Chapter 6

Implementation and Software

The techniques explained in Chapters 3 and 5 reveal correct sequence of operations that the transformed program must perform in order to calculate the correct derivative values. The insight that the AD framework gives us is valuable even if we propose to carry out the program transformation by modifying the source code manually.

However, manual transformation is time consuming and prone to error. It is also hard to manage; if the underlying program is subject to continual change, then keeping track of these changes, in order to maintain the integrity of the transformed version, is also time consuming. Therefore, it is usually desirable to automate at least partially the process of transformation. Regardless of how automatically, i.e., with what level of automation, program transformation is done there is a trade-off between the sophistication of the transformation process and the efficiency with respect to time and space bounds of the transformed program. As a rule, a general-purpose AD tool will not produce transformed code as efficient as that produced by a special-purpose translator designed to work only with underlying code of a particular structure, since the latter can make assumptions often with far-reaching consequences, where as the former can only guess.

In many cases, an unsophisticated approach suffices to produce AD code that is within a constant factor of the optimal performance bounds. In this situation, subsequent effort is devoted just to reducing the value of the constant. In other cases, a careful analysis of the structure of the code will reveal that several orders of magnitude can be gained by the use of more sophisticated techniques, such as preaccumulation (see section 10.2), or by the careful exploitation of by-products of the underlying computation, such as LU decompositions, in the derivative process (see Exercise 3.4). Such transformation processes may involve a substantial amount of human intervention, including some modification of the code, or a much greater degree of sophistication in the design of the automated tools.

For example, our primary motivation might be the wish to explore new models or algorithms to see whether they merit further investigation. In this case the

principal benefit of AD is the ability to provide accurate derivative information for newly coded numerical functions while avoiding the labor traditionally associated with developing accurate derivative evaluation code. Hence the emphasis will be on the ease of the transformation process, rather than on the efficiency of the transformed program.

Later, when our preferred model is stable, we may be willing to spend more effort designing our code to produce derivative code that runs in the shortest possible time. Later still, if our approach is a success and we wish to apply it to larger and larger problems, we may be willing to spend more effort in developing a special-purpose AD tool designed to work only with this model and in encapsulating our insights about the fundamental structure of the corresponding programs.

The two basic computer science concepts employed in one guise or another by AD implementers are operator overloading and source transformation performed by compiler generators sometimes also called compiler-compilers. Although operator overloading and source transformation may look very different to the user, the end product is in both cases an object code, which we have called `eval_der.obj` in Fig. 6.1.

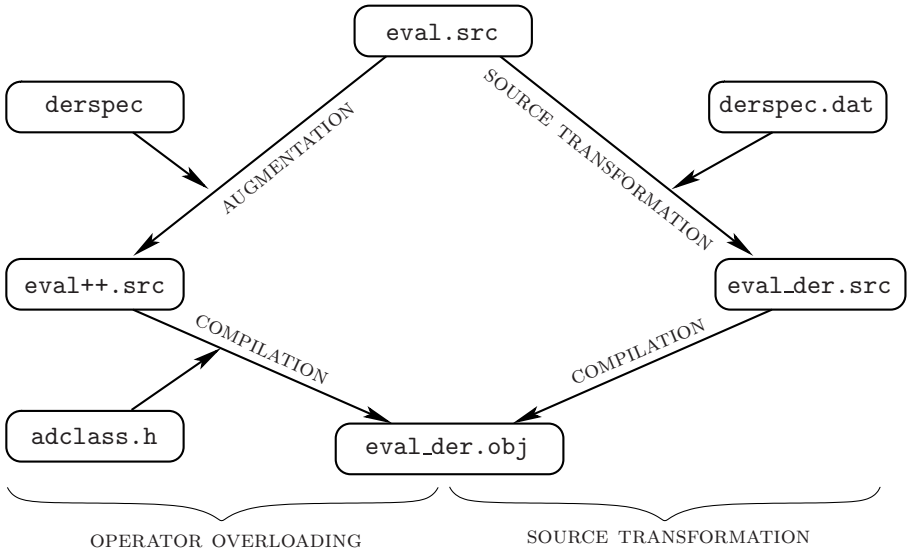

Figure 6.1: From Function Sources to Derived Object Files

There we start with a source code `eval.src` that is either transformed into another source `eval_der.src` by a preprocessor or augmented to a program `eval++.src` by the user before being compiled. Either modification requires the selection of independent and dependent variables as well as the kind of derivative objects and differentiation modes the programmer wishes to specify. We have represented this information by the boxes `derspec` and `derspec.dat`, but it can of course be imported in many different ways.

Subsequently either the source `eval_der.src` or the source `eval++.src` must be compiled, together with header files defining the new type of variable and their operations included in the second case. In either scenario the resulting object file `eval_der.obj` may then be linked with a library of AD routines to yield (possibly only a part of) an executable that evaluates derivative values $\texttt{dy} = \dot{y}$, $\texttt{bx} = \bar{x}$, and $\texttt{dbx} = \dot{\bar{x}}$ for given points x, directions $\texttt{dx} = \dot{x}$, adjoints $\texttt{by} = \bar{y}$, etc, as depicted in Fig. 6.2. The execution may generate a considerable amount of temporary data, which may be kept in internal arrays or flow out to scratch files.

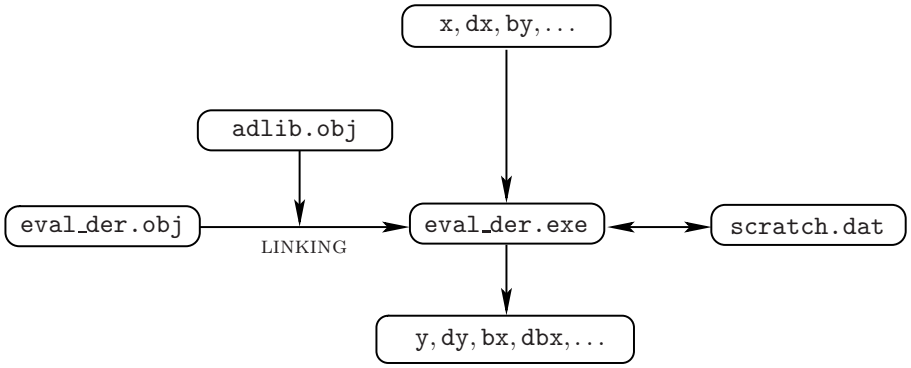

Figure 6.2: Linking and Executing Derived Object Files

There is no better way to understand AD than to implement one's own "baby" AD tool. But it is also worth pointing out that for many small- to medium-size problems, particularly those where derivative calculation does not account for a high proportion of the runtime requirements, a simple AD tool is often perfectly adequate and has the added merit that it contains no unnecessary features or bugs that the programmer did not personally put there.

Active and To-Be-Recorded Variables

The most basic task for the differentiation of a given evaluation code is the identification of all variables that need to be *active* in the following sense. In the computational graph discussed in section 2.2 all vertices lying on a path between the minimal (independent) nodes and the maximal (dependent) nodes are *active* because their values enter into the functional relation between at least one such pair of values. Moreover, unless an active vertex enters only as argument into linear operations, its value will also impact some partial derivative, namely, entry of the overall Jacobian. As discussed in section 4.3, it will then have to be saved during the forward sweep or recomputed during the return sweep. Strictly speaking, the attribute *active* or *to-be-recorded* should be assigned to each value computed during the evaluation of a code. In practice this decision is usually taken for a computer variable, namely, a symbol within a certain scope,

or at least for each one of its occurrences as a left-hand side of an assignment. This simplification usually means that some values are treated as active, even though that may not be necessary. Erring on the side of caution in this way forgoes some potential gain of efficiency but entails no loss of correctness and only rarely affects numerical accuracy. Naturally, variables or values that are not selected as *active* are called *passive* throughout this book.

In the overloading approaches described in section 6.1 activity must be decided for each program variable, which is then retyped accordingly. A simple way to avoid this tedious task is to redefine in the evaluation code all `double`s to the active type `adouble`s. Here we assume tacitly that all active calculations are performed in double precision. This brute force modification is used, for example, in the Network Enabled Optimization Server (NEOS), which employs ADOL-C for the differentiation of user codes written in C. The downside is that many passive (i.e., nonactive) variables may be appended with derivative fields whose value remains zero but that are nevertheless involved in many (trivial) calculations. This effect can be partly overcome by using an activity analysis at runtime as used, for example, by the newer versions of the C/C++ tools ADOL-C and CppAD.

With a language translation approach, a great deal more can be done to automate the dependence analysis required to determine which variables have derivatives associated with them. Of course, where it is impossible to determine at compile-time whether two variable values depend on each other, for example, because array indices or pointers are manipulated in a complex way at runtime, the translator must make a conservative assumption or rely on user-inserted directives.

Overloading is a rather flexible technique and can usually deliver within a constant factor of the theoretical performance bounds derived in Chapter 5. However, the program transformation description that we give in section 6.2 should also provide some understanding of what is going on inside sophisticated AD tools, so that programmers can choose an appropriate existing tool made available by other people and configure it correctly for a particular problem, or even develop application-specific tools themselves. Ultimately, we hope that many others will be able to exploit the structure of their applications so as to design new AD algorithms.

6.1 Operator Overloading

Overloading is supported by modern computer languages such as C++, Ada, and Fortran 90. Certain considerations regarding the use of operator overloading influence the choice of which language is appropriate. The main issues regarding language capabilities for the purpose of AD by overloading are

- whether the assignment operator can be overloaded,
- what level of control the user is permitted to exercise over the management of dynamically allocated memory,

- whether user-written constructor functions can be automatically called to initialize user-defined data types, and
- whether user-written destructor functions can be automatically called to do housekeeping when variables go out of scope and are deallocated.

