Automatic Code Generation With DAEPACK
Version 1.1

DRAFT

John E. Tolsma

December 19, 2003

Contents

Chapter 1

DAEPACK Code Generation Overview

This manual describes all of the code generation capabilities of DAEPACK.

Additional symbolic information necessary when performing state of the art computations.

Incredibly difficult to do manually for all but the most trivial cases, this is particularly true when dealing with legacy and third party codes.

Input is evaluation program. Output is a modified evaluation computing some desired quantity. Describe evaluation program.

```
>> daepack -s spec--file source1.f source2.f source3.f
```

1.1 Summary of Code Generation Specification File

When using the command–line version of DAEPACK (as well as some GUI versions) the user controls what code is to be generated with the *specification file*. The specification file is a text file containing one or more code generation *blocks*, with each block containing two or more code generation *options*. The basic form of a code generation block is as follows:
GENERATE type–keyword
option–keyword : *option value(s)*
option–keyword : *option value(s)*
⋮
option–keyword : *option value(s)*
END

That is, an option block begins with the keyword GENERATE followed by the keyword corresponding to the type of code that is to be generated and ending with the keyword END. In between on separate lines are a list of options. Each line contains an option keyword, a semi–colon, and a list of one or more option values (some options require lists of values, others simple require single values. Tables 1.1 and 1.2 contain the code generation types and code generation options, respectively, currently supported by DAEPACK. Each specification file may contain an arbitrary number of code generation blocks. The code transformation described by a given block is applied to the code which has been previously translated by

DAEPACK. The code transformation described by one block cannot be applied to the code generated by a block contained in the same specification file (regardless of which order the blocks appear). Multiple code transformations must be performed by multiple separate invocations of DAEPACK. It is possible to translate two or more completely separate evaluation programs and have multiple code generation blocks apply to different evaluation programs. The user must be cautious, however, since if there are name conflicts or other ambiguities within the translated the behavior of DAEPACK is undefined. Inspection of the symbol table may be necessary to identify the translation of multiple program units (e.g., subroutines or functions) with the same name.

Table 1.1: Summary of various code transformations possible with DAEPACK.

Specification file keyword	Description of transformation performed
DERIVS	Jacobian-matrix and matrix-Jacobian products and sparse Jacobian matrices
COMPLEX_STEP_DERIVS	Approximation of Jacobian by complex–step method
TAYLORMODEL	Arbitrary order, multivariate Taylor series expansions with rigorous bounding of remainder term
SPARSITY	Sparsity patterns
DISCONLOCK	Extraction of discontinuity functions and locked model evaluation
CONVEXIFY	Convex relaxations of nonlinear functions
INTERVALEXT	Natural interval extensions
COMPLEXIFY	Complex-valued version of original code (more than simply replacing all real types with complex types)
CALLGRAPH	Generates call graph in format used by program `dot` (http://www.research.att.com/sw/tools/graphvis/dotguide.pdf)

1.2 Common Code Generation Options

Several of the code generation options shown in Table 1.2 apply to all of the code transformations performed by DAEPACK. This section describes the common options.

1.2.1 ROOT option

DAEPACK allows you to perform the desired code transformations on fairly arbitrary Fortran source code (there are restrictions in some cases which will be described in subsequent chapters). However, it is assumed that regardless of how complex the code is there is a single entry point. That is, there is a single subroutine or function that is called to perform the model evaluation. This subroutine or function is referred to as the *root program unit* and is specified with the keyword ROOT. For example, the following line in the specification file,

```
ROOT : res
```

specifies the root program unit to the subroutine or function named `res`.

1.2.2 OUTFILE option

Not only may the evaluation program be very complex, but it may also be defined within an arbitrary number of source files. The generated code, however, will be saved in a single source file with name specified by the OUTFILE option. For example, the following line in the specification file,

```
OUTFILE : newsource.f
```

will result in the generated source code being stored in the file `newsource.f`. This file will be saved in the directory where DAEPACK is evoked (*future version of DAEPACK will use environment variables to determine where to store the generated source*).

1.2.3 SUFFIX option

During the course of every code transformation process, certain subroutines, functions, and other program constructs will be modified and augmented with additional instructions for performing the desired computation (e.g., computing derivative values). Since the original code will in general be used with the newly generated code, it is necessary to modify the new symbol names to avoid conflicts. In the case of subroutines, functions, entry points, and common blocks the new name will be generated by appending a suffix to the original name. The suffix used is specified by the SUFFIX option. For example, the following line in the specification file,

```
SUFFIX : _m
```

specifies the suffix to be `_m`. Suppose the root program unit of the original evaluation program is subroutine named `res` and we are generating derivative code, then DAEPACK will generate a new subroutine named `res_m` that is called to compute the desired derivative values.

In some situations, the specified suffix may result in name conflicts (in the case above, the original code may already contain a subroutine named `res_m`). Currently DAEPACK will simply check for the possibility of a name conflict and simply issue a warning message; the code will still be generated as usual. It is up to the user to check for true conflicts.

1.2.4 PREFIX option

During the course of the code transformation process, certain variables will be introduced that are associated with variable in the original evaluation program. What these new correspond to will be described on a case by case basis in the following chapters, however, what is important here is that the new variable is associated with an existing variable. As above, the name of this new variable must be created to avoid name conflicts with other variables. This is achieved by prepending the variable name with a prefix. The prefix used is specified by the PREFIX option. For example, the following line in the specification file,

```
PREFIX : m_
```

specifies the prefix to be `m_`. Again, suppose we are generating derivative code (using the forward mode and propagating dense vectors – described later) and there is a variable in the original code named `z` which depends on independent variables, then DAEPACK will generate a new array variable named `m_z` which holds the values of the *directional derivative* of `z`.

As before, the specified prefix may result in name conflicts. Currently DAEPACK will simply check for the possibility of a name conflict and simply issue a warning message; the code will still be generated as usual. It is up to the user to check for true conflicts.

1.2.5 AUX_PREFIX option

In some situations new program variables will be introduced during the course of the code transformation process. In contrast to the cases above, these new symbols do not correspond to any symbol in the original code (examples include, new loop control variables, intermediate variables, new return values, etc.). Once again the names of these new symbols should be selected to avoid name conflicts with other program symbols. This is achieved by prepending the name of an intermediate variable with a prefix. This prefix is specified by the AUX_PREFIX (auxiliary variable prefix) option. The following line in the specification file,

```
PREFIX : aux_
```

specifies the auxiliary variable prefix to be `aux_`. For example, in the code generated for derivative computation, new variables are introduced which hold the values for gradients of the expressions appearing on the right–hand–side of assignments. Using this auxiliary variable prefix, one of these quantities may be named `aux_vbar5`. User control of this option is particularly important when applying multiple code transformation to the same set of code (e.g., generating higher order derivatives). Different prefixes must be selected to avoid conflict between intermediate variables introduced during successive code transformations.

As you might expect, the specified prefix may result with name conflicts. Currently DAEPACK will simply check for the possibility of a name conflict and simply issue a warning message; the code will still be generated as usual. It is up to the user to check for true conflicts.

1.2.6 DO_NOT_PROCESS option

During the code transformation process DAEPACK will perform a *dependency analysis* to identify which program variables are *active*[1]. Any program unit that manipulates active variables (or passes active variables to other program units that manipulate them) will be considered active. New, augmented, program units will be constructed for each active program unit in the original code. However, in some cases there may be program units we do not want to be transformed. Situations where this would occur will be described later. The user can explicitly turn off the analysis of these routines by listing them in the DO_NOT_PROCESS option list. For example, the following line in the specification file,

```
DO_NOT_PROCESS : sub1, sub5, func3, func7
```

specifies that subroutines/functions `sub1`, `sub5`, `func3`, and `func7` will not be analyzed (i.e., treated as "black boxes"). In the generated code, rather than calling new, modified subroutines/functions corresponding to these, the original code will be called. As may be expected, using this option could result in incorrect code being generated (we may neglect some of the operations performed which computing values for the program outputs from the program inputs). The option appears most frequently in the construction of discontinuity–locked code, where it is used to turn off the processing of discrete events within some parts of the code.

[1] By active we mean that a particular variables depends upon some special variable, such as an independent or interval variable. This will be described in more detail on a case by case basis in subsequent chapters.

