CHAPTER 3

Dependence Testing

3.1 Introduction

Calculating data dependence for arrays is complex because arrays contain multiple elements; determining whether two array references access the same memory location is not an easy task. *Dependence testing* is the method used to determine whether dependences exist between two subscripted references to the same array in a loop nest. For the purposes of this explication, control flow (other than the loops themselves) will be ignored. Dependence testing in its full generality is illustrated by the problem of determining whether a dependence exists from statement S_1 to statement S_2 in the following model loop nest:

```
    DO i1 = L1, U1
       DO i2 = L2, U2
          ...
          DO in = Ln, Un
S1           A(f1(i1,...,in),...,fm(i1,...,in)) = ...
S2           ... = A(g1(i1,...,in),...,gm(i1,...,in))
          ENDDO
```

```
        . . .
      ENDDO
    ENDDO
```

Let α and β be iteration vectors within the iteration space of the loop nest described above. Thus, α and β are vectors of length `n`, the kth entry of which is an integer value between the lower and upper bounds of the kth loop in the nest. Recall from Definition 2.1 that two requirements must be met for a data dependence to exist between two statements: the two statements must both access some memory location, and there must exist a feasible control flow path between the identical accesses. Expressed in terms of this example, this definition says that a dependence exists from S_1 to S_2 if and only if there exist values of α and β such that α is lexicographically less than or equal to β (the control flow requirement) that satisfies the following system of *dependence equations* (the common access requirement):

$$f_i(\alpha) = g_i(\beta) \text{ for all } i,\ 1 \le i \le m \qquad \textbf{(EQ 3.1)}$$

Equation 3.1 is based on the simple observation that two array accesses are to the same memory location if and only if each corresponding subscript entry is identical.[1] Otherwise the two references are independent.

Dependence testing has two goals. The first goal (and the most desired result) is to prove that no dependence exists between given pairs of subscripted references to the same array variable. The mechanism used to achieve this goal is to show that Equation 3.1 has no solutions in the region of appropriate α and β. When it cannot achieve this goal, dependence testing attempts to characterize the possible dependences in some manner, usually as a minimal complete set of distance and direction vectors. Throughout all of this, testing must be conservative; that is, it must assume the existence of any possible dependence whose existence it cannot explicitly disprove.

The remainder of this chapter discusses methods for determining solutions (or proving the absence of solutions) to Equation 3.1 within the control flow restrictions imposed by appropriate α and β.

1. This statement is not true if a program makes an array access outside of stated array bounds. Such accesses are illegal in FORTRAN; this book assumes only legal array references.

3.2 Notation

Before developing methods for determining the solutions or absence of solutions to Equation 3.1, a small amount of notation is necessary. Distance vectors, direction vectors, and their application to data dependence have already been introduced in Chapter 2. That knowledge is assumed throughout this chapter. The goal of dependence testing is to construct the complete set of distance and direction vectors representing potential dependences between an arbitrary pair of subscripted references to the same array variable. Since distance vectors may be treated as precise direction vectors, only direction vectors will be used throughout the remainder of this chapter.

3.2.1 Indices and Subscripts

Probably want to put a reference to the appropriate chapter in here for this as well.

Normalization?

For convenience, the term *index* will be used to mean the index variable for some loop surrounding both of the references. Furthermore, it will be assumed that all *auxiliary induction variables* (see Chapter 4 for more details) have been detected and replaced by linear functions of the loop indices \cite{Allen:Unify, AlKe:PFC84, AZ90, Wolfe:Masters}. *Subscript* will be used to refer to one of the subscripted positions in a pair of array references. Dependence tests always consider a pair of array references, but for brevity a subscript pair will always be referred to as a subscript. For example, in the pair of references to array `A` in the following loop nest,

```
     DO i
        DO j
           DO k
S1            A(i, j) = A(i, k) + C
           ENDDO
        ENDDO
     ENDDO
```

index `i` occurs in the first subscript and indices `j` and `k` occur in the second subscript.

For the sake of simplicity, it will be assumed that all loops have a step of 1. Non-unit step values may be normalized on the fly as needed via methods discussed in Chapter 4.

3.2.2 Exact Tests

In its full generality, dependence testing is obviously an undecidable problem. Subscript values can be arbitrary expressions whose values are not known until runtime, making compile-time determination of dependence impossible. While some instances of this occur in real programs (necessitating the conservative assumption that all possible dependences exist), most subscripts are simpler—generally polynomials of induction variables—and thus are subject to some form of compile-time analysis. Even general polynomials are too complex to permit solution of Equation 3.1 using current mathematical theory, however, necessitating one more simplification. Unless otherwise stated, it will be assumed that array subscripts are linear expressions of the loop index variables. That is, all subscript expressions are of the form:

$$a_1 i_1 + a_2 i_2 + ... + a_n i_n + e \qquad \text{(EQ 3.2)}$$

where i_k is the index for the loop at nesting level k; all a_k, $1 \le k \le n$, are integer constants; and e is an expression possibly containing loop-invariant symbolic expressions.This restriction is not onerous in practice, as most programs meet these constraints. With this restriction, dependence testing is equivalent to the problem of finding integer solutions to systems of linear Diophantine equations, an NP-complete problem \cite{Cook71,Dantzig63}. Solving this problem exactly is obviously very difficult; as a result, most dependence tests seek efficient approximate solutions. In general, the most common approximation is inexactness; *inexact* tests can prove many no solution exists to the dependence equation, but cannot prove the existence. In other words, if an inexact test determines that no dependence exists, the references are independent, but the references may also be independent when an inexact test determines that dependence exists. *Exact* tests, on the other hand, are dependence tests that will detect dependences if and only if they exist.

3.2.3 Complexity

Complexity refers to the number of indices appearing within a subscript—the more distinct loop indices that appear within a subscript position, the more complex dependence testing becomes. A subscript is said to be ZIV (zero index variable) if the subscript position contains no index in either reference. A subscript is said to be SIV (single index variable) if only one index occurs in that position. Any subscript with more than one index is said to be MIV (multiple index variable). For instance, consider the following loop:

```
      DO i
         DO j
            DO k
S1             A(5, i+1, j) = A(N, i, k) + C
            ENDDO
         ENDDO
      ENDDO
```

When testing for a true dependence between the two references to A in the sample fragment, the first subscript is ZIV, the second is SIV, and the third is MIV. Remember again the convention that "subscript" in this context means a pair of corresponding subscripts.

3.2.4 Separability

Separability describes whether a given subscript interacts with other subscripts for the purpose of dependence testing. When testing multidimensional arrays, a subscript position is said to be *separable* if its indices do not occur in the other subscripts \cite{Allen:Dissertation,Call:Weak}. If two different subscripts contain the same index, they are *coupled* \cite{-LYZ:Lambda}. For example, in the loop below,

```
      DO i
         DO j
            DO k
S1             A(i, j, j) = A(i, j, k) + C
            ENDDO
         ENDDO
      ENDDO
```

the first subscript is separable, but the second and third are coupled because they both contain the index `j`. ZIV subscripts are vacuously separable because they contain no indices.

Separability is important because multidimensional array references can cause imprecision in dependence testing. One suggested approach, called subscript-by-subscript testing, is to test each subscript separately and intersect the resulting sets of direction vectors \cite{Wolfe:Dissertation}. This method permits simple and efficient testing, but in some cases also provides a conservative approximation to the set of directions within a coupled group. That is, it may yield direction vectors that do not exist. For instance, consider the following loop:

```
      DO i
```

```
S1      A(i+1, i+2) = A(i, i) + C
    ENDDO
```

A subscript-by-subscript test would yield the single direction vector (<,<). However, a careful examination of the statement reveals that this direction vector is invalid (albeit overly conservative), because no dependence exists. Both subscript pairs cannot be simultaneously equal.

On the other hand, if all subscripts are separable, the direction vector for each subscript may be computed independently. These vectors can be merged on a positional basis will full precision. For example, in the loop nest below,

```
    DO i
       DO j
          DO k
S1            A(i+1, j, k-1) = A(i, j, k) + C
          ENDDO
       ENDO
    ENDDO
```

the leftmost direction in the direction vector is determined by testing the first subscript, the middle direction by testing the second subscript and the rightmost direction by testing the third subscript. The resulting direction vector, (<,=,>), is precise. The same approach applied to distances allow us to calculate the exact distance vector (1,0,-1).

The term “separable” derives from linear algebra and differential equations, and applies to systems of equations with distinct variables that may be solved independently. The independent solutions can then be merged to form an exact solution set, a property which also applies to separable subscripts.

3.3 Partitioning Subscripts

Given the differing levels of complexity involved in testing various subscript classifications introduced in the previous section, a natural first step is to partition subscripts according to their complexity, and test accordingly. Doing so leads to a partition-based dependence testing algorithm for a pair of array references:

1. Partition the subscripts into separable and minimal coupled groups.
2. Label each subscript as ZIV, SIV or MIV.

3. For each separable subscript, apply the appropriate single subscript test (ZIV, SIV, MIV) based on the complexity of the subscript. This will produce independence or direction vectors for the indices occurring in that subscript.
4. For each coupled group, apply a multiple subscript test to produce a set of direction vectors for the indices occurring within that group.
5. If any test yields independence, no dependences exist.
6. Otherwise merge all the direction vectors computed in the previous steps into a single set of direction vectors for the two references.

This algorithm takes advantage of separability by classifying all subscripts in a pair of array references either as separable or as part of some minimal coupled group. A coupled group is *minimal* if it cannot be partitioned into two non-empty subgroups with distinct sets of indices. Once a partition is achieved, each separable subscript and each coupled group have completely disjoint sets of indices. Each partition may then be tested in isolation and the resulting distance or direction vectors merged with no loss of precision.

The first step, that of partitioning subscripts, is detailed in the algorithm presented in Figure 3.1. *Partition* initially places each subscript pair in its own partition, then merges together partitions that refer to the same loop variable. While the algorithm may initially appear to be of time complexity $O(n^2)$, it is actually linear when a data structure that permits constant time unioning is used for partitions.

FIGURE 3.1 Subscript Partitioning Algorithm

```
procedure partition(S,P)

    /* S is a set of m subscript pairs S1, S2, ...Sm each enclosed in n */
    /* loops with indices I1, I2, ... In, which is to be partitioned into*/
    /* separable or minimal coupled groups. */
    /* P is an output variable, containing the set of partitions P1, P2, */
    /* ...Pn', n' ≤ n, of separable or minimal coupled groups. */

    for i =1 to n do
        Pi ← {Si}
    od

    for i =1 to n do
        k ← <none>
        for each remaining partition Pj do
```

```
            if there exists s ε P_j such that s contains I_i then
                if k = < none > then
                    k ← j
                else
                    P_k ← P_k ∪ P_j
                    discard P_j
                fi
            fi
        od
    od
end
```

The merge operation described in the test algorithm merits more explanation. Since each separable and coupled subscript group contains a unique subset of indices, merge may be thought of as Cartesian product. In this loop nest,

```
    DO i
       DO j
S1        A(i+1, j) = A(i, j) + C
       ENDDO
    ENDDO
```

the first position yields the direction vector (<) for the `i` loop, while the second position yields the direction vector (=) for the `j` loop. The resulting Cartesian product is the single vector (<,=). A more complex example is shown below:

```
    DO i
       DO j
S1        A(i+1, 5) = A(i, N) + C
       ENDDO
    ENDDO
```

The first subscript yields the direction vector (<) for the `i` loop. Since there is no variance on the `j` loop (`j` does not appear in any subscript and `N` does not indirectly vary with `j`), both references to `A` access the same memory location throughout each iteration of the `j` loop. Thus, if they access a common memory element, all patterns of access to that element occur. In other words, the full set of direction vectors {(<),(=),(>)} must be assumed for the j-loop. The merge yields the following set of direction vectors: {(<,<), (<,=), (<,>)}

Dependence test results for ZIV subscripts are treated specially. If a ZIV subscript proves independence, the dependence test algorithm halts immediately. If independence is not proved, the ZIV test does not produce direction vectors, so no merge is necessary.

