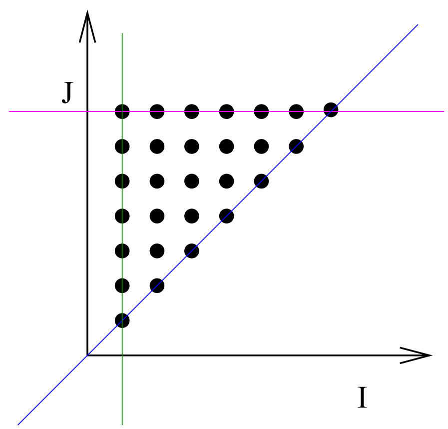
Transformations and Dependences

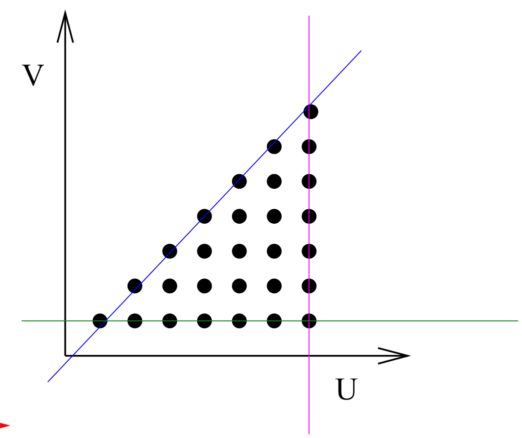
Recall:

- Polyhedral algebra tools for
  - determining emptiness of convex polyhedra
  - enumerating integers in such a polyhedron.
- Central ideas:
  - reduction of matrices to echelon form by unimodular column operations,
  - Fourier-Motzkin elimination

Let us use these tools to determine (i) legality of permutation and (ii) generation of transformed code.

Loop permutation can be modeled as a linear transformation on iteration space:





$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

```
DO  I =  1, N
  DO J = I,N
    X(I,J) = 5
```

```
DO U = 1, N
  DO V = 1,U
    X(V,U) = 5
```

Permutation of loops in n-loop nest: nxn permutation matrix P

$$P\,\underline{I} = \underline{U}$$

Questions:

(1) How do we generate new loop bounds?

(2) How do we modify the loop body?

(3) How do we know when loop interchange is legal?

Code Generation for Transformed Loop Nest

Two problems: (1) Loop bounds (2) Change of variables in body

(1) New bounds:

Original bounds: $A * \underline{I} \leq b$ where A is in echelon form

Transformation: $\underline{U} = T * \underline{I}$

Note: for loop permutation, T is a permutation matrix
=> inverse is integer matrix

So bounds on U can be written as $A * T^{-1}\underline{U} \leq b$

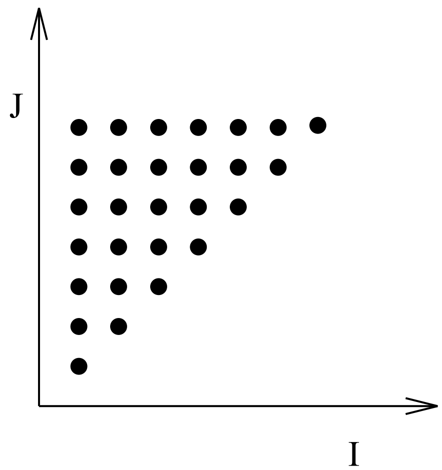
Perform Fourier-Motzkin elimination on this system of inequalities to obtain bounds on $\underline{U}$.

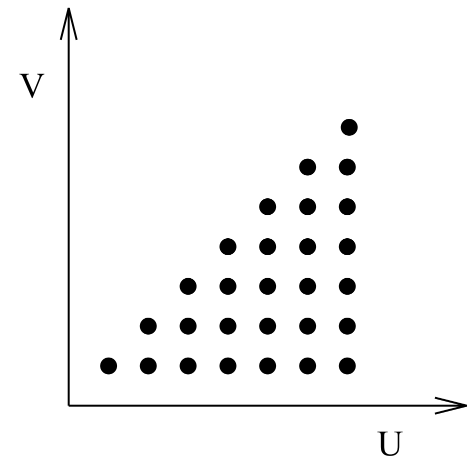
(2) Change of variables:

$\underline{I} = T^{-1}\underline{U}$

Replace old variables by new using this formula

Example:





$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

```
DO I = 1, N
  DO J = I,N
    X(I,J) = 5
```

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I \\ J \end{bmatrix} \leq \begin{bmatrix} -1 \\ N \\ 0 \\ N \end{bmatrix}$$

```
DO U = 1, N
  DO V = 1,U
    X(V,U) = 5
```

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} U \\ V \end{bmatrix} \leq \begin{bmatrix} -1 \\ N \\ 0 \\ N \end{bmatrix}$$

Fourier-Motzkin elimination

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} U \\ V \end{bmatrix} \leq \begin{bmatrix} -1 \\ N \\ 0 \\ N \end{bmatrix}$$

$$\begin{bmatrix} 0 & -1 \\ 0 & 1 \\ -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} U \\ V \end{bmatrix} \leq \begin{bmatrix} -1 \\ N \\ 0 \\ N \end{bmatrix}$$

Projecting out V from system gives

$$1 \leq U \leq N$$

Bounds for V are

$$1 \leq V \leq \min(U,N)$$

These are loop bounds given by FM elimination.
With a little extra work, we can simplify the upper bound of V to U.

Key points:

- Loop bounds determination in transformed code is mechanical.
- Polyhedral algebra technology can handle very general bounds with max's in lower bounds and min's in upper bounds.
- No need for pattern matching etc for triangular bounds and the like.

When is permutation legal?

Position so far: if there is a dependence between iterations, then permutation is illegal.

```
DO I = 1, 100
 DO J = 1, 100
  X(2I,J) = .... X(2I-1,J-1)...
```

Is there a flow dependence between different iterations?

$$\begin{aligned} 1 &\leq Iw, Ir, Jw, Jr \leq 100 \\ (Iw, Jw) &\prec (Ir, Jr) \\ 2Iw &= 2Ir - 1 \\ Jw &= Jr - 1 \end{aligned}$$

ILP decision problem: is there an integer in union of two convex polyhedra?

No => permutation is legal.

Permutation is legal only if dependence does not exist: too simplistic.

Example:

```
DO I = 1, 100
 DO J = 1, 100
  X(I,J) = .... X(I-1,J-1)...
```

Only dependence is flow dependence:

$$\begin{aligned} 1 &\leq Iw, Jw, Ir, Jr \leq 100 \\ (Iw, Jw) &\prec (Ir, Jr) \\ Iw &= Ir - 1 \\ Jw &= Jr - 1 \end{aligned}$$

ILP problem has solution: for example, $(Iw = 1, Jw = 1, Ir = 2, Jr = 2)$

Dependence exists but loop interchange is legal!

Point: Existence of dependence is a very “coarse” criterion to determine if interchange is legal.

Additional information about dependence may let us conclude that a transformation is legal.

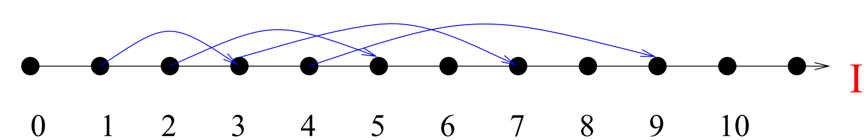
To get a handle on all this, let is first define dependence precisely.

Consider single loop case first:

```
DO I = 1, 100
  X(2I+1) = ....X(I)...
```

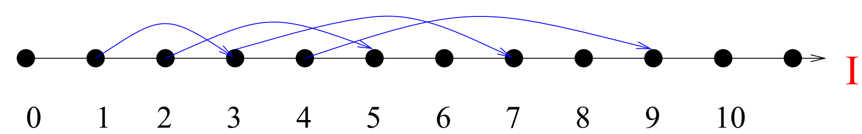
```
Flow dependences between iterations:
   Iteration 1 writes to X(3) which is read by iteration 3.
   Iteration 2 writes to X(5) which is read by iteration 5.
   ....
   Iteration 49 writes to X(99) which is read by iteration 99.
```

If we ignore the array locations and just think about dependence between iterations, we can draw this geometrically as follows:



Dependence arrows always go forward in iteration space. (eg. there cannot be a dependence from iteration 5 to iteration 2)

Intuitively, dependence arrows tell us constraints on transformations.



Suppose a transformed program does iteration 2 before iteration 1. OK!

