Polynomial Loop Recognition in Traces

Alain Ketterlin

▶ To cite this version:

Alain Ketterlin. Polynomial Loop Recognition in Traces. IMPACT 2025 – 15th International Workshop on Polyhedral Compilation Techniques, Jan 2025, Barcelona, Spain. hal-04922050

HAL Id: hal-04922050

https://inria.hal.science/hal-04922050v1

Submitted on 30 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Polynomial Loop Recognition in Traces

(tool paper)

Alain Ketterlin
Inria/Camus, CNRS/Icube, Université de Strasbourg (France)
alain@unistra.fr

Abstract

This paper describes a tool named PLR which recognizes polynomial loop nests in traces. After providing background on affine (i.e., *polyhedral*) loop recognition with NLR, it exposes how a particular representation for integer polynomials leads to a simple and efficient interpolation technique. The new PLR algorithm, a slight modification of the original NLR, leverages this interpolation technique to recognize polynomial loops, i.e., loops where all bounds and values are expressed as multivariate polynomials in the loop counters.

1 Background on NLR

The *Nested Loop Recognition* (NLR) algorithm [8] takes a trace as input and outputs, whenever possible, an affine loop nest that produces that trace. Its input is made of a sequence of possibly tagged vectors of integers, like the one on the left part of this display:

```
A 1
B 2 3
B 6 60
[...]
A 5
B 66 3
[...]
```

```
for i = 0 to 17
  val A, 1 + 4*i
  for j = 0 to 42 + i
    val B, 2 + 64*i + 4*j, 3 + 57*j
```

Its output, shown on the right, is a loop nest where every loop bound and vector element is an affine combination of the enclosing loop counters.

NLR has been originally designed for trace compression and memory access prediction [8], and more generally to handle profiling data [2]. It has also been used for dynamic optimization [9], or to assist static analysis [3, 13]. Let us also mention the work of Rodríguez et al. [12], which has similar objectives and an interesting application to the optimization of sparse structure accesses [1]; however, their approach is significantly different, and we are unable to assess how much of what follows could apply to their algorithm.

1.1 The Algorithm

The NLR algorithm is similar to a shift-reduce parsing algorithm. It maintains an internal stack holding already recognized loops and raw vectors. Any incoming vector is pushed onto the stack, whose upper part is then searched for possible reductions. There are two possible reduction operations:

1. forming a *new loop* from three consecutive blocks of stack items, identified as its first three iterations;
2. recognizing a *new iteration* from an upper segment of the stack and the loop immediately underneath it.

These reductions are at the heart of the algorithm, and we will examine their details shortly. As soon as a reduction is possible, it is greedily applied: some stack items are removed, a new or modified loop is pushed, and the search for a reduction restarts with the updated stack. If no reduction is possible, NLR waits for the next input.

When it is looking for a reduction, NLR considers increasingly long segments at the high end of the stack: on each segment, applicable reductions are attempted. A parameter K bounds the search, by limiting the number of stack items in an iteration. Below is an illustration of this search with $K = 4$, where potential iterations are shown as boxes around stack items, and potentially extendable loops are shadowed:

length	stack (top on the right)	attempt
2		iter
3		loop
		iter
4		iter
5		iter
6		loop
9		loop
12		loop

Up to $3K$ stack items are examined in search for a new loop, and up to $K + 1$ in search for a new iteration. In pathological cases (e.g., random input) with N vectors, NLR will test $O(NK)$ segments but perform $O(NK^2)$ item comparisons, and find no reduction at all. In more favorable cases, the maximum stack size depends on the shape of the loops found, not on the size of the input trace, which is processed incrementally and not kept in memory.

1.2 Recognizing Loops and Iterations

While searching the stack, NLR has to assess the feasibility of various reductions. Stack items are either raw vectors or already recognized loops, i.e., essentially abstract syntax trees where internal nodes are loops and leaves are vectors. However, loops have bounds and vectors contain elements, both of which are affine combinations of the enclosing loop counters. Therefore, NLR uses a combination of *syntactic* and *numeric* criteria.

IMPACT 2025: 15th *Intl Workshop on Polyhedral Compilation Techniques* (in conjunction with HiPEAC 2025). January 22, 2025, Barcelona, Spain.

When NLR examines whether it can form a new loop from three consecutive blocks of stack items, it first tests whether the three blocks are isomorphic: this is the syntactic part. It then tests whether the numbers that appear as constants in the various affine functions form arithmetic progressions: this is the numeric part. If both tests succeed, NLR considers that the three blocks are three successive iterations of a loop.

