数値解析 第 2 回レポート

(1)、(5)を選択した。使用した計算環境は

CPU: Intel Core i5, メモリ: 8 GB, OS: macOS, プログラミング言語: Python 3.8.5

である。

(1)

Gauss-Legendre 数値積分が$(2n-1)$次多項式まで真値を与える性質が、一般の直交多項式に基づく Gauss 型公式においても成立することを示す。

$\{A_n\}$が区間(a,b)、重み$w(x)$で、$(A_m, A_n)_w = 0 \ \ (m \neq n)$、$\deg A_n = n \ \ (n = 0,1,2,\cdots)$なる直交多項式系を成すとする。このとき、任意の高々$2n-1$次多項式$f(x)$は、

$$f(x) = A_n(x)Q(x) + R(x) \ \ (\deg Q, \deg R \leq n-1)$$

と表せる。ここで積分Iを、$I := \int_a^b f(x)w(x)dx$で定義すると、

$$\begin{aligned} I &= \int_a^b f(x)w(x)dx \\ &= \int_a^b A_n(x)Q(x)w(x)dx + \int_a^b R(x)w(x)dx \\ &= \int_a^b \sum_{k=0}^{n-1} q_k A_n(x)A_k(x)\, w(x)dx + \int_a^b R(x)w(x)dx \\ &= \sum_{k=0}^{n-1} q_k \,(A_n, A_n)_w + \int_a^b R(x)w(x)dx \\ &= \int_a^b R(x)w(x)dx \end{aligned}$$

となる。ここで、$Q(x)$は高々$n-1$次の多項式だから、$Q(x) = \sum_{k=0}^{n-1} q_k\, A_k(x)$と展開できること、任意の$k \leq n-1)$について、$(A_n, A_n)_w = 0$となることを用いた。

次に Gauss-Legendre 公式を両辺に適用する。$\{x_k\}_{k=1}^n$を$A_n(x)$の零点として、$f(x)$を多項式補間により、$f_n(x) = Q'_n(x) + R_n(x)$と高々$n-1$次の多項式で近似する。このとき、

$$Q'_n(x) = \sum_{k=1}^{n} A_n(x_k)Q(x_k)l_k(x)$$

$$R_n(x) = \sum_{k=1}^{n} R(x_k)\, l_k(x)$$

と表す。ここで積分I_nを$I_n := \int_a^b f_n(x)w(x)dx = \sum_{k=1}^n w_k f(x_k)$ $(w_k := \int_a^b l_n(x)w(x)dx)$と定義すると、

$$I_n = \sum_{k=1}^{n} w_k f(x_k)$$

$$= \int_a^b \sum_{k=1}^{n} A_k(x_k) Q(x_k)\, l_n(x) w(x) dx + \int_a^b \sum_{k=1}^{n} R_k(x_k)\, l_n(x) w(x) dx$$
$$= \int_a^b R(x) w(x) dx$$

ここで$\{x_k\}_{k=1}^n$が$A_n(x)$ の零点であることを利用した。

以上により$I = I_n$ と分かり、示された。

(5)

(i)

Dahlquist のテスト方程式

$$\frac{dy}{dt} = cy$$

に BDF を適用した時の数値解について考える。

(i)(I)

付録に添付したコードを用いて次数の計算を行なった。第 4 章のスライドより、次数は、分点数を10倍した時に解析解との誤差を10^{-k}倍と表した場合の指数kであることから、両者を対数スケールのグラフで表した時の傾きにより計算することにした。

まず$n\ (= \frac{et}{\Delta t})$（$et$：最終時刻）に対する挙動を見るために、Dahlquist のテスト方程式の係数cを$c = -2$ とし、n を縦軸、誤差（error）の絶対値を横軸としたグラフを作成すると図 1 のようになった。図 1 を見るとどの BDF 公式の次数についても、n を大きくしていくとある時から誤差の減少が緩やかになってしまうことが分かる。これはn を大きくすることによりy の値の更新量が小さくなり、丸め誤差が発生してしまうからだと考えた。また、いくつかのc について実験したところ、c の実部を負としてその絶対値がある程度小さい場合には、最終的な誤差が同じようなオーダーになってしまっていた。

