C プログラミング

— ポインタを用いた行列の取り扱い —

早稲田大学

本日の目標

- 配列を用いた行列・ベクトルの表現
- 行列の演算
- 配列とポインタ
- メモリの動的確保
- 行列ベクトル積への応用

ベクトル，行列の表現法

C 言語で，ベクトルや行列を表現するには，１次元配列，２次元配列をそれぞれ使う．

- ベクトル：（１次元）配列を利用する．

```
double x[N];    /*x[0]〜x[N-1] の N 個の要素を確保*/
```

$$x[N] = \big(x[0],\quad x[1],\quad \cdots\quad ,x[N-1]\big)$$

- 行列：２次元配列を利用する．

```
double a[N][N]; /*a[0][0]〜a[N-1][N-1] の N^2 個の要素を確保*/
```

$$a[N][N] = \begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][N-1] \\ a[1][0] & a[1][1] & \cdots & a[1][N-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[N-1][0] & a[N-1][1] & \cdots & a[N-1][N-1] \end{pmatrix}$$

行列・ベクトルの演算

行列とベクトルの積

N 次正方行列 A と N 次元ベクトル $\vec{x}$ の積 $\vec{y} = A\vec{x}$ を計算するプログラム.

- $\vec{y} = A\vec{x}$ を具体的に成分で表せば,

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,N-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,N-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix}$$

- ベクトル $\vec{y}$ の各成分 $y_i (i = 0, \ldots, N-1)$ は

$$y_i = a_{i,0}x_0 + a_{i,1}x_1 + \ldots + a_{i,N-1}x_{N-1} = \sum_{k=0}^{N-1} a_{i,k}x_k$$

と表せる.

行列・ベクトルの演算

- $y[i]$ の値を求めるには $a_{i,k}x_k$ の総和を求めればよい.

```
y[i]=0.0;
for(k=0; k<N; k++){
    y[i] += a[i][k]*x[k];
}
```

- $\vec{y} = A\vec{x}$ を求めたければ,

```
for(i=0; i<N; i++){
  y[i]=0.0;
  for(k=0; k<N; k++){
      y[i] += a[i][k]*x[k];
  }
}
```

- はじめに $y[i] = 0.0$ と初期化をすることを忘れないように.

行列・行列の演算

行列と行列の積

$N \times L$ 行列 A と $L \times M$ 行列 B の積 $Y = AB$ を計算するプログラムを作ろう.

- $Y = AB$ を具体的に成分で表せば,

$$\begin{pmatrix} y_{0,0} & \cdots & y_{0,M-1} \\ \vdots & \ddots & \vdots \\ y_{N-1,0} & \cdots & y_{N-1,M-1} \end{pmatrix} = \begin{pmatrix} a_{0,0} & \cdots & a_{0,L-1} \\ \vdots & \ddots & \vdots \\ a_{N-1,0} & \cdots & a_{N-1,L-1} \end{pmatrix} \begin{pmatrix} b_{0,0} & \cdots & b_{0,M-1} \\ \vdots & \ddots & \vdots \\ b_{L-1,0} & \cdots & b_{L-1,M-1} \end{pmatrix}$$

- 行列 Y の各成分 $y_{i,j} (i = 0, \ldots, N-1; j = 0, \ldots, M-1)$ は

$$y_{i,j} = a_{i,0}b_{0,j} + a_{i,1}b_{1,j} + \ldots + a_{i,L-1}b_{L-1,j} = \sum_{k=0}^{L-1} a_{i,k}b_{k,j}$$

と表せる.

行列・行列の演算

- $y[i][j]$ の値を求めるには $a_{i,k}b_{k,j}$ の総和を求めればよい.

```
y[i][j]=0.0;
for(k=0; k<L; k++){
  y[i][j] += a[i][k]*b[k][j];
}
```

- 行列積 AB を求めるためには,

```
for(i=0; i<N; i++){
  for(j=0; j<M; j++){
    y[i][j]=0.0;
    for(k=0; k<L; k++){
      y[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

- 同様に初期化を忘れないように.

例題 1

例題 1

次の行列 A とベクトル $\vec{x}$ の積 $\vec{y} = A\vec{x}$ を計算せよ.

$$A = \begin{pmatrix} 3.0 & 2.0 & 5.0 \\ 1.0 & 4.0 & 3.0 \\ 0.0 & 1.0 & 6.0 \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} -3.0 \\ 2.0 \\ -1.0 \end{pmatrix}$$

例題 1 の答え

