C言語速習コース
設計と実装

前回学習したこと

- 変数・関数
  - 宣言・定義
  - 配列・多次元配列
  - 関数と引数
  - 演算子
  - スコープ
- 制御構造
  - 条件分岐・条件式
  - While ループ・do-while ループ
  - For ループ
- 入出力
  - printf
  - ファイル入出力

ガウスの消去法

- $$\begin{cases} a_{00}x_0 + a_{01}x_1 + \cdots + a_{0n}x_n = b_0 \\ a_{10}x_0 + a_{11}x_1 + \cdots + a_{1n}x_n = b_1 \\ \cdots \\ a_{(n-1)0}x_0 + a_{(n-1)1}x_1 + \cdots + a_{(n-1)(n-1)}x_{n-1} = b_{n-1} \end{cases}$$

- n元連立一次方程式をガウスの消去法で解く

いわゆるAx=bのxを求める問題

ガウスの消去法の処理手順

- ガウスの消去法のフローは？
  - 講義を思い出して設計してみましょう
    - （ファイルからの入力）
    - 前進消去ステップ
    - 後退代入ステップ
    - 答えの出力

それぞれ関数化して main 内で順番に呼べば良さそう！

補足：引数Nは実質的に不要でした

Hands On !!

ガウスの消去法をしよう（その１）

- 全体のフローを考えて、プログラムの全体像を書いてみよう
  - 関数の作り方（引数）を思いだそう
  - 行列の読み書きだけ作ってみよう
  - ファイルには行列Aとベクトルbの要素が書いてあると仮定する
    - data1.dat

```
#include <stdio.h>
#include <stdlib.h>

#define N 3  // 今回は3行3列の行列を対象とする
void makemat( 引数 ) { 中身 }
void forward_elimination( 引数 ) { /* 後で書く*/ }
void backward_substitution( 引数 ) { /* 後で書く*/ }
void printresult( 引数 ){ 中身 }

int main(int argc, char** argv) {
        double A[N][N], b[N];
        /* ここに引数個数確認を入れておくべき */
        makemat(A, b, argv[1]);
        forward_elimination(A, b, N);
        backward_substitution(A, b, N);
        printresult(A, b, N);
        return EXIT_SUCCESS;
}
```

引数でファイル名を指定しよう

参考：C言語の区切り方

TIPS!!

- C言語は行単位の言語ではない
  - 色々なところで改行できる
  - 1行に複数の処理を書いても良い

```
void funcA(int n, int m);
void funcB
    (int n, int m);
void funcC(int
    n, int m);
```

```
funcA(n,m); funcB(o,p); funcC(q,r);
```

- 変数名や関数名の途中では改行できない
- 改行が意味を持つこともある（例：#define）

- ifやforの中身を括弧で括らない書き方もある
  （括らない場合は直後の１命令だけ実行）

```
if( x == y ) funcA(a,b);  funcB(c,d);
```

- x==yのときはfuncAとfuncBが実行される
- x!=yのときはfuncBだけが実行される

読みにくくなりすぎないように注意すること

参考：コンパイラオプション

TIPS!!

- コンパイルする時にオプションを付加できる
  - 提供される機能はコンパイラにより異なる
  - 同じ機能が異なるオプションで提供されることもある
  - 指定する順番はある程度自由
- 出力ファイル名を変える：-o ファイル名
- 警告を表示する：-Wall
  - gccには警告（warning）を出してくれる-Wallオプションがある
  - 「ちょっとした問題」に容易に気がつけるようになるため、バグを減らす助けになることがある

```
$ gcc -Wall -o b.out source.c
$ ./b.out
```

ガウスの消去法をしよう（その２）

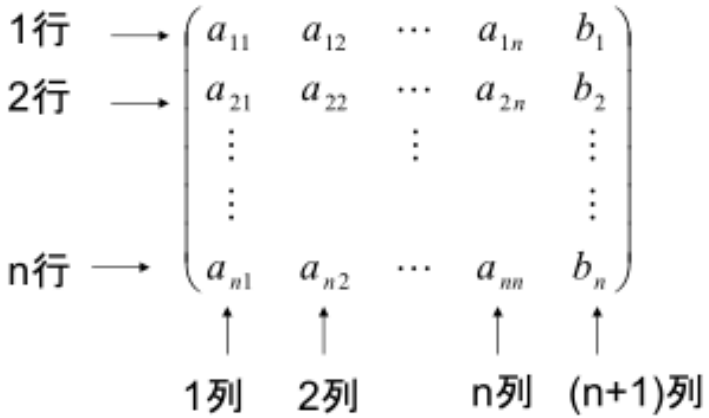
Hands On !!

