Compilation aspects of Automatic Differentiation by examples

Christèle Faure*

Presented at "Les séminaires du projet A3" in March 2000

1 Introduction

This document describes challenging problems general enough to serve as target applications to the compilation community. This tentative description is rather naïve in the vocabulary used as well as in the examples given. In our point of view, Automatic Differentiation could be an ideal application domain for compilation techniques.

But one must stay really close to the final objective of **Automatic Differentiation** which is **the extraction of derivative information from an original code**. From a practical point of view, there is no point to get really complicated information if the process of generation of the derivative code cannot take advantage of them. An other key point is that the AD tools must handle really large codes (100000 lines), so precise analysis can only be performed on small parts of the original code.

We try to give samples (written in Fortran 77) for all the points which are developed in this document. These examples come from various applications treated with `Odyssée`, but are general enough to be representative of AD problems.

2 AD basic

AD technology is based on the use of the chain rule to differentiate a program as a the composition of elementary functions. Each elementary function is implemented within a statement which is easily differentiated as shown in Section 2.1. From this, a straight line program is also derived in a straight forward manner as shown in Section 2.2. The only thing to be dealt with is the overwritting of variables in reverse mode. A general program can be differentiated is the control is managed. In reverse mode, the management of the control is made difficult by the necessity to go back in the computation as shown in Section 2.3.

2.1 One assignment

The assignment A shown in Figure 1(a) can be seen as the mathematical elementary function f which input and output domains are $\mathbb{R}^2$ and $\mathbb{R}$.
In order to built up the composition of the elementary functions corresponding to a sequence of statements, one has to extend the input and output domains: $\mathbb{R}^3$ and $\mathbb{R}^3$ in our example. This leads to consider all the variables involved in the computation as potential input and output.

*Email : Christele.Faure@polyspace.com

The product of the Jacobian matrix of A by some direction $(dX, dY, dZ)_i$ can be easily built. Figure 2 shows two equivalent representations of this computation using mathematical notation: Figure 2(a) shows the matrix computation and Figure 2(b) shows the equivalent scalar computation where the subscripts $_i$,$_o$ indicate input and output respectively.

The product of the transposed Jacobian matrix of $\mathcal{F}$ by the direction in the dual space $(dX^\star, dY^\star, dZ^\star)_i$ can also be easily built. Figure 3 shows two equivalent representations of this computation in the same manner as Figure 2.

From the two scalar computations shown in Figure 2(b) and Figure 3(b), one can simply get the corresponding tangent and cotangent straight line programs. We introduce the $_i$ and $_o$ subscripts because a mathematical variable can be set but can never be modified in place. On the contrary, computer variables are memory locations and can therefore be modified in place. In order to optimise the code in terms of number of intermediate variables, the X_i and X_o variables are identified and denoted by X. The statement $dX_o = dX_i$ is then transformed into $dX = dX$ and discarded. Figures 1(b) and 1(c) show respectively the optimised tangent code and the optimised cotangent code of A.

2.2 Straight line programs

Let consider two new statements B (Example 4(a)) and C (Example 5(a)). They are both differentiated using the same method as for A as shown in Example 4 and Example 5.

Sequence $A; B$ (Example 6(a)) can be differentiated in a straight forward way using the derivatives of A and B because both computations are independents. The tangent code $(A; B)'$ (Example 6(b)) of $A; B$ is only the sequence $A'; B'$ computed with the initial values X_0 and Y_0 of `X` and `Y`. The cotangent code $(A; B)^\star$ is $B^\star; A^\star$ computed with the initial values of `X` and `Y`. The only constraint here is to run B after B' and $B^\star$.

On the contrary, sequence $B; A$ introduces a dependency between B and A because `Y` is computed in B and used in A. Let denote by Y_0 the initial value of Y and Y_1 its value after the execution of B: B' and $B^\star$ must be evalauted on `Y=`Y_0 whereas A' and $A^\star$ must be evaluated on `Y=`Y_1. This dependency constraint on the original variables leads to the generation of $(B; A)'$ as shown in Example 7(b): as a result statement B is executed between B' and A'. This is the general method for generating the tangent code: the derivative statement is executed before the original one. In reverse mode a new constraint is added: if statement B is executed before A in the original code, $B^\star$ must be executed after $A^\star$. As a result some storage (or recomputation) must be performed to have the correct value of Y at the right time. If one stores the two necessary values Y_0 and Y_1, the generated code is Example 7(c). The general method for generating cotangent codes is to execute first the `forward` part: the original function plus the storage of the modified necessary values and then the b̂ackward part: the derivative function plus the retrieval of the modified necessary values.

The generation methods illustrated in Example 7 are the standard method: they are certainly correct but under-optimal in some cases.

2.3 Complex statement

The control used in the original source is reproduced in the derivative code. For example, the loop Example 8(a) is differentiated in Example 8(b) in direct mode: the loop is reproduced and the body A is replaced by A'. In reverse mode, the same loop can be differentiated using various method: Example 8(c) or Example 8(d). In reverse mode, the number of scalar values to be stored are quite different from one example to the other example.

The same is performed for `if-then-else`, `dowhile` commands ... As for the straight line program differentiation these strategies could certainly be refined to get optimal (storage or recomputation) derivative

codes. For example in reverse mode a problem is: what is the smallest amount of control values to be stored to be able to follow the call-graph backward.