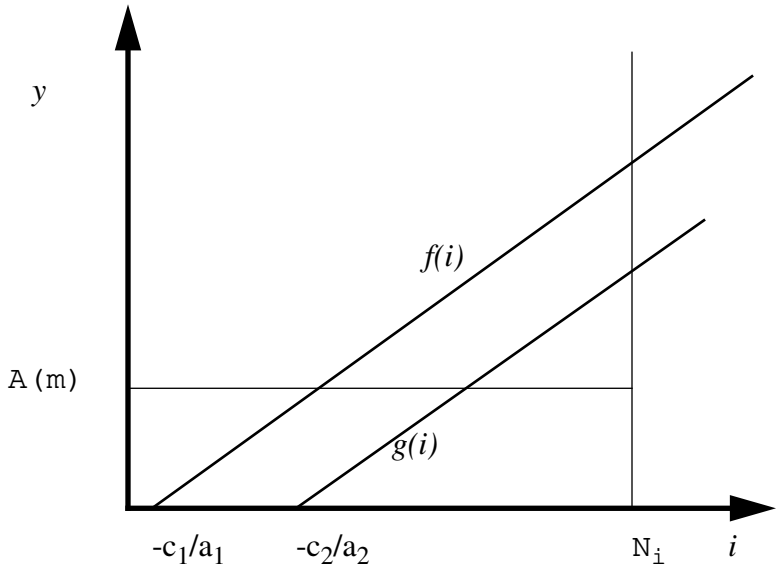
3.4 ZIV and SIV Tests

Once subscripts have been partitioned and classified, it is time to apply specific tests to determine whether dependences exist and to characterize those that do. This section develops the tests that can be applied to individual ZIV and SIV subscripts. Later sections will develop tests for MIV subscripts.

3.4.1 ZIV Test

Since ZIV subscripts contain no references to any loop induction variables, they do not vary within any loop (assuming all symbolic terms are loop invariant). Thus, if the two expressions can be proved not to be equal, then the corresponding array references are independent. If the expressions cannot be shown to be different, then the subscript does not contribute to any direction vector (since it contains no loop induction variable), and may be ignored. The ZIV test can be easily extended for symbolic expressions by forming the expression representing the difference between the two subscript expressions. If the difference simplifies to a non-zero constant, the subscripts are independent.

FIGURE 3.2 Geometric View of Strong SIV Tests



3.4.2 SIV Tests

The next step up in complexity from ZIV subscripts is obviously SIV subscripts. SIV subscripts are the most commonly occurring in practice, and a number of authors, notably Banerjee, Cohagan, and Wolfe \cite{-Bane:88,Cohagan73,Wolfe:Book}, have published a Single-Index exact test for linear SIV subscripts. These methods are based on finding all solutions to a simple Diophantine equation in two variables. The methods presented in this section are somewhat simpler than the exact test listed above. The methods gain their simplicity by separating SIV subscripts into two categories: strong SIV and weak SIV.

3.4.2.1 Strong SIV Subscripts

An SIV subscript for index *i* is said to be *strong* if it has the form <$ai + c_1$, $ai' + c_2$>; *i.e.*, if it is linear and the coefficients of the two occurrences of the index *i* are constant and equal. Figure 3.2 gives a geometric picture of the equations corresponding to a strong SIV pair. Since the loop coefficients are identical for each reference, a strong SIV pair maps into a pair of parallel lines. The horizontal line marked `A(m)` indicates one point where the two subscripts access the same element. Because of the parallel nature of the lines, accesses to common elements will always be separated by the same distance in terms of loop iterations. This distance, called the dependence *distance* can be calculated by the following:

$$d = i' - i = \frac{c_1 - c_2}{a} \qquad \text{(EQ 3.3)}$$

A dependence exists between two references only if access to common elements occur within the bounds set by the loop (given SIV, there must be only one loop involved in the two references). This access will occur only when d is an integer (if it is not an integer, references to the same element cannot occur on a loop iteration for both subscripts) and $|d| \leq U-L$, where U and L are the loop upper and lower bounds (otherwise, at most one reference can occur within the iteration space of the loop). These facts provide for a very simple dependence test: calculate d; if it is an integer and its absolute value falls within the range of the loop, there exists a dependence whose direction can be determined by the sign of d.[1] Otherwise, no dependence exists.

This is not the right diagram. I need to either think it out or dig it out of the Toronto proceedings.

One advantage of the strong SIV test is that it can be easily extended to handle loop-invariant symbolic expressions. The extension is to first evaluate the dependence distance d symbolically. If the result is a constant, then the test may be performed as above. Otherwise, the difference between the loop bounds can be calculated and also compared with d symbolically. The following loop is an example of code that is not atypical of many programs:

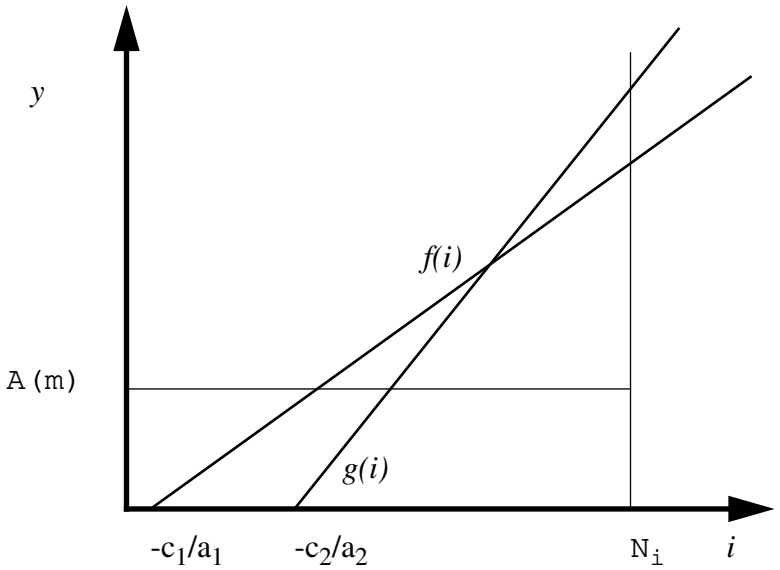
```
    DO i = 1, N
S1     A(i+2N) = A(i+N) + C
    ENDDO
```

The strong SIV test can evaluate the dependence distance d as 2N−N, which simplifies to N. This is compared with the loop bounds symbolically, proving independence since N > N−1.

1. One notable exception here is a distance of 0, which can only give rise to a loop independent dependence. A zero distance between references within a single statement does not indicate a dependence.

3.4.2.2 Weak SIV Subscripts

FIGURE 3.3 Geometric View of Weak SIV Subscripts

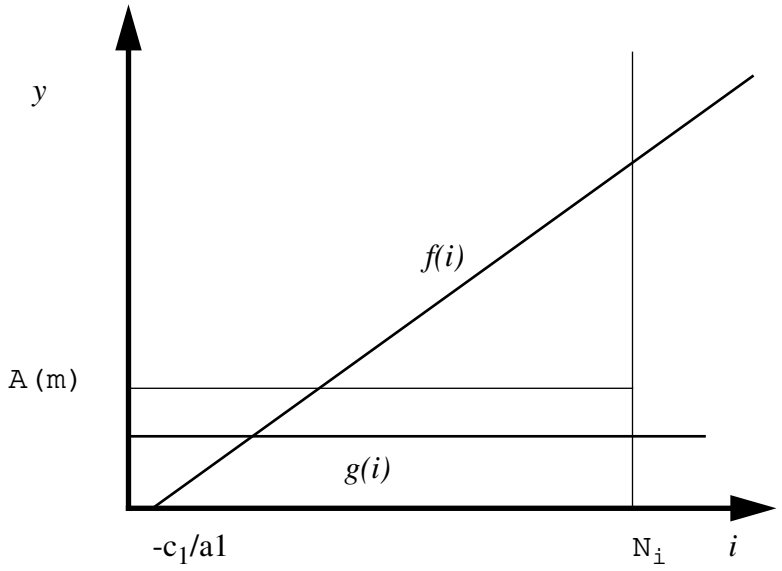


In contrast to a strong SIV subscript, a weak SIV subscript has differing coefficients on the loop induction variable, and thus takes the form $<a_1 i + c_1, a_2 i' + c_2>$. As stated previously, weak SIV subscripts may be solved using the Single-Index exact test. However, it also may be helpful to view the problem geometrically, where the dependence equation:

$$a_1 i + c_1 = a_2 i' + c_2 \qquad \textbf{(EQ 3.4)}$$

describes a line in the two dimensional plane with i and i' as the axes \cite{Call:Weak}. The weak SIV test can then be formulated as determining whether the line derived from the dependence equation intersects with any integer points in the space bounded by the loop upper and lower bounds, as shown in Figure 3.3. There are two special cases that are advantageous to identify.

FIGURE 3.4 Geometric View of Weak-zero SIV Subscripts



3.4.2.3 Weak-zero SIV Subscripts

If one of the two coefficients is 0 (i.e., $a_1 = 0$ or $a_2 = 0$), the subscript is a *weak-zero SIV* subscript. If a_2 is equal to zero, the dependence equation reduces to

$$i = \frac{c_2 - c_1}{a_1} \qquad \text{(EQ 3.5)}$$

The reference in which the coefficient is zero references only one array element during the loop (the horizontal line in Figure 3.4). Given that the other coefficient is nonzero, the two lines can intersect at only one point—that defined in Equation 3.5. Thus, the test consists merely of checking that the computed value is an integer and is within the loop bounds. A symmetric test applies when a_1 is zero.

We need to add a description of the Single-Index exact test somewhere here.