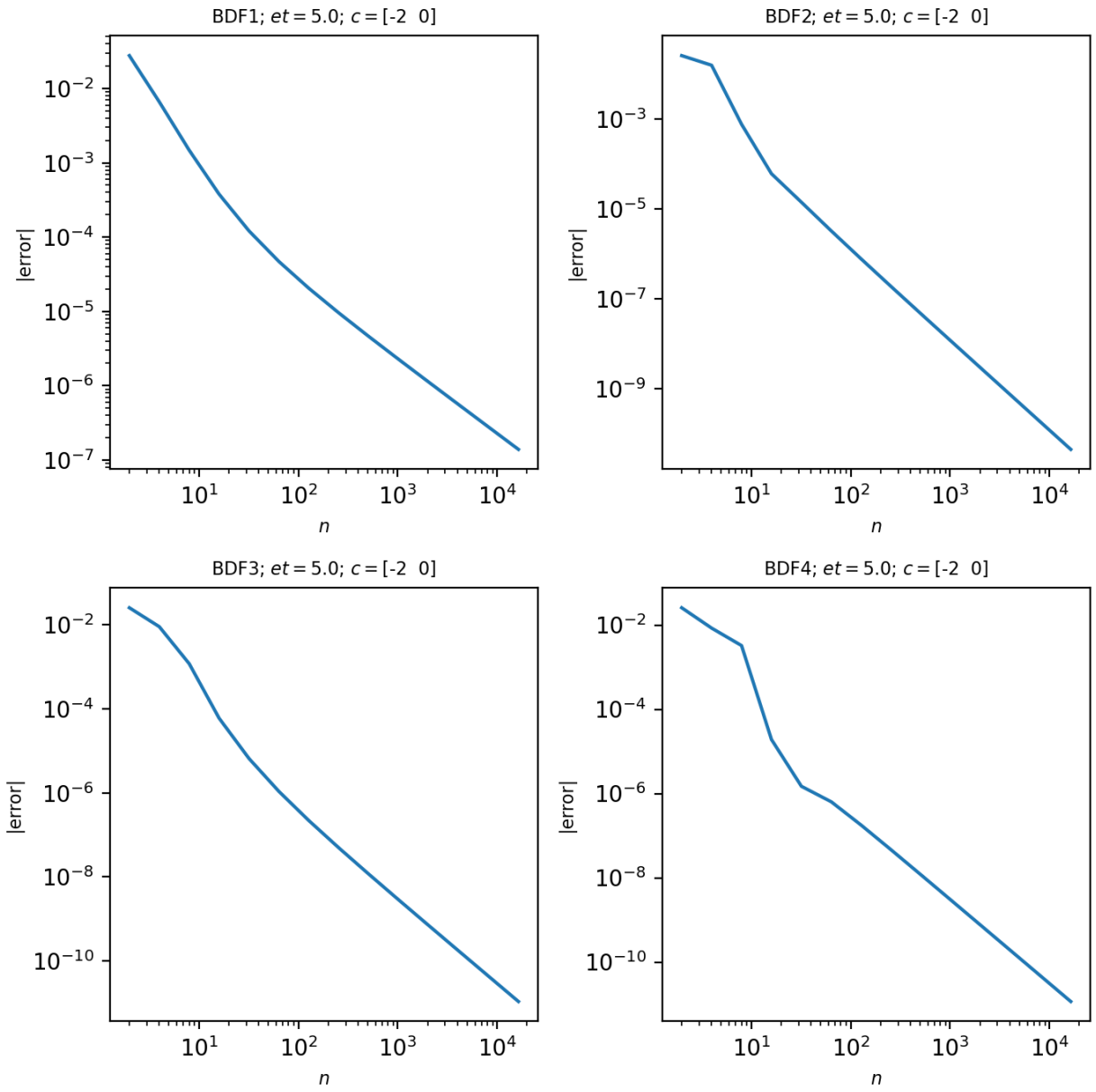

図 1. $c = -2$ の時の誤差のn に対する挙動。初期値を 1.0、最終時刻を$et = 5.0$とし、c は$c = x + yi$ の時 $[x, y]$ と表した。誤差は数値解と真値との差のノルム（$||(y^{(m)}) - y(t_m)||$)で定義した（以下同様）。

このような事情を踏まえ、図 1 と同様のグラフ中で最も改善が大きかったプロット間の傾きを調べることにし、これを最大次数とした。最大次数をいくつかの絶対値がある程度小さいcについて調べ、表にまとめると表 2 のようになった。ここでは、参考文献[1] に記載のある絶対安定領域に注意して、負の実数（負の実軸上は絶対安定）、$-2 + i$（これと原点を結ぶ線分上はどの BDF の次数についてもおそらく絶対安定）をc として選んだ。また、初期値を 1.0、最終時刻を$et = 5.0$とした。

表 2. 係数c を変化させた時の最大次数（文中で定義した）

c	BDF2	BDF3	BDF4
-0.25	2.3074205234449296	2.073915486121502	2.0010543487444936
-0.5	3.47276049340734	2.4439745628298035	2.174324813123576
-1	2.463287879042629	3.210508884665157	2.4419263972784395
-2	4.361077411320418	4.314062549601695	7.40628753741948
$-2+i$	3.4805593409754314	6.0136202927888736	5.926519035192854
-4	8.503718111623169	8.666008665249548	13.104895391486306

表 2 より、c 実部の絶対値がある程度大きければ、（最大次数については）各 BDF の次数程度の精度は保証されていると分かる。しかし、c の実部の絶対値の値に応じて傾向にばらつきが出てしまい、これだけでは各 BDF の精度について系統的な判断は難しいと言える結果となった。また、c 実部の絶対値が大きくなるにつれて最大次数が大きくなる傾向があったが、ここから次数について様々なc について同じような値を持たせられるような定義や、c 毎に得られた次数について適切な統計処理があり得ると考えたが、難しかった。

