Contents

Chapter 2

Common Issues

2.1 疎行列格納形式

J. Dongarra

本書で扱う多くの反復法の性能は主として行列ベクトル積の性能によって達成される。それゆえ、行列の格納形式にも影響を受ける。しばしば、使われる行列の格納形式は特定の応用問題から自然に生じたものである。

本節では ITPACK [263], NSPCG [345], and SPARSPAK [191], のような数値計算パッケージで使われているもっとも一般的な疎行列格納形式のいくつかと、BLAS Technical Forum (詳細は ETHOME を見よ) による新しい標準化ソフトウェアの一部としてとり入れらているものを概観する。. §2.2.2 では、２種類の疎行列格納形式によってどのように行列ベクトル積が計算されるかを示す。

行列 A が疎行列ならば、A のゼロ要素が更新も格納もされないとき、もっとも効率的に大規模固有値問題が計算できる。疎行列格納形式は、行列の非ゼロ要素ときわめて小数のゼロ要素をメモリ上の連続した領域に配置する。もちろん、疎行列格納形式はそれらの要素が行列全体のどこに収まるかを知る仕組みを持っている。

データの格納方法はいろいろである (たとえば、Saad [386] や Eijkhout [156] を見よ)。ここでは、圧縮行格納形式、圧縮列格納形式、ブロック行圧縮格納形式、対角格納形式、Jagged Diagonal 格納形式、そしてスカイライン格納形式について述べる。

2.1.1 圧縮行格納法（Compressed Row Storage, CRS 法）

圧縮行格納形式と圧縮列格納形式（次節）は最も一般的な形式で、行列の疎構造にいかなる仮定もしないし、，ひとつとして不要な要素も格納しない。一方，行列–ベクトル積または前処理解法では，それぞれのスカラー演算に対して間接アドレス参照を必要とするため、大変効率が良いというわけではない。

圧縮行格納形式（CRS 法）では行列の行内の非ゼロ要素をメモリ上に連続した配置する。非対称疎行列 A を仮定すると，１本の浮動小数点数ベクトル（`val`）、２本の整数ベクトル（`col_ind`, `row_ptr`）、計 3 本のベクトルがいる。`val` ベクトルは行列 A の非ゼロ要素を行方向に格納する。`col_ind` ベクトル

は val ベクトルの要素の列インデックスを格納する。つまり，val(k) $= a_{i,j}$ なら col_ind(k) $= j$ である。row_ptr ベクトルは val ベクトル上での行の開始位置を格納する。つまり，val(k) $= a_{i,j}$ なら row_ptr(i) $\leq k <$ row_ptr(i $+ 1$) である。便宜上，row_ptr(n $+ 1$) $= nnz + 1$ と定義する。ここで nnz は行列 A の非ゼロ要素数である。この方法による格納領域の節約は顕著であり、n^2 要素を格納する代わりに，たった $2nnz + n + 1$ の格納場所で済む。

ひとつの例として，次のような非対称行列 A を考える：

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix} . \tag{2.1}$$

この行列に対する CRS 形式は以下のような値の配列 {val，col_ind，row_ptr} で表される：

val	10	-2	3	9	3	7	8	7	$3 \cdots 9$	13	4	2	-1
col_ind	1	5	1	2	6	2	3	4	$1 \cdots 5$	6	2	5	6

row_ptr	1	3	6	9	13	17	20

行列 A が対称なら，行列の上三角部分または下三角部分だけを格納すればよい。トレードオフはいくぶん異なるデータアクセスパターンを持つより複雑なアルゴリズムになることである。

2.1.2 圧縮列格納形式（Compressed Column Storage, CCS 法）

圧縮行格納形式と同様，圧縮列格納形式法（CCS 法）という方法もある。この方法は *Harwell-Boeing* 疎行列形式 [139] とも呼ばれている。CCS 形式では行方向ではなく列方向に A が格納されている以外は CRS 形式と同じである。言い換えると，CCS 形式は A^T に対する CRS 形式である。

CCS 形式は 3 つの配列 {val，row_ind，col_ptr} によって表される。row_ind は非ゼロ要素の行インデックスを格納し，col_ptr は val ベクトル上での列の開始位置を格納する。(2.1) の行列 A に対する CCS 形式は以下のようになる：

val	10	3	3	9	7	8	4	8	$8 \cdots 9$	2	3	13	-1
row_ind	1	2	4	2	3	5	6	3	$4 \cdots 5$	6	2	5	6

col_ptr	1	4	8	10	13	17	20

2.1.3 ブロック圧縮行格納法（Block Compressed Row Storage, BCRS 法）

疎行列 A が非ゼロ要素からなる正方密ブロックの規則的なパターンからできているなら，そのようなブロックパターンを活用するように CRS 格納形式（または CCS 格納形式）を修正できる。典型的なブロック行列はひとつの格子点に複数の自由度 *degrees of freedom* を持つ偏微分方程式の離散化からでてくる。

そこで自由度に等しい大きさの小さなブロックに分け，いくつかのゼロを含んでいたとしてもそれらのブロックを密行列として扱う。

n_b を各ブロックの次元，$nnzb$ を $n \times n$ 行列 A 中の非ゼロブロック数とすると，全体で必要な格納領域は $nnz = nnzb \times n_b^2$ である。A のブロック次元 n_d は $n_d = n/n_b$ として定義される。

CRS 形式と同様に，BCRS 形式も 3 つの配列を必要とする：浮動小数点数のための長方形配列（`val`$(1 : nnzb, 1 : n_b, 1 : n_b)$）には（ブロックの）行方向にそって非ゼロブロックを、整数配列（`col_ind`$(1 : nnzb)$）には各非ゼロブロックの $(1, 1)$ 要素の元の行列 A における列インデックスを、そしてポインタ配列（`row_blk`$(1 : n_d + 1)$）には `val(:,:,:)` と `col_ind(:)` 中のそれぞれのブロックのある行開始位置を格納している。BCRS 法における格納位置と間接アドレス付けに要する計算時間は、大きな n_b を持つ行列に対して大きく削減できる。

2.1.4 圧縮対角格納法（Compressed Diagonal Storage, CDS 格納形式）

行列 A のそれぞれの行がほとんど一定の帯幅を持つ帯行列であるとき，行列の副対角要素を連続した場所に格納する格納形式を用いてこの構造を利用することには大きな利点がある。列や行を識別するベクトルを排除できるだけでなく，行列–ベクトル積がより効率良く実行するよう非ゼロ要素を詰め込むことができる。特にこの格納形式は，行列がテンソル積格子上の有限要素離散化あるいは有限差分離散化で作られたときに役立つ。

左帯半幅，右帯半幅と呼ばれる非負の定数 p, q に対して、$i - p \le j \le i + q$ のときにのみ $a_{i,j} \neq 0$ となるなら，行列 $A = (a_{i,j})$ を帯行列という。この場合，行列 A に配列 `val(1:n,-p:q)` を割り当てる。行と列を逆にした宣言 `(-p:q,n)` は `LINPACK` の帯形式 [132] に対応している。CDS 法とは異なり，`LINPACK` の帯形式は `p + q` が小さいときは行列–ベクトル積の効率的なベクトル化ができない。

通常，帯形式はいくつかのゼロを格納している。CDS 形式は行列要素にまったく対応していない配列要素を含んでさえいる。以下に定義された非対称行列 A を考える：

$$A = \begin{bmatrix} 10 & -3 & 0 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 0 & 0 & 8 & 7 & 5 & 0 \\ 0 & 0 & 0 & 9 & 9 & 13 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{bmatrix}. \tag{2.2}$$

CDS 形式では，射影：

$$\texttt{val}(\texttt{i}, \texttt{j}) = a_{i,i+j}. \tag{2.3}$$

を用いて、行列 A を `(6,-1:1)` 次元の配列に格納する。それゆえ，配列 `val(:,:)` の行は

`val(:,-1)`	0	3	7	8	9	2
`val(:, 0)`	10	9	8	7	9	-1
`val(:,+1)`	-3	6	7	5	13	0

となる。存在しない行列要素に対応した 2 つのゼロに注意せよ。

2.1.5 Jagged Diagonal 格納形式（Jagged Diagonal Storage, JDS 形式）

Jagged Diagonal 格納形式は並列プロセッサやベクトルプロセッサ上で反復法を実現するのに役立つ（Saad [385] 参照）。CDS 形式のように，本質的には行列サイズと同じ長さのベクトル長を与える。集積/散布演算のコストの点で CDS 法よりもより格納効率が良い。

ITPACK 格納形式や Purdue 格納形式と呼ばれる JDS 法を単純化した形式ものは以下のように記述できる。先ほどの非対称行列では，すべての非ゼロ要素が左寄せされ，

$$\begin{bmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{bmatrix} \longrightarrow \begin{bmatrix} 10 & -3 & 1 & \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 9 & 13 & & \\ 5 & -1 & & \end{bmatrix}$$

となり，その後各列が連続して格納される。すべての行には長さを等しくするため右にゼロが詰められる。行列要素の配列 `val(:,:)` に対応して，列インデックスの配列 `col_ind(:,:)` も格納される。

`val`$(:,1)$	10	9	3	6	9	5
`val`$(:,2)$	-3	6	8	7	13	-1
`val`$(:,3)$	1	-2	7	5	0	0
`val`$(:,4)$	0	0	0	4	0	0

,

`col_ind`$(:,1)$	1	2	1	2	5	5
`col_ind`$(:,2)$	2	3	3	4	6	6
`col_ind`$(:,3)$	4	5	4	5	0	0
`col_ind`$(:,4)$	0	0	0	6	0	0

.