3 AD internal

In this document, we are only looking at AD as a compile-time technique. Any of those tools is based on the composition of phases which are transformation phases as well as analysis phases. In this way AD can be seen as a specific source transformation. The main difference with other general transformation (translation from a language to an other, optimisation ...) is that the generated code does not only computes the original values but some others which have a mathematical meaning. An other crucial difference is that the analysis to be performed are not local but global all over the program.

In the following sections, we describe briefly the main different phases necessary to differentiate a code.

3.1 Pre-process

This phase consists of a standardisation of the original code before differentiation. The transformation performed at this level depends on the treated language as well as the limitations of the system.

Within `Odyssée` this phase consists in:

- rewriting the non-differentiable operators `abs`, `min` into `if`-statements,
- in-lining of the `function` statements,
- splitting of expressions into equivalent binary expressions by introducing intermediate variables.

3.2 AD dependency analysis

During this phase, the system makes an inter-procedural analysis of the program, which results in a dependency graph between the input/output variables of each unit. Only variable names are considered, even if the dependencies between components of array would help.

The information required to differentiate a routine is: at each point in a program which variables are (possibly) impacted by the active inputs chosen by the user. From this the system is able to extract two information:

1. which variable must be associated to a derivative at each point in the program ,
2. which variable is modified by each sub-program of the program.

The first information depends on the active variables chosen by the user, whereas the second one depends only on the original code.

Within `Odyssée`, two passes are used to generated the whole information. During the first pass (bottom-up in the call-graph) the dependencies read/write are computed. During the second pass (top-down in the call-graph) the activity flag is propagated from the input of the top sub-program to all the other sub-programs.

3.3 Trajectory analysis

This information is necessary to generate a code that stores all the modified variables to reproduce the execution by only restoring these variables. Such a code is absolutely necessary to the reverse mode of AD.

Within the present version of `Odyssée`, all the variables modified by the original code are stored, as well as the context of all sub-program called. The storage analysis is then equivalent to the AD dependencies analysis if you consider all the inputs as active.

3.4 Generation of the derivative code

This phase aims at generating the derivative code which computes the derivatives as well as the original values. At this level, the unit by unit differentiation of the program is done, according to the following two rules.

3.5 Post-process

The fourth phase aims at simplifying the resulting code. The algebraic expressions are simplified in the same way as in Computer Algebra Systems except that the arguments of sums and products cannot be reordered modulo associativity and commutativity. First, each sub-program of the program is differentiated with respect to its input variables, and not with respect to the input variables of the head-unit. Secondly, the differentiation is *maximal* in the sense that a sub-program is differentiated with respect to the maximum set of input variables appearing in every `call` statement. In this way, only one sub-program for each unit of the original program. The methods used to differentiate a sub-program within `Odyssée` are described by examples in [FP98]. Other methods which should be automatized within AD tools are described in [Fau99].

4 Norm verification

As described in the following section, some default within the original code accepted by compilers lead to great trouble using the reverse mode. The main reason is that the status (read, written) of the adjoint variables is the transposed status of the corresponding original variable.

If the compilers where able to check for these problems, AD would be a lot more straight forward.

4.1 Aliasing through calls to sub-programs

The derivative in reverse mode of one assignment is different depending on the fact that the variable is overwritten. Let consider the two instructions Example 9(a) and Example 9(b): the first one induces no rewriting whereas the second one rewrites `Y`. Example 10(a) shows the cotangent code when the assigned variable `y` is not overwritten, Example 10(b) shows the corresponding derivative code when `y` is overwritten. This dependency is easy to detect for scalar variables, a little bit more difficult for array but can still be handled.

If the same problem appears through a call to a sub-program this becomes really difficult.

Let look at the following example named `cumul (X,Y,A,B,N)` witch computes $\forall i \in [1..N] Y(i) = X(i) + A(i) * B(i)$ as shown in Example 11(a). The cotangent code of `cumul` with respect to `x,a,b` is shown in

Example 11(b).

If `cumul` is called with `X` and `Y` pointing to the same memory zone like with `call cumul (Y,Y,A,B,N)`, the corresponding call to the cotangent code `call cumulcl (Y,Y,A,B,N,YCCL,YCCL,ACCL,BCCL)` will not compute the correct derivatives. It will compute `YCLL(i)` correctly but reset it to zero afterwards. When this problem arises in a large code, it is really difficult to detect.

One must notice that this aliasing case has no effect one the derivatives generated in direct mode. Essentially, the direct mode will translate the aliasing pattern from the original variables to the derivative ones.

The first problem is to detect if this case happens in the code to be differentiated.

AD tools have chosen to generate a unique derivative for each original sub-program. In this case, there is no way to generate a cotangent code correct for any aliasing pattern. The only possibility is then to introduce intermediate variables in the original code or in the derivative code.

A second possibility would be to generate clones of the original aliased sub-program and to call them at the correct points in the original code. Then the automatic differentiation will be standard.

A third possibility is to generate a derivative with one entry for each aliasing pattern. The derivative code would test the aliasing pattern and use the correct entry.