(i)(II)

次に実際に安定領域が広いかを数値的に検証する。最終時刻を$et = 20.0$とし、c 、$c\Delta$t を変化させ、2, 4 次の BDF を適用したときのy の挙動を調べると以下の図 3〜図 5 のようになった。

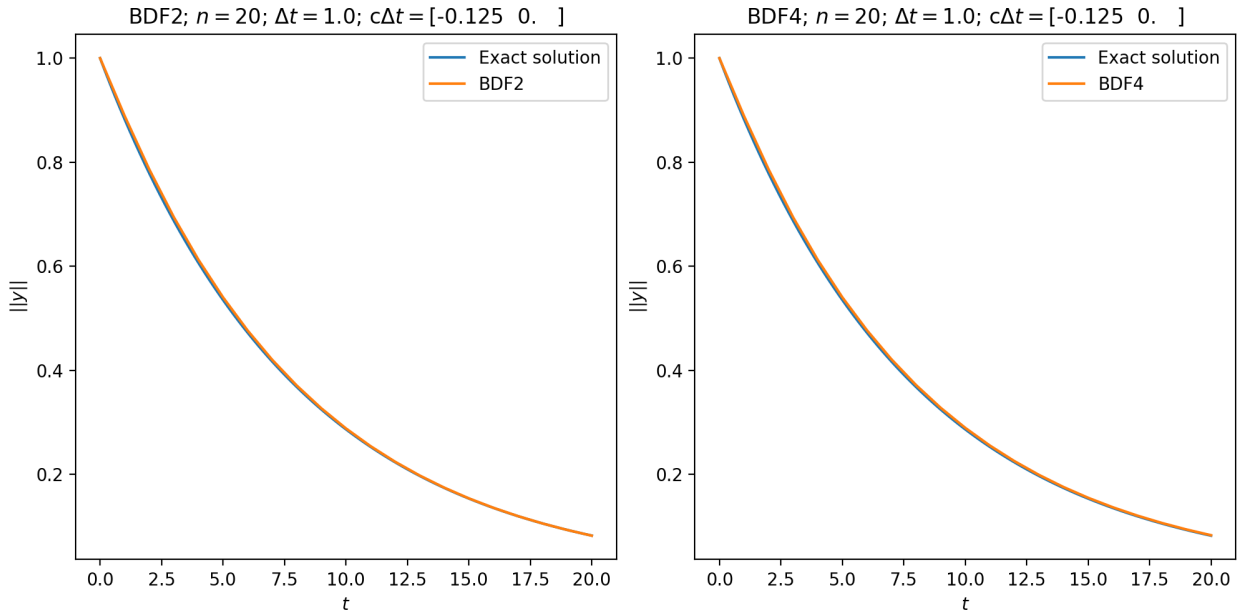

図 3. $c = -0.125$、$n = 20$ としたときの数値解の挙動

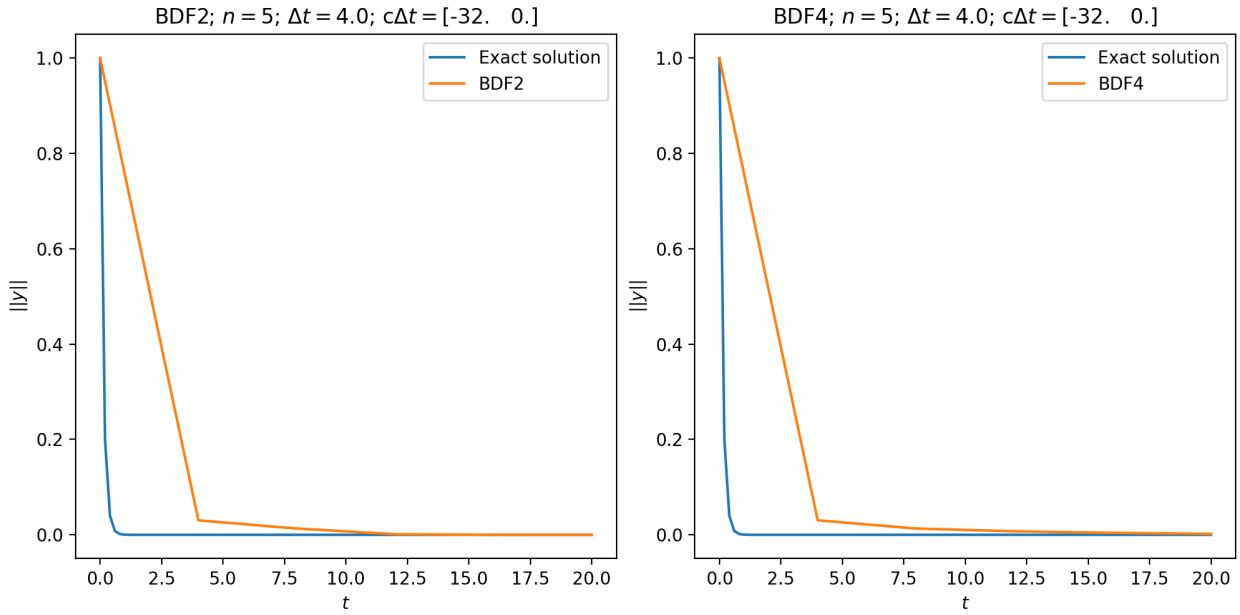

