この実習の位置づけ： 1 年次の「情報入門」を基礎として 2 年次では「プログラミング I」（C によるプログラミング）と「数式処理実習」（Mathematica による数式処理）を学びました。本実習では統計計算とグラフィクスのための言語環境として知られる R を使い、確率的なシミュレーションについて学びます。Mathematica は数式を扱うためのソフトウェアでしたが、R は大量の数値（データ）を扱うためのソフトウェアです。もし C によるプログラミングを身に付けていれば、ほとんど迷うことなく R のプログラミングをすることができます。

単位： 出席状況と実習課題の達成度によって評価します。実習は、ペアプログラミング方式で行います。

プログラミング作業の流れ： まず本実習用のフォルダ（ディレクトリ）を作っておき、プログラムはすべてそこに保存するようにします。フォルダを作る場所とフォルダの名前はたとえば `H:¥R` とすればよいでしょう。作成手順はマウスで「スタート」→「コンピューター」→「ホームフォルダ (H:)」と辿って、右クリックで「新規作成」→「フォルダー」を選び、最後にキーボードから「R」と入力します。さて R のソースコードを書くには、まず R エディタを開きます。手順は「スタート」→「すべてのプログラム」→「R」→「日本語版」→「R i386 2.15.1」で R を起動して、「ファイル」から「新しいスクリプト」を選びます。これによって自動的に R エディタが起ち上がります。R エディタで実際のプログラムコードを書き始めるまえに、まず先にファイル名を決めて、先ほどの場所（`H:¥R`）に保存する習慣をつけておくと安全です。拡張子は `.R` としましょう。書いたプログラムコードを実行するには、マウスで「編集」→「全て選択」→「カーソル行または選択中の R コードを実行」を選びます。しかし、いちいちマウスに手を移すのは面倒なので、この操作はキーボードから「`Ctrl+A Ctrl+R`」とすることでも実現できます。

ベクトルの処理（その 1）： R におけるデータ型の基本はベクトルです。ベクトルを上手に操作できるようになれば、R を上手に扱うことができます。R のベクトルは C の配列よりも柔軟なので、扱いは簡単です。

- ベクトルを作るにはコマンド `c` を使います。この `c` は英語の concatenate（繋げる）あるいは combine から来ています。例を見ましょう。

– ベクトル $x = (2, 5, 9)$ を作る。3 つの数値 $2, 5, 9$ を繋げる、と考えます。

```
x=c(2,5,9)
```

C 言語と違って変数 `x` の型を宣言しておく必要はありません。

– ベクトル $x = (2, 5, 9)$ にベクトル y を付け足す。

```
x=c(2,5,9)
x=c(x,y)
```

この例のように、ベクトルは後からそのサイズを次々に変更していくことができます。コマンド `c` は数値も文字もベクトルも関係なく繋ぎます。

– コマンド `c` の代わりにコマンド `append` も使えます。後者のほうがやや高機能で、たとえば `append(x,y,after=3)` と書けば、ベクトル x の第 3 成分のすぐあとにベクトル y を挿入します。

- ベクトルの成分をとりだすには鉤括弧 `[,]` を使います。

– ベクトル x の第 1 成分をとりだすには `x[1]` と書きます。C 言語と違ってベクトルの要素番号は 1 から始まります。

– 複数の成分を一斉にとりだすこともできます。たとえばベクトル x の第 1 成分と第 3 成分をとりだすには `x[c(1,3)]` とします。

基本的な演算子： R ではベクトルに対する 2 項演算子 `+`, `-`, `*`, `/`, `^` が用意されています。これらの演算子の意味はそれぞれ、足し算、引き算、掛け算、割り算、累乗で、それぞれの計算はすべてベクトルの成分ごとに行われます。すなわち同じ次元の 2 つのベクトル $x = (x_1, \ldots, x_n), y = (y_1, \ldots, y_n)$ に対して

$$\begin{aligned}
\texttt{x+y} &= (x_1 + y_1, \ldots, x_n + y_n) \\
\texttt{x-y} &= (x_1 - y_1, \ldots, x_n - y_n) \\
\texttt{x*y} &= (x_1 y_1, \ldots, x_n y_n) \\
\texttt{x/y} &= (x_1 / y_1, \ldots, x_n / y_n) \\
\texttt{x\^{}y} &= ({x_1}^{y_1}, \ldots, {x_n}^{y_n})
\end{aligned}$$

です。実際にはベクトル x, y の次元は必ずしも同じである必要はなく、高いほうの次元が低いほうの次元の倍数になっていれば計算が定義されます。その場合には、低いほうの次元のベクトルを何度か繰り返してもう一方のベクトルの次元に揃えてから、成分ごとに計算が実行されます。これは後述するように、とくに低いほうの次元が 1（すなわち数）である場合に便利に利用できます。
また、このほかの演算子として `%/%`, `%%` もあり、これらはそれぞれ割り算の商と割り算の余りを意味します。たとえば `10%/%3` の値は `3` であり `10%%3` の値は `1` です。ベクトルに対してはやはり成分ごとに計算します。

ベクトルの処理（その 2）： 規則性のあるベクトルは次のように生成することができます。

- ベクトルの成分が等差数列をなす場合。

– たとえば、ベクトル `c(1,2,3,4,5,6,7,8,9,10)` は短く

```
1:10
```

とも書きます。これは 1 から 10 までの整数を順番にならべよ、という意味で、この表記は今後いろいろの場面で使います。あるいは `seq(1,10)` と書いても同じ結果を得ます。また `10:1` と書けば、10 から 1 までの整数を順番にならべよ、という意味になり、これは `c(10,9,8,7,6,5,4,3,2,1)` と書くのと一緒です。

– より一般に、たとえば公差 2 の等差数列は

```
seq(1,10,by=2)
```

と書きます。これは初項が 1, 公差が 2, 末項が 10 を超えない最大の値となるようなベクトルを作れ、という意味で、具体的には `c(1,3,5,7,9)` と書くのと同じです。末項の値でなく項数を指定したい場合には `seq(1,by=2,length=5)` と書きます。

- 等比数列は、累乗の演算子 `^` を使って作ることができます。たとえばベクトル `c(2,4,8,16,32,64,128,256,512,1024)` は

```
2^(1:10)
```

と書けます。この式は R では次のように解釈されます。ベクトル `2` とベクトル `1:10` では次元が違うので、まず次元を揃えてから、そのあとで成分ごとに累乗の計算をします。すなわち

$$\texttt{2\^{}(1:10)} = (2, 2, \ldots, 2) \texttt{\^{}} (1, 2, \ldots, 10) = \left(2^1, 2^2, \ldots, 2^{10}\right)$$