For instance, consider the following three blocks of two stack items (shown here one item per line, with the last pushed item at the bottom):

```
      [...]
j=0 ⎡- val 25
    ⎣- for i = 0 to 15 { val 13 + 7i; }
j=1 ⎡- val 49
    ⎣- for i = 0 to 27 { val 19 + 7i; }
j=2 ⎡- val 73
    ⎣- for i = 0 to 39 { val 25 + 7i; }
```

Because the three blocks are isomorphic (modulo numbers), their common shape could be the shape of a loop body. And because the constants appearing in corresponding positions (shown on a gray background above) form arithmetic progressions, they can each be turned into an affine function of a new counter ranging from 0 to 2. NLR recognizes this segment as (and replaces it with) a new loop:

```
- for j = 0 to 3
 j ⎡val 25 + 24j
   ⎣for i = 0 to 15 + 12j { val 13 + 6j + 7i; }
```

Once isomorphism is established, NLR essentially performs interpolation on a set of slightly over-constrained series of integers: the first two blocks define the interpolation, while the third provides *some* guarantee that the interpolation is non-anecdotal. Note however that not all numbers are subject to interpolation: since NLR targets affine functions, numbers that are coefficients of a variable (like 7 in the example) are considered to be part of the syntactic structure, and must match exactly in all three blocks.

As much as forming a new loop is based on interpolation, recognizing a new iteration is based on extrapolation. For instance, if the previous loop is on the top the stack and, after 53 more inputs (and 49 reductions, one of which forms a new loop), the following items appear on top of it:

```
⎡- val 97
⎣- for i = 0 to 51 { val 31 + 7i; }
```

then at some point in its search, NLR will test whether these last two items actually represent a new iteration of the loop on j. To do this, it extrapolates the body of the loop for its next iteration (j=3), finds that the result matches the following block of items, removes these items from the stack, and increments the loop upper bound.

1.3 Corner Cases

The recognition of a new iteration of an existing loop is not as simple as we made it appear above, and complications may appear in, well, corners of iteration spaces. Consider for example a trace produced by the loop on the left:

```
for (x=0; x<5; x++)
  for (y=0; y<5-i; y++)
    ... 10*x+y ...
```



```
- for i=0 to 3
    for j=0 to 5-i
      val 10*i+j
- val 30
- val 31
- val 40
```

Assume that the first 3 iterations on x each give rise to a loop (on j) that NLR recognizes, and that these 3 loops are then themselves recognized as a new loop on i, as shown on the right. Unfortunately, the iteration i=3 does not produce enough terms to form a new loop on j, and leaves items on the stack that do not match the (extrapolated) preceding loop body. This would prevent NLR from extending its loop on i and cover the whole trace with a single loop.

NLR includes a coping mechanism for this situation: when it extrapolates the body of an existing loop in the hope of finding its next iteration in later stack items, it (virtually) unrolls any inner loop that would have too few iterations (i.e., the inner loop on j in the example), before comparing the result to the rest of the stack. It does the same immediately after forming of a new loop, extrapolating iteration −1 to gather potential previous iterations that did not fully match the loop body.

We do not describe these details here because they are probably of limited general interest and would take too much space (as they do in our implementation, where they take more lines that everything else).

2 Integer Polynomial Interpolation

NLR spends a lot of time on interpolation, which is easy and fast for affine functions. Adapting the algorithm to polynomial loops requires both a suitable representation for integer polynomials, and an efficient way to determine the coefficients of a polynomial given its successive values.

2.1 Integer Polynomials and Naive Interpolation

An integer polynomial is an integer-valued polynomial of integer variable(s). In another line of research [7], we have argued for an alternative representation of integer polynomials, using binomial powers:

$$x^{\underline{k}\rfloor} \triangleq \binom{x}{k} = \frac{x \cdot (x-1) \cdots (x-k+1)}{k!}$$

The binomial power $x^{\underline{k}\rfloor}$, defined for any integer x and non-negative exponent k, is simply a binomial coefficient in general form; the only purpose of the special notation is to simplify formulas below. An integer polynomial on indeterminate x is then defined as a combination of various binomial

powers of x with integer coefficients:

$$p(x) = a_0 + a_1 x^{\underline{1}} + \cdots + a_n x^{\underline{n}} \qquad (a_i \in \mathbb{Z},\ \forall i \in [0, n])$$

This representation is both correct (all such polynomials are integer-valued) and complete (all integer-valued polynomials admit such a representation), and also completely avoids the use of rational coefficients (e.g., $\frac{x(x-1)}{2}$ is just $x^{\underline{2}}$).

