

# システム数理演習 レポート

広域システム4年  
40413 諸町大地

## 0. 概略

ケプラー問題を解くプログラムを用いて、次の問題を解く。初期条件は、 $\text{pos}=(1.0,0.0,0.0)$ ,  $\text{vel}=(0.0,1.0,0.0)$ とする。このとき解析解は、位置 $=(\cos(t),\sin(t),0)$ , 速度 $=(\sin(t),\cos(t),0)$ である。dt や t\_end や dt\_out などは、以下の問題に適宜合うように選んで実行すること。すべての問題において、時間ステップ dt は固定で良い。

### 0.1. プログラム

“main.rb”と以下に示す“vector.rb”・“lboby.rb”を用いた。“main.rb”は問題ごとに変更を加えた為、各問題で示す。

“lboby.rb”には“forward”・“leapfrog”・“rk2”・“rk4”・“yo6”・“yo8”・“ms2”・“ms4”の8つの方法を実装し、問題では3つの方法のみが指定されているが、全ての方法で行うことにした。また、テキストのプログラムに対し以下に示す変更を加えた。

- ①“ms”法に対応するため、“nsteps”は“@nsteps”に変更。
- ②出力を同期させる為、“STDERR”は使用せず、全て“printf”に変更。
- ③EXCELで解析を行う関係上、半角スペース区切りで出力できるように変更。
- ④問題によって不必要な出力命令文文頭に“#”を追加。

グラフ作成は出力を半角スペース区切りで EXCEL で読み込み、問題ごとにマクロを作成し、グラフを作成した。(マクロのミスによるグラフのミスの可能性がある。)

#### 0.1.1. “vector.rb”

```
class Vector < Array
  def +(a)
    sum = Vector.new
    self.each_index{|k|
      sum[k] = self[k] + a[k]
    }
    return sum
  end
  def -(a)
    diff = Vector.new
```

```

    self.each_index{|k|
      diff[k]=self[k]-a[k]
    }
    return diff
end
def *(a)
  if a.class==Vector
    product=0
    self.each_index{|k|
      product+=self[k]*a[k]
    }
  else
    product=Vector.new
    self.each_index{|k|
      product[k]=self[k]*a
    }
  end
  return product
end
def /(a)
  if a.class==Vector
    raise
  else
    quotient=Vector.new
    self.each_index{|k|
      quotient[k]=self[k]/a
    }
  end
  return quotient
end
end
class Array
  def to_v
    return Vector[*self]
  end
end
end

```

### 0.1.2. "lbody.rb"

```
require "vector.rb"
class Body
  attr_accessor :mass, :pos, :vel
  def initialize(mass=0, pos=Vector[0, 0, 0], vel=Vector[0, 0, 0])
    @mass, @pos, @vel = mass, pos, vel
  end
  def evolve(integration_method, dt, dt_dia, dt_out, dt_end)
    time=0
    @nsteps=0
    e_init
    write_diagnostics(@nsteps, time)
    t_dia=dt_dia-0.5*dt
    t_out=dt_out-0.5*dt
    t_end=dt_end-0.5*dt
    while time<t_end
      send(integration_method, dt)
      time+=dt
      @nsteps+=1
      if time>=t_dia
        write_diagnostics(@nsteps, time)
        t_dia+=dt_dia
      end
      if time>=t_out
        simple_print
        t_out+=dt_out
      end
    end
  end
end
def acc
  r2=@pos*@pos
  r3=r2*sqrt(r2)
  @pos*(-@mass/r3)
end
def forward(dt)
  old_acc=acc
```

```

    @pos+=@vel*dt
    @vel+=old_acc*dt
end
def leapfrog(dt)
    @vel+=acc*0.5*dt
    @pos+=@vel*dt
    @vel+=acc*0.5*dt
end
def rk2(dt)
    old_pos=pos
    half_vel=vel+acc*0.5*dt
    @pos+=vel*0.5*dt
    @vel+=acc*dt
    @pos=old_pos+half_vel*dt
end
def rk4(dt)
    old_pos=pos
    a0=acc
    @pos=old_pos+vel*0.5*dt+a0*0.125*dt*dt
    a1=acc
    @pos=old_pos+vel*dt+a1*0.5*dt*dt
    a2=acc
    @pos=old_pos+vel*dt+(a0+a1*2)*(1/6.0)*dt*dt
    @vel=vel+(a0+a1*4+a2)*(1/6.0)*dt
end
def yo4(dt)
    d=[1.351207191959657,-1.702414383919315]
    leapfrog(dt*d[0])
    leapfrog(dt*d[1])
    leapfrog(dt*d[0])
end
def yo6(dt)
    d=[ 0.784513610477560e0,
        0.235573213359357e0,
        -1.17767998417887e0,
        1.31518632068391e0]

```

```

for i in 0..2 do
  leapfrog(dt*d[i])
end
leapfrog(dt*d[3])
for i in 0..2 do
  leapfrog(dt*d[2-i])
end
end
def yo8(dt)
  d=[ 0.104242620869991e1,
      0.182020630970714e1,
      0.157739928123617e0,
      0.244002732616735e1,
      -0.716989419708120e-2,
      -0.244699182370524e1,
      -0.161582374150097e1,
      -0.17808286265894516e1]
  for i in 0..6 do
    leapfrog(dt*d[i])
  end
  leapfrog(dt*d[7])
  for i in 0..6 do
    leapfrog(dt*d[6-i])
  end
end
def ms2(dt)
  if @nsteps==0
    @prev_acc=acc
    rk2(dt)
  else
    old_acc=acc
    jdt=old_acc-@prev_acc
    @pos+=vel*dt+old_acc*0.5*dt*dt
    @vel+=old_acc*dt+jdt*0.5*dt
    @prev_acc=old_acc
  end
end