この構造でゼロを詰めることは，特に行列の帯幅が大きく変わるときに不利になりかねないことは明らかである。それゆえ，JDS 形式では、行あたりの非ゼロ要素数がだんだんと少なくなるよう行列の行を並べ換える。圧縮されて，入れ換えられた対角要素は 1 次元配列に格納される。その新しいデータ構造が *jagged diagonals* と呼ばれる。

特に、 すべての最初は `val` に（密）ベクトルを格納する。`col_ind` にはそれぞれの行における対応する要素の列インデックスを格納する。2 番目の Jagged Doagonal が左から 2 番目に対応した場所に入る。これらの Jagged Diagonal をもっともっと（その長さの降順に）つなげていく。

Jagged Diagonal の数は第 1 行目の非ゼロ要素の数，すなわち A のすべての行の中の非ゼロ要素の最大数に等しい。それゆえ $n \times n$ の行列 A を表すためのデータ構造は，行の番号付けを変える置換配列（`perm(1:n)`），それぞれの Jagged Diagonal を連続して格納する浮動小数点数の配列（`jdiag(:)`），対応した列インデックスを含む整数配列（`col_ind(:)`），そして各 Jagged Diagonal の開始位置を示す要素を持つポインタ配列（`jd_ptr(:)`）からなる。行列の積に関する JDS 法の利点は Saad [385] で議論されている。

先ほどの行列 A に対する 1 次元配列 {`perm`, `jdiag`, `col_ind`, `jd_ptr`} を用いた JDS 形式は以下のようになる（Jagged Diagonal をセミコロンで区切る）：

jdiag	6	10	9	3	9	5;	7	-3	1 ··· -1;	4	1	-2	7;	4
col_ind	2	1	2	1	5	5;	4	2	3 ··· 6;	5	4	5	4;	6

perm	4	1	2	3	5	6

jd_ptr	1	7	13	17

.

2.1.5.1 スカイライン格納形式（Skyline Storage, SKS 法）

われわれの考える最後の格納形式ものはスカイライン行列用のもので、可変帯行列，またはプロファイル行列とも呼ばれている（Duff, Erisman,and Reid [138] 参照）。この手法の重要性の多くは直接法に対してであるが，ブロック行列分解で対角ブロックを扱うのにも使用できる。スカイライン行列を係数持つ線形方程式を解くいちばんの利点は，軸選択が必要ないとき、スカイライン構造がガウス消去法の間を通じて保たれることにある。行列が対称なら，行列の下三角部分のみを格納すればよい。スカイライン行列の要素を格納するための直接的なアプローチでは，浮動小数点数配列（`val(:)`）にすべての行を順番に置き，それから各行の開始位置を示す整数配列（`row_ptr(:)`）を確保する。`val(:)` に格納されている非ゼロ要素の列インデックスは容易に導きだせるので格納しない。

図 2.1 に示したような非対称なスカイライン行列に対しては，下三角部分の要素を SKS 形式，上三角行列部分の要素を列方向の SKS 形式（転置行列を行方向の SKS 形式で格納したもの）で格納する。これら 2 つに分けられた部分構造は様々な方法で連結できる。Saad [386] によって議論されたひとつのアプローチは，下三角部分の行と上三角部分の列を浮動小数点数配列（`val(:)`）の中で続けて格納することである。そのときは下三角部分の要素と上三角部分の要素を分けている対角要素がどこにあるかを示すための付加的なポインタが必要になる。

2.2 行列–ベクトル積と行列–行列積

J. Dongarra, P. Koev, and X. Li

2.2.1 BLAS

ここ 15 ほど、線形代数問題のアルゴリズムと解法ソフトウェアの分野で大きな動きがあっｔ。線形代数のコミュニティは、長いこと、アルゴリズムからソフトウェア開発までに何らかの支援が必要なことを認識していた。何年か前、コミュニティの努力の成果として、線形代数アルゴリズムとソフトウェアで必要とされる基本的な操作をデファクトスタンダードとしてまとめた。計画では、基本線形代数プログラム (Basic Linear Algebra Subprograms, BLAS) として知られる標準として集められたルーチンが、多くのメーカによって最新のアーキテクチャのコンピュータで効率的な実装がなされ、広範囲のマシンで効率的な実装がなされることで移植性という利点を得られるはずでした。目標の多くは達成されました。

最新のアーキテクチャのコンピュータに対する線形代数アルゴリズムを設計する際の鍵となるわれわれの考えは、高性能を達成するためには異なるメモリ階層間のデータ移動を最小化するである。したがって、われわれの実装でベクトル化と並列性を見いだすためのアルゴリズムアプローチは、特に高度に最適化された行列–ベクトルと行列–行列演算カーネル (レベル 2 BLAS とレベル 3 BLAS) のカーネルを併用したブロック化アルゴリズムの使用である。一般に、ブロック化アルゴリズムではベクトルやスカラー単

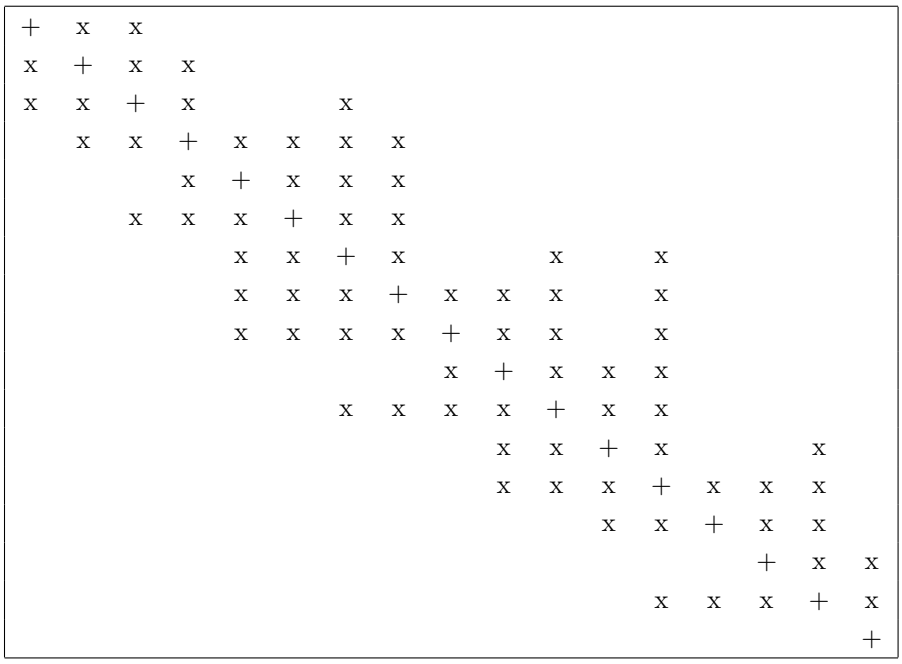

Figure 2.1: 非対称スカイライン行列あるいは可変帯行列のプロファイル

位でなく、ブロック単位でのデータ移動を必要とするが、データの総移動量は変わらず、データ移動に必要なメッセージが大幅に減るため、データ移動に伴うレーテンシイ（立ち上がりに要するコスト）が大きく削減される。

もうひとつの鍵となるアイデアは、データ配置を特定するパラメータをユーザが変化させることでアルゴリズムのパフォーマンスを調整することです。これは、共有メモリマシンではブロックサイズの制御、分散メモリマシンではブロックサイズと論理プロセスグリッド (logical process mesh) の構成の制御になります。

「マシンが提供するレベルの高性能を得るために、性能に関係するこれらのいろいろな要素をどのように制御すればよいのか」という疑問がわくでしょう。疑問に対する答えは、「きちんと BLAS を使うこと」です。

3つのレベルの BLAS が存在します：

Level 1 BLAS [288]: ベクトル演算 $y \leftarrow \alpha x + y$;

Level 2 BLAS [133]: 行列–ベクトル演算 $y \leftarrow \alpha Ax + \beta y$;

Level 3 BLAS [134]: 行列–行列演算 $C \leftarrow \alpha AB + \beta C$.

