Javaプログラミングの Tip集

第13回

2008年3月12日
アイティエス

数値計算を考える

数値計算は関数を近似的に求めるときに使う

今回取り上げるやり方(有名)

①Newton法
②Euler法
③Runge・Kutta法

Javaで作成して『R』でグラフ化する

<前回の応用>
④待ち行列シミュレーション

Newton法

●ニュートン法(Newton's method)
非線形方程式(non-linear equation)
$f(x) = 0$ **・・・数値計算で近似解を求める**

➢ Tayler展開の特別ケースとして捉える

$f(x_k+h) = f(x_k) + f'(x_k) * h/1! + f''(x_k) * h^2/2! + \cdots$

$f(x_k+h) = f(x_k) + f'(x_k) * h/1!$ **・・・以降を無視**

$x_{k+1} = x_k - f(x_k) / f'(x_k)$ ⇐ 採用

➢ 第1近似　　第2近似

x_1　　$x_2 = x_1 - f(x_1) / f'(x_1)$

$\sqrt{2}$ を求める(ニュートン法)

```
>java newtonTest 1.0 2
```
初期値　ルート値

ニュートン法 Yk+1 = Yk - f(Yk)/f'(Yk), f = sqrt(y) 反復回数=5
No1 反復=1　データy=1.5
No2 反復=2　データy=1.4166666666666665
No3 反復=3　データy=1.4142156862745097
No4 反復=4　データy=1.4142135623746899
No5 反復=5　データy=1.414213562373095 ⇐ 取得

Euler法

●オイラー法(Euler's method)
微分方程式(differential equation)

$dy/dx = f(x,y)$　　初期条件 $y(x_0) = y_0$

$y = y_0 + f(x_0, y_0) * (x - x_0)$

$y_1 = y_0 + f(x_0, y_0) * (x_1 - x_0)$　　条件 $x_1 = x_0 + h$

$= y_0 + f(x_0, y_0) * h$

$y_{k+1} = y_k + f(x_k, y_k) * h$ ⇐ 採用

➢微分方程式の解は、初期条件を定めて特殊解を求めることになる
➢精度を得るには刻み幅hを小さくするが、計算量大と誤差に注意が必要になる

---

eulerメソッド

```
double[][] euler(double x,double y,double h,int upper)
        ⋮
 for(i=1;i<=upper;i++){
        x=xdt[i-1];
        y=ydt[i-1];
        k1 = h*func(x,y);
        ydt[i] = y + k1;
        xdt[i] = xdt[0]  +h*(double)i;
        eudata[i][0] = xdt[i];
        eudata[i][1] = ydt[i];
 }
        ⋮
```

```
public double func(double x,double y){
    double f;
    f = y/(2*x);
    return f;
}
```

√x を求める(オイラー法)

```
>java eulerTest 1.0 1.0 0.1 5
```

x0初期値　y0初期値　h刻み幅　num個数指定

```
オイラー法 f' = y/(2*x), y = sqrt(x) 計算データ数:40
No1 データx=1.0  データy=1.0
No2 データx=1.1  データy=1.05
No3 データx=1.2  データy=1.0977272727272727
No4 データx=1.3  データy=1.143465909090909
No5 データx=1.4  データy=1.187445367132867
No6 データx=1.5  データy=1.2298541302447552
No7 データx=1.6  データy=1.2708492679195804
No8 データx=1.7000000000000002  データy=1.3105633075420673
No9 データx=1.8  データy=1.3491092871756576
No10 データx=1.9  データy=1.3865845451527592
No11 データx=2.0  データy=1.4230736121304635
No12 データx=2.1  データy=1.4586504524337252
```

```
upper = (int)((num-x0)/h);
```

Runge･Kutta法

●ルンゲ･クッタ法(**Runge･Kutta's method**)

微分方程式(differential equation)

$dy/dx = f(x,y)$　　初期条件(x_0, y_0)