4.2 Implicit aliasing

The code Example 14(a) is to be differentiated with respect to `x, a, b`. In this piece of code, the variables `y2, y3` are modified even though it does not appear clearly in the source. As a result, only `y1` depends on `x` which leads to a two levels problem:

1. no derivative is associated to `y2, y3`,
2. the values of `y2,y3` are nor stored nor restored.

The first problem leads to wrong tangent and cotangent derivatives. Example 15(a) shows that no derivative is associated to `y3`. The correct tangent code Example 15(b) can only be generated if `y3` depends on `x, a, b`. It is the same in reverse mode even if it is less clear.

The second problem only influences the cotangent derivatives: `y2` is nor set before the call to `cumul` nor reset before the call to `cumulcl` as shown in Example 16(a). The correct version Example 16(b) introduces the necessary trajectory management statements as well as the supplementary derivative computation.

These problems are really difficult to isolate within a large derivative code. Is such an error possible to detect on the original code ?

5 Derivative code optimisation

5.1 Selection of output derivatives

Until now, within `Odyssée` one is able to generate the derivative of a sub-program P (the whole sub-tree from the routine P in the call tree) with respect to some of its inputs I. In general, what is really needed is the derivative of some outputs O of P with respect these inputs I. The code generated by `Odyssée` is

not optimal because it computes the derivative of all the outputs instead of a subset O. This problem can be solved by slicing the original code, or the generated code.

Example 17 shows the original code as well as the standard tangent code with respect to `X1` generated by `Odyssée`. Let consider a first case for which only the derivatives of `Z` with respect to `X1` are needed. The automatically generated code Example 17(b) is clearly not optimal.

Example 19 shows the original code Example 17(a) sliced with respect the output variable `Z` in Example 18(a) and then differentiated with respect to `X1` in Example 18(b). This result is better compared to the initial derivative, but can still be optimised because the computation of `Z` is not required any more. On the contrary, if the tangent code Example 17(b) is sliced with respect to `ZTTL` then the derivative code Example 19(a) is minimal.

Let call pre-slicing the combination slicing w.r.t original variables and automatic differentiation, and post-slicing the combination automatic differentiation followed by slicing w.r.t derivative variables. In general, the optimal method is the post-slicing. On some examples, it is even the only way to minimise the computation. Let consider a second case for which only the derivatives of `K` with respect to `X1` are required. The pre-slicing will lead to no gain whereas the post-slicing will result in Example 19(b) where the computation of `Y, K` have been removed.

The main drawback of the post-slicing is that it is applied on a generated code that can be a lot larger that the original one and the slicing process would certainly be expansive. For example, the number of variables is doubled (without considering the store variables) and the number of lines is multiplied by 2 in direct mode and by n in reverse mode (where n is the number of active variables in the right hand side of the assignment). In reverse mode, a second problem arises as original and cotangent statements can be far from one another.

A specific algorithm for slicing the derivative when it is generated would be the best solution. Such a technique should predict the "to be sliced statements" and tell the system not to generate it into the derivative code.

5.2 Trajectory management optimisation

In reverse mode, the generated code has to store and retrieve a large amount of values. We have proven that on a large example this pop/push of scalar values are 1/3 of the total execution time (see [Fau98]).

The problem is the same for the direct and reverse part with transposed constraints. The constraints due to the chosen adjoining strategy are not studied here but certainly have to be taken into account. For clarity reasons, we only consider the direct parts of the calculation: original code plus trajectory storage.

The standard strategy is to store the values of all the modified variables before modification, the more this analysis is precise the minimal the trajectory is. An other remark is that the user will run larger and larger test cases. As a conclusion one can say that the trajectory analysis has to be more and more precise to minimise its size, but it does not exclude trajectory management optimisation.

The first idea is that the scalar pop and push commands could be gathered to use array pop and push commands. This can be viewed as a bufferization of the values before the whole buffer is pushed, except that a various sized buffer is used. If a push of n scalar values is cheaper in time than n pushs of a scalar values, the derivative code will be optimised. One can notice that one or zero pop statement is added to each original statement in a standard cotangent code. It is possible to let the push statements float (as latex figures) along the original code to gather some of them.

Let call **trajectory management ratio** the ratio between the number of trajectory moves `push, pop` and the number of original operations `+,*,sin`. In Example 20(a) each iteration of the loop pushes one scalar value and computes one scalar value. This is certainly penalising because the trajectory

management ratio n/n is high, in Example 20(b) the trajectory management ratio is 1/n In the same way, the Example 20(c) can be optimised into Example 20(d).

The second idea is that the original computation could be run on one processor and the push statements could be executed on a second processor.

5.3 General checkpointing

Let consider you want the adjoint code of a loop of length n Example 21(a) that modifies some variable `Y`. You can either store the n values of `Y` like in Example 21(b) or recompute them from the initial value like in Example 21(c). Using the first solution you compute only n times F but store the n values of Y, whereas using the second solution you compute $n + n(n + 1)/2$ computations of F but store only one value of Y.

A trade-off between execution time and memory requirement is called *checkpointing*. It consists in storing some values of Y in checkpoints and recomputing the others values from these checkpoints like in Example 21(d).

If you have $N < n$ checkpoints to reverse a loop with n steps, it is possible to compute their optimal repartition that minimises the recomputation of F: the `next_check` choose the optimal value. This optimal schedule is described in different papers [GPRS96, Gri92] and uses the hypothesis that each iteration execution cost t, and that the size of the values to checkpoint Y is the same.

