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.

AIMMS B.V.
Diakenhuisweg 29-35
2033 AP Haarlem
The Netherlands
Tel.: +31 23 5511512

AIMMS Inc.
11711 SE 8th Street
Suite 303
Bellevue, WA 98005
USA
Tel.: +1 425 458 4024

AIMMS Pte. Ltd.
55 Market Street #10-00
Singapore 048941
Tel.: +65 6521 2827

AIMMS
SOHO Fuxing Plaza No.388
Building D-71, Level 3
Madang Road, Huangpu District
Shanghai 200025
China
Tel.: ++86 21 5309 8733

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

Chapter 11

External Procedures and Functions

Why call external procedures

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

- 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

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.

11.1 Introduction

Getting started

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 34.1 contains a more elaborate example of an external procedure which uses AIMMS API functions to obtain additional information about the passed arguments.

External procedures and functions

The interface to external procedures and functions is arranged through special `ExternalProcedure` and `ExternalFunction` 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.

The Cobb-Douglas function

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.

```
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".

Linking to AIMMS

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

```
ExternalFunction CobbDouglasExternal {
    Arguments     : (a,c);
    Range         : nonnegative;
    DLLName       : "Userfunc.dll";
    ReturnType    : double;
    BodyCall      : 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 145, 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 InputFactors {
    Index        : f;
}
Parameter a {
    IndexDomain  : f;
}
Variable c {
    IndexDomain  : f;
}
```

Explanation

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.

Calling external functions

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.

```
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.

Use in constraints

Unlike internal functions, external functions can be called inside constraints. To accomplish this, the declaration has to be extended with a DerivativeCall 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 DerivativeCall 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.

Setting up external libraries

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.

Save library as include file

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.

Hiding the interface

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.

11.2 Declaration of external procedures and functions

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.

Attribute	Value-type	See also page
DllName	*string, file-identifier*	
ReturnType	integer, double	
Property	FortranConventions, UndoSafe	144
BodyCall	*external-call*	
DerivativeCall	*external-call*	

Table 11.1: Additional attributes of external procedures and functions

The DllName attribute

With the mandatory DllName 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.

Search path

If you only specify a DLL name, AIMMS will search for the DLL in all directories in the AIMMSUSERDLL environment variable, and the PATH environment variable on Windows, or the LD_LIBRARY_PATH environment variables on Linux, respectively. In addition, on Windows, AIMMS will also search for the DLL in the project folder. If you specify a relative path including a folder (possibly ./), AIMMS will take this path relative to the project folder. If you specify an absolute path, AIMMS will try to open the DLL at the specified location.

File identifier and unit conventions

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 32.8). 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.

Default argument scaling

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 32.4.1.

The ReturnType attribute

The ReturnType 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 ExternalProcedure, or as the (numerical) function value of the ExternalFunction, whichever is applicable. If you do not specify the ReturnType attribute, AIMMS will discard any value returned by the function.

Restricted use

You cannot directly use the returned value of a DLL function as the function value of an ExternalFunction 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.

Example

Consider a C function Cobb_Douglas_Arg with prototype

```
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 ExternalFunction declaration provides a link with Cobb_Douglas_Arg and obtains its function value via the argument list.

```
ExternalFunction CobbDouglasArgument {
    Arguments     : (a,c);
    Range         : nonnegative;
    DllName       : "Userfunc.dll";
    BodyCall      : {
        Cobb_Douglas_Arg( card : InputFactors, array: a, array: c,
            scalar: CobbDouglasArgument );
    }
}
```

The Property attribute

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.

Formal argument types

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:

- simple and compound Sets,
- scalar and indexed Parameters,
- scalar and indexed Variables (external functions only), and
- Handles (external procedures only).

Argument handling

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).

Handle arguments

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 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 34).

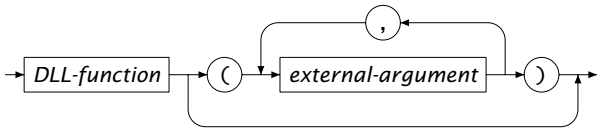
The BodyCall attribute

In the mandatory BodyCall attribute you must specify the call to the DLL procedure or function, to which the execution of the ExternalProcedure or Function must be relayed. Such an external call specifies:

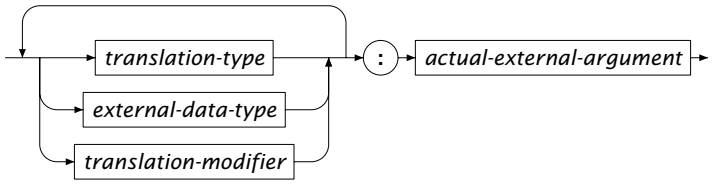
- the name of the DLL function or procedure that must be called, and
- how the actual AIMMS arguments must be translated into arguments suitable for the DLL function or procedure.

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

external-call : *Syntax*



external-argument :



Mandatory translation type

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.

- **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 `BodyCall` 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 34) 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.

Actual external argument

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

- a reference to a formal argument of the ExternalProcedure 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 ExternalProcedure 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 BodyCall attribute (for the literal translation type).

Input-output type

For every formal argument of an ExternalProcedure, you can specify its associated *input-output* type through the Input, InOut (default) or Output properties in the Propert 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**: 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 ExternalFunction arguments are Input by definition. The return value of an ExternalProcedure and the function value of an ExternalFunction are considered as an (implicit) Output argument when passed to the DLL function as an external argument.

External data type

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	Passed as
integer	4-byte (signed) integer
double	8-byte double precision floating number
string	C-style string
integer8	1-byte (signed) integer
integer16	2-byte (signed) integer
integer32	4-byte (signed) integer

Table 11.2: External data types

Allowed combinations

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 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

Passing array arguments

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.

Encoding of string arguments

The strings communicated with your DLL have an encoding. This encoding is set by the option external_string_character_encoding, which has a default of UTF8. This option can be overridden by using the Encoding attribute of string parameters, similar to the Encoding attribute of a File, see Page 499. On Windows, using the encoding UTF-16LE and on Linux, using the encoding UTF-32LE, the strings are passed as a wchar_t* array, otherwise the strings are passed as a char * array.

Output string arguments

When you pass a scalar or multidimensional output string argument, AIMMS will pass a single char buffer of fixed length, or an array of such buffers. The length is determined by the option external function string buf size. The default of this option is 2048. 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.

Full versus sparse data transfer

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 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 34.4 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.

Translation modifiers . . .

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

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

are passed.

. . . for special values

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 ±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 34.4 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 handles

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:

- `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 25.3). 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 34.4.

... for set elements

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:

- 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.

Passing element parameters

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.

When to use

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.

Passing set arguments

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.

Pass as one-dimensional array

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

- 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.

Pass as indicator parameter

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:

- 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.

Set argument defaults

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`.

Passing set handles

Sets can also be passed by an integer handle. AIMMS offers various API functions (see also Section 34.2) to obtain information about the domain of the set, its cardinality and elements, and to add or remove elements to the set.

11.3 WIN32 calling conventions

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.