Given successive integer values $v_0, \ldots, v_n$, finding the interpolating polynomial $p(x) = a_0 + a_1 x^{\underline{1}} + \cdots + a_n x^{\underline{n}}$ such that $v_i = p(i)$ is easily done by writing down these equations:

$$\begin{aligned} v_0 &= p(0) = a_0 \\ v_1 &= p(1) = a_0 + a_1 \cdot 1^{\underline{1}} \\ v_2 &= p(2) = a_0 + a_1 \cdot 2^{\underline{1}} + a_2 \cdot 2^{\underline{2}} \\ &\ldots \end{aligned}$$

This system is triangular by construction (because $x^{\underline{k}} = 0$ whenever $k > x \geq 0$), and every equation introduces a new unknown a_i with coefficient $i^{\underline{i}} = 1$. Therefore, the integer solution always exists, is unique, and has degree at most n. It can be calculated incrementally:

$$a_0 = v_0, \qquad a_i = v_i - \sum_{j=0}^{i-1} a_j \cdot i^{\underline{j}} \quad (0 < i \leq n)$$

or simultaneously:

$$a_i = \sum_{j=0}^{i} (-1)^{\underline{i-j}} \cdot i^{\underline{j}} \cdot v_j$$

The equivalence between these two forms is easily proved by induction.

2.2 Difference-Based Interpolation

These last equations make it easy to determine the coefficients of the polynomial interpolating a sequence of successive values. However, there is an even more efficient way to obtain the same result, based on the notion of finite difference, which is the discrete analog to the continuous derivative. For any integer function f of an integer variable x, its finite difference with respect to x is defined as:

$$\Delta f(x) = f(x+1) - f(x) \tag{1}$$

The d-th order finite difference $\Delta^{(d)} f$ is defined inductively:

$$\Delta^{(0)} f = f, \qquad \Delta^{(d+1)} f = \Delta\left(\Delta^{(d)} f\right) \quad (d \geq 0) \tag{2}$$

Binomial powers have especially simple expressions of their finite differences:

$$\Delta x^{\underline{k+1}} = x^{\underline{k}}$$

This is a simple variation on the addition rule for binomial coefficients, $(x+1)^{\underline{k+1}} = x^{\underline{k}} + x^{\underline{k+1}}$, which also governs the construction of Pascal's triangle. The finite difference of an integer polynomial is:

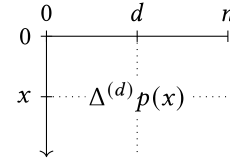
$$\Delta(a_0 + a_1 x^{\underline{1}} + \cdots + a_n x^{\underline{n}}) = a_1 + \cdots + a_n x^{\underline{n-1}}$$

which amounts to "shifting the coefficients"; more generally:

$$\Delta^{(d)} \left(\sum_{i=0}^{n} a_i \cdot x^{\underline{i}} \right) = \sum_{i=d}^{n} a_i \cdot x^{\underline{i-d}} \tag{3}$$

whose immediate consequence is a distinctive property of the use of binomial powers: $\Delta^{(d)} p(0) = a_d$, i.e., the coefficients *are* the values of the successive finite differences at 0. Note also that polynomials have only a finite number of not uniformly zero finite differences: for all $d > n$, $\Delta^{(d)} p = 0$.

That is a lot of definitions and formalism, but it broadens the perspective on interpolation by adding a new dimension, namely the finite difference order. The idea is to consider values and coefficients at all orders simultaneously, as a 2-dimensional discrete space, with axes along x and d:



The border conditions are $\Delta^{(0)} p(x) = v_x$ and $\Delta^{(d)} p(0) = a_d$, as per equations (2) and (3) respectively. It turns out that this space captures a fundamental symmetry between the values $v_0, \ldots, v_n$ and the coefficients $a_0, \ldots, a_n$ of a polynomial.