- ファイルから行列Aとベクトルbを読み込む関数 makemat() を作成しましょう
  - main 関数と同様にフロー設計してみましょう
  - 必要な関数等は一日目に学んでいます
  - データ形式は
    - 1行目 行列Aの0行目の要素 (スペース区切り)
    - N行目 行列AのN-1行目の要素 (スペース区切り)
    - N+1行目 ベクトルbの要素 (スペース区切り)
- 行列Aとベクトルbを表示する関数 printresult() を作成しましょう

前進消去ステップ(1/4)

- 講義を思い出して設計してみましょう。
  - 前進消去ステップ
    - i(最初は1) 行目に注目
      - 各要素をa_{ii}で除算する
      - j=i+1行目に注目
        - 各要素について、i行目の要素をa_{ji}倍した値を減算する
        - N行目まで、jをj+1にして繰り返す
      - N行目まで、iをi+1にして繰り返す

※ベクトルについても行列とあわせて
計算する必要があるので注意が必要



前進消去ステップ(2/4)

- 反復作業をループとして考える

```
void forward_elimination(A, b, n){
    for(i=0; i<n;i++){ // i行目に注目、N行目まで繰り返す
        // i行目の各要素に対する除算
        for(){ // i+1行目に注目、N行目まで繰り返す
            // j行目の各要素に対する減算
        }
    }
}
```

前進消去ステップ(3/4)

- 各要素に対する処理もループとして考える必要がある

```
void forward_elimination(A, b, n){
    for(i=0; i<n;i++){ // i行目に注目、N行目まで繰り返す
        for(){ } // i行目の各要素に対する除算
        for(){ // i+1行目に注目、N行目まで繰り返す
            for(){ } // j行目の各要素に対する減算
        }
    }
}
```

前進消去ステップ(4/4)

- ループの初期値やベクトルbについて考えてみる

```
void forward_elimination(A, b, n){
    for(i=0; i<n;i++){ // i行目に注目、N行目まで繰り返す
        for(j=0; j<n; j++){ A[i][j] = … } // i行目の各要素に対する除算
        b[i] = …
        for(j=i+1; j<n; j++){ // i+1行目に注目、N行目まで繰り返す
            for(k=i; k<n; k++){ A[j][k] = … } // j行目の各要素に対する減算
            b[j] = …;
        }
    }
}
```

ヒント
計算前の値を保存しておかねばならない場合があるのでは？

ガウスの消去法をしよう（その３）

Hands On !!

- 残りを全部作ってみましょう
  - 前進消去を完成させよう
  - 後退代入も同様に作ってみよう

前進消去を完成させた時点で実行し、値を確認してみると
プログラムのミスを早く見つけられるかもしれません

$$\begin{aligned} 2x_1+3x_2+x_3&=5\\ 2x_1+x_2-2x_3&=1\\ x_1+2x_2+3x_3&=7 \end{aligned} \Rightarrow \begin{pmatrix} 2 & 3 & 1 & 5\\ 2 & 1 & -2 & 1\\ 1 & 2 & 3 & 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3/2 & 1/2 & 5/2\\ 0 & 1 & 3/2 & 2\\ 0 & 0 & 1 & 2 \end{pmatrix} \Rightarrow \begin{aligned} x_3&=2\\ x_2&=2-\frac{3}{2}x_3=-1\\ x_1&=\frac{5}{2}-\frac{3}{2}\times(-1)-\frac{1}{2}\times 2=3 \end{aligned}$$

途中の時点でおかしかったら
最終結果が正しいはずがない！

参考：デバッグの仕方

TIPS!!