```
#include <stdio.h>
#define N 3
int main(void) {
  int i, j;
  double A[N][N] = {{3.0, 2.0,5.0},{1.0, 4.0, 3.0},{0.0, 1.0, 6.0}};
  double x[N]= {-3.0, 2.0, -1.0};
  double y[N];  /* 計算結果を代入するための行列を用意 */
  printf("A = \n"); /* 行列 A の表示 */
  for (i = 0; i < N; i++) { /* i 行 */
    for (j = 0; j < N; j++) { /* j 列 */
      printf("%6.1f\t", A[i][j]); /* 各列はタブ区切りで表示 */
    }
    printf("\n"); /* 次の行を表示するための改行*/
  }
  printf("\n"); /* 区切りのための改行 */
  printf("x=\n");
  for (i = 0; i < N; j++) printf("%6.1f\n", x[i]); /* 各列はタブ区切りで表示 */
  printf("\n");
  /* y = Ax の計算 */
  for (i = 0; i < N; i++) {
    y[i] = 0; /* 初期化 */
    for (j = 0; j < N; j++) {
      y[i] += A[i][j] * x[j];
    }
  }
  printf("\n");
  printf("y = \n"); /* ベクトル y(=Ax) の表示 */
  for (i = 0; i < N; j++) printf("%6.1f\n", y[i]); /* 各列はタブ区切りで表示 */
  printf("\n"); /* 区切りのための改行 */
  return 0;
}
```

例題 1 について

例題 1

次の行列 A とベクトル $\vec{x}$ の積 $\vec{y} = A\vec{x}$ を計算せよ.

$$A = \begin{pmatrix} 3.0 & 2.0 & 5.0 \\ 1.0 & 4.0 & 3.0 \\ 0.0 & 1.0 & 6.0 \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} -3.0 \\ 2.0 \\ -1.0 \end{pmatrix}$$

- 配列を使って行列，ベクトルを用いると，他の次元にするたびに書き直さなければいけない.
  $\Rightarrow$ メモリを動的に確保
  $\Rightarrow$malloc 関数を利用しベクトルや行列をポインタを使って表現する.
- 配列とポインタの関係について学習しよう.

配列とポインタ

n 次元ベクトル $\vec{x}$ をプログラム上で表現する際は，配列を利用する．
double x[N]; と宣言することで double 型で N 個分の記憶領域がメモリ上に連続して確保される：

アドレス		0xbfff660	0xbfff668	0xbfff676	$\cdots$		
メモリ内容	$\cdots$	x[0]	x[1]	x[2]	$\cdots$	x[N-1]	$\cdots$

- 各要素のアドレスはアドレス演算子 (&) を用いて，

  `&x[0]`, `&x[1]`, `&x[2]`, $\cdots$, `&x[N-1]`

  により参照できる
- （double 型は 8byte(64bit) のため 8 ずつ確保される）

配列とポインタ

- 要素 $x[i]$ のアドレスは $\&x[i]$ により参照できる.

アドレス		0xbfff660 =&x[0]	0xbfff668 =&x[1]	0xbfff676 =&x[2]	$\cdots$	&x[N-1]	
メモリ内容	$\cdots$	x[0]	x[1]	x[2]	$\cdots$	x[N-1]	$\cdots$

- 一方，配列名 x は先頭要素へのポインタとして利用できる
- ポインタの演算を用いることにより，各要素のアドレスは $x+i$ $(i=0,1,2,\cdots,N-1)$ でも参照できる.

アドレス		0xbfff660 =x	0xbfff668 =x+1	0xbfff676 =x+2	$\cdots$	x+(N-1)	
メモリ内容	$\cdots$	x[0]	x[1]	x[2]	$\cdots$	x[N-1]	$\cdots$

アドレス

$$x+i \quad = \quad \&x[i] \qquad (i=0,1,\cdots,N-1)$$

配列とポインタ

- ポインタに対して間接参照演算子 (*) を適用することにより，ポインタの指すアドレスに格納された値を参照できる.
- ポインタ $x+i$ に対しては，$*(x+i)$ とすることで各要素の値を参照できる.

アドレスの参照		x ↓	x+1 ↓	x+2 ↓		x+(N-1) ↓	
メモリ内容	···	x[0]	x[1]	x[2]	···	x[N-1]	···
値の参照		↑ *x	↑ *(x+1)	↑ *(x+2)		↑ *(x+(N-1))	

値の参照

$$x[i] \quad \leftrightarrow \quad *(x+i)(i=0,1,2,\cdots,N-1)$$

配列を引数で渡す

- 配列を引数で渡す時は，受け側ではポインタ型を用いる．