図 4. $c = -8$、$n = 5$ としたときの数値解の挙動

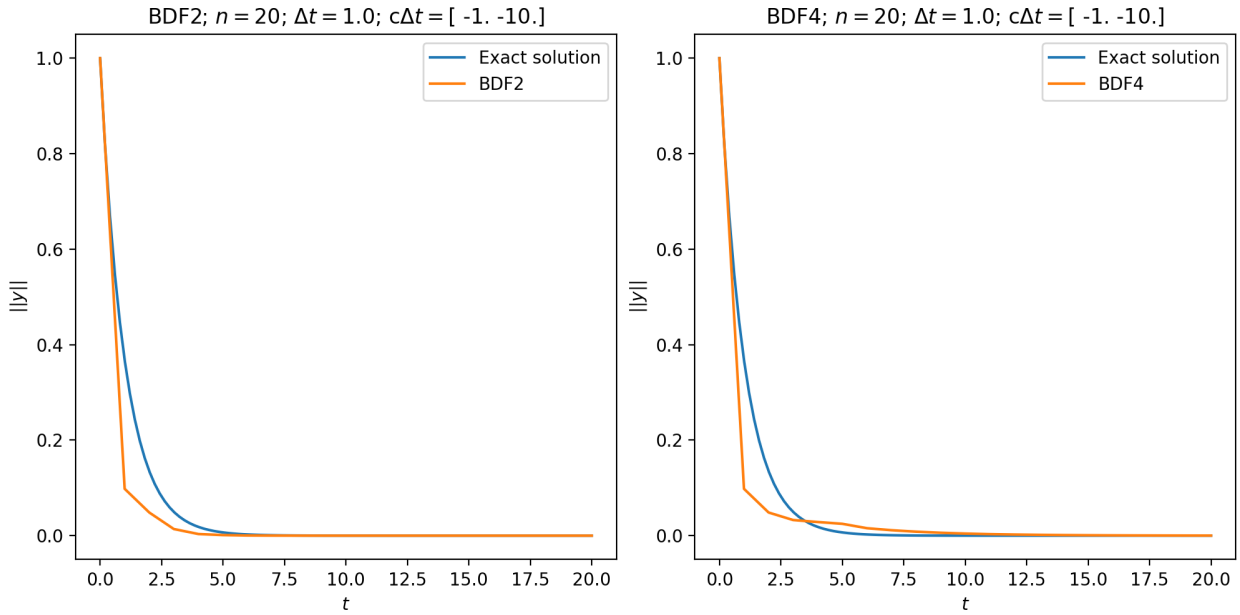

図 5. $c = -1 - 10i$、$n = 20$ としたときの数値解の挙動

図 3〜図 5 の 3 つの例と先の(I)で誤差を求めたものを考えるといずれも数値解は明らかに有界であり、実際に左半平面の広い部分が安定領域となっていることが確かめられた。

(ii)