```

```

end
def ms4(dt)
  if @nsteps==0
    @ap3=acc
    rk4(dt)
  elsif @nsteps==1
    @ap2=acc
    rk4(dt)
  elsif @nsteps==2
    @ap1=acc
    rk4(dt)
  else
    ap0=acc
    jdt=ap0*(11.0/6.0)-@ap1*3+@ap2*1.5-@ap3/3.0
    sdt2=ap0*2-@ap1*5+@ap2*4-@ap3
    cdt3=ap0-@ap1*3+@ap2*3-@ap3
    @pos+=vel*dt+(ap0/2.0+jdt/6.0+sdt2/24.0)*dt*dt
    @vel+=ap0*dt+(jdt/2.0+sdt2/6.0+cdt3/24.0)*dt
    @ap3=@ap2
    @ap2=@ap1
    @ap1=ap0
  end
end
end
def ekin
  0.5*(@vel*@vel)
end
def epot
  -@mass/sqrt(@pos*@pos)
end
def e_init
  @e0=ekin+epot
end
def write_diagnostics(nsteps,time)
  etot=ekin+epot
  printf("at time t=%g after %d steps :Yn",time,nsteps)
  printf("  E_kin= %.3gYn",ekin)

```

```

    printf(" E_pot= %.3g¥n",epot)
    printf(" E_tot= %.3g¥n",etot)
    printf(" E_tot-E_init= %.3g¥n",etot-@e0)
    printf(" (E_tot-E_init)/E_init= %.3g¥n",(etot-@e0)/@e0)
end
def to_s
  " mass = "+@mass.to_s+"¥n"+
  " pos  = "+@pos.join(",")+ "¥n"+
  " vel  = "+@vel.join(",")+ "¥n"
end
def pp
  print to_s
end
def simple_print
  printf("%25.16e¥n",@mass)
  @pos.each{|x|
    printf("%25.16e",x)
  }
  print "¥n"
  @vel.each{|x|
    printf("%25.16e",x)
  }
  print "¥n"
end
def simple_read
  @mass=gets.to_f
  @pos=gets.split.map{|x| x.to_f}.to_v
  @vel=gets.split.map{|x| x.to_f}.to_v
end
end

```

## 1. 問題 1

最初の 1 ステップ目で計算を止めて様子を見る。位置(x と y)・速度(vx と vy)・エネルギーについて、誤差がステップサイズにどのように依存しているかを見る。そのために、横軸に  $\log_{10}(dt)$  (10 を底とする対数)、縦軸に  $\log_{10}$ (各種誤差の絶対値)をとったグラフを描く。ステップサイズは  $dt=0.001,0.01,0.1$  の 3 種類は必ずみること。時間積分のメソッドについては、ホイン法、リープフロッグ法、4 次のシンプレクティック法の 3 種類で行い、結果を比較せよ。

### 1.1. プログラム

"main.rb"を以下のように変更した。

#### 1.1.1. "main.rb"

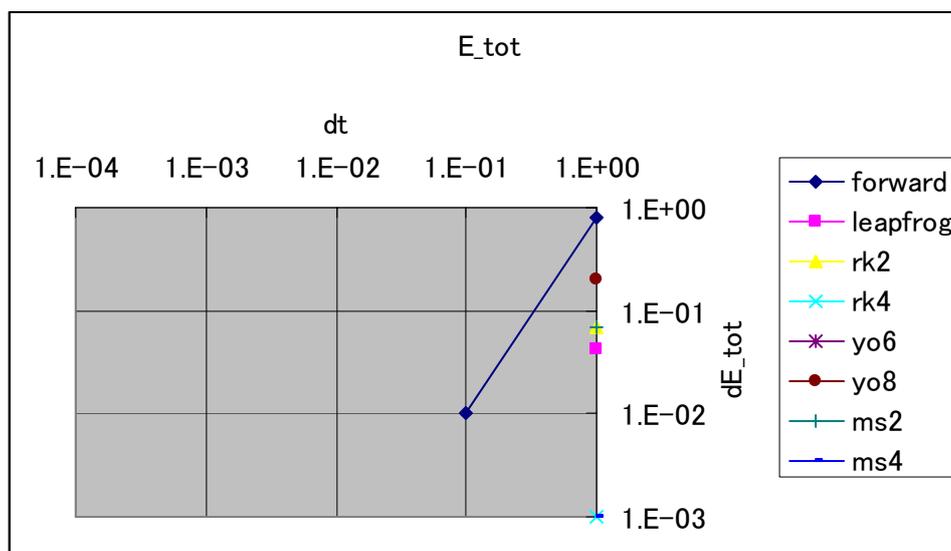
```
require "lbody.rb"
include Math

method=["forward","leapfrog","rk2","rk4","yo6","yo8","ms2","ms4"]
dt=[1,0.1,0.01,0.001,0.0001]

method.each{|m|
  printf("method= %s¥n",m)
  dt.each{|d|
    printf("dt= %g¥n",d)
    b=Body.new(1.0,Vector[1.0,0.0,0.0],Vector[0.0,1.0,0.0])
    b.evolve(m,d,d,d,d)
  }
}
```