By now, a large number of AD tools exist based on overloading in ADA (see, e.g., [BBC94]), Fortran 90 (see, e.g., [DPS89] and [Rho97]), C++ (see, e.g., [Mic91], [GJU96], [BS96]), [Bel07]), and Matlab (see, e.g., [RH92] and [CV96]). There are also some C++ implementations (see, e.g., [ACP99]) using expression templates [Vel95] to generate derivative code that is optimized at least at the statement level. In terms of runtime efficiency the selection of a suitable compiler and the appropriate setting of flags may be as important as the choice of language itself. Naturally, we cannot make any recommendation of lasting validity in this regard as software environments and computing platforms are subject to continual change.

The examples given below are coded in a subset of C++ but are carefully designed to be easy to reimplement in other languages that support operator overloading. Reading this chapter does not require a detailed knowledge of C++, since we deliberately do not exploit the full features of C++. In particular, the sample code that we give is intended to illustrate the corresponding algorithm, rather than to be efficient.

Simple Forward Implementation

In this section we describe a simple approach to implementing the basic forward mode described in section 3.1. The implementation presented here is based on the tapeless forward-mode of the AD tool ADOL-C. It propagates a single directional derivative for any number of independent or dependent variables.

Implementing a Forward-Mode Tool

To build a simple forward-mode package, we need to carry out the following tasks:

- Define the new data type or class that contains the numerical values of v_i and $\dot{v}_i$. Here, we will use the new class

```
class  adouble
{  double  val;
   double  dot;      }
```

- Define arithmetic operations on the `adouble` class corresponding to the usual floating point operations on scalars. These overloaded operations must manipulate the second part of the `adouble` corresponding to $\dot{v}_i$ correctly, according to the chain rule; see, for instance, Table 3.3. Some languages allow the programmer to define several different procedures with the same name, but operating on or returning arguments of different types.

For example, we may define a function `sin` that operates on a `adouble` and returns another `adouble`.

```
adouble sin (adouble a)
{  adouble b;
   b.val = sin(a.val);
   b.dot = cos(a.val) * a.dot;
   return b;       }
```

For certain programming languages operator overloading allows the programmer the same freedom with symbols corresponding to the built-in operations. For example, we might define the following multiplication of `adouble`s.

```
adouble operator* (adouble a, adouble b)
{  adouble c;
   c.val = a.val * b.val;
   c.dot = a.dot * b.val + a.val * b.dot;
   return c;       }
```

In the case of binary operations, we also need "mixed-mode" versions for combining `adouble`s with constants or variables of type `double`.

To reduce the likelihood of errors one may define the derivative field `dot` as "private" so that the user can access it only through special member functions.

- Define some mechanism for initializing `adouble` components to the correct values at the beginning, for example by providing the member functions `setvalue()` and `setdotvalue()`. These functions should also be used to initialized the derivative values of the independent variables. For extracting variable and derivative values of the dependent variables, member functions like `getvalue()` and `getdotvalue()` should be available. Uninitialized derivative fields may default to zero.

In some applications one may wish to deliberately assign a `double` to an `adouble`. Then one needs corresponding facilities.

- Conversions from `double`s to `adouble`s may occur implicitly with the `dot` value of the `adouble` set to zero. For example, a local variable `a` may be initialized to zero by an assignment `a=0` and later used to accumulate an inner product involving active variables. In C++ [Str86] the assignment operator may invoke a suitable constructor, that is, a user-supplied subroutine that constructs a `adouble` from given data. Here we may define in particular the following constructor.

```
adouble::adouble (double value)
{  val = value;
   dot = 0;     }
```

This constructor sets the derivative field of a `adouble` to zero whenever it is assigned the value of a passive variable. Another approach is to define assignments of the form `x = b` by overloading the assignment operator.

```
adouble& adouble::operator= (double b)
{  val = b;
   return *this;    }
```

Note that this is the only point at which we have proposed to overload the assignment operator. Some languages do not allow the assignment operator to be overloaded. Then such implicit conversions cannot be performed, and one has to ensure full type consistency in assignments.

Using the Forward-Mode Tool

To apply our forward-mode AD implementation to a particular numerical program, we need to make a number of modifications to the code that evaluates the function. Minimizing these modifications is one of the design goals of AD implementation, since rewriting large pieces of legacy code is not only tedious but also error prone. The changes that cannot be avoided are the following.

Changing Type of Active Variables

Any floating-point program variable whose derivatives are needed must be active and thus redeclared to be of type `adouble` rather than of type `double`.

Initializing Independents and their Derivatives

We also need a way of initializing independent variables. In the C++ package ADOL-C [GJU96] the binary shift operator was overloaded such that the statement `x<<=b` has the effect of specifying `x` an independent variable and initializing its value to `b`. There exist several alternatives to this approach.

At the point where independent variables are assigned numerical values, the corresponding derivative values should also be initialized, unless they may default to zero.

Deinitializing Derivatives Dependents

At the point where the dependent variable values are extracted, the corresponding derivative values can also be extracted.

Nothing else in the user-defined function evaluation code needs to change. In particular, all the rules for derivative propagation are concealed in the AD package nominated in a suitable header file.

A Small Example of Simple Forward

As a small example for the simple forward-mode implementation presented here, we consider the scalar-valued function

$$f : \mathbb{R}^2 \to \mathbb{R}, \qquad f(x) = \sum_{i=1}^{n} \left[\left(x_1 - \alpha_i\right)^2 + \left(x_2 - \beta_i\right)^2 \right] . \tag{6.1}$$

Here we assume that the real scalars $x = \left(x_1, x_2\right)^\top$ are the independent variables and the parameters α_i and β_i are constants. One may be interested in the directional derivative of f in direction $\dot{x} = \left(\dot{x}_1, \dot{x}_2\right)^\top$. For this purpose, first we present the code to evaluate $f(x)$. Note that `cin` and `cout` are the standard input/output stream operations used in C++ to read and write variable values.

```
double alphai, betai;
double x1, x2, y;
cin >> x1, cin >> x2;
y = 0.0;
for (int i=0; i<n; i++)
{ cin >> alphai; cin >> betai;
  y = y + (x1-alphai)*(x1-alphai) + (x2-betai)*(x2-betai);}
cout << y;
```

The augmented code to calculate the directional derivative is given below, where the vertical bar `|` denotes a modified or inserted line.

```
 1   double alphai, betai;
 2 | adouble x1, x2, y;
 3 | double dx1, dx2;
 4   cin >> x1;  cin >> x2;
 5 | cin >> dx1; cin >> dx2;
 6 | x1.setdotvalue(dx1);
 7 | x2.setdotvalue(dx2);
 8   y = 0.0;
 9   for (int i=0; i<n; i++)
10    { cin >> alphai;  cin >> betai;
11      y = y + (x1-alphai)*(x1-alphai) + (x2-betai)*(x2-betai);}
12   cout << y;cout << y.dot;
```

The changes introduced into the augmented program correspond to the points enumerated at the beginning of this section, and the input/output stream operations have been overloaded to read and write `adoubles`, so that `cin` will assign input values to the `adoubles` `x` and `y`, together with a `dot` value of zero.

The lines 4 to 7 of the augmented code correspond to the initialization in the first loop of Table 3.4. The function evaluation in the second loop of Table 3.4 is performed in the lines 8 to 11. Line 12 represent the third loop of Table 3.4, namely, the extraction of the dependent variables.

Note, in particular, that the "main body" of the code, which actually calculates the function value, is unchanged. Of course, in this tiny example the main body is a relatively small proportion of the code, but in a more realistic application it might constitute 99% of the code, including nested loops and recursive procedure calls with parameter passing, as well as other programming structures. Note that our example already includes overwriting of `y`. Apart from redeclaring `double`s as `adouble`s, no changes need be made to any of this code. However, a judicious choice of all variables that must be redeclared because they are active as defined at the beginning of this chapter, may not be easy. However, performing this selection carefully is important for efficiency to avoid redundant calculations.

Vector Mode and Higher Derivatives

To evaluate a full Jacobian with a single pass of the forward-mode, we can redefine the `adouble` type so that the `dot` field is an array of tangent components

```
class v_adouble            with     class vector
{  double val;                      {
   vector dot; };                      double comp[p]; };
```

where `p` is the number of directional components sought. For efficiency it is preferable that `p` is a compile-time constant, but for flexibility it may also be defined as a static member of the class `vector`.

We also define a number of overloaded operations on the type `vector`, for example,

```
vector operator* (double a, vector b)
{  vector c;
   for (int i=0; i<p; i++)
      c.comp[i] = a * b.comp[i];
   return c;                     }
```

Once we have done so, the same code used to define the overloaded operations on scalars will also work correctly on vectors, allowing us to evaluate an entire Jacobian or, more generally, an arbitrary family of tangents, in a single forward sweep.

We can also extend the definition of a `adouble` to encompass higher-order derivatives. For example, we can define a type

```
class s_adouble
{  double val;
   double dot;
   double dotdot;  };
```

with appropriate overloaded operations acting on the various components to compute second-order derivative, or a more general type `Taylor`.

Simple Reverse Implementation

In this section we describe a simple approach to implementing the basic reverse method introduced in section 3.2. This implementation evaluates a complete adjoint vector. This adjoint vector can be a normal vector corresponding to the gradient of one of the dependent variables, or it may be an arbitrary linear combination of such normal vectors, for example, a set of constraint normals scaled by Lagrange multipliers such as $\bar{x}^\top = \sum_i \bar{y}_i \nabla_x F_i$.

During the execution of the evaluation procedure we build up an internal representation of the computation, which we will call here *trace*. The trace is in effect a three-address code, consisting of an array of operation codes and a sequence of variable indices, both encoded as integers. In addition to these two symbolic data structures there is a third numerical record for storing the floating-point numbers that represent the pre-values discussed in section 4.2. Both the symbolic information and the numerical data may be reused in various ways, so their distinct roles should be understood. Generally, the symbolic trace is specific to overloading tools, whereas the record of pre-values occurs also in source transformation based AD tools. The return sweep will be carried out by a simultaneous interpretation of the three records, which are illustrated in Fig. 6.3.

