Linearity Analysis for Automatic Differentiation*

Michelle Mills Strout[1] and Paul Hovland[2]

[1] Colorado State University, Fort Collins, CO 80523
[2] Argonne National Laboratory, Argonne, IL 60439

Abstract. Linearity analysis determines which variables depend on which other variables and whether the dependence is linear or nonlinear. One of the many applications of this analysis is determining whether a loop involves only linear loop-carried dependences and therefore the adjoint of the loop may be reversed and fused with the computation of the original function. This paper specifies the data-flow equations that compute linearity analysis. In addition, the paper describes using linearity analysis with array dependence analysis to determine whether a loop-carried dependence is linear or nonlinear.

1 Introduction

Many automatic differentiation and optimization algorithms can benefit from linearity analysis. Linearity analysis determines whether the dependence between two variables is nonexistent, linear, or nonlinear. A variable is said to be linearly dependent on another if all of the dependences along all of the dependence chains are induced by linear or affine functions (addition, subtraction, or multiplication by a constant). A variable is nonlinearly dependent on another if a nonlinear operator (multiplication, division, transcendental functions, etc.) induces any of the dependences along any of the dependence chains.

One application of linearity analysis is the optimization of derivative code generated by automatic differentiation (AD) via the reverse mode. AD is a technique for transforming a subprogram that computes some function into one that computes the function and its derivatives. AD works by combining rules for differentiating the intrinsic functions and elementary operators of a given programming language with the chain rule of differential calculus. One strategy, referred to as the forward mode, is to compute partials as the intrinsic functions are evaluated and to combine the partials as they are computed. For example, AD via the forward mode transforms the loop in Figure 1 into the code in Figure 2. This code would need to be executed N times, with appropriate initialization of variable `d_x`, to compute the derivatives of `f` with respect to all N elements of array `x`. The reverse mode of AD results in less computation in the derivative code when the number of input variables greatly exceeds the number of output variables. Figure 3 shows the derivative code after applying the reverse mode.

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy under Contract W-31-109-Eng-38.

```
a = 0.0
f = 0.0
for i = 1, N
  a += x[i]*x[i]
  t = sin(a)
  f += t
end
```

Fig. 1. Example loop.

```
a = 0.0
d_a = 0.0
f = 0.0
d_f = 0.0
for i = 1, N
  a += x[i]*x[i]
  d_a += 2*x[i]*d_x[i]
  t = sin(a)
  d_t = cos(a)*d_a
  f += t
  d_f += d_t
end
```

Fig. 2. Example loop after automatic differentiation via the forward mode.

Hascoet et al.[6] observed that the forward computation and adjoint accumulation can be fused if the original loop is parallelizable. In fact, a weaker condition suffices: the two computations can be fused whenever there are no loop-carried, nonlinear dependences. The example in Figure 1 includes only one loop-carried dependence and the dependence is linear; therefore, the adjoint loop may be reversed and fused with the original loop as shown in Figure 4.

Such transformations can result in significant performance improvements and storage savings, by eliminating the need to store or recompute overwritten intermediate quantities such as variable `a`. Data dependence analysis [2, 4, 17, 13] is used to determine whether a loop is parallelizable. Precise dependence analysis techniques can determine which variables are involved in a loop-carried dependence. In the example of Figure 1, such techniques can determine that there is a loop-carried dependence involving the variable `f` and itself. In this paper, we present a technique for determining if the loop carried dependence is linear or nonlinear.

We present a data-flow formulation for linearity analysis. Linearity analysis determines which variables depend linearly or nonlinearly on which other variables. The result of the analysis is a determination of the non-existence or existence of a dependence between all pairs of variables in a program and the nature of the dependences if it exists. Determining if a loop-carried dependence is linear only requires checking whether the dependence between the variables involved in the loop-carried dependence is linear. For the example in Figure 1, the analysis summarizes the dependences between variables as shown in Table 1.

2 Formulation of Linearity Analysis as a Data-flow Analysis

Linearity analysis can be formulated as a forward data-flow analysis [10]. Data-flow analysis involves representing the subroutine to be analyzed as a control flow graph, such as the one shown in Figure 6 for the code in Figure 5. A control flow

```
a[0] = 0.0
f = 0.0
for i = 1, N
  a[i] = a[i-1] + x[i]*x[i]
  t = sin(a[i])
  f += t
end
for i = N, 1, -1
  a_t = a_f
  a_a[i] = cos(a[i])*a_t
  a_a[i-1] = a_a[i]
  a_x[i] = 2*x[i]*a_a[i]
end
```

Fig. 3. Example loop after reverse mode automatic differentiation. Notice that the temporary variable a must be promoted to an array to store results needed in the reverse loop.

```
a[0] = 0.0
f = 0.0
for i = 1, N
  a[i] = a[i-1] + x[i]*x[i]
  t = sin(a[i])
  f += t
end
for i = N, 1, -1
  a_t = a_f
  a_a[i] = cos(a[i])*a_t
  a_a[i-1] = a_a[i]
  a_x[i] = 2*x[i]*a_a[i]
end
```

Fig. 4. Adjoint code after reversing the adjoint loop and fusing it with the original computation.