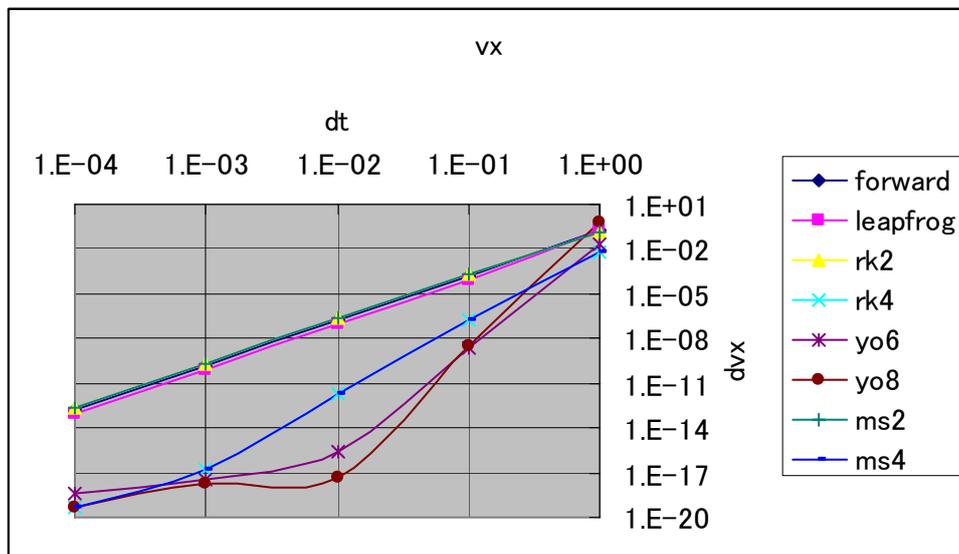
### 1.2. 結果

#### 1.2.1. E\_tot

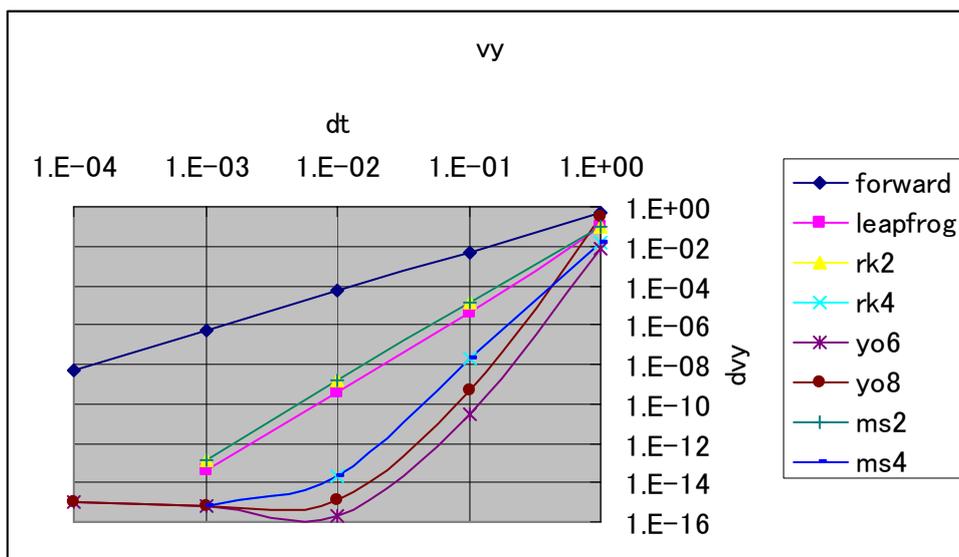




### 1.2.4. vx



### 1.2.5. vy



## 1.3. 考察

まず対象によって大まかな傾向を言うと、 $E_{tot}$  は  $dt$  がそれほど大きくなければ  $pc$  の有効数字内の誤差は出ないことがわかった。 $forward$  のみが特に高い誤差を示している。

$x$  と  $v_y$  は似た傾向を示している。 $dt=1$  のときはどの方法でも誤差に大差はない。しかし傾きが違う為、 $dt$  が小さくなると方法によって違いが出てくる。 $Forward$  は傾き 1、 $rk2 \cdot leapfrog \cdot ms2$  が傾き 2、 $rk4$  と  $ms4$  が傾き 3、 $yo6$  が傾き 4、 $yo8$  が傾き 4 強を示している。ただし、 $yo4$  と  $yo6$  は  $dt=0.01$  のとき最も誤差が小さいという不思議な結果となった。

$y$  と  $vx$  も似た傾向を示す。 $dt=1$  ではほとんど誤差に大差はなく、 $dt$  を小さくしたときの誤差の小さくなり方が方法によって違うのは  $x$  や  $vy$  と同じである。しかし、その傾きが多少異なる。`forward`・`leapfrog`・`rk2`・`ms2` が傾き 1、`rk4`・`ms4` が傾き 2 弱、`yo6` が傾き 2 強、`yo8` が傾き 3 を示している。

$E_{tot}$  は今回対象とした  $dt$  の値ではほとんど誤差 0 であったので、いえることはほとんどないが、 $x \cdot vy$  および  $y \cdot vx$  についてはつぎのことが言える。 $x \cdot vy$  および  $y \cdot vx$  がそれぞれ似ているのは、解析解が `cos` 系、`sin` 系であることに対応していると考えられる。この問題で調べた  $t=0$  付近では `cos` 系は傾きが大きく変化しない為、近似次数に応じた傾きとなっている。一方、`sin` 系は  $t=0$  付近では大きく傾きが変わる為、一次・二次で大差がなくなってしまっていると考えられる。`yo` の 4・6・8 に関して、 $dt=0.001$  から  $dt=0.01$  にかけてグラフが曲がってしまっているのは、この付近で解析解に対し大小が入れ替わった為と考えられる。

