



微分方程式

モデリングとシミュレーション

2020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

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

微分

➡ 独立変数 t とその従属変数 $x(t)$

➡ 一階微分
$$\frac{dx}{dt} = \lim_{h \rightarrow 0} \frac{x(t+h) - x(t)}{h}$$

➡ 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

$$v(t) = \frac{dx}{dt}$$

$$\frac{dv}{dt} = \lim_{h \rightarrow 0} \frac{v(t+h) - v(t)}{h}$$

- ➡ 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

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

- ➡ 例2：変化の割合が関数自身に依存

$$\frac{dx}{dt} = ax \quad \Rightarrow \quad x(t) = b \exp(at)$$

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：

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

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

➡ $v(t) = dx(t)/dt$ を導入

$$\frac{d^2x}{dt^2} = -\omega^2 x$$



$$\frac{dv}{dt} = -\omega^2 x$$

$$\frac{dx}{dt} = v$$

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

$$\frac{d\vec{y}}{dt} = \vec{f}(t, \vec{y})$$

微分方程式を数値的に解くということ

➡ 微分方程式 $\frac{dx}{dt} = f(t, x)$

$$\begin{aligned}x(t + \Delta t) &= x(t) + \frac{dx}{dt} \Delta t + O((\Delta t)^2) \\ &= x(t) + f(t, x) \Delta t + O((\Delta t)^2)\end{aligned}$$

- ➡ ある t での $x(t)$ と $f(t, x)$ が分かれば、 $x(x + \Delta t)$ の値を近似的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

$$\frac{dy}{dt} = f(t, y)$$

- ▶ 近似的な時間発展式

$$y(t+h) \approx y(t) + h \times f(t, y(t))$$

Euler法

- ➡ n 元一階微分方程式の場合

$$\frac{dy_i}{dt} = f_i(t, \vec{y})$$

- ➡ 近似的な時間発展式

$$y_i(t+h) \approx y_i(t) + h \times f_i(t, \vec{y}(t))$$

- ➡ 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
であることを示す注釈

```
@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

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

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = g$$

```
1. Int n=2;
2. DifferentialEquation eq
3.   = new DifferentialEquation(){
4.     //メソッドの実装
5.     public double[] rhs(double t, double[] y){
6.         double dy = new double[n];
7.         dy[0] = y[1]; //dx/dt
8.         dy[1] = g; // dv/dt
9.         return dy;
10.    }
11.   };
```

関数の定義：一様重力の例 ラムダ式を使うと

$$\frac{dx}{dt} = v$$
$$\frac{dv}{dt} = g$$

```
1. Int n=2;
2. DifferentialEquation eq
3.   = (double t, double y[]) -> {
4.     double dy = new double[n];
5.     dy[0] = y[1]; // dx/dt
6.     dy[1] = g; // dv/dt
7.     return dy;
8.   };
```

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

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

```

1. public class Euler {
2.     public static double[] euler(double t, double y[],
3.     double h, DifferentialEquation eq){
4.         int n = y.length;
5.         double yt[] = new double[n];
6.         double dy[] = eq.rhs(t, y);
7.         for(int i=0; i<n; i++){
8.             yt[i] = y[i] + h * dy[i];
9.         }
10.        return yt;
11.    }
12. }

```

独立変数 t と従属変数 \vec{y}

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

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

Javaにおけるラムダ式

- ➡ 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

```
1. import java.util.function. DoubleFunction;
2. public class LambdaMain {
3. public static double generalSum(
4. double data[], DoubleFunction<Double> op){
5. double sum=0.;
6. for(double d:data){
7. sum += op.apply(d);
8. }
9. return sum;
10. }
11. public static void main(String[] args) {
12. double data[]={1.,5.,8.,11.};
13. //二乗の和を計算
14. double sum = generalSum(data, d->d*d);
15. System.out.println(sum);
16. }
17. }
```

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