a skeleton `DllMain` implementation which you can directly copy into your DLL source code.

*DLL
initialization*

```

#include <windows.h>

BOOL WINAPI DllMain(HINSTANCE hdll, DWORD reason, LPVOID reserved)
{
    switch( reason ) {
        case DLL_THREAD_ATTACH:
            break;
        case DLL_PROCESS_ATTACH:
            /* Your DLL initialization code goes here */
            break;
        case DLL_THREAD_DETACH:
            break;
        case DLL_PROCESS_DETACH:

```

```

        /* Your DLL exit code goes here */
        break;
    }
    return 1; /* Return 0 in case of an error */
}

```

To prevent name mangling to take place, you can best declare the function `DllMain` as follows.

```

#ifdef __cplusplus
extern "C" BOOL WINAPI DllMain(HINSTANCE hdll, DWORD reason, LPVOID reserved);
#else
BOOL WINAPI DllMain(HINSTANCE hdll, DWORD reason, LPVOID reserved);
#endif

```

11.4 External functions in constraints

AIMMS allows you to use external functions in the constraints of a mathematical program. To accommodate this, AIMMS makes a distinction between function arguments of type `PARAMETER` and arguments of type `VARIABLE`. When a function is executed as part of an expression in an ordinary assignment, AIMMS makes no distinction between both types of arguments. In the context of a mathematical program, however, AIMMS will provide the solver with the derivative information for all variable arguments of the function, while it will not do so for parameter arguments. The actual computation of the derivatives is explained in the next section.

Variable arguments

11.4.1 Derivative computation

Whenever you use external functions with variable arguments in constraints of a mathematical program, the following rules apply.

Functions in constraints

- AIMMS requires that the mathematical program dependent on these constraints be declared as nonlinear.
- All the actual variable arguments must correspond to formal arguments which have been locally declared as `VARIABLES`.

If you fail to comply with these rules, a compiler error will result.

During the solution process of a mathematical program containing such functions, partial derivative information of the function with respect to all the variable arguments must be passed to the solver. AIMMS supports three methods to compute the derivatives of a function:

Providing derivatives

- you provide the actual statements for computing the derivatives as a part of the function declaration,
- AIMMS estimates the derivatives using a simple differencing scheme.

In the DERIVATIVE CALL attribute of an external function you can specify the call to the DLL procedure or function, to which the derivative computation must be relayed. The syntax of the DERIVATIVE CALL attribute is the same as that of the BODY CALL, and is most conveniently completed using the wizard in the Model Explorer.

The DERIVATIVE CALL attribute

If the nonlinear solver only needs a function value, AIMMS will simply call the function specified in the BODY CALL attribute. If the nonlinear solver requests derivative information as well, AIMMS will only call the function specified in the DERIVATIVE CALL attribute, and require that this function compute the function value as well. By combining these two computations in a single call, AIMMS allows you to take advantage of any possible optimization that can be obtained in your code from computing the function value and derivative at the same time.

Function value and derivative

For every function argument which is a variable, you must assign the partial derivative value(s) to the .Derivative suffix of that variable. Note that this will have an impact on the number of indices. If the result of a block-valued function is m -dimensional, the derivative information with respect to an n -dimensional variable argument will result in an $(m + n)$ -dimensional identifier holding the derivative.

The .Derivative suffix

Consider a function f with an index domain (i_1, \dots, i_m) and a variable argument x with index domain (j_1, \dots, j_n) . Then the matrix with partial derivatives of f with respect to the argument x must be provided as assignments to the suffix x .Derivative $(i_1, \dots, i_m, j_1, \dots, j_n)$. Each element of this identifier represents the partial derivative

Abstract example

$$\frac{\partial f(i_1, \dots, i_m)}{\partial x(j_1, \dots, j_n)}$$

Consider the Cobb-Douglas function discussed above. Although AIMMS is capable of computing its partial derivatives automatically, you may verify that the derivative with respect to argument c_i can also be written more compactly as follows:

Cobb-Douglas function revisited

$$\frac{\partial q}{\partial c_i} = \frac{a_i}{c_i} CD(c_1, \dots, c_k)$$

Consider the following C function Cobb_Douglas_Der which computes the Cobb-Douglas function and, if required, also the partial derivatives with respect to the input argument c . The function Cobb_Douglas_No_Der is added to support computation of the Cobb-Douglas function without derivatives.

Implementation in C

```
double Cobb_Douglas_Der( int n, double *a, double *c, double *c_der ) {
    int i;
    double CD = 1.0 ;
```


When you have not specified the DERIVATIVE CALL attribute to compute the derivatives of an external function yourself, AIMMS employs a simple differencing scheme to estimate the derivatives. For example, if AIMMS requires the derivative of a function $f(x_1, x_2, \dots, x_k)$ at the point $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_k)$, then AIMMS will approximate each partial derivative as follows:

$$\frac{\partial}{\partial x_i} f(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_k) \approx \frac{f(\bar{x}_1, \dots, \bar{x}_i + \varepsilon, \dots, \bar{x}_k) - f(\bar{x}_1, \dots, \bar{x}_k)}{\varepsilon}$$

where ε is the current value of the global option Differencing_Delta.

