Worksheet for ODE in Python

Utils to define integration methods

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Feb 9 10:43:39 2022
@author: H. El-Otmany
@ This file contain all numerical method  for ODE
@ Input data :
    - dydt = f(t,y),
    - start value t=a,
    - end value t=b,
    - number of subdivison n
    - Initial condition y0
@ Output : numerical solution y
@Algorithme d Euler (Runge Kutta d ordre un)
Euler(f , y0, t_0, t_f , N)
Input: f fonction donnees
(t0; y0) point initial
t f abscisse final
N nombre de points de [t_0; t_f ]
    h <--- (t_f -t_0)/N
    Ly <--- y0, Lt ← t0
Pour k de 1 a N faire
    y0 <---y0 + h.f(t0, y0)
    t0 <---t0 + h
    Ly <---Ly, y0; # stocker les solutions
    Lt <---Lt, t0 # stocker les abscisses
Ouptut : Ly liste des ordonnees yk, k = 0; 1;... ; N
         Lt liste des ordonnees tk, k = 0; 1;... ; N
"""

import numpy as np
import matplotlib.pyplot as plt

from scipy.optimize import newton
from scipy.integrate import odeint
from scipy.integrate import solve_ivp
#Euler method for iniline function
def ode_EulerExp(f, a, b, y0, N):
    h = (b-a) / N #step size if h is constant
    Lt = [a]  #Time list
    Ly = [y0] #Initial condition of velocity dy/dt
    t = a
    y = y0
    for i in range(1,N+1):
        #if h isn't constant, we use h=t[i+1]-t[i]
        y = y + h*f(t, y)
        t = t+ h
        Lt.append(t)
        Ly.append(y)
    return (Lt, Ly)

#Euler method for vector functions F(t,y1,y2,.....)
#Ok for inline function
def ode_VectEulerExp(f, a, b, ic, N):
      h = (b - a) / N #step size if h is constant
      Lt = np.linspace(a, b, N)
      Ly = np.empty((N, np.size(ic)), dtype = float)
      Ly[0,:] = ic
      for i in range(N-1):
          #if h isn't constant, we use h=t[i+1]-t[i]
          Ly[i+1,:] = Ly[i,:] + h*f(Lt[i],Ly[i,:])
      return (Lt, Ly)

#Runge-Kutta second order for iniline function & vector function
def ode_RK2(f, a, b, ic, N):
      h = (b - a) / N      #step size if h is constant
      Lt = np.linspace(a, b, N)
      Ly = np.empty((N, np.size(ic)),dtype = float)
      Ly[0,:] = ic
      for i in range(N-1):
          #if h isn't constant, we use h = t[i+1]-t[i]
          k1 = h*f(Lt[i], Ly[i,:])
          k2 = h*f(Lt[i] + h/2, Ly[i,:]+k1/2)
          Ly[i+1,:] = Ly[i,:] + k2
      return (Lt, Ly)

#Runge-Kutta fourth order for iniline function & vector function
def ode_RK4(f, a, b,ic, N):
      h = (b - a) / N      #step size if h is constant
      Lt = np.linspace(a, b, N)
      Ly = np.empty((N, np.size(ic)),dtype = float)
      Ly[0,:] = ic
      for i in range(N-1):
          #if h isn't constant, we use h=t[i+1]-t[i]
          k1 = h*f(Lt[i], Ly[i,:])
          y1 = Ly[i,:] + 1/2*k1
          k2 = h* f(Lt[i]+h/2, y1)
          y2 = Ly[i,:] + 1/2*k2
          k3 = h* f(Lt[i]+h/2,y2)
          y3 = Ly[i,:] + k3
          k4 = h* f(Lt[i]+h, y3)
          k = (k1+2*k2+2*k3+k4)/6
          Ly[i+1,:] = Ly[i,:] + k
      return (Lt, Ly)

#Backward Euler or Implicit Euler for iniline function
def ode_ForwardEuler(f, a, b, y0, N):
    h = (b-a) / N #step size if h is constant
    Lt = np.linspace(a,b,N)
    Ly = np.zeros(N)
    Ly[0] = y0
    for i in range(N-1):
        #if h isn't constant, we use h=t[i+1]-t[i]
        s = newton(lambda u: u - Ly[i]- f(Lt[i+1],u) * h, Ly[i+1])
        Ly[i+1] = s
    return (Lt, Ly)