となります。

- 同じものの繰返し

– ベクトル x を 3 回繰り返して新しいベクトルを作るには

```
rep(x,3)
```

とします。特に 3 次元の零ベクトル `rep(0,3)` を `numeric(3)` とも書きます。

ベクトルの処理（その 3）： ベクトルの成分を並べ替えたり成分を取り出したりするのに便利なコマンドとして、たとえば次のものがあります。

- `sort` は成分を値の小さいほうから順に並べ替えます。
- `sample` は成分をシャッフル（ランダムに並べ替え）します。このコマンドは後にシミュレーションをする際に便利に使えます。
- `rev` は成分を逆順に並べ替えます。
- `unique` は重複した値を取り除きます。
- `sum` は成分の総和を計算します。

データ処理実習課題（1 回目）　　学籍番号の下 2 桁 ☐☐ 名前＿＿＿＿＿＿＿＿＿＿

学籍番号の下 2 桁 ☐☐ 名前＿＿＿＿＿＿＿＿＿＿

R では数値が取り扱えないくらい大きくなると `Inf`（無限大）と表示します。この `Inf` はどのくらい大きい数値になると表示されるのでしょうか。たとえば `1/0` は `Inf` と表示されて自然ですが `10^500` も `Inf` となります。そこで `Inf` と表示されないようなぎりぎりの整数値を探してください。

前回の実習課題について： まず 10^(300:400) などを実行すれば 1e+308 より先がずっと Inf となるので、話題を整数に限れば、R ではたかだか 309 桁の整数までしか扱えないことが分かります。これをもうすこし詳しく調べてみましょう。まず 10 の冪で計算するよりも 2 の冪で計算した方が便利でしょう。そこで 2^(1000:1100) を実行すれば 2^1023 までは有限値で、2^1024 以降が Inf となるのが分かります。そこで試しに sum(2^(1023:0)) を実行してみると、この値は Inf となってしまいます。もっと小さくして sum(2^(1023:1000)) を実行すると、この値は 1.797693e+308 です。すこし増やして sum(2^(1023:990)) を実行すると、この値もやはり 1.797693e+308 と表示され、変化がありません。どうやら 1.797693e+308 が「ぎりぎりの整数値」になりそうです。しかしこれは上位桁だけに注目した大雑把な数値なので、ほんとうに「ぎりぎり」の値だとは言えません。正確な値（および正確な値を求めることにあまり意味がないこと）については、プリントの最後で説明します。

....

今回は if 文と for 文と while 文、および関数定義（自作の関数をつくる方法）について実習します。プログラミング I（C 言語）を履修した人にとってはお馴染みの内容です。

条件分岐（if 文）： 条件によって計算方法を変えたり、場合分けをして処理を行いたい場合には if 文を用います。if 文の書式は

```
if (条件式) {
    もし条件式が TRUE ならばこの部分が実行される
} else {
    もし条件式が FALSE ならばこの部分が実行される
}
```

です。条件分岐にもちいる「条件式」では、次に示す関係演算子を使って 2 つの値を比較することが多く、その結果は TRUE か FALSE のどちらか一方になります。また、後半の else の部分は必ずしも必要ではありません。

演算子	意味	演算子	意味
==	等号（左辺と右辺が等しい）	!=	$\neq$
>	$>$	<	$<$
>=	$\geq$	<=	$\leq$

繰り返し（for 文）： コンピュータは一定の仕事を繰り返して実行し続けるのが得意であって、そのような仕事に高速計算の威力を発揮します。繰り返しの基本構造には 2 つの種類があります。ひとつは、繰り返しの実行回数を定めておくものであり、これには for 文を用います。もうひとつは、繰り返しのたびに条件判断を行って続行や打ち切りを決定するものであり、これには while 文を用います。for 文と while 文の書式はそれぞれ

```
for (ループ変数 in リスト) {
    指定した回数だけこの部分が繰り返し実行される
}
```

```
while (条件式) {
    もし条件式が TRUE ならばこの部分を実行したのちにふ
たたび条件式を評価する。もし条件式が FALSE ならばこの部
分は実行しないで while 文を抜ける。
}
```

です。また、先ほどの例でも見たように、for や while などの繰り返し文の途中でその繰り返しから抜けるには break を使います。

関数定義： これまで見てきたように、R での処理は一連のコマンド（関数）を用いて行なわれます。R には始めから多くの関数が定義されていて、それだけでも一通りのデータ解析ができますが、R の真の実力は「自分で関数を書く」ことで発揮されます。自作の関数はつぎの書式で作ります。

```
関数名 = function (引数) {
    関数本体
}
```

ここでは if 文と for 文の練習をかねて、自然数の階乗を計算する関数を 2 通りの方法で定義してみましょう。まず for 文で書いてみます。

```
fact=function(n) {
    x=1                # 初期値を 1 とする。
    for (i in 1:n) { # i が n 以下のあいだ、
        x=i*x          # 積 i*x の値を改めて x とおく。
    }
    return(x)          # 最後に x の値を出力する。
}
fact(10)               # 試しに 10 の階乗を計算させる。
```

return は計算結果を出力するための命令です。次に if 文で書きましょう。階乗が等式 $n! = n \cdot (n-1)!$ をみたすことを利用して、関数を再帰的に定義します。

```
fact2=function(n) {
    if (n==1) {
        return(1)
    } else {
        return(n*fact2(n-1)) # ここがポイント
    }
}
```

これら 2 つのプログラムは汎用的な書き方であり、どのプログラミング言語にも通用する「プログラミングのいろは」的内容です。まずはこの 2 つの書き方をマスターしましょう。それを理解したうえで、実は R では階乗の定義はつぎのようにも書けます。

```
fact3=function(n) {
    return(prod(n:1)) # sum は総和だが prod は総積
}
```

プログラムの行数が半分で済みました。これがベクトルを生かした R らしい書き方です。前回の実習で述べたように R ではデータ型の基本がベクトルのため、ほとんどループを書く必要性が消えてしまうのです。このように、ベクトルを上手に扱うことで、書く者にとっても読む者にとっても R のプログラミングが簡単になります。

....