1.3 Specifying Special Variables

With the exception of generating discontinuity-locked code and call graphs, all code transformations performed by DAEPACK require the user to specify special variables. For example, in the case of generating derivative, Taylor model, and sparsity pattern code, the user must specify which of the program symbols are to be treated as *independent variables* and which are to be treated as *dependent variables*. When generating interval extension or complex–valued code, the user must specify which symbols are to be treated as *interval–valued variables* and *complex–valued variables*[2]. In the case of generating convex relaxations, the user not only specifies independent and dependent variables, but also has the ability to specify which dependent variables are known a priori to be assigned the values of *convex* or *concave functions*.

DAEPACK allows the user to specify these special or distinguished variables with a high degree of flexibility. The syntax for specifying these variables is:

`vartype` : *var–list*

where `vartype` is INDEPENDENT, DEPENDENT, INTERVAL, CONVEX, CONCAVE, or COMPLEX and *var–list* is a comma–separated list of array or scalar names (e.g., `x` or `alpha`), array elements (e.g., `x(1)`, `x(n-4)`, or `x(i**2+3)`, array slices (e.g., `x(1:4)` or `x(n:m-2)`), or arrays with *index sets* (e.g., `x(index(1:4))` or `y(irange(n:m))`). The index set, useful for specifying a variable ordering at run–time, is an integer array where the value of each entry is within the range of the array it indexes (`x` in this example). Within the option list, the index set must be specified with a range (in this example, `1:4` and `n:m`). Specifying variables with array slices or with index sets is not supported in every code transformation. However, these capabilities are particularly useful in transformations involving computations exploiting sparsity (sparse derivative matrix computation, sparsity patterns construction, and Taylor model construction). Here is an example of specifying a set of independent variables:

```
INDEPENDENT : x, y(1), y(3:5), z(indx(1:n)), alpha, beta
```

Suppose `x` is an array with `nx` elements and `alpha` and `beta` are scalars, then the list above corresponds to: `x(1)`, `x(2)`, ..., `x(nx)`, `y(1)`, `y(3)`, `y(4)`, `y(5)`, `z(indx(1))`, `z(indx(2))`, ..., `z(indx(n))`, `alpha`, and `beta`. Note that the components of array `z` that are treated as independent variables will be determined at run–time when values for `indx(1:n)` have been specified.

Symbols appearing in this list can be arguments, common block variables, or even local variables of the root program unit. As shown above, array symbols may be indexed with expressions and/or other symbols, as with the examples, `z(indx(1:n))` and `x(n-4)`. If this is the case, the user must make sure the values of these indexing symbols and/or contents of index arrays have been specified before they are used in the generated code. If these symbols and/or index arrays are arguments of the root program unit or are contained in common blocks within the root program unit then the user must simply make sure they contain the desired values when the transformed code is called. If, however, the symbol values are computed in the body of the root program unit then the user must edit the code to make sure these computations occur and the symbol of interest is set prior to its use within the generated code. **DAEPACK will issue a warning message and instruct the user on what to do if this is the case.**

In some code transformations, the order in which the distinguished variables are present in the option list is important. For example, when generating code for computing the sparse Jacobian matrix or

[2]DAEPACK will perform a dependency analysis to find other variables which must also be treated as interval or complex variables.

sparsity pattern, the user must know what dependent variable is associated with a particular row and what independent variable is associated with a particular column. **With DAEPACK, the ordering of the distinguished variables in the option list is the same as the ordering in the generated code.** For example, if we generate the Jacobian matrix of an evaluation program computing $y = f(x)$ and specify:

```
INDEPENDENT : x(4), x(1), x(9:11)
DEPENDENT   : y(4:6), y(2), y(3)
```

the Jacobian matrix will be returned in the following order:

$$\begin{pmatrix} \frac{\partial y_4}{\partial x_4} & \frac{\partial y_4}{\partial x_1} & \frac{\partial y_4}{\partial x_9} & \frac{\partial y_4}{\partial x_{10}} & \frac{\partial y_4}{\partial x_{11}} \\ \frac{\partial y_5}{\partial x_4} & \frac{\partial y_5}{\partial x_1} & \frac{\partial y_5}{\partial x_9} & \frac{\partial y_5}{\partial x_{10}} & \frac{\partial y_5}{\partial x_{11}} \\ \frac{\partial y_6}{\partial x_4} & \frac{\partial y_6}{\partial x_1} & \frac{\partial y_6}{\partial x_9} & \frac{\partial y_6}{\partial x_{10}} & \frac{\partial y_6}{\partial x_{11}} \\ \frac{\partial y_2}{\partial x_4} & \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_9} & \frac{\partial y_2}{\partial x_{10}} & \frac{\partial y_2}{\partial x_{11}} \\ \frac{\partial y_3}{\partial x_4} & \frac{\partial y_3}{\partial x_1} & \frac{\partial y_3}{\partial x_9} & \frac{\partial y_3}{\partial x_{10}} & \frac{\partial y_3}{\partial x_{11}} \end{pmatrix}.$$

As may be expected, if the names of the arrays are used without subscripts or ranges then the ordering in the Jacobian matrix or sparsity pattern is the same as the ordering of the array (e.g., `x(1)`, `x(2)`, `x(3)`, ...). ***Describe automatic construction of permuted Jacobian matrices using index sets – describe in context of BLKSLV.***

Not all of the code generation options supported by DAEPACK can utilize this flexible variable description. For example, when generating interval extension code if `x(1)` is declared to be an interval then the entire `x` array is treated as an interval vector. These special cases will be described where appropriate throughout this manual.

The remaining chapters of this manual describes each of the different types of codes that may be generated automatically with DAEPACK. As described above, DAEPACK takes as input a collection of one or more Fortran source files containing the evaluation program and produces as output a Fortran source file containing code for performing the desired evaluation. In the discussion that follows we will use the generic Fortran subroutine,

`subroutine sub(`*argument list*`)`

to indicate how the original name and argument list are modified during code transformation. In some cases, a specific example will be necessary to illustrate the changes. We will use the subroutine interface for the DAE residual evaluation subroutine used by the DAEPACK family of numerical integrators DSL48S, DSL48E, and DSL48SE:

```
subroutine res(ny,t,y,ydot,delta,ires,ichvar,rpar,ipar).
```

Table 1.3 contains a summary of the argument in this subroutine. This information is relevant when discussing the argument list modification during code transformation in the following chapters.

Table 1.2: Summary of code generation options.

Specification file keyword	Description of option
ROOT	Name of root program unit (e.g., subroutine, function, etc.) in the target evaluation program
SUFFIX	Suffix appended to certain symbols in the new code
PREFIX	Prefix prepended to certain symbols in the new code
AUX_PREFIX	Prefix used for new variables introduced in the new code
OUTFILE	Name of generated file containing new code
DO_NOT_PROCESS	List of program units in original evaluation program that are to be treated as "black boxes"
INDEPENDENT	List of variables to be treated as independent variables
DEPENDENT	List of variables to be treated as dependent variables
INTERVAL	List of variables to be treated as interval-valued variables
CONVEX	List of variables in code that are assigned the value of functions that are known to be convex
CONCAVE	List of variables in code that are assigned the value of functions that are known to be concave
COMPLEX	List of variables to be treated as complex-valued variables
JACOBIANXMATRIX	Construct derivative code computing Jacobian-matrix or matrix-Jacobian product (only used for code exploiting sparsity) (option value is `true` or `false`)
SPARSESTORAGE	Construct derivative code which computes only derivative values that are not identically non-zero (option value is `true` or `false`)
EXPLOIT_SPARSITY	Same as above
ACCUMULATION_MODE	Mode used for derivative accumulation (e.g., `forward_mode`, `reverse_mode`, etc.)
VECTOR_ACCUMULATION	Construct derivative code that either propagates vectors (if value is `true`) or scalars (if value is `false`)
AD_SVM_TABLESIZE	Sets size of active variable table in sparse vector manager (only used for derivative code that exploits sparsity)
AD_SVM_SEGMENTSIZE	Sets length of memory segment holding sparse vectors (only used for derivative code that exploits sparsity)
AD_SVM_NUMSEGMENTS	Sets number of memory segments holding sparse vectors (only used for derivative code that exploits sparsity)
AD_SVM_WRITESTATS	Generates a file containing information about the sparse vector manager (set this option to the name of the file to be generated) (only used for derivative code that exploits sparsity)
AD_SEED_NUMBER	Sets the maximum number of "seed directions" or "weights" that can be propagated through the code.
LINEARIZE	Linearization of convex relaxations of nonlinear functions (integer value only used for convex relaxation code generation)
CONVEXIFICATION	Convex relaxation method to be employed (only used for convex relaxation code generation)

Table 1.3: Description of arguments in residual evaluator `RES` used by DSL48S/E/SE.