Until now, the optimal checkpointing method has been only applied on explicit loops. If one wants to use it in a more general way, two problems arise:

1. the original execution has to be split into *equal measure* pieces,
2. the code that recomputes the original function from one checkpoint to an other must be written.

What is the smallest set of variables to be stored at one checkpoint to be able to recompute the original function until the next checkpoint ?

5.4 Iterative directional derivatives

Let consider the computation of two directional derivatives with the same original input values but two different input directions. If these directions are independent, they can be computed at the same time and an n-directional tangent or cotangent code can be computed. If they are dependent, the only solution is to use two 1-directional tangent Example 22(a)or cotangent as shown in Example 22(b). In this case, it is clear that the original function is computed twice.

We are studying some way to share the execution of the function between two executions [Gri00] of a 1-directional tangent or cotangent code. A first solution Example 23(c) is: to execute once the original code Example 23(a) that stores some `costly` original values and then apply several times a fast version Example 23(b) of the standard 1-directional tangent. A second solution Example 24(c) is: to execute first a 1-directional tangent code Example 24(a) that stores some `costly` original or derivative values and then apply several times a fast version Example 23(b) of the standard 1-directional tangent. Both solutions can be merged by implementing some modified version of the original code where `costly` original and derivative values are stored.

The optimised iterative multi-directional tangent code Example 23(c) or Example 24(c) is more efficient than the standard one if the trade-off between storage and recomputation is good. Is there any way to chose automatically the values to be stored to be sure of the gain.

5.5 Really used array dimension

A lot of implementation of numerical codes must be run for various sizes of problems (size of the grid for example). To do so, a maximal size is chosen at compile time to allocate all the array, and the real size is given at runtime.

This really standard method Example 25(a) Example 25(b) has a great influence on the efficiency of the cotangent code as shown in Example 25(c). The `ndim` components of `X` are stored and restored whereas only `n` should be. The first solution for us is to ask the user to declare the really used size of its arrays. But then this could lead to errors, if this property is not verified all along the source code. If the system could detect such a property, it would help.

6 AD enhancement

6.1 AD dependency propagation

For this problem, we did not found any example within our experiments. But this will arise when coupled codes will have to be differentiated. The dependency `Z` depends on `Y` is difficult to detect in Example 26: the dependency comes from the write of `Y` in and the read of `Z` from the same file `'output.data'`. As a result if one differentiates `sub0` with respect to `X`, the variable `Z` must become active. The present version of `Odyssée` generates Example 27 that computes no derivative for `Z`. The correct tangent code is presented in Example 28.

To solve this problem, one can think of considering all read/write instructions to be active, but then the activity propagation would be really under-optimal. The generated code would compute and store a lot of zero derivatives.

Is it possible to detect the dependency of `Z` to `Y` ?

6.2 Completely overwritten arrays

At a glance to Example 29, one can see that each component of vector X is set to 224 in `sub1`. This is not detected by `Odyssée` dependency analysis.

To compute the directional derivative he wants, the user has to set the input direct `XTTL` to the value he wants before calling `sub0tl`. But the code automatically generated Example 30 resets the input direction `XTTL` to zero. As a result, all the computed derivatives `ZTTL, KTTL` are computed to zero. This is mathematically correct: the function `sub0` is constant with respect to `X` and its derivative is zero.

On a large code, this leads to a really strange behaviour for the derivative code because the direction may be reset to zero in the middle of the code. The system must raise a warning saying that `X` is not an input variable for `sub2` and that `sub2tl` would have a constant derivative. Then the user differentiates the program from `compute` instead of `sub0` and obtains the derivative values he wants. To raise this warning, the system must be sure that the whole array `X` is overwritten.

6.3 Array dimension

Array dimensions can be declared to be `*` or `1` wherever in the source code. This is a ficticious size used just to tell the compiler that the variable is an array.

The AD system has to store all the components of `X` before the call to `sub2` and restore it before the call to `sub2cl`. To generate a correct derivative, it needs to know the declared size `n` even though it could use the used size `imax*jmax`. One must notice that the part of the code loaded by the system is not in general the complete but only a part of it.

Is it possible to propagate the declared size of each arrays through the call-graph? If it is so, the system could raise warning or generate special comments to tell the user that some size declarations are missing.

6.4 Inter-procedural analysis

Example 32 shows an original program where a sub-program is passed as an arguments to `sub1`. The current version of `Odyssée` does not treat such a case and the tangent code automatically generated Example 33 is not correct. The correct code for this example should be Example 34.

An other case that `Odyssée` does treat is the recursivity which is accepted in some f77 compilers.

Both problems are certainly not difficult to sort out in AD tools because it must be treated within compilers.

6.5 Sensitivity of the generated code to numerical errors

There is a measure function that checks the coincidence between the tangent and cotangent codes. We have used this measure to compare on one example the cotangent code written by hand and the automatically generated one (using `Odyssée`). It appears that for the same tangent code, the fit is much more important for the hand written cotangent code than for the automatically generated one.

We think that the sensitivity of the generated code to numerical errors should be investigated.

References

[Fau98] C. Faure. Le gradient de THYC3D par Odyssée. Rapport de recherche 3519, INRIA, October 1998.

[Fau99] C. Faure. Adjoining strategies for multi-layered programs. Rapports de recherche 3781, INRIA, 1999.

