クレジット:
Mathematics and Informatics Center 計算機実験I 2020 藤堂眞治

本講義資料内に掲載している外部へのリンク先の著作物の利用に関しては、リンク先のそれぞれの権利者の定めるところに従ってください。

計算機実験 I (第 4 回)

藤堂眞治

2020/05/20

物理の問題にあらわれる行列演算

- 連立一次方程式・逆行列
  - 偏微分方程式の境界値問題
  - 非線形連立方程式に対するニュートン法
- 対角化・特異値分解
  - 固有値問題・行列関数
  - 最小二乗近似
- 計算機は大規模行列演算が得意
  - 直接法: $\sim 10^4$ 次元
  - 疎行列に対する反復解法: $\sim 10^9$ 次元
- 行列演算についてはライブラリがよく整備されている
  - それぞれの原理とその特徴を理解して正しく使うことが重要
  - 適切なライブラリを使うことで数十倍あるいはそれ以上速くなることも

疑似乱数とは

- 計算機でプログラムに従って生成する乱数 (のようなもの)
- 乱数は何に役立つか？
  - 等式のチェック、例外の発見
  - 初期値にランダムネスを入れることで最悪の場合を避ける
  - サンプリングを使ったシミュレーション (→計算機実験 II)
- 乱数を使う場合の注意
  - 計算式に従って生成するため周期は有限であり、必ず何らかの相関がある
  - 初期化 (種の設定) を正しく行う
  - 実際にそれらしい乱数が生成されているか目で見て確認する
- 代表的な乱数発生器のひとつ: メルセンヌ・ツイスター
  - 周期 $2^{19937}-1$、高速、日本製!
  - ヘッダファイル: mersenne_twister.h
  - サンプルプログラム: random.c

複素数

- complex.h を include することで、double complex 型 (float complex 型) が使えるようになる
- 複素数の値は、CMPLX (あるいは CMPLXF) で設定する
- cexp, csin, clog 等の初等関数が使える
- 実部は creal, 虚部は cimag で取り出せる
- プログラム例: complex.c

```
#include <complex.h>
...
double complex x, y;
x = CMPLX(0, 1); /* 虚数単位 */
y = cexp(x * M_PI);
printf("i␣=␣(%lf,%lf)\n", creal(x), cimag(x));
printf("e^{i*pi}␣=␣(%lf,%lf)\n", creal(y), cimag(y));
```

C 言語におけるポインタ

- 変数はメモリ上のどこかに格納されている
  - 変数の値: メモリに格納されている数値
  - アドレス: 変数の値が格納されているメモリ上の番地
- ポインタ変数
  - 値としてアドレスを格納する変数のこと
  - ポインタ変数の値 (アドレス) とポインタ変数のアドレスは異なるものであることに注意
- ポインタ変数の宣言、代入、実体へのアクセス
  - 整数型ポインタ変数の宣言: `int *p;`
  - 整数型変数の宣言: `int q;`
  - 変数 `q` のアドレスをポインタ変数 `p` に代入: `p = &q;`
  - ポインタ変数 `p` に格納されているアドレスに格納されている値の参照 (間接参照): `*p`

ポインタの例 (1)

- 例 2.5.1 (ハンドブック 2.5 節)

```
#include <stdio.h>
int main() {
  int *p;
  int q;
  q = 200;
  p = &q;
  printf("q␣is␣%d␣and␣*p␣is␣%d.\n", q, *p);
  return 0;
}
```

  - q のアドレスを p に代入
  - q と*p の値を出力 → 両者とも 200

ポインタの例 (2)

- 例 2.5.2 (ハンドブック 2.5 節)

```
#include <stdio.h>
int main() {
  int *p;
  int q;
  p = &q;
  *p = 300;
  printf("q␣is␣%d␣and␣*p␣is␣%d.\n", q, *p);
  return 0;
}
```

  - q のアドレスを p に代入
  - *p に 300 を代入 (ここで q=300; と書いても等価)
  - q と*p の値を出力 → 両者とも 300

関数呼び出し (ポインタ渡し)

- 例 2.6.4 (ハンドブック 2.6 節)

```
#include <stdio.h>
void division(int divident, int divisor, int *quotient,
              int *residual) {
  *quotient = divident / divisor;
  *residual = divident % divisor;
}
int main() {
  int josuu = 3;
  int hi_josuu = 13;
  int shou, amari;
  division(hi_josuu, josuu, &shou, &amari);
  printf("%d␣/␣%d␣=␣%d␣...␣%d\n", hi_josuu, josuu,
         shou, amari);
}
```

間違った例 (値渡し)

