情報処理学会論文誌　プログラミング　Vol.14　No.5　1–14 (Nov. 2021)

配列集約ループの実行時情報を用いた漸増化による効率化

松田 知樹[1]　森畑 明昌[1,a)]

受付日 2021年3月21日, 採録日 2021年8月2日

概要：配列の要素を集約し，総和・最大値・平均値などを求める処理を繰り返す場合，各繰り返しでの集約結果を再利用することで効率が改善することが多い．Liu ら（ACM TOPLAS，2005）はこの効率化を自動的に行う手法を提案した．しかし，Liu らの手法は効率的なコードの生成を制約ソルバによる論理式の単純化に頼っており，どのようなプログラムであれば効率化が可能なのか判然としなかった．これは，プログラムの些細な変更で効率化の成否が変わりかねないため，看過できる問題ではない．本論文では，Liu らの手法をベースに，制約ソルバを用いずに同様の効率化を行う手法を提案する．主要な着想は次の 3 点である．まず，効率化結果が長方形状の領域の処理の組合せになると仮定することで，論理式の単純化を回避すること．次に，実際にループを実行して得られた配列要素のアクセス履歴を用いることで，論理式を用いたプログラム解析を避けること．そして，自然に書かれたプログラムはその計算の規則性をよく表していると仮定することで，具体的なアクセス履歴からその一般的な法則性を現実的なコストで推測することである．この手法は，制約ソルバの使用を避けられることに加え，プログラムの構文的な細部に効率化の成否が依存しにくい，効率化結果の計算量を自動的に求めることができる，という長所がある．一方で，この手法には効率化結果の正しさが保証されないという欠点もある．以上をふまえた提案手法の現実的な利用方法についても論じる．

キーワード：配列集約，漸増化，実行時情報

Optimizing Array Aggregating Loops by Incrementalization using Runtime Information

Tomoki Matsuda[1]　Akimasa Morihata[1,a)]

Received: March 21, 2021, Accepted: August 2, 2021

Abstract: When a loop repeatedly summarizes array elements to calculate the total, the maximum, the average, etc., reuse of previously calculated summaries often improves efficiency. Liu et al. (ACM TOPLAS, 2005) proposed a method of automatically performing such optimization. Its drawback is the unpredictability of the result caused by the reliance on logical formula simplification using a constraint solver. This drawback is not negligible because a minor modification of the program may lead to unexpected failure. This paper proposes a method that performs optimization similar to theirs without using constraint solvers. It consists of three key ideas. First, it avoids simplifications of logical formulae by assuming that the optimized code consists of iterations over rectangular regions. Second, it uses concrete execution traces instead of logical formulae. Third, it infers invariants from the execution traces at realistic costs by postulating naturally written programs to represent the invariant. The proposed method uses no constraint solvers. Moreover, it is more agnostic to syntactic code variations and can automatically estimate the computational complexity of the optimized code. Unfortunately, the optimized code is not guaranteed to be correct. We discuss two practical use-case scenarios for which the correctness issue is less problematic.

Keywords: array aggregation, incrementalization, runtime information

1. はじめに

以下のループ（以降プログラム A と呼ぶ）は，各 `i` に対し，二次元配列 `a` の `a[0][0]` から `a[i-1][i-1]` まで

[1] 東京大学大学院総合文化研究科
Graduate School of Arts and Sciences, The University of Tokyo, Tokyo 153–8902, Japan
[a)] morihata@graco.c.u-tokyo.ac.jp

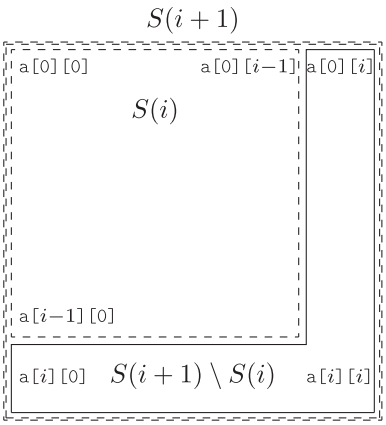

図 1　漸増化のアイデア
Fig. 1　The idea of the incrementalization.

の $\mathtt{i}^2$ 個の要素の総和を配列 b に格納するものである．ただし，各 b[i] は 0 で初期化されているものとする．

```
for i in range(N):
  for j in range(i):
    for k in range(i):
      b[i] += a[j][k]
```

単純で分かりやすいプログラムだが，この実行には $\mathrm{O}(\mathtt{N}^3)$ もの時間が必要になる．以下のプログラム（以降プログラム B と呼ぶ）は同じ結果を $\mathrm{O}(\mathtt{N}^2)$ 時間で求めることができる．

```
for i in range(1, N):
  b[i] = b[i - 1]
  for j in range(i):
    b[i] += a[j][i - 1]
  for k in range(i - 1):
    b[i] += a[i - 1][k]
```

プログラム A とプログラム B の概要を**図 1** に示す．これは各ループの繰り返しでアクセスされる a の要素を図示したものであり，左上隅が a[0][0] に対応し，たとえば a[n][m] は下に n 右に m の場所に対応する．$S(i)$ はプログラム A の i 回目の繰り返しでアクセスされる範囲，$S(i+1)$ は $i+1$ 回目でアクセスされる範囲である．図から明らかなように，$S(i)$ と $S(i+1)$ は大部分が共通している．そこで，プログラム B では，両者の差の部分，すなわち $S(i+1) \setminus S(i)$ と記載された逆 L 字型の部分，を追加することで，i 回目のループの繰り返しの結果（すなわち b[i]）から $i+1$ 回目のループの繰り返しの結果（すなわち b[i+1]）を求めている．

既知の計算結果を用いることで，少しだけ異なる計算の結果を高速に求める技法を一般に**漸増計算**と呼ぶ．また，漸増計算を前提としないプログラムを，漸増計算を用いたプログラムへと変換する技法を**漸増化**と呼ぶ．プログラム B はプログラム A を漸増化を用いて効率化したものだといえる．

総和・最大値などの演算を用いて配列の要素を集約する計算は，行列処理や画像処理などをはじめとして広く現れる．その多くでは，前述のような漸増化を用いた効率化が有益である．しかし，それらは具体的な場合ごとに，集約対象となる配列の次元や集約の範囲，また集約に用いる演算などが微妙に異なる．そのそれぞれについて手作業で効率化を行うのは，プログラムに誤りをもたらしやすく，また後の修正なども難しくなる．そのため，自明に記述した単純なプログラムに対して自動的に漸増化が行われることが望ましい．

Liu ら [24] は配列集約ループの自動漸増化を行う手法を示した．この手法は，図 1 の $S(i)$ や $S(i+1)$ が満たす条件を論理式として表現し，それらの差を表す論理式を導出し，それに基づき効率の良いループを導出するものである．たとえば，プログラム A であれば，

$$S(i) = \{\mathtt{a}[j][k] \mid 0 \le j \le i-1, 0 \le k \le i-1\}$$

という論理式を導出し，これをもとに以下のように差分を計算する．

$$\begin{aligned}
&S(i+1) \setminus S(i) \\
&= \{\mathtt{a}[j][k] \mid 0 \le j \le (i+1)-1, 0 \le k \le (i+1)-1\} \\
&\quad \setminus \{\mathtt{a}[j][k] \mid 0 \le j \le i-1, 0 \le k \le i-1\} \\
&= \{\mathtt{a}[j][k] \mid 0 \le j \le i, 0 \le k \le i, i-1 < j \vee i-1 < k\} \\
&= \{\mathtt{a}[j][k] \mid 0 \le j \le i\} \cup \{(i,k) \mid 0 \le k \le i-1\}
\end{aligned}$$

この差分の計算の自動化には，既存の制約ソルバを用いる．このようにして得られた差分を追加・削除するループを構成することで，プログラム B を得るという方針である．

しかし Liu らの手法には，制約ソルバの出力する論理式に効率化の成否が依存するという問題点がある．たとえば，上述の $S(i+1) \setminus S(i)$ の導出において，最後の式変形をしなかった場合には，以下のループが出力されるのが自然であろう．

```
for i in range(1, N):
  b[i] = b[i - 1]
  for j in range(i):
    for k in range(i):
      if i - 1 < j or i - i < k:
        b[i] += a[j][k]
```

このループは概念的にはプログラム B と同様の計算を行っており，加算の回数も減少している．しかし，実際のところ，これの計算量は $\mathrm{O}(\mathtt{N}^3)$ であり，プログラム A から改善していない．このように，Liu らの手法では「制約ソルバが十分に単純な式を出力し，それが効率的なループに変換できること」が前提となっている．一般に，制約ソルバーによる式の単純化は非常に複雑な処理を含んでおり，その結果がどのようなものになるかを予想するのは不可能に近

い．最悪の場合，プログラムの些細な変更に効率化の成否が影響されかねない．

Liu らの手法の欠点を解消するため，本研究では，制約ソルバに依存しない形で Liu らと同様の漸増化による効率化を行う手法を提案する．提案手法は以下の 3 つの着想に基づいている．