[FP98] C. Faure and Y. Papegay. Odyssée User's Guide. Version 1.7. Rapport technique 0224, INRIA, September 1998.

[GPRS96] J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Applications, Techniques, and Tools*, pages 95–106. SIAM, 1996.

[Gri92] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[Gri00] A. Griewank. *Principles and Techniques of Algorithmic Differentiation.* SIAM, 2000.

```
Z = X * Y**2
```
(a) A

```
dZ = Y**2 * dX + 2*X*Y*dY
```
(b) $A' = J_{\mathcal{F}} * d$

```
dX = dX + Y**2*dZ
dY = dY + 2*X*Y*dZ
dZ = 0.
```
(c) $A^\star = J_{\mathcal{F}}^\top * d^\star$

Figure 1: Derivatives of the assignment A

$$\begin{pmatrix} dX \\ dY \\ dZ \end{pmatrix}_o := \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Y^2 & 2XY & 0 \end{pmatrix}}_{\text{(a)}} \begin{pmatrix} dX \\ dY \\ dZ \end{pmatrix}_i \qquad \underbrace{\begin{array}{lcl} dX_o & := & dX_i \\ dY_o & := & dY_i \\ dZ_o & := & Y^2 dX_i + 2XY dY_i \end{array}}_{\text{(b)}}$$

Figure 2: Product of the Jacobian matrix of A by the direction $(dX, dY, dZ)_i$

$$\begin{pmatrix} dX^\star \\ dY^\star \\ dZ^\star \end{pmatrix}_o := \underbrace{\begin{pmatrix} 1 & 0 & Y^2 \\ 0 & 1 & 2XY \\ 0 & 0 & 0 \end{pmatrix}}_{\text{(a)}} \begin{pmatrix} dX^\star \\ dY^\star \\ dZ^\star \end{pmatrix}_i \qquad \underbrace{\begin{array}{lcl} dX^\star_o & := & dX^\star_i + Y^2 dZ^\star_i \\ dY^\star_o & := & dY^\star_i + 2XY dZ^\star_i \\ dZ^\star_o & := & 0. \end{array}}_{\text{(b)}}$$

Figure 3: Product of the transposed Jacobian matrix of A by the dual "direction" $(dX^\star, dY^\star, dZ^\star)_i$

```
Y = Y**3 * X**2
```
(a) B

```
dY = 3*Y**2*dY * X**2 + 2*Y**3*X*dX
```
(b) B'

```
dX = dX + 2*X*Y**3 * dY
dY =   3*X**2*Y**2 * dY
```
(c) $B^\star$

Figure 4: Derivatives of the assignment B

```
Y = X * Y**2
```
(a) C

```
dY = Y**2 * dX + 2*X*Y*dY
```
(b) C'

```
dX = dX + Y**2*dY
dY = 2*X*Y*dY
```
(c) $C^\star$

Figure 5: Derivatives of the assignment C

```
Z = X * Y**2
Y = Y**3 * X**2
```
(a) $A; B$

```
dZ = Y**2 * dX + 2*X*Y*dY
dY = 3*Y**2*dY * X**2 + 2*Y**3*X*dX
```

```
Z = X * Y**2
Y = Y**3 * X**2
```
(b) $(A; B)'$

```
dX = dX + 2*X*Y**3 * dY
dY =   3*X**2*Y**2 * dY

dX = dX + Y**2*dZ
dY = dY + 2*X*Y*dZ
dZ = 0.
```

```
Z  = X * Y**2
Y  = Y**3 * X**2
```
(c) $(A; B)^\star$

Figure 6: Derivatives of the sequence $A; B$

```
Y = Y**3 * X**2
Z = X * Y**2
```
(a) $B; A$

```
dY = 3*Y**2*dY * X**2 + 2*Y**3*X*dX
Y  = Y**3 * X**2

dZ = Y**2 * dX + 2*X*Y*dY
Z  = X * Y**2
```
(b) $(B; A)'$

```
Y0 = Y
Y  = Y**3 * X**2
Y1 = Y
Z  = X * Y**2

Y  = Y1
dX = dX + Y**2*dZ
dY = dY + 2*X*Y*dZ
dZ = 0.

Y  = Y0
dX = dX + Y**2*dY
dY = 2*X*Y*dY
```
(c) $(B; A)^\star$

Figure 7: Derivatives of the sequence $B; A$

```
do i=e1, e2, s
   A
end do
```
(a)

```
do i=e1, e2, s
   A'
end do
```
(b)

```
save1 = e1
save2 = e2
save3 = s
do i=e1, e2, s
   A
end do

e1 = save1
e2 = save2
s =  save3
do i=e1+k*s, e1, -s
   A*
end do
```
(c)

```
trace_idx=1
trace(trace_idx) = 1
do i=1, e, s
   trace_idx += 1
   trace(trace_idx)=2
   A
end do

do i=trace_max, 1, -1
   block = trace(i)
   if (block.eq.2) then
      A*
   end if
end do
```
(d)

Figure 8: Do loop differentiation

```
y    = x    +    a*b
```
(a) No overwritting

```
y    = y    +    a*b
```
(b) Overwritting of `Y`