The weak-zero SIV test finds dependences caused by a particular iteration. In scientific codes, this iteration is usually the first or last of the loop, eliminating one possible direction vector for the dependence. More importantly, weak-zero dependences caused by the first or last loop iteration may be eliminated by loop peeling. Consider, for instance, the following simplified loop in the program `tomcatv` from the SPEC benchmark suite:

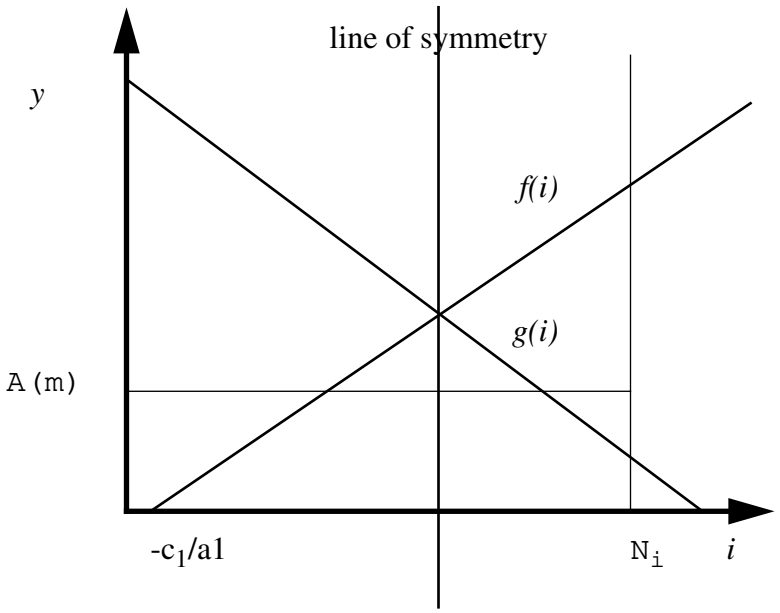
```
     DO i = 1, N
S1      Y(i, N) = Y(1, N) + Y(N, N)
     ENDDO
```

The weak-zero SIV test can determine that the use of `Y(1,N)` causes a loop-carried true dependence from the first iteration to all other iterations. Similarly, with aid from symbolic analysis, the weak-zero SIV test can discover that the use of `Y(N,N)` causes a loop-carried antidependence from all iterations to the last iteration. By identifying the first and last iterations as the only cause of dependences, the weak-zero SIV test can advise the user or compiler to peel the first and last iterations of the loop, resulting in the following parallel loop:

```
     Y(1, N) = Y(1, N) + Y(N, N)
     DO i = 2, N-1
S1      Y(i, N) = Y(1, N) + Y(N, N)
     ENDDO
     Y(N, N) = Y(1, N) + Y(N, N)
```

3.4.2.4 Weak-crossing SIV Subscripts

FIGURE 3.5 Geometric View of Weak-crossing SIV Subscripts



Subscripts in which $a_2 = - a_1$ are called *weak-crossing SIV* . In these cases, the important property that simplifies analysis is symmetry. Given that the absolute values of the slopes of the lines are the same, they

"move" away from a given point at exactly the same rate, although one moves up and the other moves down. As a result, the lines are symmetric about a vertical line through their point of intersection (see Figure 3.5). Because of this symmetry, the ends of all dependences are on opposite sides of a crossing point—the point where the lines intersect. One common example where weak-crossing SIV subscripts occur is in Cholesky decomposition. To locate the crossing point, i is set to i' and the substitution of $a_2 = - a_1$ is made, deriving:

$$i = \frac{c_2 - c_1}{2a_1} \qquad \text{(EQ 3.6)}$$

Since all dependence span the crossing point, determining whether dependences exist is a simple check that the resulting i is within the loop bounds and is either integer or has a non-integer part equal to 1/2. The second part of the condition arises because each line moves symmetrically from the crossing point; if that point is not halfway between two integers, then the lines cannot intersect integers on the y-axis at integral points on the x-axis.

Weak-crossing SIV dependences may be eliminated by loop splitting. To illustrate, consider the following loop from the Callahan-Dongarra-Levine vector test suite \cite{CaDL:Test}:

```
    DO i = 1, N
S1     A(i) = A(N-i+1) + C
    ENDDO
```

The weak-crossing SIV test determines that dependences exist between the definition and use of A, and that they all cross iteration (N+1)/2. Splitting the loop at that iteration results in two parallel loops:

```
    DO i = 1,(N+1)/2
       A(i) = A(N-i+1) + C
    ENDDO
    DO i = (N+1)/2 + 1, N
       A(i) = A(N-i+1) + C
    ENDDO
```

Need to describe the Single-Index Exact test somewhere here.

Both forms of weak SIV tests are also useful for testing coupled subscripts, described in Section 3.7. The Single-Index exact test handles the general case.

3.4.3 Complex Iteration Spaces

SIV tests as described so far have been applied to rectangular iteration spaces, where all loop bounds are independent of the loop values proper. They do not apply directly to triangular loops (where one of the loop bounds is a function of another loop) or trapezoidal loops (where both bounds are functions of another loop). However, they can be extended to apply to such loops, although losing the exactness property in the process. The overall idea is to "rectangularize" the iteration space, by pulling out the non-rectangular corners. This is done by using the maximum and minimum values that the loop bounds can take on as the limits within the dependence tests. Starting with the outermost loop in a nest and working inward, each index that occurs within a loop upper bound expression is replaced with its maximum value (or minimum value, if the term involving the index is negative). This yields the maximum value for this loop index. Similarly, doing the converse in the lower bound yields the minimum value for the loop index. When the process has moved through the innermost loop, the maximum and minimum values for all loop indices are available. These values can be straightforwardly plugged into any of the SIV tests presented so far.

3.4.4 An Exact SIV Test

An SIV subscript position in which the subscripts are of the form

$$a_1 i + a_0 \text{ and } b_1 i + b_0$$

can be tested exactly by constructing all solutions of the linear diophantine equation

$$a_1 x - b_1 y = b_0 - a_0 \qquad \textbf{(EQ 3.7)}$$

As pointed out earlier, this equation system has a solution if and only if the greatest common denominator of a_1 and b_1 divides $b_0 - a_0$.. It is well known that the Euclidean algorithm for computing the greatest common denominator can be extended to compute values n_x and n_y such that

$$n_x x + n_y y = \gcd(x,y) \qquad \textbf{(EQ 3.8)}$$

This gcd algorithm is given in Figure 3.6. It returns a triplet of the form (gcd, n_x, n_y).

FIGURE 3.6 Extended Greatest Common Divisor Routine

```
function gcd(x.,y);
    /* Computes greatest common divisor of x and y as well */
    /* as values n_x and n_y such that n_x x + n_y y = gcd(x,y) */

    /* Adjustment to handle negative values for x and y */
    if x < 0
        then a_x := –1;    s := –x
        else a_x := 1;     s := x
    fi
    if y < 0
        then a_y := –1;    t := –y
        else a_y := 1;     t:= y
    fi;

    if s < t
        then (r, r_x,r_y) := (s, 1, 0);    (s, s_x, s_y) := (t, 0, 1)
        else (r, r_x, r_y) := (t, 0, 1);   (s, s_x, s_y) := (s, 1, 0)
    fi;

    while r > 0 do
        (t, t_x, t_y) := (s, s_x, s_y)
        (s, s_x, s_y) := (r, r_x, r_y)
        q := floor(t/s)
        (r, r_x, r_y) := (t, t_x, t_y) – q*(s, s_x, s_y)
    od;

    return (s, a_x*s_x, a_y*s_y)
end
```

This algorithm can be used in a general procedure for finding all pairs (x, y) that satisfy the diophantine equation.

Theorem 3.1. Let $(g, n_a, n_b) = \gcd(a_1, -b_1)$. Let x_k and y_k be given by

$$x_k = n_a\left(\frac{b_0 - a_0}{g}\right) + k\frac{b_1}{g}$$
$$y_k = n_b\left(\frac{b_0 - a_0}{g}\right) + k\frac{a_1}{g} \qquad \text{(EQ 3.9)}$$

Then $(x_k,,y_k)$ is a solution of $a_1x - b_1y = b_0 - a_0$ for every integral value of k. Furthermore, for any solution (x, y) there exists a k such that $x = x_k$ and $y = y_k$.

Proof. Substituting x_k and y_k into the equation yields

$$a_1 x_k - b_1 y_k = (a_1 n_a - b_1 n_b)\left(\frac{b_0 - a_0}{g}\right) + k\frac{a_1 b_1 - b_1 a_1}{g} = b_0 - a_0$$

which holds because $a_1 n_a - b_1 n_b = g$ by the properties of the extended gcd algorithm. Thus, (x_k, y_k) is a solution.

Suppose that (x, y) is an arbitrary solution. Let (x_0, y_o) be the value of (x_k, y_k) when k is equal to 0. Since this is also a solution, substituting both solutions into separate equations and subtracting yields

$$a_1(x - x_0) - b_1(y - y_0) = 0$$

which means

$$\frac{a_1}{g}(x - x_0) = \frac{b_1}{g}(y - y_0)$$

But $\gcd(a_1/g, b_1/g) = 1$ so a_1/g must divide $y - y_0$ and b_1/g divides $x - x_0$. Furthermore, the respective quotients are easy. Hence,

$$x = x_0 + k\frac{b_1}{g} = x_k \qquad \text{and} \qquad y = y_0 + k\frac{a_1}{g} = y_k$$

which establishes that all solutions are of the desired form. QED.

Theorem 3.1 leads directly to an algorithm for dependence testing: compute an initial solution to the equation, then vary the parameter k to find solutions in the region specified by the upper and lower bounds. This algorithm is given in Figure 3.7.

FIGURE 3.7

Exact SIV Test

procedure *SIVexact*(*directions*);

/* Output parameter *directions* holds the list of dependence directions */
/* found by the test. If none, it returns the empty set. */

directions := ∅;
(g, n_a, n_b) := gcd(a_1, $-b_1$);

if g does not divide $b_0 - a_0$ **then return** $\emptyset$; **fi;**

$x_0 := n_a * (b_0 - a_0)/g$; $y_0 := n_b * (b_0 - a_0)/g$;

/* Find k such that x is the minimum value greater than L, */
/* the lower bound of the loop. Code is based on Equation 3.8*/
$c_x := b_1/g$; $dir_x := c_x/|c_x|$;
$c_y := a_1/g$; $dir_y := c_y/|c_y|$;

k_1 := ceiling($(L - x_0)/c_x$);
if $dir_x < 0$ **then** $k_1 := k_1 - 1$ **fi**;

$x_L := x_0 + c_x * k_1$; $y_L := y_0 + c_y * k_1$;

if $y_L < L$ **then**
  if dir_x*dir$_y < 0$ **then return** $\emptyset$; **fi**
  k_2 := ceiling($(L - y_0)/c_y$);
  if $dir_y < 0$ **then** $k_2 := k_2 - 1$**fi**;

  $x_L := x_0 + c_x * k_2$; $y_L := y_0 + c_y * k_2$;
fi;

/* Here both x_L and y_L are greater than L */
/* Furthermore, x_L cannot be decreased without leaving the */
/* region of solutions */
if $x_L > U$ **then return** $\emptyset$; **fi**
if $y_L > U$ **then**
  if $dir_x * dir_y > 0$ **then return** $\emptyset$; **fi**
  k_3 := floor($(U - y_0)/c_y$);
  if $dir_y < 0$ **then** $k_3 := k_3 + 1$ **fi**;

  $x_L := x_0 + c_x * k_3$; $y_L := y_0 + c_y * k_3$;
  if $x_L > U$ **then return** $\emptyset$; **fi**
fi:

/* Here (x_L, y_L) is a solution to the equation in range */
/* Furthermore, x_L is the smallest x that is part of a solution */
if $x_L < y_L$
  then *directions* := *directions* $\cup$ {<}
  else if $x_L > y_L$ **then** *directions* := *directions* $\cup$ {>}
  else *directions* := *directions* $\cup$ {=}
fi; **fi**

/* Here we have a solution and we have determined x_L, */
/* the minimum x that is a solution. Now we jump to the */
/* solution closest to $x = y$ */
$\alpha = (y_0 - x_0)/(c_x - c_y)$;

if $dir_x < 0$ **then** k_4 := floor(α) **else** k_4 := ceiling(α) **fi**;
$x_U := x_0 + c_x * k_4$; $y_U := y_0 + c_y * k_4$;
if $x_U \leq x_L$ **or** $x_U > U$ **or** $y_L > U$ **or** $y_L < L$ **then return** *directions*; **fi**

if $x_U = y_U$ **then**
  $directions := directions \cup \{=\}$;
  $k_5 := k_4 + dir_x$;
  $x_U := x_0 + c_x * k_4$; $y_U := y_0 + c_y * k_4$
fi;

if $x_U > U$ **or** $y_L > U$ **or** $y_L < L$ **then return** *directions* **fi**;

if $x_L < y_L$ **then** $directions := directions \cup \{<\}$
else if $x_L > y_L$ **then** $directions := directions \cup \{>\}$
fi;
return *directions*;

end *SIVexact*;

3.5 MIV Tests

When restricted to linear functions of loop induction variables, ZIV and SIV subscripts are relatively simple linear mappings from Z (the set of natural numbers) to Z. MIV subscripts are much more complicated, being more complex linear mappings from Z^m to Z, where m is the number of loop induction variables appearing in the subscript. This added complexity requires more sophisticated mathematics in order to accurately determine dependences. Accordingly, this section reviews the general dependence equations, recast in terms suitable to MIV subscripts, and the appropriate mathematics as a prelude to the MIV tests proper.