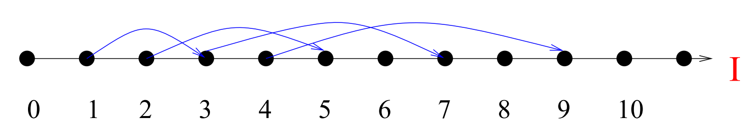
Transformed program does iteration 3 before iteration 1. Illegal!

Formal view of a dependence: relation between points in the iteration space.

```
DO I = 1, 100
  X(2I+1) = ....X(I)...
```

Flow dependence $= \{(Iw, 2Iw + 1) | 1 \leq Iw \leq 49\}$

(Note: this is a convex set)



In the spirit of dependence, we will often write this as follows:

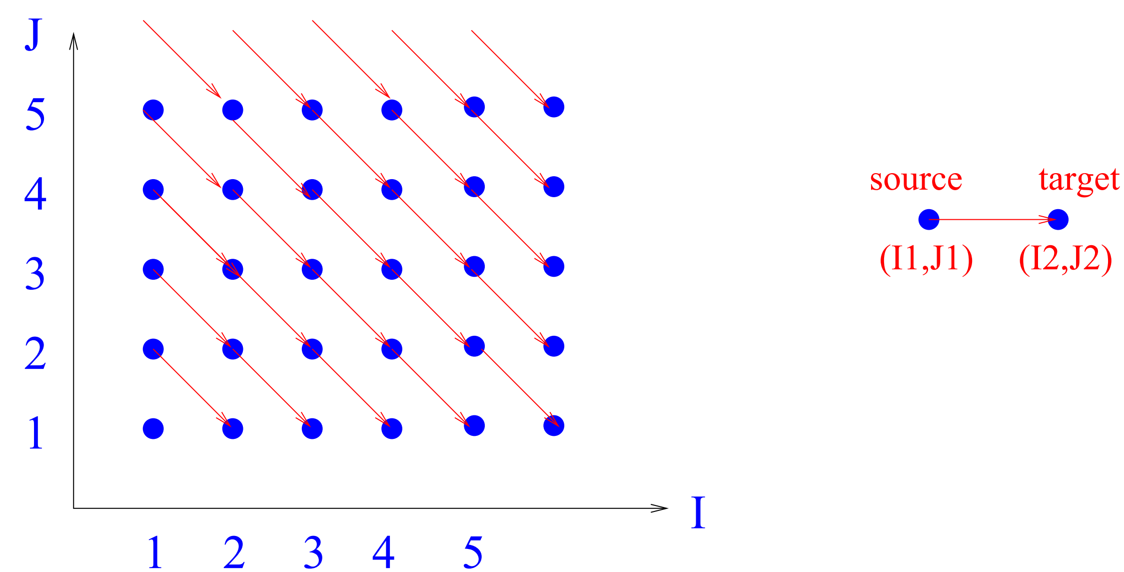
Flow dependence $= \{(Iw \rightarrow 2Iw + 1) | 1 \leq Iw \leq 49\}$

2D loop nest

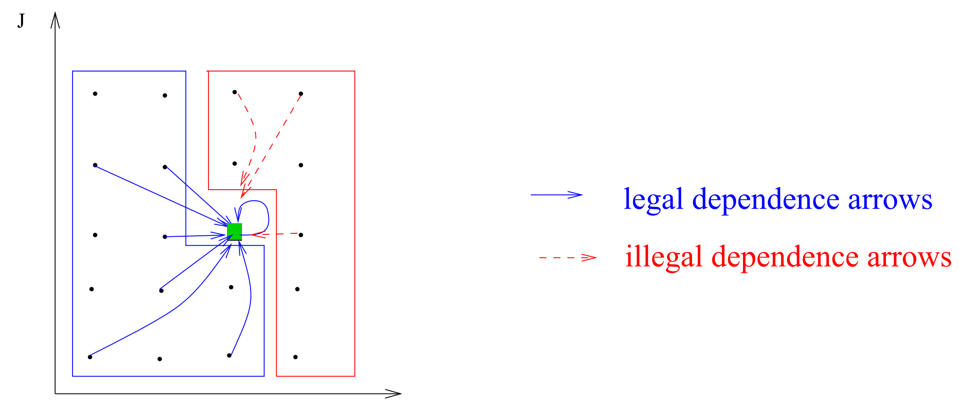
```
DO 10 I = 1,100
  DO 10 J = 1,100
  10 X(I,J) = X(I-1,J+1) + 1
```

Dependence: relation of the form $(I_1, J_1) \longrightarrow (I_2, J_2)$.