Name	Input/Output	Type	Description
ny	input	integer	Number of DAEs
t	input	double precision	Value of independent variable
y(ny)	input	double precision	State variable values
ydot(ny)	input	double precision	Time derivative values
delta(ny)	output	double precision	DAE residual values
ires	output	integer	Error return code (0=no error)
ichvar	input	integer	Control code (N/A here)
rpar(*)	input	double precision	Real-valued parameters
ipar(*)	input	integer	Integer-valued parameter

Chapter 2

Analytical Derivatives (DERIVS)

2.1 Description

This chapter describes how to use DAEPACK to generate code for computing numerical values for partial derivatives, accurate to computer round-off error. Specifically, these derivative values are free of truncation error (in contrast to finite difference approximations of derivatives). The technique employed by DAEPACK is referred to as *algorithmic (or automatic) differentiation*. A detailed discussion is beyond the scope of this manual however the interested user is encouraged to read additional references, including []. In addition, there are a number of AD tools available in addition to DAEPACK [].

2.2 Derivative Code Generation Options

Section 1.2 describes several options that are common to all code generation types. Table 2.1 contains the DAEPACK specification file options relevant to the generation of derivative code. The default values (if applicable) for these options are also listed. The first six options in Table 2.1 are common to all code transformation and have been described in Section 1.2. The remainder are described here.

2.2.1 INDEPENDENT and DEPENDENT

The syntax of the INDEPENDENT and DEPENDENT options are

`INDPENDENT` : *var–list*

`DPENDENT` : *var–list*

where *var–list* is a comma–separated list of program variables. As may be expected, this list specifies the program variables that are to be treated as independent and dependent variables. Section 1.3 describes how these variables may be specified (e.g., specifying full arrays, components, slices, etc). All variables appearing in this list must be visible in the scope of the root program unit (see Section 1.2.1). If an entry in the list is not visible then a warning message will be issued and that entry will be ignored. Other incorrect syntax errors, such as specifying an expression to be independent, will result in an error.

Table 2.1: Derivative code generation options. Boldface keywords denote options that must be supplied by the user.

Specification file keyword	Default value
ROOT	None
SUFFIX	ad (e.g. "`res`" becomes "`resad`")
PREFIX	ad (e.g. "`x`" becomes "`adx`")
AUX_PREFIX	zzz (e.g. intermediate variable "`zzzvbar1`")
OUTFILE	None
DO_NOT_PROCESS	None
INDEPENDENT	None
DEPENDENT	None
JACOBIANXMATRIX	`false` (compute pure Jacobian matrix)
SPARSESTORAGE	`true` (exploit sparsity)
EXPLOIT_SPARSITY	`true` (exploit sparsity)
ACCUMULATION_MODE	`forward_mode` (forward mode of AD)
VECTOR_ACCUMULATION	True (propagate vectors through code)
AD_SVM_TABLESIZE	Default value described in SVM documentation
AD_SVM_SEGMENTSIZE	Default value described in SVM documentation
AD_SVM_NUMSEGMENTS	Default value described in SVM documentation
AD_SVM_WRITESTATS	Default value described in SVM documentation
AD_SEED_NUMBER	10

2.2.2 SPARSESTORAGE and EXPLOIT_SPARSITY

DAEPACK provides an option for the explicit handling of any sparsity present in the original evaluation program. Specifically, the code will propagate *sparse* vectors through the code and the resulting derivative matrix will be returned in *sparse triplet form*. The sparse vectors manipulated in the code are stored in the *sparse vector manager* (SVM). See the SVM documentation for a description of the data structures and available functions for manipulating sparse vectors. This documentation also describes several parameters that may be fine tuned by the user to improve the performance of the sparse derivative evaluation. The syntax for these options (which are simply synonyms) is

`SPARSESTORAGE` : *true or false*

`EXPLOIT_SPARSITY` : *true or false*

When exploiting sparsity (e.g., SPARSESTORAGE or EXPLOIT_SPARSITY is set to true) then the resulting partial derivative matrix is returned in sparse triplet (or coordinate) form. Four quantities are returned that describe the derivative matrix: `zzzne`, `zzzderivs`, `zzzirn`, and `zzzjcn`, where `zzzne` returns the number of nonzero entries in the Jacobian matrix, `zzzderivs` is an double precision array containing the `zzzne` entries of the derivative matrix, and `zzzirn` and `zzzjcn` contain the row and column numbers, respectively, of the derivative matrix (recall that the variables corresponding to each row and column can be determined by examining the dependent and independent variable lists respectively). The nonzero entries computed are structurally nonzero, not necessarily numerically nonzero. That is,

they are nonzero for any input values for the program variables. Throughout this text, nonzero will indicate structurally nonzero unless stated otherwise. The derivative matrix entry in row `zzzirn(k)` and column `zzzjcn(k)` has numerical value `zzzderivs(k)` (assuming AUX_PREFIX has the value "zzz"). The interface of the transformed code is

`subroutine sub_ad(`*argument list*`,zzzderivs,zzzne,zzzirn,zzzjcn,zzziw)`

where SUFFIX has the value "_ad" and AUX_PREFIX has the value "zzz". The additional argument `zzziw` is an integer array used as workspace during sparse derivative propagation. The size of this array is the largest of the three quantities: number of independent variables, number of dependent variables, and maximum number of *active* variable arguments in any of the transformed program units in the original evaluation program. DAEPACK will issue a message indicating the appropriate size and it is up to the user to provide at least this amount.

In addition to providing sufficient space for `zzziw`, the user must also provide sufficient space to hold the sparse derivative matrix. Unfortunately, unlike `zzziw` the user is not given an a priori estimate of the size of `zzzne`. (Future versions of DAEPACK will provide the user with a means of checking this value, however, this is not currently supported due to backwards compatibility issues.) If any of these arrays are smaller than necessary then memory will be overwritten, most likely with detrimental consequences (i.e., core dump!).

2.2.3 JACOBIANXMATRIX

The syntax of this option is

`JACOBIANXMATRIX` : *true or false*

This option is only valid when generating sparse derivative code (where sparse vectors are propagated through the code). If this option has the value of false then the normal Jacobian matrix is returned (i.e., numerical values of the partial derivatives of the dependent variables with respect to the independent variables). If the value is true then the one of the following quantities are returned:

$$\begin{array}{ll} J(x)X' & \textit{forward accumulation mode} \\ \bar{Y}^T J(x) & \textit{reverse accumulation mode} \end{array} \tag{2.1}$$

where $J(x)$ is the normal Jacobian matrix and X' and $\bar{Y}$ are arbitrary user–supplied conformable matrices. The number of rows in X' or $\bar{Y}$ is specified by the user and the number of columns is equal to the number of independent variables (forward mode) or the number of dependent variables (reverse mode). The interface of the transformed code is

`subroutine sub_ad(`*argument list*`,zzzderivs,zzzne,zzzirn,zzzjcn,`
`zzznscol,zzzseed,zzziw)`

where `zzznscol` is the number of rows in the "seed" or "weight" matrix S and `zzzseed` is the two dimension array containing S. Note that the matrix `zzzseed` must be declared with leading dimension `zzznscol` in the calling program. (For the advanced user, `zzzseed` is declared in the generated code with leading dimension `zzznscol`, so make sure your data is stored correctly in `zzzseed`.) *This option is not recommended.* If Jacobian–matrix and matrix–Jacobian products are desired then the non–sparse versions should be employed. This is due to the fact that the derivative matrix returned will be dense and the additional overhead associated with manipulating sparse vectors is not warranted under these circumstances.

2.2.4 ACCUMULATION_MODE

DAEPACK currently supports two modes of AD for accumulating derivative values, the forward and reverse modes. The syntax for this option is

`ACCUMULATION_MODE` : `forward_mode` *or* `reverse_mode`

As described above, a detailed description of the forward and reverse modes of AD are beyond the scope of this manual and several excellent sources of information are cited. The differences in these to approaches, as well as the differences in the interfaces of the generated code are described in Section 2.3. Currently, there are a number of limitations on the types of code that the reverse mode may be applied. These are describe in more detail in Appendix XXX.

2.2.5 VECTOR_ACCUMUATION

This option is used when propagating dense vectors and the number of "seed directions" or "weight vectors" is known a priori to be one. That is, the gradient of a single dependent variable or a single directional derivative is desired. The syntax for this option is

`VECTOR_ACCUMUATION` : *true or false*

Although the generated code is less general when this option is false, the advantage is that the code will be smaller, more readable, and slightly more efficient[1]. The change in interface of the generated code will be illustrated in Section 2.3.