```
/*要素数 size の int 型配列の中身をすべて 0 にする関数*/
void setZero( int *x, int size){
    int i;
    for(i=0; i<size; i++)
        x[i]=0;
}
int main(void){
    int x[10];
    setZero(x,10);          /*ここで&x としてはいけない． */
```

- 関数 setZero の中の $x[i] = 0$ は $*(x + i) = 0$ としてもよい．

行列とポインタ

- 行列をプログラム上で表現する場合は，2 次元配列を利用する．
- 例えば，N 次正方行列 A に対しては double a[N][N] と宣言することで，double 型で $N \times N$ 個分の記憶領域がメモリ上に確保される：

行列 A：

A[0][0]	A[0][1]	A[0][2]	$\cdots$	A[0][N-1]
A[1][0]	A[1][1]	A[1][2]	$\cdots$	A[1][N-1]
A[2][0]	A[2][1]	A[2][2]	$\cdots$	A[2][N-1]
$\vdots$	$\vdots$	$\vdots$		$\vdots$
A[N-1][0]	A[N-1][1]	A[N-1][2]	$\cdots$	A[N-1][N-1]

行列とポインタ

行列を配列で宣言：

double A[N][N];

このとき，A の $N \times N$ 個の要素の領域は，
右のようにメモリ上に連続して確保される．

A は

$$A[0], A[1], \cdots, A[N-1]$$

から構成される配列の先頭を指すポインタ
（$A = \&A[0]$）
各要素 $A[i](i = 0, 1, \cdots, N-1)$ は第 i 行の
先頭要素を指すポインタである．

ポインタ	メモリ内容
A[0]	A[0][0]
	A[0][1]
	⋮
	A[0][N-1]
A[1]	A[1][0]
	A[1][1]
	⋮
	A[1][N-1]
	⋮
A[N-1]	A[N-1][0]
	⋮
	A[N-1][N-1]
	⋮

行列とポインタ

double A[N][N];

行列 A の第 i 行 j 列の要素（$A[i][j]$）を，ポインタと間接参照演算子を用いて参照する場合は，

$$*(A[i]+j)$$

となる．

$$A[i][j] \Leftrightarrow *(A[i]+j)$$

	メモリ内容
*(A[0]) →	A[0][0]
*(A[0]+1) →	A[0][1]
	⋮
*(A[0]+N-1) →	A[0][N-1]
	⋮
*(A[i]+0) →	A[i][0]
*(A[i]+1) →	A[i][1]
	⋮
*(A[i]+j) →	A[i][j]
	⋮

行列とポインタ

double A[N][N];

に対してポインタ変数 ai を導入する：

double $*$ ai;

行列 A の先頭の要素 $(A[0][0])$ へのポインタを示す $A[0]$ を利用して，

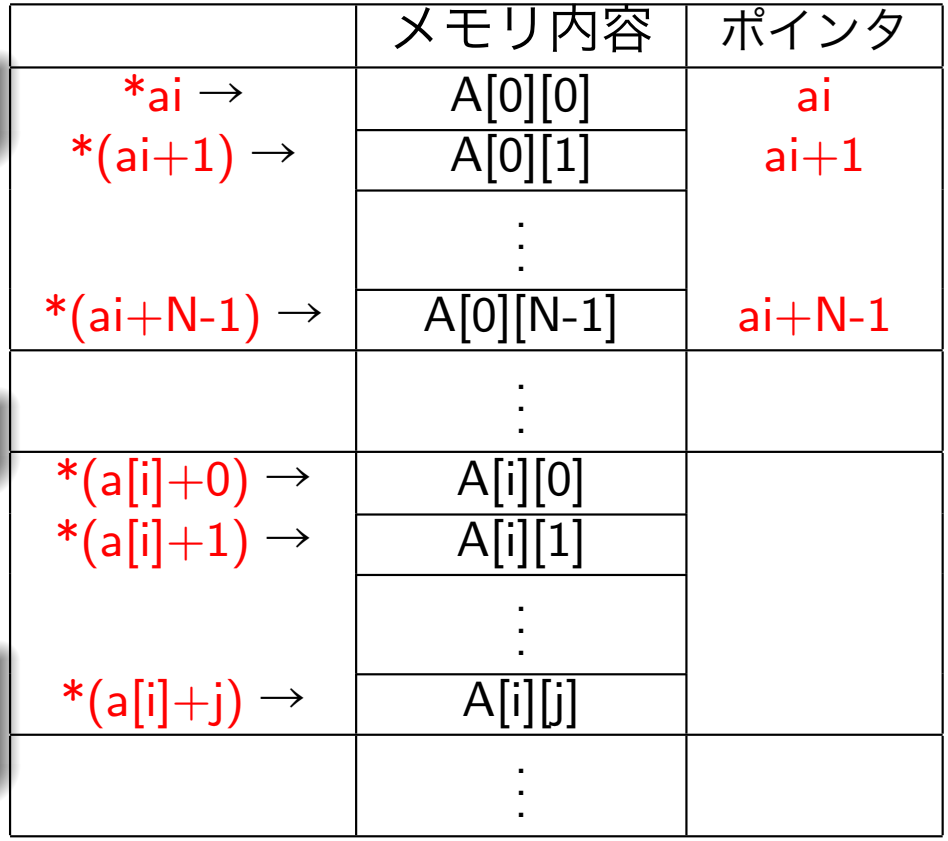