Picture in iteration space:



Legal and illegal dependence arrows:



If $(A \rightarrow B)$ is a dependence arrow, then A must be lexicographically less than or equal to B.

Dependence relation can be computed using ILP calculator

```
DO 10 I = 1,100
  DO 10 J = 1,100
  10 X(I,J) = X(I-1,J+1) + 1
```

Flow dependence constraints: $(I_w, J_w) \rightarrow (I_r, J_r)$

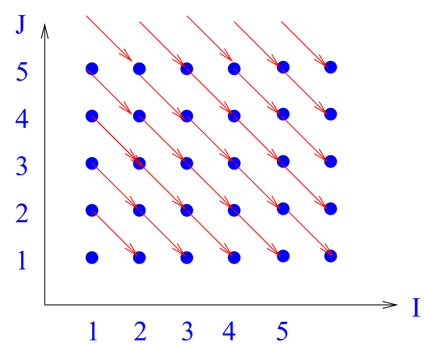
- $1 \leq Iw, Ir, Jw, Jr \leq 100$
- $(I_w, J_w) \prec (I_r, J_r)$
- $I_w = I_r - 1$
- $J_w = J_r + 1$

Use ILP calculator to determine the following relation:

$D = \{(Iw, Jw) \rightarrow (Iw + 1, Jw - 1) | (1 \leq Iw \leq 99) \wedge (2 \leq Jw \leq 100)\}$

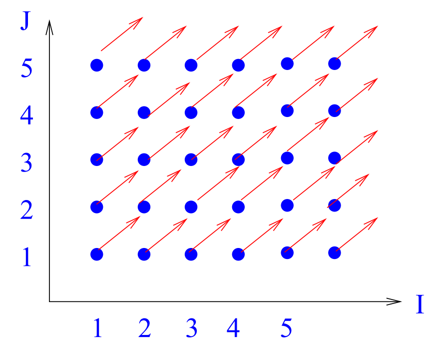
If we have the full dependence relation, can we determine when permutation is legal?

Let us look at geometric picture to understand when permutation is legal.




```
DO I = 1,N
  DO J = 1,N
    X(I,J) = X(I-1,J+1)......
```

Permutation is illegal




```
DO I = 1, N
  DO J = 1,N
    X(I,J) = X(I-1,J-1)......
```

Permutation is legal

Intuitively, if an iteration is dependent on an iteration in its ”upper left hand corner”, permutation is illegal. How do we express this formally?

Legality of permutation can be framed as an ILP problem.

```
DO 10 I = 1,100
  DO 10 J = 1,100
  10 X(I,J) = X(I-1,J+1) + 1
```

Permutation is illegal if there exist iterations $(I_1, J_1), (I_2, J_2)$ in source program such that

- $((I_1, J_1) \rightarrow (I_2, J_2)) \in D$ (dependent iterations)
- $(J_2, I_2) \prec (J_1, I_1)$ (iterations done in wrong order in transformed program)

This can obviously be phrased as an ILP problem and solved.
One solution: $(I_1, J_1) = (1, 2)$, $(I_2, J_2) = (2, 1)$.
Interchange is illegal.

General picture:

Permutation is co-ordinate transformation: $\underline{U} = P * \underline{I}$ where P is a permutation matrix.

Conditions for legality of transformation:

For each dependence D in loop nest, check that there do not exist iterations $\underline{I}_1$ and $\underline{I}_2$ such that

$$(\underline{I}_1 \rightarrow \underline{I}_2) \in D$$
$$P(\underline{I}_2) \prec P(\underline{I}_1)$$

First condition: dependent iterations
Second condition: iterations are done in wrong order in transformed program.

Legality of permutation can be determined by solving a bunch of ILP problems.

