Cálculo numérico

Nome: *Lucas Rafael Gris* ***RA: 1640496***
Turma: *A41*

Tema: Lista de Exercícios

Maio de 2017

Universidade Técnologica Federal do Paraná
Campus Medianeira

Algoritmos

A seguir apresenta-se algoritmos desenvolvidos em *Python* para a resolução de alguns exercícios.

```python
# -*- coding: utf-8 -*-

class Trapezoidal(object):
    """
    Descrição:
        Classe que representa uma integração pela regra do Trapézio.

    Nota:
        O intervalo de integração é definido automaticamente, dados o início do intervalo a, o tamanho da subdivisão e a quantidade de pontos.

        Entre sempre com um valor do tipo float no início do intervalo de integração, caso contrário o resultado poderá ser truncado.

    Args:
        f:      função a ser integrada.
        a:      início do intervalo de integração.
        h:      tamanho da subdivisão do intervalo.
        points: quantidade de pontos utilizados.
    """
    def __init__(self, f, a, h, points):
        self._f = f
        self._a = a
        self._h = h
        self._itr = points - 1

    def integr_partial(self, a):
        """
        Calcula a integral no intervalo de [a, a + h], onde h é o tamanho da subdivisão do intervalo.
        Não utilize esse método, use o método integr(). Esse método não calcula o resultado da integral para todos os pontos.

        Retorna:
            O resultado da integração.
        """
        return (self._h / 2) * (self._f(a) + self._f((a + self._h)))

    def integr(self):
        """
        Implementação da regra generalizada do Trapézio. Efetua o cálculo da integral.

```

```
        Retorna:
            O resultado da integração.
        ”””
        self._sum = 0
        xa = self._a;
        for i in range(0, self._itr):
            self._sum = self._sum + self.integr_partial(xa)
            xa = xa + self._h
        return self._sum

class Simpson(object):
    ”””
    Descrição:
        Classe que representa uma integração pela regra 1/3 de Simpson.

    Nota:
        O intervalo de integração é definido automaticamente, dados o início do intervalo a, o tamanho da subdivisão e a quantidade de pontos.

        Entre sempre com um valor do tipo float no início do intervalo de integração, caso contrário o resultado poderá ser truncado.

    Args:
        f:      função a ser integrada.
        a:      início do intervalo de integração.
        h:      tamanho da subdivisão do intervalo.
        points: quantidade de pontos utilizados.
    ”””
    def __init__(self, f, a, h, points):
        self._f = f
        self._a = a
        self._h = h
        self._points = points
        self._values = list()
        for i in range(0, self._points):
            self._values.append(f(self._a + (i * h)))

    def integr(self):
        ”””
        Integra a função.

        Retorna:
            O resultado da integração.
        ”””
        sum = 0.0
        sum += self._values[0] + self._values[self._points - 1]
        for i in range(1, self._points - 1):
            if (i % 2 == 0):
                sum += 2 * self._values[i]
            else:
                sum += 4 * self._values[i]
```

```python
        return (sum * self._h) / 3

class Simpson2(object):
    ”””
    Descrição:
        Classe que representa uma integração pela regra 3/8 de Simpson.

    Nota:
        O intervalo de integração é definido automaticamente, dados o início do intervalo a, o tamanho da subdivisão e a quantidade de pontos.

        Entre sempre com um valor do tipo float no início do intervalo de integração, caso contrário o resultado poderá ser truncado.

    Args:
        f:      função a ser integrada.
        a:      início do intervalo de integração.
        h:      tamanho da subdivisão do intervalo.
        points: quantidade de pontos utilizados.
    ”””
    def __init__(self, f, a, h, points):
        self._f = f
        self._a = a
        self._h = h
        self._points = points
        self._values = list()
        for i in range(0, self._points):
            self._values.append(f(self._a + (i * h)))

    def integr(self):
        ”””
        Integra a função.

        Retorna:
            O resultado da integração.
        ”””
        sum = 0.0
        sum += self._values[0] + self._values[self._points - 1]
        for i in range(1, self._points - 1):
            if (i % 3 == 0):
                sum += 2 * self._values[i]
            else:
                sum += 3 * self._values[i]
        return (sum * self._h * 3) / 8
```

Código 1: Métodos de integração numérica em *Python*

```python
# -*- coding: utf-8 -*-

from math import factorial

```

```python
class Euler(object):
    ”””
    Descrição:
        Classe que representa uma resolução de um P.V.I. atravéz do método de Euler melhorado.

    Args:
        f:      função em termos de x e y da equação diferencial
        step:   passo entre os cálculos das soluções
        x0:     valor inicial do P.V.I.
        y0:     resultado inicial do P.V.I.
        points: quantidade de pontos
    ”””
    def __init__(self, f, step, x0, y0, points):
        self._f = f
        self._step = step
        self._y0 = y0
        self._x0 = x0
        self._res = dict()
        self._res[x0] = y0
        self._points = points

    def calc(self):
        ”””
        Descrição
            Resolve o P.V.I.

        Nota:
            O cálculo é realizado a partir do valor inicial (x0, y0) , seguido pelos passos e a quantidade de pontos.
        ”””
        xi = self._x0
        yi = self._y0
        for n in range(0, self._points):
            k1 = self._f(xi, yi)
            k2 = self._f(xi + self._step, yi + (self._step*k1))
            yn = yi + (self._step / 2)*(k1 + k2)
            self._res[xi + self._step] = yn
            xi += self._step
            yi = yn

    def __str__(self):
        ”””
        Descrição:
            Representação do objeto, retorna uma string contendo os dados do objeto e a solução do P.V.I.

        Nota:
            Se o método calc() não for chamado, apenas o valor inicial será retornado como resultado.
        ”””
        str = ”Resolução␣de␣P.V.I␣pelo␣método␣de␣Euler␣melhorado”
```

```
        str += ”\nValor␣inicial:␣y({})␣=␣{}”.format(self._x0, self._y0)
        str += ”\nTamanho␣do␣passo:␣{}”.format(self._step)
        str += ”\nResolução:”
        for x in sorted(self._res.iterkeys()):
            str += ”\ny({})␣=␣{}”.format(x, self._res[x])
        return str

class RungeKutta(object):
    ”””
    Descrição:
        Classe que representa uma resolução de um P.V.I. atravéz do método de Runge Kutta de quarta ordem.

    Args:
        f:      função em termos de x e y da equação diferencial
        step:   passo entre os cálculos das soluções
        x0:     valor inicial do P.V.I.
        y0:     resultado inicial do P.V.I.
        points: quantidade de pontos
    ”””
    def __init__(self, f, step, x0, y0, points):
        self._f = f
        self._step = step
        self._y0 = y0
        self._x0 = x0
        self._res = dict()
        self._res[x0] = y0
        self._points = points

    def calc(self):
        ”””
        Descrição
            Resolve o P.V.I.

        Nota:
            O cálculo é realizado a partir do valor inicial (x0, y0) , seguido pelos passos e a quantidade de pontos.
        ”””
        xi = self._x0
        yi = self._y0
        for n in range(0, self._points):
            k1 = self._f(xi, yi)
            k2 = self._f(xi + (self._step / 2), yi + ((self._step*k1)/2))
            k3 = self._f(xi + (self._step / 2), yi + ((self._step*k2)/2))
            k4 = self._f(xi + self._step, yi + (self._step*k3))
            yn = yi + (self._step / 6)*(k1 + 2*(k2 + k3) + k4)
            self._res[xi + self._step] = yn
            xi += self._step
            yi = yn

    def __str__(self):
        ”””
```

```
        Descrição:
            Representação do objeto, retorna uma string contendo os dados do objeto e a solução do P.V.I.

        Nota:
            Se o método calc() não for chamado, apenas o valor inicial será retornado como resultado.
        """
        str = "Resolução de P.V.I pelo método de Runge Kutta de quarta ordem"
        str += "\nValor inicial: y({}) = {}".format(self._x0, self._y0)
        str += "\nTamanho do passo: {}".format(self._step)
        str += "\nResolução:"
        for x in sorted(self._res.iterkeys()):
            str += "\ny({}) = {}".format(x, self._res[x])
        return str

class Taylor(object):
    """
    Descrição:
        Classe que representa uma resolução de um P.V.I. atravéz do método de Taylor.

    Args:
        functions:  uma lista contendo a função e suas derivadas até a ordem n
        step:       passo entre os cálculos das soluções
        x0:         valor inicial do P.V.I.
        y0:         resultado inicial do P.V.I.
        points:     quantidade de pontos

    Nota:
        Entre com uma lista contendo a função e suas derivadas para a aplicação do método.
        No caso de uma aplicação do método de segunda ordem por exemplo, teríamos, em functions, as referencias para as funções [f, f', f'']
    """
    def __init__(self, functions, step, x0, y0, points):
        self._functions = functions
        self._step = step
        self._y0 = y0
        self._x0 = x0
        self._res = dict()
        self._res[x0] = y0
        self._points = points

    def calc(self):
        """
        Descrição
            Resolve o P.V.I.