First, we can write the definition of the finite difference of p for any x at an arbitrary order d (that is, Equation (1) with $f = \Delta^{(d)} p$), in two different ways, depending on which axis the computation moves along:

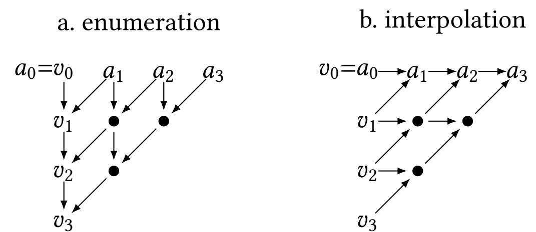
$$\Delta^{(d)} p(x+1) = \Delta^{(d)} p(x) + \Delta^{(d+1)} p(x) \tag{4a}$$

$$\Delta^{(d+1)} p(x) = \Delta^{(d)} p(x+1) - \Delta^{(d)} p(x) \tag{4b}$$

Now, the symmetry is between:

a. *enumeration*: computing the values $v_0, \ldots, v_n$, from the coefficients $a_0, \ldots, a_n$, using Equation (4a), and
b. *interpolation*: computing the coefficients $a_0, \ldots, a_n$, from the values $v_0, \ldots, v_n$, using Equation (4b);

Here is a graphical representation of each computation with $n = 3$, as far as $a_0, \ldots, a_n$ and $v_0, \ldots, v_n$ are involved:



These simple schemes immediately lead to implementation. For instance, here are code fragments for *in place* enumeration and interpolation in an array containing $n + 1$ integers (i.e., either coefficients or values). The reader is advised to look for the difference between these two fragments, because there is only one.

```
// a. enumeration                // b. interpolation
// t is [a0, . . . ,an]          // t is [v0, . . . ,vn]
for (i=1; i<=n; i++)             for (i=1; i<=n; i++)
    for (j=n; j>=i; j--)             for (j=n; j>=i; j--)
        t[j] += t[j-1];                  t[j] -= t[j-1];
// t is [v0, . . . ,vn]          // t is [a0, . . . ,an]
```

Both fragments perform exactly $(n+1)^{\underline{2}}$ additions or subtractions, and no multiplication. This matters if values or coefficients are, for instance, arbitrary precision integers, or even more complex mathematical objects like, wait for it, polynomials in a multivariate setting.

Finally, for those who wonder, there *is* a third way to write the finite difference equation:

$$\Delta^{(d)}p(x) = \Delta^{(d)}p(x+1) - \Delta^{(d+1)}p(x) \qquad (4c)$$

One use for this variant is "backward shifting", which becomes "backward enumeration" when iterated: if the array `t` contains the coefficients of $p(x)$, then, after

```
for (j=n-1; j>=0; j--)
    t[j] -= t[j+1];
```

the array contains the coefficients of $q(x) = p(x-1)$. This code is used to update a newly formed loop that gathers one previous iteration (see Section 1.3).

3 Polynomial Loop Recognition

With an adequate representation of integer polynomials, and a suitable interpolation technique, the loop recognition algorithm can be adjusted. This section looks at these adjustments, and describes a new algorithm called PLR.

3.1 Polynomial Loops

We can now have a more detailed look at polynomial loops, and describe how they can be recognized in traces. Essentially, polynomial loops have polynomials wherever affine loops have affine combinations, which means as loop bounds and vector elements. Since loops can be nested to an arbitrary depth, all these polynomials are multivariate, involving all loop counters in scope where they appear. There are two aspects to multivariate polynomials, which we examine separately even though they often appear simultaneously.

First, a multivariate polynomial can be "simply" non-linear, and include simple products of variables. Such functions appear frequently, for instance, in address calculations for cells of multi-dimensional arrays. The loop forming mechanism described in Section 1.2 is easily extended by considering all numbers appearing in a stack item, including those that act as coefficients of existing variables. Here is a trimmed down and slightly modified version of the example used in Section 1.2, where the coefficient of `i` varies:

```
j=0 [- for i = 0 to ... { val 13 + 5i; }
j=1 [- for i = 0 to ... { val 19 + 7i; }
j=2 [- for i = 0 to ... { val 25 + 9i; }

        [...] { val 13 + 6j + (5+2j)i; }
```

The coefficient of `i` is now subject to interpolation, and will be turned into a function of the newly introduced loop counter. The net effect in this example is the appearance of the non-linear term 2·j·i.