While the numerical differencing scheme does not require any action from the user, there are two distinct disadvantages.

- First of all, numerical differencing is not always a stable process, and the results may not be accurate enough. As a result, a nonlinear solver may have trouble converging to a solution.
- Secondly, the process can be computationally very expensive.

In general, it is recommended that you do not rely on numerical differencing. This is especially the case when the function body is quite extensive, or when the function, at the individual level, has a lot of variable arguments or contains conditional loops.

*Numerical
differencing*

*Disadvantages
of numerical
differencing*

11.5 C versus FORTRAN conventions

For any external procedure or function you can specify whether the DLL procedure or function to which the execution is relayed, is written in C-like languages (such as C and C++) or FORTRAN (see also Section 11.2). For FORTRAN code AIMMS will make sure that

- scalar values are always passed by reference (i.e. as a pointer), and
- multidimensional arrays are ordered in a FORTRAN-compatible manner.

By default, AIMMS will use C conventions when passing arguments to the DLL procedure or function.

AIMMS will not directly translate strings into FORTRAN format, because most FORTRAN compilers use their own particular string representation. Thus, if you want to pass strings to a fortran subroutine, you should write your own C interface which converts C strings into the format appropriate for your FORTRAN compiler.

*Language
conventions*

Strings excluded

When a multidimensional parameter (or parameter slice) is specified as a array argument to an external procedure, AIMMS passes an array of the specified type which is constructed as follows. If the actual argument has n remaining (i.e. non-sliced) dimensions of cardinality N_1, \dots, N_n , respectively, then the associated values are passed as a (one-dimensional) array of length $N_1 \cdot \dots \cdot N_n$. The value associated with the tuple (i_1, \dots, i_n) is mapped onto the element

$$i_n + N_n(i_{n-1} + N_{n-1}(\dots(i_2 + N_2 i_1) \dots))$$

for running indices $i_j = 0, \dots, N_j - 1$ (C-style programming). For PASCAL-like languages (with indices running from $1, \dots, N$) all running indices in this formula must be decreased by 1, and the final result increased by 1. This ordering is compatible with the C declaration of e.g. the multidimensional array

```
double arr[N1][N2]...[Nn];
```

The C function ComputeAverage defined below computes the average of a 2-dimensional parameter $a(i, j)$ passed as an argument in AIMMS.

```
DLL_EXPORT(void) ComputeAverage( double *a, int card_i, int card_j, double *average )
{ int i, j;
  double sum_a = 0.0;

  #define __A(i,j)  a[j + i*card_j]

  for ( i = 0; i < card_i; i++ )
    for ( j = 0; j < card_j; j++ )
      sum_a += __A(i,j);

  *average = sum_a / (card_i*card_j);
}
```

Within your AIMMS model, you can call this procedure via an external procedure declaration ExternalAverage defined as follows.

```
EXTERNAL PROCEDURE
  identifier      : ExternalAverage
  arguments       : (x,res)
  DLL name       : "Userfunc.dll"
  body call      : ComputeAverage(double array: x, card: i, card: j, double scalar: res)
```

where the argument x and res are declared as

```
PARAMETER:
  identifier      : x
  index domain    : (i,j)
  property       : Input ;
PARAMETER:
  identifier      : res
  property       : Output ;
```

*Array
dimensions and
ordering*

*Multidimen-
sional example
in C*

When you specify the FORTRAN language convention for an external procedure, AIMMS will order the array passed to the external procedure such that the tuple (i_1, \dots, i_n) is mapped onto the element

Fortran array ordering

$$i_1 + N_1(i_2 - 1 + N_2(\dots(i_{n-1} - 1 + N_{n-1}(i_n - 1))\dots))$$

for running indices $i_j = 1, \dots, N_j$. This is compatible with the default storage of multidimensional arrays in FORTRAN, and allows you to access such array arguments using the ordinary multidimensional notation.

Consider a parameter $a(i, j)$, where the index i is associated with the set $\{1, 2\}$ and j with the set $\{1, 2, 3\}$. When this parameter is passed as a array argument to an external procedure, the resulting array (as a one-dimensional array with 6 elements) is ordered as follows in the C convention (default).

Example

Element #	0	1	2	3	4	5
Value	$a(1,1)$	$a(1,2)$	$a(1,3)$	$a(2,1)$	$a(2,2)$	$a(2,3)$

With the FORTRAN language convention, the ordering is changed as follows.

Element #	1	2	3	4	5	6
Value	$a(1,1)$	$a(2,1)$	$a(1,2)$	$a(2,2)$	$a(1,3)$	$a(2,3)$