```

```
        Nota:
            O cálculo é realizado a partir do valor inicial (x0, y0) , seguido pelos passos e a quantidade de pontos.
        """
        xi = self._x0
        for n in range(0, self._points):
            yn = self._res[xi]
            for i in range(0, len(self._functions)):
                yn += (self._step**((i + 1))/factorial((i + 1)) * self._functions[i](xi, self._res[xi]))
            xi += self._step
            self._res[xi] = yn

    def __str__(self):
        """
        Descrição:
            Representação do objeto, retorna uma string contendo os dados do objeto e a solução do P.V.I.

        Nota:
            Se o método calc() não for chamado, apenas o valor inicial será retornado como resultado.
        """
        str = "Resolução de P.V.I pelo método de Taylor de ordem {}".format(len(self._functions))
        str += "\nValor inicial: y({}) = {}".format(self._x0, self._y0)
        str += "\nTamanho do passo: {}".format(self._step)
        str += "\nResolução:"
        for x in sorted(self._res.iterkeys()):
            str += "\ny({}) = {}".format(x, self._res[x])
        return str
```

Código 2: Métodos de resolução de EDO em *Python*

Lista de exercícios 5

0.1 Exercício 1

Estamos interessados em aplicar a regra do trapézio para calcular $\int_{1.00}^{1.30} \sqrt{x}dx$ utilizando os seguintes pontos:

Tabela 0.1: Pontos $\sqrt{x}$

x	1.00	1.05	1.10	1.15	1.20	1.25	1.30
$\sqrt{x}$	1.0000	1.0247	1.0488	1.0723	1.0954	1.1180	1.1401

Sabemos que:

$$
\begin{aligned}
\int_{x_1}^{x_n} f(x)dx &= \int_{x_0}^{x_1} f(x) + \int_{x_1}^{x_2} f(x) + \ldots + \int_{x_{n-1}}^{x_n} f(x) \\
&= \frac{h}{2}\left[f(x_0) + f(x_1)\right] + \frac{h}{2}\left[f(x_1) + f(x_3)\right] + \ldots + \frac{h}{2}\left[f(x_{n-1}) + f(x_n)\right] \\
&= \frac{h}{2}\left[f(x_0) + 2 \cdot (f(x_1) + f(x_2) + \ldots + f(x_{n-1})) + f(x_n)\right]
\end{aligned}
$$

Então aplicando nos pontos em 0.1 obtemos:

$$
\begin{aligned}
\int_{1.00}^{1.30} \sqrt{x}dx &= \frac{0.05}{2}\left[1.0000 + 2 \cdot (1.0247 + 1.0488 + 1.0723 + 1.0954 + 1.1180) + 1.1401\right] \\
&= \frac{0.05}{2}\left[1.0000 + (10.7184) + 1.1401\right] \\
&= 0.3214625
\end{aligned}
$$

0.2 Exercício 2

Queremos calcular $\int_0^{0.8} \cos(x)dx$ utilizando a regra do trapézio, com $h = 0.4, 0.2$ e 0.1, para os seguintes pontos:

Tabela 0.2: Pontos $\cos x$

x	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
$\cos(x)$	1	0.995	0.980	0.955	0.921	0.877	0.825	0.764	0.696

Para $h = 0.4$ temos:

$$\begin{aligned}\int_0^{0.8} \cos(x)dx &= \frac{0.4}{2}\left[1 + 2\cdot(0.921) + 0.696\right] = \frac{0.4}{2}\left[1 + (1.842) + 0.696\right] \\ &= 0.7076\end{aligned}$$

Para $h = 0.2$ temos:

$$\begin{aligned}\int_0^{0.8} \cos(x)dx &= \frac{0.2}{2}\left[1 + 2\cdot(0.980 + 0.921 + 0.825) + 0.696\right] = \frac{0.4}{2}\left[1 + (5.452) + 0.696\right] \\ &= 0.7248\end{aligned}$$

Para $h = 0.1$ temos:

$$\begin{aligned}\int_0^{0.8} \cos(x)dx &= \frac{0.1}{2}\left[1 + 2\cdot(0.995 + 0.980 + 0.955 + 0.921 + 0.877 + 0.825 + 0.764) + 0.696\right] \\ &= \frac{0.1}{2}\left[1 + (12.64) + 0.696\right] \\ &= 0.7168\end{aligned}$$

0.3 Exercício 3

Estamos interessados em obter o valor das integrais definidas discutidas em 0.1 e 0.2 através da regra de $\frac{1}{3}$ de *Simpson* e da integral discutida em 0.1 utilizando a regra $\frac{3}{8}$ de *Simpson*.

Para $\int_{1.00}^{1.30} \sqrt{x}dx$ e utilizando a regra de $\frac{1}{3}$ de *Simpson* temos:

$$
\begin{aligned}
\int_{x_0}^{x_{2n}} f(x)dx &= \int_{x_0}^{x_2} f(x) + \int_{x_2}^{x_4} f(x) + \ldots + \int_{x_{2n-2}}^{x_{2n}} f(x) \\
&= \frac{h}{3}\left[f(x_0) + 4f(x_1) + f(x_2)\right] + \frac{h}{3}\left[f(x_2) + 4f(x_3) + f(x_4)\right] + \ldots \\
&+ \frac{h}{3}\left[f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n})\right] \\
&= \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n})\right]
\end{aligned}
$$

Note que para aplicarmos a regra de *Simpson* temos que obrigatoriamente utilizar $2n+1$ pontos. Assim, utilizaremos todos os pontos de 0.1 para obter uma solução da integral definida.

$$
\begin{aligned}
2n + 1 = 7 &\implies \quad n = 3 \\
&\implies h = \frac{b-a}{2n} = \frac{1.30 - 1.00}{6} = \frac{0.3}{6}
\end{aligned}
$$

para 0.1, para 0.2 temos:

$$
\begin{aligned}
2n + 1 = 9 &\implies \quad n = 4 \\
&\implies h = \frac{b-a}{2n} = \frac{0.8 - 0.0}{8} = \frac{0.8}{8}
\end{aligned}
$$

Logo,

$$
\begin{aligned}
\int_{1.00}^{1.30} \sqrt{x}dx &= \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n})\right] \\
&= \frac{0.3}{18}\left[1.0000 + 4(1.0247 + 1.0723 + 1.1180) + 2(1.0488 + 1.0954) + 1.1401)\right] \\
&= \frac{0.3}{18}\left[1.0000 + 4(3.215) + 2(2.1447) + 1.1401)\right] \\
&= 0.3215
\end{aligned}
$$

$$
\begin{aligned}
\int_0^{0.8} \cos x dx &= \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n})\right] \\
&= \frac{0.8}{24}\left[1 + 4(0.995 + 0.955 + 0.877 + 0.764) + 2(0.980 + 0.921 + 0.825) + 0.696)\right] \\
&= \frac{0.8}{24}\left[1 + 4(3.591) + 2(2.726) + 0.696)\right] \\
&= 0.7170
\end{aligned}
$$

De forma análoga, queremos calcular $\int_{1.00}^{1.30} \sqrt{x}dx$ pela regra $\frac{3}{8}$ de *Simpson.* Neste caso temos que utilizar $3n + 1$ pontos.

Assim,

$$
\begin{aligned}
3n + 1 = 7 &\implies \quad n = 2 \\
&\implies h = \frac{b-a}{3n} = \frac{1.30 - 1.00}{6} = \frac{0.3}{6}
\end{aligned}
$$

Logo,

$$
\begin{aligned}
\int_{1.00}^{1.30} \sqrt{x}dx &= \frac{3h}{8}[f(x_0) + 3f(x_1) + 3f(x_2) + 2f(x_3) + \ldots \\
&\quad + 2f(x_{3n-3}) + 3f(x_{3n-2}) + 3f(x_{3n-1}) + f(x_{3n})] \\
&= \frac{0.9}{48}\left[1.0000 + 3(1.0247 + 1.0488 + 1.0954 + 1.1180) + 2(1.0723) + 1.1401\right] \\
&= \frac{0.9}{48}\left[1.0000 + 3(4.2869) + 2(1.0723) + 1.1401\right] \\
&= 0.32147
\end{aligned}
$$

0.4 Exercício 4

Dada a integral definida

$$\int_0^1 t^3 e^t dt$$

Queremos obter a solução da mesma através dos métodos do Trapézio e a Regra 1/3 de *Simpson* utilizando uma distancia h entre os pontos de 0.5 e 0.25 .