stiff な微分方程式

$$\frac{dy}{dt} = -50(y - \cos(t)), \quad y(0) = 0 \cdots (*)$$

に陽的 Euler 法、4 次の BDF を適用すると、挙動は以下の図 6,7 のようになった。

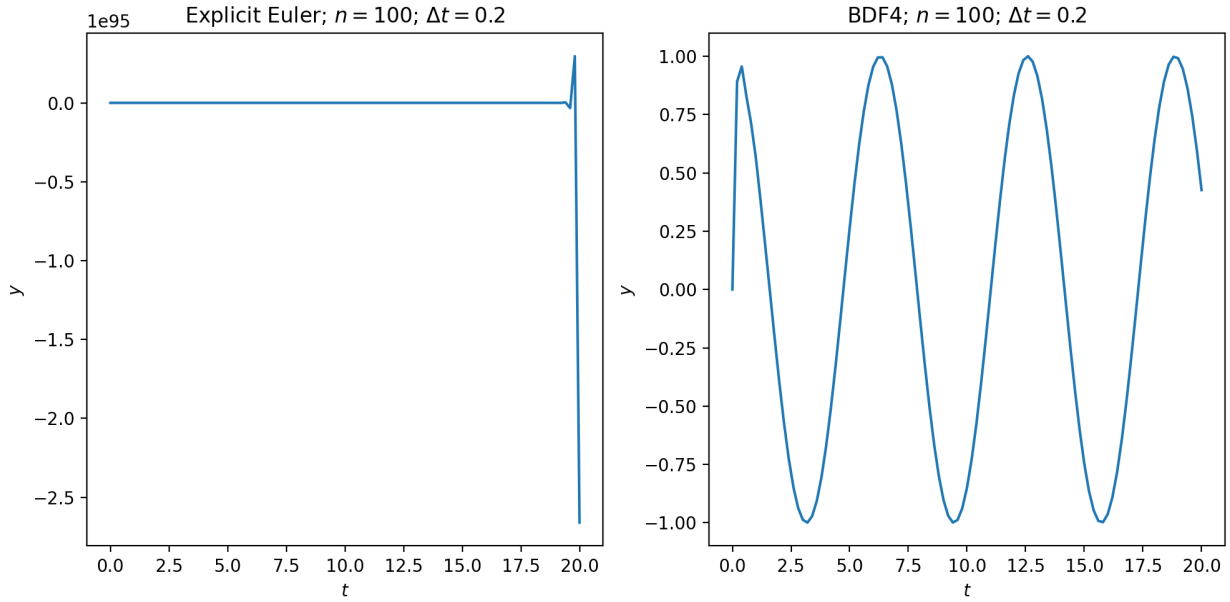

図 6. 微分方程式$(*)$を$n = 100$、$et = 20.0$ として数値的に解いた時の解の挙動（左：陽的 Euler 法、右：BDF4）。陽的 Euler 法については最終時刻での値が大きいため途中の時刻での挙動が分かりにくいが、振動しながら発散している。

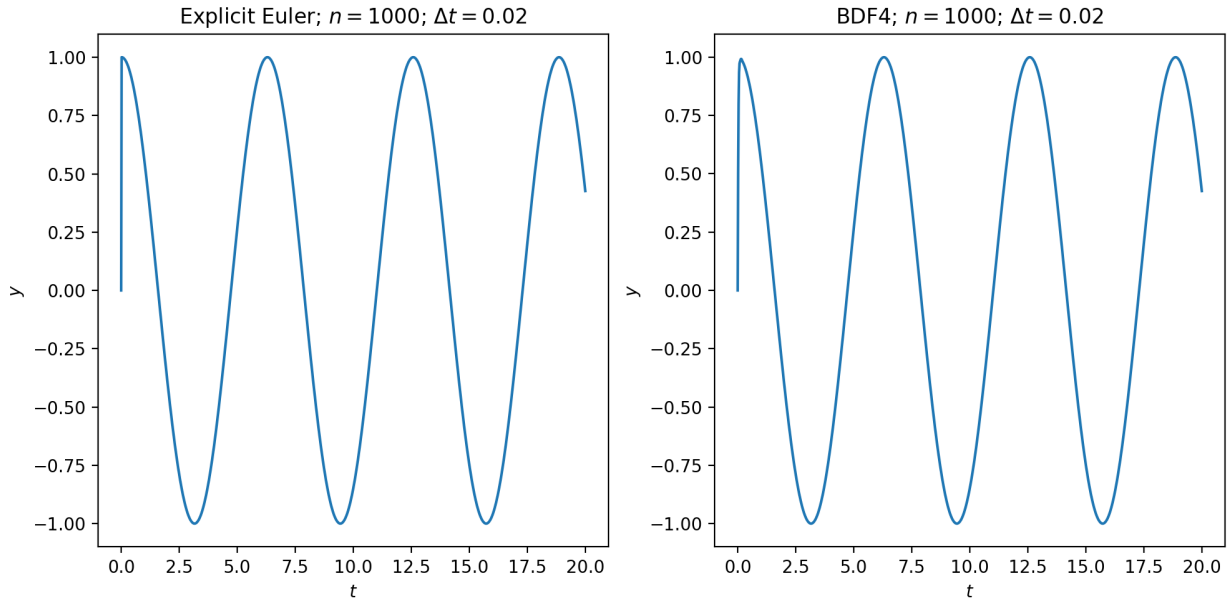

図 7. 微分方程式$(*)$を$n = 1000$、$et = 20.0$ として数値的に解いた時の解の挙動（左：陽的 Euler 法、右：BDF4）

このような数値解がこのような挙動をした理由について考察する。まず、陽的 Euler 法については、$(*)$より、

$$y((m+1)\Delta t) - y(m\Delta t) = -50\Delta t(y(m\Delta t) - \cos(m\Delta t))$$

という漸化式を得るが、この時$n = 100$では$-50\Delta t = -10$となっており安定領域から大きく外れてしまう。実際には最初の 1 ステップで$\cos(t)$が無視できる状態となるので、発散すると分かる。一方$n = 1000$では$-50\Delta t = -1$となり、ちょうど安定領域の中心に来ている。このとき$y((m+1)\Delta t) = \cos(m\Delta t)$となるので発散しないと分かる。

次に、BDF について考察する。上の例では示していないものの、BDF1、つまり陰的 Euler 法も$n = 100$で発散せず同じように振動することが分かるので（補足、図 8）、これを使って考えることにする。陰的 Euler 法の場合は、$(*)$より

$$y\big((m+1)\Delta t\big) = \frac{1}{50\Delta t + 1} y(m\Delta t) + \frac{50\Delta t}{50\Delta t + 1}\cos((m+1)\Delta t))$$

