数理解析研究所講究録 1395 巻 2004 年 267-272

Risa/Asir の新グレブナー基底計算パッケージについて

野呂 正行
神戸大理*

1 開発の経緯

Risa/Asir においては、多項式は再帰表現または分散表現により保持される。後者はグレブナー基底関連計算における基本的なデータ形式であり、単項式を表す構造体 oMP の linked list である。

```
typedef struct oMP {
    struct oDL *dl;
    P c;
    struct oMP *next;
} *MP;
```

```
typedef struct oDL {
    int td;
    int d[1];
} *DL;
```

oMP の係数は c である。oDL のメンバー d は単項式の指数ベクトルを表しており、実際には変数の個数分の長さの配列がセットされる。再帰表現された多項式は分散表現に変換され、Buchberger, F_4, あるいは change of ordering などのアルゴリズムドライバにより処理される。

Risa/Asir のグレブナー基底計算においては、ペアの選択戦略、斉次化、モジュラ計算、効率的容量除去などさまざまな効率化の工夫を採り入れることにより、有理数体上での計算効率に関しては一定の評価を得てきたが、有限体上での計算においては、Singular の最近の版と比較すると大きく性能が劣っていた。また、有理数体上においても、多倍長演算に gmp を使用している Singular などのシステムでは、近年とみに高速化した gmp の性能と、Risa/Asir で使用している自主開発の多倍長演算機能との性能差により、必ずしも Risa/Asir の優位性が主張できなくなってきた。一方で、PC に搭載できるメモリ量も数 GB に達し、CPU もどんどん高速化し、グレブナー基底計算の応用範囲はどんどん大きくなっている。そこで、これまでのさまざまな経験および、実装に関する最近の知見をもとに、できる限り高速な分散表現多項式計算およびグレブナー基底計算を実現するパッケージ **nd** (New Distributed polynomial package) を新規に書くことにした。

2 効率化の工夫

Buchberger アルゴリズムに関しては、Gebauer-Moeller の useless pair detection、sugar strategy などにより、アルゴリズム的にはある程度固まったが、最近になっていくつか実装に関する提案がなされた。今回の実装に採り入れたものについて説明する.

1. geobucket

   これは、多項式の加算を効率化するための方法であり, [1] で提案され, Macaulay2, Singular など多くのシステムで採用され, 実際に効果があることが実証されている. 正規化計算では, 数多くの多項式の加算が行われるが, 非常に項数の多い多項式に, 項数の比較的少ない多項式を繰り

*noro@math.kobe-u.ac.jp

返し足すような場合, 項どうしの比較演算のコストが大変大きくなる. geobucket とは, 多項式を要素とする配列 g であって, 適当な整数 b (例えば 2) に対し, $g[i]$ の多項式が高々 b^i の項数を持つようなものである. g に, 項数 l ($b^{i-1} < l \leq b^i$) の多項式 p を足す場合, まず $g[i]+p$ を計算する. これの項数が b^i 以下ならそのまま $g[i]$ を置き換え, b^i より大きければ $g[i+1]$ に加える, という操作を geobucket の条件が満たされるまで続ける. これにより, 和に現われる多項式の項の総数を N とするとき, $O(N \log N)$ のコストで多項式の和が計算できる.

2. 可変長指数ベクトル

oDL のメンバーでは, 単項式の変数の各指数を 32 bit 固定で表現していたが, 多くの場合これは過分である. 必要最小限の bit 長を指数に割り当てることにすれば, 32 bit 中に複数の指数を保持でき, 比較, 和などを一度に複数個実行できる. また, 指数の保持に必要なメモリ量も減る. こうすると, 和であふれが生じる場合があるが, この場合にはサイズを変更して多項式を作りなおす. これは [2] で提案されている方法である.

3. 配列による多項式の保持

Buchberger アルゴリズムにおける基本操作は, $f - mg$ (f, g は多項式, m は単項式) である. これをさらに mg を作る操作と, 多項式の和を作る操作に分解して考える. 後者は geobucket により高速化が可能である. 前者に関しては, どうしようもなさそうにも思えるが, この演算は, g の表現方法により計算効率が大きく異なる. 結論を言えば, g が 単項式の linked list で表現されているより, 単項式 (これ自身配列である) の配列として表現されているほうが, mg の計算が高速である. g は, すでに計算された中間基底なので, 配列として保持することに問題はない. ただし, mg は, 多項式加算にまわるので, linked list として表示されていないと都合が悪い. 以上により, 多項式は状況に応じて linked list と配列の 2 つの表現をとることになった.