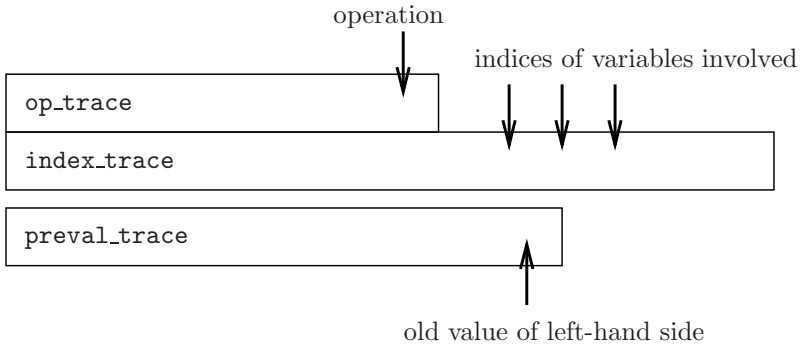

Figure 6.3: Structure of the Traces

Implementing a Reverse Mode Tool

To build a simple reverse-mode AD package, we need to carry out six tasks.

- Define a new data type, here again called `adouble`, containing the numerical value of v_i and an identifier or index.

  ```
  class adouble
  {  double val;
     int index; };
  ```

  Additionally we need the trace consisting of the three arrays `op_trace`, `index_trace` and `preval_trace` to store information about the performed

operations, the involved variables and the overwritten variables, respectively. Furthermore, a global variable `indexcount` is required to perform bookkeeping about the already used indices. This set up is illustrated in Fig. 6.3

- Corresponding to the usual floating-point operations on scalars, define arithmetic operations of the `adouble` type. These overloaded operations calculate the floating-point value for v_i as usual, but as a side-effect they also record themselves and their arguments in three arrays `op_trace`, `index_trace`, and `preval_trace`. For this purpose we also need to define an integer `opcode` for each of the elemental operations, for example,

```
const int
emptyv = 0, constv = 1, indepv = 2, bplusv = 3,
bminusv = 4, bmultv = 5, recipv = 6,  ...
expv = 11, lnv = 12, sinv = 13, cosv = 14, ...
```

Then we may define typical operations as follows.

```
adouble sin (adouble a)
{  adouble b;
   indexcount += 1;
   b.index = indexcount;
   put_op(sinv, op_trace);
   put_ind(a.index, b.index, index_trace);
   b.val = sin(a.val);
   return b;              }

adouble operator* (adouble a, adouble b)
{  adouble c;
   indexcount += 1;
   c.index = indexcount;
   put_op(bmultv, op_trace);
   put_ind(a.index, b.index, c.index, index_trace);
   c.val = a.val*b.val;
   return c;              }
```

In the case of binary operations such as `*` we again need to provide "mixed-mode" versions for combining `adouble`s with constants or variables of type `double`. Assignments between `adouble`s and their initializations from `double`s must also be recorded as operations. For example, the overloaded assignment operator to store the information required for the return sweep may be defined by

```
adouble&adouble::operator =  (adouble b)
 {  put_val(val, preval_trace);
    put_ind(b.index, index_trace);
```

```
    val = b.val;
    index = b.index;        }
```

- We also need a way of initializing independent variables (and setting the corresponding adjoint values initially to zero). In the C++ package ADOL-C [GJU96] the binary shift operator was overloaded such that the statement `x<<=b` has the effect of specifying `x` an independent variable and initializing its value to `b`. There exist several alternatives to this approach.

- Define a `return_sweep` routine that reverses through the trace and calculates the adjoint variables $\bar{v}_i$ correctly, according to the adjoint evaluation procedure with tape and overwrites given in Table 4.4. The routine `return_sweep()` consists basically of a `case` statement inside a loop. The actual derivative calculation is performed by using the two additional arrays `value` containing intermediate results and `bar` containing the derivative information.

```
void return_sweep()
 { while (NOT_END_OF_OPTAPE)
    { opcode = get_op(op_trace);
      switch(opcode)
      { ...
        case assignv:
          res = get_ind(index_trace);
          value[res] = get_val(preval_trace);
        break;
        ...
        case sinv:
          res = get_ind(index_trace);
          arg = get_ind (index_trace);
          bar[arg] += bar[res]*cos(value[arg]);
        break;
        ...
        case bmultv:
          res  = get_ind(index_trace);
          arg1 = get_ind(index_trace);
          arg2 = get_ind(index_trace);
          bar[arg1] += bar[res]*value[arg2];
          bar[arg2] += bar[res]*value[arg1];
        break;
        ...                                  } } }
```

- Define some mechanism for extracting the current floating-point value from an `adouble`, for initializing adjoint components to the correct values

at the beginning of the return sweep and for extracting gradient component values when these become available at the end, in other words, define routines `value`, `setbarvalue`, and `getbarvalue`.

- Define a routine `resettrace()` to indicate that the adjoint calculation is finished such that the trace and the additional arrays can be reused for a new derivative evaluation.

Using the Reverse-Mode Tool

To apply our reverse-mode AD implementation to a particular numerical program, we need to make a number of modifications to the users code. More specifically we must modify the evaluation routine that defines the function to be differentiated and also the calling subprograms that invoke it to obtain function and derivative values.

Changing Type of Active Variables

Just like in the simple forward implementation, all floating-point program variables that are active must be redeclared to be of type `adouble` rather than of type `double`.

Initializing Adjoints of Dependents

At some point after the dependent variables receive their final values, the corresponding adjoint values must also be initialized as in Table 3.5.

Invoking Reverse Sweep and Freeing the Traces

Before the gradient values can be extracted, the `return_sweep` routine must be invoked. This invocation can take place from within the gradient value extraction routine. We need to indicate when the adjoint calculation is finished so that we can reuse the storage for a subsequent derivative evaluation. Otherwise, even a function with a small computational graph will exhaust the available storage if it is evaluated many times.

Deinitialyzing Adjoints of Independents

When the adjoint calculation is completed, the corresponding derivative values can be extracted.

Remarks on Efficiency

While the reverse implementation sketched here incurs some overhead costs; it does bring out some salient features. Not only is the total temporal complexity a small multiple of the temporal complexity of the underlying evaluation code, but we observe that the potentially very large data structures represented by the three trace arrays are accessed strictly sequentially.

A Small Example of Simple Reverse

As a small example for the simple reverse-mode implementation discussed above, we consider again the scalar-valued function (6.1) and corresponding evaluation code given on page 118. The augmented code to calculate the gradient is follows.

```
 1   double alphai, betai;
 2 | adouble x1, x2, y;
 3 | double by, bx1, bx2;
 4   cin >> x1;  cin >> x2;
 5   y = 0.0;
 6   for (int i=0; i<n; i++)
 7   { cin >> alphai;  cin >> betai;
 8     y = y + (x1-alphai)*(x1-alphai) + (x2-betai)*(x2-betai);}
 9 | cin >> by;
10 | setbarvalue(y, by);
11 | return_sweep();
12 | getbarvalue(x1,bx1);
13 | getbarvalue(x2,bx2);
14 | resettrace();
```

Lines 5 to 8 correspond to the recording sweep of Table 3.7. The initialization of the normal is done in lines 9 and 10. The second for-loop of the return sweep in Table 3.7 is invoked in line 11. Lines 12 and 13 represent the final for-loop of Table 3.7, namely, the extraction of the adjoint values.

6.2 Source Transformation

An AD tool based on operator overloading consists essentially of a library written in the same language as the program to be differentiated. Its size can be reasonably small. In contrast, source transformation is a more laborious approach to implementing an AD tool, which is similar in its complexity to developing a compiler. The source transformation description that we provide in this chapter should facilitate some understanding of what is going on inside sophisticated AD tools based on source transformation.

In the earliest days of AD (see, e.g., [Con78] and [KK$^+$86]), users were expected to completely rewrite their code, usually replacing all arithmetic operations with function calls. Later software designers tried to live up to the expectation generated by the label "automatic differentiation" by using preprocessors (e.g., Augment used by Kedem [Ked80]) or overloading (e.g., in Pascal-SC by Rall [Ral84]). Probably the first powerful general-purpose system was GRESS, developed at Oak Ridge National Laboratory [Obl83] in the 1980s and later endowed with the adjoint variant ADGEN [WO$^+$87]. After the first international workshop on AD in 1991 in Breckenridge, three large Fortran 77 systems were developed: ADIFOR [BC$^+$92] at Argonne National Laboratory

and RICE University; Odyssée [R-S93] at INRIA, Sophia Antipolis; and TAMC as a one-man effort by Ralf Giering of the Meteorological Institute of Hamburg [GK98]. Currently, the tool Tapenade [HP04] is developed and maintained at INRIA, Sophia Antipolis, and TAF [GK98] as successor of TAMC by the company FastOpt. All these systems accept Fortran 77 codes that contain some constructs of Fortran 90 and generate derivative code in the same language. A similar one-man effort is PADRE2 [Kub96], which generates some scratch files in addition to the derived source code; this is also true of GRESS/ADGEN. While PADRE2 also calculates second-order adjoints, the other systems generate "only" first-derivative codes. In principle, however the source transformation may be applied repeatedly, yielding second and higher derivative codes if everything goes well. Naturally, the symmetry of Hessians and higher derivative tensors cannot be exploited by such repeated source transformations. Taylor coefficients and derivative tensors of arbitrary order have been a key aspect of the systems DAFOR [Ber90b] and COSY INFINITY [Ber95] by Martin Berz. The system PCOMP [DLS95], developed by Klaus Schittkowski and his coworkers, generates efficient derivative code from function specification in a restricted Fortran-like language.

