MS211 - Cálculo Numérico

Turma Z - Prof João Batista Florindo

Projeto 2

Flávio Murilo Reginato RA 197088
Júlio Moreira Blás de Barros RA 200491

Introdução

Nesse projeto foi analisado o uso dos métodos numéricos de Euler e Runge-Kutta de quarta ordem para encontrar valores numéricos para uma determinada população utilizando a equação de crescimento populacional de Verhulst:

$$y'(t) = r\,y(t)\left(1 - \frac{y(t)}{K}\right),$$

Na qual y é a população total, r é uma taxa base e K uma constante que representa a capacidade do meio em questão.
Para essa análise, foi utilizada como controle a função analítica presente na literatura:

$$y(t) = \frac{K\,y_0\,e^{r\,t}}{K + y_0\,(e^{r\,t} - 1)},$$

Metodologia

Foi criado um programa em Java (JDK 11.0.5) capaz de calcular valores para a função da população de forma padronizada, utilizando as classes de dados *Point* (ponto no plano cartesiano) e *Interval* (intervalo real fechado)

```java
class Interval {
    private static final double EPSILON = 0.000001;
    private final double start;
    private final double end;

    Interval(double start, double end) {
        this.start = start;
        this.end = end;
    }

    public double getStart() { return start; }

    public double getEnd() { return end; }

    List<Double> range(double step) {
        if (step == 0) { return Collections.emptyList(); }

        final int maxSize =
            (int) Math.round(Math.ceil( (end-start)/step ));
        final var range = new ArrayList<Double>(maxSize);

        for (double i = start; (i - end) <= EPSILON; i += step) {
            range.add(MathExtension.round(i));
        }

        return range;
    }
}
```

```java
class Point {
    private final double x;
    private final double y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public String toString() {
        return "("+x+", "+y+")";
    }
}
```

Foram criadas interfaces para a derivada (*DoubleParamDerivative*) e para a solução analítica (*SingleParamFunction*), sendo as implementações delas as respectivas equações:

```java
class VerhulstLogisticDerivative implements DoubleParamDerivative {

    private final double r;
    private final double K;

    VerhulstLogisticDerivative(double r, double k) {
        this.r = r;
        this.K = k;
    }

    @Override
    public double evaluate(double x, double y) {
        return r * y * (1 - (y/K));
    }
}

class VerhulstLogisticSolution implements SingleParamFunction {

    private final double r;
    private final double K;
    private final double y_0;

    VerhulstLogisticSolution(double r, double k, double y_0) {
        this.r = r;
        this.K = k;
        this.y_0 = y_0;
    }

    @Override
    public double evaluate(double x) {
        var exp = Math.exp(r * x);
        return (K * y_0 * exp) / (K + y_0*(exp - 1));
    }
}
```

Por fim, foram implementadas 3 funções para aplicação dos métodos (Euler, Runge-Kutta e analítico)

```java
static List<Point> eulerSolve(
    DoubleParamDerivative function, Interval interval,
    /* h: */double step, double y_0
) {
    final var range = interval.range(step);
    var lastValue = y_0;
    final var newList = new ArrayList<Point>(range.size());

    for (double x : range) {
        newList.add(new Point(x, MathExtension.round(lastValue)));
            // the first value will be the (x_0, y_0)

        lastValue = lastValue + step * function.evaluate(x, lastValue);
    }

    return newList;
}
```

```java
static List<Point> rungeKuttaSolve(
    DoubleParamDerivative function, Interval interval,
    /* h: */double step, double y_0
) {
    final var range = interval.range(step);
    var lastValue = y_0;
    final var newList = new ArrayList<Point>(range.size());

    for (double x : range) {
        newList.add(new Point(x, MathExtension.round(lastValue)));
        // the first value will be the (x_0, y_0)

        final var k1 = function.evaluate(x, lastValue);
        final var k2 = function.evaluate( x: x + step/2,  y: lastValue + k1 * step/2);
        final var k3 = function.evaluate( x: x + step/2,  y: lastValue + k2 * step/2);
        final var k4 = function.evaluate( x: x + step,  y: lastValue + k3 * step);
        final var k = (1.0/6.0) * (k1 + 2*k2 + 2*k3 + k4);

        lastValue = lastValue + step * k;
    }

    return newList;
}
```

```java
static List<Point> analyticSolve(
    SingleParamFunction function, Interval interval,/* h: */double step
) {
    final var range = interval.range(step);

    return range.stream()
        .map(x -> new Point(x, MathExtension.round(function.evaluate(x))))
        .collect(Collectors.toList());
}
```