- デバッグのためのツール（デバッガ）を使う
  - →gdb
- ……というのは、ちょっと大変なので、途中の状態を出力してみると良い
  - 条件式とprintfを組み合わせる
  - 「ここでこの変数はこの値になっているはず」を確認する
  - どこでおかしくなっているかの目安をつける
- デバッガを使うのも、基本的な使い方であれば、そんなに難しいことではない
  - （今回はやりません）

数の表現：浮動小数点演算の注意点

- C言語のfloat型やdouble型はルールに則った方式でデータを保持している
  - n*2^m形式で保持している（IEEE754形式）
- 扱うデータと計算の内容によってはルールの都合により計算結果がおかしくなることがある
- 手書きでは分数として表現できる数値も実数として持たねばならないため誤差が生じる
  - 絶対値があまりに大きな値や小さな値は正しく表現・計算できない
  - 絶対値が大きく異なる値の加減算を行うと、絶対値の小さい値が無視されてしまう
  - 値がほぼ同じ数値同士の差を取ると誤差が大きくなる
  - etc.
- printfの出力上の都合のこともあるので注意

部分ピボット選択の実装

- 何を実装すれば良いか？

- 新たな行の処理を行う際に、絶対値が一番大きな行を選ぶ処理
  - 大小の比較：if文で判定すれば良い
  - 絶対値の取得：負数の場合は正数に変換する？
- 行を入れ替える処理
  - 値を入れ替える：変数の交換は、どう書く？
  - 値を入れ替えない：たくさんの変数を入れ替えるのは処理に時間がかかる、実際には入れ替えずに入れ替えたような扱いをしてはどうか

便利な算術演算関数

- fabs(double f)/fabsf(float f)/fabsl(long double f)
  - 浮動小数変数の絶対値を求める
  - 型によって関数が違う

- 使うには準備が必要
  - コーディング時：#include <math.h> が必要
    - 三角関数や指数・対数関数なども使えるようになる
  - コンパイル時：特別な指定が必要
    - $ gcc gaussian_elimination_XX.c -lm

変数の入れ替え

- 変数aと変数bを入れ替える方法
- 代入してしまうと一方の値がわからなくなってしまう→一時変数が必要

```
x = a;  // aを待避しておく
a = b;  // aを上書きしてしまったが
b = x;  // 待避しておいたから大丈夫
```

- 変数を入れ替える関数は作れるだろうか
  - 変数を関数内で書き換えても呼び出し側に反映されないはず……

値渡しと参照渡し

TIPS!!

- 値渡し
  - 変数をそのまま渡す
  - 関数の仮引数もそのまま
  - 呼び出し側では値が変わらない

```
void func1(int a){
    a = 1;
}
int a = 0;
func1(a);
printf("a = %d¥n", a);
```

- 参照渡し
  - 変数に＆を付けて渡す
  - 関数の仮引数に * を付ける
  - 呼び出し側でも値が変わる

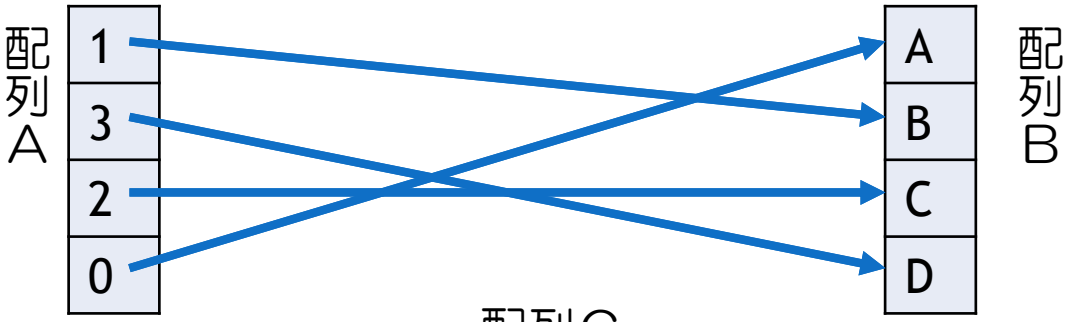
```
void func2(int *a){
    *a = 1;
}
int a = 0;
func2(&a);
printf("a = %d¥n", a);
```