ふたたび前回の実習課題について： 「ぎりぎりの整数値」を調べるためには、たとえば

```
x=10^(308:0)
y=numeric(309)
for (n in 1:length(y)) { # length(y) は y の次元
    for (i in 9:0) {
        y[n]=i
        if (is.finite(sum(x*y))) {
            break
        }
    }
}
```

などとコードを書けばよいでしょう。最大の整数が

179769313486231576914999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999999999999999999999999999999
999999999

であることが分かります（ただしこの値は使用機材に依存）。実際、これに 1 を足した整数は Inf と表示されます。試してみてください。やってみると分かりますが、試し方には注意が必要です。

データ処理実習課題（2 回目）　　学籍番号の下 2 桁 [ ][ ]　名前＿＿＿＿＿＿＿＿＿＿

学籍番号の下 2 桁 [ ][ ]　名前＿＿＿＿＿＿＿＿＿＿

1 自然数 n に対して「もし n が偶数なら 2 で割り、そうでなければ 3 倍して 1 を足す」という操作を繰り返すと、有限回で 1 に到達するであろうことが予想されている（コラッツ予想。数学における未解決問題のひとつ）。たとえば 3 から始めると $3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$ となり 7 ステップで 1 に到達する。

(1) 自然数 n に対してそれが 1 になるまでのステップを表示するためのプログラムを次のように書いた。空欄を埋めよ。ただしこのプログラムの出力は「[1]　1　2　4　8 16　5 10　3」である。

```
collatz=function(n) {
    x=n                                  # ベクトル x の初期値を 1 次元のベクトル n とする。
    while(                    ) {        # ベクトル x の第 1 成分が 1 でない限りつぎの操作を続ける。
        if(                          ) { # もしベクトル x の第 1 成分が偶数だったら
            x=                           # その数字を 2 で割ってベクトル x の先頭に追加する。
        } else {
            x=
        }
    }
    return(x)
}
collatz(3)
```

(2) 5000 以下の自然数のうち、1 になるまでのステップ数が最大であるものを探したい。そのためのプログラムを書け。
ヒント①　別のファイル（たとえば H:¥R¥collatz.R とする）に書いてある自作の関数を利用したい場合には、コマンド source を使って source("H:/R/collatz.R") とファイルを読み込むことによって、その関数が使えるようになる。
ヒント②　関数 f に対して、値 f(1),f(2),...,f(10) を計算したい場合には、コマンド sapply を使うと便利。たとえば sapply(1:10,f) と書く。詳しくは help(sapply) せよ。
ヒント③　help(which.max)

2 質問または感想を記せ。

シミュレーション入門： これまでの実習内容を基礎にして簡単なシミュレーションをしてみましょう。シミュレーションは日本語でいうと模擬実験です。「実際の実験を行うには費用または時間がかかりすぎる」場合や「実際の実験を行うための環境を設定するのが難しい」場合などに行います。シミュレーションは、実世界や何らかの仮説的状況をコンピュータ上でモデル化するもので、それによってそのシステムがどのような振舞いをするのか研究することができます。もしモデルがパラメータを含む場合には、その値を変化させることによって未来の予測を立てることもできます。

手始めに、確率のやさしい問題から考えてみましょう。

> コイン投げを 3 回行ったとき 3 回とも表がでる確率を求めよ。

すぐに $(1/2)^3 = 12.5\%$ という答えが出せます。では、確率の計算を使わずに解くにはどうすればよいでしょうか。簡単です。実際にコインをもってきて 3 回トスし、3 回とも表になるかどうかを記録します。この実験を何度も行えば、だいたいどのくらいの割合（確率）で起こるのかが調べられます。これをコンピュータ上で再現するのがシミュレーションです。この「コイン投げ 3 回」の実験を 1000 回行うシミュレーションをしてみます。たとえば次のようなプログラムを書きます。

```
count=0                      # カウンタを用意する。
for (i in 1:1000) {
    x=sample(0:1,3,replace=T) # コインを 3 回投げる
    if (sum(x)==3) {         # もし 3 回とも表だったら
        count=count+1        # カウンタの値を+1 する。
    }
}
count/1000
```

では次の問題はどうでしょうか。

> コイン投げを 10 回行ったときどこかで 3 回連続して表がでる確率を求めよ。

これをすぐに確率計算できる人は少ないだろうと思います。すぐに答えが分からないのでシミュレーションしてみましょう。

```
count=0
for (i in 1:1000) {
    x=sample(0:1,10,replace=T) # コインを 10 回投げる
    y=numeric(8)
    for (j in 1:8) {
        y[j]=x[j]*x[j+1]*x[j+2]
    }
    if (sum(y)>0) { # もしどこかで 3 回連続表だったら
        count=count+1        # カウンタの値を+1 する。
    }
}
count/1000
```

だいたい 0.5 くらいになりそうです。きちんと確率計算すると、答えは $65/128 = 0.507\cdots$ となります。たぶん。

このようなシミュレーションを一般にモンテカルロ法（あるいはモンテカルロ・シミュレーション）と呼びます。モンテカルロ法とは、乱数を用いたシミュレーションを何度も行うことにより、考えている問題の近似解を得る方法のことです。これにより、解の真値を求めることができない問題に対しても、多数回のシミュレーションを繰り返すことにより、解の近似値を求めることができます。
モンテカルロ法は、確率的な問題に対してだけ使える方法ではなく、非確率的な問題の近似解を求める方法としても使えます。たとえば単位円の面積（もちろん π です）を調べてみましょう。「モンテカルロ法というのはこのプログラムのこと」と誤解されているくらい有名なシミュレーションです。砂粒を 1 粒ずつ平面上のランダムな位置に落としていくイメージでプログラムを読みます。

```
count=0
N=10000
for (i in 1:N) {
    p=runif(2,min=-1,max=1) # [-1,1] の乱数を 2 個。
    if (sum(p*p)<=1) {      # もし円の内部にあったら
        count=count+1       # カウンタの値を+1 する。
    }
}
4*count/N
```

つぎのプログラムも単位円の面積の近似値を求めます。方法の違いは分かりますか。

```
count=0
M=100
a=seq(-1,1,length=M) # [-1,1] を M-1 等分。
for (i in 1:M) {
    for (j in 1:M) {
        p=c(a[i],a[j])
        if (sum(p*p)<=1) {
            count=count+1
        }
    }
}
4*count/M^2
```

これは領域 $[-1,1]\times[-1,1]$ を等間隔に格子に区切って、その格子上の格子点が単位円の内部に入るかどうかを調べています。乱数を使っていないので毎回同じ値になります。円のような単純な図形の場合には両プログラムの結果はさほど違いませんが、周期の短い三角関数のような場合、もし周期と格子点の間隔が合ってしまうと面積の値が大幅にずれてしまいます。たとえば単位円板 $x^2+y^2<1$ の代わりに図形 $y<\cos(100\pi x)$ の面積を考えてください。モンテカルロ法ならば図形によって精度が落ちることはなく、面積の真値にほぼ一致します。
以上 4 つのプログラムを見ました。どのプログラムも基本的な構造が同じであることに気づいたと思います。これがモンテカルロ法によってシミュレーションを行う際のプログラムの雛型です。
さて、ここでシミュレーションの精度について考えてみましょう。これは前期の「統計」で学んだ内容です。大数の法則が主張しているとおり、だいたいの目安は、精度を 1 桁（つまり 10 倍に）上げるには、データの個数を 100 倍にする必要があります。N を 100, 10000, 1000000 として実行してみてください。ただし N=1000000 の場合を処理するには時間がかかります。

余談。最後のプログラムは次のようにも書けます。

```
M=100
a=seq(-1,1,length=M)
b=a^2
f=function(x,y) { # 引数 x,y は共に M 次元の縦ベクトル
    return(x%*%t(rep(1,M))+rep(1,M)%*%t(y))
}
4*sum(as.numeric(f(b,b)<=1))/M^2
```