AD Preprocessors

AD preprocessors belong to a large family of static tools: they perform program analysis or source transformation at compile time, without requiring input data or running the program. Static tools may have various objectives, among which one can find, for example, the compilation of source files into binary code, the detection of runtime errors, and the instrumentation of source code. These tools take as input a program written in a programming language (C, C++, Ada, Fortran, Perl, and etc.), construct an internal representation of the program, perform some analysis, and possibly transform the internal representation using the result of the analysis: AD preprocessors are enhancers, that is, they generate a new program that computes supplementary values, namely derivatives of the values of interest with respect to input values. A key distinction from other enhancers is that while efficiency of the generated program is of low priority in most tools, for instance, when debugging a program or finding runtime errors, it is highly important for AD. The fact that the generated code is meant to be used in operational phase leads to specific concerns about the cost of the derivatives compared to the original function.

Enhancer tools are built from four components, as shown in Fig. 6.4. The parsing, printing, and optimization components are not specific to AD. Therefore they are not described here in detail. For the transformation task, AD preprocessors use general purpose data structures to internally represent the program.

Among them one can find the following fundamental ones:

- **The call graph** represents the "call to" relation between procedures of the program as a graph the nodes of which are the procedure names and

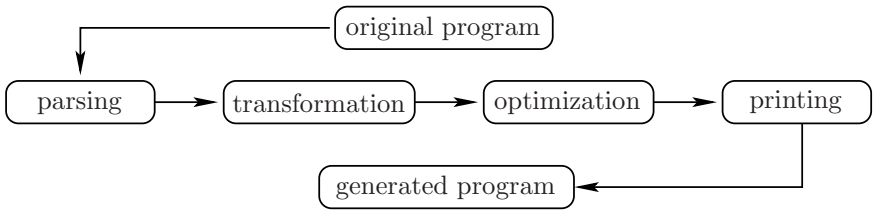

Figure 6.4: General architecture of a source preprocessor

one arc relates procedures P and Q if and only if P calls Q,

- **The abstract syntax tree** represents the procedures declarations and statements of each procedure in the program in a syntactical way.

- **The control flow graph** represents the "successor" relation between statements of a procedure as a graph the nodes of which are the statements indices. One arc relates statements s_i and s_j if and only if statement s_j must be executed directly after statement s_i.

- **The data flow graph** represents the "depends on" relation between variables of a procedure/program as a graph the nodes of which are the variables and an arc relates variables v_j and v_i if and only if the value of variable v_i is obtained directly from value of variable v_j.

The differentiation of a program is a complex process that involving the enhancement of the original program: one or more derivative statements are added in forward or reverse mode for each original active statement. This objective can be reached in two ways. One can apply differentiation to generate a completely differentiated code, then apply slicing [Tip95] with respect to dependent and independent variables to discard useless statements. This approach is called context-sesitive differentiation. Alternativly, one can apply context-free differentiation. The context-sesitive and context-free differentiation approaches differ in that in context-free differentiation only active original statements with respect to dependent and independent variables are differentiated, whereas in context-sesitive differentiation all (real) original statements are differentiated. Moreover, in context-free differentiation only variables whose values are necessary for the evaluation of derivative values are recorded and retrieved, whereas in context-sesitive differentiation the value of all modified variables is recorded or retrieved.

For clarity of the following description of an AD preprocessor, some hypotheses on the input code are imposed:

- Access paths to memory location are limited to scalar variable accesses (i.e., component of array, fields of structures, and the like are not considered here).

- Subprograms are limited to procedures (functions, function pointers, virtual functions, arrays of functions and the like are not considered here).
- Input/output statements are considered as passive statements and are therefore not differentiated.
- Goto, jump, and exit statements are not dealt with.

With these restrictions, the description below is independent of the programming language and much simpler than a complete specification. At the same time, the description is general enough to give an overview of the method. Moreover, it allows straightforward extensions to overcome these restrictions.

In forward mode, the calculation of the derivative values is performed in the same order as for the original values. Consequently, the control flow (branches, loops) and the structure (procedure definitions and calls) of the program can be maintained.

Implementing a Forward-Mode Tool

The simplest version of a forward-mode tool first copies the original abstract syntax tree to obtain the derivative abstract syntax tree, which is then transformed by applying recursively the following forward differentiation rules:

- An assignment is differentiated by
  1. generating the derivative assignments according to section 3.1,
  2. associating new indices to the derivative statements,
  3. inserting the new statements before the original one,
  4. updating the control flow graph by adding the new arcs and modifying the old ones.
- A call to a procedure `eval(x, y)` is differentiated by
  1. generating a new call `eval_tang(x, dx, y, dy)`,
  2. associating the new call to a new index,
  3. replacing the original call by the new call,
  4. replacing the name of the original procedure by the name of the differentiated procedure in the call graph,
  5. updating the control flow graph by adding the new arcs and modifying the old ones.
- A control statement is differentiated by keeping the control as it is and applying the rules to the statements from the body.
- Any other statement is left unchanged.
- A body is differentiated by applying the differentiation rules to the statements from the body line by line and concatenating the results in the original statement order.

- A declaration is differentiated by
  1. generating the derivative declaration as a copy of the original declaration for the derivative variable,
  2. inserting the derivative declaration before the original one,

- A procedure `feval(x,y)` is differentiated by
  1. generating the derivative header `feval_tang(x,dx,y,dy)` from the original header `feval(x,y)`,
  2. generating the derivative declarations from the original ones,
  3. generating the derivative body from the original body,
  4. replacing each procedure declaration component by its derivative counterpart in the original declaration.
- A program is differentiated by applying the differentiation rules to the global declarations and each procedure in the program.

Note that the call- and control-flow-graphs are updated during the transformation of the abstract syntax tree. This method is easily extendable to all forward-mode variants.

Using the Forward-Mode Tool

Using the forward-mode tool, the user gives as input the source files of the program that defines the function to be differentiated and simply runs the AD preprocessor to generate the derivative source program. Note that the derivative program must be modified to initialize the independent derivative variables to the required values. Then, the derivative program may be used as any user-level program to obtain the derivative values.

We consider the same example as in section 6.1 on operator overloading. In context-sesitive mode, the forward AD tool generates the following.

```
 1 | double dalphai, dbetai;
 2   double aplha1, beta1;
 3 | double dx1, dx2, dy;
 4   double x1, x2, y;
 5   cin >> dx1, cin >> dx2;/* Added by the user */
 6   cin >> x1, cin >> x2;
 7 | dy = 0.0;
 8   y = 0.0;
 9   for (int i=0; i<n; i++)
10   { cin >> alphai; cin >> betai;
11     cin >> dalphai; cin >> dbetai; /* Added by the user */
12 |   dy = dy + 2*(x1-alphai)*dx1 - 2*(x1-alphai)*dalphai +
13 |              2*(x2-beta2i)*dx2 - 2*(x2-beta2i)*dbetai;
14     y = y + (x1-alpha1)*(x1-alphai) + (x2-betai)*(x2-betai); }
15    cout << dy; /* Added by the user */
16    cout << y;
```

Here we have marked each extra line added by the AD preprocessor with a vertical bar. Lines 5 to 8 of the generated code correspond to the initialization in the first loop of Table 3.4. The function evaluation continued in lines 9 to 14 represents the second loop of Table 3.4. The extraction of the computed values, i.e., the third loop of Table 3.4 is performed in the lines 15 and 16.

As we said before, the generated code is suboptimal: if only the derivative with respect to `x1`, `x2` is required, `alphai` and `betai` are passive, and the only nonzero contributions to the computation of `dy` are the first and the term of the right-hand side. In context-free mode, the evaluation of `dy` will contain only these two non-zero terms.

Note that in this code, the initialization of the derivative values is performed by reading in the values from the standard stream and the derivative values are printed on the standard stream. Because of the limitations described above, these statements are input/output and must be added by the user in the derivative program.

Implementing a Reverse-Mode Tool

In reverse mode, the evaluation of the derivative values is performed in the reverse order with respect to the original values. This makes the evaluation of the derivative statement more complex than in forward mode.

This section shows the simplest implementation of the reverse-mode: each derivative procedure executes the forward sweep (the original procedure and the storage of the pre-values of the assigned variables), and the return sweep (the values are restored and the derivatives are computed). This strategy is often used when writing adjoint code by hand because of its simplicity. Other strategies (including loop checkpointing) can be implemented by simply combining the forward/return sweeps described above in different manners as discussed in Chapter 12.

A reverse-mode AD preprocessor may proceed in the same way as the forward-mode tool: it duplicates the original syntax tree to generate the forward sweep and inverts it to obtain the return sweep. This approach does not work for programs with `goto`, `jump`, or `exit` statements. To handle such constructs, the transformation must be applied on the control flow graph instead of the syntax tree.

The abstract syntax tree is transformed by applying recursively the following reverse differentiation rules that construct the forward and return sweeps in parallel.

- An assignment is differentiated by
  1. generating the derivative assignments by using the rules from section 3.2,
  2. generating the store and restore statements for the modified variable,
  3. associating new indices to the derivative and record/retrieve statements,
  4. inserting the record and original statements in the forward sweep,
  5. inserting the retrieve and derivative statements in the return sweep,
  6. updating the control flow graph.
- A call to a procedure `eval`$(\mathtt{x}, \mathtt{y})$ is differentiated by
  1. generating a new call `eval_dual`$(\mathtt{bx}, \mathtt{x}, \mathtt{by}, \mathtt{y})$
  2. generating the store and retrieve statements for the modified parameters,
  3. associating the derivative call and record/retrieve statements to new indices,
  4. inserting the record statements and original call statement in the forward sweep,
  5. inserting the retrieve and derivative statements in the return sweep,
  6. updating the call graph,
  7. updating the control flow graph.
- A branch statement is differentiated by
  1. applying the differentiation rules to the statements in the body and generating the forward and return sweep bodies,
  2. generating the forward branch statement by replicating the original test and inserting the forward sweep body,
  3. generating the return branch statement by replicating the original test and inserting the return sweep body,
  4. inserting the forward branch in the forward sweep,
  5. inserting the return branch in the return sweep,
  6. updating the call graph,
  7. updating the control flow graph.
- A loop statement is differentiated by
  1. applying the differentiation rules to the statements to the statements in the body and generating the forward and return sweep bodies,
  2. generating the forward loop statement by replicating the original header and inserting the forward sweep body,
  3. generating the return loop statement by reverting the original header and inserting the return sweep body,
  4. inserting the forward loop in the forward sweep,
  5. inserting the return loop in the return sweep,
  6. updating the call graph,
  7. updating the control flow graph.
- Any constant statement is added to the forward sweep with no counterpart in the return sweep.