4. 関数のインライン化, unrolling

開発が進むについて, ボトルネックとなる部分が次第に低レベルな部分になっていった. 特に問題となるのが, 単項式の指数ベクトルに対する操作である. すなわち, 指数ベクトルの和, 差, 比較, divisibility などである. これらの部分に関しては, インライン化, および unrolling の是非を個別に実験により判断した.

5. reducer のサーチのハッシュ化

項 t を割り切る頭項をもつ中間基底 (reducer) g_i のサーチも, 他部分の効率化が進むにつれ, そのコストが問題になってきた. reducer としては, 経験上, i の小さい順から探して, t を割り切る最初のものを用いるのがよいとされる (例外もあるが). このため, t の reducer はあれば一意にきまる. t の reducer g_t が見つかったら, t のハッシュ値 h_t を計算して, ハッシュテーブルの h_t の位置に, (t, g_t) を登録する. t の reducer を探す際には, h_t の位置に登録されたデータから, t の reducer を探して, もしあればそれを用いればよい.

6. 斉次の場合の効率化

一般には, 新たに生成された中間基底で, 既存の中間基底の正規化は行わないが, 入力が斉次の場合には, ある (weight つき) 全次数の処理が終った時点でその次数の中間基底どうしで inter reduction を行う. この場合, 頭項は変化しないので, criteria への影響はなく, また, 低い全次数から順に中間基底を生成していれば, 既に, 現次数までの簡約グレブナー基底のすべての要素が得られているので, これまでに 0 に簡約された S-poly はやはり新しい基底でも 0 に簡約される. この処理を行うことにより, 以降の計算が簡約基底により正規化されることになり, 正規化が効率化されることが期待できる.

7. メモリ管理

計算途中, さまざまな大きさの領域が繰り返し必要となる. 特に多く必要とされるいくつかの構造体用領域は, garbage collector (GC) で得たものを自前でフリーリスト管理している. これは, GC による allocation, collection が一定のコストを伴うためである. この管理は nd パッケージ内で閉じており, かつフリーリストの root を 0 にしておけば, いずれ GC により回収される.

3 基本データ構造

分散表現多項式を保持するための構造体が二つ定義されている. ND は linked list 形式の, NDV は配列形式の多項式を表す. 前者は次で述べる oNM への, 後者は oNMV へのポインタを持っている.

```
typedef struct oNM {
    struct oNM *next;
    union oNDC c;
    UINT dl[1];
} *NM;
```

```
typedef struct oNMV {
    union oNDC c;
    UINT dl[1];
} *NMV;
```

これらは, 単項式を表すための構造体である. dl は単項式の指数ベクトルを表しており、実際には, 構造体作成時点での指数の bit 長と変数の個数に応じた長さの配列の大きさ分の領域が確保される. NM は linked list 形式の, NMV は配列形式の多項式における単項式を表す. NDV は, oNMV すなわち構造体そのものの配列へのポインタを持つ.

```
typedef union oNDC {
    int m;
    Q z;
    P p;
} *NDC;
```

```
typedef struct oRHist {
    struct oRHist *next;
    int index;
    int sugar;
    UINT dl[1];
} *RHist;
```

NDC は係数を保持するための汎用の共用体である. m は, 位数が 1 ワードで収まる有限体の元を保持するためのメンバーである. RHist は reducer の履歴をハッシュテーブルに登録するための構造体である. 各エントリは, RHist のリストとして登録される.

4 各部の詳細

4.1 ドライバ

Buchberger アルゴリズムのドライバは, nd_gb と nd_gb_trace の二つがある. nd_gb は, 任意の係数体上で, sugar ストラテジーつきの Buchberger アルゴリズムを実行するためのものである. ここでは,

1. S-pair リストのメンテナンス
2. S-pair の取り出し, 正規化計算の呼び出し
3. 正規形の, content 除去, NDV への変換
4. 指数にあふれが出た場合の, 中間基底の作りなおし

などが行われる. nd_gb_trace は, 有理数体, 有理関数体上のグレブナー基底計算を trace アルゴリズムにより行うためのものであり, 上記の仕事に加え, 結果をチェックする関数の呼び出し, homogenization, dehomogenization も行われる.

さらに, 現状では有限体上のみであるが, F_4 ドライバ nd_f4 も実装した. S-pair, 中間基底の扱いに関しては nd_gb と同様である. symbolic preprocessing は, 専用の geobucket が実装されている. F_4 の核心である, 複数の S-pair から, reducer をまとめて行列として掃き出す作業を行うまえに, 各 reducer により S-poly を正規化している. この操作を行うために, 各 reducer を, 圧縮ベクトル形式に変換しておき, 正規化される側の S-poly は非圧縮のベクトル形式として正規化を行う. 最後に, 残った部分を集めて行列とし, 掃き出しを行っている. これらにより, できる限り使用メモリ量を押えている.