## 2. 問題 2

t=10.0 まで計算して、t=10.0 での、エネルギーについて、誤差がステップサイズにどのように依存しているかを、上と同じようなグラフを描いて調べ、異なる積分メソッドどうしの結果を比較・考察せよ。

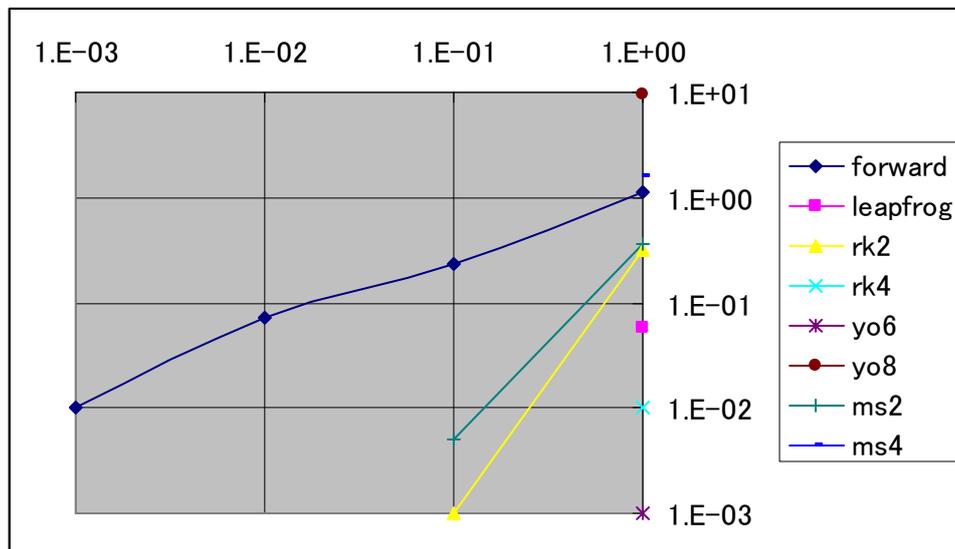
### 2.1. プログラム

“main.rb”を以下のように変更した。

#### 2.1.1. “main.rb”

```
require "lbody.rb"
include Math
method=["forward","leapfrog","rk2","rk4","yo6","yo8","ms2","ms4"]
dt=[1,0.1,0.01,0.001]
method.each{|m|
  printf("method= %s¥n",m)
  dt.each{|d|
    printf("dt= %g¥n",d)
    b=Body.new(1.0,Vector[1.0,0.0,0.0],Vector[0.0,1.0,0.0])
    b.evolve(m,d,10,10,10)
  }
}
```

### 2.2. 結果



### 2.3. 考察

forward は傾き 0.7、ms2 が 2、rk2 が 2.5 ほどで、他は pc の有効数字での優位な誤差が得られず、傾きは非常に大きなことがわかる。同じ 2 次近似の leapfrog・rk2・ms2 で違いが見られたのは興味深い。

### 3. 問題 3

$t=1000.0$  まで計算して、その結果を、横軸に時間  $t$ 、縦軸にエネルギーの誤差をとったグラフにする。これはどちらの軸もリニアスケール (log はとらない)。提出するグラフはどれかひとつのステップサイズのみで良い。ただし、ホイン法、リープフロッグ法、4 次のシンプレクティック法の 3 種類について同じ  $dt$  で計算した結果を提出すること。