Second, the very notion of polynomial functions allows the presence of various powers of a variable, which, as we have seen in the previous section, have to be understood as binomial powers. Such integer polynomials appear, for instance, in all problems that involve counting or ranking individual instructions. The loop formation mechanism can be extended to recognize higher degree polynomial progressions, provided enough stack items are available: successive values are then interpolated with the technique described in Section 2.2. Here is the same example again, modified and extended to exhibit a degree-2 polynomial progression:

```
j=0 [- for i = 0 to ... { val 13 + 7i; }
j=1 [- for i = 0 to ... { val 19 + 7i; }
j=2 [- for i = 0 to ... { val 27 + 7i; }
j=3 [- for i = 0 to ... { val 37 + 7i; }

      [...] { val 13 + 6j^1 + 2j^2 + 7i; }
```

13 | 6 | 2 | 0
19 | 8 | 2
27 | 10
37

The resulting polynomial of degree 2 is derived from 4 stack items, where 3 would suffice; the algorithm uses interpolation as part of its search for regularity, not just to represent *any* sequence of blocks of stack items. The general rule is the following: PLR will form a new loop whenever it can interpolate $n+2$ isomorphic blocks with polynomials of degree at most n (that is, represented with $n+1$ coefficients). Note that this rule covers the one used by NLR, which requires 3 values to assert an affine function (a polynomial of degree 1). It also implements a *description size* minimization heuristics: if we consider the size of a block to be the overall number of polynomial coefficients it contains, then the interpolated loop must contain fewer coefficients that the blocks it replaces.

We have focused mainly on new loop formation with polynomials functions. PLR also reduces its stack when it finds a new iteration of an existing loop, including non trivial extrapolation as explained in Section 1.3. This mechanism requires no significant modification when moving to polynomial loops.

3.2 Updating the Search Strategy

PLR uses the same search strategy as NLR, examining increasingly longer segments at the top of its internal stack. Like NLR, it limits this search with the help of the parameter K, which is the maximal length of a loop body. However, letting PLR freely attempt interpolation with higher degree polynomials on longer segments would add a new source of unbounded complexity. Therefore PLR introduces a new parameter D, which is the highest degree at which interpolation will be attempted. All degrees d from 0 to D are considered, as long as the current segment length is a multiple of $d+2$.

Having two distinct parameters raises the question of their interaction. We see no clear reason why one should be more important than the other: K is clearly bounding syntactic complexity, while D is meant to bound numerical complexity. Therefore, our current implementation uses a naive approach; here is how it enumerates its attempts:

for every segment length ℓ
  for every degree d between 0 and D
    if $d+2$ evenly divides ℓ and $\frac{\ell}{d+2} \le K$
      attempt to form a new loop

For each attempt, if the $d+2$ blocks of size $\frac{\ell}{d+2}$ are isomorphic, PLR will try to interpolate then with polynomials of degree d. Attempting to recognize new iterations of existing loops is unaffected by D, and happens for all segment lengths up to $K+1$. The following table lists some of these attempts when $K=4$ and $D=3$:

length	stack (top on the right)	attempt
2		loop ($d=0$)
		iter
3		loop ($d=1$)
		iter
4		loop ($d=0$)
		loop ($d=2$)
		iter
5		loop ($d=3$)
		iter
6		loop ($d=0$)
		loop ($d=1$)
[...]		
15		loop ($d=3$)
16		loop ($d=2$)
20		loop ($d=3$)

PLR is as greedy as NLR, and will apply the first valid reduction it finds. For *a given block size*, forming a new loop with polynomials of degree d is only considered if attempts with lower degrees have failed. Similarly, for *a given degree*, forming a loop with blocks of size b is only considered if attempts with shorter blocks have failed. However, PLR will for instance consider a loop with blocks of size 3 and degree 0 before a loop with blocks of size 2 and degree 1.

4 Examples

All core mechanisms of PLR have been described. We can now move to providing some illustration of its abilities.

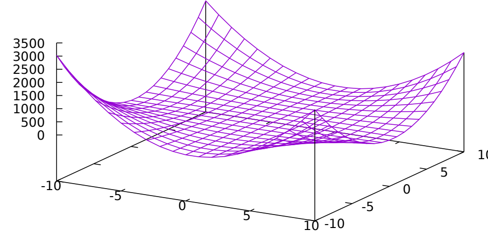
4.1 Toy Examples

Our first examples are artificially designed with the sole goal of illustrating the output of PLR. Here is a first one, whose output is exactly the loop that we used to generate the trace:

```
for i0 = 0 to 10
  val 7 + 3*i0 + 5*i0~2
  for i1 = 0 to 8 + 1*i0~2
    val 3 + 35*i0~2 + 11*i1 + 5*i0~2*i1 + 7*i0*i1~2
```

Loop counters are named according to their depth in the resulting loop nest (`i0`, `i1`, ...) and binomial powers are noted with the ~ operator. Note also that this loop has 10 iterations; loop upper bounds are always excluded.