3.5.1 Background

Consider a general nest of loops:

```
    DO i1 = L1, U1
       DO i2 = L2, U2
          ...
          DO in = Ln, Un
S1           A(f(i1,...,in)) = ...
S2           ... = A(g(i1,...,in))
           ENDDO
           ...
        ENDDO
     ENDDO
```

Determining whether there is a dependence with direction vector $\mathbf{D} = (D_1, D_2,...,D_n)$ is equivalent to determining whether there exists an integer solution to the equation system

$$f(x_1,x_2,...,x_n) = g(y_1,y_2,...,y_n) \quad \text{(EQ 3.10)}$$

in the space defined by

$$L_i \le x_i, y_i \le N_i, \forall i, 1 \le i \le n \quad \text{(EQ 3.11)}$$

with the additional restriction due to the direction vector:

$$x_i D_i y_i \quad \forall i, 1 \le i \le n. \quad \text{(EQ 3.12)}$$

(Recall that each entry of a direction vector is one of "<", "=", or ">"). Let R denote the region defined by Equation 3.11 and Equation 3.12. Then the equation has a solution if

$$h(x_1,x_2,...,x_n,y_1,y_2,...,y_n) = f(x_1,x_2,...,x_n) - g(y_1,y_2,...,y_n) = 0 \quad \text{(EQ 3.13)}$$

has an integer solution somewhere inside the region R. Since integer solutions are required, this problem lies in the theory of Diophantine equations. As mentioned earlier, exactly solving Diophantine equations in a restricted space is very difficult. As a result, finding simplifications that are not exact but that provide reasonable precision is useful in a compiler. One such simplification is removing the restriction that the solutions be integer, making the solution space continuous rather than discrete. In other words, searching for real solutions (or, more precisely, the absence of real solutions) to Equation 3.13 in the region R is useful. If the equation does not have a real solution, then it cannot have an integer solution, and accordingly, no dependence can exist. Assume for the moment that the functions f and g are continuous in the region R. The following theorem follows directly from elementary analysis:

Theorem 3.2. If f and g are continuous functions on region R, there exists a real solution to equation Equation 3.13 if and only if

$$\min_R h \le 0 \le \max_R h \quad \text{(EQ 3.14)}$$

Proof. Immediate from the Intermediate Value Theorem. QED.

This theorem is the basis for the Banerjee Inequality and several other tests.

The remainder of this chapter assumes that both f and g are affine functions; that is, they have the form

$$f(x_1,x_2,...,x_n) = a_0 + a_1x_1 + ... + a_nx_n \quad \text{(EQ 3.15)}$$

$$g(y_1,y_2,...,y_n) = b_0 + b_1y_1 + ... + b_ny_n \quad \text{(EQ 3.16)}$$

and that the dependence problem to be solved is to find solutions to the linear diophantine equation

$$a_0 - b_0 + a_1x_1 - b_1y_1 + ... + a_nx_n - b_ny_n = 0 \quad \text{(EQ 3.17)}$$

in the region R.

3.5.2 GCD Test

Rearranging the terms of Equation 3.17 yields the following

$$a_1x_1 - b_1y_1 + ... + a_nx_n - b_ny_n = b_0 - a_0 \quad \text{(EQ 3.18)}$$

which is the standard form for a linear diophantine equation. Linear diophantine equations have been the subject of extensive study for hundreds of years. A fundamental theorem about these equations is the following:

> **Theorem 3.3. GCD Test.** Equation 3.18 has a solution if and only if gcd(a_1,...,a_n,b_1,...,b_n) divides $b_0 - a_0$.

Thus, if the gcd of all the coefficients of loop induction variables does not divide the difference of the two constant additive terms, there can be no solution to the equation anywhere—hence, no dependence can exist. On the other hand, if the gcd of the coefficients does divide $b_0 - a_0$, there is a solution somewhere, although it need not be in the region of interest R.

When testing for a specific direction vector **D =** (D_1,...,D_n), some of whose directions are "=", this condition can be tightened. Assume that the direction vector for the dependence being tested is **D**, and has only one "=" component, D_i.. Then any acceptable solution must have $x_i = y_i$, making the equation:

$$a_1x_1 - b_1y_1 + ... + (a_i - b_i)x_i + ... + a_nx_n - b_ny_n = b_0 - a_0 \quad \text{(EQ 3.19)}$$

It is obvious that the gcd should include ($a_i - b_i$) and exclude both a_i and b_i in this case, which is slightly more precise. In the general case, a simi-

lar replacement should be performed in any position where the direction vector being tested is "=".

With this observation, it should be clear that the strong SIV test described in Section 3.4.2.1 is a special case of the GCD test in Theorem 3.3.

3.5.3 Banerjee Inequality

Although the GCD test is extremely useful in some cases, it is inadequate as a general dependence test. The reason is the fact that the most common gcd encountered in practice is 1, which divides everything. Furthermore, the GCD test indicates dependence whenever the dependence equation has an integer solution anywhere, not just the region *R*. The equations that derive from common subscripts generally have integer solutions somewhere, even if not in the region of interest. One test that eliminates this problem by considering iteration limits is the Banerjee Inequality.

The Banerjee Inequality follows directly from Theorem 3.2 applied in the case when f and g are affine functions:

$$h(x_1,x_2,...,x_n,y_1,y_2,...,y_n) = a_0 - b_0 + a_1x_1 - b_1y_1 + ... + a_nx_n - b_ny_n \quad \textbf{(EQ 3.20)}$$

Some preliminary definitions and notation are necessary before presenting the test itself.

Definition 3.1. Let $h_i^+ = \max_{Ri} h_i(x_i,y_i)$ and $h_i^- = \min_{Ri} h_i(x_i,y_i)$, where

$$h_i(x_i,y_i) = a_ix_i - b_iy_i \quad \textbf{(EQ 3.21)}$$

the region R_i is defined by

$$L_i \le x_i, y_i \le U_i \text{ and } x_i \, D_i \, y_i \quad \textbf{(EQ 3.22)}$$

where D_i is the direction vector element for the ith position.

In other words, h_i^+ is the maximum value of the function h_i over a region and h_i^- is the minimum value. For these values to be useful, there has to be some way of calculating them. The following definitions help accomplish that.

Definition 3.2. Let a be a real number. The *positive part* of a, denoted a^+, is given by the expression

$$a^+ = \text{if } a \ge 0 \text{ then } a \text{ else } 0. \quad \textbf{(EQ 3.23)}$$

The *negative part* of a, denoted a^-, is given by

$$a^- = \text{if } a \geq 0 \text{ then } 0 \text{ else } -a. \quad \textbf{(EQ 3.24)}$$

Both a^+ and a^- are positive and the following relationship holds

$$a = a^+ - a^- \quad \textbf{(EQ 3.25)}$$

With these definitions, the following lemma can now be proved.

Lemma 3.1. Let t, s, and z denote real numbers. If $0 \leq z \leq s$, then

$$-t^- s \leq tz \leq t^+ s$$

Furthermore, there exist values z_1, z_2 such that

$$tz_1 = -t^- s \text{ and } tz_2 = t^+ s$$

Proof. Case 1: $t \geq 0$. In this case $t^+ = t$ and $t^- = 0$. So the inequality is satisfied because $0 \leq tz = t^+ z \leq t^+ s$, since $0 \leq z \leq s$. In this case $z_1 = 0$ and $z_2 = s$.

Case 2: $t < 0$. In this case $t^+ = 0$ and $t^- = -t$. Thus, the inequality becomes $-t^- s \leq -t^- z = tz \leq 0$ and we have $z_1 = s$ and $z_2 = 0$. QED.

The next step is to generalize this inequality to arbitrary bounds.

Lemma 3.2. Let t, l, u and z denote real numbers. If

$$l \leq z \leq u$$

then

$$-t^- u + t^+ l \leq tz \leq t^+ u - t^- l$$

and there exist values z_1, z_2 such that

$$tz_1 = -t^- u + t^+ l \text{ and } tz_2 = t^+ u - t^- l.$$

Proof. Assume $l \leq z \leq u$. Rearranging terms gives $0 \leq z - l \leq u - l$. Lemma 3.1 then gives us:

$$-t^-(u - l) \leq t(z - l) \leq t^+(u - l)$$

Rearranging terms again yields:

$$-t^- u + (t + t^-)l \leq tz \leq t^+ u - (t^+ - t)l$$

Rearranging Equation 3.25 gives us

$$t^+ = t + t^- \text{ and } t^- = t^+ - t$$

which proves the inequality. The existence of z_1 and z_2 follows directly from Lemma 3.1. QED.

This inequality provides the basis for proving an important result about the minimum and maximum values of h.. A little more terminology is necessary first.

Definition 3.3. The quantities $H_i^-(D)$ and $H_i^+(D)$, where D is a direction, are defined as follows:

$$H_i^-(<) = -(a_i^- + b_i)^+(U_i - 1) + [(a_i^- + b_i)^- + a_i^+]L_i - b_i$$

$$H_i^+(<) = (a_i^+ - b_i)^+(U_i - 1) - [(a_i^+ - b_i)^- + a_i^-]L_i - b_i$$

$$H_i^-(=) = -(a_i - b_i)^-U_i + (a_i - b_i)^+L_i$$

$$H_i^+(=) = (a_i - b_i)^+U_i - (a_i - b_i)^-L_i$$

$$H_i^-(>) = -(a_i - b_i^+)^-(U_i - 1) + [(a_i - b_i^+)^+ + b_i^-]L_i + a_i$$

$$H_i^+(>) = (a_i + b_i^-)^+ (U_i - 1) - [(a_i + b_i^-)^- + b_i^+]L_i + a_i$$

$$H_i^-(*) = -a_i^-U_i^x + a_i^+L_i^x - b_i^+U_i^y + b_i^-L_i^y$$

$$H_i^+(*) = a_i^+U_i^x - a_i^-L_i^x + b_i^-U_i^y - b_i^+L_i^y$$

where L_i^x, U_i^x, L_i^y and U_i^y are used in the last two equations to handle the case where the source and sink of the dependence are in different loops with different upper and lower bounds.

This use of different upper and lower bounds in the definition for direction "*" permits the handling of cases like the following:

```
     DO I = 1, 100
        DO J = 1, 50
S1         A(I,J) = B(I,J) + C
        ENDDO
        DO J = 51, 100
S2         A(I,J) = A(I,J) + D
        ENDDO
     ENDDO
```

Because of the nonintersecting iteration ranges of the first and second loops, S_1 and S_2 are independent. Permitting the upper and lower bounds to be different makes it possible to catch this case using the Banerjee Inequality.

Lemma 3.3. Let $h_i(x_i,y_i) = a_ix_i - b_iy_i$ and let R_i be as described by Equation 3.22: $L_i \le x_i$, $y_i \le U_i$ and $x_i D_i y_i$. Then the minimum and maximum values of h_i are given by:

$$\min_R h = h_i^- = H_i^-(D) \quad \text{(EQ 3.26)}$$

$$\min_R h = h_i^+ = H_i^+(D) \quad \text{(EQ 3.27)}$$

where the values of H are given by Definition 3.3.

Proof. Case 1: D= "=". In this case $L_i \le x_i = y_i \le U_i$. Substituting x_i for y_i. in the equations:

$$L_i \le x_i \le U_i$$

$$h_i = a_ix_i - b_iy_i = (a_i - b_i)x_i.$$

Two applications of Lemma 3.2 yield the desired inequality. Since Lemma 3.2 guarantees that there exist points within R where the left and right hand side values are obtained, the result is established.