Para isso obteremos os valores da função $f(t) = t^3e^t$ em $[0, 1]$ utilizando um intervalo de 0.25 entre os pontos.

Tabela 0.3: Pontos t^3e^t

t	0	0.25	0.5	0.75	1
$f(t)$	0	0.0020062	0.206090	0.893109	2.718281

Para $h = 0.5$, utilizando a Regra do Trapézio:

$$\begin{aligned}\int_0^1 t^3e^t dt &= \frac{0.5}{2}\left[0 + 2(0.206090) + 2.718281\right] \\ &= \frac{0.5}{2}\left[(0.41218) + 2.718281\right] \\ &= 0.782615\end{aligned}$$

Com $h = 0.25$, e utilizando a Regra do Trapézio obtemos:

$$\begin{aligned}\int_0^1 t^3e^t dt &= \frac{0.25}{2}\left[0 + 2(0.0020062 + 0.206090 + 0.893109) + 2.718281\right] \\ &= \frac{0.25}{2}\left[(1.1012052) + 2.718281\right] \\ &= 0.615085\end{aligned}$$

Para $h = 0.5$, utilizando a Regra 1/3 de *Simpson*:

$$\begin{aligned}\int_0^1 t^3e^t dt &= \frac{0.5}{3}\left[0 + 4(0.206090) + 2.718281\right] \\ &= \frac{0.5}{3}\left[(0.82436) + 2.718281\right] \\ &= 0.59044\end{aligned}$$

Com $h = 0.25$, e utilizando a Regra 1/3 de *Simpson* obtemos:

$$\begin{aligned}\int_0^1 t^3 e^t dt &= \frac{0.25}{3}\left[0 + 4(0.0020062 + 0.893109) + 2(0.206090) + 2.718281\right] \\ &= \frac{0.25}{3}\left[4(0.8951152) + 2(0.41218) + 2.718281\right] \\ &= 0.5592\end{aligned}$$

0.5 Exercício 5

A fórmula de *Newton-Cotes* é exata para polinômios de grau n, onde n é o grau do polinômio usado na interpolação para a obtenção da fórmula de integração.

Por exemplo, no caso de uma aproximação do tipo:

$$\int_a^b f(x)dx \approx \int_a^b P_n(x)dx = \sum_{k=0}^{n} f_k \cdot h \cdot C_k^n$$

O cálculo para a integral definida de $P_n(x)$ é exato. Para a Regra $\frac{3}{8}$ de *Simpson*, o cálculo é exato caso a função utilizada seja uma função de grau três.

Exemplo. Considere $P_3(x) = x^3$ e o intervalo [0, 1], analiticamente temos:

$$\int_0^1 x^3 dx = \frac{1}{4}x^4\Big|_0^1 = \frac{1}{4}$$

E por *Newton-Cotes* (Regra $\frac{3}{8}$ de *Simpson*), temos:

$$h = \frac{1-0}{3} = \frac{1}{3} \quad \Longrightarrow \quad x_0 = 0,\ x_1 = \frac{1}{3},\ x_2 = \frac{2}{3},\ x_3 = 1$$

Assim,

$$\begin{aligned}\int_0^1 x^3 dx &= \frac{3}{8}\cdot\left(\frac{1}{3}\right)\cdot\left[0+3\frac{1}{27}+3\frac{8}{27}+1\right]\\ &= \frac{1}{8}\left[2\right]\\ &= \frac{1}{4}\end{aligned}$$

0.6 Exercício 6

Estamos interessados em calcular $\int_0^{1.2} e^{-x}\sin(x)$ através da Regra $\frac{3}{8}$ de *Simpson*. Sabemos que:

Tabela 0.4: Pontos e^{-x} e $\sin(x)$

x	0	0.2	0.4	0.6	0.8	1.0	1.2
e^{-x}	1.000	0.819	0.670	0.548	0.449	0.367	0.301
$\sin(x)$	0	0.198	0.398	0.565	0.717	0.841	0.932

Então é suficiente obter a fórmula da regra de *Simpson* e aplicar os pontos da função f obtidos através do produto entre e^{-x} e $\sin(x)$.

Considerando $h = 0.4$ temos que:

$$\begin{aligned}3n+1=4 &\implies \quad n=1\\ &\implies h=\frac{b-a}{3n}=\frac{1.2-0}{3}=\frac{1.2}{3}\end{aligned}$$

Logo,

$$\begin{aligned}\int_0^{1.2} e^{-x}\sin(x)dx &= \frac{3h}{8}[f(x_0)+3f(x_1)+3f(x_2)+f(x_3)]\\ &= \frac{1.2}{8}\left[0+3(0.26666+0.321933)+0.290532)\right]\\ &= \frac{1.2}{8}\left[2.056311\right]\\ &= 0.30844665\end{aligned}$$

E considerando $h = 0.2$ obtemos:

$$\begin{aligned} 3n + 1 = 7 &\implies \quad n = 2 \\ &\implies h = \frac{b-a}{3n} = \frac{1.2 - 0}{6} = \frac{1.2}{6} \end{aligned}$$

Logo,

$$\begin{aligned} \int_0^{1.2} e^{-x}\sin(x)dx &= \frac{3h}{8}[f(x_0) + 3[f(x_1) + f(x_2) + f(x_4) + f(x_5)] + 2f(x_3) + f(x_6)] \\ &= \frac{3.6}{48}[0 + 3(0.16212 + 0.26666 + 0.321933 + 0.308647) + 2(0.30962) + 0.280532] \\ &= \frac{3.6}{48}[3.17808 + 0.61924 + 0.280532] \\ &= 0.3058389 \end{aligned}$$

0.7 Execício 7

Estamos interessados em definir um intervalo igualmente espaçado h para o cálculo da integral definida utilizando a Regra do Trapézio, tal que o valor de $\int_0^1 e^{-x^2}$ tenha um erro inferior a 0.5×10^{-6}.

$$\begin{aligned} |R(f)| \le \frac{b-a}{12} \times h^2 \times \max_{a \le t \le b} |f^{''}(t)| < 0.5 \times 10^{-6} \\ \implies \quad h^2 < \frac{0.5 \times 10^{-6}}{\frac{b-a}{12} \times \max_{a \le t \le b} |f^{''}(t)|} \end{aligned}$$

$$\begin{aligned} f^{'}(x) &= -2xe^{-x^2} \\ f^{''}(x) &= 4x^2e^{-x^2} - 2e^{-x^2} = (4x^2 - 2)e^{-x^2} \end{aligned}$$

A função $f^{''}(x)$ é sempre crescente no intervalo (0, 1), logo:

$$\max_{0 \le t \le 1} |f^{''}(t)| = |f^{''}(1)| = 0.7357588$$

Portanto,

$$\begin{aligned} h^2 &< \frac{0.5 \times 10^{-6}}{\frac{1}{12} \times 0.7357588} \\ h^2 &< 8.15 \times 10^{-6} \\ h &< 2.85566 \times 10^{-3} \end{aligned}$$

Para encontrarmos um valor exato de h que satisfaça a condição, devemos obter o valor máximo de h tal que $h < 2.85566 \times 10^{-3}$ seja satisfeita e seja possível obtermos todos os pontos necessários no intervalo [0, 1].

Note que o espaçamento h é dado por:

$$h = \frac{b-a}{N}$$

onde $N+1$ é a quantidade de pontos utilizados na aplicação da regra do Trapézio. Assim:

$$\begin{aligned} h = \frac{b-a}{N} < 2.85566 \times 10^{-3} \implies N &> \frac{1}{2.85566 \times 10^{-3}} \\ N &> 350.181744 \end{aligned}$$

que não é um número inteiro. Logo devemos encontrar um valor h tal que:

$$\frac{b-a}{h} = \frac{1}{h} = 351$$

De fato,

$$h = 0.002849002849 < 2.85566 \times 10^{-6}$$

Portanto um intervalo igualmente espaçado $h = 0.002849002849$ entre os pontos satisfaz o erro desejado num total de 352 pontos.