- 例 2.6.5 (ハンドブック 2.6 節)

```
#include <stdio.h>
void division(int divident, int divisor, int quotient,
  int residual) {
  quotient = divident / divisor;
  residual = divident % divisor;
}
int main() {
  int josuu = 3;
  int hi_josuu = 13;
  int shou, amari;
  division(hi_josuu, josuu, shou, amari);
  printf("%d␣/␣%d␣=␣%d␣...␣%d\n", hi_josuu, josuu,
         shou, amari);
}
```

  - 誤った答えが出力される。なぜ？

一次元配列

- (静的) 一次元配列 (ハンドブック 2.3.1 節)

```
double v[10];
v[0] = 1.0;
v[1] = 2.0;
...
```

要素数はコンパイル時にすでに決まっている定数でなければならない

- (動的) 一次元配列 (ハンドブック 2.11 節)

```
double *v; /* ポインタ */
v = (double*)malloc((size_t)(10 * sizeof(double));
...
free(v); /* 確保した領域を開放 */
```

実行時に要素数を指定可能

ポインタと一次元配列

- 一次元配列を表す変数は、(実は) 最初の要素を指すポインタ (ハンドブック 2.5.3 節)
  - `v` と `&v[0]` は等価
  - `(v+2)` と `&v[2]` は等価
  - `*v` と `v[0]` は等価
  - `*(v+2)` と `v[2]` は等価
  - `(v+2)[3]` は?
- C 言語では配列の添字は 0 から始まることに注意
- `double v[10];` と宣言した場合、`v[0]` 〜 `v[9]` の 10 個の要素を持つ配列が作られる。`v[10]` は存在しない。値を代入したり参照しようとするとエラーとなる
- ポインタのテストプログラム: pointer.c

二次元配列

- C 言語では、二次元配列は一次元配列の先頭をさす (ポインタ) の配列として表される (と理解しておけば良い)
- a[i] は、要素 a[i][0] を指すポインタ
  - a と &a[0] は等価 (&a[0][0] ではない)
  - a[0] と &a[0][0] は等価
  - a[2] と &a[2][0] は等価
  - (a+2) と &a[2] は等価
  - (*(a+2))[3] と *(*(a+2)+3) と a[2][3] は等価
  - *(a+2)[3] と *((a+2)[3]) と *(a[5]) と a[5][0] は等価
  - [] は*よりも優先度が高い
- ポインタのテストプログラム: pointer-matrix.c

動的二次元配列の確保

- 各行を表す配列とそれぞれの先頭アドレスを保持する配列の二種類が必要

```
double **a;
n = 10;
a = (double**)malloc((size_t)(n * sizeof(double*)));
for (int i = 0; i < n; ++i)
  a[i] = (double*)malloc((size_t)(n * sizeof(double)));
```

- メモリ上では各行の要素が連続して保存される (`m[i][j]` の次に `m[i][j+1]`、“row-major” と呼ぶ)
- 各行を保持する配列が、メモリ上で連続に確保される保証はない
- 行列用のライブラリ (BLAS, LAPACK 等) を使うときに問題となる (⇒後述の `cmatrix.h` ライブラリを利用する)

BLAS ライブラリ

- 行列・行列積、行列・ベクトル積などを高速に行う最適化された関数群
- 行列・行列積を計算するサブルーチン dgemm
  http://www.netlib.org/lapack/explore-html/d7/d2b/dgemm_8f.html
  - $C = \alpha A \times B + \beta C$ を計算
  - BLAS は Fortran 言語で書かれている
- 例: multiply.c, multiply_dgemm.c

LAPACK (Linear Algebra PACKage)

- 線形計算のための高品質な数値計算ライブラリ
  - http://www.netlib.org/lapack
  - 線形方程式、固有値問題、特異値問題、線形最小二乗問題など
  - (FFT 高速フーリエ変換は入っていない)
  - LAPACK 自体も Fortran 言語で書かれている
- ほぼ全ての PC、ワークステーション、スーパーコンピュータで利用可 (インストール済)
- Netlib でソースが公開されているリファレンス実装は遅いが、それぞれのベンダー (Intel、Fujitsu、etc) による最適化された LAPACK が用意されている場合が多い (MKL、SSL2、etc)
- LAPACK を使うことにより、高速で信頼性が高く、ポータブルなコードを書くことが可能になる

