rk23

March 17, 2022

1 A Simple Runga Kutta method in action

Demonstration of an Explicit embeded RK. The method is of order 2 and the embeded method of order 3. The demonstration is based on the 1st order ODE

$$u_t(t) = u^2(t) - u^3(t),\ 0 \leq t \leq 2/\delta$$

For camparisson we will use a solver from SciPy.

[4]:
```
#IMPORTS AND FUNCTION DEFINITIONS
import matplotlib.pyplot as plt; import numpy as np
from scipy.integrate import solve_ivp

# a help function so I don't have to type every time
def standardplot(xl,yl,ti):
    plt.xlabel(xl)
    plt.ylabel(yl)
    plt.title(ti)
    plt.grid(lw = 0.5)
```

1.0.1 The solver

This is an uverisal solver and should be able to solve first orde ODE with inital value given and a given time space.

$$y' = f(t, y), y(t_0) = \text{IV (Initial Value)}$$

[7]:
```
def rk23(f,t0,t_final,IV,h_start, reps = 1e-4, aeps = 1e-4):
    #THIS IS THE SOLVER; Rung-Kutta order 2 with adaptive step size
    def h_opt(h,tol,E,p):
        return 0.9*h*(tol/E)**(1/p)

    count = 0
    p = 2
    h= h_start
    reject = 0 # count for analysis purposes
    h_ = np.array([h])
    t = np.array([t0])
    U = np.array([IV])
    tn,Un  = t0,IV
```

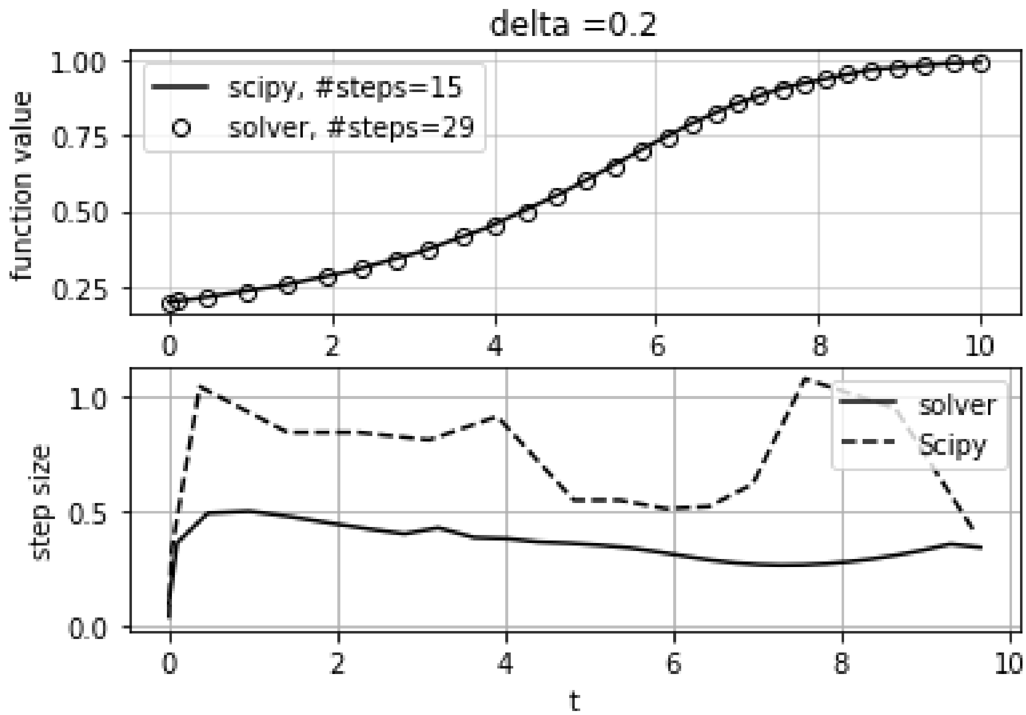
```python
    E  = np.append(np.empty(0),0)
    while tn<t_final:
        tol = reps* abs(Un) + aeps
        #STAGES
        T1  = tn + 0*h
        T2  = tn + 1*h
        T3  = tn + 0.5*h
        K1  = Un
        K2  = Un + h*f(K1,T1)
        K3  = Un + h*(0.25*f(K1,T1) + 0.25*f(K2,T2) )
        #updating
        Un1 = Un + h*( 0.5*f(K1,T1) + 0.5*f(K2,T2) )
        Un1h = Un + h*( (1/6)*f(K1,T1) + (1/6)*f(K2,T2) + (2/3)*f(K3,T3) )

        En = abs(Un1 - Un1h)
        h_candidat = h_opt(h,tol,En,p)
        if En > tol:
            # we don't accept
            reject =  reject + 1
            h = h_candidat
            h_[-1] =  h # the last one get overwritten
            #print("print:"+str(tn))
        else:
            Un = Un1
            tn = tn + h
            h = h_candidat
            h_ = np.append(h_, h)
            U = np.append(U,Un1)
            t = np.append(t,tn)

        if tn+h > t_final:
            # this step make sure that we end at t_final and don't move further ahead.
            h = t_final - tn
            h_[-1] =  h

        E  = np.append(E,En) # for analysis purpose
        #print(str(round(100*tn/t_final,1))+"% :)")
        count = count + 1
        # the return variable is a tuple. Maybe go for a dictionary...?
    return t,U,E,h_,reject
```

