Contents

1 Introduction **1**

2 2-step explicit method **2**
2.1 2-step Adams-Bashforth 3

3 How the code works **3**
3.1 Flow chart 5
3.2 Testing the code 6

4 Zero-stability **7**

5 Checking the order **9**
5.1 Fixed step-size 9
5.2 Variable step-size: error vs tolerance 10
5.3 Variable step-size: error vs step-size 11
5.4 Solution with different 2-step explicit methods 12

6 How to use the program **13**

7 Conclusion **14**

8 Appendix **16**

Abstract

This paper shows the Python implementation of a new way of constructing a variable multistep ODE solver which is not based on extending fixed step-size methods but rather on defining interpolation and collocation conditions for each new point. The implementation allows for any explicit 2-step method of order 2.

1 Introduction

There are many different ways of solving Initial Value Problems (IVP) numerically and some of those are better for certain problems than others. The goal of this project is to convert an explicit 2-step ODE solver from Matlab into Python. For each point that this method gives the program will create a piecewise polynomial of degree two. In the end the program will give an explicit solution to the IVP problem.

This paper will also discuss zero-stability and show which explicit 2-step methods are zero-stable. We will also discuss the consistency and order of the method.

In the end we want to show that it is possible to construct implementations of a variable step-size multistep method with a technique that is not based on extending fixed step-size formulas.

The reader should be informed that the code is not in professional form but is still in a developmental stage.

2 2-step explicit method

There are different methods constructed to solve initial value problems of the form

$$y' = f(t, y), \quad y(t_0) = y_0 \tag{1}$$

An explicit 2-step linear method has the general formula

$$y_n + \alpha_1 y_{n-1} + \alpha_0 y_{n-2} = h_n(\beta_1 f(t_{n-1}, y_{n-1}) + \beta_0 f(t_{n-2}, y_{n-2})) \tag{2}$$

Because this is a numerical method, the solution calculated by (2) is

$$y_n \approx y(t_n)$$

where

$$t_n = t_{n-1} + h_n.$$

This method starts with the initial points at t_0, t_1 and takes small step forward in time to calculate the next solution at t_2. Then it will advance by calculating the solution at $t_3, t_4, ..., t_f$.

The quantity h_n is called the step-size at step n. Although explicit methods can solve initial value problems by themselves, they are usually used in tandem with implicit methods in order to predict the solutions at each point. The explicit-implicit pair is called a predictor-corrector scheme.

The difference of an single-step method such as Euler´s method and an multistep method such as Adams-Bashforth 2-step method is that multistep methods use the previous information to calculate the new point while single-step methods discard all previous information except the last calculated point before calculating the next step [3].

The fixed 2-step method (2) has the formula

$$y_n + \alpha_1 y_{n-1} + \alpha_0 y_{n-2} = h(\beta_1 f(t_{n-1}, y_{n-1}) + \beta_0 f(t_{n-2}, y_{n-2}))$$

with

$$t_i = t_0 + ih.$$

There is a need to change the step-size after every step in order to make the method adaptive, and then the method becomes formula (2) with
$h_n = t_n - t_{n-1}$. The coefficients $\alpha_0, \alpha_1, \beta_0$ and β_1 are no longer constant, but are a function of the last two step-sizes. In [4] these methods are constructed as collocation methods that are intrinsically of variable step-size. The procedure is explained in section 3.

2.1 2-step Adams-Bashforth

The 2-step Adams-Bashforth method needs two values y_{n-2} and y_{n-1} to be able to compute the next step y_n [1],

$$y_n = y_{n-1} + \frac{3}{2}h_n f(t_{n-1}, y_{n-1}) - \frac{1}{2}h_n f(t_{n-2}, y_{n-2}). \tag{3}$$

Adams-Bashforth methods are the most commonly used explicit multistep methods and they are used as predictors for the implicit Adams-Moulton methods.

As the inital value problem only provides with one value and the method requires one more initial value, one can use different numerical methods such as Euler´s or a Runge-Kutta method to find the second initial value.

3 How the code works

The main purpose of the code is to solve initial value problems of the form (1) in an interval $t \in [t_0, t_f]$.