- 差分の計算を効率的なループに変換するのではなく，効率的なループの合成で差分の計算を表す．
- 論理式を用いるのではなく，実行時情報から得られた具体的なアクセス履歴を収集し，「例によるプログラミング [16]」により効率的なプログラムを合成する．
- 自然に書かれたプログラムの動作はその計算の法則性をよく表していると仮定し，計算の構造をできる限り利用して効率化を行う．

Liu らの手法では，制約ソルバの出力する論理式を効率的なループに変換しようとする部分で無理が生じてしまった．しかし，効率的なループでは表せないような論理式を得ることは，そもそも効率化の観点からは無意味である．ならば，最初から「効率的なループで表すことができる計算」に対応するような論理式のみを考え，そのような論理式で $S(i+1) \setminus S(i)$ が表現できるかを考えるほうが建設的である．このアプローチを採用することで，現れる論理式の形が単純になり，制約ソルバー特有の複雑な処理の多くを避けることができる．

とはいえ，効率的なループで表せるような制約式のみを扱ったとしても，制約式の取扱いには一定の困難をともなう．そこで本研究では，$S(i)$ などに対応する論理式を扱うのではなく，実際の計算で得た具体的な添字アクセスの履歴を扱う．これにより，制約ソルバーの必要性を排除する．

具体的な集合からのプログラム生成を目指すとなると，$S(i)$ や $S(i+1)$ などの個々の集合そのものではなく，それらの間に成り立つ法則性，つまり i から $S(i)$ を求める式を発見する必要がある．このために，本研究では法則性の推測を「i を入力すると $S(i)$ を出力するプログラムの構築」ととらえ，「例によるプログラミング」の考え方を採用する．

最先端の「例によるプログラミング」技術を採用したとしても，法則性の推測は容易でない．法則性を発見するには「何と何が対応しているのか」を見つけ出すことが重要となる．たとえば，$S(i)$ から規則的に $S(i+1)$ が得られるとすると，$S(i)$ の多くの要素は $S(i+1)$ のどれかの要素に「対応している」ことになるが，対応の可能性は膨大にあり，闇雲な探索は現実的でない．この困難を軽減するため，本研究では，自然に書かれたプログラムの動作はその計算の法則性をよく表している，という洞察を用いる．いい換えると，自然にプログラムを書いた場合，そのプログラムはわざわざその計算の法則性を乱すような動作はしない，となる．この洞察に基づけば，$S(i)$ 中の比較的早期にアクセスされた要素に対応する要素は，$S(i+1)$ 中でも比較的早期にアクセスされるだろう，という予想を立てることができる．このような予想は，ヒューリスティクスではあるものの，考慮すべき可能性を劇的に減らし，現実的なコストでの法則性の発見を可能にする．

以上をまとめ，本研究では制約ソルバを用いずに $S(i+1) \setminus S(i)$ に対応するループを得る手法を提案する（3 章）．この手法の特徴は，プログラムの実際の実行履歴から $S(i)$ に関する一般的な法則性を推測する点にある．これにより，提案手法はプログラムの構文的な細部に効率化の成否が依存しにくくなっている一方，効率化の正しさが保証されないという欠点もある．とはいえ，この欠点が比較的致命的とならないユースケースはいくつか考えられる（4 章）．さらに，提案手法のコンセプト確認のために試作したシステムを用いた実験についても報告する（5 章）．

2. 配列集約ループの制約ソルバを用いた漸増化

本章では Liu ら [24] による配列集約ループの自動漸増化手法を紹介する．本研究もこの手法を前提としており，特に漸増化対象とするプログラムに対する仮定や漸増化の基本的な方針には共通部分が非常に大きい．

Liu らは計算過程で現れる中間的な結果を記憶する配列を導入することで漸増化の適用範囲を拡大する手法や，記憶する必要のない配列を検出し削除することで漸増化結果の効率を改善する手法も同時に提案している．これらは，本章で説明する自動漸増化手法と直交するものであるため，本論文では説明しない．なお，これらは本論文での提案手法と組み合わせることもできる．

なお，本論文を通して Python に似た記法をプログラムの記述に用いる．ただし，本論文で論じる技法は Python に限らずループを用いる標準的なプログラミング言語一般に用いることができる．そのこともあり，Python では「リスト」と呼ばれるような列状のデータ構造を「配列」と呼ぶことにする．

2.1 漸増化対象とするプログラム

ここでは以下の形式のループを漸増化の対象とする．なお，説明を簡単にするためネストのないループを考えている．

```
for i in range(e1):
  for j in range(e2(i)):
    v = v ⊕ a[f(i,j)]
```

ここで a は集約対象となる配列であり，以下**寄与配列**と呼ぶことにする．また，v は計算結果を保存するための変数であり，**累積変数**と呼ぶ．累積変数は，実際には配列であるかもしれないが，その添字が i のみに依存する限り，

いい換えると寄与配列 a の内容に依存しない限り，同様に扱うことができる．

本論文を通して，n 次元配列は必ず n 個の添字をともなうこと，つまり集約対象となるような値が参照されることを仮定する．そのため，n 次元配列の「添字」はスカラではなく n 個の非負整数の並んだベクトルとして扱う．

演算 $\oplus$ は寄与配列の内容を集約する演算である．どの演算が使用可能かどうかについては，プログラムの具体的な内容に応じて様々な制約がある．詳細は Liu ら [24] を参照してほしいが，本論文では以下の制約を考える．これは実用的に Liu らの手法が適用可能な場合の大部分を含む[*1]．

- $\oplus$ は結合則 $s\oplus(t\oplus u)=(s\oplus t)\oplus u$ と交換則 $s\oplus t=t\oplus s$ を満たす．
- 逆演算 $\ominus$ が存在し，$(s\oplus t)\ominus t=s$ を満たす．ただし減少差分集合（2.2 節参照）がつねに空集合である場合は逆演算は必要ない．

関数 f は，集約対象となる配列要素を特定する関数である．これはループカウンタ `i` や `j` に依存するかもしれないが，寄与配列 a や累積変数 v には依存しないものとする．

ループカウンタ `i` は式 e_1 で，`j` は式 e_2 で，それぞれ特定される値まで順に変化するものとする．イテレータなどを用いて，整数のループカウンタを用いずに表現されるループも少なくない．しかし，多くの場合は単純な変換によって整数のループカウンタを用いたプログラムへ書き換えることができる．たとえば，以下のようなプログラムであれば，

```
for x in arr:
  v = v ⊕ x
```

次の等価なプログラムを代わりに考えることができる．

```
for i in range(len(arr)):
  v = v ⊕ arr[i]
```

また，ループカウンタが減少してゆくようなループや，ループカウンタが 2 ずつ増えるようなループも，同様に 1 ずつ増える整数を用いたループへ簡単に書き換えることができる．

2.2　貢献集合と差分集合を用いた漸増化

Liu らの手法では，まず**貢献集合**と呼ばれる集合を表す論理式を求める．貢献集合 $S(i)$ は，ループカウンタ `i` の値が i のときの繰り返し処理で集約される寄与配列の要素集合[*2]を表すものである．図 1 はプログラム A の貢献集合を図示したものである．

2.1 節のプログラムに対する貢献集合は具体的には以下となる．

[*1] ただし，数値誤差を考慮に入れた場合の浮動小数点演算は結合則を満たさないためこの制約を満たさない．

[*2] この集合は実際には多重集合であるべきだが，本論文では簡単のため両者を区別しない．

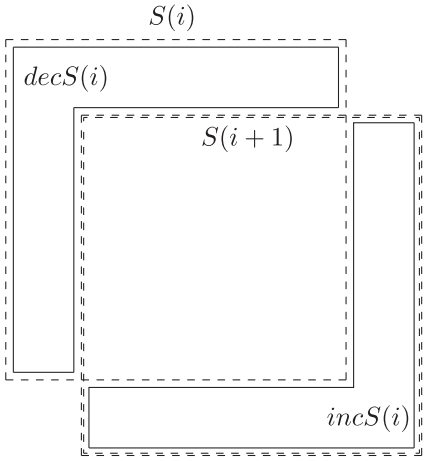

図 2　貢献集合と差分集合．破線で囲まれた領域が $S(i)$，二重破線で囲まれた領域が $S(I+1)$ を表す

Fig. 2　The contribution set and the difference set: the regions surrounded by dashed Lines and double dashed Lines respectively denote $S(i)$ and $S(i+1)$.

$$S(i)=\{\mathrm{a}[f(i,j)]\mid 0\le j<e_2(j)\}$$

Liu らの手法の骨子は，連続するループ実行間の貢献集合の差を求める点にある．これを**差分集合**と呼ぶ．差分集合には 2 種類ある．**増加差分集合** $incS(i)$ はループカウンタが i のときの繰り返し処理では集約されないが $i+1$ のときには必要なものである．また，**減少差分集合** $decS(i)$ はループカウンタが i のときの繰り返し処理では集約されるが $i+1$ のときには必要ないものである．両者を式で表すと以下となる．

$$incS(i)=S(i+1)\setminus S(i)$$
$$decS(i)=S(i)\setminus S(i+1)$$

貢献集合と差分集合の関係を**図 2** に示す．

当然のことではあるが，貢献集合と差分集合の間には以下の漸化式が成り立つ．

$$S(i+1)=S(i)\setminus decS(i)\cup incS(i)$$

この漸化式を用いて，ループカウンタが i のときの繰り返し処理の結果から $i+1$ のときの結果を求めるのが，漸増化の基本的な方針である．具体的には以下のコードを生成する．

```
v = v ⊕ a[f(0)]
for i in range(1, e):
  for x in decS(i):
    v = v ⊖ x
  for x in incS(i):
    v = v ⊕ x
```