演算子 **%*%** は行列の積を意味し、コマンド t は行列を転置します。ただしコマンド c, seq, rep などで作ったベクトルはすべて縦ベクトルとして解釈されます。関数 f は入力のベクトル $(x_i), (y_j) \in \mathbb{R}^M$ に対して M 次正方行列 (a_{ij}), $a_{ij} := x_i + y_j$ を出力する、いわば「総当たりで和をとる」ための関数です。蛇足ながら「総当たりで積をとる」操作はクロネッカー積といって、これには始めから演算子 **%x%** が用意されています。さて、このプログラムが処理している内容は最後のプログラムとまったく同じですが、こちらの方がずっと高速です。たとえば M=10000（したがってデータ数は 1 億個）にすると違いがよく分かります。コマンド system.time を使って処理時間を計ると、私の環境では（マシンの性能次第です）前者は約 221 秒かかり、後者は約 2.2 秒で済みます。本プログラムは高速でうれしいのですが、意味が分かりづらいという欠点があります。

データ処理実習課題（3 回目）　　学籍番号の下 2 桁 ☐☐ 名前＿＿＿＿＿＿＿＿＿＿

学籍番号の下 2 桁 ☐☐ 名前＿＿＿＿＿＿＿＿＿＿

1 定積分 $\int_0^1 \exp\left(-x^2\right) dx$ の近似値をモンテカルロ法で求めよ。ただし小数点以下第 2 位までが正しい値となるようにすること。

2 25 株の植物を 1 列に並べて植えたところ、ある病気が発生した。病気にかかった株 (D) と病気にならなかった株 (H) は

HHHHHHHHHHHHHHDDDHHHHHHHHHD

の順番で並んでいる。このデータから病気が伝染性をもつと考えられるか。
ヒント：21 個の H と 4 個の D をランダムに並べたときに連の個数が 4 以下となる確率を求める。

前回の実習課題について： 定積分 $\int_0^1 \exp\left(-x^2\right) dx$ の近似値をモンテカルロ法によって求めます。定積分を図形の面積と解釈して

```
count=0
N=1000
for (i in 1:N) {
    p=runif(2,min=0,max=1)
    if (p[2]<=exp(-p[1]^2)) {
        count=count+1
    }
}
count/N
```

と書きます。小数点以下第 2 位までが真値となるようにするにはどうすればよいでしょうか。シミュレーションの精度の話を思い出しましょう。データ数を 100 倍に増やすと精度がやっと 1 桁上がります。実際 N=10, 1000, 100000 に対して、このプログラムをそれぞれ 5 回ずつ実行してみます。乱数を使っているので実行結果は毎回異なりますが、たとえば次のようになります。

N=10	0.5	0.8	0.6	0.8	0.7
N=1000	0.763	0.726	0.752	0.739	0.750
N=100000	0.74375	0.74760	0.74603	0.74568	0.74754

この表から、定積分 $\int_0^1 \exp\left(-x^2\right) dx$ の値を小数点以下第 2 位まで正確に読みとることができます。ちなみに不定積分 $\int \exp\left(-x^2\right) dx$ を具体的に計算することはできません。この不定積分は初等関数の範囲で表現できないことが知られています。このことを数学では、不定積分 $\int \exp\left(-x^2\right) dx$ 自体が新しい関数を定義しているのだ、と考えます。このように一般には既知の関数を積分すると未知の関数が現れます。たとえば小学校で習う有理関数 $1/x$ を積分すると高校で習う対数関数 $\log x$ が出てきます。
次の問題。ヒントにしたがって 21 個の H と 4 個の D をランダムに並べたときに連の個数が 4 以下となる確率を求めます。たとえば次のようなプログラムを書けばよいでしょう。

```
count=0
N=10000
x=c(rep(0,21),rep(1,4)) # 0 を 21 個、1 を 4 個用意。
for (i in 1:N) {
    y=sample(x)       # 0,1 をランダムに並べ替える。
    z=numeric(24)     # 連の境界を記録するベクトル。
    for (j in 1:24) { # y の要素を順番に見ていって、
        if(y[j+1]!=y[j]) { # もし連の境界だったら、
            z[j]=1         # その位置を記録する。
        }
    }
    if (sum(z)<=3) { # もし連の個数が 4 以下だったら
        count=count+1
    }
}
count/N
```

これを実行すると、連の個数が 4 以下となる割合（確率）は約 1.2% くらいと分かります。小さな確率です。この数字を「普通ならほとんど起こらないはずのことが起こった」と解釈して「連の個数が 4 つしかないのは病気に伝染性があるためだ」と結論づけます。これは前期の「統計」で学んだ検定の考えかたです。

.....

これまでの実習でベクトルの扱い、条件分岐、処理の繰り返しについて学び、簡単なシミュレーションを行いました。今後もうすこし複雑なシミュレーションを行うために行列について説明します。

行列の処理： 行列を作るにはコマンド `matrix` や `cbind` を使います。

- たとえば行列

$$\begin{pmatrix} 2 & 5 & 9 \\ 1 & 3 & 9 \end{pmatrix}$$

を作るには次のようにします。

– 成分をひとつずつ指定して。

```
A=matrix(nrow=2,ncol=3) # row は行, column は列
A[1,1]=2
A[1,2]=5
A[1,3]=9
A[2,1]=1
A[2,2]=3
A[2,3]=9
```

これは分かりやすい方法ですが入力するのが面倒です。

– ひとつのベクトルを分割して。
これが行列をつくるときの標準的な方法です。まずはこの方法をマスターしましょう。

```
x=c(2,1,5,3,9,9) # 並べる順番に注意
matrix(x,nrow=2,ncol=3) # 列優先で格納
```

列優先で格納されることに注意します。行優先で格納するにはオプションを付けて次のようにします。

```
x=c(2,5,9,1,3,9) # 並べる順番に注意
matrix(x,nrow=2,ncol=3,byrow=T) # 行優先で格納
```

– 縦ベクトルを束ねて。

```
x=c(2,1)
y=c(5,3)
z=c(9,9)
cbind(x,y,z)
```

これはコマンド `matrix` を使って次のようにも書けます。

```
x=c(2,1)
y=c(5,3)
z=c(9,9)
w=c(x,y,z)
matrix(w,nrow=2,ncol=3)
```

– 横ベクトルを束ねて。
行列を作るコマンドとしてもうひとつ `rbind` というのがあります。これは基本的には「横ベクトルを束ねて」行列をつくるためのコマンドですが、引数として与えられたベクトルが縦ベクトルなのか横ベクトルなのかを区別しない（柔軟性を持たせるためわざとそのように設計してあります）ため、数学科の私たちにとってはかえって分かりづらく、混乱しやすいコマンドなので使わないほうがよいでしょう。横ベクトルを並べて行列を作るには、上で見たのと同様に、いったんそれらの横ベクトルをすべて結合してから、コマンド `matrix` にオプション `byrow=T` をつけて適切に分割します。