Problems with using full dependence sets:

- Expensive (time/space) to compute full relations
- Need to solve ILP problems again to determine legality of permutation
- Symbolic loop bounds ('N') require parameterized sets ('N' is unbound variable in definition of dependence set)

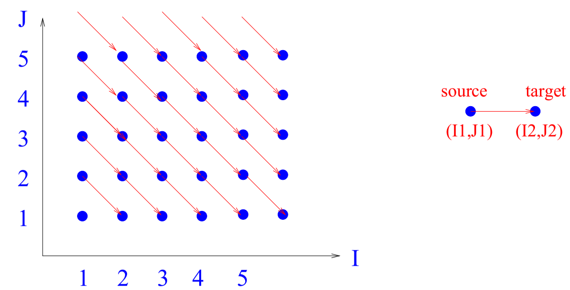
Dependence abstractions: summary of dependence set D

- less information than full set of tuples in D
- more information than non-emptiness of D
- intuitively, "as much as is needed for transformations of interest"

Distance/direction: Summarize dependence relation

Look at dependence relation from earlier slides:

$$\{(1,2) \rightarrow (2,1), (1,3) \rightarrow (2,2), ..(2,2) \rightarrow (3,1)...\}$$



Difference between dependent iterations = $(1, -1)$. That is,

$$(I_w, J_w) \rightarrow (I_r, J_r) \in \text{dependence relation, implies}$$

$$I_r - I_w = 1$$

$$J_r - J_w = -1$$

We will say that the *distance vector* is $(1, -1)$.

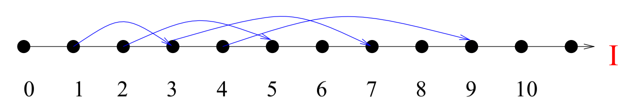
Note: From distance vector, we can easily recover the full relation.

In this case, distance vector is an *exact* summary of relation.

Set of dependent iterations usually is represented by many distance vectors.

```
DO I = 1, 100
  X(2I+1) = ....X(I)...
```

Flow dependence $= \{(Iw \rightarrow 2Iw + 1) | 1 \leq Iw \leq 49\}$



Distance vectors: $\{(2), (3), (4), ... , (50)\}$

Distance vectors can obviously never be negative (if (-1) was a distance vector for some dependence, there is an iteration I_1 that depends on iteration $I_1 + 1$ which is impossible.)

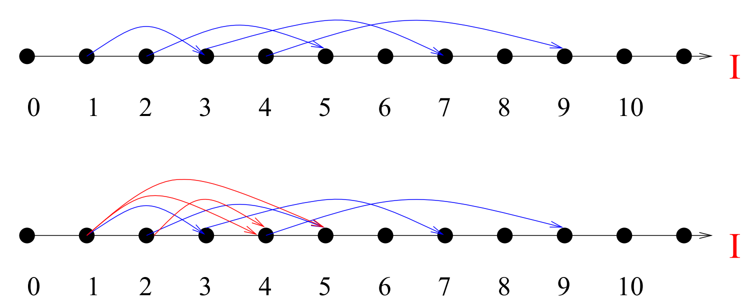
Distance vectors are an approximation of a dependence:

(intuitively, we know the arrows but we do not know their sources.)

Example: $D = \{(Iw, 2Iw+1) | 1 \leq Iw \leq 49\}$

Distance vectors: $\{(2), (3), (4), \ldots, (50)\}$

$D_1 = \{(I_1, I_2) | (1 \leq I_1 \leq 49) \wedge (50 + I_1) \geq I_2 \geq (2I_1 + 1)\}$ is a (convex) superset of D that has the same distance vectors.



Both dependences have same set of distance vectors

Computing distance vectors for a dependence

```
DO I = 1, 100
  X(2I+1) = ....X(I)...
```

Flow dependence:

$$\begin{aligned} 1 &\leq Iw < Ir \leq 100 \\ 2Iw + 1 &= Ir \end{aligned}$$

Flow dependence = $\{(Iw, 2Iw + 1) | 1 \leq Iw \leq 49\}$

Computing distance vectors without computing dependence set:
Introduce a new variable $\Delta = Ir - Iw$ and project onto Δ

$$\begin{aligned} 1 &\leq Iw < Ir \leq 100 \\ 2Iw + 1 &= Ir \\ \Delta &= Ir - Iw \end{aligned}$$

Solution: $\Delta = \{d | 2 \leq d \leq 50\}$

Example:2D loop nest

```
DO 10 I = 1,100
  DO 10 J = 1,100
  10 X(I,J) = X(I-1,J+1) + 1
```

Flow dependence constraints: $(I_w, J_w) \rightarrow (I_r, J_r)$
Distance vector: $(\Delta_1, \Delta_2) = (I_r - I_w, J_r - J_w)$

- $1 \leq Iw, Ir, Jw, Jr \leq 100$
- $(I_w, J_w) \prec (I_r, J_r)$
- $I_w = I_r - 1$
- $J_w = J_r + 1$
- $(\Delta_1, \Delta_2) = (I_r - I_w, J_r - J_w)$

Solution: $(\Delta_1, \Delta_2) = (1, -1)$

General approach to computing distance vectors:

Set of distance vectors generated from a dependence is itself a polyhedral set.