プログラムから明らかなように，ループカウンタが 0 の場合は普通に計算し，以降は前回の結果から差分集合の要素を差し引きすることで次の結果を求める形式になっている．

2.3　漸増化手法の分析

以上見てきたLiuらの手法は，方針としては明確である．しかし，実際にこの手法で効率的な漸増計算プログラムを得るためには，差分集合に対する効率的な繰り返し処理が必要となる．いい換えると，$decS(i)$ や $incS(i)$ の要素を効率的に列挙する方法が必要となる．しかし，1章でも論じたとおり，これは自明なことではない．2.2節で示した式から直截にコード生成してしまうと，多くの場合は効率の良くないコードとなってしまう．

この問題を解決するため，Liuらは Omega Test [29] で開発された数式単純化法を用いることを提案している．これは，Presburger算術の論理式で記述された集合を，否定と論理和を含まない論理式で記述される集合の直和に分解するものである．これにより，差分集合に含まれない要素の処理が生成したコードに含まれてしまうことをある程度避けることができる．

しかし，この手法は完璧ではない．一般に，論理式 A と論理式 B のどちらについてもそれを満たす要素を効率的に列挙できたとしても，論理式 $A \wedge B$ を満たす要素**のみ**の効率的な列挙は自明ではない．この難しさはたとえば命題論理の充足可能性問題に見ることができる．各節（論理変数またはその否定の論理和）を真にする付値を列挙するのは簡単だが，節が複数論理積でつながっているとき，そのすべてを真にする付値を列挙するのは困難である．

さらに事態を悪化させているのは，この数式単純化法が大変複雑なものであり，これを適用した結果差分集合がどのような論理式となるかを前もって予想しにくい点である．もし，ある程度結果が予想可能であれば，どのようなプログラムであれば効率的なコードが得られるか，ある程度見通しを立てることができる．しかし，この数式単純化法を用いる限り漸増化の成否は予想しにくい．たとえ等価なプログラムであっても，Presburger算術への翻訳が異なれば，単純化結果はまったく異なるものとなりうる．これは，プログラム開発にこの漸増化手法を組み込むことを考えれば，看過しにくい問題となりうる．

また，この手法を用いるためには，差分集合がPresburger算術で表現できなければならない．実用的な多くのプログラムはこの制約を満たすだろうが，とはいえ複雑なプログラムであれば（本質的には可能であるとしても）Presburger算術への翻訳は非自明なものとなりやすい．また，本来Liuらの手法の着想，つまり貢献集合の直接的な処理を差分集合の処理で置換すること，はPresburger算術では表現できないような差分集合であっても適用できるはずである．そのため，利用しようとしている制約ソルバ（の数式単純化手法）の限界によって漸増化手法の能力が制約されてしまうのは避けたい．

3.　配列集約ループの実行時情報を用いた漸増化

本節では2.3節で論じた問題点をふまえ，制約ソルバを用いることなく，差分集合が満たす法則を推測し，その要素を列挙するコードを出力する手法を示す．

3.1　準備

まずは提案手法の説明のために必要となるいくつかの概念を準備する．

提案手法では貢献集合や差分集合などの要素集合を扱う．これらは，特に配列の添字にあたるベクトルで特徴づけられる．そのため，ベクトルについて標準的な語彙を本論文でも用いる．まず，ベクトル $(0,\ldots,0,1,0,\ldots,0)$ のように，1つの1を除いてすべてが0のベクトルを**単位ベクトル**と呼ぶ．特に，k 番目の要素（最初の要素を0番目とする）のみが1である単位ベクトルを $\mathbf{e}_k$ と記述することにする．ベクトル対する和・差は標準的な定義を用いる．2つのベクトル $\vec{a} = (a_0, a_1, \ldots, a_m)$ と $\vec{b} = (b_0, b_1, \ldots, b_m)$ が $\vec{a} \leq \vec{b}$ であるとは，すべての $0 \leq i \leq m$ に対して $a_i \leq b_i$ が成り立つことである．ベクトル集合 V に対し，$\vec{a} \in V$ が極小であるとは，$V \ni \vec{b} \leq \vec{a}$ ならば $\vec{a} = \vec{b}$ を満たすことである．極大も同様に定義される．

提案手法では，ループで効率的に処理できる要素集合として，

$$\{\mathrm{a}[i_0][i_1]\cdots[i_n] \mid l_0 \leq i_0 \leq u_0, \ldots, l_n \leq i_n \leq u_n\}$$

という形式で特定されるものを考える．これを，図1や図2に示したような図形的直感に従い，**長方形**と呼ぶ[*3]．

長方形は，唯一の極小なベクトル（添字）と唯一の極大なベクトル（添字）によって特徴づけられる．たとえば上記要素集合であれば，$(l_0, l_1, \ldots, l_n)$ が極小なベクトル，$(u_0, u_1, \ldots, u_n)$ が極大なベクトルである．これをふまえ，極小ベクトル $\vec{l}$ と極大ベクトル $\vec{u}$ で特徴づけられる長方形を $\langle \vec{l}, \vec{u} \rangle$ と表記する．

長方形 $\langle \vec{l}, \vec{u} \rangle$（$\vec{l} = (l_0, \ldots, l_n), \vec{u} = (u_0, \ldots, u_n)$）および単位ベクトル $\mathbf{e}_k$ に対し，$\langle \vec{l}, \vec{u} \rangle$ **の** $\mathbf{e}_k$ **方向の隣接面**[*4]とは，以下の長方形である．

$$\langle (l_0, \ldots, l_{k-1}, u_k + 1, l_{k+1}, \ldots, l_n), \\ (u_0, \ldots, u_{k-1}, u_k + 1, u_{k+1}, \ldots, u_n) \rangle$$

また同様に，$\langle \vec{l}, \vec{u} \rangle$ **の** $-\mathbf{e}_k$ **方向の隣接面**とは，以下の長方形である．

$$\langle (l_0, \ldots, l_{k-1}, l_k - 1, l_{k+1}, \ldots, l_n), \\ (u_0, \ldots, u_{k-1}, l_k - 1, u_{k+1}, \ldots, u_n) \rangle$$

[*3] 必ずしも2次元とは限らないので「超直方体」と呼ぶほうが自然かもしれないが，より馴染み深い長方形という言葉を使う．

[*4] これも「超平面」と呼ぶほうが正確かもしれないが，馴染み深い語彙を採用している．

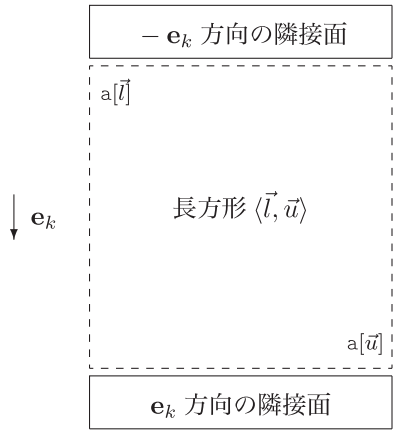

図 3　長方形 $\langle\vec{l},\vec{r}\rangle$ に対する，極小ベクトル $\vec{l}$，極大ベクトル $\vec{u}$，および隣接面の関係

Fig. 3　The maximal vector $\vec{l}$, the minimal vector $\vec{u}$, and the adjacent plane for the rectangle $\langle\vec{l},\vec{r}\rangle$.

長方形 $\langle\vec{l},\vec{u}\rangle$ にその $\mathbf{e}_k$ 方向の隣接面を加えると，$\mathbf{e}_k$ 方向に 1 列分大きな長方形となる．また，$-\mathbf{e}_k$ 方向の隣接面を加えると，$\mathbf{e}_k$ の負方向に 1 列分大きな長方形となる．**図 3** に長方形・その極小ベクトルと極大ベクトル・隣接面の関係を示す．

提案手法では，数列からその法則性を推測することを何度も行う．より具体的には，ループカウンタの値が k のときに，何らかの値が a_k であったとして，数列 $a_0, a_1, \ldots, a_m$ から $g_a(i) = a_i$ となる関数 g_a を求める．この行為を「数列 $a_0, a_1, \ldots, a_m$ を i の関数として推測する」と呼ぶことにする．本研究では特に $g_a(x)$ が x についての多項式である場合を考える．数列 $a_0, a_1, \ldots, a_m$ に対し，$g_a(i) = a_i$ となる多項式 $g_a(x) = b_0 + b_1x^1 + \cdots b_nx^n$ を求めるのは難しくない．各 $0 \le i \le m$ に対し $b_0 + b_1i^1 + \cdots b_ni^n = a_i$ を満たすよう，連立一次方程式を解くことで $b_0, b_1, \ldots, b_n$ を求めればよい．無論，解がない場合もある．その場合には「法則性の推測に失敗した」と呼ぶ．なお，この推測は各係数がベクトルである場合に自明に拡張できる．