Figure 9: Source code of Case 4

```
accl = accl + yccl*b
bccl = bccl + yccl*a
xccl = xccl + yccl
yccl = 0.
```
(a) No overwritting

```
accl = accl + yccl*b
bccl = bccl + yccl*a

yccl = yccl
```
(b) Overwritting of `Y`

Figure 10: Cotangent of Case 4(a) and 4(b) with respect to `x,a,b`

```
subroutine cumul (x,y,a,b,n)
integer n
real x(n),y(n),a(n),b(n)

do i=1, n


   y(i) = x(i) + a(i)*b(i)
end do
end
```
(a) Source code of cumul

```
subroutine cumulcl (x,y,a,b,n,
                    xccl,yccl,accl.bccl)

do i=1, n
   xccl(i) = xccl(i) + yccl(i)
   accl(i) = accl(i) + yccl(i)*bccl(i)
   bccl(i) = bccl(i) + yccl(i)*accl(i)
   yccl(i) = 0.
end do
end
```
(b) Cotangent code of cumul

Figure 11: Cotangent of `cumul` with repect to `y,a,b`

```
call cumul (X,Y,A,B,N)
```
(a) No overwritting

```
call cumul (Y,Y,A,B,N)
```
(b) Overwritting of `Y`

Figure 12: Source code of Case 5

```
call cumulcl (X,Y,A,B,N,XCCL,YCCL,ACCL,BCCL)
```
(a) No overwritting

```
call cumulcl (Y,Y,A,B,N,XCCL,YCCL,ACCL,BCCL)
```
(b) Overwritting of `Y`

Figure 13: Cotangent of Case 5(a) and 5(b) with respect to `x,a,b`

```
subroutine sub0 (x,a,b)
integer ijkm
parameter (ijkm=10)
common /total/y1, y2, y3
real y1(ijkm), y2(ijkm), y3(ijkm)
real a(3*ijkm), b(3*ijkm), x(3*ijkm)

a(1) = y2(4)**2
call cumul (x,y1,a,b,3*ijkm)
b(1) = y3(4)**2
end
```

(a) Main subprogram

Figure 14: Source code of Case 5

```
subroutine sub0tl (x, a, b, xttl, attl, bttl)

attl(1) = 2*y2ttl(4)*y2(4)
a(1) = y2(4)**2
call cumultl(x, y1, a, b, 3*ijkm,
             xttl, y1ttl, attl, bttl)
bttl(1) = 0.
b(1) = y3(4)**2
end
```

(a) Incorrect code

```
subroutine sub0tl (x, a, b, xttl, attl, bttl

attl(1) = 2*y2ttl(4)*y2(4)
a(1) = y2(4)**2
call cumultl(x, y1, a, b, 3*ijkm,
             xttl, y1ttl, attl, bttl)
bttl(1) = 2*y3(4)*y3ttl(4)
b(1) = y3(4)**2
end
```

(b) Correct code

Figure 15: Incorrect and correct tangent codes of Case 5 with respect to `y2, x, a, b`

```
subroutine sub0cl (x, a, b, xccl, accl, bccl)

save1 = a(1)
a(1) = y2(4)**2
DO nnn1 = 1, ijkm
   save2(nnn1) = y1(nnn1)
end DO
call cumul(x, y1, a, b, 3*ijkm)
save3 = b(1)
b(1) = y3(4)**2

b(1) = save3
bccl(1) = 0.
DO nnn1 = ijkm, 1, -1
   y1(nnn1) = save2(nnn1)
end DO
call cumulcl(x, y1, a, b, 3*ijkm,
             xccl, y1ccl, accl, bccl)
a(1) = save1
y2ccl(4) = y2ccl(4)+accl(1)*(2*y2(4))
accl(1) = 0.
end
```

(a) Incorrect code

```
subroutine sub0cl (x, a, b, xccl, accl, bccl

save1 = a(1)
a(1) = y2(4)**2
DO nnn1 = 1, 3*ijkm
   save2(nnn1) = y1(nnn1)
end DO
call cumul(x, y1, a, b, 3*ijkm)
save3 = b(1)
b(1) = y3(4)**2

b(1) = save3
y3ccl(4) = y3ccl(4) + 2*y3(4)*bccl(1)
bccl(1) = 0.
DO nnn1 = 1, 3*ijkm
   y1(nnn1) = save2(nnn1)
end DO
call cumulcl(x, y1, a, b, 3*ijkm,
             xccl, y1ccl, accl, bccl)
a(1) = save1
y2ccl(4) = y2ccl(4)+accl(1)*(2*y2(4))
accl(1) = 0.
end
```

(b) Correct code

Figure 16: Incorrect and correct cotangent codes of Case 5 with respect to `y2, x, a, b`

```
Y = alpha * X1 + X2
Z = beta * X1 * X2
K = Y + Z**2
```

(a) Original code

```
YTTL = alpha * X1TTL
Y    = alpha * X1 + X2

ZTTL = beta * X2 * X1TTL
Z    = beta * X1 * X2

KTTL = YTTL + 2*Z*ZTTL
K    = Y + Z**2
```

(b) Standard tangent code

Figure 17: Original and standard tangent codes of Problem 2 w.r.t `X1`

```
Z = beta * X1 * X2
```

(a) Original code

```
ZTTL = beta * X2 * X1TTL
Z = beta * X1 * X2
```

(b) Tangent code

Figure 18: Slicing of the original code with respect to `Z`

```
ZTTL = beta * X2 * X1TTL
```