$k_1 = f(x_i, y_i) * h$

$k_2 = f(x_i+h/2, y_i+ k_1/2) * h$

$k_3 = f(x_i+h/2, y_i+ k_2/2) * h$

$k_4 = f(x_i+h, y_i+ k_3) * h$

採用

$y_{i+1} = y_i + (k_i+2k_2+2k_3+k_4)/6$　条件$x_{i+1} = x_i + h$

x値の刻み幅をh

i番目のx_i とy_iから

x_{i+1}、y_{i+1}を求めていく

➢Runge･Kutta法はEuler法より解が速く得られ、精度もよい

rungeKuttaメソッド

```
double[][] rungeKutta(double x,double y,double h,int upper)

for(i=1;i<=upper;i++){
      x=xdt[i-1];
      y=ydt[i-1];
      k1 = h*func(x,y);
      k2 = h*func(x + hh,y + k1/2.0);
      k3 = h*func(x + hh,y + k2/2.0);
      k4 = h*func(x + h,y + k3);
      ydt[i] = y + (k1 + 2.0*(k2 + k3) + k4)/6.0;
      xdt[i] = xdt[0]  +h*(double)i;
      rkdata[i][0] = xdt[i];//データx
      rkdata[i][1] = ydt[i];//データy
}
```

```
upper = (int)((num-x0)/h);
```

```
public double func(double x,double y){
      double f;
      f = y/(2*x);
      return f;
}
```

√x を求める(ルンゲ･クッタ法)

```
>java rukuTest 1.0 1.0 0.1 5
```

x0初期値　y0初期値　h刻み幅　num個数指定

```
ルンゲ･クッタ法 f' = y/(2*x), y=sqrt(x) 計算データ数:40
No1 データx=1.0  データy=1.0
      真値No1 データy=1.0
No2 データx=1.1  データy=1.048808879612451
      真値No2 データy=1.0488088481701516
No3 データx=1.2  データy=1.095445168721679
      真値No3 データy=1.0954451150103321
No4 データx=1.3  データy=1.1401754953402603
      真値No4 データy=1.140175425099138
No5 データx=1.4  データy=1.1832160396499987
      真値No5 データy=1.1832159566199232
No6 データx=1.5  データy=1.2247449646841137
      真値No6 データy=1.2247448713915889
No7 データx=1.6  データy=1.2649111658616894
      真値No7 データy=1.2649110640673518
No8 データx=1.7000000000000002  データy=1.3038405900747851
      真値No8 データy=1.3038404810405297
No9 データx=1.8  データy=1.3416409018464786
      真値No9 データy=1.3416407864998738
No10 データx=1.9  データy=1.3784049961702192
      真値No10 データy=1.378404875209022
No11 データx=2.0  データy=1.414213688412753
      真値No11 データy=1.4142135623730951
```

drawRoot.R　ルート近似解描画

```
plot.new()
dt <- getData()
xmax <- length(dt$rcx)
#print(xmax)
plot(dt$rcx, dt$rcy,col = "black",pch=20,xlab="値:x",ylab="解:y")
points(dt$rtx, dt$rty, pch=21, col="red")
points(dt$eux, dt$euy, pch=22,col = "blue")
note <- c("近似値=","理論値=","オイラー近似値=")
color <- c("black","red","blue")
legend(1.2,2.3,note,col=color,pch=20:22,bty="n")
for(i in 1:xmax){
  if((dt$rcx[i] == 2) | (dt$rcx[i] == 3) | (dt$rcx[i] == 4)){
    rect(0,0,dt$rtx[i],dt$rty[i],lty=3)
    text(dt$rcx[i]+0.4,dt$rty[i],dt$rcy[i])
  }
}
text(2,2.25,expression(paste(sqrt(x))))
text(2,2.2,expression(paste(sqrt(x))))
text(2.6,2.15,expression(paste(sqrt(x))))
title("平方根近似解")
}
```

●関数plot()

データをプロットした後は、関数points()を使う

●関数legend()

見本例を表示する

ルート近似解のRコマンド

```
> source("C:¥¥Program Files¥¥R¥¥R-2.6.1¥¥drawRoot.R")
> drawRoot("rukucalcudat.txt","rukutheorydat.txt","eulercalcudat.txt")
Read 40 records
Read 40 records
Read 40 records
>
```

rukucalcudat.txt用データリード

rukutheorydat.txt用データリード

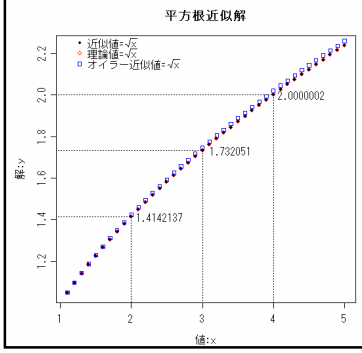
eulercalcudat.txt用データリード

●Rコマンドでのプロットとの Javaのデータ処理を連携させると非常にスピーディに描画できる

簡単に確かめるにはRコマンドだけで処理を試みる

Rコマンドを使うといろいろな統計データを処理して分析に活用できる

$\sqrt{x}$ の近似解をプロット



近似値・・・・**ルンゲ・クッタ法**
理論値・・・・**Javaのルート関数**
その他・・・・**オイラー法**

実際はRだけの関数で簡単にルートの近似解を導くことができる

待ち行列シミュレーション

- Javaでデータを作成してRで評価する
  - 前回のRandTest.javaを改造して一様分布の間隔で電話を掛けに人が来て、一様分布の間隔で通話する

```
> java simTest 0 45 100 0 20 100 240
```