3.2　漸増化の方針

Liu らの手法の問題点は，差分集合に対する効率良いループの生成が制約ソルバに依存していたことであった．本研究ではこれに対し，**効率良いループの生成方法が分からない差分集合には意味がない**と考え，「差分集合は効率良いループに対応するはず」という仮定のもと差分集合を推測する．

本研究では，「効率良いループに対応する」差分集合として**長方形の有限和**に注目する．この主な理由は以下の 3 点である．まず，長方形は自明にループに対応し，またキャッシュなどの観点からも，最も効率的に処理できる構造だといっても過言ではない．次に，実用的なプログラムの多くが配列の処理範囲として長方形（またはその有限和）を扱っており，このアプローチはそれらをうまくとらえられる可能性が高い．最後に，長方形は極小・極大ベクトルで特徴づけられる単純な構造であるため，差分集合が長方形で表すことができるかの推測も比較的簡単である．

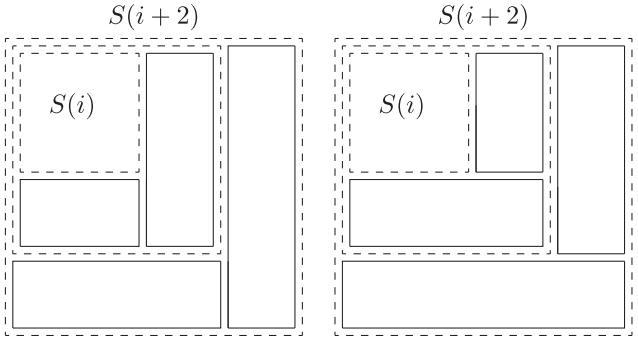

図 4　差分集合の長方形への 2 種類の分解

Fig. 4　Two ways of decomposing a difference set into rectangles.

図 4 は提案手法をプログラム A に適用した様子を示したものである．プログラム A では差分集合は逆 L 字の構造をしているが，これを 2 つの長方形（図 4 の左または右）に分解する．こうすると，各横長の長方形・縦長の長方形の極小・極大ベクトルは，ループカウンタ i の値から簡単に求めることができる．たとえば図 4 の左の分解であれば，縦長の長方形は $\langle(0,i),(i,i)\rangle$，横長の長方形は $\langle(i,0),(i,i-1)\rangle$，というものとなる．このような長方形の特徴付けが得られてしまえば，漸増計算コードの生成は難しくない．

注意深い読者は，差分集合の長方形への分解が一意でないこと，そしてどのように分解するかに漸増化の成否が依存することに気づいたかもしれない．たとえば図 4 の場合，ループカウンタごとに左の分解（縦長の長方形が大きい）と右の分解（横長の長方形が大きい）が混在してしまっている場合には，各長方形のループカウンタの値による特徴付けの推測に失敗してしまう．この問題点については，後ほど漸増化アルゴリズムの詳細とともに 3.3 節で議論する．

以上をふまえ，提案手法では以下の手順で漸増化を行う．

(1) 対象となるループを，各繰り返しにおいて寄与配列のどの要素の集約結果が累積変数に保持されるかを記録しつつ，一定回数実行する．いい換えると，各貢献集合 $S(i)$ を実際に実行しつつ求める．これを用いて，各差分集合 $incS(i)$・$decS(i)$ を求める．
(2) 各 $incS(i)$・$decS(i)$ を長方形の集合に分解する．
(3) 各 $incS(i)$・$decS(i)$ に含まれる各長方形の極小ベクトル・極大ベクトルの法則性を推測することで，その一般式を得る．
(4) 漸増化結果のコードを生成する．

以上の手順のうち，ステップ (1) は自明である．ただし，

Liu らの手法では $S(i)$ が論理式であったのに対し，提案手法での $S(i)$ は具体的なベクトル（添字）の集合である[*5]点は，重要な差異なので強調しておく．また，ステップ (4) については，2.2 節からほとんど変更はない．むしろ，提案手法では差分集合が長方形の集合として特定されるため，ループの生成は簡単になる．

以上をふまえ，以下ではステップ (2) と (3) をそれぞれ 3.3 節・3.4 節で説明する．

3.3 長方形への分割

まずは，各 $incS(i)$・$decS(i)$ を長方形の集合に分解する．領域を長方形集合へ分解する問題は VLSI のデザインなど様々な文脈で現れ，特に最小個数の長方形へ分解する場合については，二部グラフの最大マッチングに帰着する巧妙な多項式時間アルゴリズムが知られている．詳細については Eppstein によるサーベイ [14] を参照されたい．

本研究では既存の効率の良いアルゴリズムを使用しない．効率の良いアルゴリズムほど非自明な手順によって分解を求めており，それゆえ結果がどのようなものになるか予想しにくい．これは本研究での長方形分解の目的に反する．本研究では差分集合の間の一般的な法則性を推測するために長方形に注目している．そのため，各ループカウンタでの差分集合について，その集合の形状の微妙な変化によって長方形分解の結果が変化することは，たとえそれによって必要な長方形の数が減るとしても，望ましくない．たとえば，図 4 の左の分割と右の分割のどちらが得られるかが，アルゴリズムや入力の詳細に依存してしまいコントロールできなくなることは避けたい．より望ましいアルゴリズムは，元々の計算の構造をできる限り反映し，ループカウンタの値によらず一貫した分解を与えるものである．このためには，長方形分解のアルゴリズムは単純であることが望ましい．

以上をふまえ，提案手法は以下の単純なアルゴリズムを用いる．これは入力集合 D を長方形に分解し，出力集合 D' へ格納する．

(1) $D' = \emptyset$ とする．
(2) D から適当に 1 つベクトルを取り出し，そのベクトルのみからなる長方形を R とする．
(3) R の隣接面で D に含まれるものを発見し，その隣接面を R に加える．そのような隣接面が存在する限りこのステップを繰り返す．そのような隣接面が存在しなければ次へ進む．
(4) D' に R を加える．また，D から R に含まれるベクトルをすべて取り除く．
(5) D が空になれば D' を出力し終了する．D が空でなければ，ステップ (2) に戻る．

[*5] 正確には「配列とその添字（ベクトル）」の集合だが，以下の議論では説明の簡単のため添字のみに注目する．

上記アルゴリズムには，どのベクトルを入力集合から取り出すか，またどの方向の隣接面を考えるかによって非決定性がある．そのため，実際にこのアルゴリズムを使用する際には「ループ処理で早くアクセスされた順に取り出す」「配列の低次元に対応する方向を優先する」などのヒューリスティックな実装を行うことになる．我々の洞察は，このような**ヒューリスティックな実装こそが漸増化の成功のために重要**というものである．自然に書かれたプログラムでは，ループカウンタ i の値に計算の構造が大きく依存することは少ない．たとえば，ある i の値では $\mathbf{e}_k$ 方向に処理を行うが，別の値では $-\mathbf{e}_k$ 方向に処理を行う，などといったことは少ない．となると，各ループの繰り返しでの計算の進行を自然に反映した，一貫したヒューリスティクスを用いれば，ループカウンタの値によらず一貫した長方形分割が得られる可能性が高い．

具体例として 1 章で議論したプログラム A を考える．ループカウンタ $i = 0, 1, 2, 3$ について寄与集合を観測すると以下の結果が得られる．

$$
\begin{aligned}
S(0) &= \{(0,0)\} \\
S(1) &= \{(0,0),(0,1),(1,0),(1,1)\} \\
S(2) &= \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2), \\
&\qquad (2,0),(2,1),(2,2)\} \\
S(3) &= \{(0,0),(0,1),(0,2),(0,3),\ldots,(3,2),(3,3)\}
\end{aligned}
$$

これをもとに差分集合を計算すると以下となる．

$$
\begin{aligned}
decS(0) &= \emptyset \\
decS(1) &= \emptyset \\
decS(2) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
incS(0) &= \{(0,1),(1,0),(1,1)\} \\
incS(1) &= \{(0,2),(1,2),(2,0),(2,1),(2,2)\} \\
incS(2) &= \{(0,3),(1,3),(2,3),(3,0)(3,1),(3,2),(3,3)\}
\end{aligned}
$$

減少差分集合はつねに空集合であるので，長方形への分解は必要なく，また法則性の推測も自明である．よって以降は増加差分集合のみを考える．

それぞれの増加差分集合を長方形分解すると，ヒューリスティクスにもよるが，たとえば以下のようになる．なお $\uplus$ は集合の直和を表すものとする．

$$
\begin{aligned}
incS(0) &= \langle (0,1),(1,1)\rangle \uplus \langle (1,0),(1,0)\rangle \\
incS(1) &= \langle (0,2),(2,2)\rangle \uplus \langle (2,0),(2,1)\rangle \\
incS(2) &= \langle (0,3),(3,3)\rangle \uplus \langle (3,0),(3,2)\rangle
\end{aligned}
$$