Our next example is the values of $x^{\underline{2}|}y^{\underline{2}|}$, with $(x,y) \in [-10,10] \times [-10,10]$. Here is a plot of this function:



The trace is generated by scanning the domain along x and y. PLR outputs the following loop, which we have edited to make it fit the width of this column:

```
for i0 = 0 to 21
  for i1 = 0 to 21
    val 3025 - 550*i0 + 55*i0~2
        - 550*i1 + 100*i0*i1 - 10*i0~2*i1
        + 55*i1~2 - 10*i0*i1~2 + 1*i0~2*i1~2
```

This surprisingly copious result is actually exactly the expansion of $(\mathtt{i0}-10)^{\underline{2}|}(\mathtt{i1}-10)^{\underline{2}|}$. PLR always produces normalized loops, with lower bound zero and step one.

4.2 Array Memory Accesses

Our next example uses a trace of memory addresses. Here is the setting. Assume we have a kernel that has been instrumented to produce a trace of all addresses it accesses, and that uses a pre-allocated block of memory. Calling this kernel requires 3 parameters N, M, and P. Running the kernel with $N=10$, $M=15$, $P=20$ and passing the trace to PLR (or NLR) produces the following loop:

```
for i0 = 0 to 10
  for i1 = 0 to 20
    for i2 = 0 to 15
      val 0x560edc3692a0 + 120*i0 + 8*i2
      val 0x560edc3699b0 + 8*i1 + 160*i2
    val 0x560edc36a0c0 + 160*i0 + 8*i1
```

While this seems to show the effect of the parameters on loop bounds, it is not clear how they affect the memory accesses. Rather than playing a guessing game, we can run the same kernel across a sample of the parameter space, collect a concatenated trace, and pass it to PLR. Here is pseudo-code to produce a trace for this experiment, using the same arbitrary range of values for all parameters:

```
for (N=10; N<15; N++)
    for (M=10; M<15; M++)
        for (P=10; P<15; P++)
            kernel (N, M, P, ...);
```

The idea is to let PLR find regularities and interpolate across the part of the parameter space so explored. Here is its output, with base addresses elided for space reasons:

```
for i0 = 0 to 256
  val S1 , 0x7ffec37b92a0 + 257*i0 ,                         // tag , memory address
      767*i0 + 506*i0^2 - 4*i0^3 , 1*i0                      // global rank , local rank
  for i1 = 0 to 1*i0
    val S2 , 0x7ffec37b92a0 + 256*i0 + 1*i1 ,
        1 + 767*i0 + 506*i0^2 - 4*i0^3 + 1*i1 , 1*i0^2 + 1*i1
  val S3 , 0x7ffec38392b0 + 1*i0 ,
      1 + 768*i0 + 506*i0^2 - 4*i0^3 , 1*i0
  for i1 = 0 to 255 - 1*i0
    val S4 , 0x7ffec37b92a1 + 257*i0 + 1*i1 ,
        2 + 768*i0 + 506*i0^2 - 4*i0^3 + 3*i1 + 2*i0*i1 , 255*i0 - 1*i0^2 + 1*i1
    for i2 = 0 to 1*i0
      val S5a , 0x7ffec37b93a0 + 256*i0 + 256*i1 + 1*i2 ,
          3 + 768*i0 + 506*i0^2 - 4*i0^3 + 3*i1 + 2*i0*i1 + 2*i2 , 254*i0^2 - 2*i0^3 + 1*i0*i1 + 1*i2
      val S5b , 0x7ffec37b92a0 + 256*i0 + 1*i2 ,
          4 + 768*i0 + 506*i0^2 - 4*i0^3 + 3*i1 + 2*i0*i1 + 2*i2 , 254*i0^2 - 2*i0^3 + 1*i0*i1 + 1*i2
    val S6a , 0x7ffec38392b0 + 1*i0 ,
        3 + 770*i0 + 506*i0^2 - 4*i0^3 + 3*i1 + 2*i0*i1 , 255*i0 - 1*i0^2 + 1*i1
    val S6b , 0x7ffec37b93a0 + 257*i0 + 256*i1 ,
        4 + 770*i0 + 506*i0^2 - 4*i0^3 + 3*i1 + 2*i0*i1 , 255*i0 - 1*i0^2 + 1*i1
```

Figure 1. PLR output for the Cholesky kernel trace including instruction ranks.

```
for i0 = 0 to 5
  for i1 = 0 to 5
    for i2 = 0 to 5
      for i3 = 0 to 10 + 1*i0
        for i4 = 0 to 10 + 1*i2
          for i5 = 0 to 10 + 1*i1
            val 0x[...]92a0 + 80*i3 + 8*i1*i3 + 8*i5
            val 0x[...]99b0 + 8*i4 + 80*i5 + 8*i2*i5
          val 0x[...]a0c0 + 80*i3 + 8*i2*i3 + 8*i4
```