#### 3.1. プログラム

“main.rb”を以下のように変更した。また、“E\_tot”のみを表示するよう“lbody.rb”の一部に“#”をつけた。

##### 3.1.1. “main.rb”

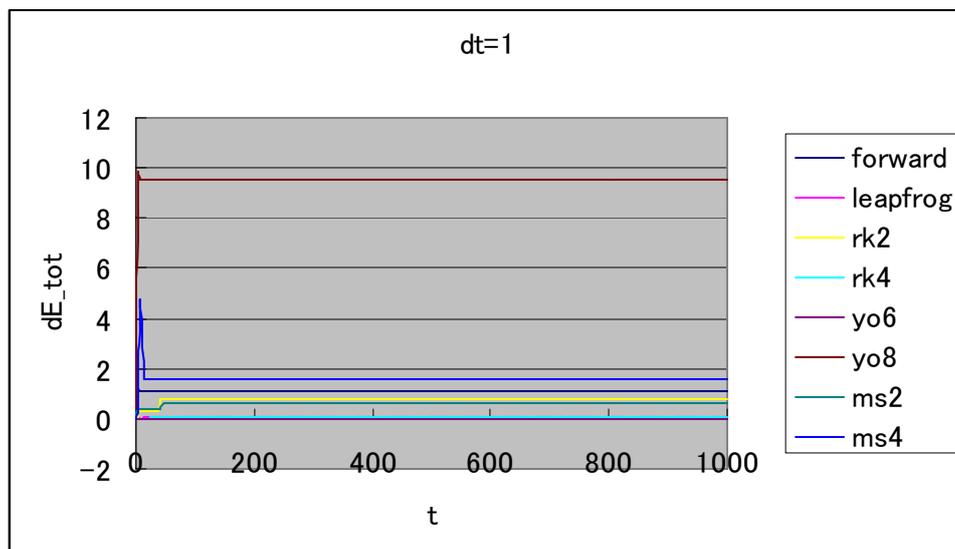
```
require "lbody.rb"
include Math

method=["forward", "leapfrog", "rk2", "rk4", "yo6", "yo8", "ms2", "ms4"]
dt=[0.001]

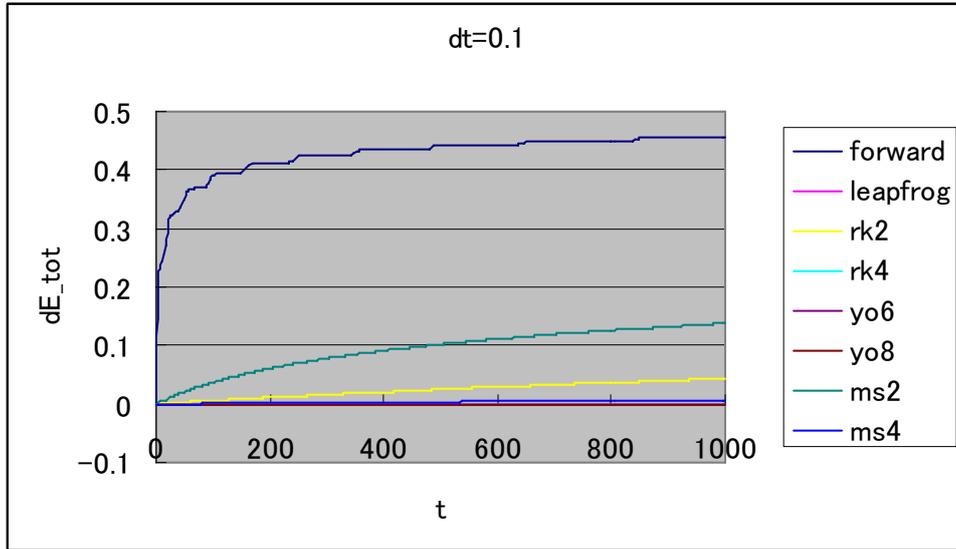
method.each{|m|
  printf("method= %s\n",m)
  dt.each{|d|
    printf("dt= %g\n",d)
    b=Body.new(1.0, Vector[1.0,0.0,0.0], Vector[0.0,1.0,0.0])
    b.evolve(m,d,4,4,1000)
  }
}
```

#### 3.2. 結果

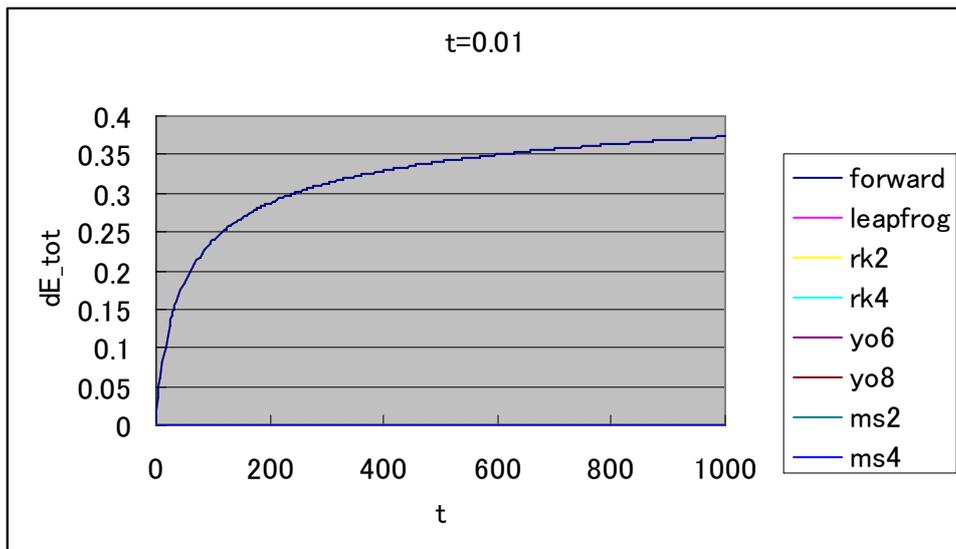
##### 3.2.1. dt=1



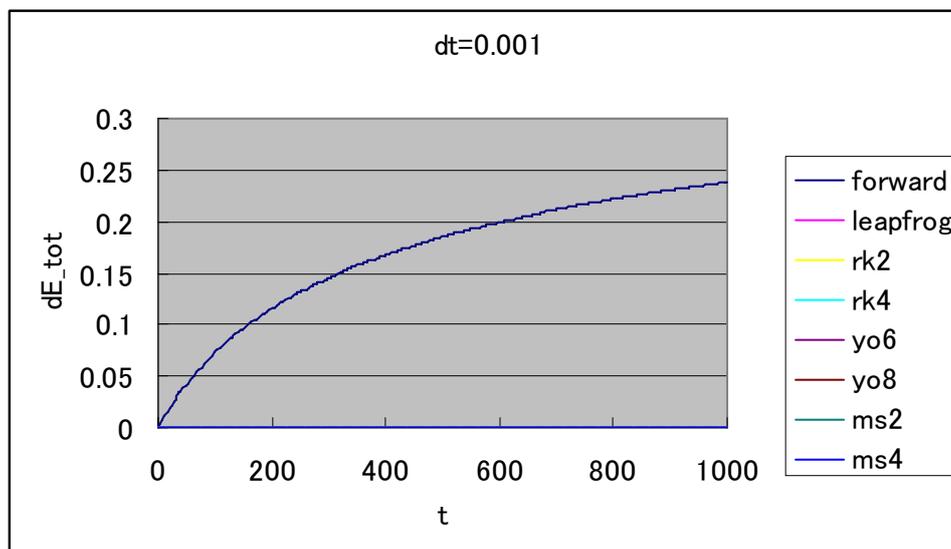
### 3.2.2. dt=0.1



### 3.2.3. dt=0.01



### 3.2.4. dt=0.001



### 3.3. 考察

誤差はある  $t$  の値から一気に大きくなることがわかる。また、その  $t$  の値は  $dt$  が大きいほど小さい。また、一気に上がるのは一回だけでなく、数回あるものもある。 $dt$  が小さくなると **forward** 以外は有為な誤差は得られない。