Computing distance vectors without computing dependence set:

To the linear system representing the existence of the dependence, add new variables corresponding to the entries in the distance vector and project onto these variables.

Reality check:

In general, dependence is some complicated convex set.

In general, distance vectors of a dependence are also some complicated convex set!

What is the point of “summarizing” one complicated set by another equally complicated set?!!

Answer: We use distance vector summary of a dependence only when dependence can be summarized by a single distance vector (called a uniform dependence).

How do we summarize dependence when we do not have a uniform dependence? Answer: use direction vectors.

Direction vectors **Example:**

```
DO 10 I = 1,100
 10 X(2I+1) = X(I) + 1
```

Flow dependence equation: $2I_w + 1 = I_r$.

Dependence relation: $\{(1 \rightarrow 3), (2 \rightarrow 5), (3 \rightarrow 7), ...\}$ (1).

No fixed distance between dependent iterations!

But all distances are +ve, so use *direction vector* instead.

Here, direction = (+).

Intuition: (+) direction = some distances in range $[1, \infty)$

In general, direction = (+) or (0) or (-).

Also written by some authors as (<), (=), or (>).

Direction vectors are not exact.

(eg):if we try to recover dependence relation from direction (+), we get bigger relation than (1):

$\{(1 \rightarrow 2), (1 \rightarrow 3), ..., (1 \rightarrow 100), (2 \rightarrow 3), (2 \rightarrow 4), ...\}$

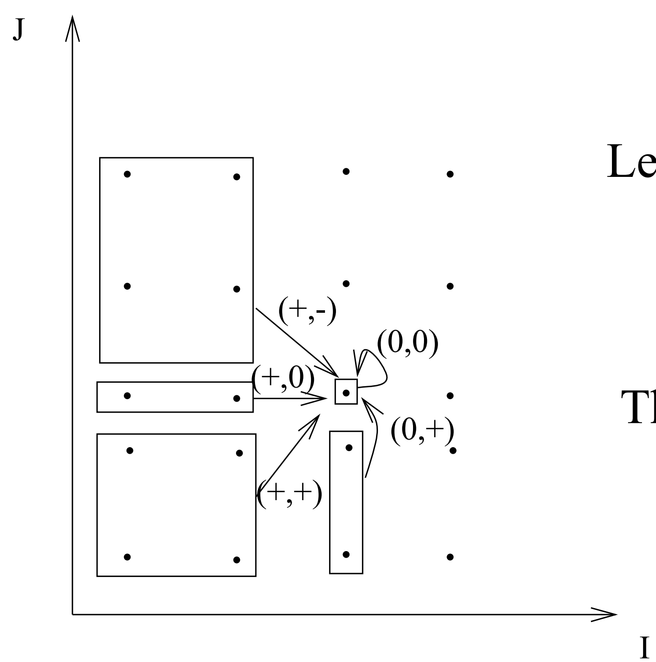
Directions for Nested Loops

Assume loop nest is (I,J).

If $(I_1, J_1) \rightarrow (I_2, J_2) \in$ dependence relation, then

$$\text{Distance} = (I_2 - I_1, J_2 - J_1)$$

$$\text{Direction} = (sign(I_2 - I_1), sign(J_2 - J_1))$$



Legal direction vectors:

(+,+) (0,+)
(+,-) (0,0)
(+,0)

The following direction vectors cannot exist:

(0,-) (-,+)
(-,0)
(-,-)

Valid dependence vectors are lexicographically positive.

How to compute Directions: Use IP engine

```
     DO 10 I = 1, 100
        X(f(I)) = ...
    10          = ...X(g(I))..
```

Focus on flow dependences:

$$f(I_w) = g(I_r)$$
$$1 \leq I_w \leq 100$$
$$1 \leq I_r \leq 100$$

First, use inequalities shown above to test if dependence exists in any direction (called (*) direction).

If IP engine says there are no solutions, no dependence.