ここで、A, B, C は行列; x と y はベクトル; α と β はスカラー。

レベル 1 BLAS は LAPACK のようなパッケージで、性能というよりは利便性のために使われ、計算のわずかな部分を実行する。現代的なスーパーコンピュータの多くでは高性能が発揮できない。

レベル 2 BLAS は多くのベクトルコンピュータ、たとえば CRAY X-MP や Y-MP, Convex C-2 のようなシングルプロセッサで最高性能に近い性能を発揮する。いっぽう、CRAY-2 や IBM 3090 VF のような他のベクトルベクトルコンピュータにおけるレベル 2 BLAS の性能は異なるメモリ階層間のデータ移動性能に制約される。

レベル 3 BLAS はこの制限をうけない。この 3 番目の BLAS は、レベル 2 BLAS が $O(n^2)$ のデータに対して $O(n^2)$ の演算 s じか実行できなかったの対し、$O(n^2)$ のデータに対して $O(n^3)$ の浮動小数点演算を実行する。レベル 3 BLAS は、レベル 3 BLAS を呼び出すソフトウェアから透過的な形で並列性を見いだしてくれる。レベル 2 BLAS が狭い範囲での並列性を提供するの対し、レベル 3 BLAS は広い範囲での並列性を提供する。

ユーザーコミュニティは、パフォーマンス上の理由だけでなく、BLAS のような共通のルーチンの核を中心にソフトウェア開発をすることがソフトウェアエンジニアリングの優れた実践であるため、BLAS の多くを受け入れました。マシン固有で高度に効率的な BLAS の実装は、最新の高性能コンピュータのほとんど利用できます。BLAS によって、ポータブルなコードで高性能を実現するソフトウェアが可能になりました。

初期の BLAS ミーティングの精神と、Massing-Passing Interface（MPI）および High-Performance Fortran（HPF）フォーラムの標準化努力により、最新のソフトウェア、言語、 およびハードウェア開発における BLAS の拡張を考える技術フォーラムが設立されました。。BLAS テクニカルフォーラム会議は、テネシー大学における 1995 年 11 月のワークショップで始まりました。会議は大学とソフトウェアおよびハードウェアのベンダーによってホストされました。

全体的な機能、言語インターフェイス、スパース BLAS、分散メモリ密行列 BLAS、拡張および混合精度 BLAS、区間演算 BLAS、既存の BLAS に対する拡張などの問題を検討するため、さまざまなワーキンググループが設立されました。フォーラムのルールは、MPI フォーラムおよび HPF フォーラムで使用されているルールが使われました。

このドキュメントに定義された標準化の主な目的は、線形代数ライブラリ（パブリックドメインと商用の両方）を効率的で、信頼性があり、簡単に相互運用できるようにすることです。われわれは、ハードウェアとソフトウェアのベンダー、ハイレベルのライブラリ作成者、およびアプリケーションプログラマーは、すべてこのフォーラムの努力から便宜を得ており、これらの標準化が想定しているエンドユーザーであると考えています。

BLAS と BLAS テクニカルフォーラムに関する詳細は本書のホームページ ETHOME にある。

2.2.2 スパース BLAS

密行列に対する BLAS の精神に基づいて、BLAS テクニカルフォーラムでは、スパース BLAS の標準を確立するための作業が進行中です。スパース BLAS インターフェイスは、 rm unstructured スパース行列向けの計算ルーチンとなります。スパース BLAS は、密行列の場合と同様、3 つのレベルの演算が含まれています。ただし、密行列の BLAS の小さなサブセットのみが指定されています：

Level 1: 疎な内積、ベクトル更新、集約／分散;

Level 2: 疎行列–ベクトル積、三角解法;

Level 3: 疎行列–密行列積、複数右辺ベクトルに対する三角解法.

これらは線形方程式や固有値問題に対する反復解法の多くで使われる基本演算である。レベル 2 とレベル 3 スパース BLAS のインターフェースは、９種類の疎行列格納形式をサポートする。９種類の格納形式は、CRS 形式のような要素エントリー形式 (*point entry* formats) と BCRS 形式のようなブロックエントリー形式 (*block entry* formats) に２分される。ブロックエントリー形式では、疎構造は小さな密行列ブロックの列として表現される。§2.1 で概観した格納形式で、JDS 形式と SKS 形式以外は９種類の格納形式に含まれていることに注意せよ。

スパース BLAS のインターフェイス設計は、密行列の BLAS とは根本的に異なります。任意のスパース行列操作に対して、唯一の「最適な」格納形式は存在しないため、レベル 2 およびレベル 3 のスパース BLAS のスパース行列引数はある特定の格納形式を使用しません。そのかわりに、これらのルーチンに対して総称インターフェース (*generic interface*) が定義されており、そこではスパース行列に対する引数は行列を表すハンドル (*handle*) (整数) となっています。スパース BLAS ルーチンを呼び出す前に、作成ルーチンを呼び出して、9 つの形式のいずれかの格納形式のスパース行列ハンドルを作成する必要があります。スパース BLAS ルーチンを呼び出した後、クリーンアップルーチンを呼び出して、行列ハンドルに関連付けられたリソースを解放する必要があります。このハンドルベースのアプローチにより、疎行列格納形式とは関係なくスパース BLAS が使用できます。内部表現は実装依存で、最高のパフォーマンスが得られるように選択できます。

行列–ベクトル積は反復法に費やされる時間のほとんどを占めることが多いため、いくつかの研究でパフォーマンスの最適化が試みられました [438, 439, 240, 239]。行列–ベクトル積では、行列の各エントリはたった１回だけの演算に関与しますが、ベクトルの各エントリは複数回の演算に関与することがあります。最適化の主な目標は、異なるレベルのメモリ階層間でのソースベクトルのデータ移動量を減らすことです。最適化手法には、行列要素の並べ替え (matrix reordering)、キャッシュブロッキング、レジスタブロッキングなどがあります。SPARSITY [240, 239] と呼ばれる最近開発されたツールボックスには、これらすべてのテクニックが含まれています。マトリックス構造にもよるが、SPARSITY の作者は単一プロセッサ上で最大 3 倍の高速化を実現しています。SPARSITY には、行列とマシンの特性に基づいて最適なブロックサイズを自動選択するメカニズムも含まれています。

これまでに議論した多くの反復解法の多くで、行列とベクトルの積、転置行列とベクトルの積が必要になります。すなわち、入力ベクトル x に対して、以下の積

$$y = Ax \qquad \text{and} \qquad y = A^T x.$$

を計算することになります。§2.1 で述べた格納形式 CRS 形式に対するアルゴリズムをつぎの２節で示そう。

2.2.2.1 CRS 形式の行列–ベクトル積

CRS 形式を用いた行列–ベクトル積 $y = Ax$ は，行列 A が行方向に格納されているため，普通の方法で

$$y_i = \sum_j a_{i,j} x_j,$$

と表され、$n \times n$ 行列 A に対して，行列–ベクトル積は以下のようになる：

```
for i = 1, n
    y(i)  = 0
    for j = row_ptr(i), row_ptr(i+1) - 1
        y(i) = y(i) + val(j) * x(col_ind(j))
    end;
end;
```

この手法は行列の非ゼロ要素に対する乗算のみなので，演算回数は A の非ゼロ要素数の 2 倍になる。これは，$2n^2$ が必要な密行列に対する演算に比べると大きな節約となる。

転置行列積 $y = A^T x$ には式

$$y_i = \sum_j (A^T)_{i,j} x_j = \sum_j a_{j,i} x_j,$$

は使えない。なぜなら、これは行列を列方向に参照することを意味していて，CRS 形式で格納された行列に対しては極端に効率の悪い操作になるからである。そこで，添字を以下のように変更する。

$$\text{for all } j, \text{ do for all } i: \qquad y_i \leftarrow y_i + a_{j,i} x_j.$$