Variable	f	t	a	x
f	linear	linear	nonlinear	nonlinear
t	$\top$	$\top$	nonlinear	nonlinear
a	$\top$	$\top$	linear	nonlinear
x	$\top$	$\top$	$\top$	$\top$

Table 1. The variables in the first column depend on the variables in the first row in the specified way. $\top$ indicates no dependence.

graph contains directed edges between basic blocks indicating possible control flow in the program. Each basic block b has a set of predecessors $pred(b)$ and a set of successors $succ(b)$, and the graph contains unique entry and exit nodes.

Data-flow analysis propagates data-flow information over the control-flow graph. For linearity analysis, the data-flow information is which variables are dependent on which other variables and whether that dependence is linear or nonlinear, which we refer to as the dependence class. The analysis assigns each ordered pair of variables a value from the lattice shown in Figure 7. For example, in the statement

```
x = 3*z + y**2 + w/(v*v),
```

x has an linear dependence on z ($\langle\langle x, z\rangle, linear\rangle$), a nonlinear dependence on y ($\langle\langle x, y\rangle, nonlinear\rangle$), a nonlinear dependence on w ($\langle\langle x, w\rangle, nonlinear\rangle$), and a nonlinear dependence on v ($\langle\langle x, v\rangle, nonlinear\rangle$).

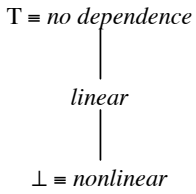

Fig. 7. Lattice for linearity analysis.

```
int foo() {
if (flag) {
  a = b;
  d = sin(a);
  u = x;
} else {
  a = b+1;
  for (i=0; i<10; i++) {
    u = u*x;
  }
}
f = u;
}
```

Fig. 5. Example code for control flow graph.

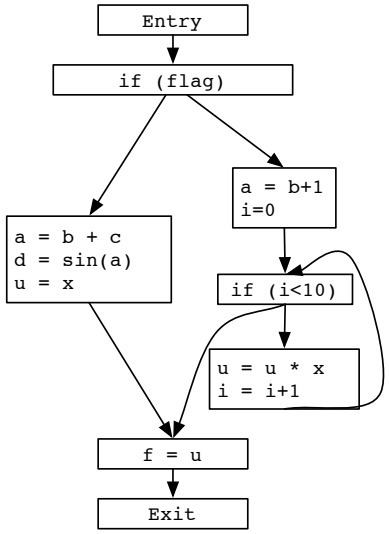

Fig. 6. Control Flow Graph

The set $IN(b)$ includes the dependence class assignments for each ordered variable pair that are valid at the entry of basic block b in the control-flow graph. A transfer function $f_b(IN(b))$ calculates the $OUT(b)$ set, which includes the dependence class assignments valid upon exiting the basic block b. The dependence class for each variable pair $\langle u, v\rangle$ is initialized to $\top$ in $IN(b)$ and $OUT(b)$ for all basic blocks in the control-flow graph.

Iterative data-flow analysis visits each node in the control-flow graph computing the $IN(b)$ and $OUT(b)$ sets until the assignment of data dependence class to each ordered variable pair converges. The set $IN(b)$ is calculated by performing the pairwise meet over all the sets of data-flow facts valid upon exiting predecessor basic blocks,

$$IN(b) = \sqcap_{p \in preds(b)} f_p(IN(p)).$$

The meet operation $\sqcap$ is performed on the lattice values assigned to each variable pair. The semantics of the meet operation $\sqcap$ is defined by the lattice. For example, $linear \sqcap \top$ equals $linear$.

The set $OUT(b)$ is computed by applying what is referred to as a transfer function to the $IN(b)$ set. The transfer function f_b is first defined for each type of statement assuming only one statement per basic block. If there are multiple statements in block b then f_b is the composition of all the transfer functions for the statements. To define the transfer function f_b for linearity analysis, we define the set $DEPS$. $DEPS(e)$ is a set containing a mapping of each variable to a data dependence class. When variable v maps to dependence class $class$, $\langle v, class\rangle$, that indicates how an expression e depends upon that variable. The

transfer function f_b for the assignment statement $x = e$ is then defined as

$$f_b(IN(b)) = \{\langle\langle x, v\rangle, class, nd, dd\rangle | \langle v, class, nd, dd\rangle \in DEPS(e).$$

The $DEPS$ set is defined in Table 2, where k represents an integer constant value or variable, v and w represent variables in the set of all variables V, `anyop` represents any operation, and `power` represents the power operation.