- A procedure is differentiated by
  1. generating the derivative declaration as a copy of the original declaration for the derivative variable,
  2. inserting the derivative declaration before the original one,
- A body is differentiated by applying the differentiation rules to the statements from the sequence in order: each statement of the forward sweep is concatenated at the end of the of the forward sweep body and each differentiated statement is concatenated at the beginning of the the return sweep body,
- A procedure with header `feval(xl, yl)` is differentiated by
  1. generating the derivative header `feval_dual(bxl, xl, byl, yl)`,
  2. replacing the procedure header by the derivative header,
  3. replacing the procedure declaration by the derivative declaration,
  4. replacing the procedure body by the derivative body.
- A program is differentiated by applying the differentiation rules to the global declarations and each procedure in the program.

Note that the call and control flow graphs are updated during the transformation of the abstract syntax tree.

The choice of the store/restore implementation is of great impact on the efficiency of the derivative program but is language and machine dependent and is therefore not described here.

Using the Reverse-Mode Tool

The reverse-mode AD preprocessor is used in the same way as a forward-mode AD preprocessor: the user gives as input all the source files of the program, simply runs the tool and insert the initializations of the derivative variables. Then, the derivative program may be used as any other user-level program to obtain the derivative values.

For the same example as in section 6.1, the context-sensitive reverse mode tool generates the following.

```
 1 | double save_aplhai, save_betai;
 2 | double balphai, bbetai;
 3   double aplha1[n], beta1[n];
 4 | double bx1, bx2, by;
 5 | double save_y[n];
 6   double x1, x2, y;
 7   cin >> x1; cin >> x2;
 8   y = 0.0;
 9   for(int i=0; i<n; i++)
10   { cin >> alphai; cin >> betai;
11 |   save_alphai[i] = alphai;
12 |   save_betai[i] = betai;
13 |   save_y[i] = y;
```

```
14     y = y + (x1-alphai)*(x1-alphai) + (x2-betai)*(x2-betai); }
15   cout << y;
16   cin >> by; /* Added by the user */
17 | bx1 = 0.0;
18 | bx2 = 0.0;
19 | balphai = 0.0;
20 | bbetai = 0.0;
21 | for (int i=n-1; i>=0; i--)
22 | { alphai=save_alphai[i];
23 |   betai=save_betai[i];
24 |   y=save_y[i];
25 |   bx1 += bx1 + 2*(x1-alphai)*by;
26 |   balphai -=  2*(x1-alphai)*by;
27 |   bx2 += 2*(x2-beta2i)*by;
28 |   bbetai -=  2*(x2-beta2i)*by; }
29   cout << bx1; /* Added by the user */
30   cout << bx2; /* Added by the user */
31   cout << balphai; /* Added by the user */
32   cout << dbetai; /* Added by the user */
```

Note that as in forward-mode, the initialization and retrieval of the derivative have been added manually but could have been generated automatically. Lines 7 to 15 represent the recording sweep of Table 3.7. The second for-loop of the return sweep in Table 3.7 can be found in lines 21 to 28. Lines 29 to 32 correspond to the final loop of the return sweep in Table 3.7. The generated code is suboptimal: if only the derivative with respect to `x1, x2` were required, `balphai` and `bbetai` should appear, which happens in context-free mode.

Context-Sensitive Transformation

If the source transformation methods presented above are applied, all (real) input variables are considered as independent, and all (real) output variables are considered as dependent variables. Hence all values of all modified variables are stored and restored in reverse mode. This naive approach corresponds to the brute-force use of the operator overloading tool when all `double` variables are retyped as `adouble`.

To generate a program that computes the derivative of the dependent with respect to the independent variables, the dispensable derivative statements must be discarded by an optimization phase, or the active statements must be determined and the differentiation process must be applied only to them. Identifying the useless statements by program optimization is much more expensive than not generating them in the first place.

The general purpose AD preprocessors apply a static analysis generally called "activity analysis" to detect whether a particular variable, statement, or procedure is to be considered active with respect to the independent and dependent variables. The AD preprocessor optimizes the generated code by generating

derivatives only for active program components (variable, statement, procedure). The activity analysis is performed on the original program before the forward or reverse mode differentiation and allows for the generation of an better derivative program by avoiding the differentiation of passive statements as much as possible.

In the same manner, storing and retrieving the values as it is performed in the naive reverse transformation are not efficient: the values of `y` are recorded all along the forward loop and restored all along the backward loop even though the value of `y` is not used. The "to-be-recorded analysis" described in [FN01] is also an a priori analysis that allows one to know for each occurrence of each variable whether it has to be recorded. However, this analysis has not yet been implemented in general purpose AD tools.

Note that, if the programming language allows the use of pointers, the previous analysis must be performed modulo aliases. Hence, for example, if a variable is active, all its aliases are also active. Alias analysis is a difficult subject on which a lot of work has been done (see, e.g. [Sta97, Deu94]), but it is not a problem specific to AD and is therefore not described here.

6.3 AD for Parallel Programs

The widespread use of large clusters and the advent of multicore processors have increased the push toward automatic differentiation of programs that are written by using libraries such as MPI [MPI] for message passing or pragma-based language extensions such as OpenMP [OMP]. The topic was first investigated in [Hov97]. We cannot hope to cover all the concepts by which parallel computing is supported, but we want to concentrate on the most important aspects of message passing as standardized by MPI and code parallelization enabled by OpenMP. The main problems arise in the context of source transformation AD but a few are applicable to AD via operator overloading as well. In practice, source transformation AD tools so far have limited coverage of parallel programming constructs.

Extended Activity Analysis

The preceding section discussed activity analysis, which determines the program variable subset that needs to carry derivative information. The data flow analysis described in section 6.2 covers the constructs of the respective programming language but not dependencies that are established through library calls such as MPI's `send`, `recv`, or the collective communication operations. On the other hand, these hidden dependencies clearly do not originate only with MPI constructs. The same effect can easily be recreated in sequential programs, for instance, by transferring data from program variable `a` to `b` by writing to and reading from a file.

The default solution to this problem is the assumption that any program variable `b` occurring in a `recv` call potentially depends on any variable `a` occur-

ring in an `send` call. In many practical applications this leads to a considerable overestimate of the active variable set. A recent attempt to reduce this overestimate is the MPI-enhanced control flow graph introduced in [SKH06]. A simple example of such a graph is shown in Fig. 6.5. We have a simple `switch` based on the MPI process rank (0-3) that determines the behavior for four processes executing in parallel. We start with the conservative assumption that all `recv`s are connected to all `send`s. In MPI the parameters that identify matching communication points are communicator, tag, and source/destination pairs. The communicator identifies a subset of the participating processes, the tag is an integer message identifier, and the source and destination are the process numbers within the respective communicator. The analysis on the MPI-enhanced control flow graph uses constant propagation to establish guaranteed mismatches between `send/recv` pairs and to exclude certain dependencies. For simplicity we left out the communicator and source/destination parameters in Fig. 6.5 and indicate only the potential data flow dependencies between the `send` and `recv` buffers.

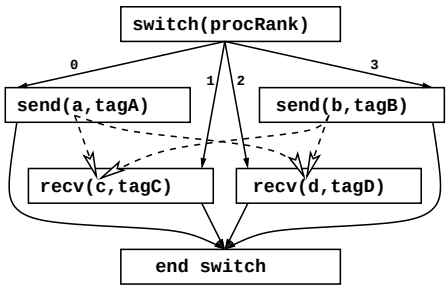

Figure 6.5: A Control Flow Graph Enhanced with Potential Communication Edges.

If the code before the `switch` vertex contained assignments for the `tag`s, for example, `tagA=tagC=1; tagB=tagD=2`, then the analysis can propagate these constants, determine the mismatch, remove the two diagonal communication edges, and thereby exclude the dependence of `d` on `a` and of `c` on `b`. Collective communications are handled in a similar fashion. While this approach is fully automatic, its efficacy depends to some extent on the coding style of the original program. Complementing the appropriate use of the identifying parameters can be optional pragmas to identify communication channels. The idea was first introduced in [Fos95] (Chapter 6), but no AD preprocessor has implemented this concept yet.

A different set of questions arises when one considers the implications for the data dependencies that can be inferred from OpenMP directives. Fig. 6.6 shows an example for a parallelizable loop where the actual dependencies are obfuscated by potential aliasing between arrays and by using a computed address that cannot be resolved with the typical induction variable mechanisms. Considering the main OpenMP workhorse `omp parallel do`, the need for this directive presumably is rooted in the inability of the automatic data flow and

dependence analyses to remove enormous dependencies from the conservative overestimate. Otherwise an autoparallelizing compiler could verify that the loop in question is indeed free of loop-carried dependencies and could generate parallel instructions right away. That is, one would not require an OpenMP directive in the first place. In turn, an AD tool that is aware of such parallelization directives can use the implied dependency exclusions to enhance the analysis result. In our example in Fig. 6.6 an `omp parallel do` directive implies that `a[i+o]` does not overlap with `b[i]`, and therefore the data flow analysis does not need to propagate along the dashed edges. This situation might not be automatically detected because the code analysis has to prove that the computed offset always yields a value $\geq$`k` and the arrays `a` and `b` are not aliased. Conservatively the analysis assumes loop-carried dependencies (dashed edges) preventing an automatic parallelization.