Cálculo da integral. O valor da integral pode ser calculado utilizando os algoritmos desenvolvidos em *Python* e em C:

```python
# -*- coding: utf-8 -*-

from integrmethods import Trapezoidal
from math import exp

def f(x):
    ”””
    Descrição:
        Definição da função a ser integrada.
    ”””
    return exp(-(x**2))

# Cálculo e exibição de resultados
fun_int_tr = Trapezoidal(f, 0, 0.002849002849, 352)
print(fun_int_tr.integr())
```

Código 3: Cálculo em *Python*

```c
#include <stdio.h>
#include <math.h>

/**
 * Definições de função, intervalo e pontos.
 */
#define   F(x)            exp((x*x)*(-1))
#define   POINTS          352
#define   H               0.002849002849
#define   BEGIN           0

#define   PRINT(result)   printf(”%.10f\n”, result);

/**
 * Calcula a integral pela regra do Trapézio, em um intervalo [a, a+h], onde h é o tamanho da subdivisão do intervalo.
 */
double integrate(double h, double *a);
/**
 * Calcula a integral de todo o intervalo [a, b], formado pelas subdivisões igualmente espaçadas h.
 */
double integrateAll(int subdivisions, double h, double a);

/**
 * Função principal do programa, calcula a integral de F(x).
 */
int main() {
```

```
  PRINT(integrateAll(POINTS - 1, H, BEGIN));
  return 0;
}

double integrate(double h, double *a) {
  return (h / 2) * (F(*a) + F((*a + h)));
}

double integrateAll(int subdivisions, double h, double begin) {
  int i;
  double result = 0;
  double a = begin;
  for (i = 1; i < subdivisions; i++) {
    result += integrate(h, &a);
    a += h;
  }
  return result;
}
```

Código 4: Cálculo de integral pela regra do Trapézio em C

Onde obtemos os valores 0.746823635144 e 0.7468236351 respectivamente.

0.8 Exercício 8

Queremos calcular $\int_1^2 \frac{e^x}{x} dx$ com erro inferior a 0.05, usando a regra do Trapézio.

$$|R(f)| \leq \frac{b-a}{12} \times h^2 \times \max_{a \leq t \leq b} |f^{''}(t)| < 0.05$$

$$\implies \quad h^2 < \frac{0.05}{\frac{b-a}{12} \times \max_{a \leq t \leq b} |f^{''}(t)|}$$

$$f^{'}(x) = \frac{e^x(x-1)}{x^2}$$

$$f^{''}(x) = \frac{e^x(x^2-2x+2)}{x^3}$$

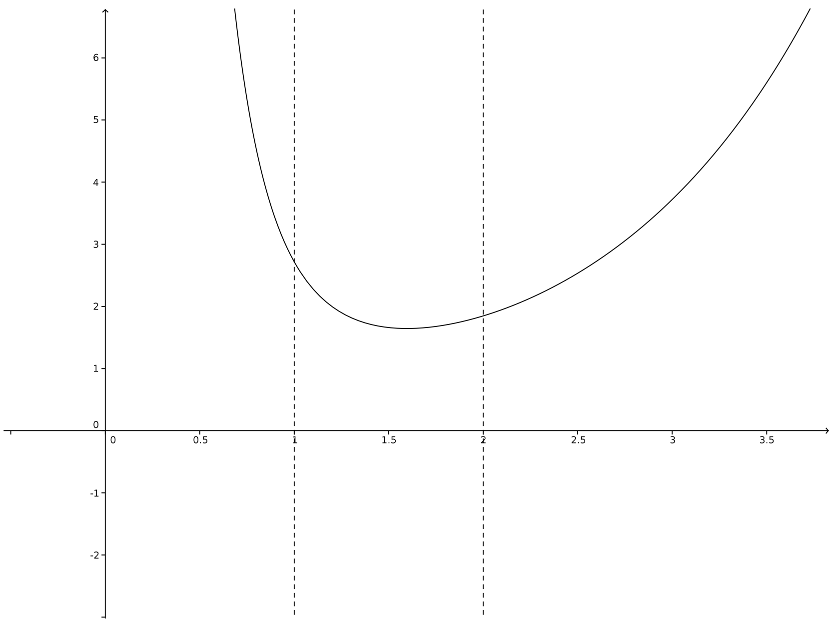

Figura 0.1: Gráfico de $f^{''}(x)$

Através do gráfico vemos que o máximo ocorre em $x = 1$. Assim:

$$\max_{0 \leq t \leq 1} |f^{''}(t)| = |f^{''}(1)| = 2.7182818$$

Portanto,

$$\begin{aligned} h^2 &< \frac{0.05}{\frac{1}{12} \times 2.7182818} \\ h^2 &< 0.22072766 \\ h &< 0.46981662 \end{aligned}$$

Note que o espaçamento h é dado por:

$$h = \frac{b-a}{N}$$

onde $N+1$ é a quantidade de pontos utilizados na aplicação da regra do Trapézio. Assim:

$$h = \frac{b-a}{N} < 0.46981662 \implies N > \frac{1}{0.46981662}$$
$$N > 2.12849$$

Logo devemos encontrar um valor h tal que:

$$\frac{b-a}{h} = \frac{1}{h} = 3$$

De fato,

$$h = 0.33333 < 0.46981$$

Portanto um intervalo igualmente espaçado $h = 0.33333$ entre os pontos satisfaz o erro desejado num total de 4 pontos.

Cálculo da integral.

```python
# -*- coding: utf-8 -*-

from integrmethods import Trapezoidal
from math import exp

def f(x):
    ”””
    Descrição:
        Definição da função a ser integrada.
    ”””
    return exp(-(x**2))

# Cálculo e exibição de resultados
fun_int_tr = Trapezoidal(f, 1, 0.3333333, 4)
print(fun_int_tr.integr())
```

Código 5: Cálculo em *Python*

Executando o código acima obtemos 3.076079.

0.9 Exercício 9

Estamos interessados em obter um número mínimo de intervalos para o cálculo de

$$\int_0^{\frac{\pi}{2}} e^{-x}\cos(x)$$

através da regra $\frac{1}{3}$ de *Simpson*, tal que o resultado obtido na integração tenha um erro menor que 10^{-5}.

Assim, devemos calcular o valor do espaçamento h entre os pontos no cálculo, de forma a garantir o erro desejado.

$$\begin{aligned} |R(f)| \leq \frac{b-a}{180} \times h^4 \times \max_{a\leq t\leq b} |f^{iv}(t)| < 10^{-5} \\ \implies \quad h^4 < \frac{10^{-5}}{\frac{b-a}{180} \times \max_{a\leq t\leq b} |f^{iv}(t)|} \end{aligned}$$

Onde $|f^{iv}(t)|$ é obtido fazendo:

$$\begin{aligned} f^{'}(x) &= -e^{-x}(\sin(x) + \cos(x)) \\ f^{''}(x) &= 2e^{-x}\sin(x) \\ f^{'''}(x) &= 2e^{-x}(\cos(x) - \sin(x)) \\ f^{iv}(x) &= -4e^{-x}\cos(x) \end{aligned}$$

Através do gráfico vemos que o máximo de $|f^{iv}(x)|$ ocorre em 0. Essa análise poderia ser feita observando que o produto $-4e^{-x}$ com $\cos(x)$ é mínimo quando e^{-x} for mínimo, o que implica no maior valor absoluto possível de f nesse intervalo ($\cos(x)$ é nulo em $\frac{\pi}{2}$).