が得られ、$y\big((m+1)\Delta t\big)$は$y(m\Delta t)$と$\cos(m\Delta t))$に一定の重率をかけて足し合わせたものと分かる。これにより、yの更新量が抑えられるため、発散していないと考えられる。また、安定領域の面だけ考えれば BDF4 も負の実軸を安定領域として含むため、陰的 Euler 法が発散せず動いていることから BDF4 がきちんと動いていると思われる。

(iii)

どの公式の次数kについても、BDF では$f(y,t)$を$(y^{(m+1)}, t^{(m+1)})$について一度だけしか使わないことから、計算量が次数kにあまり依らないと考えられる。

補足

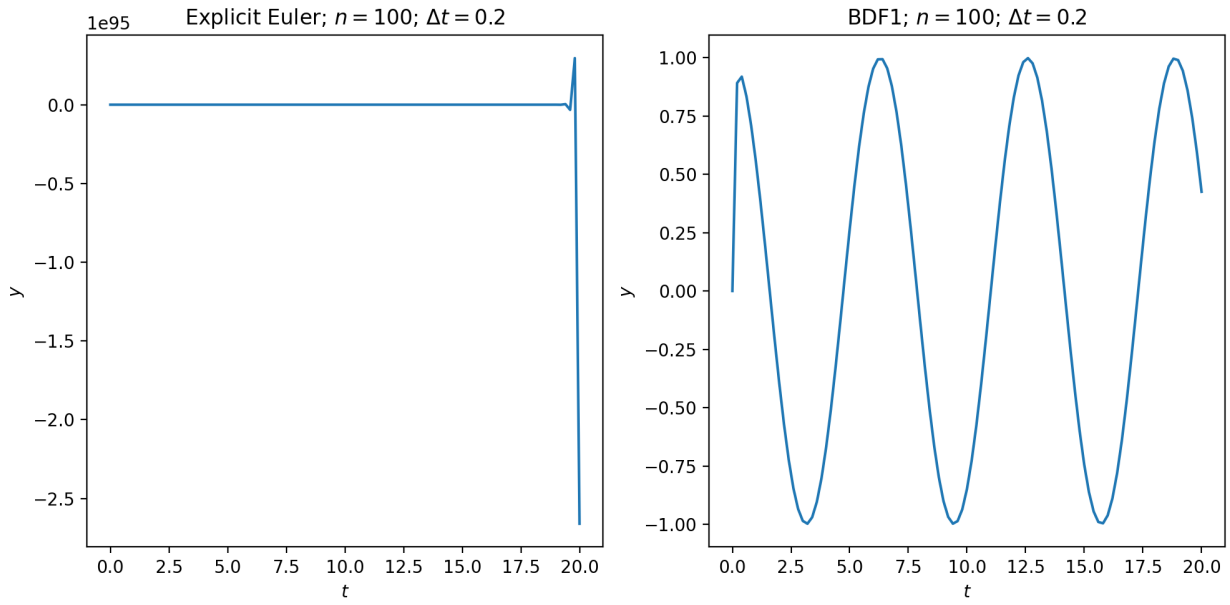

図 8. 微分方程式$(*)$を$n = 100$、$et = 20.0$ として数値的に解いた時の解の挙動（左：陽的 Euler 法、右：BDF1）

参考文献

[1] 後退微分法. Wikipedia. http://ja.wikipedia.org/w/index.php?curid=353180, (参照 2020-1-12).

付録（コード）

np.optimize.root を用いて非線形方程式の根を求めることにしたが、この際複素数を使えなかったため、サポートされている np.ndarray を用いて複素数を $x + yi = \mathrm{np.ndarray}([x, y])$ として表し、テスト方程式について複素数も対応できるようにした。以下に全体のコードを示す。