As mentioned before, the program will use an explicit 2-step method. However the program does not directly use formula (2). According to Skeel [2] there is no general method for the construction of variable step-size multistep methods other than extending constant step-size formulas. Nevertheless, in [4] it is shown that you can construct explicit multistep methods as collocation polynomials such that interpolation conditions at previous points are satisfied. This shows that you can construct a method not depending on extending fixed step-size formulas.
Below is a list of the conditions that I used in the program [4, pp16]

$$\begin{aligned}
&P \in \Pi_2 \\
&P'(t_{n-1}) = y'_{n-1} \\
&P(t_{n-1}) = y_{n-1} \\
&\cos(\theta)P(t_{n-2}) - \cos(\theta)y_{n-2} + \sin(\theta)h_{n-1}(P'(t_{n-2}) - y'_{n-2}) = 0
\end{aligned}$$

where

$$y'_n = f(t_n, y_n).$$

These four conditions define a polynomial such that the solution at t_n is given by $P(t_n)$, that is, $y_n = P(t_n)$.

The value of θ will define the method. By putting $\theta = \frac{\pi}{2}$ we will get the 2-step Adams-Bashforth method given by formula (3). All possible 2-step explicit methods can be obtained by changing the value of θ.

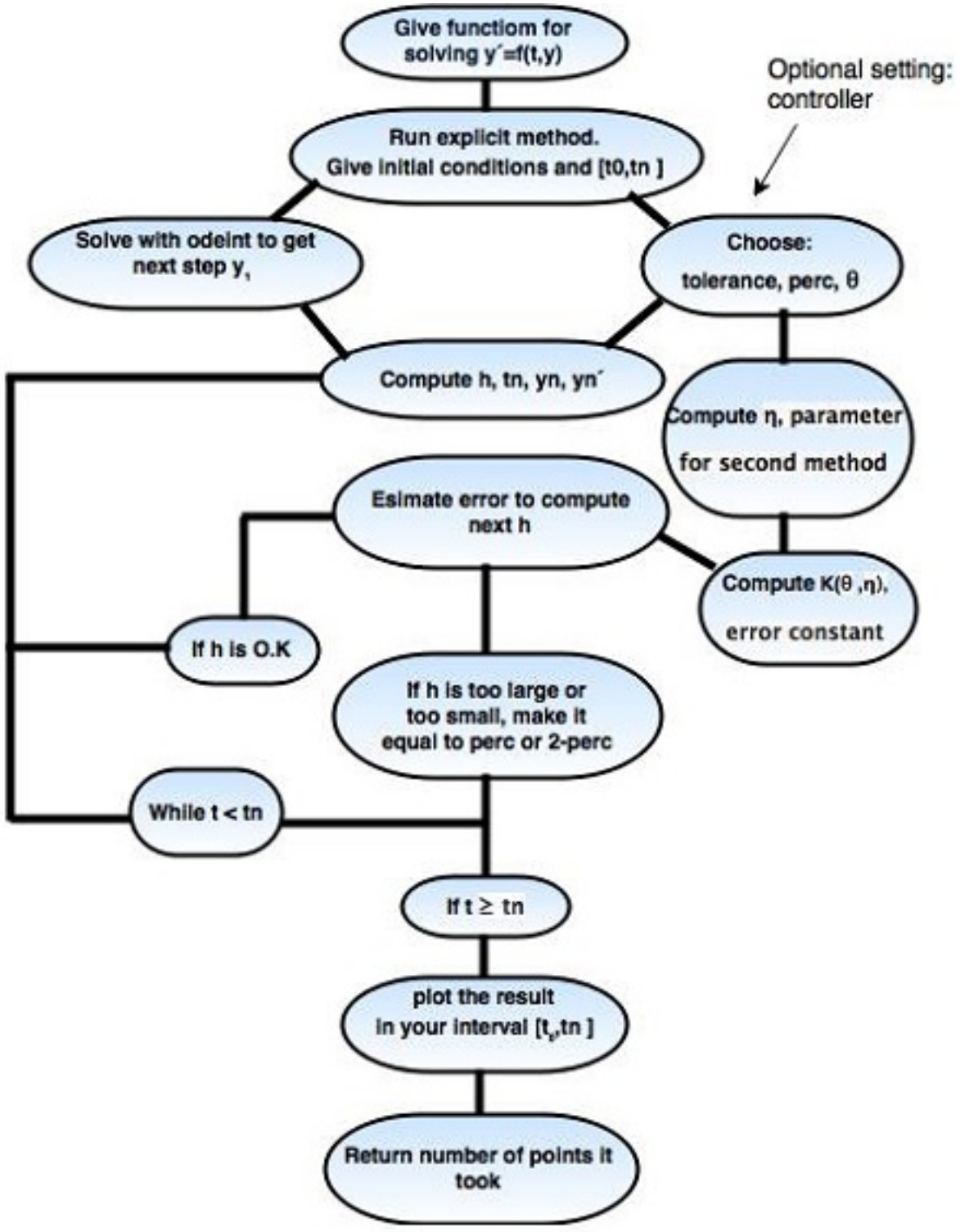
The size of the step-size h_n is another key aspect of the solver. You can choose to have fixed or variable step-sizes. By choosing variable step-size the code runs smoother. The variable step-size is constructed using control theory. Bascially what the code does is that it takes a smaller h_n if the function changes rapidly in a time interval and the other way around if nothing particular happens. In order to do this, it estimates the error at each step, and tries to keep it equal to the user-given tolerance. The same problem is solved with two different 2-step methods and from the results the error can be estimated.