- 特殊な行列は、コマンド `matrix` を使わずにもっと簡単に作ることができます。たとえば対角行列は `diag(c(2,3,5))` など。
- 行列 `A` の成分を取り出す。

– 第 1 行第 2 列の成分を取り出すには `A[1,2]` とします。

– 第 1 行をまとめて取り出すには `A[1,]` とします。

– 第 1 列をまとめて取り出すには `A[,1]` とします。

– 小行列を取り出すには例えば `A[c(1,3),c(2,5)]` とします。

- 行列の積を表す演算子は `%*%` です。たとえば `A%*%B` とします。これに対して `A*B` は要素ごとの積（アダマール積）を計算してしまいますので注意しましょう。
- 逆行列は `solve` で求めます。

データ処理実習課題（4 回目）　　学籍番号の下 2 桁 □□　名前＿＿＿＿＿＿＿＿

学籍番号の下 2 桁 □□　名前＿＿＿＿＿＿＿＿

1 A と B の 2 人がさいころを交互に 2 回ずつふる。もし A が目の和 6 をだせば A を勝者とし、もし B が目の和 7 をだせば B を勝者としてゲームを終える。ただしゲームは A から始めるものとする。

(1) A が勝者となる確率をシミュレーションによって推定せよ。

(2) 確率 0.95 の判断では何試合すればゲームが終わるか。その試合数をシミュレーションによって推定せよ。ただし A または B がさいころを 2 回ふることを 1 試合と数える。

ヒント：勝負がつくまでの試合数を X とし、ちょうど k 試合目に勝負がつく確率を $P(X=k)$ と書くとき、$P(X=1)+P(X=2)+\cdots+P(X=n)\geq 0.95$ をみたす最小の n が知りたい。

前回の実習課題について： いろいろな書き方ができますが、正解プログラムの例を挙げます。

```
hand=function(){
    sum(sample(1:6,2,replace=T))
}
count=0
N=10000          # ゲームの回数
for (i in 1:N) {
    A=hand()     # A の手
    B=hand()     # B の手
    while (A!=6 && B!=7) {
        A=hand() # A の手を更新
        B=hand() # B の手を更新
    }
    if (A==6) {count=count+1}
}
count/N
```

次の問題はすこし難しかったかもしれません。

```
# 手
hand=function(){
    sum(sample(1:6,2,replace=T))
}
# ゲームを 1 回行って、要した試合数を返す関数
howmany=function(){
    count=1
    A=hand()     # A の手
    B=hand()     # B の手
    while (A!=6 && B!=7) {
        A=hand() # A の手を更新
        B=hand() # B の手を更新
        count=count+1
    }
    if (A==6) {  # A が勝ったなら
        2*count-1
    } else {     # B が勝ったなら
        2*count
    }
}
# k 試合目でゲームが決まる確率（の近似値）を返す関数
prob=function(k) {
    count=0
    trial=1000
    for(i in 1:trial) {
        if (howmany()==k) {
            count=count+1
        }
    }
    count/trial
}
n=1
x=prob(n)
while (x<0.95) {
    n=n+1
    x=x+prob(n)
}
n
```

これを 1000 回繰り返し（つまりプログムの最後の部分に `for` ループをもうひとつ追加し）て、その 1000 回の平均値をとります。すると、実行結果の値は毎回揺らぎますが、たとえば `18.022` や `18.192` や `18.038` などとなります。すなわち確率 0.95 の判断ではゲームが終わるまでの試合数は 19 回です。言い換えると、試合数が 20 回以上となるゲームの割合は 5% 未満です。さて、この結論は妥当でしょうか。結論を検証してみるには、たとえば `howmany()` を 100 回くらい実行してみればよいでしょう。すると 100 回のうち、20 以上の数値が表示される回数は大体 4 回くらいとなります。どうですか。納得できましたか。

.....

グラフィクスの基礎： グラフィクスは R のセールスポイントのひとつです。簡単なコマンドで見映えのするグラフを作ることができますが、ここでは特に折線または曲線のグラフの作図方法を説明します。一般に R でグラフを描くには次のような手順を踏みます。

1. プロットするデータや数式を用意する。
2. 高水準作図関数でグラフを描く。
3. 低水準作図関数でグラフに装飾を施す。
4. 描いたグラフを保存する。

高水準作図関数で描くことのできるグラフは、散布図、ヒストグラム、箱ひげ図、棒グラフ、円グラフ、分割表の図、スターチャート、対散布図、1 変数関数のグラフ、2 変数関数のグラフ、などです。折線グラフは散布図にオプションを付けることで描くことができます。まず散布図は次のようにして出力します。

```
y=c(1,3,2,4,8,5,7,6,9)
plot(y)
```

これにオプション `type="l"` を付けると折線グラフになります。

```
y=c(1,3,2,4,8,5,7,6,9)
plot(y,type="l")
```

これらのプログラムでは x 座標の値が勝手に `1`, `2`, ..., `length(y)` と解釈され、`length(y)` 個の点列 `c(1,y[1])`, `c(2,y[2])`, ..., `c(length(y),y[length(y)])` がプロットされます。もし x 座標と y 座標をそれぞれ指定したい場合には、次のように書きます。

```
x=runif(10)
y=runif(10)
plot(x,y)
```

複数のグラフを重ね合わせる場合には、コマンド `par` を使います。

```
plot(sin,-pi,pi,col=1)
par(new=T)
plot(cos,-pi,pi,col=2)
```

色のオプション `col` は番号が若いほうから順に、黒、赤、緑、青、水色、紫、黄色、灰、となります。また、複数のグラフを重ね合わせると、グラフごとに x 座標や y 座標の縮尺がまちまちになって、混乱をきたす場合があります。そのような混乱を避けるために x 軸のプロット範囲 `xlim` と y 軸のプロット範囲 `ylim` を明示的に指定する習慣を付けたほうが無難です。たとえば次のようなプログラムです。このプログラムでは、コマンド `plot` の代わりに、1 変数関数のグラフを描くためのコマンド `curve` を使っています。

```
f=function(x,a,b) {b*sin(a*x)}
curve(f(x,1,1),xlim=c(-5,5),ylim=c(-2,2),col=1)
par(new=T)
curve(f(x,2,2),xlim=c(-5,5),ylim=c(-2,2),col=2)
```

このプログラムで、オプション `xlim` または `ylim` をはずして実行してみてください。そうすると縮尺がおかしくなって、グラフを重ねる意味がなくなってしまうのが分かります。
最後に、もし必要ならば、低水準作図関数で凡例 (legend) をつけます。上記のプログラムの最後に

```
legend(3,-1.5,c("a=b=1","a=b=2"),lty=1,col=1:2)
```