2.2.6 AD_SVM_TABLESIZE

The DAEPACK specification file allows several of the SVM parameters to set by the user. Although these are described in detail in the SVM documentation, the implications of setting these parameters appropriately is great enough to warrant a discussion here. Generally it is good practice to decouple the underlying implementation from the user interface to minimize the impact of implementation changes. However, for the sake of the performance of the generated code some implementation will be discussed here. Associated with each *active* variable in the generated code is an entry in the SVM holding a sparse vector. An active variable is a program variable that is dependent on some independent variable or influences some dependent variable (this is an overly simple description, but suitable for this discussion). The table used to hold these quantities within the SVM will grow to accommodate as many active variables as necessary. However, performance will be better if the initial size of the table is comparable to the number of actual number of active variables in a given derivative computation. ***The current SVM implementation uses a hash table to store the active variables. Setting the table size to be approximately 50% larger than the number of active variables has shown good performance on average due to fewer "collisions."*** The SVM active variable table size is set as follows:

`AD_SVM_TABLESIZE` : *integer–literal*

Section **??** describes how to generate a file containing a summary of the SVM statistics so this option value can be tuned for a given problem.

This option will insert a call to the SVM function `SetTableSize(isize)` into the generated code. The table size can changed after the code has been generated by editing the call to this routine. This is described both in the SVM documentation and below.

[1] How much more efficient depends on the quality of your compiler and BLAS library.

2.2.7 AD_SVM_SEGMENTSIZE

The sparse vectors associated with each active variable in the SVM are stored in a list of data segments of a given size. As many data segments as necessary will be allocated to store the nonzero entries (remember, structurally nonzero) of the sparse vector. The SVM sparse vector segment size is set as follows:

`AD_SVM_SEGMENTSIZE` : *integer–literal*

Let n denote the full dimensionality of the vector and $\hat{n}$ denote the number of entries actually stored in the SVM. The advantage to selecting a large value for AD_SVM_SEGMENTSIZE (i.e., close or equal to n) is that copying to and from a single data segment is more efficient than if the data was distributed across multiple segments. On the other hand, using larger data segments wastes memory if they are largely unfilled. ***The value selected for AD_SVM_SEGMENTSIZE is obviously problem specific. A good value to use is the average value for $\hat{n}$.*** Section **??** describes how to generate a file containing a summary of the SVM statistics, including the average value for $\hat{n}$.

This option will insert a call to the SVM function `SetSegmentSize(isize)` into the generated code. The segment size can changed after the code has been generated by editing the call to this routine. This is described both in the SVM documentation and below.

2.2.8 AD_SVM_NUMSEGMENTS

As stated above, the sparse vectors held in the SVM are stored in multiple data segments with each segment able to hold AD_SVM_SEGMENTSIZE vector component values. Parameter AD_SVM_NUMSEGMENTS is used to specify how many data segments can be held. The number of SVM segments used to hold sparse vectors is set as follows:

`AD_SVM_NUMSEGMENTS` : *integer–literal*

Usually, a good value for AD_SVM_NUMSEGMENTS is one such that product of of the number of segments and the segment size is equal to the maximum number of nonzero entries expected in any sparse vector stored. If more data segments than AD_SVM_NUMSEGMENTS are required then additional will be allocated, albeit at higher cost than if this value was provided a better value. Again, section **??** describes how to generate a file containing a summary of the SVM statistics that can be examined to determine an appropriate value for this option.

This option will insert a call to the SVM function `SetNumSegments(inum)` into the generated code. The number of segments can changed after the code has been generated by editing the call to this routine. This is described both in the SVM documentation and below.

2.2.9 AD_SVM_WRITESTATS

As mentioned above, this option is used to create a text file containing a summary of the sparse vectors stored in the Sims. (Several SVMs may be used for a given derivative evaluation calculation. Specifically, there is an SVM for potentially every local scope present in the original evaluation program.) For each SVM, this information includes the number of active variables stored, default parameter values, and minimum, maximum, and average number of nonzero entries in the vectors associated with all of the active variables. The syntax for generating the summary file is

`AD_SVM_WRITESTATS` : *file–name*

where *file–name* is an unquoted valid file name.

This option will insert a call to the SVM function `WriteSVMStats(imodel,filename)` into the generated code. The name of the file may be later changed or call to this function removed by editing the generated code. This is described both in the SVM documentation and below. ***Writing the SVM statistics during each call to the derivative code can increase the evaluation time significantly. Make sure to remove or comment out the call to*** *`WriteSVMStats(imodel,filename)`* ***once the necessary information has been obtained.***

2.2.10 AD_SEED_NUMBER

The non-sparse versions of AD produce code that propagates dense vectors through the code. Suppose the original evaluation program computes $y = f(x)$, $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$. Using the forward mode, the following quantity is computed:

$$Y' = J(x)X'$$

where $J(x)$ is the Jacobian matrix and $X' \in \mathbb{R}^{n \times n_p}$ is the "bundle" of directions. Each column of $Y' \in \mathbb{R}^{m \times n_p}$ is the directional derivative of y in the direction contained in the corresponding column of X'. Using the reverse mode, the following quantity is computed:

$$\bar{X}^T = \bar{X}^T + \bar{Y}^T J(x)$$

where $\bar{Y} \in \mathbb{R}^{m \times n_p}$ are "weight" vectors. The option AD_SEED_NUMBER sets the maximum size for n_p. The syntax for this option is

`AD_SEED_NUMBER` : *integer–literal*

This value can, however, be easily changed after the code has already been generated.

2.3 Description of AD Modes and Transformed Interfaces

DAEPACK supports the construction of several types of derivative codes. These are summarized in Table 2.2. The following subsections describe each of the cases in this table. In this section, assume the evaluation program compute $y = f(x)$ where $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$. The independent and dependent variables are x and y, respectively, and the Jacobian matrix, $J(x)$, is equal to $\partial f / \partial x$.

Currently, the reverse mode of automatic differentiation has not been fully implemented. A description of its usage however is provided here nevertheless.

The first four examples in Table 2.2 exploit sparsity by propagating sparse vector data structures through the code. A detailed discussion of these data structures and supporting functions are described in the SVM documentation. The remaining approaches propagate scalars or normal (dense) vectors through the code (represented by arrays in Fortran). If the problem exhibits sparsity (i.e., there are several partial derivative values that are identically zero for any value of the evaluation program inputs), then the propagation of sparse vector will result in fewer arithmetic operations when computing the derivative values. The cost of this however is the additional overhead associated with the manipulation of the non-native, sparse vector data structures. This additional cost includes table look–ups and indirect array addressing. Consequently, the trade–offs between exploitation of sparsity by propagating sparse vectors versus propagating dense vectors is not always clear. The user is encouraged to experiment

Table 2.2: AD modes supported by DAEPACK.

Example number	Accumulation direction	Vector storage	"seed"/"weight" matrix	Accumulated quantities		Computed quantity
1	forward	sparse	no	sparse vectors		$J(x)$
2	forward	sparse	yes	sparse vectors		$J(x)X'$
3	reverse	sparse	no	sparse vectors		$J(x)$
4	reverse	sparse	yes	sparse vectors		$Y^T J(x)$
5	forward	dense	N/A	vectors		$J(x)X'$
6	forward	dense	N/A	scalars		$J(x)x'$
7	reverse	dense	N/A	vectors		$X^T + Y^T J(x)$
8	reverse	dense	N/A	scalars		$\bar{x}^T + \bar{y}^T J(x)$

Evaluation program computes $y = f(x)$, $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$.
$J(x) = \partial f / \partial x$ is $m \times n$ Jacobian matrix of $f(x)$.
$X' \in \mathbb{R}^{n \times n_p}$ is an arbitrary valued user–supplied matrix.
$x' \in \mathbb{R}^n$ is an arbitrary valued user–supplied vector.
$\bar{Y} \in \mathbb{R}^{m \times n_p}$ is an arbitrary valued user–supplied matrix.
$\bar{y} \in \mathbb{R}^m$ is an arbitrary valued user–supplied vector.
$\bar{X} \in \mathbb{R}^{n \times n_p}$ is an arbitrary valued user–supplied matrix.
$\bar{x} \in \mathbb{R}^n$ is an arbitrary valued user–supplied vector.
Number of "seed" directions or "weighting" vectors, n_p, is set by user.

between the various approaches supported by DAEPACK. In practice, the sparse vector approaches tend to do better on *very* large and *very* sparse problems where the sparsity pattern has little structure and a relatively large number of **structurally orthogonal** columns and rows.