$ai = A[0];$

とすることで，A の第 0 行の各要素が

$*ai, *(ai+1), \cdots, *(ai+N-1)$

で参照できる．このとき，

$ai[j] \Leftrightarrow *(ai+j)$



行列とポインタ

次の行に処理を移すときは,

$$ai+ = N;$$

によりポインタ ai を更新すれば, ai は常に,

「第 i 行の先頭要素 $A[i][0]$ へのポインタ」

として機能し，第 i 行の各要素が

$$ai[0], ai[1], \cdots, ai[N-1]$$

で参照できる,

	メモリ内容	ポインタ
	A[0][0]	ai
	A[0][1]	ai+1
	⋮	
	A[0][N-1]	ai+N-1
ai[0] →	A[1][0]	ai
ai[1] →	A[1][1]	ai+1
	⋮	
ai[N-1] →	A[1][N-1]	ai+N-1
ai[0] →	A[2][0]	ai
	⋮	

行列とポインタ

次の行に処理を移すときは,

$$ai+ = N;$$

によりポインタ ai を更新すれば, ai は常に,

「第 i 行の先頭要素 $A[i][0]$ へのポインタ」

として機能し，第 i 行の各要素が

$$ai[0], ai[1], \cdots, ai[N-1]$$

で参照できる,

	メモリ内容	ポインタ
	A[0][0]	ai
	A[0][1]	ai+1
	⋮	
	A[0][N-1]	ai+N-1
ai[0] →	A[1][0]	ai
ai[1] →	A[1][1]	ai+1
	⋮	
ai[N-1] →	A[1][N-1]	ai+N-1
ai[0] →	A[2][0]	ai
	⋮	

行列とポインタ

次の行に処理を移すときは,

$$ai+ = N;$$

によりポインタ ai を更新すれば, ai は常に,

「第 i 行の先頭要素 $A[i][0]$ へのポインタ」

として機能し，第 i 行の各要素が

$$ai[0], ai[1], \cdots, ai[N-1]$$

で参照できる,

	メモリ内容	ポインタ
	A[0][0]	ai
	A[0][1]	ai+1
	⋮	
	A[0][N-1]	ai+N-1
ai[0] →	A[1][0]	ai
ai[1] →	A[1][1]	ai+1
	⋮	
ai[N-1] →	A[1][N-1]	ai+N-1
ai[0] →	A[2][0]	ai
	⋮	

例題 2

例題 2

次の行列 A とベクトル $\vec{x}$ の積 $\vec{y} = A\vec{x}$ をポインタの表現を用いたプログラムに書き直し計算せよ.

$$A = \begin{pmatrix} 3.0 & 2.0 & 5.0 \\ 1.0 & 4.0 & 3.0 \\ 0.0 & 1.0 & 6.0 \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} -3.0 \\ 2.0 \\ -1.0 \end{pmatrix}$$

- 下のように，２次元配列による表現をポインタによる表現に変更せよ.

【２次元配列による表現】