Resultados Obtidos

A partir das implementações, foi efetuado o cálculo dos valores da função população no intervalo [0, 4] com r = 0.5, K = 10, $\hbar$ = 0.05 (passos de 0.05) e y_0 = 1

Valores obtidos pelos diferentes métodos:

t	y(t)	y(t) Euler	y(t) Runge-Kutta
0,00	1,0000000000	1,0000000000	1,0000000000
0,05	1,0227260771	1,0225000000	1,0227260771
0,10	1,0459086083	1,0454487344	1,0459086082
...			
0,45	1,2215014596	1,2191825498	1,2215014592
0,50	1,2485630526	1,2459460983	1,2485630521
0,55	1,2761370235	1,2732137966	1,2761370230
...			
0,95	1,5158484591	1,5101793726	1,5158484580
1,00	1,5482809896	1,5422322526	1,5482809885
1,05	1,5812781026	1,5748418581	1,5812781015
...			
1,95	2,2754520324	2,2608974235	2,2754520300
2,00	2,3196931668	2,3046407162	2,3196931644
2,05	2,3645311688	2,3489783120	2,3645311662
...			
2,95	3,2690330406	3,2446116782	3,2690330369
3,00	3,3242786174	3,2994082078	3,3242786137
3,05	3,3799889982	3,3546781767	3,3799889944
...			
3,95	4,4467136317	4,4156138349	4,4467136271
4,00	4,5085306038	4,4772600670	4,5085305992

Diferença entre resultados dos métodos e resultado analítico:

t	Δy(t) Euler	Δy(t) Runge-Kutta
0,00	0,0000000000	0,0000000000
0,05	0,0002260771	0,0000000000
0,10	0,0004598739	0,0000000001
...		
0,45	0,0023189098	0,0000000004
0,50	0,0026169543	0,0000000005
0,55	0,0029232269	0,0000000005
...		
0,95	0,0056690865	0,0000000011
1,00	0,0060487370	0,0000000011
1,05	0,0064362445	0,0000000011
...		
1,95	0,0145546089	0,0000000024
2,00	0,0150524506	0,0000000024
2,05	0,0155528568	0,0000000026
...		
2,95	0,0244213624	0,0000000037
3,00	0,0248704096	0,0000000037
3,05	0,0253108215	0,0000000038
...		
3,95	0,0310997968	0,0000000046
4,00	0,0312705368	0,0000000046

Conclusão e Análise de Dados

Observando os resultados dos métodos, é evidente o aumento da precisão fornecido pelo método de Runge-Kutta. O aumento de iterações de Runge-Kutta não é tão significativa pois não aumenta a ordem de crescimento da função, que é proporcional somente a (tamanho do intervalo) / (passo). Mesmo assim, para cálculos com passo pequeno, Euler pode ser uma alternativa melhor dado que necessita de menos cálculos.

Referências

Aulas do Prof. Rex Medeiros (ECT/UFRN) - disponível em https://www.youtube.com/watch?v=l8sOUHnPkgw, acesso em 20/11/2019

Método de Euler (NUMÉRICO) - Toda a matemática - disponível em https://www.youtube.com/watch?reload=9&v=j2P6qeVPH0U, acesso em 20/11/2019

Round a double to 2 decimal places [duplicate] - disponível em https://stackoverflow.com/questions/2808535/round-a-double-to-2-decimal-places, acesso em 20/11/2019