Case 2: D= "<". In this case $L_i \le x_i < y_i \le U_i$. To use Lemma 3.2, the inequality needs to be translated to

$$L_i \le x_i \le y_i - 1 \le U_i - 1.$$

Note that h_i, (the quantity to minimized and maximized) is given by

$$h_i = a_ix_i - b_iy_i = a_ix_i - b_i(y_i - 1) - b_i.$$

An application of Lemma 3.1 on the left and on the right can eliminate x_i:

$$-a_i^-(y_i - 1) + a_i^+L_i - b_i(y_i - 1) - b_i \le h_i$$

$$\le a_i^+(y_i - 1) - a_i^-L_i - b_i(y_i - 1) - b_i.$$

Next Lemma 3.2 is applied again to eliminate y_i.

$$-(a_i^- + b_i)^+(U_i - 1) + (a_i^- + b_i)^-L_i + a_i^+L_i - b_i \le h_i$$

$$\le (a_i^+ - b_i)^+(U_i - 1) - (a_i^+ - b_i)^-L_i - a_i^-L_i - b_i$$

which establishes the result for this case. Once again, Lemma 3.1 guarantees that there exist points within R where the minima and maxima occur.

Case 3: D = ">". In this case $L_i \leq y_i < x_i \leq U_i$. First the inequality is translated to

$$L_i \leq y_i \leq x_i - 1 \leq U_i - 1.$$

Rearranging h_i, yields

$$h_i = a_i x_i - b_i y_i = a_i(x_i - 1) - b_i y_i + a_i.$$

Applying Lemma 3.2 eliminates y_i:

$$a_i(x_i - 1) - b_i^+(x_i - 1) + b_i^- L_i + a_i \leq h_i$$

$$\leq a_i(x_i - 1) + b_i^-(x_i - 1) - b_i^+ L_i + a_i.$$

A second application eliminates x_i:

$$-(a_i - b_i^+)^-(U_i - 1) + (a_i - b_i^+)^+ L_i + b_i^- L_i + a_i \leq h_i$$

$$\leq (a_i + b_i^-)^+(U_i - 1) - (a_i + b_i^-)^- L_i - b_i^+ L_i + a_i,$$

which is the desired inequality.

Case 4: D= "*". In this case $L_i^x \leq x_i \leq U_i^x$, $L_i^x \leq y_i \leq U_i^y$, with no implied relationship between x_i and y_i. Four applications of Lemma 3.2, two on the left and two on the right of $h_i = a_i x_i - b_i y_i$, yield the inequality

$$H_i^-(*) \leq h_i \leq H_i^+(*).$$

Since Lemma 3.2 guarantees that there exist points within R where the minima and maxima occur, the result is established for this case.QED.

Since each of the terms in h involves a different induction variable, each term can be maximized and minimized independently. This yields the following important result.

Theorem 3.4. Banerjee's Inequality. There exists a real solution to $h = 0$ (Equation 3.13) for direction vector $\mathbf{D} = (D_1, D_2, ..., D_n)$ if and only if the following inequality is satisfied on both sides:

$$\sum_{i=1}^{n} H_i^-(D_i) \leq b_0 - a_0 \leq \sum_{i=1}^{n} H_i^+(D_i) \qquad \textbf{(EQ 3.28)}$$

Proof. The dependence equation provides the following formula for h:

$$h = a_0 - b_0 + \sum_{i=1}^{n} h_i = a_0 - b_0 + \sum_{i=1}^{n} (a_i x_i - b_i y_i)$$

Lemma 3.3 can then be applied to minimize and maximize h:

$$\min_R h = a_0 - b_0 + \sum_{i=1}^{n} H_i^-(D_i)$$

$$\max_R h = a_0 - b_0 + \sum_{i=1}^{n} H_i^+(D_i)$$

Substituting these formulas into the inequality from Equation 3.14 in Theorem 3.2 and subtracting $a_0 - b_0$ from all three sides of the inequality gives the desired result. QED.

3.5.3.1 Trapezoidal Banerjee Inequality

The Banerjee test presented in Section 3.5.3 assumes that the upper and lower bounds of all the loops in the loop nest are constants, independent of the values of other loop induction variables. Not all loops meet this requirement, however, and it is common in practice to encounter loop nests where the iteration range of the inner loops depends on the values of outer loop indices. The following is an example:

```
    DO I = 1,100
       DO J = 1, I-1
S1        A(J) = A(I+J-1) + C
       ENDDO
    ENDDO
```

In this case, no dependence can be carried by the inner loop because for a given value of `I`, the references on the left hand side fall in the subarray `A(1:I-1)` while the references on the right fall in the subarray `A(I:I*2-1)`. Since these two subarrays are nonintersecting, no dependence exists with direction vector (=,*). Note that

$$a_1 = 0;\ a_2 = 1;\ b_1 = 1;\ b_2 = 1$$

Using these values in the Banerjee Inequality gives

$$H_1^-(=) + H_2^-(*)$$

$$= -(a_1 - b_1)^- U_1 + (a_1 - b_1)^+ L_1 - (a_2^- + b_2^+) U_2 + (a_2^+ + b_2^-) L_2$$

$$= -(0-1)^{-}(100) + (0\text{-}1)^{+}1 - (0+1)(\text{I-}1) + (1+0)1 = -\text{I} - 100$$

$$\leq b_0 - a_0 = 1 - 1 = 0$$

$$\leq H_1^{+}(=) + H_2^{+}(*)$$

$$= (a_1 - b_1)^{+}U_1 - (a_1 - b_1)^{-}L_1 + (a_2^{+} + b_2^{-})U_2 - (a_2^{-} + b_2^{+})L_2$$

$$\leq (0-1)^{+}(100) - (0-1)^{-}1 + (1+0)(\text{I}-1) - (0+1)1 = \text{I}-3,$$

which can certainly be satisfied, since `I` takes on values greater than 3. As a result, a dependence must be assumed. The problem is that the test as currently formulated cannot take advantage of knowledge about values of loop induction variables in the upper bound expression.

To alleviate this deficiency, let us start by assuming that the upper and lower bound expressions can be rewritten as an affine combination of the loop induction variables:

$$J_i = U_{i0} + \sum_{j=1}^{i-1} U_{ij} i_j \qquad L_i = L_{i0} + \sum_{j=1}^{i-1} L_{ij} i \qquad \textbf{(EQ 3.29)}$$

The desired result is a modification of the Banerjee inequality that considers these bound expressions. This result is not straightforward, because the evaluation of minimum and maximum for each loop is no longer independent of the other loops. For example, evaluating the minimum and maximum at the innermost loop level can modify the coefficients associated with outer loops.

To understand how this happens, consider the evaluation of $H_i^{-}(<)$. Minimizing h_i according to Lemma 3.2 yields

$$\begin{aligned} H_i^{-}(<) = &-\left(a_i^{-} + b_i\right)^{+}\left(U_{i0} - 1 + \sum_{j=1}^{i-1} U_{ij} i_j\right) \\ &+\left[\left(a_i^{-} + b_i\right)^{-} + a_i^{+}\right]\left(L_{i0} + \sum_{j=1}^{i-1} L_{ij} i_j\right) - b_i \end{aligned} \qquad \textbf{(EQ 3.30)}$$

where i_j can be either x_j or y_j, so long as it is uniformly one or the other in a single summation. Following the proof in Lemma 3.2, the lower bound of x_i is introduced in the first step to eliminate x_i and produce

$$-a_i^-(y_i - 1) + a_i^+L_i - b_i(y_i - 1) - b_i \le h_i$$

$$\le a_i^+(y_i - 1) - a_i^-L_i - b_i(y_i - 1) - b_i.$$

Thus, the summation with x_j should be used for the lower bound multiplied by a_i^+ in the left hand side above and a_i^- in the right hand side above. When eliminating the remaining terms it is proper to use upper and lower bounds for y_i. Thus the final expression is

$$\begin{aligned} H_i^-(<) \;=\; & -\left(a_i^- + b_i\right)^+\left(U_{i0} - 1 + \sum_{j=1}^{i-1} U_{ij}y_j\right) \\ & +\left(a_i^- + b_i\right)^-\left(L_{i0} + \sum_{j=1}^{i-1} L_{ij}y_j\right) \\ & + a_i^+\left(L_{i0} + \sum_{j=1}^{i-1} L_{ij}x_j\right) - b_i \end{aligned} \qquad \text{(EQ 3.31)}$$

This means that the coefficients of x_j and y_j, for $1 \le j < i$, used in minimizing h_j must be adjusted by the appropriate term in the equation above. A similar adjustment needs to be made to coefficients on the maximum side of the inequality.

Figure 3.8 contains the algorithm for evaluating the Trapezoidal Banerjee Inequality for any direction vector such that all the "*" directions occur at the end of the vector (i.e., no specific direction can occur after the occurrence of a "*" from left to right). Note that it evaluates the inequality starting at the innermost loop and moving outward. At each stage, it updates the accumulated left and right hand sides of the inequality, then it adjusts, as appropriate, the coefficients corresponding to outer loops. These coefficients are held in the temporary arrays A_l[] and B_l[], which hold coefficients for the left hand inequality, and A_r[] and B_r[], which hold coefficients for the right hand inequality.

FIGURE 3.8 Trapezoidal Banerjee Inequality Evaluation

boolean function *Banerjee(D)*;

/* $A_l[i]$ is the adjusted coefficient a_i used on the minimum side */
/* $B_l[i]$ is the adjusted coefficient b_i used on the minimum side */
/* $A_r[i]$ is the adjusted coefficient a_i used on the maximum side */
/* $B_r[i]$ is the adjusted coefficient b_i used on the maximum side */

```
/* V_min is the accumulated Banerjee left hand side                */
/* V_max is the accumulated Banerjee right hand side               */

for i := 1 to n do
    A_l[i] := a_i; A_r[i] := a_i; B_l[i] := b_i; B_r[i] := b_i;
od;

V_min := 0; V_max := 0;

for i := n to 1 do

    if D_i = "=" then
        V_min := V_min – (A_l[i] – B_l[i])⁻U_i0 + (A_l[i] – B_l[i])⁺L_i0;
        V_max := V_max + (A_r[i] – B_r[i])⁺U_i0 – (A_r[i] – B_r[i])⁻L_i0;
        nU := 0; nL := 0;
        /* v(<) = –1, v(=) = 0 and v(>) = 1 */
        for j := 1 to i–1 do
            nL := nL + L_ij v(D_j); nU := nU + U_ij v(D_j)
        od;
        if nL ≤ 0 then
            for j := 1 to i–1 do
                /* Signs reversed for updates to B */
                B_l[j] := B_l[j] – (A_l[i] – B_l[i])⁺L_ij;
                B_r[j] := B_r[j] + (A_r[i] – B_r[i])⁻L_ij
            od
        else
            for j := 1 to i–1 do
                A_l[j] := A_l[j] + (A_l[i] – B_l[i])⁺L_ij
                A_r[j] := A_r[j] – (A_r[i] – B_r[i])⁻L_ij
            od
        fi;
        if nU ≤ 0 then
            for j := 1 to i–1 do
                A_l[j] := A_l[j] – (A_l[i] – B_l[i])⁻U_ij
                A_r[j] := A_r[j] + (A_r[i] – B_r[i])⁺U_ij
            od
        else
            for j := 1 to i–1 do
                /* Signs reversed for updates to B */
                B_l[j] := B_l[j] + (A_l[i] – B_l[i])⁻U_ij;
                B_r[j] := B_r[j] – (A_r[i] – B_r[i])⁺L_ij
            od
        fi;

    else if D_i = "<" then
```