この分解は図 4 の左側のものである．各増加差分集合について，左の長方形が縦長の長方形に対応する．

3.4 法則性の推測

次にループカウンタから各長方形を求める法則を推測する．この手順は以下のものである．以下の説明で

は増加差分集合について考えているが，減少差分集合についても同様に処理する．以下，各増加差分集合が $incS(i) = \langle \vec{l_1^i}, \vec{u_1^i} \rangle \uplus \cdots \uplus \langle \vec{l_n^i}, \vec{u_n^i} \rangle$ と分解されているとする．

(1) まず，各増加差分集合に含まれる長方形の数が同じであることを確認する[*6]．そうでない場合，このアルゴリズムは失敗する．

(2) 各 j（$0 \leq j \leq n$）について，数列 $(\vec{l_j^i}) = \vec{l_j^0}, \vec{l_j^1}, \ldots$ および数列 $(\vec{u_j^i}) = \vec{u_j^0}, \vec{u_j^1}, \ldots$ をループカウンタ i の関数として推測し，得られた関数をそれぞれ g_l^j および g_u^j とする．1 つでも推測に失敗した場合，このアルゴリズムは失敗する．

(3) 各 g_l^j および g_u^j を出力する．

このアルゴリズムは各長方形の極小・極大ベクトルをループカウンタから求める関数を推測する．各差分集合が複数の長方形に分解されていた場合には，どの長方形の間の法則性を推測するかが問題となる．

ナイーブな考え方としては，法則性が見つかるまで様々な可能性を試すものがあげられるだろう．しかし，この方法は現実的でない．増加差分集合を m 個求めており，それぞれが n 個の長方形を含むとき，各増加差分集合から 1 つずつ長方形を取り出す組合せは n^m もある．かつ，この組合せのほとんどは，互いに関係の乏しい長方形を含んでいるために法則性の推測に失敗するはずである．つまり，このアプローチは，非常にコストが高いうえに成功する可能性が低い．

そこで本研究では 3.3 節と同様の考えに基づいたヒューリスティクスを採用する．具体的には，各 3.3 節の長方形分解アルゴリズムで長方形が得られた順番を覚えておき，その順番どおりに長方形を対応させる．たとえば，各増加差分集合で最初に抽出された長方形どうしについて，法則性を推測するということである．これは，各 3.3 節の自明なアルゴリズムを適切なヒューリスティクスのもとで実行した場合，得られた長方形は，その順番まで含め，計算の構造を自然に反映しているだろう，よって図 4 の左と右の分解が混在するようなことはないだろう，という考えに基づいている．

再び具体例としてプログラム A を考える．各増加差分集合について 1 つ目・2 つ目の長方形を抽出するとそれぞれ以下の列となる．

$\langle (0,1),(1,1) \rangle, \langle (0,2),(2,2) \rangle, \langle (0,3),(3,3) \rangle$
$\langle (1,0),(1,0) \rangle, \langle (2,0),(2,1) \rangle, \langle (3,0),(3,2) \rangle$

これらの極大ベクトル・極小ベクトルがそれぞれ 1 次式をなすと仮定して法則性を推測すると，ループカウンタを i としたときのこれら長方形の一般式としてはそれぞれ以下のものが求まる．

$\langle (0,i+1),(i+1,i+1) \rangle$
$\langle (i+1,0),(i+1,i) \rangle$

これをもとにコード生成を行えばプログラム B が得られる．

3.5　漸増化アルゴリズムの計算量

以下，提案手法による漸増化アルゴリズムの時間計算量を分析する．以下では，各差分集合の大きさを k，推測で得る多項式の次数を l，ループを実行して寄与集合を求める回数を m，各差分集合から得られる長方形の個数を n とする．

まず，長方形分割に要する計算量を考える．ここでは，あるベクトルが差分集合に含まれるかどうかの問合せの回数で計算量を分析する．実際の計算量は差分集合の実装に依存する．たとえば，差分集合が平衡木で実装されている場合，1 回の問合せに $\mathrm{O}(\log k)$ を要する．また，実装がハッシュテーブルでありハッシュ関数が競合しない場合は，1 回の問合せに要する時間は $\mathrm{O}(1)$ で済む．

1 つの長方形を発見する際に，同じベクトルに対する問合せは 2 度行う必要はない．そのベクトル（を含む隣接面）が長方形に加えられた場合は再度問合せられることはない．また，そのベクトルを長方形に加えられなかった場合，その方向の隣接面をそれ以降その長方形に対して考慮する必要はない．なぜなら，長方形は大きくなる一方であり，小さな長方形のときに加えられなかった隣接面が大きな長方形に対して加えられるようになることはありえないからである．そのため，1 つの差分集合を長方形分割するのに要する問合せの数は $\mathrm{O}(kn)$，長方形分割全体で要する問合せの数は $\mathrm{O}(kmn)$ である．

次に長方形の法則性を推測するために要する計算量を考える．1 回の推測に現れる長方形の数は m であるから，この推測には未知数 $l+1$・式数 m の連立一次方程式の求解が必要となり，計算量は $\mathrm{O}(l^2 m)$ である．よって，全体としての計算量は $\mathrm{O}(l^2 mn)$ となる．

以上から，漸増計算に要する計算量は，差分集合への問合せが単位時間だとして，$\mathrm{O}((k+l^2)mn)$ となる．実用上は l および n は定数程度だと仮定できることが多く[*7]，その場合には計算量は $\mathrm{O}(km)$ となる．差分集合の大きさがたかだか寄与集合の大きさ程度であることに注意すると，$\mathrm{O}(km)$ という計算量は，入力されたプログラムをループカウンタ m まで実行する計算量と同程度である．つまり，通常の実行に比べて定数倍程度のオーバヘッドであり，現実的であるといえる．

[*6] 初めの数個には例外的に異なる数の長方形が含まれることもありうるだろう．一般的には，i がある定数以上の場合について法則性が推測できれば十分である．

[*7] 少なくとも，l や n を大きくしないと漸増化できない場合は諦める，といった実装方針が可能である．

4. 議論

4.1 提案手法の正しさ

提案手法では与えられたプログラムを実際に実行してみて，その結果をもとに漸増化を行う．このアプローチには，2 つの自然な疑問が生じるだろう．

- プログラムを実際に実行するなら，計算結果はすでに得られているはずで，それをもとに漸増化を行い効率化を行っても無意味ではないか．
- プログラムの実行結果を用いて行った最適化は，パラメータの異なるプログラムの実行でも正しい保証はないのではないか．

これら 2 つの問題は関連している．仮に，2 つ目の問題が解消した，つまりパラメータを変えても最適化の正しさが保存されるとしよう．この場合，小さな，比較的時間のかからない入力に対する実行をふまえて効率化を行い，それをより大きく時間のかかる入力に適用する，というようなアプローチが考えられる．また，JIT コンパイラのように，ループの一定回数の実行を監視し，その実行をふまえ効率的なコードを生成し，以降のループは生成したコードを用いて実行する，というような利用法もありうるだろう．しかし残念ながら，提案手法の正しさは原理的に保証できない．

正しさが保証されない効率化には意義が乏しいと感じるかもしれない．しかし，提案手法には少なくとも 2 つの有用な利用方法がある．

1 つは，提案手法を効率的なコードの候補の生成手法として用い，その正しさは別途確認するという利用方法である．この正しさの確認にたとえ制約ソルバを用いることになったとしても，この際に必要となるのは多くの場合式の単純化ではなく制約充足問題の求解であり，よりプログラムの細部に依存しにくいと期待できる．また，そもそも非自明なプログラムの生成に比べれば一般にプログラムの正しさの確認のほうが簡単な問題であり[*8]，現実的なコストで正しい可能性の高いコード候補が得られるだけでも有益である．特に，得られたプログラムの正しさの確認には推測された「法則性」の検証で十分であり，この「法則性」がループ内不変条件に対応することは注目に値する．多くのプログラム検証手法がループ内不変条件を推測したうえで検証していることに比べれば，すでに確認すべきループ内不変条件が得られている状況は，かなり問題が簡単になっているといえる．

なお，提案手法はループの実行を観察することで，テストも同時に行っていると見なすことができる．そのため，少なくともテストを十分行うことで確認できる程度の正しさは，提案手法にもある．また，闇雲にループを実行するのではなく，テストケースに対する実行を用いて効率化を行うことで，コーナーケースに関してもより正しい可能性が高いコードを生成することもできるだろう．

もう 1 つは，正しさの最終的な保証はプログラマに任せるという考え方である．数値誤差をともなう計算に関する高度な最適化や，自動並列化などでは，最適化オプションやプラグラマによって指定される最適化の正しさが必ずしも保証されないことは珍しくない．このような場合，プログラマは最適化で行われる内容を大まかに理解したうえで，それが今作成しているプログラムにとって有害でないことを自分で判断する．たとえば数値誤差であれば，多少の計算誤差が問題とならないことを，また並列化であれば処理の実行順序が変わっても問題ないことを，プログラマが確認することになる．提案手法の場合，ループを実際に実行して観察した際の挙動と，パラメータを変えて実行した際の挙動に，特に寄与配列集合の形状やその走査順序に関して大きな違いがないことをプログラマが確認すればよい．多くの場合，特に提案手法が成功するような単純で自然なプログラムの場合，この確認はそれほど難しくないだろう．