(a) with respect to `ZTTL`

```
YTTL = alpha * X1TTL

ZTTL = beta * X2 * X1TTL
Z    = beta * X1 * X2

KTTL = YTTL + 2*Z*ZTTL
```

(b) with respect to `KTTL`

Figure 19: Slicing of the standard tangent code

```
do i=1, n
   call push_scalar (x(i),1)
   x(i) = x(i)**2
end do
```

(a) Standard

```
call push_buffer (x(1),n)
do i=1, n
   x(i) = x(i)**2
end do
```

(b) Optimised

```
call push_scalar (z,1)
z  = y1
call push_scalar (y1,1)
y1 = x**2
call push_scalar (y2,1)
y2 = a*x
call push_scalar (y3,1)
y3 = b
```

(c) Standard

```
call push_scalar (z,1)
z  = y1
call push_buffer (y1,1,y2,1,y3,1)
y1 = x**2
y2 = a*x
y3 = b
z  = y1
```

(d) Optimised

Figure 20: Standard and optimised direct parts

```
do i=1, n
   Y = F(Y)
end do
```

(a) Original

```
do i=1, n
   S(i) = Y
   Y    = F(Y)
end do

do i=n, 1, -1
   Y    = S(i)


   YCCL = F'(Y,YCCL)
end do
```

(b) All storage

```
S = Y
do i=1, n

   Y = F(Y)
end do

do i=n, 1, -1
   Y = S
   do j=1, i-1
      Y = F(Y)
   end do
   YCCL = F'(Y,YCCL)
end do
```

(c) All recomputation

```
push_check (Y,0)
do i=1, n

   Y = F(Y)
end do

do i=n, 1, -1
   pop_check (Y,p)
   if (p.eq.n-1) then
      YCCL = F'(Y,YCCL)
   else next_check(p,n,q)
        do j=1, q
           Y = F(Y)
        end do
        push_check (Y,q)
end do
```

(d) Checkpoints

Figure 21: Loop cotangent code

```
subroutine sub0 (x,y,xttl,yttl)

lttl = xttl + 2
l    = x+2

yttl = -cos(l)*lttl
y    = sin(l)
end
```

(a)

```
call compute_direction (dx1)
call sub0tl (x,y,dx1,dy1)

call change_direction (dx1,dy1,dx2)
call sub0tl (x,y,dx2,dy2)
```

(b)

Figure 22: Standard iterative direction computation

```
subroutine sub0_store (x,y)

l    = x+2

s2   = sin(l)
push (s2)
y    = s2
end
```

(a)

```
subroutine sub0tl_fast
          (x,y,xttl,yttl)

lttl = xttl + 2
l    = x+2

yttl = -cos(l)*lttl

pop (s2)
y    = s2
end
```

(b)

```
call sub0_store (x,y)

call compute_direction (dx1)
call sub0tl_fast (x,y,dx1,dy1)

call change_direction (dx1,dy1,dx2)
call sub0tl_fast (x,y,dx2,dy2)
```

(c)

Figure 23: Optimised iterative direction computation: First solution

```
subroutine sub0tl_store
          (x,y,xttl,yttl)

lttl = xttl + 2
l    = x+2

s1   = cos(l)
push(s1)
yttl = -s1*lttl

s2   = sin(l)
push (s2)
y    = s2
end
```

(a)

```
subroutine sub0tl_fast
          (x,y,xttl,yttl)

lttl = xttl + 2
l    = x+2

yttl = -cos(l)*lttl

pop (s2)
y    = s2
end
```

(b)

```
call compute_direction (dx1)
call sub0tl_store (x,y,dx1,dy1)

call change_direction (dx1,dy1,dx2)
call sub0tl_fast (x,y,dx2,dy2)
```

(c)

Figure 24: Optimised iterative direction computation: Second solution

```
subroutine sub1 (n,X,Y)
parameter (ndim = 100000)
real X(ndim), Y

do i=1, n
   X(i) = X(i)**2
end do
Y = 0
do i=1, n
   Y = Y+X(i)
end do
end
```

(a)

```
subroutine sub0 (n,X,Y)
parameter (ndim = 100000)
real X(ndim), Y

call sub1 (10,X,Y)
end
```

(b)

```
subroutine sub0cl (n,X,Y)
parameter (ndim = 100000)
real X(ndim), Y
real XCCL(ndim), YCCL

call setsave (X,ndim)
call sub1(10,X,Y)

call getsave (X,ndim)
call sub1cl (10,X,Y,XCCL,YCCL)
end
```

(c)

Figure 25: Really used array size

```
  subroutine sub0 (X,Y)

C Computes Y from X
  CALL compute (X,Y)

  CALL writefile (Y,'output.data')
  CALL readfile  (Z,'output.data')

C Computes K from Z
  CALL compute (Z,K)
  end
```

(a) Main sub-program

```
subroutine writefile (X,filename)

open (1,file=filename)
write (1,*) X
close(1)
end

subroutine readfile (X,filename)

open (1,file=filename)
read (1,*) X
close(1)
end
```