$V_{min} := V_{min} - (A_l[i]^- + B_l[i])^+(U_{i0} - 1)$
$\quad + ((A_l[i]^- + B_l[i])^- + A_l[i]^+)L_{i0} - B_l[i];$
$V_{max} := V_{max} + (A_r[i]^+ - B_r[i])^+(U_{i0} - 1)$
$\quad - ((A_r[i]^+ - B_r[i])^- + A_r[i]^-)L_{i0} - B_r[i];$
for $j := 1$ **to** $i-1$ **do**
/* Signs reversed for updates to B */
$A_l[j] := A_l[j] + A_l[i]^+L_{ij};$
$B_l[j] := B_l[j] + (A_l[i]^- + B_l[i])^+U_{ij} - (A_l[i]^- + B_l[i])^-L_{ij};$
$A_r[j] := A_r[j] - A_r[i]^-L_{ij};$
$B_r[j] := B_r[j] - (A_r[i]^+ - B_r[i])^+U_{ij} + ((A_r[i]^+ - B_r[i])^-L_{ij}$
od

else if $D_i =$ ">" **then**
$V_{min} := V_{min} - (A_l[i] - B_l[i]^+)^-(U_{i0} - 1)$
$\quad + ((A_l[i] - B_l[i]^+)^+ + B_l[i]^-)L_{i0} + A_l[i];$
$V_{max} := V_{max} + (A_r[i] + B_r[i]^-)^+(U_{i0} - 1)$
$\quad - ((A_r[i] + B_r[i]^-)^- + B_r[i]^+)L_{i0} + A_l[i];$
for $j := 1$ **to** $i-1$ **do**
/* Signs reversed for updates to B */
$B_l[j] := B_l[j] - B_l[i]^-L_{ij};$
$A_l[j] := A_l[j] - (A_l[i] - B_l[i]^+)^-U_{ij} + (A_l[i] - B_l[i]^+)^+L_{ij};$
$B_r[j] := B_r[j] + B_r[i]^+L_{ij};$
$A_r[j] := A_r[j] + (A_r[i] + B_r[i]^-)^+U_{ij} - ((A_r[i] + B_r[i]^-)^-L_{ij}$
od

else if $D_i =$ "*" **then**
$V_{min} := V_{min} - A_l[i]^-U_i^x + A_l[i]^+L_i^x - B_l[i]^+U_i^y + B_l[i]^-L_i^y;$
$V_{max} := V_{max} + A_r[i]^+U_i^x - A_r[i]^-L_i^x + B_r[i]^-U_i^y - B_r[i]^+L_i^y$
for $j := 1$ **to** $i-1$ **do**
/* Signs reversed for updates to B */
$A_l[j] := A_l[j] - A_l[i]^-U_{ij} + A_l[i]^+L_{ij};$
$B_l[j] := B_l[j] + B_l[i]^+U_{ij} - B_l[i]^-L_{ij};$
$A_r[j] := A_r[j] + A_r[i]^+U_{ij} - A_r[i]^-L_{ij};$
$B_r[j] := B_r[j] - B_r[i]^-U_{ij} + B_r[i]^+L_{ij}$
od
fi
od;

/* Now we are ready for the test */
if $V_{min} \le b_0 - a_0$ **and** $b_0 - a_0 \le V_{max}$
then return *true*
else return *false*
fi

end *Banerjee*;

The following theorem establishes the correctness of function *Banerjee.*

> **Theorem 3.5. Trapezoidal Banerjee Inequality.** The values of V_{min} and V_{max} computed by function *Banerjee* in Figure 3.8 is such that:
>
> $$V_{min} \leq \min_R h \text{ and } \max_R h \leq V_{max}.$$
>
> Therefore, if *Banerjee* returns *false*, no dependence can exist.

Proof. Correctness for the cases D_i= "<", D_i= ">" and D_i= "*" follow directly from the proof of Theorem 3.4, with the adjustments being made to coefficients for outer loops along the lines described in the discussion of Equation 3.30 and Equation 3.31. In fact, the inequality is precise for those directions by Lemma 3.3.

The only tricky case occurs when D_i= "=". In this case, the following expansions for $H_i^-(=)$ and $H_i^+(=)$ are valid:

$$H_i^-(=) = -(a_i - b_i)^-\left(U_{i0} + \sum_{j=1}^{i-1} U_{ij} i_j\right) + (a_i - b_i)^+\left(L_{i0} + \sum_{j=1}^{i-1} L_{ij} i_j\right)$$
$$H_i^+(=) = (a_i - b_i)^+\left(U_{i0} + \sum_{j=1}^{i-1} U_{ij} i_j\right) - (a_i - b_i)^-\left(L_{i0} + \sum_{j=1}^{i-1} L_{ij} i_j\right) \quad \text{(EQ 3.32)}$$

The problem is that the source and the sink of the dependence can have different upper and lower bounds. Since the requirement is that x_i=y_i, the dependence cannot exist unless x_i is greater than *both* lower bounds and less than *both* upper bounds. Unfortunately, simply substituting the maximum of two summations back into the equation will not yield meaningful coefficients for the outer loops. Hence, one of the lower bounds and one of the upper bounds must be chosen, with a loss of precision.

Choosing the upper and lower bound which preserves as much precision as possible requires an intelligent heuristic. Since the direction vector is known, the following can be evaluated.

$$\iota L = \sum_{j=1}^{i-1} L_{ij} v(D_j) \qquad \text{and} \qquad nU = \sum_{j=1}^{i-1} U_{ij} v(D_j$$

where

$$v(D_i) = \begin{cases} -1 & \text{if } D_i = \text{`<'} \\ 0 & \text{if } D_i = \text{`='} \\ 1 & \text{if } D_i = \text{`>'} \end{cases}$$

If $nL \leq 0$ then it is likely that the lower bound for y_i is greater than the one for x_i, so that lower bound in Equation 3.32 is used. Similarly, if $nU \leq 0$, the upper bound for x_i is likely to be lower than the upper bound for y_i, so the upper bound for x_i is used in Equation 3.32. This heuristic, which is precise if there is only one nonzero L_{ij}, is the source of the somewhat branchy adjustment in the code for *Banerjee*. QED.

Returning to the example, the key loop constants have the following values:

$$a_1 = 0;\ a_2 = 1;\ b_1 = 1;\ b_2 = 1$$

$$N_{10} = 99;\ N_{20} = -1;\ N_{21} = 1$$

We leave it to the reader to verify that the trapezoidal Banerjee Inequality does not hold for this case.

3.5.4 Testing for All Direction Vectors

Burke and Cytron have given an algorithm to test for all direction vectors. That algorithm is reproduced in Figure 3.9. The function *try* is called with a direction vector of all "*"'s and an empty *dvlist* (i.e., $D :=$ (*,*,...,*); *dvlist* := *try*(D,1, $\emptyset$);).

This section needs to be fleshed out a little more.

FIGURE 3.9 MIV Direction Vector Test

```
set function try(D, k, dvlist);
    if not Banerjee(D)
        then return dvlist
    fi;
    if k = n
        then return dvlist ∪ {D}
    fi;
    for d := "<", "=", ">" do
        D_i := d;
        dvlist := try(D, k+1, dvlist)
    od
```

end *try*

3.5.5 RDIV Test

One special case of MIV subscripts that are worthy of special attention are those with the form $<a_1i+c_1, a_2j+c_2>$. These subscripts, called RDIV (Restricted Double Index Variable) subscripts, are similar to SIV subscripts, except that *i* and *j* are distinct indices. By observing different loop bounds for *i* and *j*, SIV tests may also be extended to exactly test RDIV subscripts.

3.6 Symbolic Tests

Most of the dependence tests that have been presented so far are precise when all terms involved are compile-time constants, but are unable to cope well with symbolic quantities. There are a few exceptions—for instance, loop-invariant symbolic additive constants have been treated by assuming that the difference $(c_2 - c_1)$ may be formed symbolically and simplified, then used like a constant. Symbolic quantities appear frequently in subscripts, arising from programming practices such as passing a multidimensional array to a procedure that only expects a singly-dimensioned array. As a result, symbolic tests are important.

This section describes a special test for independence between references that are contained in two different loops at the same nesting level. For instance, the test can be applied in the following pair of loops.

```
     DO i = 1, N1
S1      A(a1*i + c1) = ...
    ENDDO
     DO j = 1, N2
S2      ... = A(a2*j + c2)
    ENDDO
```

Assume for the sake of simplicity that a_1 is greater or equal to zero. A dependence exists if the following dependence equation is satisfied:

$$a_1i - a_2j = c_2 - c_1 \qquad \textbf{(EQ 3.33)}$$

for some value of *i*, $1 \le i \le N_1$, and *j*,$1 \le j \le N_2$. There are two cases to consider. First, a_1 and a_2 may have the same sign. In this case, $a_1i - a_2 j$ assumes its maximum value for $i = N_1$ and $j = 1$ and its minimum value

for $i = 1$ and $j = N_2$ (remember, a_1 and a_2 are non-negative). Hence, there is a dependence only if:

$$a_1 - a_2N_2 \leq c_2 - c_1 \leq a_1N_1 - a_2 \quad \text{(EQ 3.34)}$$

If either inequality is violated, the dependence cannot exist.

In the second case, a_1 and a_2 have different signs. In this case, $a_1i - a_2j$ assumes its maximum for $i = N_1$ and $j = N_2$, so there is a dependence only if:

$$a_1 - a_2 \leq c_2 - c_1 \leq a_1N_1 - a_2N_2 \quad \text{(EQ 3.35)}$$

If either inequality is violated, the dependence cannot exist.

It should be noted that these inequalities are just special cases of the Banerjee inequality. However, stated in this form, they obviously can be formulated for symbolic values of c_1, c_2, N_1 and N_2. Furthermore, this test may also be used to test for dependence in the same loop, with $N_1 = N_2$.

3.7 Delta Test

The tests used for separable subscripts can also be used on each subscript of a coupled group—if any test proves independence, then no dependence exists. However, several examples have already shown that subscript-by-subscript testing in a coupled group may produce false dependences (see, for example, page 74). Many researchers have worked on overcoming this deficiency \cite{LYZ:Lambda,Wallace:Matrix,WT:-Power}; this section presents one specific work, the Delta test. The Delta test is a multiple subscript test designed to be exact yet efficient for common coupled subscripts. The algorithm is overviewed in Figure 3.10.

The main insight driving the Delta test is that constraints derived from SIV subscripts may be propagated into other subscripts in the same coupled group efficiently, usually without any loss of precision. Since most coupled subscripts in scientific `FORTRAN` codes are simple, the Delta test is in practice an exact yet fast multiple subscript test.

The Delta test can detect independence if any of its component ZIV or SIV tests detect independence. Otherwise, it converts all SIV subscripts into constraints, propagating them into MIV subscripts where possible. This process repeats until no new constraints are found. Constraints are then propagate into coupled RDIV subscripts. Remaining MIV subscripts

are tested and the results are intersected with existing constraints. The following sections develop the Delta test in greater detail.