Otherwise, determine the direction(s) of dependence.

Test for direction (+): add inequality $I_w < I_r$

Test for direction (0): add inequality $I_w = I_r$

In a single loop, direction (−) cannot occur.

Computing Directions: Nested Loops

Same idea as single loop: *hierarchical testing*

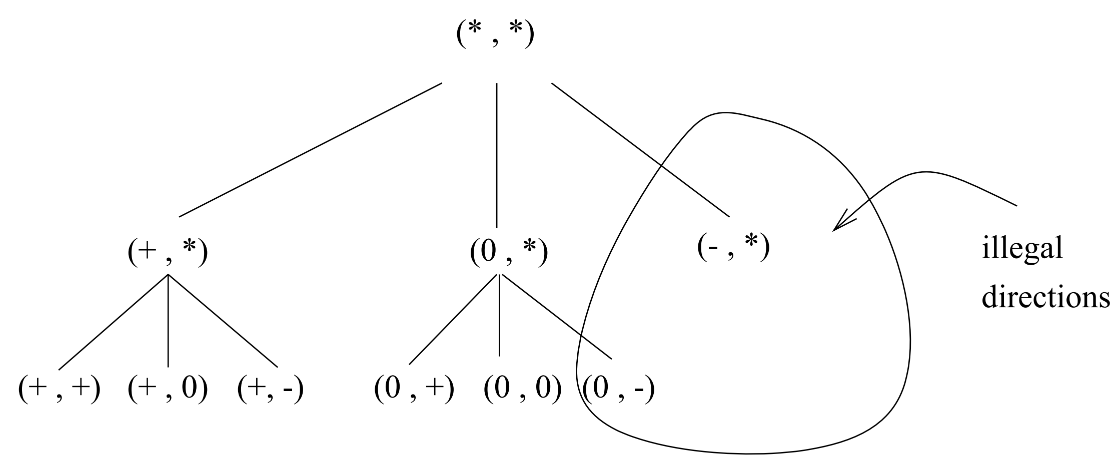


Figure 1: Hierarchical Testing for Nested Loop

Key ideas:

(1) Refine direction vectors top down.

(eg),no dependence in $(*, *)$ direction

$\Rightarrow$ no need to do more tests.

(2) Do not test for impossible directions like $(-, *)$.

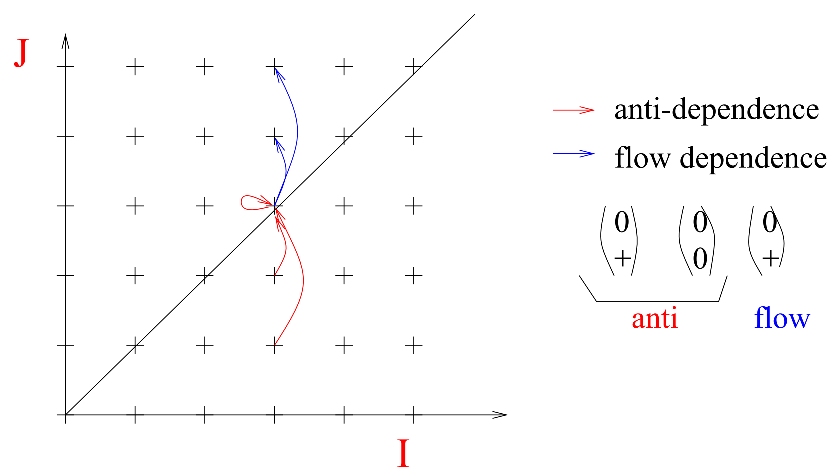
It is also possible to compute direction vectors by projecting on the variables in the Δ, the iteration difference vector.

Similar to what we did for distance vectors.

Left as an exercise for you.

Big hairy example: Compute dependences for following program:

```
DO I = 1,N
  DO J = 1,N
    X(I,J) = ...X(I,I)...
```



Linear system for anti-dependence:

$$I_w = I_r$$
$$J_w = I_r$$
$$1 \leq I_w, I_r, J_w, J_r \leq N$$
$$(I_r, J_r) \preceq (I_w, J_w)$$
$$\Delta 1 = (I_w - I_r)$$
$$\Delta 2 = (J_w - J_r)$$

Projecting onto $\Delta 1$ and $\Delta 2$, we get

$$\Delta 1 = 0$$
$$0 \leq \Delta 2 \leq (N - 1)$$

So directions for anti-dependence are

```
0   and 0
0       +
```

Similarly, you can compute direction for flow dependence

```
0
+
```

and also show that no output dependence exists.