#### 4. 問題 4

初期条件  $\text{pos}=(1.0,0.0,0.0)$   $\text{vel}=(0.0,0.5,0.0)$  を、10周まわるまで積分し、結果を横軸に時間  $t$ 、縦軸に位置  $x$  をとったグラフにせよ。また、横軸に  $x$ 、縦軸に  $y$  をとったグラフも描け。これはどちらのグラフも提出すること。積分方法は、3種類のうちのどれでも良いが提出するレポートには、どれを使ったか明記せよ。また、タイムステップは、 $dt=0.1$  とせよ。(これで10周までできなかつた場合は、積分法を変えるか、適宜  $dt$  を小さくするかしてみよ。)

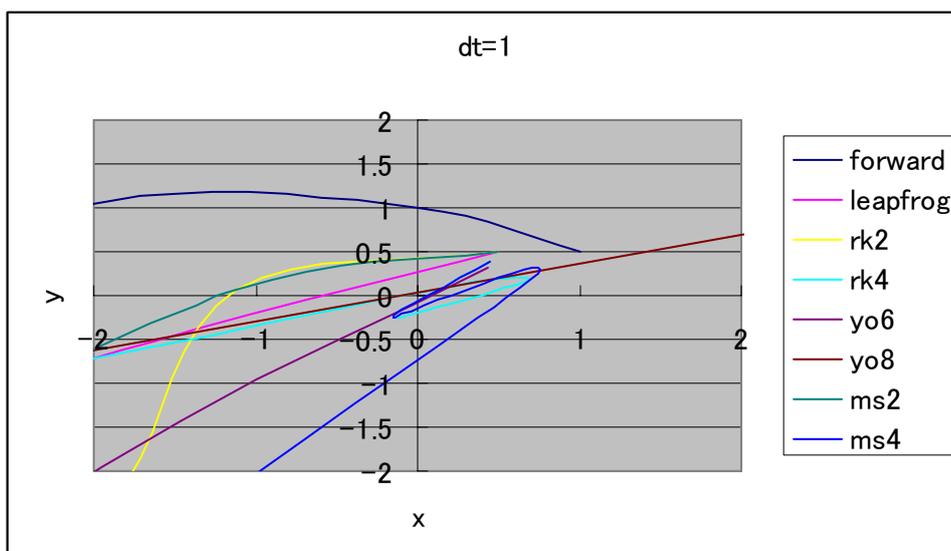
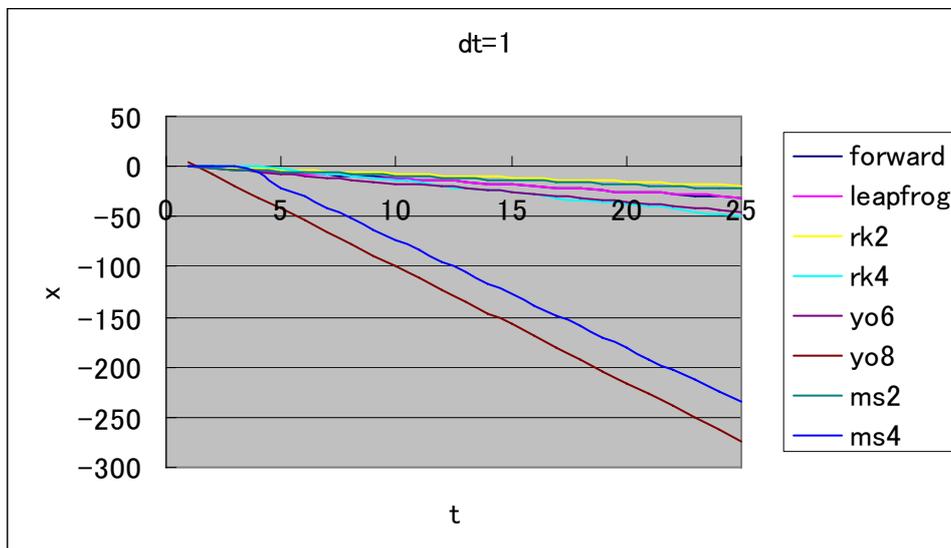
##### 4.1. プログラム