```
#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 */
}
```

Prevent C++ *name mangling*

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++.

```
#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.

DLL initialization

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.

```
#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

Variable arguments

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.

11.4.1 Derivative computation

Functions in constraints

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

- 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.

Providing derivatives

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:

- 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.

The DerivativeCall attribute

In the DerivativeCall 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 DerivativeCall attribute is the same as that of the BodyCall, and is most conveniently completed using the wizard in the Model Explorer.

Function value and derivative

If the nonlinear solver only needs a function value, AIMMS will simply call the function specified in the BodyCall attribute. If the nonlinear solver requests derivative information as well, AIMMS will only call the function specified in the DerivativeCall 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.

The .Derivative suffix

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.

Abstract example

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

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

Cobb-Douglas function revisited

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:

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

Implementation in C

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.

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

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

  /* Check if derivatives are needed */
  if ( c_der )
    for ( i = 0; i < n; i++ )
      c_der[i] = CD * a[i] / c[i] ;

  return CD;
}

double Cobb_Douglas_No_Der( int n, double *a, double *c ) {
  return Cobb_Douglas_Der( n, a, c, NULL );
}
```

Always skip unwanted derivatives

Note that in the above example the derivative computation is skipped whenever the pointer `c_der` is null. You should *always* check for this condition when implementing a derivative computation, because AIMMS will pass a null pointer (and hence reserve no memory for storing the derivative) whenever the corresponding actual argument is not a variable but a parameter.

... in FORTRAN *code*

When an internal function makes a call to a FORTRAN procedure to compute derivative values, then it is not so easy to discover the presence of null pointer argument. To overcome this, you can call your FORTRAN procedure from within a wrapper function written in C, and provide your FORTRAN code with the information whether or not derivatives need to be computed for a particular variable argument via an additional argument to your FORTRAN routine.

Passing derivative arguments

To pass the partial derivatives computed in the external procedure back to AIMMS, the argument list of the external procedure called in the `Derivative` attribute of the internal function should contain arguments for the `.Derivative` suffices of all variable arguments. AIMMS will implicitly consider such derivative arguments as `Output` arguments. They can be passed either as a full array

or as an integer handle. In the latter case AIMMS API functions have to be used to pass back the relevant partial derivatives (see also Chapter 34).

Example continued

The following external function declaration provides an interface to the above Cobb-Douglas function with derivative computations, which is ready to be used both inside and outside the context of constraints.

```
ExternalFunction CobbDouglasPlusDerivative {
    Arguments       : (a,c);
    Range           : nonnegative;
    DLLName         : "Userfunc.dll";
    ReturnValue     : double;
    BodyCall        : Cobb_Douglas_No_Der( card : InputFactors, array: a, array: c );
    DerivativeCall  : {
        Cobb_Douglas_Der( card : InputFactors, array: a,
            array: c, array: c.Derivative );
    }
}
```

Numerical differencing

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

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

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

Disadvantages of numerical differencing

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.

11.5 C versus FORTRAN conventions

Language 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.

Strings excluded

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.

Array dimensions and ordering

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 $N_1, \ldots, N_n$, respectively, then the associated values are passed as a (one-dimensional) array of length $N_1 \cdots N_n$. The value associated with the tuple $(i_1, \ldots, i_n)$ is mapped onto the element

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

for running indices $i_j = 0, \ldots, N_j - 1$ (C-style programming). For PASCAL-like languages (with indices running from $1, \ldots, N$) 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];
```

Multidimensional example in C

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.

```
ExternalProcedure ExternalAverage {
    Arguments     : (x,res);
    DLLName       : "Userfunc.dll";
    BodyCall      : ComputeAverage(double array: x, card: i, card: j, double scalar: res);
}
```

where the argument x and res are declared as

```
Parameter x {
    IndexDomain   : (i,j);
    Property      : Input;
}
Parameter res {
    Property      : Output;
}
```

Fortran array ordering

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 $(i_1, \ldots, i_n)$ is mapped onto the element

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

for running indices $i_j = 1, \ldots, N_j$. 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.

Example

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).

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)