を追記しましょう。凡例の左上の座標が `3,-1.5` となります。オプション `lty` は線の種類です。描いたグラフの保存方法については、必要になった時点で、また後日説明することにします。

データ処理実習課題（5 回目・その 1）　　学籍番号の下 2 桁 □□　名前＿＿＿＿＿＿＿＿

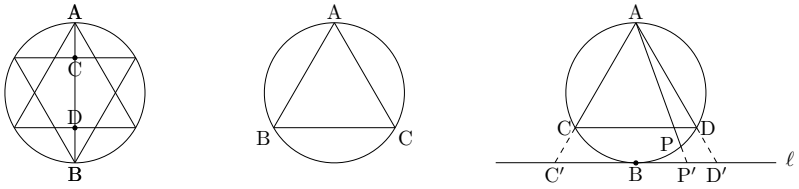
学籍番号の下 2 桁 □□　名前＿＿＿＿＿＿＿＿

1 問題「単位円周に任意に弦をひくとき、その弦の長さが $\sqrt{3}$（＝単位円周の内接正三角形の一辺の長さ）よりも長くなる確率を求めよ」に対して X 君、Y 君、Z 君の 3 人がそれぞれ次のような解答を与えた。

- X 君の解答：　弦は水平にひくとして一般性を失わない。そしてその水平弦が鉛直な直径 AB と交わる点 P の位置によって弦の長さが決まる。図のように点 C と点 D をとると、点 P が線分 CD 上にあるとき条件を満足する。線分 CD の長さは 1 だから、直径 AB 上に任意に点をとったときそれが線分 CD 上にある確率は 1/2 である。したがって求める確率は 1/2 である。

- Y 君の解答：　弦は円周上の 2 点によって決まる。最初の点 A はどこにとっても同じだから第 2 の点 P の相対的な位置によって弦の長さが決まる。点 A を頂点のひとつとする内接正三角形を ABC とすると、点 P が弧 BC 上にあるとき条件を満足する。全円周上に任意に点 P をとるとき、それが弧 BC 上にある確率は 1/3 である。したがって求める確率は 1/3 である。

- Z 君の解答：　円周上の 1 点 A の対蹠点 B で接線 ℓ をひく。弦 AP を延長し、直線 ℓ との交点を P′ とする。弦 AP と ℓ 上の点 P′ は 1 対 1 に対応している。点 A を頂点のひとつとする内接正三角形を ACD とする。ℓ 上に任意に P′ をとったときそれが線分 C′D′ 上にあれば条件を満足するが、直線 ℓ の長さが無限大であるのに対し線分 C′D′ の長さは有限である。したがって求める確率は 0 である。



このように最初の問いに対していろいろの答えがあり、どの答えが正しくどの答えが間違いであるかは判定できない。これはベルトランのパラドクスとして知られている。なお現代的な視点からはこれはパラドクスではなく、確率モデルの違いとして理解されている。ベルトランのパラドクスは、「円周に任意に弦をひく」という表現が実は曖昧であって、確率モデルがひとつに確定できなかったために生じた混乱である。

確率モデルを自由に設定して、モンテカルロ・シミュレーションにより最初の問いに答えよ。ヒント：モデルの一例を挙げる。まず ${x_0}^2 + {y_0}^2 < 1$ かつ $0 \leq \theta_0 < 2\pi$ をみたす実数 x_0, y_0, θ_0 を適当な乱数コマンドを用いて与え、連立方程式

$$\begin{cases} (y - y_0)\cos\theta_0 - (x - x_0)\sin\theta_0 = 0, \\ x^2 + y^2 = 1 \end{cases}$$

の解を $(x_1, y_1), (x_2, y_2)$ とする。これが確率モデルとなる。このとき x_1, x_2, y_1, y_2 は

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 = 4 - 4\left(y_0\cos\theta_0 - x_0\sin\theta_0\right)^2$$

をみたすから、この確率モデルのもとでは $\left(y_0\cos\theta_0 - x_0\sin\theta_0\right)^2 < 1/4$ となる確率をモンテカルロ・シミュレーションで推定すればよいことになる。

データ処理実習課題（5 回目・その 2）　　学籍番号の下 2 桁 □□　名前＿＿＿＿＿＿＿＿

学籍番号の下 2 桁 □□　名前＿＿＿＿＿＿＿＿

2 「チャンスを公平にすると結果は不公平になる (?)」というシミュレーションをしてみよう。次のようなゲームを行う。

1. 6 人のチームを作る。チーム内の各人に 1 番から 6 番まで番号をふる。
2. さいころ 1 つとチップ 30 枚を用意し、テーブルの中央に置く。
3. さいころを振り、n の目が出たら n 番の人が場からチップを 1 枚とる。これを 30 回行い、場にある 30 枚のチップをすべて配る。
4. 次のようなルールでチップをやりとりする。さいころを 2 回振り、1 回目に振ったときに出た目の番号の人は場にチップを 1 枚出す。2 回目に振ったときに出た目の番号の人は場のチップをとる。このチップのやりとり 1 回を 1 セットと呼ぶ。ただし 1 回目に振ったときに出た目の番号の人がチップを持っていない場合は、そのさいころは振りなおすものとする。
5. チップのやりとりを 150 セット繰り返す。

(1) 各人が所持するチップ枚数の変化をシミュレートし、折線グラフにして示せ。ただし 6 つの折線グラフを同時にひとつのグラフ用紙に描くこと。また、シミュレーションの結果を見て気付いたことを述べよ。

(2) セット回数を 1000 回に変えて、同じゲームをシミュレートせよ（プログラムを載せる必要はない）。また、そのシミュレーションの結果を見て、(1) の結果との比較において気付いたことを述べよ。

指定された分布にしたがう乱数の生成方法：　いままでの実習では、コマンド `runif` を用いて一様乱数を発生させ、シミュレーションに利用してきました。しかし多様なシミュレーションをするためには、一様乱数だけでは不十分で、任意に指定された分布にしたがうような乱数が必要となります。そこで今日はまず、一様分布とは限らない一般の分布にしたがって値をとるような乱数を作ってみます。ここで説明する方法（逆関数法）を用いると、所望の分布にしたがうような乱数を簡単に生成することができます。

逆関数法は、区間 $[0,1]$ 上の一様乱数をもとに、所望の分布関数 F にしたがう乱数を発生させる方法です。一様分布にしたがう確率変数を Y としましょう。すなわち任意の $y \in [0,1]$ に対して

$$P(Y \le y) = \int_0^y 1\,dy = y$$

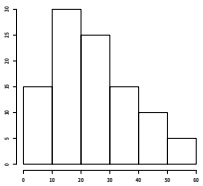
であるとします。このとき $X := F^{-1}(Y)$ は分布 F にしたがう確率変数となります。なぜなら

$$P(X \le x) \;=\; P\left(F^{-1}(Y) \le x\right) \;=\; P(Y \le F(x)) \;=\; F(x)$$