変数の入れ替えを関数化したいときには気をつけよう

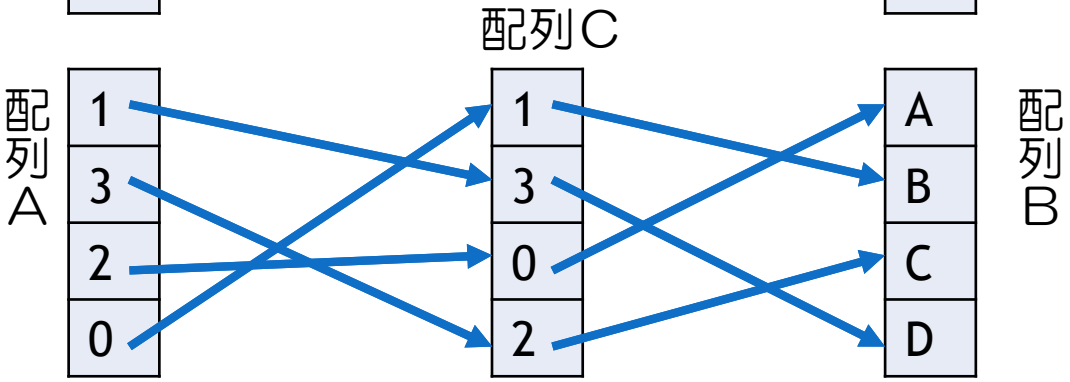
配列の間接参照

- 配列を参照する際に別の配列を経由させることで、参照先を変えることができる
  - メモリ使用量が増える、アクセス速度が低下しうる、というデメリットもある

X[i] = B[A[i]]



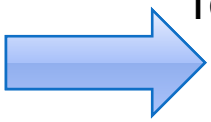
X[i] = B[C[A[i]]]



ガウスの消去法をしよう（その４）

Hands On !!

- 部分ピボッティングを行うガウスの消去法を作ってみましょう



ここで確認と入れかえをすれば良さそう

```
void forward_elimination(A, b, n){
    for(i=0; i<n;i++){ // i行目に注目、N行目まで繰り返す
        for(j=0; j<n; j++){ A[i][j] = … } // i行目の各要素に対する除算
        b[i] = …
        for(j=i+1; j<n; j++){ // i+1行目に注目、N行目まで繰り返す
            for(k=i; k<n; k++){ A[j][k] = … } // j行目の各要素に対する減算
            b[j] = …;
        }
    }
}
```

入力データ：data2.dat
ヒント：間接参照を使う場合は
その後の処理全てに影響する

ガウスの消去法をしよう（その５）

Hands On !!

- 完全ピボッティングを行うガウスの消去法を作ってみましょう

問題サイズの変更

- 対象の行列サイズを変更したくなったらどうすれば良いだろうか
- 毎回ソースコードを書き換えてNを変えるのは大変であり、現実的ではない
  - 大きなプログラムだとコンパイルだけでも結構な時間がかかる
- 実行中に大きさが決まる配列が必要なときはどうすれば良いだろうか
- ソースコードを書き換えられない
  - （十分に大きな配列を確保しておく？）

動的配列の利用(1/3)

- 動的配列（可変長配列）
  - プログラム実行中に任意の長さの配列を作る方法
    - 問題サイズに合わせて配列サイズを変える時などに使う

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
  int *a;
  n = atoi(argv[1]);
  a = (int*) malloc (sizeof(int) * n);
  if (a == NULL){
     perror("MALLOC:");
     exit(EXIT_FAILURE);
  }
  ここで配列 a を使う
  free(a);
}
```

※atoiは文字列から数字を得る関数

nは実行中に決まる

int型で要素数がnの配列を確保
（確保しただけ、中身は不定）

確保できなかったときの処理
ここではエラーメッセージの
出力と異常終了

使い終わったら片付ける

動的配列の利用(2/3)

- 動的配列の確保はmallocで行う
  - 確保できたかどうかはNULL（0のようなもの）でわかる
- 確保された時点では中身は不定
  - 0が入っているかも知れない、入っていないかも知れない（→calloc）
- 確保した配列（メモリ）はどうなるか
  - プログラムが終われば自動的に解放されるため、ほったらかしにしてプログラムを終了しても良い
  - 確保しすぎると足りなくなる（メモリ不足でエラーする）ため不要になったら解放するのが良い
- 明示的に解放するにはfreeを使う
  - mallocとfreeの対応を間違えないように注意すること
    - 同じ配列を複数回freeしてはいけない
    - mallocし直したい時にはfreeしてから

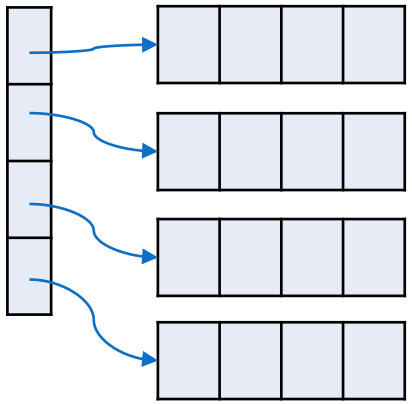
→ if(a!=NULL){ free(a); a=NULL; }

動的配列の利用(3/3)

- 二次元配列の動的確保
  - 主な方法は２つある
  1. 配列のための配列を確保する

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

for(i=0; i<n; i++)free(a[i]);
free(a);
```