Expression e	$DEPS(e)$
k k anyop k k power k	$\{\langle v, \top\rangle \mid v \in V\}$
v	$\{\langle v, linear\rangle\}$ $\cup \{\langle w, class\rangle \mid \langle\langle v, w\rangle, class\rangle \in IN(b)\}$
$e_1 \pm e_2$	$\{\langle v_1, (class_1 \sqcap class_2)\rangle$ $\quad \mid v_1 = v_2$ and $\langle v_1, class_1\rangle \in DEPS(e_1)$ $\quad\quad$ and $\langle v_2, class_2\rangle \in DEPS(e_2)\}$ $\cup \{\langle v, class\rangle$ $\quad \mid \langle v, class\rangle \in DEPS(e_1)$ and $v \notin DEPS(e_2)\}$ $\cup \{\langle v, class\rangle$ $\quad \mid \langle v, class\rangle \in DEPS(e_2)$ and $v \notin DEPS(e_1)\}$
$e_1 * e_2$ e_1 / e_2	$\{\langle v_1, (nonlinear \sqcap class_1 \sqcap class_2)\rangle$ $\quad \mid v_1 = v_2$ and $\langle v_1, class_1\rangle \in DEPS(e_1)$ $\quad\quad$ and $\langle v_2, class_2\rangle \in DEPS(e_2)\}$ $\cup \{\langle v, nonlinear)\rangle$ $\quad \mid \langle v, class\rangle \in DEPS(e_1)$ and $v \notin DEPS(e_2)$ $\}$ $\cup \{\langle v, nonlinear)\rangle$ $\quad \mid \langle v, class\rangle \in DEPS(e_2)$ and $v \notin DEPS(e_1)$ $\}$
e_1 power 1 e_1 power k	$\{\langle v, (linear \sqcap class)\rangle \mid \langle v, class\rangle \in DEPS(e_1)\}$ $\{\langle v, (nonlinear \sqcap class_1)\rangle$ $\quad \mid \langle v, class\rangle \in DEPS(e_1)\}$

Table 2. Definition of the $DEPS$ set for each expression.

The worst-case complexity of linearity analysis is $O(N^4(E + V))$, where N is the number of variables, E is the number of edges in the control-flow graph, and V is the number of nodes in the control flow graph. Each pair of variables has a lattice value associated with it, and there are N^2 pairs. Each lattice value may be lowered at most twice; therefore, the graph may be visited $2 * N^2$ times. The size of the graph is $E+V$. When each node is visited, the computation may apply meet and transfer operations to each variable pair, $O(N^2)$.

2.1 Detecting Nonlinear Loop Carried Dependences

Data dependence analysis provides information about which variable references are involved in loop-carried dependences. If a particular variable is involved in

```
for i = 1 to N
  b[i] = 3 * x[i]
  c[i] = b[i-1] + b[i] *x[i]
```

Fig. 8. Example where the current formulation of linearity analysis combined with data dependence analysis is overly conservative.

```
a = 0.0
for i = 1, N
  a += x[i]*x[i]
  a_t = a_f
  a_a = cos(a)*a_t
  a_x[i] = 2*x[i]*a_a
end
```

Fig. 9. Reverse mode AD, computing only derivatives via predictive slicing.

a loop-carried dependence, and the variable depends on itself nonlinearly based on the results of linearity analysis, then the loop may involve a nonlinear loop carried dependence.

2.2 Limitations

As formulated, linearity analysis is incapable of determining that the loop shown in Figure 8 has no loop-carried, nonlinear dependences. Specifically, there is a loop-carried dependence between b and c due to `c[i] = b[i-1]`, and there is a nonlinear dependence between `b` and `c` due to `c[i] = b[i]*x[i]`. However, the nonlinear dependence is not loop carried.

One can prove that there are no nonlinear, loop-carried dependences if the data-flow analysis is done on a use-by-use basis. This approach could be much more expensive than basic linearity analysis. In order to achieve higher precision at a reasonable cost, while reusing as much analysis as possible, a closer coupling between linearity analysis and data-dependence analysis may be required.

3 Other Applications

Linearity analysis is also useful for a sort of "predictive slicing." In a so-called "pure" derivative computation [8], one wants to compute only the derivatives of a function and not the function itself. However, by default AD produces code that computes both the function and its derivatives. A primary reason is that many of the intermediate function values are required to compute derivatives. However, when it can be determined that the dependent variables depend only linearly on an intermediate function value, then that intermediate value is not needed in the derivative computation. Therefore, the generated derivative code may omit the computation of these intermediates. This is equivalent to generating the derivative code, then performing a backward slice [16] from the derivative variables. Figure 9 illustrates the use of predictive slicing on the example of Figure 1. The dependent variables f and g depend nonlinearly only on a and x; therefore, b, c, f, and g don't need to be computed and even the value of independent variable y is not needed.