PLR has correctly recognized the parameter space (over `i0`, `i1`, and `i2`); then, three innermost loops constitute a single-execution model like the one shown earlier. However, this model is now parameterized by `i0`, `i1`, and `i2`, which act as surrogates for N, M, and P, and addresses have non-linear expressions. Omitting the base address and the common 8 factor, these expressions can be understood as:

address	major index range	minor index range
`i3*(i1+10)+i5`	`i3 ∈ [0,i0+10)`	`i5 ∈ [0,i1+10)`
`i5*(i2+10)+i4`	`i5 ∈ [0,i1+10)`	`i4 ∈ [0,i2+10)`
`i3*(i2+10)+i4`	`i3 ∈ [0,i0+10)`	`i4 ∈ [0,i2+10)`

The non-linear terms appearing in the address expressions hint at 2-dimensional arrays, and the ranges of the suspected indices seem to confirm the hypothesis that memory accesses are made inside three arrays of respective sizes:

`(i0+10)×(i1+10)`, `(i1+10)×(i2+10)`, and `(i0+10)×(i2+10)`

which involve only parameters and could be rewritten as $N \times M$, $M \times P$ and $N \times P$. (The kernel is what you think it is.)

The analysis we have done manually and informally here is called *array delinearization*. It was first introduced and formalized by Maslov [10]; see also Grosser et al. [6] for a contemporary treatment in a polyhedral setting.

4.3 Instruction Ranks

One theoretical development of the polyhedral model that requires the use of polynomials is counting the number of integer points inside an arbitrary polyhedron. Two distinct approaches have been developed [4, 14], and the related problem of ranking instructions in a loop nest has enabled new applications [5]. Our goal in this section is to evaluate the ability of PLR to determine ranking polynomials while recognizing loops in traces.

This experiment uses the Cholesky benchmark kernel from the `polybench` suite version 3 [11]. Below is the source loop, with labels on instructions:

```
for (i=0; i<n; ++i) {
S1:    x = A[i][i];
       for (j=0; j<=i-1; ++j)
S2:        x -= A[i][j] * A[i][j];
S3:    p[i] = 1.0 / sqrt(x);
       for (j=i+1; j<n; ++j) {
S4:        x = A[i][j];
           for (k=0; k<=i-1; ++k)
S5:            x -= A[j][k] * A[i][k];
S6:        A[j][i] = x * p[i];
}      }
```

This kernel was instrumented to produce a trace of all array accesses it performs. Every address traced is tagged with the corresponding instruction label, with `a` or `b` appended when an instruction has two distinct accesses.

A complete trace was produced with $n = 256$, containing $2n + 4n^{\underline{2}|} + 2n^{\underline{3}|} = 5{,}658{,}112$ entries. After the trace is produced, each entry receives two additional fields: the first is a sequential entry number (its global rank), the second is a per-access sequential number (its "local" rank). The first seven entries are:

```
S1  0x7ffec37b92a0 0 0      S6b 0x7ffec37b93a0 4 0
S3  0x7ffec38392b0 1 0      S4  0x7ffec37b92a2 5 1
S4  0x7ffec37b92a1 2 0      S6a 0x7ffec38392b0 6 1
S6a 0x7ffec38392b0 3 0      ...
```

PLR takes about 5 seconds on a recent laptop to output the loop shown in Figure 1. Note that ranks are represented with polynomials of degrees up to 3; we have verified that these polynomials conform to their statistically computed versions [7]. PLR obtains an identical loop when certain fields are omitted, as long as one of the first two fields is present (for instance, with only the address and the global rank). We consider this to be an excellent result.

Less stellar is the result when PLR is given a trace containing only the last two fields (the global and local ranks). Its abridged output is:

```
for i0 = 0 to 5
  val 0 , 1*i0
for i0 = 0 to 254
  for i1 = 0 to 3
    val 1 + 1*i0 , 5 + 3*i0 + 1*i1
val 1 , 767
[...]
for i0 = 0 to 252
  ... (same loop as in Figure 1) ...
```

There is good news and bad news in this result. The good news is that it ends with a loop capturing 252 out of 256 iterations. This is surprising given that the only information present in the trace is ranking information; it means that a pair of ranking polynomials somehow represents the loop structure, and that PLR is able to extract this structure.