A^T を含む行列–ベクトル積は以下のようになる：

```
for i = 1, n
    y(i) =  0
end;
for j = 1, n
    for i = row_ptr(j), row_ptr(j+1)-1
        y(col_ind(i)) = y(col_ind(i)) + val(i) * x(j)
    end;
end;
```

上で述べた両方の行列–ベクトル積はおおよそ同じ構造をしていて，両方とも間接番地付けを用いている。ゆえに，ベクトル化の可能性はいかなる計算機上でも同じである。しかしながら，最初の積 ($y = Ax$) は外側ループの 1 反復あたり 2 つのベクトルデータ（行列 A の行と入力ベクトル x）を読み込み，ひとつのスカラーを書き出すという点でより有利なメモリ参照パターンになっている。一方，転置行列の積 ($y = A^T x$) は入力ベクトルの 1 要素と行列 A の 1 行を読み込み，結果のベクトル y に対して読み込みと書き出しの両方をする。これらの手法が 3 つの独立なメモリパスを持つ（すなわち Cray のベクトルコンピュータのような）コンピュータに実装されているのでなければ，メモリ参照が実行性能を制限することになる。これは RISC に基づいたアーキテクチャでは重要な考慮事項である。

2.2.2.2 CDS 形式の行列–ベクトル積

$n \times n$ 行列 A が CDS 形式で格納されていれば、依然として行ごとのまたは列ごとの行列–ベクトル積 $y = Ax$ が実行できるが、しかしこれで CDS 形式の利点は活かせない。アイデアは 2 重にネストしたルー

プ中の座標を変化させることにある。$j \to i+j$ と置き換えて以下の式を得る：

$$y_i \leftarrow y_i + a_{i,j}x_j \quad \Rightarrow \quad y_i \leftarrow y_i + a_{i,i+j}x_{i+j} \ .$$

内側ループで添字 i を用いると，式 $a_{i,i+j}$ は行列の j 番目の副対角要素を参照することがわかる（ここで主対角は 0 番目である）。

今，アルゴリズムは外側ループが `diag=-p,q` と動く 2 重にネストしたループを持つ。p と q は非負で、主対角要素の左側，右側の要素数を示す。内側ループの範囲は

$$1 \le \mathtt{i}, \mathtt{i}+\mathtt{j} \le n.$$

から導かれ、アルゴリズムは以下のようになる：

```
for i = 1, n
    y(i) = 0
end;
for diag = -diag_left, diag_right
    for loc = max(1,1-diag), min(n,n-diag)
        y(loc) = y(loc) + val(loc,diag) * x(loc+diag)
    end;
end;
```

転置行列–ベクトル積 $y = A^T x$ は上記アルゴリズムのちょっとした変形である。更新の式

$$\begin{aligned} y_i &\leftarrow y_i + a_{i+j,i}x_j \\ &= y_i + a_{i+j,i+j-j}x_{i+j} \end{aligned}$$

を使えば、次のアルゴリズムを得る:

```
for i = 1, n
    y(i) = 0
end;
for diag = -diag_right, diag_left
    for loc = max(1,1-diag), min(n,n-diag)
        y(loc) = y(loc) + val(loc+diag, -diag) * x(loc+diag)
    end;
end;
```

CDS 形式に基づいた行列–ベクトル積 $y = Ax$ (あるいは $y = A^T x$) のメモリ参照は，内側の反復あたり 3 本のベクトルである。一方，間接番地付けはないし，アルゴリズムは本質的に行列の次元 n のベクトル長でベクトル化可能である。規則的なデータ参照のため，ほとんどの計算機は 3 つのベースレジスタを確保し，単純なオフセット番地付けを用いることで、このアルゴリズムを効率良く実行できる。

2.2.3 構造化行列に対する高速な行列–ベクトル積

実用上、しばしば $O(n)$ 個のパラメータから決まる密行列が現れる。例として、以下の行列が挙げられる。

ヴァンデルモンド行列：

$$V = \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_n \\ x_1^2 & x_2^2 & \dots & x_n^2 \\ & & \ddots & \\ x_1^{n-1} & x_2^{n-1} & \dots & x_n^{n-1} \end{bmatrix},$$

テプリッツ行列：

$$T = \begin{bmatrix} t_0 & t_{-1} & \dots & t_{2-n} & t_{1-n} \\ t_1 & t_0 & t_{-1} & & t_{2-n} \\ \vdots & t_1 & t_0 & \ddots & \vdots \\ t_{n-2} & & \ddots & \ddots & t_{-1} \\ t_{n-1} & t_{n-2} & \dots & t_1 & t_0 \end{bmatrix},$$

ハンケル行列： $H = (H_{ij}) = (c_{i+j})$,

コーシー行列： $C = (C_{ij}) = (\frac{1}{x_i - y_j})$, その他 [257] である。

これらの行列とその逆行列のベクトル積は $O(n \log^k n)$ 時間で実行できます。構造によって異なりますが、$O(n^2)$ または $O(n^3)$ 時間でなく $0 \le k \le 2$ になります。このことは、これらの行列に対して反復ソルバを使用する場合に特に便利です。$(A + B)x = b$ という形式のシステムで A と B の両方が構造化されているが、別の種類の構造化クラスに属している場合、あるいは A が構造化行列で B が帯行列や疎行列などの場合を反復ソルバで解く場合にも役立ちます。反復法は、ブロックが構造化行列であるが、一部のブロックが疎である可能性のある異なる構造化クラスに属している場合のブロックシステムを解くのにも役立ちます。たとえば、積

$$\begin{bmatrix} A & B \\ C & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} Ax + By \\ Cx \end{bmatrix}$$

は、 A が対角行列で、 B と C がテプリッツ行列のときには $O(n \log n)$ 時間で構成できる.

ヴァンデルモンド行列とヴァンデルモンド行列の逆行列とベクトルの積は $O(n \log^2 n)$ [196, 195] 時間で計算できる。同じことは、ヴァンデルモンドもどきの行列 (Vandermonde-like) とその逆行列とベクトルの積にもいえ、ここでもどきの行列 ("-like" matrices) とは § 2.3.4 で定義されている。テプリッツ行列とテプリッツもどきの行列に対する行列ベクトル積は、高速フーリエ変換 (FFT) を使って $O(n \log n)$ 時間で計算できる [198]。コーシー行列に対する行列ベクトル積 Cz は、関数

$$\Phi(w) = \sum_{i=1}^{n} \frac{-z_i}{y_i - w}$$

n 個の異なる点 $x_1, x_2, \dots, x_n$ の評価と同等であり、高速多重極展開法用いて $O(n)$ 時間で計算できる [76, 205].

この節の残りの部分では、テプリッツ行列に対する行列ベクトル積が、どのようにして $O(n \log n)$ でできるかを示す [198]。その他の構造化行列に対しては、[196] と [195] を示して読者にお任せしよう。

循環行列 (circulant matrix) は以下の形をした特殊なテプリッツ行列である：

$$C_n = \begin{bmatrix} a_0 & a_{-1} & \dots & a_{2-n} & a_{1-n} \\ a_1 & a_0 & a_{-1} & & a_{2-n} \\ \vdots & a_1 & a_0 & \ddots & \vdots \\ a_{n-2} & & \ddots & \ddots & a_{-1} \\ a_{n-1} & a_{n-2} & \dots & a_1 & a_0 \end{bmatrix},$$

ここで $1 \le k \le n-1$ に対して $a_{-k} = a_{n-k}$ という性質を持つ. 循環行列は FFT で対角化できる。すなわち

$$C_n = F_n^* \cdot \mathrm{diag}(F_n a) \cdot F_n,$$

ここで $a = [a_0, a_1, \dots, a_n]$、$F_n$ はフーリエ行列 $F_n(j,k) = \frac{1}{\sqrt{n}} e^{-(j-1)(k-1)2\pi\sqrt{-1}/n}$ である. よく知られているように、 F_n はユニタリ行列で、その積 $F_n x$ は $O(n \log n)$ 時間で構成できる [453]. 以下の４ステップにしたがって、積 $y = C_n x$ もまた $O(n \log n)$ 時間で構成できる:

(1) $f = F_n x$,
(2) $g = F_n a$,
(3) $z^T = [f_1 g_1, f_2 g_2, \dots, f_n g_n]$,
(4) $y = F_n^* z$.