FIGURE 3.10 Delta Test Algorithm

```
procedure delta(subscripts, constraints)

    /* subscripts is a set of all coupled SIV and/or MIV subscripts. */
    /* constraints is an output parameter of hybrid distance/direction*/
    /* vector and constrained MIV subscripts. */

    initialize elements of constraint vector C to < none >

    while ∃ untested SIV subscripts in subscripts do
        apply SIV test to all untested SIV subscripts,
            return independence or
            derive new constraint vector C'
        C' ← C ∩ C'
        if C' = ∅
            then return independence
            else if C ≠ C' then
                C ← C';
                propagate constraint C into MIV subscripts,
                    possibly creating new ZIV or SIV subscripts;
                apply ZIV test to untested ZIV subscripts,
                    return independence or continue
            fi
        fi
    od

    while ∃ untested RDIV subscripts do
        test and propagate RDIV constraints
    od

    test remaining MIV subscripts, then
        intersect resulting direction vectors with C
    return distance/direction vectors from C
end delta;
```

3.7.1 Constraints

In the context of this book, *constraints* are assertions on indices that must hold for a dependence to exist. Given the tie to dependence, constraints are usually derived from subscripts. As an example, the dependence

equation applied to the subscript $<a_1 i + c_1, a_2 i' + c_2>$ generates the constraint $a_1 i - a_2 i' = c_2 - c_1$ for index *i*. Another example of a simple constraint is a dependence distance. The *constraint vector* $\mathbf{C} = (\delta_1, \delta_2, ..., \delta_n)$ is a vector with one constraint for each of the *n* indices in the coupled subscript group. In the Delta test, constraint vectors are used to store constraints generated from SIV tests. Because of the simple nature of SIV subscripts, these constraints can be easily converted to distance or direction vectors.

A constraint δ may have the following form:

- *dependence line* — a line $<ax + by = c>$ representing the dependence equation
- *dependence distance* — the value $<d>$ of the dependence distance; it is equivalent to the dependence line $<x - y = -d>$
- *dependence poin*t — a point $<x, y>$ representing dependence from iteration x to y

Dependence distances and lines derive directly from the strong and weak SIV tests. Dependence points result from intersecting constraints, as described in the next section.

3.7.2 Intersecting Constraints

Since dependence equations from all subscripts must simultaneously have solutions for any dependences to exist, intersecting constraints from each subscript can improve precision. If the intersection is the empty set, no dependence is possible. This may seem a little counterintuitive, because the term “constraint” usually implies a restriction or restraint, and thus, the lack of constraints on indices for dependences would seem to imply that all dependences can exist. The important fact to remember is that constraints are actually logical expressions guaranteed to be true when some dependence exists. If there are no such logical expressions, there can be no dependence. Constraint intersection has been employed for both direction vectors and coupled SIV subscripts. The version employed by the Delta test is equivalent to an exact multiple subscript SIV test.

Dependence distances are the easiest to intersect, by simply comparing the distances. If the distances are not all equal, then they cannot simultaneously hold, and no dependences exist. For example, reconsider the following loop nest from page 74:

```
DO i
```

```
S1      A(i+1, i+2) = A(i, i) + C
    ENDDO
```

The strong SIV test applied to the first subscript shows a dependence distance of 1; applied to the second subscript, it shows a distance of 2. Intersecting the two constraints is performed by comparing them. They are not equal, producing the empty set as the constraint set and proving independence. This reflects the fact that any value *i'* substituted into the subscripts on the left side cannot simultaneously be both 1 and 2 less than any value of *i* substituted into the subscripts on the right side.

Even complex constraints from SIV subscripts may be intersected exactly. Since each dependence equation from a SIV subscript may be viewed as a line in a two-dimensional plane, intersecting constraints from multiple SIV subscripts corresponds to calculating the point(s) of intersection for lines in a plane. No dependence exists if the lines do not intersect at a common point within the loop bounds, or if the coordinates of this point do not have integer values. If all dependence equations intersect at a single *dependence point*, its coordinates are the only two iterations that actually cause dependence.

```
    DO i
S1      A(i, i) = A(1, i-1) + C
    ENDDO
```

In this example loop, testing the first and second subscripts in the pair of references to `A` derives the dependence lines < i = 1 > and < i = i' - 1 >, respectively. These dependence lines intersect at the dependence point <1,2>, indicating that the only dependence is from the first to the second iteration. Since calculating the intersection of lines in a plane can be performed precisely, constraint intersection is exact.

The full constraint intersection algorithm is shown in Figure 3.11. For simplicity, dependence distances are also treated as lines at places in the algorithm.

FIGURE 3.11 Constraint Intersection Algorithm

constraint **function** *intersect*(δ_1, δ_2, *L,U)*

```
    /* intersect takes 2 constraints δ1 and δ2 for the same loop index */
    /* plus the lower and upper bounds for the loop (L and U). */
    /* It returns the intersection of the constraints; by convention, if */
    /* that intersection is ∅, independence is shown. */
```

```
/* either δ1 or δ2 = < none > */
if δ1 = < none > then return δ2 fi
if δ2 = < none > then return δ1 fi

/* both δ1 and δ2 are dependence distances */
if δ1 = < d1 > and δ2 = < d2 > then
    if d1 = d2
        then return < d1 >
        else return ∅
fi  fi

/* both δ1 and δ2 are dependence points */
if δ1 = <x1, y1> and δ2 = <x2, y2> then
    if x1 = x2 and y1 = y2
        then return < x1, y1 >
        else  return ∅
fi  fi

/* both δ1 and δ2 are dependence lines/distances */
if δ1 = <a1x+b1y=c1> and δ2 = <a2x+b2y=c2> then
    /* lines are parallel if slopes are equal */
    if a1/b1 = a2/b2 then
        if c1/b1 = c2/b2
            then return <a1x+b1y=c1>
            else return ∅
        fi
    /* lines must intersect if not parallel */
    else
        <x1, y1> ← intersection of δ1 and δ2
        if L ≤ x1 ≤ U and x1 is an integer and
                L ≤ y1 ≤ U and y1 is an integer
            then return <x1, y1>
            else return ∅
    fi  fi

/* either δ1 is a dependence line/distance */
/* and δ2 is a dependence point, or vice versa */
/* without loss of generality, assume the former */}
else if δ1 = <a1x+b1y=c1> and δ2 = <x1, y1>
    if a1x1+b1y1=c1
        then return <x1, y1>
        else return ∅
fi  fi

/* This point is unreachable, so an error should be asserted. */
```

end *intersect*

3.7.3 Constraint Propagation

3.7.3.1 SIV Constraints

A major contribution of the Delta test is its ability to propagate constraints derived from SIV subscripts into coupled MIV subscripts, usually with no loss of precision. The resulting constrained subscript can then be tested with greater efficiency and precision. Figure 3.12 lists a constraint propagation algorithm for the Delta test. The goal of this algorithm is to utilize SIV constraints for each index to eliminate instances of that index in the target MIV subscript. To make the algorithm more concrete, consider the following example:

```
    DO i
       DO j
S1        A(i+1, i+j) = A(i, i+j)
       ENDDO
    ENDDO
```

Applying the strong SIV test to the first subscript of array `A` reveals a dependence distance of <1> for index *i*. Propagating this constraint into the second subscript to eliminate both occurrences of *i* results in the constrained SIV subscript < *j*–1, *j* >. The strong SIV test applied to this subscript gives a distance of –1 on loop *j*. Since this completes testing on all subscripts, it also completes the Delta test. Merging the elements of the constraint vector gives a dependence with distance vector (1, -1).

Probably need to mention more about the λ test heuristic for this paragraph to make sense. It might make more sense to kill the paragraph.

Constraint propagation in this example is exact, because both instances of index *i* in the constrained subscript were eliminated. Empirical studies \cite {Goff, Kennedy, etc. in 92 compiler construction} show that this is frequently the case for scientific codes. In general the algorithm may only eliminate one occurrence of an index. This results in improved precision when testing coupled groups, but is not exact. If desired, additional precision may be gained by utilizing the constraint to reduce the range of the remaining index, as in Fourier-Motzkin Elimination \cite{Schrijver86}.

The constraint propagation algorithm is an incremental adaptation of the λ-test heuristic for selecting linear combinations of subscript expressions. It has also been extended to efficiently handle constraints from SIV

tests and linearly dependent subscripts \cite{LYZ:Lambda}. Later sections present some more examples.

3.7.3.2 Multiple Passes

The Delta test algorithm iterates if MIV subscripts are reduced to SIV subscripts, since that action may produce new constraints. The following loop nest provides an example:

```
    DO i
       DO j
          DO k
S1           A(j-i, i+1, j+k) = A(j-i, i, j+k)
          ENDDO
       ENDDO
    ENDDO
```

In the first pass of the Delta test, the second subscript is tested, producing a dependence distance of < 1 > on the *i* loop. This constraint is then propagated into the first subscript, resulting in the subscript < j+1, j >.

Since a new SIV subscript has been created, the algorithm repeats. On the second pass, the new subscript is tested to produce a distance of 1 on the *j* loop. This constraint is then propagated into the third subscript to derive the subscript (k-1, k). The new SIV subscript causes another pass that discovers a distance of -1 on the *k* loop. Since all SIV subscripts have been tested, the Delta test halts at this point, returning the distance vector (1, 1, -1).

FIGURE 3.12 Constraint Propagation Algorithm

constraint **function** *propagate(S,***C***)*

/* *propagate* takes a MIV subscript *S* with form */
/* < a_1 i_1 + ... + a_n i_n + e, \; a_1' i_1'} + ... + a_n' i_n' + e'> */
/* and constraint vector **C**= <δ_1, δ_2, ..., δ_n>. It returns a */
/* hopefully further constrained ZIV, SIV, or MIV subscript */

for each index i_k with nonzero a_k or a_k' **do**

if δ_k = dependence distance < d > **then**
$e \leftarrow e - a_k d$;
$a_k \leftarrow 0$;
$a_k' \leftarrow a_k' - a_k$

```
        else if δk = dependence line <α x+ β y=c> then
            if α = 0 then
                e ← e - ak'c/ β;
                ak' ← 0
            else if β  = 0 then
                e ← e + akc/α;
                ak ← 0
            else if α  =  β then
                e ← e + ak c/ α;
                ak ← 0;
                ak' ← ak' + ak
            else

                /* multiply terms of subscript by α to */
                /* retain integer coefficients in result */
                for each τ \in {a1, ..., an, a1', ..., an', e, e'} do
                    τ ← α τ
                od
                e ← e + akc;
                ak ← 0;
                ak' ← ak' + akβ
            fi

        else if δk = dependence point <x, y> then
            e ← e + ak x - ak' y
            ak ← 0; ak' ← 0
        fi
    od
end propagate
```

3.7.3.3 Improved Precision

The Delta test may improve the precision of other dependence tests on remaining constrained MIV subscripts.

```
    DO i = 1,100
       DO j = 1,100
S1        A(i-1,2i) = A(i,i+j+110)
       ENDDO
    ENDDO
```

Banerjee's inequalities alone applied to the subscripts in this example loop cannot prove independence. The Delta test improves on this result by first converting the leftmost subscripts into a dependence distance constraint of <-1>. This constraint is then propagated into the rightmost

subscripts to produce the constrained MIV subscript < 2, j-i+110 >, which can be successfully handled (i.e. independence detected) by Banerjee's inequalities.

```
    DO i
       DO j
S1        A(i,2j+i) = A(i,2j-i+5)
       ENDDO
    ENDDO
```

Similarly, in this example loop the GCD test shows integer solutions for both subscripts. However, propagating the distance constraint < 0 > for *i* from the first subscript into the second subscript yields the constrained MIV subscript <2j, 2j-2i+5>. The GCD test can now detect independence since the GCD of the coefficients of all the indices is 2, which does not divide evenly into the constant term 5.