・メインのコード

```
# -*- coding: utf-8 -*-
import numpy as np
from scipy.optimize import root
import matplotlib.pyplot as plt

#parameters
initf = np.array([1.0,0]) #Dahlquistのテスト方程式用の初期値
initg = np.array([0,0]) #(ii)のODE用の初期値
c = np.array([12,0]) #ahlquistのテスト方程式の係数
et = 20.0 #最終時刻
n = 100 #間隔の数
dt = et/n
tv = np.linspace(0,et,n+1) #時間のグリッド
```

```
def f(a): #Dahlquistのテスト方程式
    x0,x1 = a[0][0],a[0][1] #複素数を(x0,x1):=x0+x1iで表現
    return np.array([c[0]*x0-c[1]*x1,c[0]*x1+c[1]*x0])

def g(a): #(ii)のstiffな微分方程式
    x0,t = a[0][0], a[1]
    return np.array([-50*(x0 - np.cos(t)),0])

def BDF(f,init,mtd): #BDFの準備
    def BDF1(y,y0,t): #1次
        a = [y,t]
        return y - y0 - dt*f(a)

    def BDF2(y,y0,y1,t): #2次
        a = [y,t]
        return 3/2*y - 2*y0 + 1/2*y1 - dt*f(a)

    def BDF3(y,y0,y1,y2,t): #3次
        a = [y,t]
        return 11/6*y - 3*y0 + 3/2*y1 - 1/3*y2 - dt*f(a)

    def BDF4(y,y0,y1,y2,y3,t): #4次
        a = [y,t]
        return 25/12*y - 4*y0 + 3*y1 - 4/3*y2 + 1/4*y3 - dt*f(a)

    y = [init] #解を格納
    mtdtxt = str() #k次のBDFのラベル

    for i in range(n): #BDFの実行
        if  mtd == 1 or i==0:
            mtdtxt = 'BDF1'
            sol = root(lambda x: BDF1(x,y[i],tv[i+1]), x0=y[i])
            y.append(sol.x)

        elif mtd == 2 or i==1:
            mtdtxt = 'BDF2'
            sol = root(lambda x: BDF2(x,y[i],y[i-1],tv[i+1]), x0=y[i])
            y.append(sol.x)

        elif mtd == 3 or i==2:
            mtdtxt = 'BDF3'
            sol = root(lambda x: BDF3(x,y[i],y[i-1],y[i-2],tv[i+1]), x0=y[i])
            y.append(sol.x)

        else:
            mtdtxt = 'BDF4'
            sol = root(lambda x: BDF4(x,y[i],y[i-1],y[i-2],y[i-3],tv[i+1]), x0=y[i])
            y.append(sol.x)

    return y,mtdtxt #解のリストとラベルを出力
```

```
def ExpE(f,init): #陽的Euler法
    lab = 'Explicit Euler'
    y = [init] #解を格納
    for i in range(n):
        y.append(y[i]+dt*f([y[i],tv[i]]))
    return y,lab

tvf = np.linspace(0,et,100) #真値用の時間のリスト
eyf = np.exp(c[0]*tvf)*( #真値
        ((initf[0]*np.cos(c[1]*tvf) - initf[1]*np.sin(c[1]*tvf))**2)+
        (initf[0]*np.sin(c[1]*tvf) + initf[1]*np.cos(c[1]*tvf))**2)**0.5
z = c*dt
```