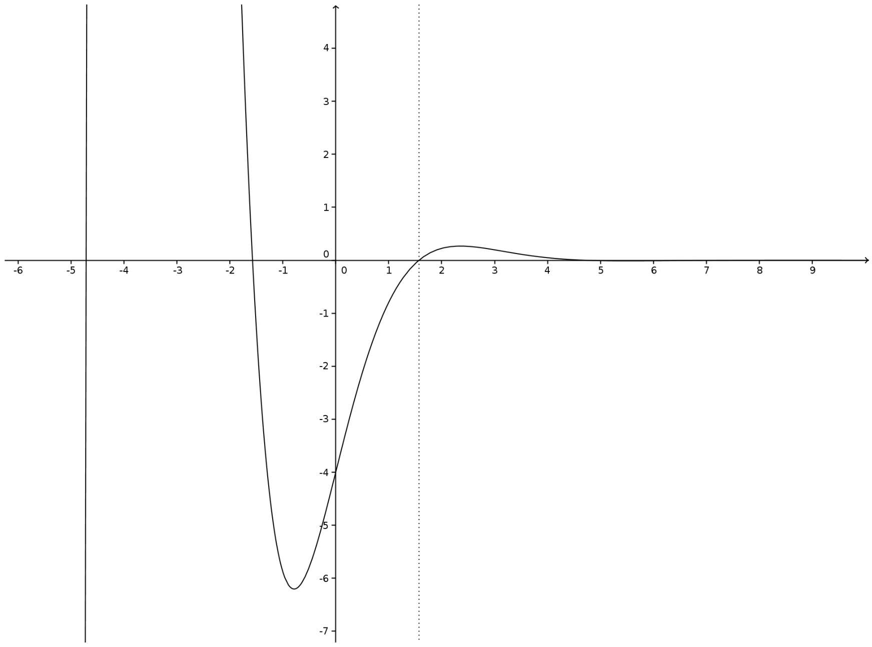

Figura 0.2: Gráfico de $-4e^{-x}\cos(x)$

Continuando o cálculo de h obtemos:

$$\begin{aligned} h^4 &< \frac{10^{-5}}{\frac{b-a}{180} \times \max_{a \leq t \leq b} |f^{iv}(t)|} = \frac{10^{-5}}{\frac{\frac{pi}{2}-0}{180} \times |f^{iv}(0)|} = \frac{10^{-5}}{\frac{\pi}{90}} \\ h^4 &< 0.00028647 \\ h &< 0.13 \end{aligned}$$

Note que o espaçamento h é dado por:

$$h = \frac{b-a}{2N}$$

e que o número de pontos necessários para aplicar a regra de $\frac{1}{3}$ de *Simpson* é dado por $2N+1$. Assim:

$$\begin{aligned} h = \frac{b-a}{2N} < 0.13 \implies 2N &> \frac{\frac{\pi}{2}}{0.13} \\ 2N &> 12.0830486677 \\ N &> 6.04152433385 \end{aligned}$$

Então um valor inteiro $N = 7$ satisfaz a condição, o que implica que devemos utilizar pelo menos $2N + 1 = 15$ pontos para garantir o erro desejado. Logo devemos utilizar um valor h tal que:

$$h = \frac{\frac{\pi}{2}}{14} = 0.112199737629$$

Portanto um intervalo igualmente espaçado $h = 0.112199737629$ entre os pontos satisfaz o erro desejado,em um total de $n = 15$ pontos.

Prova. Calculando a integral analiticamente temos que:

$$\begin{aligned}
\int e^{-x} cos(x)dx &= e^{-x}\sin(x) + \int e^{-x}\sin(x)dx \\
&= e^{-x}\sin(x) + \Big[- e^{-x}\cos(x) - \int e^{-x}\cos(x) \Big] \\
2\int e^{-x} cos(x)dx &= e^{-x}\sin(x) - e^{-x}\cos(x) \\
\int e^{-x} cos(x)dx &= \frac{e^{-x}\sin(x) - e^{-x}\cos(x)}{2}
\end{aligned}$$

$$\begin{aligned}
\int_0^{\frac{\pi}{2}} e^{-x} cos(x)dx &= \frac{e^{-x}\sin(x) - e^{-x}\cos(x)}{2}\Bigg|_0^{\frac{\pi}{2}} \\
&= \frac{1}{2}\Big[e{-}\frac{\pi}{2} - 1(-1)\Big] \\
&= 0.60393978
\end{aligned}$$

E computacionalmente como segue:

```
# -*- coding: utf-8 -*-

from integrmethods import Simpson
from math import exp
from math import cos

def f(x):
    """
    Descrição:
```

```python
        Definição da função a ser integrada.
    """
    return exp(-x)*cos(x)

# Cálculo e exibição de resultados
fun_int_tr = Simpson(f, 0, 0.112199737629, 15)
print(fun_int_tr.integr())
```

Código 6: Cálculo em *Python*

```c
#include <math.h>
#include <stdio.h>

/**
 * Definições de função, intervalo e pontos.
 */
#define   F(x)            exp(x*(-1))*cos(x)
#define   POINTS          15
#define   H               0.112199737629
#define   BEGIN           0

#define   PRINT(result)   printf("%.10f\n", result);

/**
 * Configura os valores da função nos pontos em um vetor.
 */
void setValues(double begin, double increment,double values[POINTS]);
/**
 * Integra a função F(x) na regra 1/3 de Simpson.
 */
double integrate(double h, double values[POINTS]);

/**
 * Função principal, calcula e exibe o resultado da integral.
 */
int main() {
  double values[POINTS];
  setValues(BEGIN, H, values);
  PRINT(integrate(H, values));
  return 0;
}

void setValues(double begin, double increment, double values[POINTS]) {
  int i;
  double x = begin;
  for (i = 0; i < POINTS; i++) {
    values[i] = F(x);
    x += increment;
  }
}
```

```
double integrate(double h, double values[POINTS]) {
  double sum = 0;
  sum += values[0] + values[POINTS - 1];
  int i;
  for (i = 1; i < POINTS - 1; i++) {
    if (i & 1) {
      sum += 4 * values[i];
    } else {
      sum += 2 * values[i];
    }
  }
  return (sum*h) / 3;
}
```

Código 7: Cálculo de integral pela regra 1/3 de *Simpson* em *C*

Obtemos 0.603937665466 e 0.6039376655 respectivamente.

0.10 Exercício 10

Similar ao exercício anterior, queremos definir um intervalo h de modo que a regra $\frac{3}{8}$ de *Simpson* forneça um valor de $\int_{0.2}^{0.8} \sin(x)dx$ com um erro inferior a 0.5×10^{-3}.

O valor de h utilizando a regra $\frac{3}{8}$ de *Simpson* é obtido fazendo:

$$|R(f)| \leq \frac{b-a}{80} \times h^4 \times \max_{a \leq t \leq b} |f^{iv}(t)| < 0.5 \times 10^{-3}$$
$$\implies \quad h^4 < \frac{0.5 \times 10^{-3}}{\frac{b-a}{80} \times \max_{a \leq t \leq b} |f^{iv}(t)|}$$

Assim,

$$\begin{aligned} f^{'}(x) &= \cos(x) \\ f^{''}(x) &= -\sin(x) \\ f^{'''}(x) &= -\cos(x) \\ f^{iv}(x) &= \sin(x) \end{aligned}$$

Onde $\max_{0.2}^{0.8} |\sin(x)|$ ocorre em 0.8. Logo:

$$
\begin{aligned}
h^4 &< \frac{0.5 \times 10^{-3}}{\frac{b-a}{80} \times \max_{a \le t \le b} |f^{iv}(t)|} = \frac{0.5 \times 10^{-3}}{\frac{0.8-0.2}{80} \times |f^{iv}(0.8)|} = \frac{0.5 \times 10^{-3}}{0.717356} \\
h^4 &< 0.000697004 \\
h &< 0.162483332738
\end{aligned}
$$

Note que o espaçamento h é dado por:

$$h = \frac{b-a}{3N}$$

e que o número de pontos necessários para aplicar a regra de $\frac{3}{8}$ de *Simpson* é dado por $3N+1$. Portanto:

$$
\begin{aligned}
h = \frac{b-a}{3N} < 0.162483332738 \implies 3N &> \frac{0.6}{0.162483332738} \\
3N &> 3.692694 \\
N &> 1.260898
\end{aligned}
$$

Note que um valor inteiro $N = 2$ satisfaz a condição, então devemos utilizar pelo menos $3N + 1 = 7$ pontos para garantir o erro desejado. Logo devemos utilizar um valor h tal que:

$$h = \frac{0.6}{6} = 0.1$$

Portanto um intervalo igualmente espaçado $h = 0.1$ entre os pontos satisfaz o erro desejado, o que implica em uma quantidade de pontos $n = 7$.