#Heun method
def ode_Heun(f, a, b, ic, N):
    h = (b - a) / N      #step size if h is constant
    Lt = np.linspace(a, b, N)
    Ly = np.empty((N, np.size(ic)),dtype = float)
    Ly[0,:] = ic
    for i in range(N-1):
        #if h isn't constant, we use h=t[i+1]-t[i]
        k1 = f(Lt[i], Ly[i,:])
        k2 = f(Lt[i+1], Ly[i,:] + h * k1)
        Ly[i+1,:] = Ly[i,:] + h * (k1 + k2) / 2
    return (Lt, Ly)
```

```python
#Computing errors
def error(method, f, a, b, ic, N, sol):
    Lt, Ly = method(f, a, b, ic, N)
    err = 0
    for i in range(N):
        err = max(err, abs(Ly[i] - sol(Lt[i])))
    return err

# Print errors
def error_printing(f, a, b, ic, N, sol):
    print("Errors for N = {} : ".format(N))
    print("Explicit Euler : {}".format(error(ode_EulerExp,f, a, b, ic, N, sol)))
    print("RK2 : {}".format(error(ode_RK2,f, a, b, ic, N, sol)))
    print("RK4 : {}".format(error(ode_RK4,f, a, b, ic, N, sol)))
    print()

# Necessary rank N to obtain an uniform apporximation to eps
def necessary_rank(method, f, a, b, ic, N, sol, eps):
    if error(method, f, a, b, ic, N, sol)>eps: #error(methode, n0) > epsilon:
        return None
    p = 2
    r = N
    while r - p > 1:
        q = (p + r) // 2
        if error(method, f, a, b, ic, q, sol)>eps: #error(methode, c) > epsilon:
            p = q
        else:
            r = q
    return r

# def order_log2():
#     print('The order of a numerical approximation methods')
#     for method in ['ode_VectEulerExp', 'ode_RK2', 'ode_RK4']:
#         meth = eval(method)
#         order = error(meth,f3,a,b,ic,N,exac3)/error(meth,f3,a, b,ic,N,exac3)
#         print("order "+ method +" : {}".format(log(order,2)))
#     print()
```

testCase to define test functions

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Feb  8 21:15:05 2022
@ This file contain all test cases
@author: H. El-Otmany
@ This file contain all functions used in TP2
"""
import numpy as np
from math import exp,sin,pi

#Example 1: definition of function y'=-y + t
def f1(t, y):
    return - y + t
def exac1(t):
    return t - 1 + exp(-t)

#Example 2: definition of function for ODE: y" + y = t
#Y =(y,y'), compute Y' =(y',y") = (y', t-y)
def f2(t,y):
    [z, dz] = y
    return np.array([dz, -z +t])
def exac2(t):
    return t - sin(t)

#Example 3: definition of function y' = y + y**2
def f3(t, y):
    return y + y**2
def exac3(t):
    return exp(t)/(2-exp(t))

#Example 4: exo3-TD3, ODE: y" + ty'+ (1 - t)y = 2
#Y =(y,y'), compute Y' =(y',y") =
#F(t,Y)= (y', 2-ty'+(1-t)y)
def Fexo3_TD3(t, Y):
    v = Y[0]                    # y
    dv = Y[1]                   # y'
    ddv = 2 -t*dv + (1-t)*v     # y" = 2-ty'+(1-t)y
    return np.array([dv , ddv])

#Example 5: exo4-TD3, ODE: y" + ty'+ (1 - t)y = 2
#Y =(y,y', z), compute Y' =(y',y", z')=F(t,Y)
#F(t,Y)= (y', t+ty'-2z, (4t^2+ 1  y'  exp(t)y  2z)/3)
def Fexo4_TD3(t, Y):
    v = Y[0]                  #y
    u = Y[1]                  #y'
    w = Y[2]                  #z
    dy = u                    # y'
    ddy = t + t*v             # y" = t+ty'-2z
    dz = (4*t*t+1-u-exp(t)*v-2*w)/3 #z'=(4t^2+ 1  y'  exp(t)y  2z)/3
    return np.array([dy ,ddy , dz])

#Example 6: exo4-TD3, ODE: y' = sin(y(t)) +sin(t) on [0;T]
def Fexo3_TD4(t, y):
    return sin(y) + sin(t)
```