4.2 生成コードの計算量の推測

提案手法の特徴の 1 つとして，生成したコードの計算量を簡単に見積もることができるという点があげられる．これにより，生成したコードの計算量がもとのプログラムより良い場合にのみ効率化を行う，といったことが可能となる．

漸増計算では長方形を処理するループが生成される．長方形の面積（体積）はその極小・極大ベクトルから簡単に計算できる．具体的には，長方形 $\langle \vec{l}, \vec{u} \rangle$ について，$\vec{l} = (l_0, l_1, \ldots, l_m)$，$\vec{u} = (u_0, u_1, \ldots, u_m)$ のとき，この長方形の面積は

$$\prod_{i=0}^{m} (u_i - l_i + 1)$$

である．これは漸増計算でのループカウンタを 1 つ増やすのに要する時間の見積りを与える．

さらに，ループカウンタ i に関する長方形の法則をふまえ，i に関する総和を取ることで，計算全体で要する時間を見積ることができる．特に，漸近計算量にのみ興味がある場合，i についての最高次数より 1 つ大きい次数となるため，見積りはごく簡単である．

再び具体例としてプログラム A を考える．求まった長方形の一般式は以下のものであった．

$$\langle (0, i+1), (i+1, i+1) \rangle, \langle (i+1, 0), (i+1, i) \rangle$$

これから面積（体積）を求めるとそれぞれ以下となる．

[*8] もちろん理論上はどちらも決定不能問題となるだろうが，Liu らの手法にも見られるように，非自明なプログラムの生成では制約ソルバやプログラム検証器をその一部として使うものも多い．

$(i+1-0+1)((i+1)-(i+1)+1) = i+2$
$((i+1)-(i+1)+1)(i-0+1) = i+1$

よって漸増計算ではループカウンタ i のときの実行に $\mathrm{O}(i)$ の時間を要し，全体ではループカウンタの最大値を N として $\mathrm{O}(N^2)$ の計算量となる．

4.3　提案手法の適用可能性

提案手法はより一般的なプログラムへと自然に拡張することができる．

まず前提として，提案手法は Liu らの手法とは異なり，ループが**構文的に**特定の形式であること，すなわち寄与配列要素を集約し累積変数へ代入するだけのプログラムであること，は要求しない．代わりに**実際の挙動として**そのような動作が観測されることを要求する．この差は小さくはない．実用的なプログラムでは，構文的な解析は，外部関数の呼び出しや配列の別名の可能性などのためにすぐに難しくなってしまう．しかし，構文的には少々複雑なプログラムであっても，実際の挙動はそれほど複雑でないことも多い．また，構文的に挙動が明らかなプログラムは，当然実際の挙動も単純である．よって，この点は提案手法の長所であるといえる．

3 章では累積変数への代入が 1 つだけある状況を考えた．代入が複数ある場合についても，代入先が異なるならばそのそれぞれに対して，寄与集合や差分集合，その長方形への分解などを求めればよい．ただし，必ずしも同じ代入どうしの寄与集合に共通部分が多いとは限らない．場合によっては異なる代入間の寄与集合について差分集合を求めるのがよいかもしれない．この点のさらなる分析は今後の課題である．

また，3 章では寄与配列の要素をそのまま結合・交換則を満たす演算子で集約する状況を考えた．これは必須ではない．たとえば，以下のように各要素が定数倍されるような状況でも同様のアルゴリズムが機能する．

```
for i in range(N):
  for j in range(i):
    b[i] = b[i] + 2 * a[j]
```

より一般に，寄与配列の要素に対し，その添字に依存しない計算が行われている場合，つまり適当な関数 g があって $v = v \oplus g(\mathrm{a}[f(i)])$ の形式の計算が行われている場合には，まったく同様にして漸増化を行うことができる．ただし，g の計算が複雑になるに従い，プログラムの挙動が確かにこの形式の計算となっていること，そしてその際の寄与集合を知ることが難しくなるかもしれない．

このような場合の典型例は，減算や除算による集約である．減算は -1 倍した値の加算による集約，除算は逆数の乗算による集約に対応する．そのため，上述の理由によって漸増化が可能である．

4.4　提案手法の限界

提案手法は差分集合が有限個の長方形に分解できない場合には漸増化ができない．例としては以下のプログラムがあげられる．

```
for i in range(N):
  for j in range(i):
    for k in range(i - j):
      b[i] = b[i] + a[j][k]
```

このプログラムの差分集合は $\{\mathrm{a}[j][k] \mid j+k=i\}$ という斜めの線のような領域になる．提案手法はこのように斜めの形状となる場合には漸増化できない．なお，この例については Liu らの手法では漸増化が成功する可能性が高い．

また，漸増化の際には直前のループの繰り返しの結果を用いるため，直前の結果との共通がない場合には効率が向上しない．例としては以下のプログラムがあげられる．

```
for i in range(N):
  for j in range(i):
    b[i] = b[i] + a[2 * j + i % 2]
```

このプログラムでは i の値の偶奇によって集約する配列要素の範囲が大きく変わるために漸増化ができない．ただし，このようなプログラムであれば，以下のようなループ展開によって状況を改善させることができる．簡単のため N は偶数であると仮定する．

```
for i in range(N // 2):
  i1 = 2 * i
  i2 = 2 * i + 1
  for j in range(2 * i):
    b[i1] = b[i1] + a[2 * j]
    b[i2] = b[i2] + a[2 * j + 1]
  b[i2] = b[i2] + a[4 * i + 1]
```

このプログラムであれば，直前の繰り返しの寄与集合との共通部分がある（ただし，異なる代入の寄与集合である）．このように，ループ展開は，各繰り返しでの寄与集合を大きくするため，一般に漸増化が成功しやすくなる．なお，この問題とループ展開の有用性については，Liu らの手法と共通する性質である．

4.5　多項式以外の法則性の推測

本研究では多項式による法則性の推測を行った．そのため，たとえば以下のループのように，等比的に寄与集合の範囲が変化するプログラムを効率化することはできない．

```
for i in range(N):
  for j in range(2 ** i):
  b[i] = b[i] + a[j]
```

もちろん，等比的な法則の推測も可能ではある．しかし，等比的な法則の推測は実用上あまり有益ではないと考えている．なぜなら，等比的な挙動するようなループは寄与集

表 1　実験結果（単位：秒）
Table 1　The results for the experiments (unit: second).

	N = 100	N = 200	N = 400	N = 800	計算量
実験例 1（通常実行）	0.23	0.46	1.06	2.07	O(N)
実験例 1（漸増化）	0.08	0.09	0.10	0.23	O(N)
実験例 2（通常実行）	0.28	1.97	16.38	129.57	$O(N^3)$
実験例 2（漸増化）	0.02	0.06	0.30	1.06	$O(N^2)$
実験例 3（通常実行）	0.32	2.55	21.46	138.82	$O(N^3)$
実験例 3（漸増化）	0.08	0.14	0.44	1.68	$O(N^2)$

合が急速に大きく（または小さく）なってしまい，ループの繰り返し回数をあまり大きくできないからである．漸増計算は前回の計算と今回の計算での共通部分が多く，そしてループの繰り返し回数が多い場合に効果的である．等比的な挙動の場合，このどちらの前提も満たされないため，漸増化があまり有益でない可能性が高い．

多項式・等比以外の形の法則を推測することも考えられる．とはいえ，一般的な，実用的なプログラムを考えた場合，これら以外の法則が自然に書かれたプログラムに現れる可能性は低いと思われる．ただし，アプリケーション特有の法則がありうる場合には，それに特化した推測を行うことは有益だろう．

5. 実験

提案手法の有効性を確認するため，簡単なプロトタイプ実装と実験を行った．

プロトタイプ実装は，Python の小さなサブセットを入力とする．入力プログラムは代入，集約演算，および一部の基本的なリスト操作のみを含む for 文（2.1 節参照）であることを仮定している．集約演算としては加算のみを許している．また寄与配列は二次元までを許している．

プロトタイプ実装は，与えられたプログラムに対し，寄与配列の要素を適当な乱数値で初期化し，ループの繰り返しを最初の数回実行し，寄与集合を観測し，漸増化を行う．漸増化に成功した場合，コードを生成し，そのコードを exec 関数で実行する．つまり，JIT コンパイラのように，生成した効率的なコードを実行することで，所望の計算を達成するシステムとなっている．これに加え，システムは漸増化後のコードの計算量の見積もりも行う（4.2 節参照）．

以上のシステムに**図 5**，**図 6**，**図 7** に示すプログラムを入力し，ループの繰り返し回数 n を変化させながら，漸増化後のコードが最終的な計算結果を求めるまでに要した時間を計測した．なお，この時間は漸増化・コード生成などのすべてを含んだ時間であることに注意せよ．ただし，実装の簡単のため，構文解析機は実装しておらず，抽象構文木に対応するデータを直接与える構造となっているため，実行時間にはファイルの読み込みや構文解析に要した時間は含まれない．さらに，比較として，漸増化を行わずそのまま実行した場合の時間も計測した．

```
ans = []
for i in range(N):
  ans.append(0)
  for l1 in range(100):
    for l2 in range(100):
      ans[i] += b[l1 + i][l2]
```

図 5　実験例 1：処理領域が平行移動する例
Fig. 5　Example 1: parallel translation of a rectangle.

```
for i in range(N):
  for k in range(2 * i + 1):
    for l in range(2 * i + 1):
      s[i] = s[i] + a[k][l]
```

図 6　実験例 2：処理領域が右下へ 2 ずつ拡大する例
Fig. 6　Example 2: extending a rectangle to the right down direction.

```
ans=[]
for i in range(N):
  ans.append(0)
  for l1 in range(-i + 1000, i + 1001):
    for l2 in range(-i + 1000, i + 1001):
      ans[i] += b[l1][l2]
```

図 7　実験例 3：処理領域が上下左右へ拡大する例
Fig. 7　Example 3: extending a rectangle on every sides.