Prova. Calculando analiticamente obtemos:

$$
\begin{aligned}
\int_{0.2}^{0.8} \sin(x)dx &= -\cos(x)\Big|_{0.2}^{0.8} \\
&= -0.696706709347 - (-0.980066577841) \\
&= 0.283359868494
\end{aligned}
$$

Calculando pela regra 3/8 de *Simpson* temos que:

Tabela 0.5: Pontos $\sin(x)$

x	0.2	0.3	0.4	0.5	0.6	0.7	0.8
$f(x)$	0.198669	0.295520	0.389418	0.479425	0.564642	0.644217	0.71735

$$
\begin{aligned}
\int_{0.2}^{0.8} \sin(x)dx &= \frac{3h}{8}[f(x_0) + 3[f(x_1) + f(x_2) + f(x_4) + f(x_5)] + 2f(x_3) + f(x_6)] \\
&= \frac{0.3}{8}[0.198669 + 3(0.295520 + 0.389418 + 0.564642 + 0.644217) + 2(0.479425) \\
&+ 0.71735] \\
&= \frac{0.3}{8}[0.916019 + 3(1.893797) + 0.95885] \\
&= \frac{0.3}{8}[7.55626] \\
&= 0.28335975
\end{aligned}
$$

Observe que o erro obtido está em conformidade com o esperado.

0.11 Exercício 11

Estamos interessados em calcular $\Gamma(\alpha) = \int_0^\infty e^{-x}x^{\alpha-1}dx$ com $\alpha = 5$, através da Quadratura de Gauss.

Note que o intervalo de integração e a função peso coincidem com o polinômio ortogonal de *Laguerre*. Os polinômios ortogonais de *Laguerre* são obtidos através do produto escalar:

$$
(f, g) = \int_0^{+\infty} e^{-x} f(x)g(x)dx
$$

Iremos calcular a integral $\int_0^\infty e^{-x}x^4dx$ utilizando um polinômio de grau três, onde os valores de x_i e A_i são dados por:

$$
\begin{aligned}
x_0 &= 0.4157745567 & A_0 &= 0.7110930099 \\
x_1 &= 2.294280360 & A_1 &= 0.2785177335 \\
x_2 &= 6.289945082 & A_2 &= 0.01038925650
\end{aligned}
$$

Logo,

$$
\begin{aligned}
\int_0^{\infty} e^{-x}x^4dx &= A_0f(x_0) + A_1f(x_1) + A_2f(x_2) \\
&= 0.7110930099 \times (0.4157745567)^4 + 0.2785177335 \times (2.294280360)^4 \\
&+ 0.01038925650 \times (6.289945082)^4 \\
&= 0.0212499565433 + 6.8806059301 + 16.2619223537 \\
&= 23.1637782403
\end{aligned}
$$

Lista de Exercícios 6

0.12 Exercício 1

a) Dado o problema de valor inicial abaixo, queremos encontrar uma aproximação para $y(5)$ usando o método de Euler melhorado.

$$
\begin{cases}
y' & = & 4 - 2x \\
y(0) & = & 2
\end{cases}
$$

Dado o método de Euler melhorado:

$$
y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2)
$$

onde,

$$
\begin{aligned}
k_1 &= f(x_n, y_n) \\
k_2 &= f(x_n + h, k_1 \times h + y_n))
\end{aligned}
$$

Através do seguinte algoritmo em *Python*:

```
# -*- coding: utf-8 -*-
from odemethods import Euler

def f(x, y):
    return 4 - 2*x

# Cálculo de resultados
res_pvi = Euler(f, 0.5, 0, 2, 10)
res_pvi.calc()
with open("output/ex01-2_h05.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 8: Cálculo do P.V.I em *Python* com h = 0.5

```
# -*- coding: utf-8 -*-
from odemethods import Euler

def f(x, y):
    return 4 - 2*x

# Cálculo de resultados
res_pvi = Euler(f, 0.25, 0, 2, 20)
res_pvi.calc()
with open("output/ex01-2_h025.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 9: Cálculo do P.V.I em *Python* com h = 0.25

```
# -*- coding: utf-8 -*-
from odemethods import Euler

def f(x, y):
    return 4 - 2*x

# Cálculo de resultados
res_pvi = Euler(f, 0.125, 0, 2, 40)
res_pvi.calc()
with open("output/ex01-2_h0125.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 10: Cálculo do P.V.I em *Python* com h = 0.125

Obtemos os resultados para h = 0.5, h = 0.25 e h = 0.125 respectivamente, como segue:

Resultado

```
Resolução de P.V.I pelo método de Euler melhorado
Valor inicial: y(0) = 2
Tamanho do passo: 0.5
Resolução:
y(0) = 2
y(0.5) = 3.75
y(1.0) = 5.0
y(1.5) = 5.75
y(2.0) = 6.0
y(2.5) = 5.75
y(3.0) = 5.0
y(3.5) = 3.75
y(4.0) = 2.0
y(4.5) = -0.25
y(5.0) = -3.0
```

Resultado

```
Resolução de P.V.I pelo método de Euler melhorado
Valor inicial: y(0) = 2
Tamanho do passo: 0.25
Resolução:
y(0) = 2
y(0.25) = 2.9375
y(0.5) = 3.75
y(0.75) = 4.4375
y(1.0) = 5.0
y(1.25) = 5.4375
y(1.5) = 5.75
y(1.75) = 5.9375
y(2.0) = 6.0
y(2.25) = 5.9375
y(2.5) = 5.75
y(2.75) = 5.4375
y(3.0) = 5.0
y(3.25) = 4.4375
y(3.5) = 3.75
y(3.75) = 2.9375
y(4.0) = 2.0
y(4.25) = 0.9375
y(4.5) = -0.25
y(4.75) = -1.5625
y(5.0) = -3.0
```

Resultado

```
Resolução de P.V.I pelo método de Euler melhorado
Valor inicial: y(0) = 2
```

```
Tamanho do passo: 0.125
Resolução:
y(0) = 2
y(0.125) = 2.484375
y(0.25) = 2.9375
y(0.375) = 3.359375
y(0.5) = 3.75
y(0.625) = 4.109375
y(0.75) = 4.4375
y(0.875) = 4.734375
y(1.0) = 5.0
y(1.125) = 5.234375
y(1.25) = 5.4375
y(1.375) = 5.609375
y(1.5) = 5.75
y(1.625) = 5.859375
y(1.75) = 5.9375
y(1.875) = 5.984375
y(2.0) = 6.0
y(2.125) = 5.984375
y(2.25) = 5.9375
y(2.375) = 5.859375
y(2.5) = 5.75
y(2.625) = 5.609375
y(2.75) = 5.4375
y(2.875) = 5.234375
y(3.0) = 5.0
y(3.125) = 4.734375
y(3.25) = 4.4375
y(3.375) = 4.109375
y(3.5) = 3.75
y(3.625) = 3.359375
y(3.75) = 2.9375
y(3.875) = 2.484375
y(4.0) = 2.0
y(4.125) = 1.484375
y(4.25) = 0.9375
y(4.375) = 0.359375
y(4.5) = -0.25
y(4.625) = -0.890625
y(4.75) = -1.5625
y(4.875) = -2.265625
y(5.0) = -3.0
```

Logo a resolução para o P.V.I em $y(5)$ é igual a -3.

b) A equação diferencial é do tipo separável, portanto podemos obter a solução geral como se segue:

$$\frac{dy}{dx} = 4 - 2x$$
$$\int dy = \int 4 - 2x dx$$
$$y = 4x - x^2 + C$$

$$C = x^2 - 4x + y \implies C = (0)^2 - 4(0) + 2 = 2$$

Portanto a solução exata para o P.V.I. pode ser obtida:

$$y(5) = 4(5) - (5)^2 + 2 = -3$$

Exatamente o mesmo valor obtido nos algoritmos com os espaçamentos 0.5, 0.25 e 0.125
.

0.13 Exercício 2

Dado o P.V.I. abaixo, estamos interessados em calcular uma aproximação para y(16), por Runge-Kutta de segunda ordem e Runge-Kutta de quarta ordem.

$$\begin{cases} y' & = & \frac{-x}{y} \\ y(0) & = & 20 \end{cases}$$

a) Por Runge-Kutta de segunda ordem, aplicamos:

```
# -*- coding: utf-8 -*-
from odemethods import Euler

def f(x, y):
    return -x/y

# Cálculo e exibição de resultados
res_pvi = Euler(f, 2.0, 0, 20, 8)
res_pvi.calc()
with open("output/ex02a-2_h2.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 11: Cálculo do P.V.I em *Python* com h = 2