Some discussion of the temporal and spatial complexity is described here. A particularly good description of this complexity can be found in []. Suppose we are computing $y = f(x)$ as described above. Let $\text{cost}(\cdot)$ and $\text{space}(\cdot)$ denote the computational cost (or time) and memory, respectively, associated with computing some quantity. For example, $\text{cost}(f(x))$ may be the CPU time required to compute numerical values for y and $\text{space}(f(x))$ may be the number of bytes used by the evaluation program during this computation. Using the forward mode of automatic differentiation, the cost and space required to compute a single directional derivative, $J(x)x'$, is described by the following:

$$\begin{aligned} \text{cost}(J(x)x') &\leq 3 \cdot \text{cost}(f(x)) \qquad \text{Forward mode} \\ \text{space}(J(x)x') &= \text{space}(f(x)). \end{aligned} \tag{2.2}$$

By propagating several directions simultaneously, we have the following bounds:

$$\begin{aligned} \text{cost}(J(x)X') &\leq (2.5n_p + 1) \cdot \text{cost}(f(x)) \qquad \text{Forward mode} \\ \text{space}(J(x)X') &= n_p \cdot \text{space}(f(x)), \end{aligned} \tag{2.3}$$

where n_p is the number of columns of X' (i.e., the number of seed directions propagated simultaneously. Notice that it is better to propagate several vectors simultaneously rather than separate evaluations with different x'.

Using the reverse mode of automatic differentiation, the cost and space required to compute $\bar{y}^T J(x)$ is described by the following:

$$\begin{aligned} \text{cost}(\bar{y}^T J(x)) &\leq 3 \cdot \text{cost}(f(x)) \qquad \text{Reverse mode} \\ \text{space}(\bar{y}^T J(x)) &\propto \text{cost}(f(x)). \end{aligned} \tag{2.4}$$

Setting $\bar{y}$ to Cartesian basis vector e_k results in the computation of the gradient of the k-th component of $f(x)$. Notice that this gradient is computed at a small fixed multiple of the cost of computing the underlying function $f(x)$, regardless of the dimension of x! This is referred to as the *cheap gradient result* []. The unfortunate characteristic of the reverse mode is that the spatial requirements are proportional to the run–time of the evaluation program. Approaches for mitigating this issue are described in []. By propagating several "weights" simultaneously, we have the following bounds:

$$\begin{aligned} \text{cost}(\bar{Y}^T J(x)) &\leq (1.5 n_p + 1) \cdot \text{cost}(f(x)) \qquad \text{Reverse mode} \\ \text{space}(\bar{Y}^T J(x)) &\propto n_p \cdot \text{cost}(f(x)), \end{aligned} \tag{2.5}$$

where n_p is the number of columns of $\bar{Y}$ (i.e., the number of weights propagated simultaneously. As with the forward mode, it is better to propagate several vectors simultaneously rather than separate evaluations with different $\bar{y}$.

Now suppose the sparsity pattern of the Jacobian matrix has on average $\hat{n}$ structurally nonzero entries per row and $\hat{m}$ structurally nonzero entries per column. The complexity for the sparse vector accumulation approaches are:

$$\begin{aligned} \text{cost}(J(x)) &\leq (2.5\hat{n} + 1) \cdot \text{cost}(f(x)) + \text{overhead} \qquad \text{Sparse forward mode} \\ \text{space}(J(x)) &= \hat{n} \cdot \text{space}(f(x)) + \text{overhead}, \end{aligned} \tag{2.6}$$

$$\begin{aligned} \text{cost}(J(x)) &\leq (1.5\hat{m} + 1) \cdot \text{cost}(f(x)) + \text{overhead} \qquad \text{Sparse reverse mode} \\ \text{space}(J(x)) &\propto \hat{m} \cdot \text{cost}(f(x)), + \text{overhead}, \end{aligned} \tag{2.7}$$

The temporal and spatial overhead are associated with the manipulation of the sparse vectors in the SVM and can be reduced by proper tuning of the SVM parameters (see above).

2.3.1 Example 1: $J(x)$, Sparse Forward Mode

Using the following code generation options:

```
ACCUMULATION_MODE   : forward_mode
EXPLOIT_SPARSITY    : true
JACOBIANXMATRIX     : false
```

will result in code that computes a sparse Jacobian matrix, $J(x)$, using the forward mode of automatic differentiation. Sparse vectors are propagated in contrast to native arrays. Note that when sparse storage is requested (either with EXPLOIT_SPARSITY or SPARSESTORAGE) it is not necessary to specify vectors are to be accumulated.

Assume SUFFIX is "_ad" and AUX_PREFIX is "zzz". The generated code will have interface:

`subroutine sub_ad(`*argument list*`,zzzderivs,zzzne,zzzirn,zzzjcn,zzziw)`

The additional arguments have been described in Section 2.2.2.

2.3.2 Example 2: $J(x)X'$, Sparse Forward Mode

Using the following code generation options:

```
ACCUMULATION_MODE   : forward_mode
EXPLOIT_SPARSITY    : true
JACOBIANXMATRIX     : true
```

will result in code that computes the Jacobian matrix post–multiplied by an arbitrary user–supplied conformable matrix, X'.

Assume SUFFIX is "_ad" and AUX_PREFIX is "zzz". The generated code will have interface:

```
subroutine sub_ad(argument list,zzzderivs,zzzne,zzzirn,zzzjcn,
                  zzznscol,zzzseed,zzziw)
```

The additional arguments have been described in Sections 2.2.2 and 2.2.3. The "seed" matrix in the argument list of `sub_ad`, is dimensioned as `zzzseed(zzznscol,*)` and is equal to the ***transpose*** of matrix X'. (The argument `zzznscol` refers to the number of "seed" matrix columns, or rows of the transposed matrix.) The reason for requesting the transpose of the desired matrix is due to the fact that in Fortran, two dimensional matrices are stored in a *column–major* format. Memory access is faster by accessing columns at a time rather than a row at a time. The number of columns of `zzzseed(zzznscol,*)` is equal to the number of independent variables, i.e., dimension of x (see Section 2.2.1).

In the current implementation, any sparsity in the seed matrix in the seed matrix is not exploited. Thus, even though the Jacobian matrix may be sparse, the product of the Jacobian and the matrix X' will be dense. Consequently, this option is not recommended due to the overhead of propagating dense vectors using data structures to represent sparse vector. The approach shown in example 5 below is a far better choice. Future implementation will support a sparse representation for X', making this approach suitable in some situations.

2.3.3 Example 3: $J(x)$, Sparse Reverse Mode

Using the following code generation options:

```
ACCUMULATION_MODE   : reverse_mode
EXPLOIT_SPARSITY    : true
JACOBIANXMATRIX     : false
```

will result in code that computes a sparse Jacobian matrix, $J(x)$, using the reverse mode of automatic differentiation. As in example 1, sparse vectors are propagated in contrast to native arrays. The interface of the generated code is identical to that shown in example 1.

2.3.4 Example 4: $\bar{Y}^T J(x)$, Sparse Reverse Mode

Using the following code generation options:

```
ACCUMULATION_MODE   : reverse_mode
EXPLOIT_SPARSITY    : true
JACOBIANXMATRIX     : true
```

will result in code that computes the Jacobian matrix pre–multiplied by an arbitrary user–supplied conformable matrix, $\bar{Y}$. As in example 2, the number of "weights" or columns of $\bar{Y}$ is argument `zzznscol`. The matrix `zzzseed(zzznscol,*)` is the transpose of $\bar{Y}$. The number columns of `zzzseed(zzznscol,*)` is equal to the number of dependent variables, i.e., dimension of y (see Section 2.2.1). For the same reasons described in example 2, this approach is not currently recommended. If the quantity $\bar{Y}^T J(x)$ is desired then either use the approach described in example 8 or compute the sparse Jacobian and do the pre–multiplication explicitly, possibly exploiting sparsity in $\bar{Y}$.

2.3.5 Example 5: $J(x)X'$, Forward Mode

Using the following code generation options:

```
ACCUMULATION_MODE   : forward_mode
EXPLOIT_SPARSITY    : false
VECTOR_ACCUMULATION : true
```

will result in code that computes the Jacobian matrix post–multiplied by an arbitrary user–supplied conformable matrix, X'.