(b) Auxiliary sub-programs

Figure 26: Source code of Case 1

```
  subroutine sub0tl (X,Y,XTTL,YTTL)

C Computes Y from X and YTTL from X, XTTL
  CALL computetl (X,Y,XTTL,YTTL)

  CALL writefiletl (Y,'output.data',YTTL)
  CALL readfile  (Z,'output.data')

C Computes K from Z
  CALL compute (Z,K)

  end
```

(a)

```
subroutine writefiletl (X,XTTL,filename)

open (1,file=filename)
write (1,*) X
close(1)
end
```

(b)

Figure 27: Incorrect tangent of Case 1

```
  subroutine sub0tl (X,Y,XTTL,YTTL)

C Computes Y from X and YTTL from X, XTTL
  CALL computetl (X,Y,XTTL,YTTL)

  CALL writefiletl (Y,'output.data',YTTL)
  CALL readfiletl  (Z,'output.data',ZTTL)

C Computes K from Z and KTTL from Z, ZTTL
  CALL computetl (Z,K,ZTTL,KTTL)

  end
```

(a)

```
subroutine writefiletl (X,filename,XTTL)

open (1,file=filename)
write (1,*) X, XTTL
close(1)
end

subroutine readfiletl (X,filename,XTTL)

open (1,file=filename)
read (1,*) X, XTTL
close(1)
end
```

(b)

Figure 28: Correct tangent of Case 1

```
      subroutine sub0 (X,Y)

      REAL    X(IC,JC,1), Y, Z
      INTEGER K, J, I




      CALL sub1 (X,224)

C Computes Z from X
      CALL compute (X,Z)
      end
```

(a) Main sub-program

```
subroutine sub1 (X,Y)

REAL    X(IC,JC,1), Y
INTEGER K, J, I

DO K=1,KC
   DO J=1,JC
      DO I=1,IC
         X(I,J,K)=Y
      END DO
   END DO
END DO
end
```

(b) Aux. sub-program

Figure 29: Original code of Case 3

```
      subroutine sub0tl (X,Y,XTTL,YTTL)

      REAL    X(IC,JC,1), Y, Z
      REAL    XTTL(IC,JC,1), YTTL, ZTTL
      INTEGER K, J, I



      VAL    = 224
      VALTTL = 0.
      CALL sub1tl (X,VAL,XTTL,VALTTL)

C Computes Z from X
C      and ZTTL from X and XTTL
      CALL computetl (X,Z,XTTL,ZTTL)
      end
```

(a)

```
subroutine sub1tl (X,Y,XTTL,YTTL)

REAL    X(IC,JC,1), Y
REAL    XTTL(IC,JC,1), YTTL
INTEGER K, J, I

DO K=1,KC
   DO J=1,JC
      DO I=1,IC
         XTTL(I,J,K)=YTTL
         X   (I,J,K)=Y
      END DO
   END DO
END DO
end
```

(b)

Figure 30: Tangent code of Case 3 with respect to `X`

```
subroutine sub0 (X)
common /size/n,imax,jmax
real X(n), Y(n)

call sub1(X)
end
```

(a)

```
subroutine sub1 (X,Y)
real X(1), Y(1)

call sub2(X,Y)
end
```

(b)

```
subroutine sub2 (X,Y)
common /size/n,imax,jmax
real X(*)
real Y(*)

do i=1, imax*jmax
   X(i) = X(i)*Y(i)
end do
end
```

(c)

Figure 31: Variable array dimension

```
function sub0 (x,y,z)
real x(n), y(n), z(n), sub2
external sub2

call sub1 (sub2,x,y,sub0,n)
end
```

(a)

```
function sub2 (x,y)
real x, y, sub2

sub2 = x + y
end
```

(b)

```
subroutine sub1 (fun,x,y,z,n)
real x(n), y(n), z(n)
do i=1, n
   z(i) = fun (x(i),y(i))
end do
end
```

(c)

Figure 32: Original code

```
subroutine sub0tl (x, y, z, n,
                   xttl, yttl, zttl)

integer n
real x(n), y(n), z(n)
real xttl(n), yttl(n), zttl(n)

call sub1tl(sub2, x, y, z, n,
                  xttl, yttl, zttl)
end
```

(a)

```
subroutine sub1tl (fun, x, y, z, n,
                           xttl, yttl, zttl)

real x(n), y(n), z(n)
real xttl(n), yttl(n), zttl(n)

do i = 1, n
   call funtl(x(i), y(i), z(i),
              xttl(i), yttl(i), zttl(i))
end do
end
```

(b)

Figure 33: Incorrect tangent code

```
function sub0tl (x,y,xttl,yttl)
integer n
real x(n), y(n), z(n),sub2
external sub2
real xttl(n), yttl(n), sub2tl
external sub2tl

call sub1tl (sub2,x,y,z,n,
         sub2tl,xttl,yttl,zttl)
end
```

(a)

```
subroutine sub2tl (x,y,z,
                   xttl,yttl,zttl)
real x, y, z

zttl = xttl + yttl
z = x + y
end
```

(b)

```
subroutine sub1tl (fun,x,y,z,n,
                   funtl,xttl,yttl,zttl)
integer n
real x(n), y(n), z(n)
real xttl(n), yttl(n), zttl(n)

do i=1, n
   call funtl(x(i),y(i),z(i),
              xttl(i),yttl(i),zttl(i))
end do
end
```

(c)

Figure 34: Correct tangent code