Resultado

```
Resolução de P.V.I pelo método de Euler melhorado
Valor inicial: y(0) = 20
```

```
Tamanho do passo: 2.0
Resolução:
y(0) = 20
y(2.0) = 19.9
y(4.0) = 19.5964414429
y(6.0) = 19.0796306414
y(8.0) = 18.3315709692
y(10.0) = 17.3223870103
y(12.0) = 16.0028838939
y(14.0) = 14.2877117956
y(16.0) = 12.009988779
```

```python
# -*- coding: utf-8 -*-
from odemethods import Euler

def f(x, y):
    return -x/y

# Cálculo e exibição de resultados
res_pvi = Euler(f, 1.0, 0, 20, 16)
res_pvi.calc()
with open("output/ex02a-2_h1.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 12: Cálculo do P.V.I em *Python* com h = 1

Resultado

```
Resolução de P.V.I pelo método de Euler melhorado
Valor inicial: y(0) = 20
Tamanho do passo: 1.0
Resolução:
y(0) = 20
y(1.0) = 19.975
y(2.0) = 19.8997803475
y(3.0) = 19.7737681933
y(4.0) = 19.5959839791
y(5.0) = 19.3650021812
y(6.0) = 19.0788911792
y(7.0) = 18.7351259946
y(8.0) = 18.3304639613
y(9.0) = 17.8607674966
y(10.0) = 17.3207482613
y(11.0) = 16.7035896045
y(12.0) = 16.000371923
y(13.0) = 15.1991620136
y(14.0) = 14.2834928887
y(15.0) = 13.2296485066
y(16.0) = 12.0013551483
```

b) O método de Runge Kutta de quarta ordem é dado por:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2(k_2 + k_3) + k_4)$$

onde,

$$\begin{aligned}
k_1 &= f(x_n, y_n) \\
k_2 &= f(x_n + \frac{1}{2}h,\ \frac{1}{2}hk_1 + y_n)) \\
k_3 &= f(x_n + \frac{1}{2}h,\ \frac{1}{2}hk_2 + y_n)) \\
k_4 &= f(x_n + h,\ y_n + hk_3)
\end{aligned}$$

Por Runge-Kutta de quarta ordem, aplicamos os seguintes algoritmos em *Python*:

```
# -*- coding: utf-8 -*-
from odemethods import RungeKutta

def f(x, y):
    return -x/y

# Cálculo e exibição de resultados
res_pvi = RungeKutta(f, 4.0, 0, 20, 4)
res_pvi.calc()
with open("output/ex02b-2_h4.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 13: Cálculo do P.V.I. com h = 4

```
# -*- coding: utf-8 -*-
from odemethods import RungeKutta

def f(x, y):
    return -x/y

# Cálculo e exibição de resultados
res_pvi = RungeKutta(f, 2.0, 0, 20, 8)
res_pvi.calc()
with open("output/ex02b-2_h2.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 14: Cálculo do P.V.I. com h = 2

Onde obtemos:

Resultado

```
Resolução de P.V.I pelo método de Runge Kutta de quarta ordem
Valor inicial: y(0) = 20
Tamanho do passo: 4.0
Resolução:
y(0) = 20
y(4.0) = 19.5959040578
y(8.0) = 18.3302295412
y(12.0) = 15.9996958854
y(16.0) = 11.9980017253
```

Resultado

```
Resolução de P.V.I pelo método de Runge Kutta de quarta ordem
Valor inicial: y(0) = 20
Tamanho do passo: 2.0
Resolução:
y(0) = 20
y(2.0) = 19.8997485317
y(4.0) = 19.595917044
y(6.0) = 19.0787817434
y(8.0) = 18.3302978644
y(10.0) = 17.3204979319
y(12.0) = 15.9999781927
y(14.0) = 14.2828033184
y(16.0) = 11.9998242097
```

0.14 Exercício 3

Dado o P.V.I:

$$\begin{cases} y^{'} & = & yx^2 - y \\ y(0) & = & 1 \end{cases}$$

Cuja solução analítica é:

$$
\begin{aligned}
\frac{dy}{dx} &= yx^2 - y \\
\frac{dy}{dx} &= y(x^2 - 1) \\
\frac{dy}{y} &= (x^2 - 1)dx \\
\int \frac{dy}{y} &= \int (x^2 - 1)dx \\
\ln(y) &= \frac{1}{3}x^3 - x + C \\
e^{\ln(y)} &= e^{\frac{1}{3}x^3 - x} + C \\
y &= e^{\frac{1}{3}x^3 - x} + C
\end{aligned}
$$

$$
C = y - e^{\frac{1}{3}x^3 - x} + C \implies C = 1 - e^0 = 0
$$

a) Podemos calcular a solução aproximada pelo método de Euler, ou seja, pelo método de Taylor de ordem 1, onde:

$$
y_{n+1} = y_n + h \times f_n
$$

$$
f_n = yx^2 - y
$$

Assim, ao passo de h = 0.25 em [0, 2] temos:

$n = 0$

$$
\begin{aligned}
y_1 &= y_0 + 0.25(f_0) \\
&= 1 + 0.25[(1)(0) + 2(1)(0) - (1)] \\
&= 0.75
\end{aligned}
$$

$n = 1$

$$
\begin{aligned}
y_2 &= y_1 + 0.25(f_1) \\
&= 0.75 + 0.25((0.75)(0.0625) - (0.75) \\
&= 0.57421875
\end{aligned}
$$

...

Calculemos todas as iterações através do seguinte algoritmo:

```
# -*- coding: utf-8 -*-
from odemethods import Taylor

def f(x, y):
    return y*(x**2) - y

functions = [f]
# Cálculo de resultados
res_pvi = Taylor(functions, 0.25, 0, 1, 8)
res_pvi.calc()
with open("output/ex03a-2_h025.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 15: Cálculo do P.V.I. com h = 0.25

Onde obtemos:

Resultado

```
Resolução de P.V.I pelo método de Taylor de ordem 1
Valor inicial: y(0) = 1
Tamanho do passo: 0.25
Resolução:
y(0) = 1
y(0.25) = 0.75
y(0.5) = 0.57421875
y(0.75) = 0.466552734375
y(1.0) = 0.415523529053
y(1.25) = 0.415523529053
y(1.5) = 0.473956525326
y(1.75) = 0.62206793949
y(2.0) = 0.94282172079
```

b) A solução aproximada pelo método de Euler melhorado, com h = 0.25 em [0, 2] é obtida executando:

```python
# -*- coding: utf-8 -*-
from odemethods import Euler

def f(x, y):
    return y*(x**2 - 1)

# Cálculo de resultados
res_pvi = Euler(f, 0.25, 0, 1, 8)
res_pvi.calc()
with open("output/ex03b-2_h025.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 16: Cálculo do P.V.I. com h = 0.25

Onde obtemos:

Resultado

```
Resolução de P.V.I pelo método de Euler melhorado
Valor inicial: y(0) = 1
Tamanho do passo: 0.25
Resolução:
y(0) = 1
y(0.25) = 0.787109375
y(0.5) = 0.638373374939
y(0.75) = 0.550160647836
y(1.0) = 0.520073737407
y(1.25) = 0.556641422069
y(1.5) = 0.694986384878
y(1.75) = 1.03874674029
y(2.0) = 1.89693008236
```

c) Pelo método de Runge Kutta de quarta ordem, temos:

```python
# -*- coding: utf-8 -*-
from odemethods import RungeKutta

def f(x, y):
    return y*(x**2 - 1)

# Cálculo de resultados
res_pvi = RungeKutta(f, 0.25, 0, 1, 8)
res_pvi.calc()
with open("output/ex03c-2_h025.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 17: Cálculo do P.V.I. com h = 0.25

Onde obtemos:

Resultado

```
Resolução de P.V.I pelo método de Runge Kutta de quarta ordem
Valor inicial: y(0) = 1
Tamanho do passo: 0.25
Resolução:
y(0) = 1
y(0.25) = 0.782872257133
y(0.5) = 0.632341022435
y(0.75) = 0.543693095419
y(1.0) = 0.513418828908
y(1.25) = 0.54938452231
y(1.5) = 0.687279450803
y(1.75) = 1.03699845284
y(2.0) = 1.94631881037
```

d)