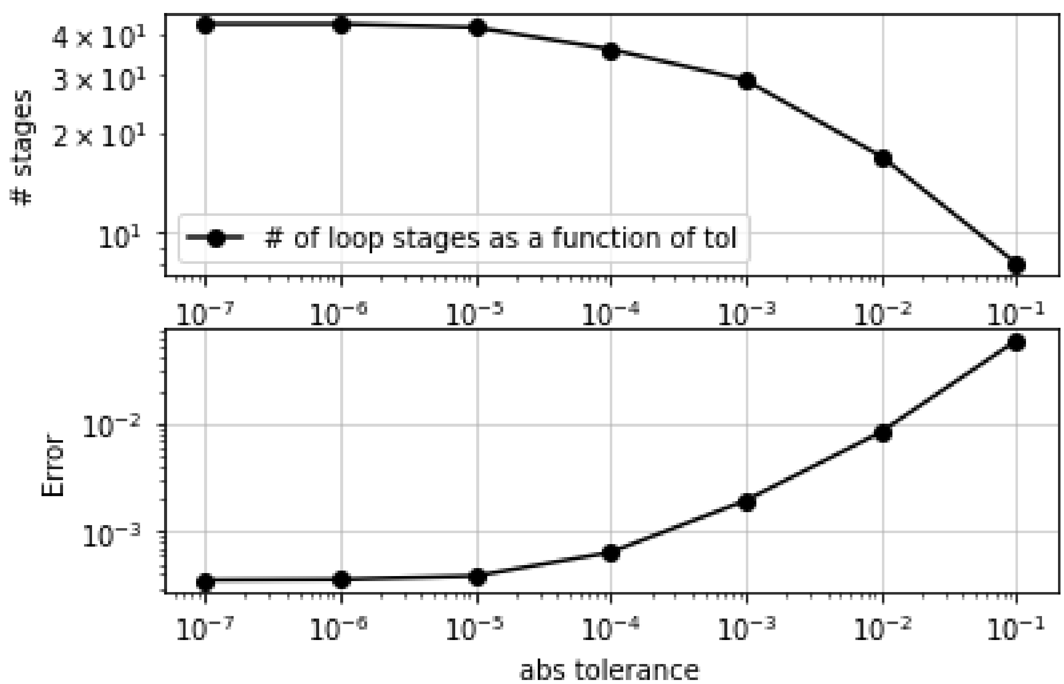
1.0.2 Some results and runs

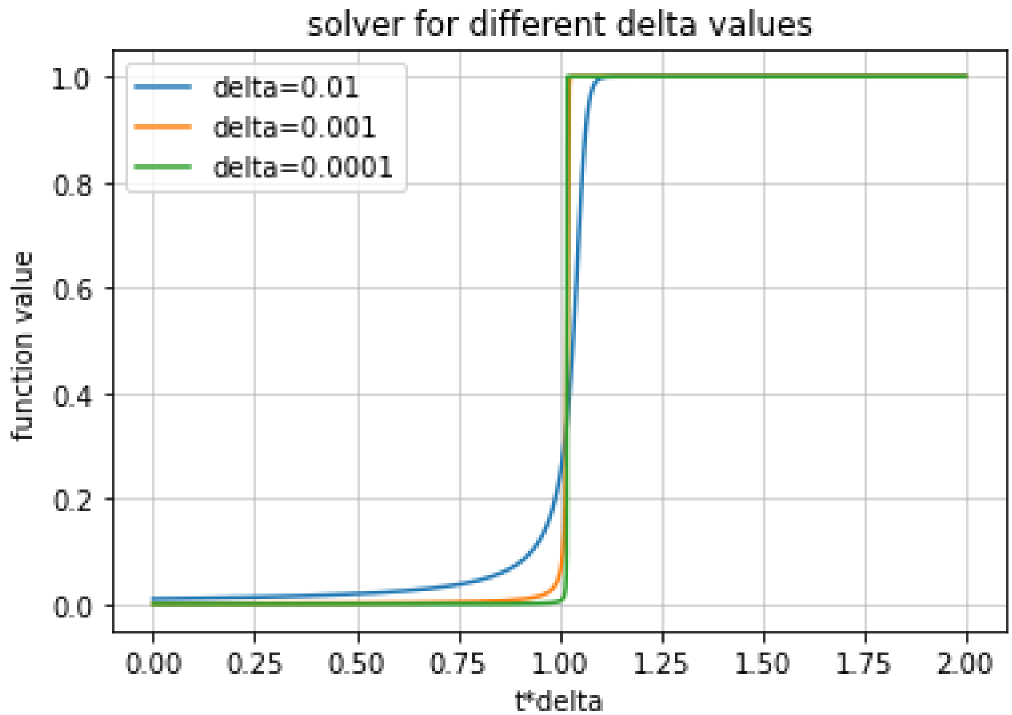
[8]:
```
def f(u,t): return u**2-u**3
def f2(t,u): return u**2-u**3 # for SciPy.

delta = 0.2
t_final = 2/delta;
IV   = delta
t0   = 0

sol = rk23(f,t0,t_final,IV,0.1)
t,U,h_,reject = sol[0], sol[1], sol[3],sol[4]


spsol = solve_ivp(f2,[t0,t_final],[IV], method='RK23',rtol = 1e-4, atol=1e-4) #␣
↪scipy solution
sph_ = spsol.t[1:] - spsol.t[:-1] # determining the stepsize of Scipy solver


# plot 1
plt.figure(1)
plt.clf()
plt.subplot(2,1,1)
plt.plot(spsol.t,spsol.y[0], color = "black", label="scipy,␣
↪#steps="+str(len(spsol.t)-1));
plt.plot(t,U,'o', color = "black", markerfacecolor ="none",label ="solver,␣
↪#steps="+str(len(t)-1));
plt.legend()
standardplot("t","function value","delta ="+str(delta))
plt.subplot(2,1,2)
plt.plot(t[0:-1],h_[0:-1],color="black",label ="solver")
plt.plot(spsol.t[0:-1],sph_,"--",color="black",label ="Scipy")
plt.xlabel("t");plt.ylabel("step size");plt.grid("lw=0.5")
plt.legend()


#plot2
#high resolution test
U_ref = solve_ivp(f2,[t0,t_final],[IV], method='RK23',rtol = 1e-12, atol=1e-12).
↪y[0,-1]
atol_ = np.array([1e-1,1e-2,1e-3,1e-4,1e-5,1e-6,1e-7])
control      = np.empty(0)
control2     = np.empty(0)
for i in atol_:
    tmp      = rk23(f,t0,t_final,IV,0.1, aeps = i)
    out      = len(tmp[0])-1+tmp[4]
    out2     = abs(tmp[1][-1]-U_ref)
```