3.7.3.4 Distance Vectors

The Delta test is particularly useful for analyzing dependences in skewed loops, including upper triangular loops skewed by loop normalization. Consider the following simplified kernel from the Livermore Loops:

```
    DO i = 1, N
       DO j = 1, N
S1        A(i,j) = A(i-1,j) + A(i,j-1) +
                   A(i+1,j) + A(i,j+1)
       ENDDO
    ENDDO
```

Since all subscripts are separable, the strong SIV test will calculate distance vectors of (1, 0) and (0, 1) for the true dependences in the loop nest. As later chapters will show, a dependence pattern such as this does not permit exploitation of much parallelism. A transformation known as *loop skewing* can enhance the available parallelism. Loop skewing applied to this nest results in

```
    DO i = 1, N
       DO j = 1+i, N+i
S1        A(i,j-i) = A(i-1,j-i) + A(i,j-i-1)
                   + A(i+1,j-i) + A(i,j-i+1)
       ENDDO
    ENDDO
```

While loop skewing may increase parallelism, it also increases the difficulty of dependence analysis. In this example, for instance, most dependence tests are unable to calculate distance vectors due to the presence of MIV subscripts. The Delta test, however, does permit accurate analysis of these subscripts. The test propagates distance constraints for *i* from the first subscript into the second subscript, deriving the distance vectors (1, 1) and (0, 1). Note that these distance vectors are not the same as the original example; loop skewing has changed the distance vectors, although it has not changed the underlying dependences.

3.7.3.5 Restricted DIV Constraints

The previous section showed how SIV constraints may be propagated. MIV constraints may also be propagated, but are expensive to propagate in the general case. However, in the special case of coupled RDIV subscripts(introduced in Section 3.5.5 on page 103), we present a method to handle an important special case. For simplicity, only the following types of array references are considered:

```
    DO i
       DO j
S1         A(i1+c1, i2+c2) = A(i3+c3, i4+c4)
       ENDDO
    ENDDO
```

When i_1, i_2 are both instances of index *i*, and i_3, i_4 are both instances of index *j*, a constraint between *i* and *j* is derived from the first subscript. That constraint may be propagated into the second subscript using the algorithms for SIV subscripts discussed previously. The only additional consideration is that bounds for *i* and *j* may differ.

More commonly, i_1,i_4 are both instances of index *i*, and i_2,i_3 are both instances of index *j*. In this case, the following set of dependence equations result:

$$i+c_1 = j'+c_3$$

$$j+c_2 = i'+c_4$$

Each dependence equation may be tested separately without loss of precision when checking for dependence. However, both equations must be considered simultaneously when determining which distance or direction vectors are possible.

These constraints can be propagated by considering instances of index *i* in the second reference as $i + \Delta_i$, where Δ_i is the dependence distance between the two occurrences of *i*. The index *j* is treated in the same fashion, producing following set of dependence equations:

$$i+c_1 = j+\Delta_j+c_3$$

$$j+c_2 = i+\Delta_i+c_4$$

Adding these equations together and rearranging terms slightly yields the following equation:

$$\Delta_i + \Delta_j = c_1 + c_2 - c_3 - c_4$$

This dependence equation can then be checked when testing for a specific distance or direction vector.

To illustrate this more concretely, consider the following example of array transposition.

I don't know about this example -- when you put the bounds in, array transposition is trivial to pick up by normal tests.

```
     DO i
        DO j
S1         A(i,j) = A(j,i) + C
        ENDDO
     ENDDO
```

Propagating RDIV constraints results in the dependence equation $\Delta_i + \Delta_j = 0$. As a result, distance vectors must have the form (d,-d), and the only valid direction vectors are (<,>) and (=,=). All dependences are thus carried on the outer loop; the inner loop may be executed in parallel.

3.7.4 Precision and Complexity

The precision of the Delta test depends on the nature of the coupled subscripts being tested. The SIV tests applied in the first phase are exact. The constraint intersection algorithm is also exact, since the intersection of any number of lines in a plane can be calculated precisely. The Delta test is thus exact for any number of coupled SIV subscripts.

In the constraint propagation phase, weak-zero SIV constraints and dependence points may always be applied exactly, since they assign values to occurrences of an index in a subscript. Dependence distances (from strong SIV subscripts) may also be propagated into MIV subscripts without loss of precision when the coefficients of the corresponding index are equal. Fortunately, this is frequently the case in scientific codes.

When constraints can be propagated exactly and all subscripts uncoupled by eliminating shared indices, the Delta test prevents loss of precision due to multiple subscripts. At its conclusion, if the Delta test has tested all subscripts using ZIV and SIV tests, the answer is exact. If only separable MIV subscripts remain, the Delta test is limited by the precision of the single subscript tests applied to each subscript. Research has shown that the Banerjee-GCD test is usually exact for single subscripts \cite{Bane:Masters,KPK:Accuracy,LY:Exact}, so the Delta test is also likely to be exact for these cases.

There are three sources of imprecision for the Delta test. First, constraint propagation of dependence lines and distances may be imprecise if an index cannot be completely eliminated from both references in the target subscripts. Second, complex iteration spaces such as triangular loops may impose constraints between subscripts not utilized by the Delta test.- Finally, the Delta test does not propagate constraints from general MIV subscripts. As a result, coupled MIV subscripts may remain at the end of the Delta test. More general but expensive multiple subscript dependence tests such as the λ or Power tests may be used in these cases.

Since each subscript in the coupled group is tested at most once, the complexity of the Delta test is linear in the number of subscripts. However, constraints may be propagated into subscripts multiple times.

3.8 Implementation Tradeoffs

I think we may want to put a section in here giving the important tradeoffs to be made in real implementations, and how to make them. Putting the statistics from the Goff et al paper in an appendix also seems like a good idea.

3.9 Related Work

This section discusses the large body of work in the field of dependence testing. The suite of tests presented in this paper are distinguished by the fact that they combine high precision and efficiency by targeting a simple yet common subset of all possible subscripts.

3.9.1 Integer and Linear Programming

Since testing linear subscript functions for dependence is equivalent to finding simultaneous integer solutions within loop limits, one approach has been to employ integer programming methods \cite{Feautrier88, Schrijver86}. Linear programming techniques such as Shostak's loop residue \cite{Shostak81} or Karmarkar's method \cite{Karmarkar} are also applicable, though integer solutions are not guaranteed. Unfortunately, while integer and linear programming techniques are suitable for solving large systems of equations, their high initialization costs and implementation complexity make them less desirable for dependence testing in a production compiler.

The earliest work on dependence tests concentrated on deriving distance vectors from strong SIV subscripts \cite{KMC:72,Lamport,Muraoka:Dissertation}. Cohagan \cite{Cohagan73} described a test that analyzes general SIV subscripts symbolically. Banerjee and Wolfe \cite{Bane:Dissertation,Wolfe:Dissertation} developed the current form of the Single-Index exact test.

For MIV subscripts, the GCD test may be used to check unconstrained integer solutions \cite{Bane:Masters,Ken:80}. Banerjee's inequalities provide a useful general purpose single subscript test for constrained real solutions \cite{Bane:Dissertation}. It has also been adapted to provide many different types of dependence information \cite{AlKe:Automatic,Bane:88,BuCy:86,Ken:80,Ken:Triangular,Wolfe:Dissertation}. Research has shown that Banerjee's inequalities are exact in many common cases \cite{Bane:Masters,KPK:Accuracy,LY:Exact}, though results have not yet been extended for direction vectors or complex iteration spaces.

The I-test developed by Kong *et al* integrates the GCD and Banerjee tests and can usually prove integer solutions \cite{KKP:Itest}. Gross and Steenkiste propose an efficient interval analysis method for calculating dependences for arrays \cite{GS:Interval}. Unfortunately their method does not handle coupled subscripts, and is unsuitable for most loop transformations since distance and direction vectors are not calculated. Lichnewsky and Thomasset describe symbolic dependence testing in the vatil vectorizer \cite{LT:Symbolic}. Haghighat and Polychronopoulos propose a flow analysis framework to aid symbolic tests \cite{HaPo:Symbolic}.

Execution conditions may also be used to refine dependence tests. Wolfe's All-Equals test checks for loop-independent dependences invalidated by control flow within the loop \cite{Wolfe:Dissertation}. Lu and Chen's *subdomain test* incorporates information about indices from conditionals within the loop body \cite{LC90:Subdomain}. Klappholz and Kong have extended Banerjee's inequalities to do the same \cite{KK:Execute}.

3.9.2 Multiple Subscript Tests

Early approaches to impose simultaneity in testing multidimensional arrays include intersecting direction vectors from each dimension \cite{Wolfe:Dissertation} and linearization \cite{BuCy:86,GiPo:88}; they proved inaccurate in many cases. True multiple subscript tests provide precision at the expense of efficiency by considering all subscripts simultaneously. In comparison, the Delta test propagates constraints incrementally as needed.

Many of the earliest multiple subscript tests utilized Fourier-Motzkin elimination, a linear programming method based on pair-wise comparison of linear inequalities. Kuhn \cite{Kuhn:Dissertation} and Triolet *et al* \cite{TIF:Call} represent array accesses in convex regions that may be intersected using Fourier-Motzkin elimination. Regions may also be used to summarize memory accesses for entire segments of the program. These techniques are flexible but expensive. Triolet found that using Fourier-Motzkin elimination for dependence testing takes from 22 to 28 times longer than conventional dependence tests \cite{Triolet:Inter}.

The Constraint-Matrix test developed by Wallace is a simplex algorithm modified for integer programming \cite{Wallace:Matrix}. Its precision and expense are difficult to ascertain since it halts after an arbitrary number of iterations to avoid cycling. The simplex algorithm has worst case exponential complexity, but takes only linear time for most linear programming problems. However, Schrijver states that in combinatorial problems where coefficients tend to be 1, 0, or -1, the simplex algorithm is slow and will cycle for certain pivot rules \cite{Schrijver86}.

Li *et al* present the λ-test, a multidimensional version of Banerjee's inequalities that checks for simultaneous constrained real-valued solutions \cite{LYZ:Lambda}. The λ-test forms linear combinations of subscripts that eliminate one or more instances of indices, then tests the result using Banerjee's inequalities. Simultaneous real-valued solutions exist if and

only if Banerjee's inequalities finds solutions in all the linear combinations generated.

The λ-test can test direction vectors and triangular loops. Its precision may be enhanced by also applying the GCD or Single-Index exact tests to the pseudosubscripts generated. However, there is no obvious method to extend the λ-test to prove the existence of simultaneous integer solutions. The λ-test is exact for two dimensions if unconstrained integer solutions exist and the coefficients of index variables are all 1, 0 or -1 \cite{LY:Exact}. However, even with these restrictions it is not exact for three or more coupled dimensions.

The Delta test may be viewed as a restricted form of the λ-test that trades generality for greater efficiency and precision.

Banerjee's multidimensional GCD test checks for simultaneous unconstrained integer solutions in multidimensional arrays \cite{Bane:88}. It applies Gaussian elimination modified for integers to create a compact system where all integer points provide integer solutions to the original dependence system. It can also be extended to provide an exact test for distance vectors \cite{WT:Power}.

Wolfe and Tseng's Power test gains great precision by applying loop bounds using Fourier-Motzkin elimination to the dense system resulting from the multidimensional GCD test \cite{WT:Power}. The Power test is expensive, but is also flexible and well-suited for providing precise dependence information such as direction vectors in imperfectly nested loops, loops with complex bounds, and non-direction vector constraints.

Both the Constraint-Matrix and λ-tests require that a pretest be used to eliminate linearly dependent subscripts. In comparison, the Power and Delta tests can detect and discard linearly dependent subscripts as part of their basic algorithm.

Li *et al* showed that for coupled subscripts, multiple subscript tests may detect independence in up to 36% more cases than subscript-by-subscript tests in libraries such as `EISPACK` \cite{LYZ:Lambda}. Our results for `EISPACK` demonstrate that the Delta test is as effective in testing coupled subscripts. A comprehensive empirical study of array subscripts and conventional dependence tests was performed by Shen *et al* \cite{SLY:-TOPDS}.