Linearity analysis can be combined with array data flow analysis to identify functions $f(x) : R^n \mapsto R$ that can be decomposed into the form: $f(x) =$

$\sum_{i=1}^{m} F_i(x)$ where each F_i is a function of only a few elements of the vector x. Such a function is said to be partially separable. The Jacobian of $F(x) : R^n \mapsto R^m$ is sparse and this sparsity can be exploited using compression techniques [1]. The gradient of f is the sum of the rows of this Jacobian. Thus, gradients of partially separable functions can be computed efficiently using the forward mode.

Linearity analysis is also directly useful in numerical optimization. Optimization algorithms distinguish between linear and nonlinear constraints in order to reduce the cost of derivative evaluations (the derivatives of linear constraints are constant), to reduce the problem size via preprocessing, and to improve the performance of the optimization algorithm. Experimental results from Gould and Toint [5] indicate that preprocessing of the linear and bound constraints reduces the number of constraints by 19% and the total time to solution by 11% on average. Combined with the added savings from fewer constraint evaluations, derivative evaluations, and faster convergence, the savings can be substantial. Preliminary experiments indicate that when all constraints can be identified as linear, savings of 50% or more are possible.

4 Related Work

Karr [9] and Cousot [3] determine linear equalities and linear inequalities between variables. The focus for such techniques is to find program invariants for use with automated reasoning tools. More recent research [12, 14] discovers a subset of nonlinear relationships, polynomial relationships of bounded degree. None of these techniques distinguishes between a nonlinear dependence and a lack of dependence. Therefore, they are not suitable for the types of program optimization we have described. To-be-recorded (TBR) analysis [7] identifies the set of variables that are needed for derivative computation and thus must be recorded if overwritten. This analysis is similar to linearity analysis, but includes index variables, excludes variables that are never overwritten, and does not identify pairwise dependence. Linearity analysis can be readily extended to polynomial degree analysis. We have also extended polynomial degree analysis to a restricted form of rationality analysis. Polynomial degree analysis and rationality analysis have applications in code validation [11].

5 Conclusions and Future Work

We have presented a formal data-flow formulation for linearity analysis. Linearity analysis has several applications in automatic differentiation and numerical optimization. In addition to the applications already discussed, linearity and polynomial degree analysis have applications in code derivative-free optimization, nonlinear partial differential equations, and uncertainty quantification. We are implementing linearity and polynomial degree analysis in the OpenAnalysis framework [15] to provide compiler infrastructure-independent analysis. We are investigating ways to tightly couple linearity analysis with dependence analysis to address the limitations discussed in Section 2.2.

References

1. B. M. Averick, J. J. Moré, C. H. Bischof, A. Carle, and A. Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput.*, 15(2):285–294, 1994.
2. U. Banerjee. *Dependence analysis for supercomputing.* The Kluwer international series in engineering and computer science. Parallel processing and fifth generation computing. Kluwer Academic, Boston, MA, USA, 1988.
3. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1978. ACM Press.
4. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), February 1991.
5. N. Gould and P. L. Toint. Preprocessing for quadratic programming. *Math. Programming*, 100(1):95–132, 2004.
6. L. Hascoët, S. Fidanova, and C. Held. Adjoining independent computations. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 35, pages 299–304. Springer, New York, NY, 2001.
7. L. Hascoët, U. Naumann, and V. Pascual. "To be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2005.
8. T. Kaminski, R. Giering, and M. Voßbeck. Efficient sensitivities for the spin-up phase. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005.
9. M. Karr. Affine relationships among variables of a proram. *Acta Informatica*, 6(2):133–151, 1976.
10. G. A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM, ACM, October 1973.
11. R. Kirby and R. Scott. Personal communication, 2004.
12. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 330–341, New York, NY, USA, 2004. ACM Press.
13. W. Pugh. Omega test: A practical algorithm for exact array dependency analysis. *Comm. of the ACM*, 35(8):102, 1992.
14. E. Rodriguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 266–273, New York, NY, USA, 2004. ACM Press.
15. M. M. Strout, J. Mellor-Crummey, and P. Hovland. Representation-independent program analysis. In *Proceedings of the Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2005.
16. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
17. M. Wolfe and C. W. Tseng. The power test for data dependence. *IEEE Trans. Parallel Distrib. Syst.*, 3(5):591–601, 1992.