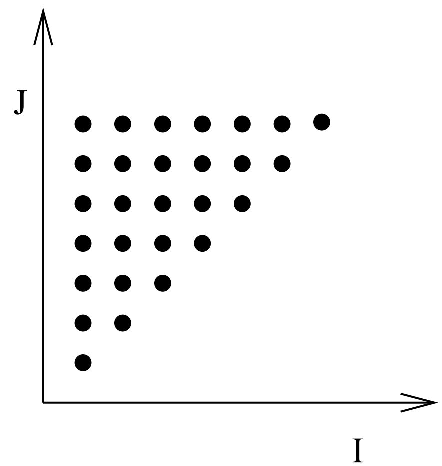
Dependence matrix for a loop nest

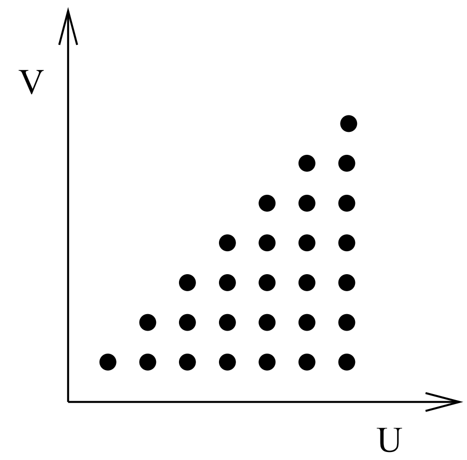
Matrix containing all dependence distance/direction vectors for all dependences of loop nest.

In our example, the dependence matrix is

```
0  0
0  +
```

Dependence direction/distance are adequate for testing legality of permutation.






```
DO I = 1, N
  DO J = I,N
  ..........
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix}$$

```
DO U = 1, N
  DO V = 1,U
  .........
```

$$\begin{bmatrix} I1 \\ J1 \end{bmatrix} \rightarrow \begin{bmatrix} I2 \\ J2 \end{bmatrix} \qquad T\begin{bmatrix} I1 \\ J1 \end{bmatrix} \quad T\begin{bmatrix} I2 \\ J2 \end{bmatrix}$$

Dependence distance $= \begin{bmatrix} I2 - I1 \\ J2 - J1 \end{bmatrix}$

Distance between iterations =

$$T\begin{bmatrix} I2 \\ J2 \end{bmatrix} - T\begin{bmatrix} I1 \\ J1 \end{bmatrix} = T\begin{bmatrix} I2 - I1 \\ J2 - J1 \end{bmatrix} = \begin{bmatrix} J2 - J1 \\ I2 - I1 \end{bmatrix}$$

Check for legality: interchange positions in distance/direction vector & check for lex +ve

If transformation P is legal and original dependence matrix is D, new dependence matrix is T*D.

Correctness of general permutation

Transformation matrix: T

Dependence matrix: D

Matrix in which each column is a distance/direction vector

Legality: $T.D \succ 0$

Dependence matrix of transformed program: $T.D$

Examples:

```
DO I = 1,N
  DO J = 1,N
   X(I,J) = X(I-1,J-1)....
```

Distance vector = (1,1) => permutation is legal

Dependence vector of transformed program = (1,1)

```
DO I = 1,N
  DO J = 1,N
   X(I,J) = X(I-1,J+1)....
```

Distance vector = (1,-1) => permutation is not legal

Remarks on dependence abstractions

A good dependence abstraction for a transformation should have the following properties.

- Easy to compute
- Easy to test for legality.
- Easy to determine dependence abstractions for transformed program.

Direction vectors are a good dependence abstraction for permutation.

Engineering a dependence analyzer

In principle, we can use IP engine to compute all directions.

Reality: most subscripts and loop bounds are simple!

Engineering a dependence analyzer:

First check for simple cases.

Call IP engine for more complex cases.

Conclusions

Traditional position: exact dependence testing (using IP engine) is too expensive

Recent experience:

(i) exact dependence testing is OK provided we first check for easy cases (ZIV,strong SIV, weak SIV)

(ii) IP engine is called for 3-4% of tests for direction vectors

(iii) Cost of exact dependence testing: 3-5% of compile time