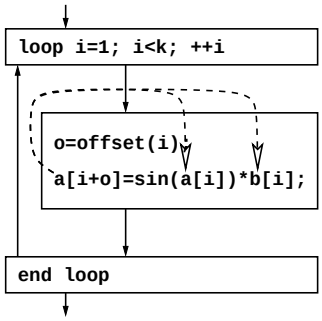

Figure 6.6: A Control Flow Graph with a Parallelizable Loop.

Further consequences of OpenMP directives on the reverse-mode code generation, are explained at the and of this section.

Parallel Forward Mode

From a naive point of view the transformation of a given parallel program for forward mode would amount to merely mirroring the parallelization constructs for the derivative data. There are, however, a number of technical problems and the potential for amplifying a load imbalance that the reader should be aware of.

In a simple implementation, the forward mode adds derivative data and operations for derivative computations in a uniform fashion to the entire original program. Section 4.5 gives a theoretical overhead factor $\in [2, 5/2]$ for the forward mode. When one employs vector forward mode with p directions, this overhead factor grows with a problem-dependent fraction of p. Consequently, runtime differences caused by a load imbalance that were already present in the original program will be amplified by this factor; see also [RBB07]. An AD-specific source of load imbalance in forward mode is the propagation of

dynamic sparse vectors [BK+97], when the nonzero elements are distributed unevenly across the processes.

The correct association between program variables and their respective derivatives under MPI might be considered a negligible implementation issue but has been a practical problem for the application of AD in the past [HB98, CF96].

There is a noteworthy efficiency aspect to the differentiation of reduction operations. A good example is the product reduction which is logically equivalent to the Speelpenning example discussed in section 3.3. Here, as well as in the other cases of arithmetic operations encapsulated within the MPI library calls, a direct AD transformation of the MPI implementation is not recommended. One has to determine a way to compute the product itself, the partial derivatives and propagate the directional derivatives. The computational complexity becomes easier to understand if one considers the reduction operations on a binary tree, as was done in [HB98]. The best forward-mode efficiency would be achieved with a special user-defined reduction operation that, for each node c in the tree with children a and b, simply implements the product rule $\dot{c} = b\dot{a} + a\dot{b}$ along with the product $c = ab$ itself. This approach allows the complete execution in just one sweep from the leafs to the root. Details on this topic can be found in [HB98]. MPI provides a user interface to define such specific reduction operations.

For OpenMP one could choose the safe route of reapplying all directives for the original program variables to their corresponding derivatives and stop there. When one considers the propagation of derivatives in vector mode, it can be beneficial to parallelize the propagation. When the derivative vectors are sufficiently long, a speedup of more than half the processor count was achieved for moderate processor counts [BL+01]. The approach made use of the OpenMP *orphaning* concept, which permits specifying parallelization directives outside the parallel region in which they are applied. The parallelization directives are applied to the routines that encapsulate the propagation without the overhead of parallel regions that are internal to these routines. In [BRW04] the concept of explicitly nesting the parallelization of the derivative vector propagation inside the given parallelization was explored, aimed at using a larger number of threads for the computation. The theoretical benefits have not yet been exploited in practice.

Parallel Reverse Mode

Solving the problems mentioned in previous two subsections is a prerequisite for generating the adjoint model for a parallel program. One can consider a `send(`a`)` of data in a variable a and the corresponding `recv(`b`)` into a variable b to be equivalent to writing b`=`a. The respective adjoint statements are $\bar{a}$`+=`$\bar{b}$`;` $\bar{b}$`=0`. They can be expressed as `send(`$\bar{b}$`);` $\bar{b}$`=0` as the adjoint of the original `recv` call and `recv(t)`; and $\bar{a}$`+=t` as the adjoint of the original `send` call, using a temporary variable `t` of matching shape. This has been repeatedly discovered and used in various contexts [FDF00, Cha90].

Interpreting a `recv` call as an assignment $b = a$, it is clear that one has

to replicate all the actions to record and restore the old values overwritten by the call to `recv(`b`)` when this is warranted, e.g. by TBR analysis. To keep the examples simple, here we omit the recording code altogether and also leave out any statements in the return sweep that would restore recorded values in overwritten `recv` buffers.

A concern for parallel programs is the correctness of the communication patterns and, in particular, the avoiding of deadlocks. Proving that a given program is free of deadlocks in practice is possible only for relatively simple programs. A deadlock can occur if there is a cycle in the *communication graph*. The communication graph for a program (see, e.g., [SO98]) is similar to the MPI-enhanced control flow graph shown in our example in Fig. 6.5; but instead of just adding edges for the communication flow, the communication graph also contains edges describing the dependencies between the communication endpoints. Often the noncommunication-related control flow is filtered out. The cycles relevant for deadlocks have to include communication edges – not just, for instance, loop control flow cycles. For the plain (frequently called "blocking") pairs of `send`/`recv` calls, the edges linking the vertices are bidirectional because the MPI standard allows a blocking implementation; that is, the `send`/`recv` call may return only after the control flow in the counterpart has reached the respective `recv`/`send` call. A cycle indicating a deadlock and the use of reordering and buffering to resolve it are shown in Fig. 6.7.

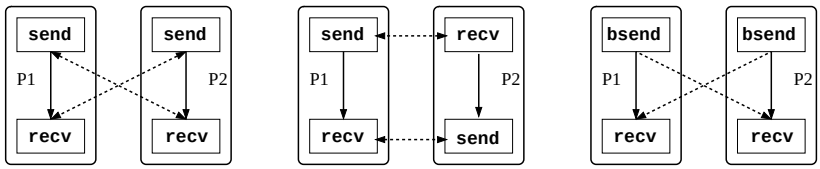

Figure 6.7: Deadlock, Reordered send/recv, buffered sends.

In complicated programs the deadlock-free order may not always be apparent. For large data sets one may run out of buffer space, thereby introducing a deadlock caused by memory starvation. A third option to resolve the deadlock, shown in Fig. 6.8, uses the non-blocking `isend(`a`,r)` which keeps the data in the program address space referenced by variable a and receives a request identifier `r`. The program can then advance to the subsequent `wait(r)` after whose return the data in the send buffer a is known to be transmitted to the receiving side. After the `wait` returns, the send buffer can be overwritten. The options for treating such programs have been explored in [TBD].

For the adjoint of the parallel program and the corresponding adjoint communication graph, the direction of the communication edges needs to be reversed. This imposes rules on the choice of the `send` call as the adjoint for a given `recv` and the treatment of `wait` calls. We can determine a set of patterns where simple rules suffice for the adjoint generation. To limit the number of distinct cases we assume that `send(`a`)` is equivalent to `isend(`a`,r); wait(r)` and similarly for `recv`.

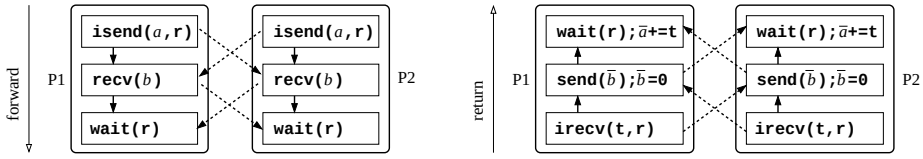

Figure 6.8: Nonblocking Send `isend` Followed by `wait` to Break Deadlock.

The straight forward edge direction reversal as shown in Fig. 6.8 is implied when the original program contains only calls fitting the adjoining rules listed in Table 6.1. We omit all parameters except the buffers a, b, and a temporary buffer `t` and the request parameter `r` for non-blocking calls.

Table 6.1: Rules for Adjoining a Restricted Set of MPI `send`/`recv` Patterns.

	Forward Sweep		Return Sweep	
	Call	Paired with	Call	Paired with
1	`isend(`a`,r)`	`wait(r)`	`wait(r);`$\bar{a}$`+=t`	`irecv(t,r)`
2	`wait(r)`	`isend(`a`,r)`	`irecv(t,r)`	`wait(r)`
3	`irecv(`b`,r)`	`wait(r)`	`wait(r);`$\bar{b}$`=0`	`isend(`$\bar{b}$`,r)`
4	`wait(r)`	`irecv(b,r)`	`isend(`$\bar{b}$`,r)`	`wait(r)`
5	`bsend(`a`)`	`recv(`b`)`	`recv(t);`$\bar{a}$`+=t`	`bsend(`$\bar{b}$`)`
6	`recv(`b`)`	`bsend(`a`)`	`bsend(`$\bar{b}$`);`$\bar{b}$`=0`	`recv(t)`
7	`ssend(`a`)`	`recv(`b`)`	`recv(t);`$\bar{a}$`+=t`	`ssend(`$\bar{b}$`)`
8	`recv(`b`)`	`ssend(`a`)`	`ssend(`$\bar{b}$`);`$\bar{b}$`=0`	`recv(t)`

The combinations of nonblocking, synchronous, and buffered send and receive modes not listed in the table can be easily derived. As evident from the table entries, the proper adjoint for a given call depends on the context in the original code. One has to facilitate the proper pairing of the `isend`/`irecv` calls with their respective individual `wait`s for rules 1–4 and also of `send` mode for a given `recv` for rules 5–8. An automatic code analysis may not be able to determine the exact pairs and could either use the notion of communication channels identified by pragmas or wrap the MPI calls into a separate layer. This layer essentially encapsulates the required context information. It has distinct `wait` variants and passes the respective user space buffer as an additional argument, for example, `swait(r,`a`)` paired up with `isend(`a`,r)`. Likewise the layer would introduce distinct `recv` variants; for instance, `brecv` would be paired with `bsend`.

Multiple Sources and Targets

In the examples considered so far, we have had only cases where the communication edges in the communication graph had single sources and targets. This is a critical ingredient inverting the communication. There are three common

scenarios where the single source/target property is lost.