mainProg used for Execution

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 19 18:43:39 2022
@author: Hammou El-Otmany
@ this file used for testing
FYI: corrinfo12
@ solving with solve_ip, we can use :
  *Without tolerence
    - sol = solve_ivp(f2, [a, b], [0, 0], t_eval=t)
    - sol = solve_ivp(f2, [a, b], [0, 0], method= 'RK23', t_eval=t)
    - sol = solve_ivp(f2, [a, b], [0, 0], method= 'RK45', t_eval=t)
    - sol = solve_ivp(f2, [a, b], [0, 0], method= 'LSODA', t_eval=t)
  *with relative and obsolute tolerences
     - sol = solve_ivp(f2, [a, b], [0, 0], method= 'LSODA', t_eval=t,rtol = 1e-8, atol = 1e-8
@ solving with odeint, we can use :
    - sol = odeint(f2, y0, t, p, tfirst=True)
@Computing errors and orders by using : order_log2()
@Computing necassary rank N to obtain an uniform apporximation to eps by using
    - necessary_rank(method, f, a, b, ic, N,sol, eps)))
"""
import sys
from math import *
import numpy as np
from numpy import inf
import matplotlib.pyplot as plt
from matplotlib.pylab import *
from scipy.integrate import odeint
from scipy.integrate import solve_ivp

# Call all functions defined on utils and testCase
from utils import *
from testCase import *
```

```python
#define variables (a,b,N) and initial conditions ic=y0
a = 0
b = 2
ic = 0
# Question 1 - Exemple 1
def example1_Q1():
    """To Compute for different value of N, change the line
    'for N in [100]' to  'for N in [10, 100,1000]' """
    for N in [10, 100, 1000]:
        plt.figure(N+1)
        ###You can also use T,Y = ode_EulerExp(f1, a, b, ic, N)
        T,Y= ode_VectEulerExp(f1, a, b, ic, N)
        #T,Y = ode_VectEulerExp(f1, a, b, ic, N)
        plt.plot(T,Y,label = 'Explicit Euler with N='+ str(N))
        ###Runge-Kutta 2 - vector
        T,Y= ode_RK2(f1, a, b, ic, N)
        plt.plot(T,Y,label = 'RK2 with N='+ str(N))
        ###Runge-Kutta 2 - vector
        T,Y = ode_RK4(f1, a, b, ic, N)
        plt.plot(T,Y,label = 'RK4 with N='+ str(N))
        if exac1:
            Y  = [exac1(x) for x in T]
            plt.plot(T,Y, label = 'Theoretical solution')
            plt.xlabel('t')
            plt.ylabel('y and dy/dt')
        plt.title('Comparison methods-solving ODE: y’=-y +t on [0; 2]')
        plt.legend()
        plt.grid()
        ###Saving figure for reporting
        #plt.savefig("/Users/mac/Desktop/IPSA2022/Module_Ma322_2022/TP_Ma322/PythonTex2/ex1Q1.png ")
example1_Q1()

#Question 1 - Exemple 2
#define variables (a,b,N) and initial conditions (y(0),y'(0))
a = 0
b = 5
ic = np.array([0,0])
def example2_Q1():
    """To Compute for different value of N, change the line
    'for N in [100]' to  'for N in [10, 100,1000]' """
    for N in [10, 100, 1000]:
        plt.figure(N+2)
        ###Explicit Euler for vector function
        T,Y= ode_VectEulerExp(f2, a, b, ic, N)
        y= Y[:,0]
        dy = Y[:,1]
        ###To plot the phase portrait, plot of (y, dy)
        #plt.plot(y, dy,label = 'Explicit Euler: phase with N='+ str(N))
        ###To plot the solutions y as a function of time
        plt.plot(T, y,label = 'Explicit Euler: solution y(t) with N='+ str(N))
        ###Using solve_ip of scipy.integrate
        t = np.linspace(a,b,N) # you can use direclty t_eval=T
        sol = solve_ivp(f2, [a, b], [0, 0], t_eval=t)
        ###To plot the phase portrait, plot (y,dy)
        #plt.plot(sol.y[0], sol.y[1],label = 'Solve_ip: phase with N='+ str(N))
        ###To plot the solutions y as a function of time
        plt.plot(T, sol.y[0],label = 'Solve_ip: solution y(t) with N='+ str(N))
        ###Runge-Kutta 2 - vector
        T,Y = ode_RK2(f2, a, b, ic, N)
        y, dy = Y[:,0], Y[:,1]
        plt.plot(T,y,label = 'RK2: solution y(t) with N='+ str(N))
        ###Runge-Kutta 4 - vector
        T,Y = ode_RK4(f2, a, b, ic, N)
        y, dy = Y[:,0], Y[:,1]
        plt.plot(T,y,label = 'RK4: solution y(t) with N='+ str(N))
        if exac2:
            Y = [exac2(x) for x in T]
            plt.plot(T,Y, label = 'Theoretical solution')
            plt.xlabel('t')
            plt.ylabel('y and dy/dt')
        plt.title('Comparison methods-solving ODE: y"+y=t on [0; 5]')
        plt.legend()
        plt.grid()
        ###Saving figure for reporting
        #plt.savefig("/Users/mac/Desktop/IPSA2022/Module_Ma322_2022/TP_Ma322/PythonTex2/ex2Q1.

