微分方程式

モデリングとシミュレーション

2019年度

質点の運動のモデル化

- 粒子と粒子に働く力
- 粒子の運動→粒子の位置の時間変化
  - 粒子の位置の変化の割合→速度
  - 速度の変化の割合→加速度
- 力と加速度の結び付け
  - Newtonの運動方程式：微分方程式
  - 解は、時間の関数としての位置

Newtonの運動方程式

- 質点の運動は、Newtonの運動方程式で記述される
  - 加速度は力に比例する
  - 位置の二階微分方程式

$$m\frac{\mathrm{d}^2\vec{x}}{\mathrm{d}t^2}=\vec{F}$$

微分

- 独立変数tとその従属変数$x(t)$
- 一階微分

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \lim_{h \to 0} \frac{x(t+h) - x(t)}{h}$$

  - 変数tが微小に増加した際の、xの増分の割合：一定とは限らない
  - 関数$x(t)$の時間に関する変化率
  - (t, x)空間内の関数$x(t)$の傾き

- 二階微分

$$v(t)=\frac{\mathrm{d}x}{\mathrm{d}t}$$

$$\frac{\mathrm{d}v}{\mathrm{d}t}=\lim_{h\to 0}\frac{v(t+h)-v(t)}{h}$$

- 関数$x(t)$の変化率$v(t)$の変化
- 関数$x(t)$の曲り具合（上に凸、下に凸など）

微分方程式

- 関数の変化を記述することで、それに従う関数を求める
- 例1：変化の割合が一定

$$\frac{\mathrm{d}x}{\mathrm{d}t} = a \quad \Rightarrow \quad x(t) = at + b$$

- 例2：変化の割合が関数自身に依存

$$\frac{\mathrm{d}x}{\mathrm{d}t} = ax \quad \Rightarrow \quad x(t) = b\exp\left(at\right)$$

微分方程式で定まらない定数（積分定数）bが現れる

二階微分方程式

- 独立変数による二階微分を含む方程式
- 例：

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = -\omega^2 x \quad \Rightarrow \quad x(t) = a\cos(\omega t) + b\sin(\omega t)$$

微分方程式で定まらない定数（積分定数）
aとbが現れる

二階微分方程式を一階連立微分方程式へ

- $v(t) = \mathrm{d}x(t)/\mathrm{d}t$を導入

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = -\omega^2 x \quad \Rightarrow \quad \frac{\mathrm{d}v}{\mathrm{d}t} = -\omega^2 x, \quad \frac{\mathrm{d}x}{\mathrm{d}t} = v$$

位置xの時間変化はvで定まる

数値積分の観点で微分方程式を見ると

- 微分方程式を解くことを「積分」と呼ぶ
- 微分方程式は、「左辺(導関数)が右辺で与えられる」と読む
  - 導関数→微小変化量→直後の関数の変化値

$$\frac{\mathrm{d}\vec{y}}{\mathrm{d}t} = \vec{f}\left(t, \vec{y}\right)$$

微分方程式を数値的に解くということ

- 微分方程式 $\dfrac{\mathrm{d}x}{\mathrm{d}t}=f\left(x,t\right)$

$$x\left(t+\Delta t\right)=x(t)+\frac{\mathrm{d}x}{\mathrm{d}t}\Delta t+O\left(\left(\Delta t\right)^{2}\right)$$
$$=x(t)+f\left(x,t\right)\Delta t+O\left(\left(\Delta t\right)^{2}\right)$$

- あるtでの$x(t)$と$f(x,t)$が分かれば、$x(x+\Delta t)$の値を近似的$O((\Delta t)^2)$に得ることができる

Euler法

- 最も簡単な微分方程式の数値解法
- 一元一階微分方程式の場合

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f\left(t, y\right)$$

- 近似的な時間発展式

$$y\left(t+h\right) \simeq y\left(t\right) + h \times f\left(t, y\left(t\right)\right)$$

Euler法

- n元一階微分方程式の場合

$$\frac{\mathrm{d}y_i}{\mathrm{d}t} = f_i\left(t, \vec{y}\right)$$

- 近似的な時間発展式

$$y_i\left(t+h\right) \simeq y_i\left(t\right) + h \times f_i\left(t, \vec{y}\left(t\right)\right)$$