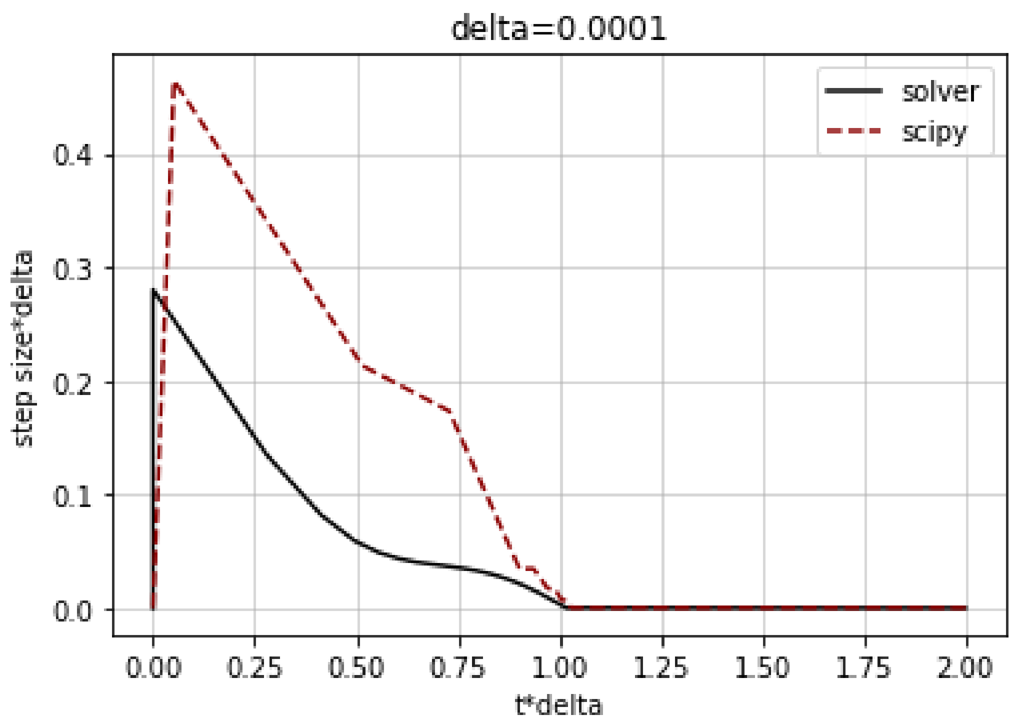
```
    control = np.append(control,out)
    control2 = np.append(control2,out2)

plt.figure(2)
plt.clf()
plt.subplot(2,1,1)
plt.loglog(atol_,control,'-o',color="black",label ="# of loop stages as a␣
↪function of tol")
plt.legend()
plt.xlabel("abs tolerance");plt.ylabel("# stages");plt.grid(lw=0.5)
plt.subplot(2,1,2)
plt.loglog(atol_,control2,'-o',color="black", label ="Error as a function of␣
↪tol")
plt.xlabel("abs tolerance");plt.ylabel("Error");plt.grid(lw=0.5)

#plot 3
plt.figure(3)
plt.clf()
delta_ = np.array([0.01,0.001,0.0001])
for i in delta_:
    tmp = rk23(f,t0,2/i,i,0.1)
    plt.plot(tmp[0]*i,tmp[1], label = "delta="+str(i))
plt.legend()
plt.grid(lw=0.5)
standardplot("t*delta", "function value", "solver for different delta values")



#plot 4
plt.figure(4)
delta = 0.0001
sp = solve_ivp(f2,[t0,2/delta],[delta], method='RK23',rtol = 1e-4, atol=1e-4)
sph_ = sp.t[1:] - sp.t[:-1] # determining the stepsize of Scipy solver
plt.clf()
plt.plot(tmp[0]*delta,tmp[3]*delta, label="solver",color="black")
plt.plot(sp.t[0:-1]*delta,sph_*delta,"--", label="scipy",color="darkred")
plt.xlabel("t*delta"); plt.ylabel("step size*delta")
standardplot("t*delta", "step size*delta", "delta="+str(delta))
plt.legend()
```

[8]: <matplotlib.legend.Legend at 0x1ba8f17ec40>


delta =0.2
scipy, #steps=15
solver, #steps=29
function value
step size
solver
Scipy
t


stages
of loop stages as a function of tol
Error
abs tolerance


solver for different delta values
delta=0.01
delta=0.001
delta=0.0001
function value
t*delta
0.0
0.2
0.4
0.6
0.8
1.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


delta=0.0001
solver
scipy
step size*delta
t*delta
0.0
0.1
0.2
0.3
0.4
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

[ ]:

```

```