実験環境は，CPU が Intel (R) Core (TM) i5-8250U（ベース周波数 1.60 GHz），RAM が 8.00 GB，OS が 64 ビット版 Windows 10 Pro，言語処理系が CPython 3.7.0 である．

実験結果を**表 1** に示す．いずれの例についても，漸増化は成功し，漸増化後のプログラムは通常実行と同じ計算結果を出力した．また，計算量の見積もり結果もすべて正しかった．実行時間から見て取れるように，漸増化に要する時間を含めたとしても，漸増化後のコードは通常実行に比べ十分に早い．以上は，あくまで予備的な実装・実験の結果ではあるが，提案手法が有望であることを示唆するものである．

参考として，**図 8** に，実験例 3 に対し，N = 100 の場合にシステムが出力したコードを示す．この例では，差分集合は環状の領域となり，4 つの長方形に分解される．そのため，漸増化処理は 4 つの 2 重ループを含む複雑なものと

```
ans=[]
ans.append(0)
for l1 in range(-0 + 1000, 0 + 1001):
  for l2 in range(-0 + 1000, 0 + 1001):
    ans[0] += b[l1][l2]
#漸増化パート
for i in range(1, 100):
  ans[i] = ans[i - 1]
  for p0 in range(998 + (i - 2) * -1,
                  999 + (i - 2) * -1):
    for p1 in range(998 + (i - 2) * -1,
                    1003 + (i - 2) * 1):
      ans[i] += 1 * b[p0][p1]
  for p0 in range(999 + (i - 2) * -1,
                  1003 + (i - 2) * 1):
    for p1 in range(998 + (i-2) * -1,
                    999 + (i - 2) * -1):
      ans[i] += 1 * b[p0][p1]
  for p0 in range(999 + (i - 2) * -1,
                  1003 + (i - 2) * 1):
    for p1 in range(1002 + (i - 2) * 1,
                    1003 + (i - 2) * 1):
      ans[i] += 1 * b[p0][p1]
  for p0 in range(1002 + (i - 2) * 1,
                  1003 + (i - 2) * 1):
    for p1 in range(999 + (i - 2) * -1,
                    1002 + (i - 2) * 1):
      ans[i] += 1 * b[p0][p1]
```

図 8　実験例 3 に対して出力されるコード
Fig. 8　The codes generated for Example 3.

なっている．このような非自明な例についても提案手法は自動的に漸増化を行うことができた．

6. 関連研究

本研究は Liu らの手法 [24] を元にして，漸増化を用いて配列集約ループの効率化を行った．漸増化を用いてループの効率化を行うアイデアは，少なくとも Paige らの finite differencing [8], [27], [28] にまで遡る．Paige らはこれを演算子の強度低減（strength reduction）の一種であると述べている．Paige らは主に集合を処理する計算を対象にして議論を行った．Liu らの手法は配列処理ループという実用上より多く現れる場面について finite differencing の着想を応用したものである．本研究もこの流れを継承している．特に，長方形という形状に注目する点は，集合ではなく配列を処理するという前提によって導かれたものである．

Gupta と Rajopadhye [17] はより複雑な集約ループに対して漸増化を行う手法を論じている．この手法では，寄与集合や差分集合を有限個の凸多面体の直和として表現する．これにより，漸増化が可能であるだけでなく，漸増化結果の計算量の見積もりや，それをふまえた最適な漸増化方針の探索などが可能であると論じている．提案手法は，凸多面体一般ではなく長方形に限定したうえで，この手法の現実的な実装を与えたと解釈できる．長方形から凸多面体に拡張するのは，その要素の列挙や差分の計算などに少なくないコストがかかり，実装をかなり複雑にする．特に，凸多面体の列に対し，それを特徴付ける法則を推論するのはかなり困難であり，本研究のような実行履歴の観察に基づくアプローチが有効かどうかは定かでない．一方，このような困難に見合った利得があるかどうかは不明瞭である．確かに，静的解析の文脈では，アクセスされる配列要素を凸多面体（の有限和）で特徴付けることは少なくない．しかし，その際には何かしらの近似が行われることがほとんどである．一方，漸増化のためには，アクセスされる配列要素を正確に把握する必要がある．整数値しかとらない離散的なベクトル群を正確に表現する際の凸多面体の有用性は，少なくとも自明ではない．

本研究は漸増化のアイデアを用いることでループの効率化を行っているが，これは典型的な漸増化とは異なる．典型的な漸増化では，入力の変更を前提としないプログラムを，入力が変更された際に高速に出力を更新するプログラムへと変換する．このような漸増化の研究については，属性文法に基づくもの [5], [13], [22], [36]，部分評価に基づくもの [34]，値の記憶に基づくもの [25], [30]，実行時での値の依存関係の記録に基づくもの [1], [3], [4], [10]，プログラムの微分に基づくもの [9], [15], [19]，関数融合に基づくもの [26] などがある．いずれも配列の集約処理には特化しておらず，そのため本研究のような漸増化を自動的に行うことはできない．

特に関数型言語の文脈では，以前の計算結果を再利用することで再帰プログラムを効率化する手法として，累積化 [6], [20], [23]，組化 [11], [21]，メモ化 [2] などが使われることが多い．これらの手法は原則として再帰呼び出しの結果や引数を再利用する．再帰プログラムでは，再帰呼び出しの構造が計算の構造とよく対応することが多く，そのためこれらの効率化が有効に働きやすい．一方で，配列を処理するループでは，ループの構造は実際の計算（たとえばどの配列要素をどの順序でアクセスするか）を必ずしも反映しているとは限らず，そのため本研究のようにその取り扱いに特化した手法が必要になる．

提案手法は，実際にプログラムを実行し，その挙動を観察することで効率化を行う．このアプローチは既存のいくつかの提案と類似性がある．まず，プログラムが配列要素の集約演算であると認識する部分は，投機的リダクション並列化 [18], [31], [35] の文脈で，並列化可能なリダクションループを発見する際に行われる解析と同一である．また，寄与集合を求める部分や，それを長方形の有限和に分解する部分は，実行時情報を用いた凸多面体解析 [7], [32], [33] に近い．ただし，前述のとおり，既存の凸多面体解析が近似を行うのに対し，漸増化では正確な把握が必要となるため，既存手法がそのまま使えるわけではない．同様の配列アクセス履歴の記憶とその法則性の推測は，効率の良い並列動的計画法プログラムの生成 [12] にも用いられている．

本研究は例によるプログラミング [16] に大きな影響を受けている．例によるプログラミングでは，プログラマは望ましい入出力例を与え，システムがその入出力例に合致する適切なプログラムを自動的に構築する．提案手法は，「ループカウンタの値 i を入力，そのときの差分集合を出力」として，例によるプログラミングを行っていると見なすことができる．例によるプログラミングでは，入出力例を満たすプログラムであればどのようなものを出力してもよいわけではない．特に，入出力例が有限種類であれば，条件分岐を羅列した自明なプログラムがつねに構成できるが，これは明らかに目的に合致しない．そのため，出力として望ましいプログラムがどのようなものであるかを特定し，その形式のプログラムを出力できるアルゴリズムを構成することが重要となる．本研究では特に，配列集約ループの漸増化という目的をふまえ，「長方形を処理するループ」が望ましいプログラムだと特定した点が要点となっている．

7. まとめと今後の課題

本論文では配列集約処理を行うループを漸増化により効率化する手法を示した．提案手法の要点は，実際にそのループを実行し集約される配列要素を注意深く観察することで，プログラムの挙動がなす法則性を推測する点にある．特に，アクセスされた配列要素の集合を直方体の有限和に分解することと，計算過程の情報をできる限り維持することで，その法則性の推論を現実的なコストで行うことを提案した．提案手法は先行研究である Liu らの漸増化手法に比べ，必ずしも適用範囲が広いわけでなく，また提案手法によって得られるコードは必ずしも正しいわけではない．一方で，制約ソルバの利用を避けたことで効率化の成否がプログラムの細部に依存しにくくなった点，また実行時情報の観察に基づくことで構文的な細部に影響を受けにくくなった点は，提案手法の長所である．

提案手法で効率化できるプログラムの範囲は更に拡大できると思われる．特に Gupta と Rajopadhye [17] の手法をふまえ，長方形をなさないような凸多面体への分解も検討することは，自然な拡張である．また，提案手法の骨子はループ内の計算の法則性の推測であり，必ずしも漸増化に特化したものではない．同様のアプローチで行える他の効率化や解析などを模索することも，意義ある研究の方向性だろう．

謝辞　実行時情報を用いた多面体解析とその応用についての関連研究を教えてくださった東京大学の佐藤重幸氏に感謝する．また，論文の改善に有益なコメントをいただいた匿名の査読者に感謝する．本研究は日本学術振興会科学技術研究費基盤研究（C）19K11896 の助成を受けている．

参考文献

[1] Acar, U.A., Blelloch, G.E., Blume, M., Harper, R. and Tangwongsan, K.: An experimental analysis of self-adjusting computation, *ACM Trans. Program. Lang. Syst.*, Vol.32, No.1, pp.3:1–3:53 (2009).