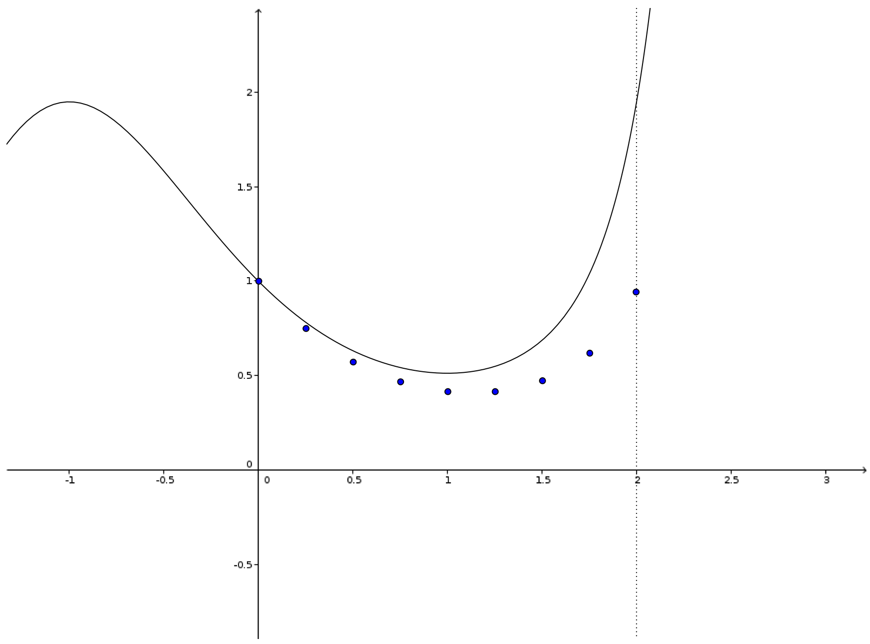

Figura 0.3: Pontos obtidos no método de Taylor de grau 1 e solução exata

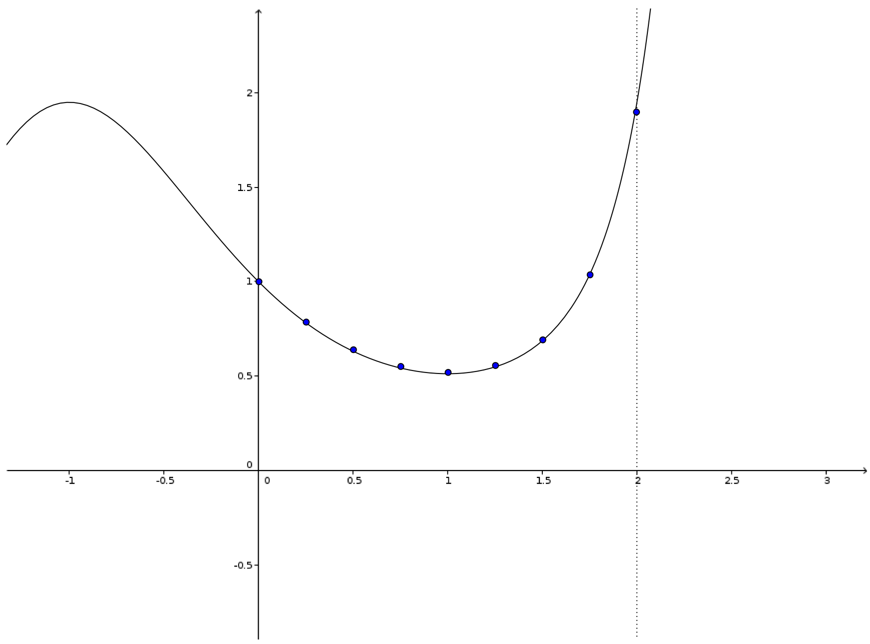

Figura 0.4: Pontos obtidos no método de Euler melhorado e solução exata

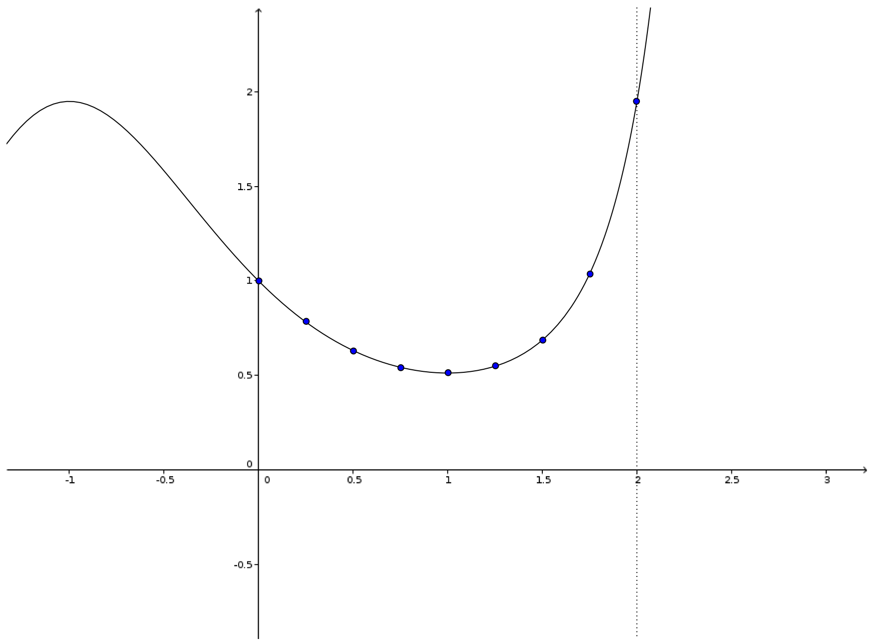

Figura 0.5: Pontos obtidos no método de Runge Kutta de quarta ordem e solução exata

0.15 Exercício 4

Queremos encontrar a fórmula de Taylor de ordem 2 para o P.V.I.

$$\begin{cases} y' + y & = & x \\ y(0) & = & 0 \end{cases}$$

com h = 0.1

A fórmula de Taylor de segunda ordem é da forma:

$$y_{n+1} = y_n + (h \times f_n) + (\frac{h^2}{2} \times f'_n)$$

onde,

$$\begin{aligned} f_n &= x - y \\ f'_n &= 1 - y' \\ &= 1 - x + y \end{aligned}$$

Portanto,

$$y_{n+1} = y_n + (0.1 \times (x_n - y_n)) + (0.05 \times (1 - x_n + y_n))$$

b) De fato $y(x) = e^{-x} + x - 1$ é solução do P.V.I. pois:

$$y(0) = e^0 + 0 - 1 = 0$$

e,

$$y' = -e^{-x} + 1 \implies (-e^{-x} + 1) + e^{-x} + x - 1 = x$$

0.16 Exercício 5

Estamos interessados em resolver o P.V.I.

$$\begin{cases} xy' - x^2y - 2 & = & 0 \\ y(1) & = & 3 \end{cases}$$

$\forall x \in [1, 2]$

E encontrar uma solução aproximada em $x = 1.5$, através do método de Euler melhorado e o Runge Kutta de quarta ordem.

Assim, definimos um intervalo h para que a solução em 1.5 seja definida, e isolamos a função para utilizar os métodos como segue:

$$y' = \frac{2 + x^2y}{x}$$

Executando os códigos abaixo:

```
# -*- coding: utf-8 -*-
from odemethods import Euler

def f(x, y):
    return (2 + (x**2)*y)/x

# Cálculo de resultados
res_pvi = Euler(f, 0.1, 1, 3, 5)
res_pvi.calc()
with open("output/ex05-2-euler.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 18: Cálculo do P.V.I. com h = 0.25

```
# -*- coding: utf-8 -*-
from odemethods import RungeKutta

def f(x, y):
    return (2 + (x**2)*y)/x

# Cálculo de resultados
res_pvi = RungeKutta(f, 0.1, 1, 3, 5)
res_pvi.calc()
```

```
with open("output/ex05-2-rt.txt", "w") as outputFile:
    print >> outputFile, res_pvi
```

Código 19: Cálculo do P.V.I. com h = 0.25

Obtemos como soluções:

Resultado

```
Resolução de P.V.I pelo método de Euler melhorado
Valor inicial: y(1) = 3
Tamanho do passo: 0.1
Resolução:
y(1) = 3
y(1.1) = 3.53340909091
y(1.2) = 4.14822315152
y(1.3) = 4.87019692963
y(1.4) = 5.73111318631
y(1.5) = 6.77111081059
```

Resultado

```
Resolução de P.V.I pelo método de Runge Kutta de quarta ordem
Valor inicial: y(1) = 3
Tamanho do passo: 0.1
Resolução:
y(1) = 3
y(1.1) = 3.5334545489
y(1.2) = 4.14881983008
y(1.3) = 4.87203310627
y(1.4) = 5.73517013633
y(1.5) = 6.77881918432
```