In contrast to example 2, sparse vector data structures are not used in the generated code. This results in faster vector manipulation at the expense of performing potentially unnecessary operations on hard zeros. The interface of the generated code is more complex than that of the sparse cases and will be explained in the following example. Consider the subroutine interface for the residual evaluator used in the DAEPACK DSL48 family of codes:

```
subroutine res(ny,t,y,ydot,delta,ires,ichvar,rpar,ipar).
```

The arguments of this routine are described in Table **??** in Chapter 1. Suppose that the only argument modified is `delta(ny)` which is a function of `t`, `y(ny)`, `ydot(ny)`, and `rpar(*)`. Adding the following additional options to the specification file,

```
SUFFIX      : _ad
PREFIX      : ad
AUX_PREFIX  : zzz
DEPENDENT   : delta
INDEPENDENT : y,ydot
```

results in code with the interface:

```
subroutine res_ad(ny,t,y,ady,ydot,adydot,delta,addelta,ires,
                  ichvar,rpar,ipar,adnp).
```

The additional arguments in the derivative code are `ady(adnp_max,ny)`, `adydot(adnp_max,ny)`, `addelta(adnp_max,ny)`, and `adnp`. A new file name `res_ad_adsize.f` will be created and included in the generated code. This file will contain the following:

```
integer adnp_max
parameter(adnp_max=10)
```

The value for `adnp_max` currently defaults to ten. This file may have to be edited if the code is called with `adnp` larger than ten. The default value can be overridden with the AD_SEED_NUMBER option. ***Make sure the derivative arguments are dimensioned with `adnp_max` in the calling program.*** Suppose the `res` subroutine computes $\delta = r(t, y, \dot{y}, p)$ (the outputs δ are returned in array `delta`). The new subroutine `res_ad` will compute:

$$\Delta' = \frac{\partial r}{\partial y} Y' + \frac{\partial r}{\partial \dot{y}} \dot{Y}',$$

where the **transpose** Δ' is returned in `addelta`, the **transpose** of Y' and $\dot{Y}'$ are passed in as `ady` and `adydot`, respectively, and the number of columns of Δ', Y', and $\dot{Y}'$ is equal to input argument `adnp`.

The flexible specification of independent and dependent variables described in Section 1.3 do not apply here. For example, if any component of `y` is declared as independent then the whole `ny`–dimensional array will be treated as independent, as indicated by the argument `ady(adnp_max,ny)`.

In general, the name of the new program unit will be the original root program unit name augmented with suffix given by the SUFFIX option (e.g., `res` becomes `res_ad`). The argument list will be modified as follows. All variables in the original argument list are present, in the original order, and their meanings are unchanged in the new argument list. For example, if `delta` returns the residual values in the original `res` subroutine then `delta` will return precisely the same quantities in `res_ad`. In the new argument list, each *active* argument is immediately followed by a new argument named by prefixing the original name with the prefix given by the PREFIX option (e.g., `adx` corresponds to `x`). This new argument will have one extra dimension than the original argument and the leading dimension will be <*AUX_PREFIX*>`np_max`, i.e., the name `np_max` will be prefixed by the string given by the AUX_PREFIX option). An *active* argument is any argument that is declared to be independent or dependent or depends on another active variable within the code. The final argument in the list is <*AUX_PREFIX*>`np`, the number of rows in the new derivative arguments, or mathematically, the number of columns in the "seed" matrix inputs and directional derivative outputs. Finally, a new Fortran source file will be created containing the declaration and initialization of <*AUX_PREFIX*>`np_max`. The name of this new file will be created by concatenating the new program unit name with `_adsize.f` (e.g., `res_ad_adsize.f`). This file, which will be included in the generated code, should be edited to make the dimensioning variable large enough for a given problem. The default value can be overridden with the AD_SEED_NUMBER option. When calling the generated code, all of the input variables in the new argument list that appear in the original argument list must be assigned values. The user can expect that all of the original output variables to return the expected values (i.e., the same values returned by the original program unit given the same input variable values). All derivative arguments (or direction vectors) corresponding to input variables must be initialized by the user prior to calling the new code. Upon successful execution of the new program unit, all derivative arguments corresponding to output variables will contain values for the desired directional derivative values. If an argument is both an input and an output variable then the corresponding derivative array must be initialized prior to calling the new code. This array will also return a value that may be of interest. To illustrate this, consider the following code where the argument is both the input independent variable and the output dependent variable:

```
subroutine res(t)
double precision t
t=t**2+3.0*t
return
end
```

The generated code is (comments removed):

```
subroutine res_ad(t,adt)
implicit none
double precision t
double precision adt
double precision zzzv1,zzzvbar1
double precision zzzv2,zzzvbar2
double precision zzzv3,zzzvbar3
double precision zzzv4,zzzvbar4
double precision zzzv5,zzzvbar5
double precision zzzv6,zzzvbar6
double precision zzzv7,zzzvbar7

zzzv3=t**2
zzzv5=3.0*t
zzzv6=zzzv3+zzzv5
zzzvbar1=2*t+3.0
t=zzzv6
adt=zzzvbar1*adt

return
end
```

Notice in this code that if same value for variable `t` is passed to both `res` and `res_ad`, the same values are returned in `t`. Also note that the return value for `adt` is the partial derivative of the expression with respect to `t` multiplied by the input value for `adt`.

2.3.6 Example 6: $J(x)x'$, Forward Mode

Using the following code generation options:

```
ACCUMULATION_MODE   : forward_mode
EXPLOIT_SPARSITY    : false
VECTOR_ACCUMULATION : false
```

will result in code that computes the Jacobian matrix post–multiplied by an arbitrary user–supplied vector x'. The computed quantity is a directional derivative in the x' direction. The code generated is a special case of the previous example where the number of directions, `adnp`, is fixed at unity. Using the same example, the interface of the new code is

```
subroutine res_ad(ny,t,y,ady,ydot,adydot,delta,addelta,ires,
                  ichvar,rpar,ipar),
```

where `ady(ny)` is the direction corresponding to argument `y`, `adydot(ny)` is the direction corresponding to argument `ydot`, and `addelta(ny)` is the computed directional derivative. Although this code is less general than the previous example, the code may be more efficient (depending on the compiler and BLAS routines available). Aside from this, the discussion in example 5 is applicable here.

2.3.7 Example 7: $\bar{X}^T + \bar{Y}^T J(x)$, Reverse Mode

Using the following code generation options:

```
ACCUMULATION_MODE   : reverse_mode
EXPLOIT_SPARSITY    : false
VECTOR_ACCUMULATION : true
```

will result in code that computes the Jacobian matrix pre–multiplied by an arbitrary user–supplied conformable matrix, $\bar{Y}$ added to an arbitrary user–supplied conformable matrix $\bar{X}$, that is,

$$\bar{X}^T + \bar{Y}^T J(x).$$

In contrast to example 4, the derivative matrix is computed by propagating dense vectors, rather than sparse vector data structures. Using the same subroutine in example 5 and adding the following options to the specification file:

```
SUFFIX      : _ad
PREFIX      : ad
AUX_PREFIX  : zzz
DEPENDENT   : delta
INDEPENDENT : y,ydot
```

results in code with the interface:

```
subroutine res_ad(ny,t,y,ady,ydot,adydot,delta,addelta,ires,
                  ichvar,rpar,ipar,adnp).
```

Similar to the corresponding forward mode case, the additional arguments in the derivative code are `ady(adnp_max,ny)`, `adydot(adnp_max,ny)`, `addelta(adnp_max,ny)`, and `adnp`. A new file name `res_ad_adsize.f` will be created and included in the generated code. As before, this file will contain the following:

```
integer adnp_max
parameter(adnp_max=10)
```

The value for `adnp_max` currently defaults to ten. This file may have to be edited if the code is called with `adnp` larger than ten. The default value can be overridden with the AD_SEED_NUMBER option. ***Make sure the derivative arguments are dimensioned with `adnp_max` in the calling program.*** Suppose the `res` subroutine computes $\delta = r(t, y, \dot{y}, p)$ (the outputs δ are returned in array `delta`). The new subroutine `res_ad` will compute:

$$\begin{bmatrix} \bar{Y}^T & \bar{\dot{Y}}^T \end{bmatrix} = \begin{bmatrix} \bar{Y}^T & \bar{\dot{Y}}^T \end{bmatrix} + \bar{\Delta}^T \begin{bmatrix} \frac{\partial r}{\partial y} & \frac{\partial r}{\partial \dot{y}} \end{bmatrix}$$

where the **transpose** of $\bar{Y}$ and $\bar{\dot{Y}}$ are passed in and returned in `ady` and `adydot`, respectively, and $\bar{\Delta}$ is passed in as `addelta`. The number of columns in $\bar{Y}$, $\bar{\dot{Y}}$, and $\bar{\Delta}$ is equal to input argument `adnp`.