いま、T_n をテプリッツ行列

$$T_n = \begin{bmatrix} t_0 & t_{-1} & \dots & t_{2-n} & t_{1-n} \\ t_1 & t_0 & t_{-1} & & t_{2-n} \\ \vdots & t_1 & t_0 & \ddots & \vdots \\ t_{n-2} & & \ddots & \ddots & t_{-1} \\ t_{n-1} & t_{n-2} & \dots & t_1 & t_0 \end{bmatrix},$$

とすれば、積 $T_n x$ は、 T_n を $2n \times 2n$ の循環行列 C_{2n} へ埋め込むことによって $O(n \log n)$ 時間で計算できる。なぜなら、

$$C_{2n} \cdot \begin{bmatrix} y \\ 0 \end{bmatrix} \equiv \begin{bmatrix} T_n & B_n \\ B_n & T_n \end{bmatrix} \cdot \begin{bmatrix} y \\ 0 \end{bmatrix} = \begin{bmatrix} T_n y \\ B_n y \end{bmatrix}$$

と

$$B_n = \begin{bmatrix} 0 & t_{n-1} & \dots & t_2 & t_1 \\ t_{1-n} & 0 & t_{n-1} & & t_2 \\ \vdots & t_{1-n} & 0 & \ddots & \vdots \\ t_{-2} & & \ddots & \ddots & t_{n-1} \\ t_{-1} & t_{-2} & \dots & t_{1-n} & 0 \end{bmatrix}.$$

だからである。

2.3 直接解法の概略

J. Demmel, P. Koev, and X. Li

本書で最も効果的な方法の多くは、特にシフトと反転を用いる場合、$(A - \ sigmaI)\ x = b$ または $(A - \ sigmaI)^* x = b$ の形の連立一次方程式の解法が必要になります。ここで $\ sigma$ はシフト点 (em shift) で、通常は A の近似固有値に選びます。この方法ではほとんどの時間が連立一次方程式の解法に費やされ、したがって、利用可能な最善の解法を使うことか重要となります。

$(A - \ sigmaI)\ x = b$ を解くには、直接法と反復法の 2 つの基本アプローチがあります。直接法では、通常、ガウスの消去法の変形を用います。 $\ sigma$ を固有値に近くに選べば、$A - \ sigmaI$ はほぼ特異になり、多くの反復法が収束の問題を起こすような場合でも効果的です。セクション ref sec：IterativeSolver で反復法と、Jacobi–Davidson 法のような反復法が効果的に使用できるアルゴリズムについて述べます。ここでは直接的法のみを検討します。

直接法では、$(A - \sigma I)x = b$ を次の 2 ステップで解きます：

1. $A - \sigma I$ の分解 (*factorization*). この部分はもっともコストがかかる部分になります。

2. 分解結果を使った $(A - \sigma I)x = b$ あるいは $(A - \sigma I)^* x = b$ の計算.

通常、ステップ 1 はステップ 2 より大幅なコスト増となる (密行列の場合、ステップ 2 が $O(n^2)$ に対してステップ 1 は $O(n^3)$)。多くの異なる右辺に対して $(A - \sigma I)x = b$ を解く場合は分解は再利用される。分解はシフト点 σ 変更された場合に限って再計算される。幸いなるかな、多くに反復解法は同一のシフト点に対して異なる右辺を持つ連立一次方程式を何回も解くので、ステップ 2 に対してステップ 1 はたいへん少ない回数の計算ですむ。

ステップ 1，2 のアルゴリズムの選択は、 $A - \sigma I$ の数学的な構造 (*mathematical structure*) と行列 A の格納形式 (*storage structure*) の両方に依存する。数学的構造の観点では、$A - \sigma I$ がエルミートかエルミートでないか、定値かそうでないかを意味する。

エルミートで定値の場合*:* この場合は、コレスキー分解 (*Cholesky factorization*) $A - \sigma I = \pm LL^T$, L は下三角行列を計算します。符号は、$A - \sigma I$ が正定値ならば (+), 負定値ならば (−) とします。疎行列の場合は置換行列 P を用いたコレスキー分解 $A - \sigma I = \pm PLL^* P^*$ を計算します。

エルミートだが定値でない場合*:* この場合は、*Hermitian indefinite factorization* $A - \sigma I = PLDL^* P^*$ を計算します。L は下三角行列、P は置換行列、D は 1 行 1 列と 2 行 2 列のブロックをもつブロック対角行列、あるいは 3 重対角行列である。

エルミートでない場合*:* この場合は、三角分解 (*triangular factorization*) $A - \sigma I = P_1 LU P_2$ を計算する。ここで、L は下三角行列、U は上三角行列、P_1 と P_2 は置換行列である (しばしば、片方の P_i は省略される)。

A がテプリッツ行列のときは $a_{i,j}$ の値は $i - j$ にのみ依存するなど、他にも同じような醸造や分解がある.

格納形式は、密、帯、疎 (§2.1 の節で説明した格納形式いずれか)、あるいは構造化（他の要素が決定できるよう、テプリッツ行列の最初の行と列を保存するなど) などを意味します。 §2.1 の節で述べたように、疎な非エルミート行列も疎なエルミート格納形式として格納できる。

数学的構造と格納形式の多くの組み合わせに特化したアルゴリズムとソフトウェアがあり、適切なアルゴリズムを選択すると、大規模行列問題では計算時間には数桁の差が生じる可能性がある。この節では、最良のアルゴリズムとソフトウェアを要約します。多くの場合、研究論文で最良とされたアルゴリズムは、適切に設計されて簡単にアクセス可能なソフトウェアとはなっていないため、利用可能なソフトウェアに注目します。利用可能なソフトウェアがないのであれば、十分に管理されたパブリックドメインのソフトウェア、低コストで利用できるソフトウェアをお勧めします。

われわれは 4 つのケース、密行列に対する方法、帯行列に対する方法、疎行列に対する方法、およびテプリッツ行列のような構造化行列に対する方法を検討します。

2.3.1 密行列向け直接解法

変換法を使用する代わりに、密行列に対する反復法を検討する理由は 2 つあります。

1. ひとつもしくはいくつかのシフト点 σ の近くにある少数の固有値、あるいは少数の固有値・固有ベクトルが必要な場合、変換法を使うのでなく、反復スキームによるシフトと反転 shift-and-invert を使うほうが安価である。変換法がより高速なエルミート行列よりも非エルミート行列の場合によりあてはまります。当てはまります。

2. 行列の疎度が小さい（スパースでない）場合や、行列が巨大でない場合、密行列に対する直接法はスパースソルバよりも高速である。

密行列に対する解法の選択は、以下に述べる $A - \sigma I$ の数学的な構造に依存する。

$A - \sigma I$ がエルミート定値 この場合はコレスキー分解を選択する。LAPACK では、分解が `xPOTRF`、分解を使った求解が `xPOTRS` として実装されている (両者は LAPACK driver routine `xPOSVX` にまとめられている)。 圧縮方式の格納形式版も利用可能で、それには `PO` を `PP` と置き換えればよい。コレスキー分解は `PxPOTRF`, `PxPOTRS`, `PxPOSVX` などの似た名前で ScaLAPACK ルーチンとしても実装されている。MATLAB のコレスキー分解は `chol` である。

$A - \sigma I$ がエルミートだが、定値でない この場合は Bunch–Kaufman を選択する。実数版 (複素数版) の LAPACK では、分解が `xSYTRF(xHETRF)`、分解を用いた求解が `xSYTRS(xHETRS)` として実装されている (両者は LAPACK driver routine `xSYSVX(xHESVX)` にまとめられている)。 圧縮方式の格納形式版も利用可能で、それには `SY(HE)` を `SP(HP)` と置き換えればよい。ScaLAPACK と MATLAB では利用できない。

$A - \sigma I$ がエルミートでない この場合はガウスの消去法を選択する。LAPACK では、分解が `xGETRF`、分解を用いた求解が `xGETRS` として実装されている (両者は LAPACK driver routine `xGESVX` にまとめられている)。`PxGETRF`, `PxGETRS`, `PxGESVX` などの似た名前で ScaLAPACK ルーチンとしても実装されている。MATLAB では `lu` である。