```
for(i=0;i<N;i++){
    y[i] = 0.0;
    for(j=0;j<N;j++){
        y[i] += A[i][j]*x[j];
    }
}
```

【ポインタによる表現】

```
ai=A[0];
for(i=0;i<N;i++){

        【この箇所は各自考えてみよ】
}
```

例題 2 のヒント

- 計算するプログラムを以上で説明した表現を用いて書き直してみよう.

```
double x[N];
double A[N][N];
double *ai;
```

- ベクトル x の各要素

```
x[i]    (i=0,1,2,⋯,N-1)
```

- 行列 A の各要素：ai が第 i 行の先頭要素へのポインタであるとして,

```
ai[0], ai[1], ⋯,ai[j],⋯,ai[N-1]
```

- 次の行（第 $i+1$ 行）へ移るときには,

```
ai +=N;
```

解答例

```
#include <stdio.h>

#define N 3

void PrintVector(double *y, int n){
int i;
for(i=0;i<n;i++){
printf("%6.1f\n",*(y+i));
}
}

int main(void) {
  int i, j;
  /* 各行の先頭へのポインタを表すポインタを用意する*/
  double *ai;
  double A[N][N] =
   {{3.0, 2.0,5.0},{1.0, 4.0, 3.0},{0.0, 1.0, 6.0}};
  double x[N]= {-3.0, 2.0, -1.0};
  double y[N];  /* 計算結果を代入するための行列を用意 */
  printf("A = \n"); /* 行列 A の表示 */
  ai = A[0];
  for (i = 0; i < N; i++) { /* i 行 */
    for (j = 0; j < N; j++) { /* j 列 */
      printf("%6.1f", *(ai+j));
    }
    ai+=N;
    printf("\n"); /* 次の行を表示するための改行*/
  }
  printf("\n"); /* 区切りのための改行 */
  printf("x=\n");
  PrintVector(x,N);

  /* y = Ax の計算 */
  ai = A[0];
  for (i = 0; i < N; i++) {
     *(y+i) = 0; /* 初期化 */
    for (j = 0; j < N; j++) {
    *(y+i) += *(ai+j) * *(x+j);
     }
     ai+=N;
  }
  printf("\n");
  printf("y = \n"); /* ベクトル y(=Ax) の表示 */
  PrintVector(y,2);
  printf("\n"); /* 区切りのための改行 */

  return 0;
}
```

行列×ベクトルの計算

これまでのプログラムでは

```
double A[3][3], x[3];
```

のように配列の大きさを宣言してから利用していた．しかしこれでは，

- 次元 N（行列，ベクトルのサイズ）に依存している
- 次元 N や行列 A，ベクトル $\vec{x}$ の内容が変わるたびにプログラムの修正，コンパイル，実行を行わなければいけない．

そこで，次元 N や行列 A，ベクトル $\vec{x}$ の内容に依存しないプログラムに変更してみよう．

- 行列 A やベクトル $\vec{x}$ の内容変更への対応
  ここでは，scanf( ) などの関数を利用して，実行時に行列，ベクトルの各要素をキーボードから入力することにする．
- 次元 N の変更への対応
  動的なメモリ確保を行う関数 malloc( ) を利用する．
  ⇒プログラムの実行中に行列やベクトルの大きさに応じたメモリ領域を確保することができる．

行列×ベクトルの計算

これまでのプログラムでは

```
double A[3][3], x[3];
```

のように配列の大きさを宣言してから利用していた．しかしこれでは，

- 次元 N（行列，ベクトルのサイズ）に依存している
- 次元 N や行列 A，ベクトル $\vec{x}$ の内容が変わるたびにプログラムの修正，コンパイル，実行を行わなければいけない．

そこで，次元 N や行列 A，ベクトル $\vec{x}$ の内容に依存しないプログラムに変更してみよう．

- 行列 A やベクトル $\vec{x}$ の内容変更への対応
  ここでは，scanf( ) などの関数を利用して，実行時に行列，ベクトルの各要素をキーボードから入力することにする．
- 次元 N の変更への対応
  動的なメモリ確保を行う関数 malloc( ) を利用する．
  ⇒プログラムの実行中に行列やベクトルの大きさに応じたメモリ領域を確保することができる．

動的なメモリ確保

ベクトルの場合