・グラフや必要な値を取得するためのコード（メインのコードと一緒に用いる）

```
def draw_graph(dat): #引数は、[解,ラベル]（BDF、ExpEの出力）、数値解の挙動を表示
    y,lab = dat[0],dat[1]
    fig,ax = plt.subplots()
    ax.set_xlabel('$t$')
    ax.set_ylabel('$y$')
    ax.set_title(lab+'; $n=$'+str(n)+'; $\Delta t=$'+str(dt)+'; c$\Delta t=$'+str(z))
    ax.plot(tvf,eyf,label='Exact solution')
    ax.plot(tv,[np.linalg.norm(x) for x in y],label=lab)
    #ax.plot(tv,[x[0] for x in y],label=lab)
    ax.legend()
    plt.show()

def fgraph(): #Dahlquistのテスト方程式用のグラフ、BDF2,BDF4をまとめて表示
    y2,lab2 = BDF(f,initf,2)
    y4,lab4 = BDF(f,initf,4)

    fig = plt.figure(figsize=(10,5))
    ax2 = fig.add_subplot(121)
    ax4 = fig.add_subplot(122)

    ax2.set_xlabel('$t$')
    ax2.set_ylabel('$||y||$')
    ax2.set_title(lab2+'; $n=$'+str(n)+'; $\Delta t=$'+str(dt)+'; c$\Delta
t=$'+str(z))
    ax2.plot(tvf,eyf,label='Exact solution')
    ax2.plot(tv,[np.linalg.norm(x) for x in y2],label=lab2)
    ax2.legend()

    ax4.set_xlabel('$t$')
    ax4.set_ylabel('$||y||$')
    ax4.set_title(lab4+'; $n=$'+str(n)+'; $\Delta t=$'+str(dt)+'; c$\Delta
t=$'+str(z))
    ax4.plot(tvf,eyf,label='Exact solution')
    ax4.plot(tv,[np.linalg.norm(x) for x in y4],label=lab4)
    ax4.legend()
```

```
    plt.rcParams["font.size"] = 5
    fig.tight_layout()
    plt.show()

def order_graph(): #次数についての挙動を見るためのグラフ、BDF1~4についてまとめて表示
    global n
    global dt
    global tv
    n = 1
    y1er,y2er,y3er,y4er = [],[],[],[] #誤差を格納
    nv = []
    for i in range(14): #10^4程度まで求める
        n = n*2
        dt = et/n
        tv = np.linspace(0,et,n+1)
        nv.append(n)
        ey = np.exp(c[0]*et)*np.array([initf[0]*np.cos(c[1]*et) -
initf[1]*np.sin(c[1]*et), #真値
        initf[0]*np.sin(c[1]*et) + initf[1]*np.cos(c[1]*et)])
        y1er.append(np.linalg.norm(BDF(f,initf,1)[0][-1] - ey)) #誤差
        y2er.append(np.linalg.norm(BDF(f,initf,2)[0][-1] - ey))
        y3er.append(np.linalg.norm(BDF(f,initf,3)[0][-1] - ey))
        y4er.append(np.linalg.norm(BDF(f,initf,4)[0][-1] - ey))
    fig = plt.figure(figsize=(7,7))
    ax1 = fig.add_subplot(221)
    ax2 = fig.add_subplot(222)
    ax3 = fig.add_subplot(223)
    ax4 = fig.add_subplot(224)

    ax1.set_xlabel('$n$',fontsize=8)
    ax1.set_ylabel('|error|',fontsize=8)
    ax1.set_xscale('log')
    ax1.set_yscale('log')
    ax1.plot(nv,y1er,label="BDF1")
    ax1.set_title('BDF1'+'; $et=$'+str(et)+'; $c=$'+str(c),fontsize=8)

    ax2.set_xlabel('$n$',fontsize=8)
    ax2.set_ylabel('|error|',fontsize=8)
    ax2.set_xscale('log')
    ax2.set_yscale('log')
    ax2.plot(nv,y2er,label="BDF2")
    ax2.set_title('BDF2'+'; $et=$'+str(et)+'; $c=$'+str(c),fontsize=8)

    ax3.set_xlabel('$n$',fontsize=8)
    ax3.set_ylabel('|error|',fontsize=8)
    ax3.set_xscale('log')
    ax3.set_yscale('log')
    ax3.plot(nv,y3er,label="BDF3")
    ax3.set_title('BDF3'+'; $et=$'+str(et)+'; $c=$'+str(c),fontsize=8)

    ax4.set_xlabel('$n$',fontsize=8)
    ax4.set_ylabel('|error|',fontsize=8)
    ax4.set_xscale('log')
```

```
    ax4.set_yscale('log')
    ax4.plot(nv,y4er,label="BDF4")
    ax4.set_title('BDF4'+'; $et=$'+str(et)+'; $c=$'+str(c),fontsize=8)

    plt.rcParams["font.size"] = 8
    fig.tight_layout()
    plt.show()

def order_check(): #誤差の数値を得るための関数
    global n
    global dt
    global tv
    n = 1
    y1er,y2er,y3er,y4er = [],[],[],[] #誤差を格納
    nv = []
    for i in range(14):
        n = n*2
        dt = et/n
        tv = np.linspace(0,et,n+1)
        nv.append(n)
        ey = np.exp(c[0]*et)*np.array([initf[0]*np.cos(c[1]*et) -
initf[1]*np.sin(c[1]*et),
        initf[0]*np.sin(c[1]*et) + initf[1]*np.cos(c[1]*et)])
        y1er.append(np.linalg.norm(BDF(f,initf,1)[0][-1] - ey))
        y2er.append(np.linalg.norm(BDF(f,initf,2)[0][-1] - ey))
        y3er.append(np.linalg.norm(BDF(f,initf,3)[0][-1] - ey))
        y4er.append(np.linalg.norm(BDF(f,initf,4)[0][-1] - ey))

    d1m = max([np.log10(y1er[i])-np.log10(y1er[i+1]) for i in range(len(y1er)-1)]) #誤
差の差の最大値を取得
    d2m = max([np.log10(y2er[i])-np.log10(y2er[i+1]) for i in range(len(y2er)-1)])
    d3m = max([np.log10(y3er[i])-np.log10(y3er[i+1]) for i in range(len(y3er)-1)])
    d4m = max([np.log10(y4er[i])-np.log10(y4er[i+1]) for i in range(len(y4er)-1)])
    print('c='+str(c),d1m/np.log10(2),d2m/np.log10(2),d3m/np.log10(2),d4m/np.log10(2))
#傾きを対数スケールで求めて表示

def ggraph(): #(ii)のODE用のグラフ
    ye,labe = ExpE(g,initg) #解、ラベルを取得
    y4,lab4 = BDF(g,initg,1)

    fig = plt.figure(figsize=(10,5))
    ax1 = fig.add_subplot(121)
    ax4 = fig.add_subplot(122)

    ax1.set_xlabel('$t$')
    ax1.set_ylabel('$y$')
    ax1.set_title(labe+'; $n=$'+str(n)+'; $\Delta t=$'+str(dt))
    ax1.plot(tv,[x[0] for x in ye],label=labe)

    ax4.set_xlabel('$t$')
    ax4.set_ylabel('$y$')
    ax4.set_title(lab4+'; $n=$'+str(n)+'; $\Delta t=$'+str(dt))
    ax4.plot(tv,[x[0] for x in y4],label=lab4)
```

```
plt.rcParams["font.size"] = 8
fig.tight_layout()
plt.show()
```