2.3.2 帯行列向け直接解法

本セクションは §2.3.1 の密行列に対する直接解法と同様の説明である：

$A - \sigma I$ がエルミート定値 この場合はコレスキー分解を選択する。LAPACK では、分解が `xPBTRF`、分解を使った求解が `xPBTRS` として実装されている (両者は LAPACK driver routine `xPBSVX` にまとめられている)。コレスキー分解は `PxPBTRF`, `PxPBTRS`, `PxPBSVX` などの似た名前で ScaLAPACK ルーチンとしても実装されている。

$A - \sigma I$ がエルミートでない この場合はガウスの消去法を選択する。LAPACK では、分解が `xGBTRF`、分解を用いた求解が `xGBTRS` として実装されている (両者は LAPACK driver routine `xGBSVX` にまとめられている)。部分軸選択付きが `PxGBTRF`, `PxGBTRS`, `PxGBSVX`、部分軸選択付きよりも速いがリスクの大きい軸選択なしが `PxDBTRF`, `PxDBTRS`, `PxDBSV` などの似た名前で ScaLAPACK ルーチンとしても実装されている。

定値でないエルミート行列の構造を活用するルーチンはありません（軸選択によって帯幅がどれだけ増加しうるかについての単純な上限がないため)。MATLAB には帯行列ルーチンがありません。$A - \sigma I$ がエルミート行列で帯幅がじゅうぶんに狭い場合、たとえば 3 重対角行列なら、§**??** で述べた直接法による固有値解法を使うべきである。

2.3.3 疎行列に対する直接解法

疎行列に対する直接解法は、密行列の場合よりもはるかに複雑なアルゴリズムが含んでいます。主な問題は、L および U 要素の em fill-in を効率的に処理する必要性に起因しています。典型的なスパースソルバは、密行列の場合の 2 ステップとは対照的に、4 つの異なるステップから構成されています。

1. 分解後にできるだけ fill が増えないように、あるいは行列がブロック三角形のような特別な構造を持つように、行と列を並べ替えるオーダリングステップ
2. 分解後の要素の非ゼロ構造を決定し、それに適したデータ構造を作成する分析ステップまたは記号分解ステップ
3. L および U 要素を計算する数値分解ステップ
4. 分解結果を使って、前進代入と交代代入を行う求解ステップ

各ステップには関連づけられた非常に多様なアルゴリズムがあります。Duff [137] ([135, Chapter 6] も参照）および Heath、Ng、Peyton [219] によるレビュー論文は、さまざまなアルゴリズムに対する優れたリファレンスとなっています。通常、ステップ 1 とステップ 2 には行列グラフのみが含まれ、したがって整数演算のみで実行されます。ステップ 3 とステップ 4 には浮動小数点演算が含まれます。通常、ステップ 3 が最も時間がかかる部分で、ステップ 4 はそれよりも約 1 桁高速です。ステップ 1 で使用されるアルゴリズムは、ステップ 3 で使用されるアルゴリズムとは完全に独立である。しかし、多くの場合、ステップ 2 のアルゴリズムはステップ 3 のアルゴリズムと密接に関連してる。最も単純な方程式、すなわち対称

かつ正定値の方程式では 4 つのステップがうまく分離されている。最も一般的な非対称な方程式の場合、ソルバーはステップ 2 とステップ 3 を組み合わせたり（SuperLU など）、ステップ 1、ステップ 2、ステップ 3 を組み合わせたり（UMFPACK など）、計算された値が消去順序の決定に関与する。

過去 10 年間、メモリ階層や並列処理などの新しいアーキテクチャの特徴を活かした多くの新しいアルゴリズムとより新しいソフトウェアが登場した。表 2.1 は、疎行列に対する直接解法のかなり包括的なリストである。ソフトウェアは、逐次計算機用ソフトウェア、SMP 用ソフトウェア、および分散並列計算機用ソフトウェアの 3 つのカテゴリに分類するのが最も便利である。

すべてのタイプの連立一次方程式に対して最適となる単一のアルゴリズムやソフトウェアはありません。あるソフトウェアは対称正定値行列を対象とし、またあるものはより一般的なケースを対象にしています。これは、表の 3 列目の「対象」に反映させています。対象が同じであっても、ソフトウェアは特定のアルゴリズムや特定の実装技法を選ぶことがあり、それはある応用には良いのですが、他のものに対してはよくありません。 2 列目の「技法」には、高水準のアルゴリズム記述を示します。left-looking, right-looking, multifrontal, その実装とパフォーマンスについては、の違いを示したレビュー論文が「列」の列 3 に反映されています。同じ範囲であっても、ソフトウェアは特定のアルゴリズムまたは実装手法を使用することを決定する場合があります。列 2 の「テクニック」では、高度なアルゴリズムの説明を示します。左向き、右向き、およびマルチフロンタルの違いとパフォーマンスへの影響のレビューについては、Heath, Ng, and Peyton [219] と Rothberg [370] の論文を示す。最良の（あるいは唯一の）ソフトウェアはパブリックドメインではないかもしれないが、市販品あるいは研究用プロトタイプは利用できることがある。 4 列目の「作成者」は、市販品を提供する会社名または研究用コードの作者名である。

固有値解析におけるシフトおよび反転スペクトル変換（SI）のコンテキストでは、固定の A について $A - sigmaI$ を分解する必要がある。したがって、$A - sigmaI$ の非ゼロ要素の構造は不変である。同じシフト点 σ に対して、同じ行列と異なる右辺に対する多くの方程式を解くことが一般的です（この場合、解法のコストは分解のコストに匹敵する）。ステップ 1 とステップ 2 にもう少し時間をかけて、ステップ 3 とステップ 4 を高速化するのが合理的である。つまり、さまざまなオーダリングを試して、記号分解に基づいた数値分解と解法のコストを推定し、最適なオーダリングが使用できる。たとえば、SVD の計算では、AA^*, A^*A, $\left[\begin{smallmatrix} 0 & A \\ A^* & 0 \end{smallmatrix}\right]$ に対するシフトと反転からいずれかを選択できます。〜 ref chap：svd の章で説明されているように、これらはすべてかなり異なる分解コストを持つている。

一部のソルバにはオーダリングスキーマが組み込まれていますが、他のソルバには組み込まれていません。また、組み込みのオーダリングスキーマが目的とする応用に最適でないこともあります。ときには、組み込みのオーダリングスキーマを外部のオーダリングスキーマに置き換えたほうが良いこともあります。多くのソルバは、ユーザーがこのような置き換えが簡単にできるよう、明確に定義されたインターフェイスを提供します。どうすればよいかを確認するため、あるいは推奨のオーダリング法を見つけるため、ソルバの文書を読んでください。

2.3.4 構造化行列に対する直接解法

§2.2.3 に記述されている構造化行列を係数にもつ連立 1 次方程式には、$O(n^3)$ ではなく、$O(n^2)$ の時間で解けるという魅力的な性質があります。

このように、連立一次方程式を $O(n^2)$ 時間で高速に解くことを可能にする共通の特徴は、これらが低

変位次元 *low displacement rank* [257] を持っているという事実です。

変位次元は次のように定義できます：与えられた行列 A と F, および行列 T に対して、演算子

$$\Delta_{A,F}(T) = A \cdot T - T \cdot F$$

と定義し、$\alpha = \operatorname{rank}(\Delta_{A,F}(T))$ と表します。α を、演算子 $\Delta_{A,F}$ に対する T の「変位次元 "displacement rank" 」と言います。α が小さい場合（通常は $O(1)$ ）、行列 T は「低変位次元」を持つと言う。$\operatorname{rank}(\Delta_{A,F}(T)) = \alpha$ から、$\Delta_{A,F}(T) = G \cdot B^T$ となる $n \times \alpha$ 次元の行列 G と B が存在する。行列 G と B は、T の生成子 em generators と呼ばれる。低変位次元の行列を変位構造 *displacement structure* を持つと言うことがある。

たとえば、コーシー行列 $C = (C_{ij}) = (1/(x_i - y_j))$ の場合

$$\Delta_{\operatorname{diag}(x),\operatorname{diag}(y)}(C) = \operatorname{diag}(x) \cdot C - C \cdot \operatorname{diag}(y) = (1, 1, \ldots, 1)^T \cdot (1, 1, \ldots, 1).$$

なので、コーシー行列は変位次元が 1 、生成子が $G = B = (1, 1, \ldots, 1)^T$ である。

演算子 $\Delta_{\operatorname{diag}(x),\operatorname{diag}(y)}$ に対する低変位次元 α を持つ行列 (この場合 1 である必要性はない) はコーシーライク Cauchy-like と呼ばれる。同様に、バンデルモンドライク Vandermonde-like 行列, テプリッツライク Toeplitz-like 行列等々を定義する。これらの高次の変位次元を持つ連立一次方程式は $O(\alpha n^2)$ の時間で求解可能である。