[2] Acar, U.A., Blelloch, G.E. and Harper, R.: Selective memoization, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.14–25, ACM (2003).

[3] Acar, U.A., Blelloch, G.E. and Harper, R.: Adaptive functional programming, *ACM Trans. Program. Lang. Syst.*, Vol.28, No.6, pp.990–1034 (2006).

[4] Acar, U.A., Blelloch, G.E., Harper, R., Vittes, J.L. and Woo, S.L.M.: Dynamizing static algorithms, with applications to dynamic trees and history independence, *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, Munro, J.I. (Ed.), pp.531–540, SIAM (2004).

[5] Alblas, H.: Incremental Attribute Evaluation, *Attribute Grammars, Applications and Systems, International Summer School SAGA, Proceedings*, Alblas, H. and Melichar, B. (Eds.), Lecture Notes in Computer Science, Vol.545, pp.215–233, Springer (1991).

[6] Bird, R.S.: The Promotion and accumulation strategies in transformational programming, *ACM Trans. Program. Lang. Syst.*, Vol.6, No.4, pp.487–504 (1984).

[7] Caamaño, J.M.M., Selva, M., Clauss, P., Baloian, A. and Wolff, W.: Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones, *Concurr. Comput. Pract. Exp.*, Vol.29, No.15 (2017).

[8] Cai, J. and Paige, R.: Program derivation by fixed point computation, *Sci. Comput. Program.*, Vol.11, No.3, pp.197–261 (1989).

[9] Cai, Y., Giarrusso, P.G., Rendel, T. and Ostermann, K.: A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, O'Boyle, M.F.P. and Pingali, K. (Eds.), pp.145–155, ACM (2014).

[10] Chen, Y., Dunfield, J. and Acar, U.A.: Type-directed automatic incrementalization, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, Vitek, J., Lin, H. and Tip, F. (Eds.), pp.299–310, ACM (2012).

[11] Chin, W.-N., Khoo, S.-C. and Jones, N.: Redundant Call Elimination via Tupling, *Fundamenta Informaticae*, Vol.69, No.1-2, pp.1–37 (2006).

[12] Chowdhury, R., Ganapathi, P., Tschudi, S.L., Tithi, J.J., Bachmeier, C., Leiserson, C.E., Solar-Lezama, A., Kuszmaul, B.C. and Tang, Y.: Autogen: Automatic Discovery of Efficient Recursive Divide-&-Conquer Algorithms for Solving Dynamic Programming Problems, *ACM Trans. Parallel Comput.*, Vol.4, No.1, pp.4:1–4:30 (2017).

[13] Demers, A.J., Reps, T.W. and Teitelbaum, T.: Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors, *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, White, J., Lipton, R.J. and Goldberg, P.C. (Eds.), pp.105–116, ACM (1981).

[14] Eppstein, D.: Graph-Theoretic Solutions to Computational Geometry Problems, *Graph-Theoretic Concepts in Computer Science, 35th International Workshop, WG 2009, Revised Papers*, Paul, C. and Habib, M.

(Eds.), Lecture Notes in Computer Science, Vol.5911, pp.1–16 (2009).
[15] Giarrusso, P.G., Régis-Gianas, Y. and Schuster, P.: Incremental λ-Calculus in Cache-Transfer Style - Static Memoization by Program Transformation, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings*, Caires, L., (Ed.), Lecture Notes in Computer Science, Vol.11423, pp.553–580, Springer (2019).
[16] Gulwani, S., Polozov, O. and Singh, R.: Program Synthesis, *Found. Trends Program. Lang.*, Vol.4, No.1-2, pp.1–119 (2017).
[17] Gupta, G. and Rajopadhye, S.V.: Simplifying reductions, *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, Morrisett, J.G. and Jones, S.L.P. (Eds.), pp.30–41, ACM (2006).
[18] Han, L., Liu, W. and Tuck, J.: Speculative parallelization of partial reduction variables, *Proc. CGO 2010, The 8th International Symposium on Code Generation and Optimization*, Moshovos, A., Steffan, J.G., Hazelwood, K.M. and Kaeli, D.R. (Eds.), pp.141–150, ACM (2010).
[19] Horn, R., Perera, R. and Cheney, J.: Incremental relational lenses, *PACMPL*, Vol.2, No.ICFP, pp.74:1–74:30 (2018).
[20] Hu, Z., Iwasaki, H. and Takeichi, M.: Calculating Accumulations, *New Generation Computing*, Vol.17, No.2, pp.153–173 (1999).
[21] Hu, Z., Iwasaki, H., Takeichi, M. and Takano, A.: Tupling Calculation Eliminates Multiple Data Traversals, *Proc. 2nd ACM SIGPLAN International Conference on Functional Programming, ICFP'97*, pp.164–175, ACM (1997).
[22] Hudson, S.E.: Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update, *ACM Trans. Program. Lang. Syst.*, Vol.13, No.3, pp.315–341 (1991).
[23] Kühnemann, A., Glück, R. and Kakehi, K.: Relating Accumulative and Non-accumulative Functional Programs, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Proceedings*, Middeldorp, A. (Ed.), Lecture Notes in Computer Science, Vol.2051, pp.154–168, Springer (2001).
[24] Liu, Y.A., Stoller, S.D., Li, N. and Rothamel, T.: Optimizing aggregate array computations in loops, *ACM Trans. Program. Lang. Syst.*, Vol.27, No.1, pp.91–125 (2005).
[25] Liu, Y.A., Stoller, S.D. and Teitelbaum, T.: Static Caching for Incremental Computation, *ACM Trans. Program. Lang. Syst.*, Vol.20, No.3, pp.546–585 (1998).
[26] Morihata, A.: Incremental computing with data structures, *Sci. Comput. Program.*, Vol.164, pp.18–36 (2018).
[27] Paige, R. and Henglein, F.: Mechanical Translation of Set Theoretic Problem Specifications into Efficient RAM Code-A Case Study, *J. Symb. Comput.*, Vol.4, No.2, pp.207–232 (1987).
[28] Paige, R. and Koenig, S.: Finite differencing of computable expressions, *ACM Trans. Program. Lang. Syst.*, Vol.4, No.3, pp.402–454 (1982).
[29] Pugh, W.: The Omega test: A fast and practical integer programming algorithm for dependence analysis, *Proc. Supercomputing '91*, pp.4–13 (1991).
[30] Pugh, W. and Teitelbaum, T.: Incremental Computation via Function Caching, *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pp.315–328, ACM (1989).
[31] Rauchwerger, L. and Padua, D.A.: The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization, *IEEE Trans. Parallel Distributed Syst.*, Vol.10, No.2, pp.160–180 (1999).
[32] Selva, M., Gruber, F., Sampaio, D., Guillon, C., Pouchet, L. and Rastello, F.: Building a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of Nonaffine Programs Scalable, *ACM Trans. Archit. Code Optim.*, Vol.16, No.4, pp.45:1–45:26 (2020).
[33] Simburger, A., Apel, S., Größlinger, A. and Lengauer, C.: PolyJIT: Polyhedral Optimization Just in Time, *Int. J. Parallel Program.*, Vol.47, No.5-6, pp.874–906 (2019).
[34] Sundaresh, R.S. and Hudak, P.: Incremental Compilation via Partial Evaluation, *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, Wise, D.S. (Ed.), pp.1–13, ACM Press (1991).
[35] Wang, Z., Tournavitis, G., Franke, B. and O'Boyle, M.F.P.: Integrating profile-driven parallelism detection and machine-learning-based mapping, *ACM Trans. Archit. Code Optim.*, Vol.11, No.1, pp.2:1–2:26 (2014).
[36] Yellin, D.M. and Strom, R.E.: INC: A Language for Incremental Computations, *ACM Trans. Program. Lang. Syst.*, Vol.13, No.2, pp.211–236 (1991).

松田 知樹

1995 年生．2018 年東京大学工学部化学生命工学科卒業．2021 年同大学大学院総合文化研究科修了，修士（学術）．

森畑 明昌　（正会員）

1981 年生．2004 年東京大学工学部計数工学科卒業．2006 年同大学大学院情報理工学系研究科修士課程修了．2009 年同研究科博士後期課程修了．同年日本学術振興会特別研究員，2010 年東北大学電気通信研究所助教，2014 年東京大学大学院総合文化研究科講師，2017 年同准教授となり現在に至る．博士（情報理工学）．プログラム変換，プログラム自動合成，関数プログラミング，並列プログラミング，アルゴリズムの系統的導出等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．