###### 4.1.1. “main.rb”

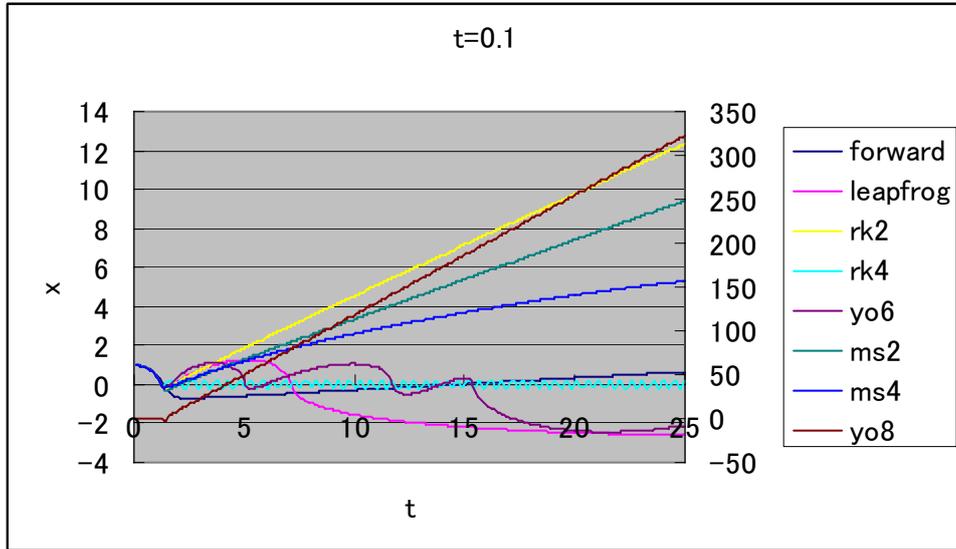
```
require "lbody.rb"
include Math
method=["forward","leapfrog","rk2","rk4","yo6","yo8","ms2","ms4"]
dt=[0.001]
method.each{|m|
  printf("method= %s\n",m)
  dt.each{|d|
    printf("dt= %g\n",d)
    b=Body.new(1.0,Vector[1.0,0.0,0.0],Vector[0.0,0.5,0.0])
    b.evolve(m,d,0.1,0.1,25)
  }
}
```