高速アルゴリズム Fast algorithms は、行列 F の LU 分解を生成するために変位方程式を活用します。アルゴリズムは、マトリックス自体ではなくマトリックスの生成子に機能するため、より高速に処理できます。

ブロック-テプリッツ行列および $T^T T$ タイプの行列（ここで、T はテプリッツ）は、低変位次元 [257] を持つ。

A と B が変位構造を持っている場合、$(A + \sigma B)x = b$ の形の方程式は、異なる変位演算子に対しても $A + \sigma B$ と同様な変位構造を持つ場合に限って、高速な低変位次元解法を用いて解くことができる。たとえば、A がハンケルで $B = I$ のとき、 $A + \sigma B$ はテプリッツ・ハンケル (Toeplitz-plus-Hankel) で、これも変位構造を持つ。$A + \sigma B$ に変位構造がない場合、シフト点ゼロ $\sigma = 0$ または高速な行列ベクトル積のみが使われる高速な反復解法の使用に制限されます [257]。

いくつかの方法は、行列が対称（Toeplitz）であるとか、正定値であるといった追加の制限を課します。これによって、利用可能なシフト点の選択にも追加の制限が生じることがあります。

構造化マトリックスに対して高速アルゴリズムを使用する場合、高速性はしばしば精度を犠牲にするため、特別な注意が必要です。一部の変位次元アルゴリズムは、与えられた行列自体に対してではなくジェネレータ上で動作することによって、ガウスの消去（部分軸選択、完全軸選択、または軸選択なし）をシミュレートする [193]。それでも、これらのアルゴリズムはガウス消去法と同じ数値特性を持たないことがあります。なぜなら、非常に珍しいが、きわめて条件のよい行列に対しても発生する「ジェネレータの成長 ("generator growth") [257]」の可能性があるからです。

ジェネレータの成長の問題を含む高速アルゴリズムを安定させるための多くの方法があります [257, p. 111]、そして時折不安定になるにもかかわらずこれらの方法は常に魅力的です。

高速アルゴリズムは文献で非常によく説明されていますが、信頼できるソフトウェアライブラリはありません。$O(n^2)$ 表記に隠されている定数と、それによって n がじゅうぶん大きい場合（構造にもよる

が、通常は少なくとも数百）に限って従来の n アルゴリズムより高速になることを知っておく必要がある。従来からの $O(n^3)$ アルゴリズムは、多くの場合、コードがプロセッサの最大速度近くで実行されるよう、BLAS 3 を使用し、コンピュータのメモリ階層を効率的に使用するように最適化されている (§2.2.1 を見よ)。高速アルゴリズムは通常 BLAS 2 のみが使用でき、コンピュータのキャッシュと高速メモリを効率的に使用するように高速アルゴリズムを最適化することは困難または不可能なことがある。これは、高速な $O(n^2)$ アルゴリズムと低速な $O(n^3)$ アルゴリズムが同数の浮動小数演算を実行する場合でも、これらの最適化によって「低速」アルゴリズムの方が非常に高速になりうることを意味する。

また、Vandermonde および 3 項の Vandermonde 方程式を解くための Bjorck–Pereyra タイプのアルゴリズムについても言及する必要があります [51, 230]。これらの $O(n^2)$ 解法には、顕著な数値特性があります。 Vandermonde 行列のノードの順序と右辺の符号パターンに関する付加的な制限によって、これらのシステムは完全なマシン精度で解くことができる。

2.4 反復解法の概要

H. van der Vorst

固有問題の文脈では、たとえば次の状況で線形連立一次方程式を解く必要がある場合があります。

- Jacobi–Davidson 法では、線形補正式の（近似）解が必要です。
- **??**章で説明したような inexact methods は、近似的なシフトと反転のステップに基づいており、そのため線形連立一次方程式を近似的に解く必要がある。
- 固有値のよい近似が与えられた場合、対応する左または右の固有ベクトルは、シフトされた行列を持つ連立一次方程式をから計算されます。

これらのすべての場合に、シフトされた行列 $A - \theta I$ に対する連立一次方程式を解かなければなりません。Jacobi–Davidson 法のように射影のなか投影に埋め込まれることもあります。そのような方程式を正確に解く必要のある場合、最初は直接法（スパースソルバ) が検討されます。多くの場合、同じシフト点に対する方程式を何回も解く必要があり、これはスパース LU 分解に必要な大きなコストの償却に役立ちます。方程式を正確に解かなくてもよい場合、または直接法が高くつきすぎる場合、反復法が検討されます。一般的な反復解法は、*Templates for the Solution of Linear Systems* [41] で解説されています。手短な概要説明の前に、固有値問題に関係する連立一次方程式は、通常、シフト点を含むため定値行列にならないことを読者に警告しておきます。シフト点の値が外側の固有値に近い場合、これは必ずしもすべての場合に障害になるとは限りませんが、シフト点が内部の固有値に近い場合は必ず収束の問題があります。またシフト点が固有値に非常に近い場合、たとえば左固有値または右固有値を決定したい場合、反復法ではほぼ特異な方程式を解くのが非常に難しいことを理解する必要があります。これは、左固有ベクトルや右固有ベクトルに対する逆反復の場合のように、（ほとんど）特異な方向に関心があるような状況で特に当てはまります。反復法を魅力的なものにするために、反復法には効率的な前処理が必要であるということが一般に受け入れられています。これは特に、シフトされた行列の場合です。あいにく、不定行列のための効果的な前処理の構築は扱いにくい。詳細は、 **??** 章を参照してください。

現在、最も人気のある反復法は、クリロフ部分空間法 Krylov suspace methods に属します。これらの方法では、いわゆるクリロフ部分空間で解の近似値を構成します。連立一次方程式 $Ax = b$ （前処理付きの場合、 A と b は前処理後の値）と残差ベクトルが $r_0 = b - Ax_0$ となる初期ベクトル x_0 に関連付けられた i 次元のクリロフ部分空間 $\mathcal{K}^i(A; r_0)$ は、ベクトル $\{r_0, Ar_0, A^2r_0, \ldots, A^{i-1}r_0\}$ によって張られる部分空間ととして定義される。

様々な方法はつぎのように分類されます：

(a) A が対称正定値の場合、共役勾配法 *conjugate gradient* method [226] は、２項漸化式（回帰式？）を用いて、クリロフ部分空間 $\mathcal{K}^i(A; r_0)$ 内のすべてのベクトルに対して、A-ノルムまたはエネルギーノルムと呼ばれる $(x - x_i, A(x - x_i))$ が最小になるような近似解 x_i を生成する。

(b) A が対称だが、正定値ではない場合、ランチョス法 *Lanczos*[286] と MINRES 法 *MINRES* methods [350] が考えられる。MINRES 法では $x_i \in\in \mathcal{K}^i(A; r_0)$ は残差ベクトルの 2-norm $\|b - Ax_i\|_2$ が最小化となるように、それに対してランチョス法ではクリロフ部分空間内で x_i と $b - Ax_i$ が直交するように決められる。

(c) A が非対称の場合、一般には、短い漸化式で $x_i \in K^i(A, r_0)$ で最適な近似解を決めることができない。[163] で証明されています。しかしながら、共役勾配法と MINRES 法のような短い漸化式で、$b - Ax_i \perp \mathcal{K}^i(A^T; s_0)$ (通常、$s_0 = r_0$ とする) となるような $x_i \in \mathcal{K}^i(A; r_0)$ を計算することはできる。これが双共役勾配法 *bi-conjugate gradient method* [169] である。より洗練されたものが quasi-minimal residual (QMR) [179] であり、よりスムーズな収束特性を持ち、双共役勾配法よりも頑健である。

(d) A が非対称の場合、ユークリッドノルムとして残差ノルムを最小化するように $x_i \in \mathcal{K}^i(A, r_0)$ を計算でき、これは GMRES method [389] として実現されました。この方法では、第 i 反復で i 回の内積と i 回のベクトル更新が必要となり、反復 i とともに付加的な演算コストが線形に増加することを意味します。