C から BLAS/LAPACK を呼び出す際の注意事項

- 関数名はすべて小文字、最後に _ (下線) を付ける
- スカラー、ベクトル、行列は全て「ポインタ渡し」とする
- ベクトルや行列は最初の要素へのポインタを渡す
- 行列の要素は (0,0) $\to$ (1,0) $\to$ (2,0) $\to \cdots \to (m-1,0) \to$ (0,1) $\to$ (1,1) $\to \cdots \to (m-1, n-1)$ の順で連続して並んでいなければならない (column-major)
  - C 言語の二次元配列では `a[i][j]` の次には `a[i][j+1]` が入っている (row-major)
  - 行列が転置されて解釈されてしまう!
- コンパイル時には`-llapack -lblas` オプションを指定し、LAPACK ライブラリと BLAS ライブラリをリンクする (ハンドブック 2.1.6 節)

cmatrix.h ライブラリ

- Column-major 形式の二次元配列の確保 (alloc_dmatrix)、開放 (free_dmatrix)、出力 (print_dmatrix)、読み込み (read_dmatrix) を行うためのユーティリティ関数、(i,j) 成分にアクセスするためのマクロ (mat_elem) 他を準備
- ソースコード: cmatrix.h
- 使用例

```
#include "cmatrix.h"
...
double **mat;
mat = alloc_dmatrix(m, n);
mat_elem(mat, 1, 3) = 5.0;
...
free_dmatrix(mat);
```

- サンプルコード: matrix_example.c

alloc_dmatrix での動的二次元配列の確保

- 長さ $m \times n$ の一次元配列を用意し、各列 (それぞれ m 要素) の先頭アドレスを長さ n のポインター配列に格納する

```
double **a;
m = 10;
n = 10;
a = (double**)malloc((size_t)(n * sizeof(double*));
a[0] = (double*)malloc((size_t)(m*n * sizeof(double));
for (int i = 1; i < n; ++i)
  a[i] = a[i-1] + m;
```

- 行列の (i,j) 成分を a[j][i] に格納することにする (column-major)

要素アクセス・先頭アドレス

- 行列の (i,j) 成分を a[j][i] に格納することにする (column-major)
  - cmatrix.h ではマクロ (mat_elem) を準備

```
#define mat_elem(mat, i, j) (mat)[j][i]
```

  - このマクロを使うと、例えば (i,j) 成分への代入は以下のように書ける

```
mat_elem(a, i, j) = 1;
```

- LAPACK にベクトルや行列の最初の要素へのポインタを渡す
  - ベクトルの最初の要素 (0) へのポインタ: &v[0]
  - 行列の最初の要素 (0,0) へのポインタ: &a[0][0]
  - cmatrix.h にマクロ (vec_ptr、mat_ptr) が準備されているのでそれぞれ、vec_ptr(v)、mat_ptr(a) と書ける

行列のノルム

- 行列のノルム: $\|A\|$
  - 正定値性: $\|A\| \geq 0$. $\|A\| = 0$ となるのは $A = O$ のときのみ
  - 斉次性: $\|\alpha A\| = |\alpha| \|A\|$
  - 劣加法性: $\|A + B\| \leq \|A\| + \|B\|$
- さまざまなノルム
  - 1-ノルム: $\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|$
  - ∞-ノルム: $\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}|$
  - スペクトルノルム: $A^* A$ の最大固有値の平方根 (固有値分解が必要)
  - フロベニウスノルム: $\|A\|_{\mathrm{F}} = \left[\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right]^{1/2}$
  - 最大ノルム: $\|A\|_{\max} = \max\{|a_{ij}|\}$
    (劣乗法性 $\|AB\| \leq \|A\| \|B\|$ を満たさない)
- 例題: 適当な $m \times n$ 行列についてノルムを計算するプログラムを作成せよ

LAPACK による行列のノルムの計算

- 倍精度 (単精度) 実行列の場合 dlange (flange) 関数、倍精度 (単精度) 複素行列の場合 zlange (clange) 関数を使う
  http://www.netlib.org/lapack/explore-html/dc/d09/dlange_8f.html
- C 言語から呼び出すための関数宣言: dlange.h

```
double dlange_(char *NORM, int *M, int *N, double *A, int *LDA,
     double *WORK);
```

- dlange 関数の利用例: dlange.c

```
norm = 'F';
m = 10;
n = 10;
a = alloc_dmatrix(m, n);
...
res = dlange_(&norm, &m, &n, mat_ptr(a), &m, vec_ptr(work));
```

LAPACK による連立一次方程式の求解

- 連立一次方程式: $Ax = b$
- C 言語から呼び出すための関数宣言: dgesv.h

```
void dgesv_(int *N, int *NRHS, double *A, int *LDA, int *IPIV,
     double *B, int *LDB, int *INFO);
```

- dgesv の利用例: dgesv.c

```
n = 3;
a = alloc_dmatrix(n, n);
b = alloc_dvector(n);
...
dgesv_(&n, &nrhs, mat_ptr(a), &n, vec_ptr(ipiv), vec_ptr(b), &n,
      &info);
```

配列 b は解で上書きされる

- dgesv の中ではまず行列を LU 分解し、その結果を利用して解を求めている