  2. 大きな配列を確保して途中を参照する
     - 「ポインタ」を少し知らないとできないため、今回は説明しない

参考：動的配列の関数内確保

TIPS!!

- 関数内でmallocした配列を関数外でも使いたいときはどうするか

```
void func(int *x, int n){
  x = (int*)malloc(sizeof(int)*n);
}
int *a=NULL;
func(a, n);
```

NG
呼び出し元のaはNULLのまま
（配列として使おうとするとエラー）

```
void func(int **x, int n){
  *x = (int*)malloc(sizeof(int)*n);
}
int *a=NULL;
func(&a, n);
```

OK
呼び出し元のaは配列として利用可能

今回の問題では、main関数内でmalloc/freeし、得られた配列を関数に渡すようにすれば大丈夫です
（新しいことを考えなくて済みます）

参考：C言語の仕様について

TIPS!!

- 暗黙な型変換
  - ある程度は勝手に変換してくれる
    - 例：double型とint型を混合して計算・代入できる
    - 基本型については精度の高い型に統一して処理される
    - 代入する場合は代入先の精度に依存
  - 情報が欠落してしまい想定外の結果になることも
  - コンパイラが警告してくれる
    - -Wallオプションを付けておくと確実
  - 必要に応じて明示的に変換する
    - (int)や(double)などを付ける：double d = (double)0;
- main関数
  - int main(int, char**) でないこともある
  - voidだったり引数が無かったりする資料を見ても気にしなくて良い

高速化のヒント

- 正しい結果を得られるプログラムの書き方は１つではない
- プログラムの作り方次第で性能（実行時間）に大きな差が生じることがある
- 100倍、1000倍以上の差が生じることもあるため、高速化はとても重要
  - １日かかるプログラムが、高速化によって数秒で終わるようになるかもしれない
  - １時間で終わるはずのプログラムが、作り方が悪いと１週間以上かかってしまうかも！？
- 高速化手法を幾つか紹介する

CPUが行いやすいような処理にする

- CPUはどんな計算も同じ速度で行えるわけではない（計算式の書き方で性能が変わる）
  - 一般的に、除算はとても重い
  - 加算・減算・乗算はあまり変わらない
  - 論理演算が高速なことがある
- 同じ計算を何度も行う場合はまとめると良い
  - 一度変数に格納しておいて、変数を参照するだけにする
- 分岐処理を減らすと良い
  - 例：多重ループの最内部でifを使わない

配列アクセスを高速化する

- 同じ回数の配列アクセスでも所要時間に差が生じる
  - 連続したアクセスは高速
  - 連続でない（ランダムな、飛び飛びな）アクセスは低速
- 多次元配列のアクセス順序をよく見てみよう
  - ループを入れ替えても計算結果が変わらないが、性能が全く異なってしまうことがある
- 同じ配列要素に何度も書き込む場合には、一旦局所変数上で計算を行って、最後に配列に書き込んだ方が良いことがある
  - 例：多重ループの最内部における足し込み処理