## 4.2. 結果

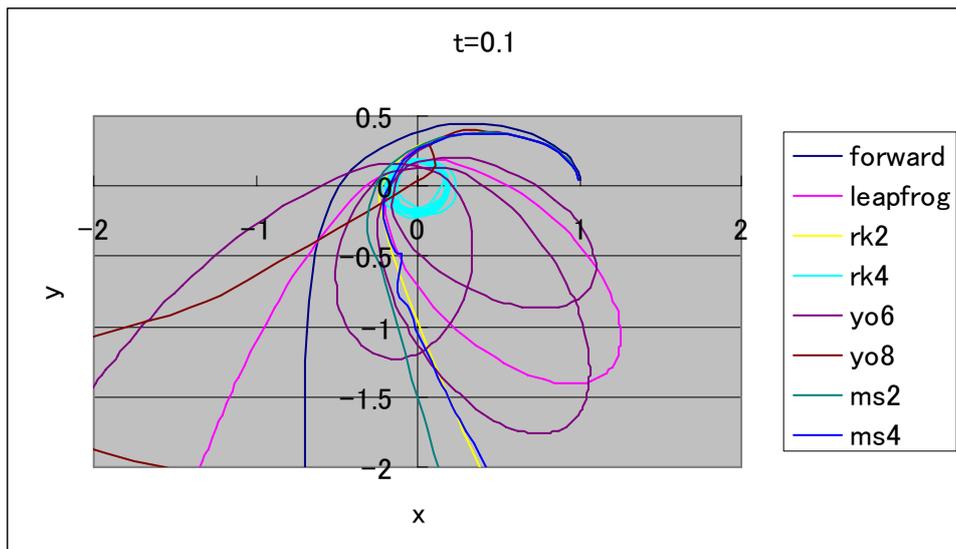
### 4.2.1. $t=1$



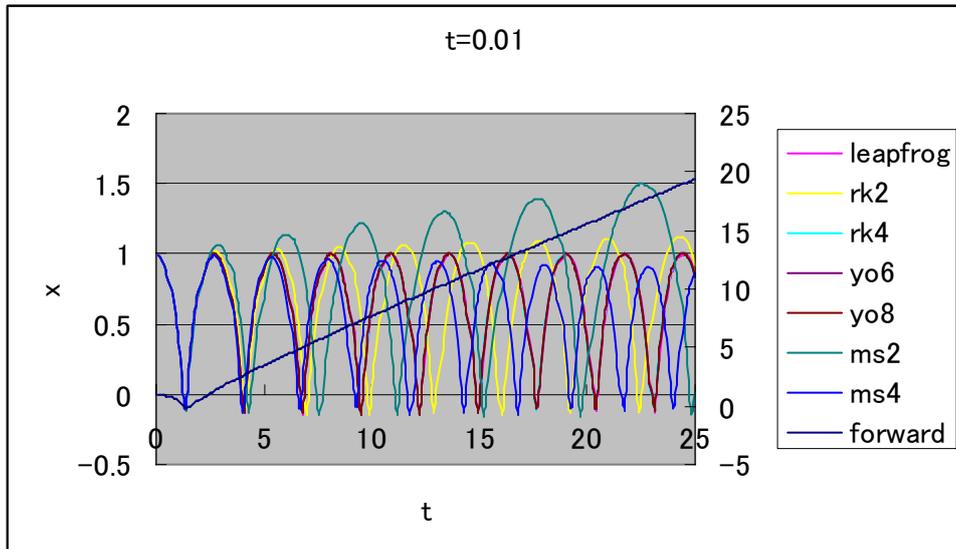
#### 4.2.2. t=0.1



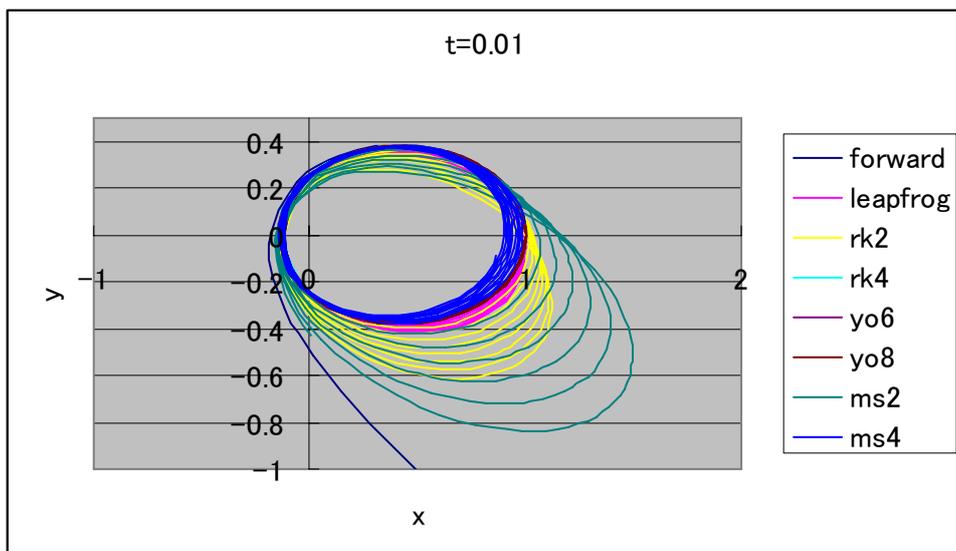
(yo8 は右軸)



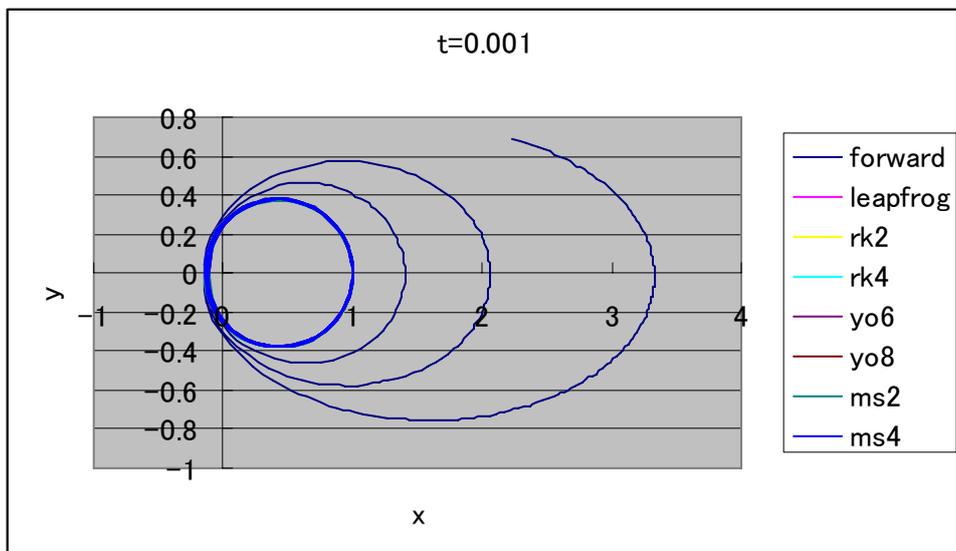
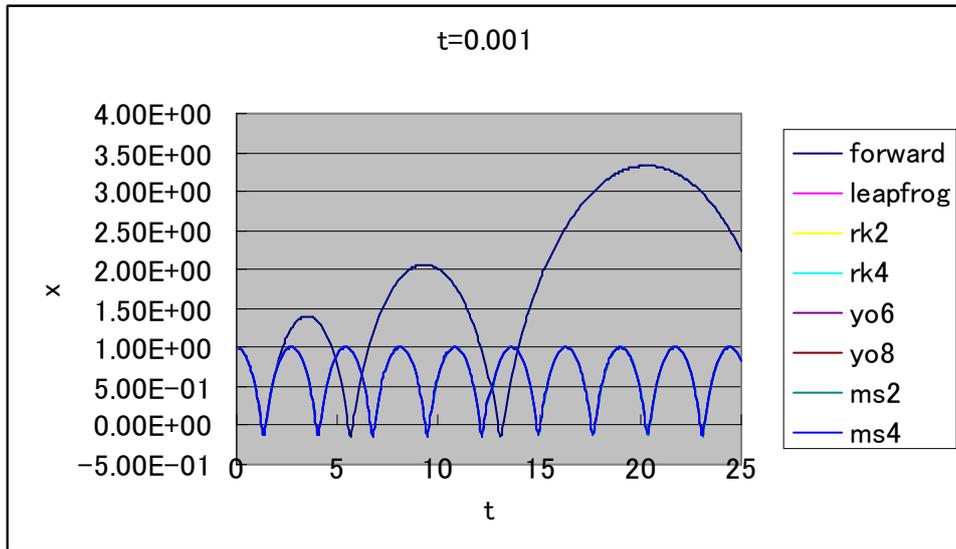
### 4.2.3. $t=0.01$



(forward は右軸)



#### 8 4.2.4. t=0.001



#### 4.3. 考察

$t=1$  では全く回らない。

$t=0.1$  ではほとんどが回ることなく発散してしまうが、一部の方法で数周回ってから発散している。また、 $rk4$  のみがいつまでも回転できている。ただし、その軌道半径は本来の値よりも小さく、周期も短い。

$t=0.01$  では **forward** を除く全ての方法で 10 周回ることが出来た。ただし、だんだん軌道を離れてしまう傾向のものが多い。 $rk4 \cdot yo6 \cdot yo8$  のみがきれいな軌道を描いている。

$t=0.001$  では **forward** を除く全ての方法で完全にきれいな軌道を描いた。**forward** も何とか回転出来ているが、徐々に軌道を離れてしまっている。