L:0～45分　R　U:0～20分　R　T:240分

L:下限値
U:上限値
R:乱数発生個数
T:シミュレーション時間

シミュレーション

Rに引き渡すためのシミュレーションデータをテキストファイル形式で作成する

①～③のファイルフォーマット

データ数	データCRデータCR・・・・

CR:改行

①arrivaldat.txt ・・・到着時刻データ
②servicedat.txt ・・・通話時間データ
③queuedat.txt ・・・・待ち行列データ
④calcudat.txt ・・・・・統計データ

simTest.java

- 待ち行列には通常ポアソン分布を使うが、今回は一様分布で公衆電話の利用度をシミュレーションする

人の到着時間間隔(乱数Aで作成) { x_0 = 50249347 ・・・Seed, k = 23
通話時間(乱数Bで作成) { x_0 = 513 ・・・Seed, k = 2045

```
//統計データ計算
  nf.setMaximumFractionDigits(2);
  awt = nf.format((double)quesum/person); //平均待ち時間
  dutil = (double)boxsum/tmax;
  util = nf.format(dutil*100);                //利用率
  atu = nf.format((double)boxsum/person);
  acon = nf.format((double)quesum/tmax); //平均待ち行列長さ
  calcResult[0] = String.valueOf(quelen);   //最大待ち行列長
  calcResult[1] = awt;                      //平均待ち時間
  calcResult[2] = String.valueOf(person);   //全利用者数
  calcResult[3] = util;                     //利用率
  calcResult[4] = atu;                      //平均通話時間
```

クラス図モドキ的表現

- 本体のsimTestクラスとキューを管理するfifoクラスで作成する

作成クラスの属性、コンストラクタは省略して表現する

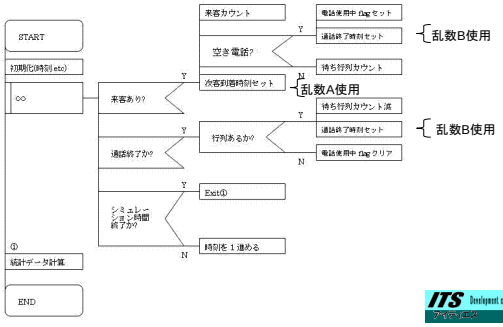
class simTest
略
int[] randProc(String h1,String h2,String h3,int sw)
int lehmer(int x,int k)
void queProc(int[] dataA,int[] dataB,int tmax)
int getMemo(String dt,int item,int talkend)
void makeFile(int[] arrivalTime,int[] serviceEnd,int[] queueStart,String[] calcResult,int tmax)
void main(String[] args)

← シミュレーション処理本体

class fifo
略
void put(int pnt,String info)
String read()
int memoCount()
void entry(int que,int tim)
int del()
boolean isEmpty()
int getQuecnt()
int getSize()

randProc本体のPAD

- シミュレーションの時刻管理に乱数を用いる



Rコマンドプログラミング

- ユーザ定義の関数functionでプログラミングする

```
<使い方>
drawRect <- function(nam1,nam2,nam3,nam4){
  BODY(Rコマンドを記述する範囲)
}
```

関数名 (命名は自由)
この名称を使う
カッコ始まり
引数(複数可、無しも可)
カッコ閉じ

drawRect① 関数の中に関数使用

```
drawRect <- function(nam1,nam2,nam3,nam4){
 getData <- function(){
   fname1 <- paste("C:¥¥Program Files¥¥R¥¥R-2.6.1¥¥",nam1,sep="")
   fname2 <- paste("C:¥¥Program Files¥¥R¥¥R-2.6.1¥¥",nam2,sep="")
   fname3 <- paste("C:¥¥Program Files¥¥R¥¥R-2.6.1¥¥",nam3,sep="")
   x1 <- scan(fname1)
   x2 <- scan(fname2)
   x3 <- scan(fname3)
   dat <- data.frame(x1,x2,x3)
   return (dat)
 }
 rsltData <- function(){
   fname4 <- paste("C:¥¥Program Files¥¥R¥¥R-2.6.1¥¥",nam4,sep="")
   x4 <- scan(fname4)
   rdat <- data.frame(x4)
   return (rdat)
 }
```