1. Use of wildcard for the tag or the source parameter
2. Use of collective communication (reductions, broadcasts, etc.)
3. Use of the collective variant of `wait` called `waitall`

The use of the MPI wildcard values for parameters `source` or `tag` implies that a given `recv` might be paired with any `send` from a particular set; that is, the `recv` call has multiple communication in-edges. Inverting the edge direction for the adjoint means that we need to be able to determine the destination. A simple solution is to store the values of the actual tag and source during the recording sweep which may then be retrieved through MPI calls. Conceptually this means that we pick at runtime an incarnation of the communication graph in which the single source/target property is satisfied and that therefore can be inverted by replacing the wildcard parameters in the return sweep with the previously recorded actual values.

For collective communications the transformation of the respective MPI calls is essentially uniform across the participating calls. To illustrate the effect, we can consider a product reduction followed by a broadcast of the result, which could be accomplished by calling `allreduce` but here we want to do it explicitly. Essentially we compute the product $p = \prod a_i$ and broadcast the value to all processes i so that $b_i = p, \forall i$. The apparent adjoint is the summation reduction $\bar{p} = \sum \bar{b}_i$ followed by a broadcast of $\bar{p}$ and subsequent increment $\bar{a}_i$ `+=` $\frac{p}{a_i}\bar{p}$ assuming $p \neq 0$. The respective communication graphs are shown in Fig. 6.9 the processes to retain their rank between the forward and the reverse sweep.

The calculation of $\partial p/\partial a_i$ as p/a_i does not work when $a_i = 0$ and may be inaccurate when $a_i \approx 0$. In section 3.3, we recommend a division-free way of differentiating Speelpenning's product, which has a similar operations count but less parallelism.

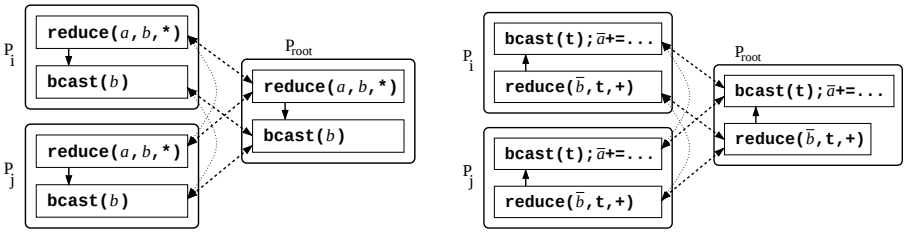

Figure 6.9: Adjoining of Collective Reduction and Broadcast.

Similar to treating the reduction in forward mode, there is again an efficiency concern that is best visualized by considering an execution of the reduction on a tree. In principle, the partials could be computed explicitly by using prefix and postfix reduction operations during the recording sweep. Alternatively, one could record the a_i and then in the return sweep first compute all the

intermediate products from the leaves to the root in the reduction tree followed by propagating the adjoints from the root to the leaves. This approach requires only two passes over the tree and is less costly than any approach using the explicit computation of the partials. Unlike the custom reduction operation for the forward case or the explicit partials computation using pre- and postfix reductions, MPI does not provide interfaces facilitating the two-pass approach. Consequently, one would have to implement it from scratch.

The use of `waitall` as a collective completion point poses the most complex problem of adjoining MPI routines that AD, at least in theory, can handle at the moment. This is a commonly used MPI idiom and occurs, for instance, in the logic of the MIT general circulation model [MIT]. There the adjoint MPI logic for the exchange of boundary layers of the grid partitions has been hand-coded because currently no source transformation tool can properly handle non-blocking MPI calls. Any change to the grid implementation necessitates a change to the respective hand-coded adjoint. This and the desire to provide choices for the grid to be used illustrate the practical demand for such capabilities.

Both the use of non-blocking point-to-point communications and the associated collective completion aim at reducing the order in processing the messages imposed on the message passing system by the program that uses it. Often such a program-imposed order is artificial and has been shown to degrade the efficiency on processors with multiple communication links. While one can imagine many different orders of calls to `isend`, `irecv`, and `wait`, without loss of generality we consider a sequence of `isend` calls, followed by a sequence of `irecv` calls followed by a `waitall` for all the request identifiers returned by the `isend`s and `irecv`s. For simplicity we assume all processes have the same behavior. We distinguish the buffers by an index, denote `r` as the vector of all request identifiers ($\mathtt{r}_1$,$\mathtt{r}_2$,...), and show in the communication graph only placeholder communication edges that refer to nodes in the representer process; see Fig. 6.10.

One could, of course, separate out all requests, introduce individual `wait` calls and follow the recipe in Table 6.1. That approach, however, imposes an artificial order on the internal message-passing system, which the use of `waitall` tries to avoid. Instead we introduce a nonoperational counterpart with nonoperational communication edges in the original code and perform a simple vertex transformation in which the original `waitall` vertex along with its problematic edges are rendered nonoperational, as shown in Fig. 6.10. The completion `waitall` has multiple communication in-edges which makes a simple vertex based adjoint transformation impossible. Instead we can introduce a symmetric nonoperational counterpart *anti wait* denoted as `awaitall` in the recording sweep and form the adjoint by turning the `awaitall` into a `waitall` and the original `waitall` into the non-operational `awaitall` in the return sweep.

With the transformations performed in the fashion suggested, we can now state that we are able to generate the adjoint communication graph by reversing the edge direction. Consequently, if the original communication graph was free of cycles then the adjoint communication graph also will be free of cycles and we can be certain that the adjoint transformations do not introduce deadlocks.

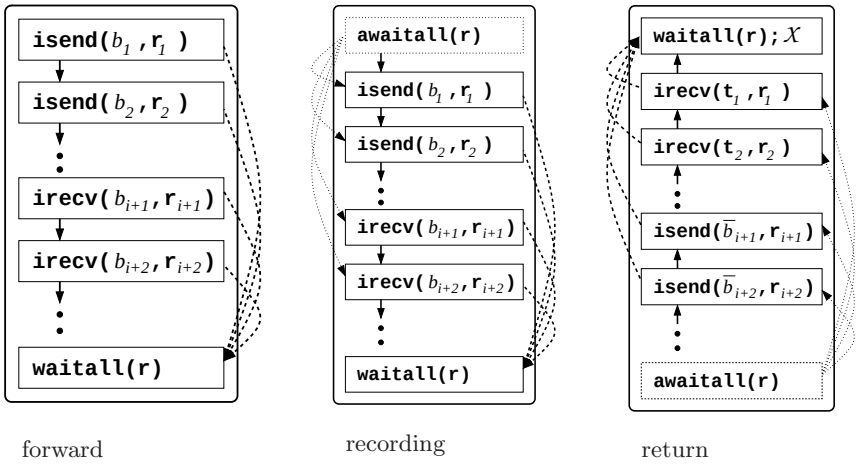

Figure 6.10: Adjoint `waitall` With the Use of an `anti_waitall`.

The final $\mathcal{X}$ in the rightmost graph in Fig. 6.10 denotes the buffer updates $\bar{b}_j$`+=t`$_j, j = 1, \ldots, i$ and $\bar{b}_j = 0, j = i+1, \ldots$ that have to wait for completion of the non-blocking calls. Here, similar to the recipes in Table 6.1, the additional context information that is required to accomplish $\mathcal{X}$ could be avoided if one wrapped the MPI calls and performed the bookkeeping on the buffers to be incremented and nullified inside the wrapper. However, the most efficient implementation would turn these wrappers into a specific set of library calls defined within the MPI standard which are guaranteed to be adjoinable. This should include the semantics for `send` calls that nullify the passed in buffer upon completion and also the semantics for `recv` calls the increment the passed buffer. The message passing system internally already needs to manipulate the buffer contents for heterogeneous environments that require data marshaling.

One frequently used MPI call is `barrier`, for instance in the context of `rsend`; see Fig. 6.11. The standard requires that a `recv` has to be posted by the time `rsend` is called, which typically necessitates a `barrier` in the recording sweep.

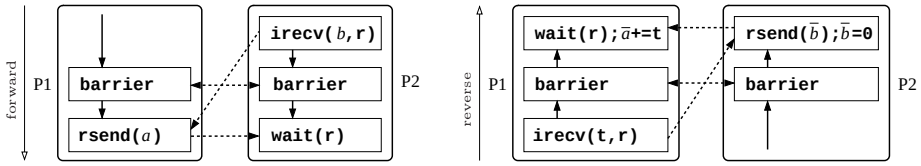

Figure 6.11: Handling of the `barrier` Routine.

For the adjoint the `barrier` call stays in place; however, a vertex transformation recipe requires context information or a nonoperational counterpart to the `rsend` similar to the treatment of `waitall` in Fig. 6.10. In a logically correct program

we can leave the `barrier` call in place for the adjoint transformation. We point out that a logically correct use of `barrier` to demarcate a critical section always requires synchronization on entry or exit in the original program whenever such synchronization is required for respective exit and entry of the adjoint of that critical section. MPI one-sided communication routines that resemble a shared-memory programming style require explicit library calls on entry and exit of the section performing remote memory accesses.

A naive view of the adjoint transformation of an OpenMP parallel loop assumes that dependencies in the adjoint loop could arise only from antidependencies (i.e., overwrites) in the original loop, and vice versa. While any such dependency would prevent the original loop from being parallelizable, the absence of such dependencies asserted by the OpenMP directive does not imply that the adjoint loop is parallelizable. We illustrate the problem in Fig. 6.12. The original loop (left) can be parallelized. A standard transformation inverting the loop direction and generating the adjoint of the loop body (right) exhibits various increment statements of `bx`. When this loop is parallelized, there is a race between the reads and the writes of the `bx` elements.

```
loop i=2; i<n; ++i
     a[i]=x[i-1]-2*x[i]+x[i+1]
     b[i]=a[i]+sqrt(x[i])
end loop
```

```
loop i=n-1; i>=2;--i
     ba[i]    += bb[i]
     bx[i]    += bb[i]*1./(2*sqrt(x[i])
     bb[i]    =  0
     bx[i-1]  += ba[i]
     bx[i]    += (-2)*ba[i]
     bx[i+1]  += ba[i]
     ba[i]    =  0
enddo
```

Figure 6.12: Introduction of a Race Condition by Adjoining.