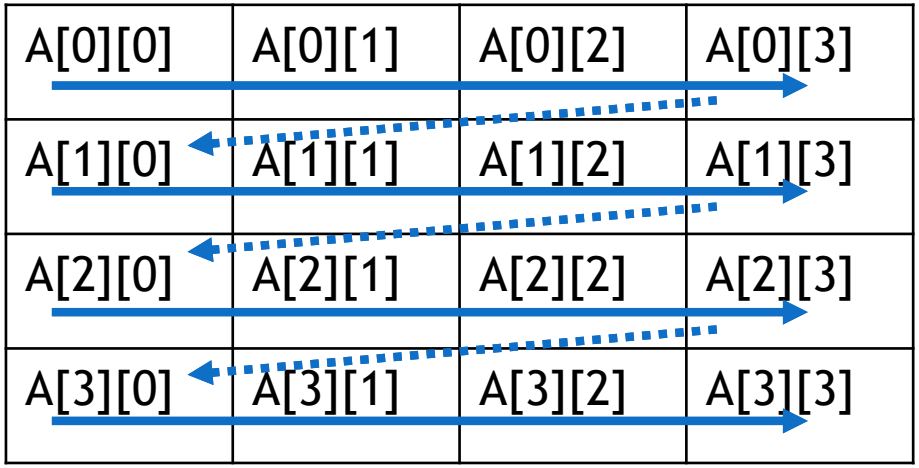
二次元配列への連続アクセスの例

- 配列に順番にアクセスできているかを考える

```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    A[i][j] = …
  }
}
```

どちらが高速？

```
for(j=0; j<n; j++){
  for(i=0; i<n; i++){
    A[i][j] = …
  }
}
```



コンパイラに高速化してもらう

- 世の中の多くのコンパイラにはプログラムを最適化する機能が備わっている
- 最適化オプションを使う
  - 例えばgccでは-O1 -O2 -O3（数字が大きい方が強力な最適化を行う）など
  - コンパイラによって異なるため確認が必要
- 常にうまく行くわけではない
  - 簡単な（コンパイラにとってわかりやすい）プログラムはうまく行きやすい
  - 手動での最適化も重要

- これまでに挙げた高速化技術を適用しなくても、コンパイラによる最適化だけで十分に良い性能が得られることもある
  - 最適化の有無とコンパイラオプションの組み合わせで性能が変わる

配列の扱いに関する注意点

- malloc/freeには時間がかかる
  - ループ内で何度もmalloc/freeするようなプログラムは作らないようにする
- あまりに大きなローカル変数は扱えない
  - 巨大な配列（行列）を静的に使いたいときはグローバル配列にする
  - 動的配列であれば大丈夫

ガウスの消去法をしよう（その6）

Hands On !!

- ガウスの消去法を高速化してみよう
  - 計算を置き換えられるところは無いか？
  - 計算順序・配列アクセス順序を変更できないか？
  - コンパイルオプションを変えてみよう

  - 問題サイズが小さいと差がわからないため、大きな問題を作ってみましょう
  - 実行時間の測定には time コマンドが便利です

timeコマンドに続けて測定したいコマンド列を与える
$ time ./a.out data.dat

参考：ライブラリの利用

TIPS!!

- それぞれの分野における「よく使うお決まりの処理（関数）」というものが存在する　（例：行列積など）
- プログラミングの得意な人が作った（最適化された）処理を使い回せれば便利
- ソースコードを共有する？
  - プログラムの中身が全部見られてしまう……
- ライブラリを使う
  - 簡単に言えば、関数を切り出して別のプログラムからも使えるようにしたもの
- ライブラリを作る方法
  - コンパイラにオプションを指定、専用プログラムでまとめる
- ライブラリを使う方法
  - 宣言を行う：#include <xxxx.h>
    - 指定したファイルの中に宣言が書かれている
  - コンパイラに使うライブラリを教える
    - gcc source.c -lxxxx という書式で指定する

参考：古い記法

TIPS!!

- 古めの教科書に書かれたC言語のプログラムを読んでいると古い書き方のプログラムに出くわすことがある
- 読み方自体は難しくないので安心
- 古い記法の例

```
int func(a, b, n)
double a[][N], b[];
int n;
{
    int i,j, ...
```

  - 仮引数に型情報が無い
  - 関数本体の前に変数名と型情報が並んでいる
- 関数の型と名前だけを宣言として書く（仮引数を書かない）記法もある

速習コースに関する問い合わせは担当助教
實本（jitumoto at cc.u-tokyo.ac.jp）
大島（ohshima at cc.u-tokyo.ac.jp）
までメールでお願いします

（gmail等からの送信でも特に問題ありませんが、
送信者の所属と氏名は明記してください）