だから。したがって、一様乱数列 y_n を発生させて $x_n := F^{-1}(y_n)$ とおけば、数列 x_n は分布 F にしたがう乱数列となります。

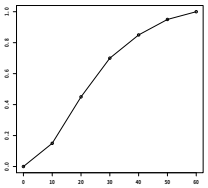
コンビニの混雑を予想するシミュレーション：　逆関数法の応用として、簡単なシミュレーションをしてみましょう。大学前のコンビニが昼休み（60 分）にどれくらい混むか予想するためのシミュレーションです。まず、どんなシミュレーションをするにしても、来客状況を観察し、プログラムの中で再現する作業が必要です。

ある日の昼休みに 10 分刻みで来客数をかぞえて、右のグラフのようになったとします。最初の 10 分間に総来客数の 15%が来店し、次の 10 分に 30%、20–30 分に 25%、30–40 分に 15%、40–50 分に 10%、最後の 10 分間に 5%です。

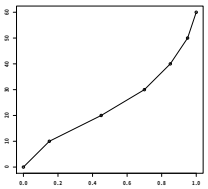


そこでこの割合で来客がある状況を想定します。問題を簡単にするために、それぞれの 10 分間は均一な頻度で来客があるとします。

右のグラフは、この分布の分布関数のグラフです。最初の 10 分間は $(0,0)$ と $(10,0.15)$ を結ぶ線分 $p = 0.015t$ で、次の 10 分間は $(10,0.15)$ と $(20,0.45)$ を結ぶ線分 $p = 0.03t - 0.15$ です。その次の 10 分間は $(20,0.45)$ と $(30,0.70)$ を結んで…、というように求めます。



最後のグラフは、この分布関数の逆関数のグラフです。最初の 10 分間は $t = p/0.015$ で、次の 10 分間は $t = (p+0.15)/0.03$ です。逆関数法で説明したとおり、一様乱数を 100 個発生させてそれらをこの逆関数で写すと、上記の来客分布にしたがう乱数が得られます。



それでは試しに 100 人分の来客時刻をシミュレートしてみましょう。

```
inv=function(p) { # 逆関数
    if (p<0.15) t=p/0.015
    else if (p<0.45) t=(p+0.15)/0.03
    else if (p<0.70) t=(p+0.05)/0.025
    else if (p<0.85) t=(p-0.25)/0.015
    else if (p<0.95) t=100*(p-0.45)
    else t=(p-0.7)/0.005
    return(t)
}
y=runif(100) # 一様乱数を 100 個生成して、
x=sapply(y,inv) # それを逆関数で変換する。
hist(x,breaks=seq(0,60,10))
```

これを実行してください。ここで作った乱数列 `x` が、たしかに始めに指定した来客分布にしたがっていそうなことが確認できます。

以上のことを準備として、レジの待ち状態をシミュレートします。レジに並んでいる人数の変化に注目しましょう。まずはレジが 1 つの場合です。総来客数は 40 とします。さきほどは来客時刻と書きましたが、簡単にするために、これをレジに来る時刻と解釈します。また、レジで客ひとりの対応にかかる時間は 30 秒とします。第 i 番目の客がレジに並んだときに、会計を待っている客の数（自分も含めて）を w_i とし、ベクトル $w = (w_1, \ldots, w_{40})$ を作ります。

```
setwd("/Users/nozomu/tex/fukuoka/2013/R/")
source("convenience-inv.R")
N=40
m=0.5                  # 客一人あたりにかかる時間（分）
x=sapply(sort(runif(N)),inv) # 来客分布に従う乱数列
w=numeric(N)  # w[i] は客 x[i] が来た時の会計待ち人数
w[1]=1
extra=0
for (i in 2:N) {
    # 次の客 x[i] が来るまでのあいだに、会計待ちの客を
    # 何人捌くことができるか計算。
    time=x[i]-x[i-1]+extra
    n=w[i-1]
    while (time>=m & n>0) {
        n=n-1                  # 客をひとり捌くのに
        time=time-m            # m 分かかる。
    }
    w[i]=n+1  # 客 x[i] の来店直前は n 人が並んでいる。
    # もし n>0 ならば、レジ先頭の客が会計の途中だから、
    # その場合には残り時間を次のループに引き継ぐ必要
    # がある。
    extra=ifelse(n>0,time,0)
}
plot(x,w)
```

これを何度も実行してみる（いちばん外側に `for` ループを追加する）と、ピーク時にはだいたい 4 人弱が並ぶことが分かります。運が悪いと 10 人程度並ぶこともあります。

さて、レジが 2 台あるときはどうなるでしょうか。これが今日の実習課題です。もしウェブで参考資料を探すなら、キーワードは「待ち行列」と「シミュレーション」です。

データ処理実習課題（6 回目）

学籍番号の下 2 桁 | | | 名前 _____

学籍番号の下 2 桁 | | | 名前 _____

シミュレーション（その 1・デュースの価値）： 対戦型のスポーツの多くは、小さいゲームの勝ち負けを繰り返して、試合全体の勝敗を決める方式を採用しています。たとえば、テニスの試合は「ポイント」→「ゲーム」→「セット」→「マッチ」の順に進行します。小さい勝ち負けを積み重ねる仕組みは、どのくらい力関係を明瞭にするのでしょうか。シミュレーションで調べてみましょう。次のような設定で考えます。どのポイントにおいてもプレイヤー A がプレイヤー B に勝つ確率は 60%とします。各「ゲーム」は 3,5,7 ポイントマッチ（2,3,4 ポイント先取で勝ち）の 3 種類をシミュレートします。問題を単純化して、「セット」やタイブレーク（「セット」の勝敗を決める特別ルール）は考えません。このとき 3,5,7 ポイントマッチのそれぞれの試合でプレイヤー A が勝つ確率を求めます。

```
p=0.6                          # 各ポイントにおける勝率
N=5000                         # ゲーム数
game=matrix(0,nrow=3,ncol=N) # 各ゲームの勝敗を記録
win=numeric(3)                 # 試合に勝つ確率
for (i in 1:3) {
    pt=2*i+1                   # 3,5,7 ポイントマッチ
    for (j in 1:N) {
        s=sample(c(1,0),pt,c(p,1-p),replace=T)
        game[i,j]=as.numeric(sum(s)>pt/2)
    }
    win[i]=sum(game[i,])/N  # 勝率を計算
}
plot(win,ylim=c(0,1))
```