The flexible specification of independent and dependent variables described in Section 1.3 do not apply here. For example, if any component of `y` is declared as independent then the whole `ny`–dimensional array will be treated as independent, as indicated by the argument `ady(adnp_max,ny)`.

The interface of the generated derivative code using the set of code generation options described here is the same as that in example 5. However, the modification will be described again. In general, the name

of the new program unit will be the original root program unit name augmented with suffix given by the SUFFIX option (e.g., `res` becomes `res_ad`). The argument list will be modified as follows. All variables in the original argument list are present, in the original order, and their meanings are unchanged in the new argument list. For example, if `delta` returns the residual values in the original `res` subroutine then `delta` will return precisely the same quantities in `res_ad`. In the new argument list, each *active* argument is immediately followed by a new argument named by prefixing the original name with the prefix given by the PREFIX option (e.g., `adx` corresponds to `x`). This new argument will have one extra dimension than the original argument and the leading dimension will be <*AUX_PREFIX*>`np_max`, i.e., the name `np_max` will be prefixed by the string given by the AUX_PREFIX option). An *active* argument is any argument that is declared to be independent or dependent or depends on another active variable within the code. The final argument in the list is <*AUX_PREFIX*>`np`, the number of rows in the new derivative arguments, or mathematically, the number of columns in the "weight" matrix inputs and adjoint matrix outputs. Finally, a new Fortran source file will be created containing the declaration and initialization of <*AUX_PREFIX*>`np_max`. The name of this new file will be created by concatenating the new program unit name with `_adsize.f` (e.g., `res_ad_adsize.f`). This file, which will be included in the generated code, should be edited to make the dimensioning variable large enough for a given problem. The default value can be overridden with the AD_SEED_NUMBER option. When calling the generated code, all of the input variables in the new argument list that appear in the original argument list must be assigned values. The user can expect that all of the original output variables to return the expected values (i.e., the same values returned by the original program unit given the same input variable values). All weight/adjoint arguments corresponding to input and output variables must be initialized by the user prior to calling the new code. Upon successful execution of the new program unit, all adjoint arguments corresponding to output variables will contain values for the desired derivative values. If an argument is both an input and an output variable then the corresponding derivative array must be initialized prior to calling the new code. This array will also return a value that may be of interest.

2.3.8 Example 8: $\bar{x}^T + \bar{y}^T J(x)$, Reverse Mode

Using the following code generation options:

```
ACCUMULATION_MODE   : reverse_mode
EXPLOIT_SPARSITY    : false
VECTOR_ACCUMULATION : false
```

will result in code that computes the Jacobian matrix pre–multiplied by an arbitrary user–supplied vector $\bar{y}$ and added to an arbitrary user–supplied vector $\bar{x}$. The code generated is a special case of the previous example where the number of weights, `adnp`, is fixed at unity. Using the same example, the interface of the new code is

```
subroutine res_ad(ny,t,y,ady,ydot,adydot,delta,addelta,ires,
                  ichvar,rpar,ipar),
```

where `ady(ny)` is the direction corresponding to argument `y`, `adydot(ny)` is the direction corresponding to argument `ydot`, and `addelta(ny)` is the computed directional derivative. Although this code is less general than the previous example, the code may be more efficient (depending on the compiler and BLAS routines available). Aside from this, the discussion in example 5 is applicable here.

2.4 Multiple Application of the Code Transformations

The examples above illustrate how to compute Jacobian matrices, matrix–Jacobian, and Jacobian–matrix products. In these cases, only first order derivative values are computed. As may be expected, code for computing higher order derivatives values may be generated by successive applications of DAEPACK. To illustrate this consider the function

$$y = f(x)$$

where $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$. The input to this code is the vector x and output is the vector y. A single application of the scalar forward mode results in code computing the following

$$\begin{aligned} y &= f(x) \\ y' &= f'(x)x' \end{aligned}$$

where $x \in \mathbb{R}^n$ and $x' \in \mathbb{R}^n$ are input and $y \in \mathbb{R}^m$ and $y' \in \mathbb{R}^m$ are outputs. A subsequent application of the scalar reverse mode on this code results in code for computing the following:

$$\begin{aligned} y &= f(x) \\ y' &= f'(x)x' \\ \bar{x}^T &= \bar{x}^T + \bar{y}^T f'(x) \\ \bar{x}'^T &= \bar{x}'^T + \bar{y}'^T f''(x)x'. \end{aligned}$$

In this case, $\bar{x}$ and $\bar{x}'$ are additional inputs and outputs (they are both input and output variables) and $\bar{y}$ and $\bar{y}'$ are additional inputs. Notice that components of the second derivative tensor may be computed by appropriate initializations of x' and $\bar{y}'$.

Care must be taken to avoid name conflicts multiple code generations are performed on the same code. **In particular, the AUX_PREFIX option must be set to different values for each application of DAEPACK.** Currently DAEPACK does not allow a subsequent code generation step to be applied to derivative code directly exploiting sparsity (i.e., code that propagates sparse vectors). This is due to the fact that this code uses library functions for which source will not be available. Sparsity may be exploited (e.g., options EXPLOIT_SPARSITY or SPARSESTORAGE equal to true) on the final code generation step. Finally, if dense vectors are propagated then calls to BLAS routines will be inserted into the generated code. Source code must be available for these routines (`daxpy` and `dcopy` only). The user must also provide source for utility routine `ad_zero`, which may simply be

```
subroutine ad_zero(n,v,inc)
integer n,inc
double precision v(*)
do i=1,n,inc
  v(i)=0.0d0
end do
return
```

Chapter 6 describes how to use DAEPACK to compute arbitrary order Taylor coefficients from Fortran code.

2.5 Complex-step Derivative Approximation Method (COMPLEX_STEP_DERIVS)

In addition to computing derivatives using the various modes of automatic differentiation, DAEPACK can also generate code which computes derivative values using the *complex-step derivative approximation method*. As the name suggest, this method computes approximations for the numerical values of the partial derivatives, in contrast to the other approaches which compute analytical derivative values accurate to computer round–off. However, in contrast to computing derivatives using the familiar finite differencing approach (differencing in the real domain), complex–step derivative approximations are free of cancellation error and are thus surprisingly accurate.

Suppose we have a real–valued function $f(x)$, $f : \mathbb{R} \longrightarrow \mathbb{R}$. Now consider the *complex–valued extension* of f:

$$\begin{aligned} w &= x + iy \\ f(w) &= u(w) + iv(w) \end{aligned}$$

If f is *analytic* (i.e., differentiable in the complex domain), the familiar Cauchy–Riemann equations hold:

$$\begin{aligned} \frac{\partial u}{\partial x} &= \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} &= -\frac{\partial v}{\partial x} \end{aligned}$$

We are interested in restricting ourself to the real domain, thus:

$$y = 0 \quad u = f(x) \quad \tfrac{\partial u}{\partial x} = \tfrac{\partial f}{\partial x}$$

We can approximate $\partial v/\partial y$ using finite–differences in the complex domain:

$$\frac{\partial v}{\partial y} \approx \frac{v(x + i(y + h)) - v(x + iy)}{h}$$

Using Cauchy–Riemann equations with fact $y = 0$ and $v(x) = 0$, we have:

$$\frac{\partial f}{\partial x} \approx \frac{\mathrm{Im}\,[f(x + ih)]}{h}$$

Expand f as a Taylor series around x:

$$f(x + ih) = f(x) + ihf'(x) - h^2 \frac{f''(x)}{2!} - ih^3 \frac{f'''(x)}{3!} + \cdots$$

$$f'(x) = \frac{\mathrm{Im}\,[f(x + ih)]}{h} + h^2 \frac{f'''(x)}{3!} + \cdots$$

The approximation is $O(h^2)$ and since no differencing is performed, the complex–step derivative approximation is **NOT** subject to cancellation errors.

The options avaiable for the complex–step derivative approximation approach are ROOT, SUFFIX, PREFIX, AUX_PREFIX OUTFILE, DO_NOT_PROCESS, INDEPENDENT, and DEPENDENT. All of these options are the same as those described for generating analytical derivatives with AD.

Chapter 3

Sparsity Patterns (SPARSITY)

3.1 Description

3.2 Sparsity Pattern Code Generation Options

Table 3.1: Sparsity pattern code generation options. Boldface keywords denote options that must be supplied by the user.