(e) 双共役勾配法における転置演算 A^T （転置行列ベクトル積）は、$\langle x, A^T y\rangle$ equals $\langle Ax, y\rangle$ （ここで $\langle \ldots \rangle$ は内積）という事実から A に対する演算に置き換えることができる。共役勾配法における A^T をかける機能は、残差を直交させるための双空間を維持するだけのために使われているので、これを A についての演算に置き換えてクリロフ部分空間を拡張し、よりよい近似解をみつけることが１反復あたり双共役勾配法と同一コストで仮想的にできる。これが、いわゆるハイブリッド法で、二乗共役勾配法 Conjugate gradients squared [418], Bi-CGSTAB [445], Bi-CGSTAB(ℓ) [409], TFQMR [174], hybrids of QMR [78] などがある。

(f) 定値でない場合、正規方程式 $A^TAx = A^Tb$ に共役勾配法を適用することも効果的かもしれない。これを直接的に適用すると、A^TA の条件数は A の条件数の２乗になるため、数値的な不安定性が生じるだろう。より洗練された頑健な実装が least squares QR [351] にある。

これらの方法の多くは [41] で説明され、ソフトウェアも利用可能です。ソフトウェアの入手方法に関するガイドラインは、本書のホームページ ETHOME を参照してください。一部の手法では、 [135] のような

テンプレート形式の説明になっています。この本は、またさまざまな前処理の方法についての概要も含まれています。

2.5 並列性

J. Dongarra and X. Li

この節では，本書で取り上げた反復法に関する並列性を議論する。反復法は多くの計算カーネルを共有するので，これらを解法とは独立に議論する。反復法で多くの時間を消費する基本カーネルは：

- 内積;
- ベクトル更新;
- 行列–ベクトル積, すなわち $Ap^{(i)}$ (いくつかの手法では $A^T p^{(i)}$ も);
- $(A - \sigma B)x = b$ の解法.

である。

内積 2 つのベクトルの内積計算は、各プロセッサが各ベクトルの対応するセグメントの内積（局所的な内積、LIPs）を計算することで容易に並列化できる。分散メモリ方式の計算機では，グローバルな内積に結合させるため、 LIPs を他のプロセッサに送らなければならない。これは、すべてのプロセッサが LIPs の総和を計算するために all-to-all 送信をするか，あるいはひとつのプロセッサが累積をとってその結果を放送することになる。明らかに，このステップでは通信が必要になる。

共有メモリ方式の計算機では，LIPs の足し込みはすべてのプロセッサがそれぞれの局所的な結果を順番に全体の結果に加算するクリティカルセクションとして実装するか、あるいは逐次コードの一部としてひとつのプロセッサが総和を実行する。

ベクトル更新. ベクトル更新の並列化は自明であり、それぞれのプロセッサが自分のセグメントを更新する。

行列–ベクトル積 行列–ベクトル積は，行列の行を短冊状に分割してベクトルセグメントに対応させることで、共有メモリ方式の計算機では容易に並列化できる。各プロセッサは短冊ひとつ分の行列–ベクトル積を計算する。分散メモリ方式の計算機では，各プロセッサがメモリ上にベクトルの 1 セグメントだけしか持っていないことが問題かもしれない。行列の帯幅に応じて他のベクトル要素を通信する必要が生じて、通信のボトルネックとなるかもしれない。しかしながら，多くの疎行列問題は近くの節点とだけ接続するネットワークから生じている。たとえば有限差分法や有限要素法の問題から生じる行列は，典型的な局所的接続のみを含む。すなわち，変数 i と j が物理的に近いときに限り行列要素 $a_{i,j}$ が非ゼロ要素となる。そのような場合，ネットワークまたは格子を適切なブロックに小分けして，それらをプロセッサに分散させるが自然だろう。$Ap^{(i)}$ を計算する際，各プロセッサは隣接したブロック内のいくつかの節点での $p^{(i)}$ の

値を必要とする。これら隣接したブロックへの接続の数が内部の節点数に比べて小さければ，通信時間は計算時間に重ね合わされる。

最近、 em graph partitioning は、非局所接続をもつ一般的な問題に対処するための強力なツールとして使われています。$y\ leftarrow Ax$ と（対称）行列 A に対応する無向グラフ G を考えます。頂点 i は、すべての j に対して x_i, y_i、および非ゼロの $a_{i,\ j}$ を格納すると仮定します。 頂点 i は $y_i = \ sum_j a_{i,\ j} x_j$ の計算ジョブを表します。y_i を計算する操作の数を頂点 i の重みとして割り当て、辺（i、j）に重み 1 を割り当てます。これは、頂点 i と j が異なるプロセッサに割り当てられた場合に、x_j を頂点 i に送信するコストを表します。優れたグラフ分割ヒューリスティックは、以下の条件で G を P プロセッサに対応させ、頂点を P 個のサブセットに分割します。

- それぞれの部分集合で頂点の重みの和がほぼ等しい,
- 部分集合を横切る辺の総数が最小化されている.

これは良好な負荷バランスを実現し、通信を最小限に抑えます。よいグラフパーティションソフトウェアとして、Chaco [223] と METIS [259] (並列版である ParMETIS) があります。

分割後、通信を削減するためのさらなる最適化を実行できます。たとえば、頂点のサブセットに別のサブセットへの複数のエッジが含まれる場合、ベクトルの対応する要素を 1 つのメッセージにパックして、各プロセッサは別のプロセッサにメッセージをひとつだけしか送信できなようにします。分散メモリシステムへの実装の詳細については、De Sturler and van der Vorst [111, 112] と Pommerell [369] を参照してください。

分散メモリマシン用の高品質並列アルゴリズムは、Aztec [236] や PETSc [38] などのソフトウェアパッケージに実装されています。それらのソフトウェアは、本書のホームページ ETHOME から入手できます。

ソルバー. 上記の 3 つのカーネルに加えて、シフトと反転を使用する反復法では、線形システムの直接解法を必要とします。行列分解と三角解法は、特に超並列マシンでは、はるかに複雑な並列アルゴリズムになります。この分野では多くの研究活動が行われており、密行列および帯行列に対する最新の並列アルゴリズムの多くは ScaLAPACK に実装され (§ 2.3 を参照)、疎行列に対する最新の並列アルゴリズムが実装されたソフトウェアは表 2.1 にまとめられています。

Table 2.1: 直接法を用いた疎行列解法ソフトウェア.

Code	技法	対象	作成者	
逐次計算環境				
MA27	Multifrontal	Sym	HSL	[140]
MA41	Multifrontal	Sym-pat	HSL	[6]
MA42	Frontal	Unsym	HSL	[144]
MA47	Multifrontal	Sym	HSL	[141]
MA48	Right-looking	Unsym	HSL	[142]
SPARSE	Right-looking	Unsym	Kundert	[281]
SPARSPAK	Left-looking	SPD	George	[191]
SPOOLES	Left-looking	Sym and Sym-pat	Ashcraft	[21]
SuperLLT	Left-looking	SPD	Ng	[339]
SuperLU	Left-looking	Unsym	Li	[126]
UMFPACK	Multifrontal	Unsym	Davis	[103]
SMP マシン				
Cholesky	Left-looking	SPD	Rothberg	[405]
DMF	Multifrontal	Sym	Lucas	[308]
MA41	Multifrontal	Sym-pat	HSL	[7]
PanelLLT	Left-looking	SPD	Ng	[211]
PARASPAR	Right-looking	Unsym	Zlatev	[471]
PARDISO	Left-right looking	Sym-pat	Schenk	[395]
SPOOLES	Left-looking	Sym and Sym-pat	Ashcraft	[21]
SuperLU_MT	Left-looking	Unsym	Li	[127]
分散並列環境				
CAPSS	Multifrontal	SPD	Raghavan	[220]
DMF	Multifrontal	Sym	Lucas	[308]
MUMPS	Multifrontal	Sym and Sym-pat	Amestoy	[8]
PaStiX	Left-right looking*	SPD	CEA	[224]
PSPASES	Multifrontal	SPD	Gupta	[210]
SPOOLES	Left-looking	Sym and Sym-pat	Ashcraft	[21]
SuperLU_DIST	Right-looking	Unsym	Li	[306]
S+	Right-looking†	Unsym	Yang	[182]

* 論文のタイトルとは異なる

† LU 分解における任意の fill-in に静的に対応するため QR 法の格納形式を使用

表中の略記は以下のとおり: SPD = 対称かつ正定値; Sym = 対称、おそらく定値でない; Sym-pat = 配置は対称だが、値は非対称; Unsym = 非対称; HSL = Harwell Subroutine Library: `http://www.cse.clrc.ac.uk/Activity/HSL`