The code also uses a variable called perc. What perc does is that it gives the user the freedom to chose what percentage of h_n is allowed to change. With the error estimate, the new step-size is proposed. If the proposed h_n is too large or too small it is made equal to a predefined fraction of the previous step-size.

After the points are calculated the program uses these points and the interpolation conditions above to create a polynomial of degree two over the subinterval $[t_{n-2}, t_{n-1}]$. The program does this for every point and in the end you get a graph which is constructed with a piecewise polynomial of degree two.

On the next page there is a flow chart of how the code roughly works.

3.1 Flow chart



3.2 Testing the code

To illustrate how the code works we will run it on an ordinary differential equation. Say that we are asked to solve the following system

$$y' = \begin{bmatrix} 6 & 3 \\ -2 & 1 \end{bmatrix} y, \quad y(20) = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

The analytic answer for this problem is

$$y = \begin{bmatrix} 5e^{3t-60} - 6e^{4t-80} \\ -5e^{3t-60} + 4e^{4t-80} \end{bmatrix}$$

The first thing you should do in the program as the flow chart says is to construct the function in the following way.

```
def function(y,t):
    return array([6*y[0]+3*y[1],-2*y[0]+1*y[1]])
```

After this step you can run your 2-step method. As the flow chart says we have to give the initial conditions, the points where you want to plot your function, for which θ you want to run your code (that is, what method you want to use), what tolerance you want to have and choosing perc. How you run the class will be explained in section 6. For this example it will look the following when I run the class.

```
a=IVP_Solver(function,array([-1,-1]),pi/2,1.e-8,0.8)
a(20,22)
```

Here I wrote my initial condition in the program and we choose to solve it in an interval of lenght two. The method defined by $\theta = \frac{\pi}{2}$ is Adams-Bashforth, the tolerance was set to 10^{-8}, and perc was set to 0.8. The solver used 7285 steps.

Figure 1 shows the relative error calculated as the absolute value of the difference between the exact solution and the computed solution.

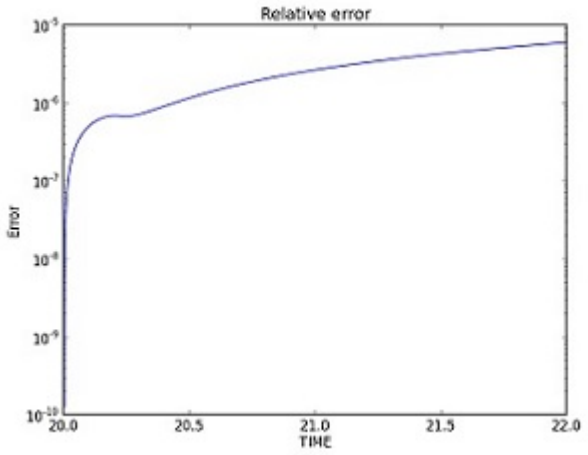


Figure 1: The relative error for the first component.

Even though the solution for this ordinary differential equation grows very fast as t grows the relative error is still small.

4 Zero-stability

The solution with a k-step method depends on $y_{n-k}, ...y_{n-2}, y_{n-1}$. Therefore we need to know if the numerical solution is stable with respect to perturbations in the starting values. A linear multistep method is said to be zero-stable if the method is consistent for the differential equation $y' = 0$ [5]. This is equivalent to checking if the roots of the characteristic polynomial

$$p(z) = z^n + \sum_{k=0}^{n-1} \alpha_k z^k$$

have modulus less than or equal to 1 and those with modulus 1 are simple roots. We then say that the root condition is satisfied [5]. If the root condition is satisfied then the method is zero-stable.