Specification file keyword	Default value
ROOT	None
SUFFIX	sp (e.g. "`res`" becomes "`ressp`")
AUX_PREFIX	zzz (e.g. intermediate variable "`zzzv1`")
OUTFILE	None
DO_NOT_PROCESS	None
INDEPENDENT	None
DEPENDENT	None

The first five options in Table 3.1 are common to all code transformation and have been described above. The remainder are described here.

3.2.1 INDEPENDENT

3.2.2 DEPENDENT

Chapter 4

Discontinuity–Locking (DISCONLOCK)

4.1 Description

4.2 Discontinuity–Locking Code Generation Options

Table 4.1: Discontinuity–locking code generation options. Boldface keywords denote options that must be supplied by the user.

Specification file keyword	Default value
ROOT	None
SUFFIX	dl (e.g. "`res`" becomes "`resdl`")
OUTFILE	None
DO_NOT_PROCESS	None

All of these options have been described previously.

Chapter 5

Interval Extensions (INTERVALEXT)

5.1 Description

5.2 Interval Extension Code Generation Options

Table 5.1: Interval extension code generation options. Boldface keywords denote options that must be supplied by the user.

Specification file keyword	Default value
ROOT	None
SUFFIX	in (e.g. "`res`" becomes "`resin`")
OUTFILE	None
DO_NOT_PROCESS	None
INTERVAL	None

The first four options in Table 5.1 are common to all code transformation and have been described above. The remainder are described here.

5.2.1 INTERVAL

Chapter 6

Multivariate Taylor Models (TAYLORMODEL)

6.1 Description

6.2 Taylor Model Code Generation Options

Table 6.1: Taylor model code generation options. Boldface keywords denote options that must be supplied by the user.

Specification file keyword	Default value
ROOT	None
SUFFIX	_tm (e.g. "`res`" becomes "`res_tm`")
PREFIX	tm_ (e.g. "`x`" becomes "`tm_x`")
AUX_PREFIX	zzz (e.g. intermediate variable "`zzzv1`")
OUTFILE	None
DO_NOT_PROCESS	None
INDEPENDENT	None
DEPENDENT	None

The first six options in Table 6.1 are common to all code transformation and have been described above. The remainder are described here.

6.2.1 INDEPENDENT

6.2.2 DEPENDENT

Chapter 7

Convex Relaxations (CONVEXIFY)

7.1 Description

Chapter 8

Call Graphs (CALLGRAPH)

8.1 Description

DAEPACK provides the option to generate the call graph of an evaluation program in the format used by the graph program `dot`[1]. The call graph is a directed acyclic graph where the vertices denote program units such as subroutines or functions and the edges point from a program unit to all other program units it calls. The ability to visualize the call graph is useful when analyzing code inherited from others. Also, the call graph generated by DAEPACK will indicate whether or not certain program units are missing, which will cause problems when generating other types of codes with DAEPACK.

8.2 Call Graph Generation Options

Table 8.1 contains a summary of the options available for creating call graphs. At this time only ROOT, OUTFILE, and DO_NOT_PROCESS are supported. Future implementations will support the ability to trace the path of certain program variables through the code.

Table 8.1: Call graph generation options. Boldface keywords denote options that must be supplied by the user.

Specification file keyword	Default value
ROOT	None
OUTFILE	None
DO_NOT_PROCESS	None

All of the options in Table 8.1 have been described in Sections 1.2.1, 1.2.2, and 1.2.6. As may be expected, ROOT specifies the root vertex of the graph, OUTFILE specifies the name of the file containing the graph description in `dot` format, and `DO_NOT_PROCESS` allows the user to select program units that are not to be analyzed. Disabling the analysis of certain program units is useful for simplifying certain sections of complex graphs.

[1]For a description of `dot` see http://www.research.att.com/sw/tools/graphvis/dotguide.pdf.

To illustrate the call graph generation capability, consider the following highly contrived Fortran source code example.

```
subroutine res(n,x,y,func0,sub0)
dimension x(n),y(n)
double precision func0
external func0,sub0
call func0(n,x,y)
call sub0(n,x,y)
call sub1(n,x,y,func0)
call sub2(n,x,y,func0)
.
.
.
return
end

subroutine sub1(n,x,y,func0)
double precision func0
external func0
dimension x(n),y(n)
call sub2(n,x,y,func0)
call sub3(n,x,y)
.
.
.
return
end

subroutine sub2(n,x,y,func0)
double precision func0
external func0
dimension x(n),y(n)
call sub3(n,x,y)
y(1)=func0(n,x,y)
.
.
.
return
end

subroutine sub3(n,x,y)
dimension x(n),y(n)
.
.
.
return
end
```

Notice that this example contains the source of four subroutines and a function and a subroutine are passed in as arguments to subroutine `res`. Assume there is no source available for the program units appearing as arguments and that there are no other program units called in the code replaced by ellipses. Applying DAEPACK to this source using `res` as the root program unit results in the following `dot`–format graph file:

```
Digraph "res-call-graph" {
ratio = fill
1 [label="res", shape=ellipse];
2 [label="func0", shape=box, style = dotted];
3 [label="sub0", shape=ellipse, style = dotted];
4 [label="sub1", shape=ellipse];
5 [label="sub2", shape=ellipse];
6 [label="sub3", shape=ellipse];
1 -> { 2; 3; 4; 5 };
4 -> { 5; 6 };
5 -> { 6; 2 };
}
```

A description of `dot` is outside the scope of this paper, however, the reader can probably guess that this file describes a directed acyclic graph (digraph) and that the vertices are described at the top this file and the graph topology is described at the bottom. Using the `dot` program, the call graph shown in Figure 8.1 is produced. Notice the conventions used by DAEPACK. ***Subroutine program units are***

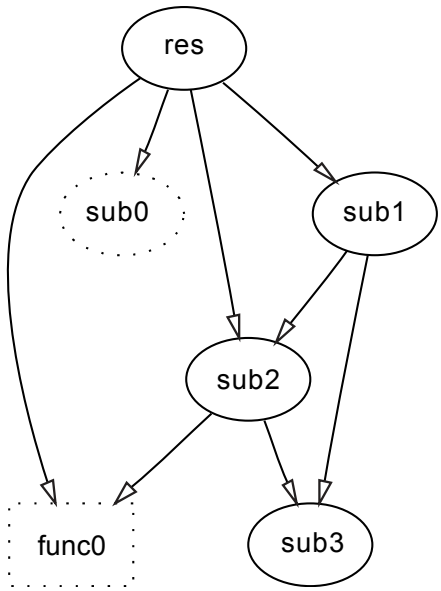

Figure 8.1: Call graph example generated with DAEPACK and `dot`.

denoted by ellipses and functions by boxes. If source is available then the shape border is

solid, otherwise it is dotted. If the source is available but the program unit appears in the do not process list then the shape border is bold. These conventions are summarized in Tables 8.2 and 8.3. Figure 8.2 shows the call graph of the same evaluation program except that `sub2` is placed in

Table 8.2: Shapes used for vertices in generated call graph.

Shape	Meaning
Ellipse	Subroutine (program unit without a return value)
Box	Function (program unit with some return value)
Diamond	Alternate entry point
Circle	Unknown (probably an undimensioned array)

Table 8.3: Borders used for vertices in generated call graph.

Border	Meaning
Thin solid	Source is available for this program unit
Dotted	Source is **not** available for this program unit
Thick solid	Source is available, but program unit appears in a DO_NOT_PROCESS list

the DO_NOT_PROCESS list.

Currently, DAEPACK will not construct call graphs below entry points. This limitation will be addressed in future implementations.

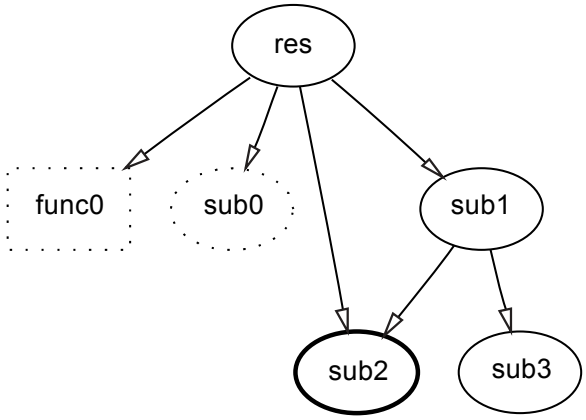

Figure 8.2: Call graph example generated with DAEPACK and `dot`. The call graph generated in this figure has `sub2` appearing in a do not process list.

Chapter 9

Conclusions