さて、テニスを始め多くのスポーツにはデュース (deuce) というルールがあります。例えば 7 ポイントマッチでの 3 対 3 のように、同点かつ次のポイントでゲームが決まる状況のとき、そこから 2 ポイント連取したほうがそのゲームを取る、というルールです。上記のプログラムで計算した 3,5,7 ポイントマッチにデュースのルールを加えると、力関係を捉えることにどれだけ貢献するのでしょうか。たとえば次のようなプログラムを書きます。

```
p=0.6
N=5000
game=matrix(0,nrow=3,ncol=N)
win=numeric(3)
for (i in 1:3) {
    pt=2*i+1
    for (j in 1:N) {
        s=①
        while(②) {
            s=③
        }
        game[i,j]=as.numeric(④)
    }
    win[i]=sum(game[i,])/N
}
plot(win,ylim=c(0,1))
```

この実行結果を最初のシミュレーションと比べてください。何が違いますか。詳しくは実習課題にします。

シミュレーション（その 2・バスの団子運転）： 日常的にバスに乗る人は分かると思いますが、バスは電車と違ってなかなか時刻表通りには来てくれません。定刻を過ぎているのにぜんぜん来ないなあ、と思ったら、今度は立て続けに 2 台 3 台と来たりします。なぜバスは団子運転をするのでしょうか。シミュレーションで団子運転を再現して、原因を探ってみましょう。単純化して、次のような設定で考えます。

- バスは始発バス停を 10 分間隔で発車
- 乗客がバス停に来る間隔は 0 ～ 3 分の一様分布
- 乗客一人当たりの乗車にかかる時間は 5 秒
- バス停間の走行時間は 2 分
- バス停の数は 50
- バスの台数は 30

シミュレーションのプログラム例を載せます。

```
buses=30          # バスの台数
stops=50          # バス停の数
interval_orig=10 # 始発バス停における運行間隔（分）
between_stops=2  # 各バス停間の走行時間（分）
geton=5/60        # 乗客 1 人当たりの乗車にかかる時間
delay=5/60      # 前バスに追いついたら delay 待って発車
M=3         # 乗客がバス停に来るのは [0,M] 上の一様分布
time=matrix(0,nrow=buses,ncol=stops) # 出発時刻一覧
# シミュレーション。各バスについて、始発から終点までの
# すべてのバス停の出発時刻を求めて行列 time に代入する
for (bus in 1:buses) {
    # 始発バス停における出発時刻
    departure=①
    time[bus,1]=departure
    # それ以降のバス停における出発時刻
    for (stop in 2:stops) {
        # バス停に着くまで between_stops 時間かかる
        departure=departure+between_stops
        # このバス停で乗客が何人待っているか計算する。
        # その準備として、ひとつ前のバスとの運行間隔
        # interval を計算しておく
        if (bus==1) {
            interval=interval_orig
        } else {
            interval=②
        }
        # 乗客数を計算すれば出発時刻が分かる
        if (interval<=0) {
            # もし前のバスに追いついたら、乗客数は 0 で
            # delay 時間（分）待ってから発車
            departure=③
        } else {
            # 前のバスとの間隔が空いていたら、interval
            # 時間（分）の間にバス停に何人集まるか計算
            timer=0
            passengers=0
            while (④) {
                ⑤
            }
            departure=⑥
        }
        time[bus,stop]=departure
    }
}
for (bus in 1:buses) {
    plot(⑦)
    if (bus<buses) {
        par(new=T)
    }
}
```

来週は休講です。したがって今日が最終回。

データ処理実習課題（7 回目）　　学籍番号の下 2 桁 [ ][ ]　名前＿＿＿＿＿＿＿＿

学籍番号の下 2 桁 [ ][ ]　名前＿＿＿＿＿＿＿＿

1 実習資料の「デュースの価値」について次の問いに答えよ。

(1) プログラムの空欄を埋めよ。

① ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿　② ＿＿＿＿＿＿＿＿

③ ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿　④ ＿＿＿＿＿＿＿＿

(2) シミュレーションの結果から読み取れることを述べよ。

2 実習資料の「バスの団子運転」について次の問いに答えよ。

(1) プログラムの空欄を埋めよ。

① ＿＿＿＿＿＿＿＿＿＿＿＿

② ＿＿＿＿＿＿＿＿＿＿＿＿

③ ＿＿＿＿＿＿＿＿＿＿＿＿

④ ＿＿＿＿＿＿＿＿＿＿＿＿

⑥ ＿＿＿＿＿＿＿＿＿＿＿＿

⑤

⑦ ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

(2) シミュレーションの結果から読み取れることを述べよ。

データ処理実習（2014 年 12 月 17 日）　　学籍番号の下 2 桁 [　][　]　名前＿＿＿＿＿＿＿＿＿＿

1 ベクトル v と行列 M を

$$v = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$$

$$M = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

とする。下記の空欄に適当なコマンドを記せ。ただし答えは一意に決まらないから、空欄の長さに応じて最適と思われるものをひとつ解答せよ。（5 点 ×9）

– ベクトル v を作るには `v=`[　]と入力する。行列 M はベクトル `v` を用いて `M=`[　　]と作成できる。

– ベクトル `v` の成分をランダムに並べ替えるには[　　]と入力する。また、重複を許して、ベクトル `v` の成分をランダムに 10 個取り出すには[　　]と入力する。

– 数列 a_n を $a_n = 3n + 2$ と定める。始めの 10 項を並べたベクトル $x = (a_0, \ldots, a_9)$ はベクトル `v` を用いて `x=`[　　]と作成できる。同様に数列 $b_n = n^2$ に対してベクトル $y = (b_0, \ldots, b_9)$ は `v` を用いて `y=`[　]と作成できる。より一般に、任意の函数 f と任意の実数 $x_0, \ldots, x_{n-1}$ に対して、ベクトル $(f(x_0), \ldots, f(x_{n-1}))$ を作るには、コマンド `sapply` を用いるとよい。たとえば函数 f を

```
f=function(n) {
    if (n==0)
        return(0)
    else
        rerutn(n*f(n-1))
}
```

と定めたとき、函数 f の 10 個の値を並べたベクトル $z = (f(0), f(1), \ldots, f(9))$ はベクトル `v` も利用して `z=`[　　]と作成できる。もしもコマンド `sapply` を使いたくなければ、代わりに `for` 文を用いて次のように書くこともできる。

```
z=numeric(10)
for ([　　]) {
    [　　　　]
}
```

ただし R 言語では `for` 文を用いるとプログラムの処理が遅くなるので、高次元のベクトルを扱う際にはできるだけベクトルのまま処理することを心がけ、`for` 文を避けたコードを書くことが望ましい。

2 楕円 $x^2 + 4y^2 = 1$ の面積をモンテカルロ法によって求めたい。そのためのプログラムを書け。（15 点）

3 ある高校のクラスで席替えを行うことになった。机は床に固定されており、各机には 1 から 40 までの番号がついている。席替えは籤引きで行い、籤の番号と同じ番号の机に移動するものとする。このとき「なんや、前と同じ席が当たった」となる生徒はどれくらいいるか。席が変わらない確率をシミュレーションで推定せよ。（20 点）