example2_Q1()


#Question 1 - Exemple 3
#define variables (a,b,N) and initial conditions (y(0),y'(0))
a = 0
b = 0.5
ic = 1
def example3_Q1():
    """To Compute for different value of N, change the line
    'for N in [100]' to  'for N in [10, 100,1000]' """
    for N in [100]:
        plt.figure(N+3)
        ###You can also use
        #X,Y= ode_EulerExp(f3, a, b, ic, N)
        ###Euler explicite-vector
        T,Y = ode_VectEulerExp(f3, a, b, ic, N)
        plt.plot(T,Y,label = 'Explicit Euler with N='+ str(N))
        ###Runge-Kutta 2 - vector
        T,Y = ode_RK2(f3, a, b, ic, N)
        plt.plot(T,Y,label = 'RK2 with N='+ str(N))
        ###Runge-Kutta 4 - vector
        T,Y = ode_RK4(f3, a, b, ic, N)
        plt.plot(T,Y,label = 'RK4 with N='+ str(N))
        if exac3:
            Y  = [exac3(x) for x in T]
            plt.plot(T,Y, label = 'Theoretical solution')
            plt.xlabel('t')
            plt.ylabel('y and dy/dt')
        plt.title('Comparison methods-solving ODE: dy/dt=y+y*y on [0; 1/2]')
        plt.legend()
        plt.grid()
        ###Saving figure for reporting
        #plt.savefig("/Users/mac/Desktop/IPSA2022/Module_Ma322_2022/TP_Ma322/PythonTex2/exo3Q
example3_Q1()

#Computing log order in base 2 of of a numerical approximation methods
#define variables (a,b,N) and initial conditions (ic=y(0)) in example 1
a = 0
b = 0.5
N = 10000
ic = 1
# print relative errors
error_printing(f3, a, b, ic, N, exac3)

#Necessary rank N to obtain an uniform apporximation to eps
print('Rank for Euler Expl., 1e-2: {}'.format(necessary_rank(ode_VectEulerExp, f3,a,b,ic,N, ex
print('Rank for RK2, 1e-2: {}'.format(necessary_rank(ode_RK2, f3, a,b,ic,N,exac3, 1e-2)))
print('Rank for RK4, 1e-2: {}'.format(necessary_rank(ode_RK4, f3, a,b,ic,N,exac3, 1e-2)))


"""
@Exo3 - TD3
#define variables (a,b,N) and initial conditions (y(0)=0,y'(0)=0)
#I=[a;b]=[0;1]
"""
a = 0
b = 1
```

```python
ic = np.array([0,0])
def exo3_TD3():
    """To Compute for different value of N, change the line
    'for N in [100]' to  'for N in [10, 100,1000]' """
    for N in [100]:
        plt.figure(N+4)
        ###Euler explicite-vector
        T, Y = ode_VectEulerExp(Fexo3_TD3, a, b, ic, N)
        y, dy = Y[:,0], Y[:,1]
        ###To plot the phase portrait, plot of (y, dy)
        #plt.plot(y, dy,label = 'Explicit Euler: phase with N='+ str(N))
        ###To plot the solutions y as a function of time
        plt.plot(T, y,label = 'Explicit Euler: solution y(t) with N='+ str(N))
        ###Using solve_ip of scipy.integrate
        t = np.linspace(a,b,N) # you can use direclty t_eval=T
        sol = solve_ivp(Fexo3_TD3, [a, b], [0, 0], t_eval=t)
        #plt.plot(sol.y[0], sol.y[1],label = 'Solve_ip: phase with N='+ str(N))
        ###To plot the solutions y as a function of time
        plt.plot(T, sol.y[0],label = 'Solve_ip: solution y(t) with N='+ str(N))
        ###Runge-Kutta 2 - vector
        T,Y = ode_RK2(Fexo3_TD3, a, b, ic, N)
        y, dy = Y[:,0], Y[:,1]
        plt.plot(T,y,label = 'RK2: solution y(t) with N='+ str(N))
        ###Runge-Kutta 4 - vector
        T,Y = ode_RK4(Fexo3_TD3, a, b, ic, N)
        y, dy = Y[:,0], Y[:,1]
        plt.plot(T,y,label = 'RK4: solution y(t) with N='+ str(N))
        plt.title('Comparison methods-solving ODE: ddydt + tdydt + (1 - t)y = 2 on [0; 1]')
        plt.legend()
        plt.grid()
        ###Saving figure for reporting
        #plt.savefig("/Users/mac/Desktop/IPSA2022/Module_Ma322_2022/TP_Ma322/PythonTex2/exo3TD3.png")
exo3_TD3()