Now we will test zero-stability for our method. Notice that by choosing $\theta = \frac{\pi}{2}$ we got Adams-Bashforth method but what happens if we take a different θ? Figure (2) shows a graph of the code solving the differential equation

$$y' = -y, \quad y(0) = 1$$

for different θ. The global error $E(h)$ is

$$E(h) = \frac{||\tilde{x} - x||_2}{m}, \tag{4}$$

where $\tilde{x}$ is the vector of computed solutions, x is the vector of exact solutions, and m is the number of points at which the solution was approximated.

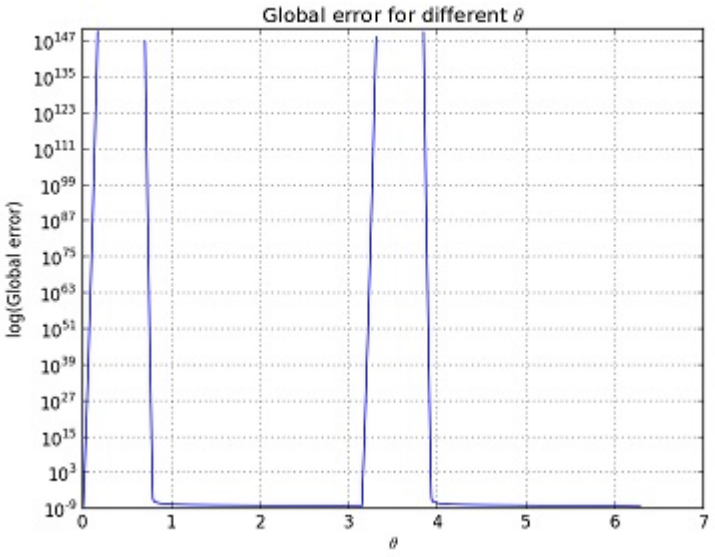


Figure 2: The picture shows us that the methods with $\theta \in [1, 3] \bigcup [4, 2\pi]$ are zero stable, but that there are regions where the methods defined by θ are not stable.

Beause the characteristic polynomial is of degree 2, its roots can be obtained analitically. Since our method is of order two we have [4, pp8].

$$\begin{aligned}\alpha_1(\theta, r) &= \frac{2\sin\theta + (r^2-1)\cos\theta}{\cos\theta - 2\sin\theta} \\ \alpha_0(\theta, r) &= -\frac{r^2\cos\theta}{\cos\theta - 2\sin\theta}.\end{aligned}$$

Where $r = h_{new}/h_{old}$. Note that when $r = 1$ the step-size is kept fixed.
From this we can build the polynomial $\rho(x) = x^2 + \alpha_1 x + \alpha_0$. All methods should satisfy the root condition at least in a vecinity of $r = 1$.
We can write ρ as

$$\rho(x) = x^2 + \frac{2\sin\theta}{\cos\theta - 2\sin\theta}x - \frac{\cos\theta}{\cos\theta - 2\sin\theta}.$$

The roots of ρ should satisfy the root conditon. We know that one root of ρ is always 1, and thus we can use this information to get the other root. Define the second root as z then

$$\begin{aligned}(x-1)(x-z) &= x^2 + \frac{2\sin\theta}{\cos\theta - 2\sin\theta}x - \frac{\cos\theta}{\cos\theta - 2\sin\theta} \\ z &= -\frac{\cos\theta}{\cos\theta - 2\sin\theta}.\end{aligned}$$

Thus we have to choose a θ such that the root condition (5) is satisfied, that is,

$$z(\theta) = \left|-\frac{\cos\theta}{\cos\theta - 2\sin\theta}\right| < 1. \tag{5}$$

By solving equation (5) we find that if $\theta \in [\frac{\pi}{4}, \pi]$ then the method is zero-stable when $r = 1$. Thus, θ must be in the open interval $(\frac{\pi}{4}, \pi)$.