Javaで作成したデータを読み込んでベクトルデータにする
- ●ユーザ定義関数getData ・・・数値データ
- ●ユーザ定義関数rsltData ・・・文字列データ

●関数paste()
sepで空白を除去して各文字列の連結を行う

②

drawRect② プロット準備

- 関数plot()で時系列プロットを準備する

```
plot.new()
dt <- getData()
rdt <- rsltData()
tmax <- dt$x1[1]
#print(tmax)
qmax <- as.integer(rdt$x4[1])
ymax <- qmax + 3
plot(c(0,tmax),c(0,ymax),axes=FALSE,type="n",xlab="分:min",ylab="人数:num")
y <- 0
indx <- 0
```

変数使用例 dt<-getData()
- ●変数dtへgetData()の数値データを読み込む
- ●変数rdtへrsltData()の文字列データを読み込む

●関数as.integer()
文字を数値化する

●関数plot ()
plot.new()でグラフィック表示画面をクリアする
axes=FALSEで座標軸表示を抑制する
type='n'以下でプロット動作をせずに軸の名称のみを描く

③

drawRect③ 矩形描画

```
for(i in 2:tmax){
  if(dt$x1[i] == 0){
    next
  }
  if(dt$x1[i] == dt$x2[i]){
    rect(dt$x1[i], 0, dt$x2[i]+0.1, 1, col = "blue")
  }else{
    if(dt$x3[i] == 0){
      y <- 0
    }else{
      if(dt$x3[i] == qmax){
        indx <- i
        y <- dt$x3[i]
      }else{
        if(i > indx){
          y <- y + dt$x3[i]
        }else{
          y <- dt$x3[i]
        }
      }
    }
    rect(dt$x1[i], y, dt$x2[i], 1+y, col = "blue")
  }
}
```

●関数for(var in m1:m2)
パラメータの範囲で処理を繰り返す
varは変数、m1は初期値パラメータ、m2は終値パラメータ

●関数if(c) s
条件を判定して処理の分岐を実現する
c(condition)は条件式、s(statement)は処理

●関数rect(x0,y0,x1,y1,・・・)
始点座標(x0,y0)、終点座標(x1,y1)を指定して矩形を描く
colで色の指定(番号または英単語)ができる

drawRect④ 処理データ配置

```
title("公衆電話利用シミュレーション")
xdata1 <- rdt$x4[1]
xdata2 <- rdt$x4[2]
xdata3 <- rdt$x4[3]
xdata4 <- rdt$x4[4]
xdata5 <- rdt$x4[5]
item1 <- paste("最大行列長:      ",xdata1,"人",sep="")
item <- c(item1,item2,item3,item4,item5)
for(k in 1:5){
   text(50, 5-(k-1)*0.3, labels= item[k], col=k)
}
axis(side=1,pos=0,at=0:tmax,col="black")
axis(side=2,pos=0,at=0:ymax,col="black",las=0)
}
```

●関数title("タイトル用文字列")
グラフ描画の外枠にタイトルを表示する

●関数text(x,y,・・・・)
指定座標(x,y)に文字列を表示する
パラメータを使うと文字の色、文字の回転角度の指定もできる

●関数axis()
描く座標軸を指定する
pos=0で原点を通る座標軸、 atパラメータで目盛を指定する

公衆電話利用のRコマンド

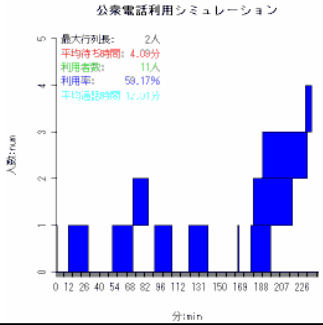
```
> source("C:¥¥Program Files¥¥R¥¥R-2.6.1¥¥drawRect.R")
> drawRect("arrivaldat.txt","servicedat.txt","queuedat.txt","calcudat.txt")
Read 241 items
Read 241 items
Read 241 items
Read 5 items
>
```

arrivaldat.txt用データリード
servicedat.txt用データリード
queuedat.txt用データリード
calcudat.txt用データリード

●Rコマンドのファイル処理
大量データを読み込むためにはファイルを活用する
メニューから「Rコードのソースを読み込み」を指定する

シミュレーションプロット

- 2種類の乱数を用い公衆電話利用度を調べる



<今回>
①240分で11人が利用して待ち
行列数は2人と判る
➢利用率は59%程度
➢平均通話時間は12分程度

②一様分布の乱数で求める