"""
@Exo4 - TD3
#define variables (a,b,N) and initial conditions (y(0)=0,y'(0)=0, y"(0)=0)
#I= [0;T]=[0;1]
#eq1: ddydt - tdydt+2z=t;
#eq2: dydt + e^ty + 3dzdt2z = 4t^2 + 1
"""
a = 0
b = 1
ic = np.array([0,0,0])
def exo4_TD3():
    """To Compute for different value of N, change the line
    'for N in [100]' to  'for N in [10, 100,1000]' """
    for N in [100]:
        plt.figure(N+5)
        ###Euler explicite-vector
        T, Y = ode_VectEulerExp(Fexo4_TD3, a, b, ic, N)
        y, dy, z = Y[:,0], Y[:,1], Y[:,2]
        ###To plot the phase portrait, plot of (y, dy)
        #plt.plot(y, dy,label = 'Explicit Euler: phase with N='+ str(N))
        ###To plot the solutions y as a function of time
        plt.plot(T, y,label = 'Explicit Euler: solution y(t) with N='+ str(N))
        plt.plot(T, z,label = 'Explicit Euler: solution z(t) with N='+ str(N))
        ###Using solve_ip of scipy.integrate
        t = np.linspace(a,b,N) # you can use direclty t_eval=T
        sol = solve_ivp(Fexo4_TD3, [a, b], [0, 0, 0], t_eval=t)
        ###To plot the solutions y as a function of time
        plt.plot(T, sol.y[0],label = 'Solve_ip: solution y(t) with N='+ str(N))
        plt.plot(T, sol.y[2],label = 'Solve_ip: solution z(t) with N='+ str(N))
        ###Runge-Kutta 2 - vector
        T,Y = ode_RK2(Fexo4_TD3, a, b, ic, N)
        y, dy, z = Y[:,0], Y[:,1], Y[:,2]
        plt.plot(T,y,label = 'RK2: solution y(t) with N='+ str(N))
        plt.plot(T,z,label = 'RK2: solution z(t) with N='+ str(N))
        ###Runge-Kutta 4 - vector
        T,Y = ode_RK4(Fexo4_TD3, a, b, ic, N)
        y, dy, z = Y[:,0], Y[:,1], Y[:,2]
        plt.plot(T,y,label = 'RK4: solution y(t) with N='+ str(N))
        plt.plot(T,z,label = 'RK4: solution z(t) with N='+ str(N))
        plt.title('Comparison methods-solving of DS')
        plt.legend()
        plt.grid()
        ###Saving figure for reporting
        #plt.savefig("/Users/mac/Desktop/IPSA2022/Module_Ma322_2022/TP_Ma322/PythonTex2/exo4TD
exo4_TD3()

"""
@Exo3 - TD4
#define variables (a,b,N) and initial conditions (y(0)=0,y'(0)=0, y"(0)=0)
#I= [0;T]=[0;1]
#ODE: y' =sin(y(t))+sin(t)'
"""
a = 0
b = 1
ic = 0
def exo3_TD4():
    """To Compute for different value of N, change the line
    'for N in [100]' to  'for N in [10, 100,1000]' """
    for N in [100]:
        plt.figure(N+6)
        ###Euler explicite-vector
        T, Y = ode_VectEulerExp(Fexo3_TD4, a, b, ic, N)
        ###To plot the solutions y as a function of time
        plt.plot(T, Y,label = 'Explicit Euler: solution y(t) with N='+ str(N))
        ###Using solve_ip of scipy.integrate
        t = np.linspace(a,b,N) # you can use direclty t_eval=T
        sol = solve_ivp(Fexo3_TD4, [a, b],[0], t_eval=T)
        ###To plot the solutions y as a function of time
        plt.plot(t, sol.y[0],label = 'Solve_ip: solution y(t) with N='+ str(N))
        ###Runge-Kutta 2 - vector
        T,Y = ode_RK2(Fexo3_TD4, a, b, ic, N)
        plt.plot(T,Y,label = 'RK2: solution y(t) with N='+ str(N))
        ###Runge-Kutta 4 - vector
        T,Y = ode_RK4(Fexo3_TD4, a, b, ic, N)
        plt.plot(T,Y,label = 'RK4: solution y(t) with N='+ str(N))
        plt.title('Comparison methods-solving of DS')
        plt.legend()
        plt.grid()
        ###Saving figure for reporting
        #plt.savefig("/Users/mac/Desktop/IPSA2022/Module_Ma322_2022/TP_Ma322/PythonTex2/exo3TD
exo3_TD4()
```