```
double *x;
x= (double *)malloc(N*sizeof(double));
```

⇒　ポインタ x が double 型の要素を N 個もつような配列名と同じように扱うことができる.

- x+i: i 番目の要素へのポインタ
- *(x+i)：i 番目の要素（$x[i]$ と同意）

Example

```
scanf("%lf",&x[i]);  /*i 番目の要素をキーボードから入力*/
printf("%.2f\n",x[i]); /*i 番目の要素を出力*/
```

動的なメモリ確保

行列の場合

```
double *A, *ai;
A = (double *)malloc(N*N*sizeof(double));
```

⇒　ポインタ A が double 型の要素を $N * N$ 個もつような配列名と同じように扱うことができる.
ポインタ変数 ai を導入し，$ai = A;$ とすると，ポインタの表現による行列の場合と同様に第 0 行の各要素が $ai[j]$ で参照でき，次の行へ処理が移るときに,

$$ai+ = N;$$

とすれば，ai が常に行列の第 i 行の先頭へのポインタとして機能する.

例題 3

例題 3

次の行列 A とベクトル $\vec{x}$ を malloc を用いて動的に確保できるようにせよ．キーボードから要素を入力できるようにし，その積 $\vec{y} = A\vec{x}$ をポインタの表現を用いて計算せよ．

$$A = \begin{pmatrix} 3.0 & 2.0 & 5.0 \\ 1.0 & 4.0 & 3.0 \\ 0.0 & 1.0 & 6.0 \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} -3.0 \\ 2.0 \\ -1.0 \end{pmatrix}$$

例題 3 のヒント

- stdlib.h のインクルードを忘れないこと
- malloc したメモリは free 関数でメモリを開放すること
- ベクトルの場合のプログラム例：

```
int N;
double *x;
printf("Input N:");
scanf("%d",&N);
x= (double *) malloc(N*sizeof(double));
if(x==NULL){
    printf("Can't allocate memory. \n");
    exit(1);
}

/*ベクトル x の各要素にキーボードからデータを入力*/
for(i=0; i<N;i++){
   printf("x[%d]=",i);
   scanf("%lf",&x[i]);
}
...
free(x);
```

例題 3 のヒント

- ベクトルの表示

```
void PrintVector(double *y, int n){
   int i;
   for(i=0;i<n;i++){
      printf("%6.1f\n",*(y+i));
   }
}
```

- 行列の表示

```
void PrintMatrix(double *A, int n){
   int i,j;
   double *ai;
   ai= A;
   for( i = 0 ; i < n ; i++){
     for( j = 0 ; j < n ; j++){
       printf("%6.1f", *(ai+j));
     }
     ai+=n;
     printf("\n");
   }
   printf("\n");
}
```

- 計算箇所

```
  ai = A;
  for (i = 0; i < N; i++) {
     *(y+i) = 0; /* 初期化 */
     for (j = 0; j < N; j++) {
       *(y+i) += *(ai+j) * *(x+j);
     }
     ai+=N;
  }
```

```c
#include <stdio.h>
#include <stdlib.h>
void PrintVector(double *y, int n){/*ベクトルを表示する関数*/}
void PrintMatrix(double *A, int n){/*行列を表示する関数*/}
int main(void) {
  int i, j, N;
  double *ai;   /* 各行の先頭へのポインタを表すポインタを用意する*/
  double *A, *x, *y;   /* 計算結果を代入するための行列を用意 */
  printf("N=");
  scanf("%d",&N);

  /*行列・ベクトルのメモリ確保*/

  /*行列・ベクトルの入力*/
  ai = A;
  for( i = 0 ; i < N ; i++ ){
     for( j = 0 ; j < N ; j++ ){
        printf("A[%d][%d] = ", i , j ); scanf("%lf", ai+j );
     }
     ai+=N;
  }
  for( i = 0 ; i < N ; i++ ){
     printf("x[%d] = ", i ); scanf("%lf", x+i);
  }
  printf("A = \n"); PrintMatrix(A,N);
  printf("x = \n"); PrintVector(x,N);

  /* y = Ax の計算 */

  printf("y = \n"); /* ベクトル y(=Ax) の表示 */
  PrintVector(y,N);

  /*メモリの解放*/
  return 0;
}
```

本日のまとめ

- 配列を用いた行列・ベクトルの表現
- 行列の演算
- 配列とポインタ
- メモリの動的確保
- 行列ベクトル積への応用