4.2 指数ベクトルの変更

指数ベクトルの変更は, 指数の和であふれが生じたときに必要となる. これが起こり得るのは, S-poly の計算と, 正規形の計算における, 単項式と多項式の積の計算においてである. この中での, 単項式どうしの積の計算のたびにチェックするのは非効率的なので, 各中間基底に対し, 各変数に対する指数の最大値を記録しておき, そのベクトルとの和があふれを起こす場合に作りなおしをしている.

4.3 その他

dp 系で提供されているのと同様に, nd においても, 中間基底をディスク上の指定されたディレクトリに置くことができる. 指定方法は dp 系と同様 dp_gr_flags() で指定する. ファイルは dp 系と同様の形式なので, bload() で読むことができる. また, 有理数体上の場合, 正規化計算途中での content 除去は, 常に行われる. 現状では頭係数が 2 倍 (固定) になったときに除去が行われる。

5 性能

一般に, 有限体上の計算の場合, nd_gr は dp_gr_mod_main より数倍高速である. また, 問題にもよるが, nd_f4 は nd_gr の数倍程度高速な場合がある. おなじみの cyclic-n で比較すると表 1 のような結果を得る.

n	nd_gr	Singular	nd_f4	dp_gr_mod_main
7	5.1	5.0	1.8	17
8	124	135	34	564
9	27810	29725	3951	—

表 1: $GF(31991)$ 上での DRL 順序グレブナー基底計算 (cyclic-n)

このように, 少なくとも cyclic-n では, nd の実装の効果が十分に現われている. 表 2 は, cyclic-8 において, 各改良の効果がどの程度あるかを示す. 可変長指数ベクトルの採用と, inline 展開により, 倍以上高速化していることが見てとれる.

表 3 は, 種々のベンチマーク問題 [3] の計算時間を示す.

有理数体上の計算の場合, 多項式や, 指数ベクトルの表現方法以外に, 途中あらわれる係数の膨張の方が, 計算時間に大きく影響を与える場合が多い. この点では nd_gr_trace と dp_gr_main とでは大差ないので割愛するが, より悪くなることはない. 特に, weight を適切に設定することにより [4], 係数膨張に関してもより挙動のよい計算が可能となることに注意しておく.

可変長指数ベクトル	inline 展開	reducer hash	geobucket	計算時間 (秒)
×	×	×	×	390
○	×	×	×	240
○	○	×	×	187
○	○	○	×	155
○	○	○	○	124

表 2: 各改良の効果

	nd_gr	Singular	nd_f4
dl	5.9	4.9	4.0
eco10	7.1	10	3.1
eco11	63	106	23
eco12	507	1012	198
extcyc6	11	9.4	4.1
extcyc7	1813	1283	447
f855	3.6	3.4	2.5
filter9	0.28	0.80	3.2
hairer2	5.9	3.8	4.5
hairer3	11	35	*
hcyclic7	6.5	4.8	3.1
hcyclic8	213	163	82
hf744	1.1	1.1	1.6
hf855	25	25	17
ilias13	11	8.4	6.0
ilias_k_2	3.1	2.7	1.1

	nd_gr	Singular	nd_f4
ilias_k_3	4.4	2.9	1.2
katsura10	285	218	80
katsura8	4.1	3.3	1.3
katsura9	35	29	11
noon7	4.4	1.8	13
noon8	35	18	220
pinchon1	3.6	1.0	7.6
rbpl	1.0	0.89	1.2
redcyc7	3.5	3.3	1.2
redeco10	2.8	2.3	1.3
redeco11	24	18	12
redeco12	177	134	74
reimer6	11	32	10
reimer7	4000	4108	956
virasoro	1.8	1.4	0.65

表 3: $GF(31991)$ 上での DRL 順序グレブナー基底計算

6 今後の予定

dp 系にあって nd にない機能として, 有理関数体係数のグレブナー基底計算と, 有理数体上の F_4 計算がある. なるべく早いうちにこれらを実装したいと考えている. また, tangent cone アルゴリズムを用いた local ring での標準基底計算も, reducer を探す関数を新たに用意することで対応可能と考えている.

参 考 文 献

[1] Yan, T., The Geobucket Data Structure for Polynomials. Journal of Symbolic Computation, **25**, 3 (1998), 285-293.

[2] Schönemann, H., Singular in a Framework for Polynomial Computations. Joswig, M. and Takayama, N. (eds.), Algebra, Geometry, and Software Systems, Springer (2003), 163-176.

[3] `http://invo.jinr.ru/`. また `http://www.symbolicdata.org` にはさらに多くのベンチマーク問題がおいてある.

[4] 木村, 野呂, グレブナー基底計算のための weight 生成アルゴリズム. 本研究集会における発表 (2003).