Here the problem lies with the increment operations of the adjoints because there can be a race condition between the reads and the writes of the `bx[i-1]`, `bx[i]`, and `bx[i+1]` when the adjoint loop is executed in parallel. Erroneous computations caused by the race condition have been observed with science applications. If the increment operation were atomic, which is not guaranteed in practice, then this problem would disappear.

While it may not be possible to parallelize the adjoint loop, an important benefit is the ability to cheaply recompute the values needed by the adjoint loop body. In general the adjoint loop needs to be executed in the order reverse to the original loop. Recomputing values (with loop carried dependencies) needed by the adjoint implies either recording them or recomputing them at a cost quadratic in the number of loop iterations. A parallelizable forward loop implies that the iterations over the loop body can be executed in any order, for instance in the reverse order that may be required in the return sweep when the adjoint loop is not parallelizable, as in our example. The values needed for the adjoint of the loop body can then be recomputed immediately prior to the return sweep of that loop body iteration by running the recording sweep over that single iteration. Therefore, the quadratic recomputation complexity vanishes; see also

section 4.2.

6.4 Summary and Outlook

While the principles and basic techniques of algorithmic differentiation may are quite simple and may even appear trivial to some their effective application to larger problems poses quite a few challenges. Here the size and difficulty of a problem depends not so much on the sheer runtime of an evaluation procedure but the complexity of its coding. Heterogeneous calculations pieced together from various software components and possibly run concurrently on several platforms can realistically not be differentiated with current AD implementation. Perspectively, each software component provided by a public or private vendor should have calling modes for propagating sensitivity information forward or backward in addition to its normal simulation functionality. This would of course require an agreed upon standard for direct and adjoint derivative matrices in a suitably compressed format. For the time being such prospect seem a fair way off and for the most part we have to be content with differentiating suites of source programs in a common language, typically from the Fortran or C family. As described in section 6.3 concurrency encoded in MPI or OpenMP can be largely preserved through the differentiation process, through some care must be token to ensure correctness and maintain lead balancing. There is even some prospect of gaining parallelism for example through strip-mining of Jacobians or concurrent recomputations as briefly described at the and of Chapter 12. Naturally these kinds of technique will remain in the domain of expert users, who develop a large scale application where savings in wall-clock time are critical.

For user with small or medium-sized AD applications the following considerations for writing (or if necessary rewriting) code are useful.

Unless they are really completely separate, the evaluation of all problem functions whose derivatives are needed (e.g., optimization objectives and constraints) should be invoked by one *top-level* function call. In doing so one should make as much use as possible of common subexpressions, which then can also be exploited by the differentiation tool. For some tools the top-level call can be merely conceptual, that is represent a certain *active section* of the user program where function evaluations take place.

Within the subroutines called by the top-level function, are should avoid extensive calculations that are passive, in that they do not depend on the actual values of variables, (e.g., the setting up of a grid). Even if the corresponding code segments are conditioned on a flag and may, for example, be executed only at an initial call, the AD tool may be unable to predict this runtime behavior and wind up allocating and propagating a large number of zero derivatives. For the same reason one should not share work arrays or local variables between passive parts of the code and the top-level call that is to be differentiated. Generally, one should separate code and data structures as much as possible into an active and a passive part. If one knows that some calculations within

an active subroutine have no or only a negligible influence on the derivative values, one might go through the trouble of deactivating them by suppressing the dependence on the independent variables.

These coding style recommendations are not really specific to AD. They would similarly assist parallelizing or ("merely") optimizing compilers. Anything that simplifies compile-time dependence analysis by making the control and data flow more transparent is a good idea. Whenever possible, one should provide fixed upper bounds on array sizes and iteration counters.

6.5 Examples and Exercises

Exercise 6.1 (*Scalar Forward by Overloading*)
Code the `adouble` proposal for the forward mode as outlined in section 6.1 including the four basic arithmetic operations $+, -, *$, and / and the intrinsic functions sin, cos, exp, and sqrt. For flexibility you may add mixed mode versions where one argument is a `double` for the four binary operations, though this is not necessary for the examples suggested.
a. Check and debug your implementation on the trivial test functions

$$y = x - x, \quad x/x, \quad \sin^2(x) + \cos^2(x), \quad \text{sqrt}(x * x), \quad \text{and} \quad \exp(x)$$

and reproduce the results listed in Table 1.2 for the baby example.
b. For the norm problem discussed in Exercise 2.2, compare the accuracy obtained by your implementation with that of difference quotient approximations.
c. Overload the incremental (C++) operation += for `adouble`s, and use it to simplify the code for the norm problem.

Exercise 6.2 (*Scalar Reverse by Overloading*)
Code the `adouble` proposal for the reverse mode as outlined in section 6.1 with the same elemental operations as for your `adouble`s in Exercise 6.1. Check and debug your code on the same trivial test functions.
a. Reproduce the results in Table 1.3 for the baby example.
b. Apply your implementation to Speelpenning's product as discussed in Exercise 3.6. At the argument $x_i = i/(1+i)$, compare the results and runtimes to a hand-coded adjoint procedure and also to n run of your forward implementation from Exercise 6.1.
c. If one of the source transformation tools listed in section 6.2 is or can be installed on your system, verify the consistency of its results with yours and compare compilation times and runtimes.

Exercise 6.3 (*Second-Order Adjoints*)
Combine the forward and the reverse mode by modifying the class `adouble` from your reverse mode implementation. Verify the correctness of this extension of the scalar reverse mode on the coordinate transformation example discussed in Exercise 5.5.

Exercise 6.4 (*Second Derivatives Forward*)
In order to propagate first and second directional derivatives, implement a version of the class `adouble` that has the same structure as the `adouble` for the forward mode, but whose field `dot` is of type `adouble`. For this purpose, you may simply replicate the code overloading the restricted set of elementals for `adouble`s. Check the correctness of your implementation on the trivial identities, and compute the acceleration of the light point in the lighthouse example of section 2.1. Observe that the new `adouble` could also be defined as a `adouble<adouble<double>>`, namely, by a recursive application of templates. Such techniques are at the heart of the expression templates techniques [Cés99] used to handle complicated right-hand sides more efficiently.

Exercise 6.5 (*Vector Forward by Overloading*)
As suggested at the end of subsection 6.2 on page 119, code a class `vector` providing additions between them and multiplication by a real scalar.
a. Replicate the source generated for the `adouble` class in Exercise 6.1 above with `dot` now a `vector` rather than a `double`. As in the previous exercises you may define `adouble`s as a class template parametrized by the type of `dot`.
b. Test the new vector implementation of the forward-mode on the trivial identities, the norm example, and Speelpenning's product. Compare the runtimes with that of the scalar forward multiplied by n.

Exercise 6.6 (*Taylor Forward*)
To propagate higher derivatives forward, write an active class `taylor` whose data member is a vector of d coefficients. Implement arithmetic operations and intrinsic functions according to the formulas given in Tables 13.1 and 13.2, respectively. Test your implementation on the trivial test functions and the lighthouse example to compute time derivatives of arbitrary order.

Exercise 6.7 (*Overloading Assignments*)
All exercises above can be performed in a language that does not allow the overloading of assignments. However, this restriction may severely impair efficiency for the following reason.

In the vector mode implementation according to the previous exercise, the results from each arithmetic operation and other elemental function are first placed into a temporary return variable and then typically copied to a named variable on the left-hand side of an assignment. To avoid the often repeated copying of the vector part of a `adouble` one may redefine its field `dot` to be merely a pointer to a `vector` of derivatives, whose length need not be limited at runtime. These objects can be dynamically allocated within each overloaded elemental and by the initialization function `makeindepvar` for the independent variables.

However, as one can easily see, for example, on Speelpenning's product, each multiplication would allocate an additional vector and none of those would ever be released again so that the storage demand would grow by a

factor of about n compared to the base implementation according to Exercise 6.5. The simplest way to avoid this effect is to overload the assignment so that the old vector pointed to by the left-hand side is deallocated, for example, by the statement `free(dot)`.

a. Modify your vector forward-mode as sketched above, and verify on Speelpenning's example that the results are correct and the storage is only of order n. Check whether the elimination of vector copying actually reduces the runtime.

b. Construct an example where an intermediate variable is first assigned to another named variable and then receives a new value itself. Show that the modified vector forward cannot work properly and correct it by adding a reference counter to the data structure `vector`. Now deallocation happens only when the reference counter has been reduced to zero.

Exercise 6.8 (*User Defined Constructors/Destructors*)
Even the last implementation sketched in Exercise 6.7 can work properly only if the `dot` is automatically initialized to a null pointer upon variable construction. This is ensured by most compilers, even if not strictly enforced by all language standards. Otherwise looking up and modifying the reference counter may already cause segmentation faults. In C++ the user can be sure of proper initialization by writing his or her own constructor function for variables of type `adouble` and such. Similarly, he or she can write a corresponding destructor routine that is called by the compiler whenever a variable of type `adouble` goes out of scope. In our context, the destructor can decrement the reference counter of the `vector` structure being pointed to by the variable being destructed. Otherwise, the reference counter mechanism introduced in part **b** of Exercise 6.7 cannot really work properly on multilayered programs, though the derivative values obtained would be correct.

Upgrade the last version of vector forward once more by adding a constructor and a destructor for `adouble` as described above. Test the efficacy of the new version, for example, on the norm problem recoded such that the squaring of the x_i happens in a function with at least one local variable.