  - 右辺は時刻tにおける量だけで表現されている

Javaを用いた微分方程式の数値解法

- 数値解法：Euler法またはRunge-Kutta法
  - 微分方程式に依存しない一般的手法
- 微分方程式をインターフェイスのインスタンスとして渡す
  - C/C++の関数ポインタに相当

微分方程式を表すインターフェイス

- 独立変数tと従属変数$y_i(t)$

関数インターフェイスであること示す注釈

```
@FunctionalInterface
public interface DifferentialEquation {
   public double[] rhs(double t, double y[]);
}
```

- 引数の値から、導関数の値を返す関数

Euler法によって数値積分を行うクラス

- 微分方程式を引数で渡す
  - hだけ独立変数を進める

独立変数tと従属変数$\vec{y}$

```
public class Euler {
    public static double[] euler(double t,double y[],
        double h, DifferentialEquation eq){
            int n = y.length;
            double yt[]=new double[n];
            double dy[] = eq.rhs(t, y);
            for(int i=0; i<n ; i++){
                yt[i] = y[i] + h * dy[i];
            }
            return yt;
    }
}
```

微分方程式
独立変数tと従属変数$\vec{y}$
から$\mathrm{d}\vec{y}/\mathrm{d}t$を求める
$\vec{f}(t,\vec{y})$に相当

$$y_i(t+h) = y_i(t) + h \times f_i\left(t, \vec{y}(t)\right)$$

Javaで微分方程式を定義する

- DifferentialEquationはInterface
  - インスタンスを生成できない
- 二つの方法
  - 生成時に抽象メソッドを定義する
  - ラムダ式を利用する

関数の定義：一様重力の例 抽象クラスの実装を使って

```
Int n=2;
DifferentialEquation eq
    = new DifferentialEquation(){
    //メソッドの実装
    public double[] rhs(double t, double[] y){
        double dy = new double[n];
        dy[0] = y[1];//dx/dt
        dy[1] = g; // dv/dt
        return dy;
    }
};
```

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v$$

$$\frac{\mathrm{d}v}{\mathrm{d}t} = g$$

関数の定義：一様重力の例 ラムダ式を使うと

```
Int n=2;
DifferentialEquation eq
   =  (double t, double y[]) -> {
      double dy = new double[n];
      dy[0] = y[1];//dx/dt
      dy[1] = g; // dv/dt
      return dy;
   };
```

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v$$

$$\frac{\mathrm{d}v}{\mathrm{d}t} = g$$

各微分方程式をどのインデクスに割り当てるかの定めはない

Javaにおけるλ式

- 関数を表すインターフェースの実装を簡潔に表す方法
  - 関数そのもの（後で引数に値が入り評価される）を表現する
  - 関数を変数として扱える

    (引数並び) -> {関数の実体};
- Java8以降で利用可能になった
  - リストなどの要素の処理でも利用

```
import java.util.function. DoubleFunction;

public class LambdaMain {

   public static double generalSum(
         double data[], DoubleFunction<Double> op){
      double sum=0.;
      for(double d:data){
         sum += op.apply(d);
      }
      return sum;
   }

   public static void main(String[] args) {
      double data[]={1.,5.,8.,11.};
      //二乗の和を計算
      double sum = generalSum(data, d->d*d);
      System.out.println(sum);
   }
}
```

データに対する具体的演算は未定義

データに対する二乗を定義

基本的な関数の定義例 java.util.functionに定義済み

- DoubleFunction<R>
  - double型の一変数に対してR型の値を返す
- UnaryOperator<T>
  - T型の一変数に対してT型の値を返す
- Function<T,R>
  - T型の一変数に対してR型の値を返す
- Predicate<T>
  - T型の一変数に対してboolean型を返す

拡張されたfor ループ

```
T array[];
// array の各要素に対して処理
for (T t : array ){
}

List<T> list;
for ( T t : list ){
}
```

ラムダ式の活用
Listなどの処理

```
List<Integer> list;
// 要素を印刷
 list.stream().forEachOrdered(
     x -> {
          int i = list.indexOf(x);
          System.out.println(i + ":" + x);
     }
);
```

```
List<Integer> list;
// 要素の和
int sum = list.stream().reduce(0,
                        (acc,item)->acc+item);
// 要素の最大値
int max = list.stream().reduce(list.get(0),
                        (acc,item)->Math.max(acc,item));
// 正の要素の数
long count = list.stream().filter(x -> (x > 0)).count();
```