The bad news however is that the final loop is preceded by about 35 stack items representing random anecdotal regularities, as the ones shown above. PLR, being greedy, was mislead early on, and lost 4 full iterations before falling back on the right track. But the early (anecdotal) loops are not simply fragments of the "real" loop (the coping mechanism of Section 1.3 would have corrected this), but are overlapping and conflicting regularities that are more complex to distinguish. We leave such problems for future investigation.

5 Final Remarks

A preliminary implementation of PLR is available from the author; this implementation has been used to produce all examples in this paper. Both the tool and the use of polynomials to model traces are fairly new, with very little experimental evaluation or even relevance; we are welcoming any potential application that could make use of them.

Even though the first results presented here look promising, the representative power of polynomials makes it easy to imagine scenarios where the current search strategy fails to recognize regular behavior. The current approach, essentially waiting for "big loops" to appear and then collecting smaller pieces (see Section 1.3), has already been shown to fail in some cases (see the end of the previous section). We do not yet have a clear view on how fallible it is.

Finally, loop nests with polynomial bounds are notably absent from our experiments, with one insignificant exception (the very first toy example in Section 4.1). We do not know of any computation kernel using polynomial loop bounds, or even any application where some phenomenon is studied on a domain bounded by polynomials. Therefore, we may very well have a tentative solution to a problem nobody has. Nevertheless, polynomial loops *are* an obvious extension of affine loops, and whether their study as an extension to the polyhedral model will advance program analysis and optimization remains to be seen.

References

[1] T. Augustine, J. Sarma, L.-N. Pouchet, and G. Rodríguez. 2019. Generating piecewise-regular code from irregular structures. In *PLDI '19*. doi: 10.1145/3314221.3314615

[2] D. Barthou, A. Charif Rubial, W. Jalby, S. Koliai, and C. Valensi. 2010. Performance Tuning of x86 OpenMP Codes with MAQAO. In *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel (Eds.).

[3] C. Bastoul, A. Ketterlin, and V. Loechner. 2023. Superloop Scheduling: Loop Optimization via Direct Statement Instance Reordering. In *13th International Workshop on Polyhedral Compilation Techniques (IMPACT 2023, in conjunction with HiPEAC 2023)*. https://impact-workshop.org/impact2023/#bastoul23-superloop

[4] P. Clauss. 1996. Counting Solutions to Linear and Nonlinear Constraints through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs. In *ICS '96*. doi: 10.1145/237578.237617

[5] P. Clauss, E. Altintas, and M. Kuhn. 2017. Automatic Collapsing of Non-Rectangular Loops. In *IPDPS '17*. doi: 10.1109/IPDPS.2017.34

[6] T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop. 2015. Optimistic Delinearization of Parametrically Sized Arrays. In *ICS '15*. doi: 10.1145/2751205.2751248

[7] A. Ketterlin. 2024. Easy Counting and Ranking for Simple Loops. In *14th International Workshop on Polyhedral Compilation Techniques (IMPACT 2024, in conjunction with HiPEAC 2024)*. https://impact-workshop.org/impact2024/#ketterlin24-counting

[8] A. Ketterlin and P. Clauss. 2008. Prediction and trace compression of data access addresses through nested loop recognition. In *CGO '08*. doi: 10.1145/1356058.1356071

[9] S. Kobeissi, A. Ketterlin, and P. Clauss. 2020. Rec2Poly: Converting Recursions to Polyhedral Optimized Loops Using an Inspector-Executor Strategy. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, A. Orailoglu, M. Jung, and M. Reichenbach (Eds.).

[10] V. Maslov. 1992. Delinearization: an efficient way to break multiloop dependence equations. In *PLDI '92*. doi: 10.1145/143095.143130

[11] L. N. Pouchet and T. Yuki. 2023. *PolyBench/C*. Retrieved September 1, 2023 from https://sourceforge.net/projects/polybench/

[12] G. Rodríguez, J. M. Andión, M. T. Kandemir, and J. Touriño. 2016. Trace-based affine reconstruction of codes. In *CGO '16*. doi: 10.1145/2854038.2854056

[13] H. Thievenaz, K. Kimura, and C. Alias. 2022. Lightweight Array Contraction by Trace-Based Polyhedral Analysis. In *High Performance Computing. ISC High Performance 2022 International Workshops*, H. Anzt, A. Bienz, P. Luszczek, and M. Baboulin (Eds.).

[14] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* 48 (05 2007), 37–66. doi: 10.1007/s00453-006-1231-0