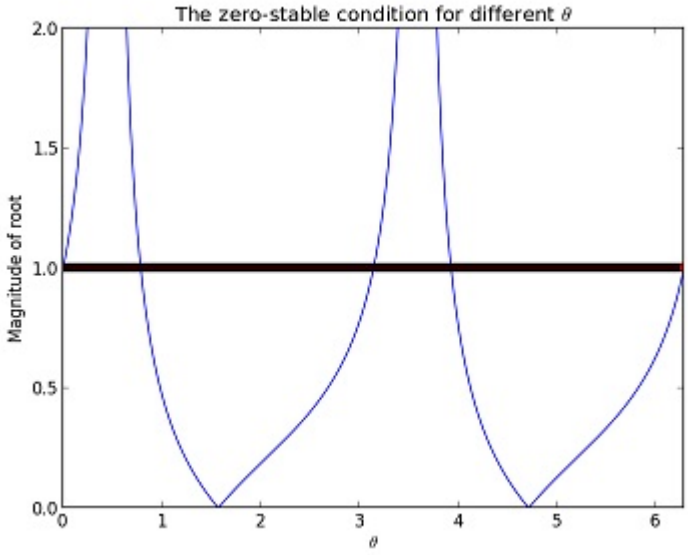


Figure 3: Picture of $\theta \in [0, 2\pi]$ where the root condition is satisfied when $r = 1$. The graph under the line satisfies the root condition, the rest does not.

5 Checking the order

We will now check that the expected order 2 of these methods is observed when this solver is used.

If the code does not give a solution of order 2 then the code is not working properly. Down below are different tests that show that the solution obtained with our code is of order two. The tests solve the differential equation

$$y' = y, \quad y(0) = 1, \tag{6}$$

in the interval $[0, 2]$, for $\theta = \frac{\pi}{2}, \frac{\pi}{3}, \frac{4\pi}{3}$. All methods gave similar results. The order was checked for both fixed and variable step-sizes.

5.1 Fixed step-size

Since equation (4) is of order 2, we can write the global error as

$$\begin{aligned} E(h) &= ch^2 + O(h^3) \\ E(h) &\doteq ch^2 \\ \log(E) &\doteq \log(c) + 2\log(h) \end{aligned}$$

which we can rewrite as $y = K + 2x$, with $x = \log(h)$ and $y = \log(E)$.
This means that if we run the program and plot $\log(E)$ vs $\log(h)$ for different fixed-step sizes we should get a graph with slope two. Figure 4 shown that our code gave the expected result.

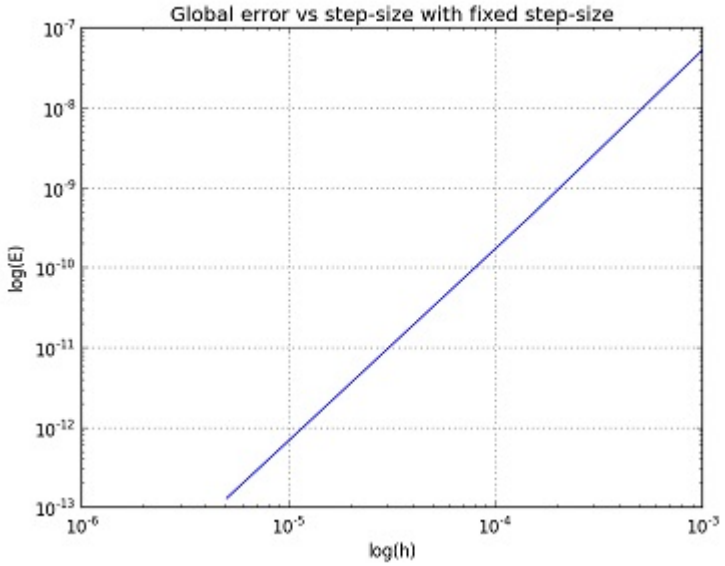


Figure 4: Graph with logarithmic x-axis and y-axis. Problem (6) was solved with fixed step-sizes, for 7 different values of h.

5.2 Variable step-size: error vs tolerance

When we change the tolerance of our method we would like to see a predicted effect on the global error. The controler in the code tries to make the local error equal to the tolerance. In a method of order 2, the local error is of order 3. This implies that

$$\text{local error} = O(h^3)$$
$$Tol^{\frac{2}{3}} \doteq c_1 h^2.$$

Earlier we said that the global error was of order two, which implies that

$$E(h) \doteq c_2 h^2 = \frac{c_2}{c_1} c_1 h^2 = K * Tol^{\frac{2}{3}}$$
$$\log(E) = \log(K) + \log(Tol^{\frac{2}{3}})$$

This can be written as $y = K + x$, where $x = \text{Tol}^{\frac{2}{3}}$. Thus when $\log(E)$ is plotted vs $Tol^{\frac{2}{3}}$, the line should have slope 1. This test shows that the global error depends linearly on $Tol^{\frac{2}{3}}$, so that changing the tolerance leads to a change of the global error in a predicted way.

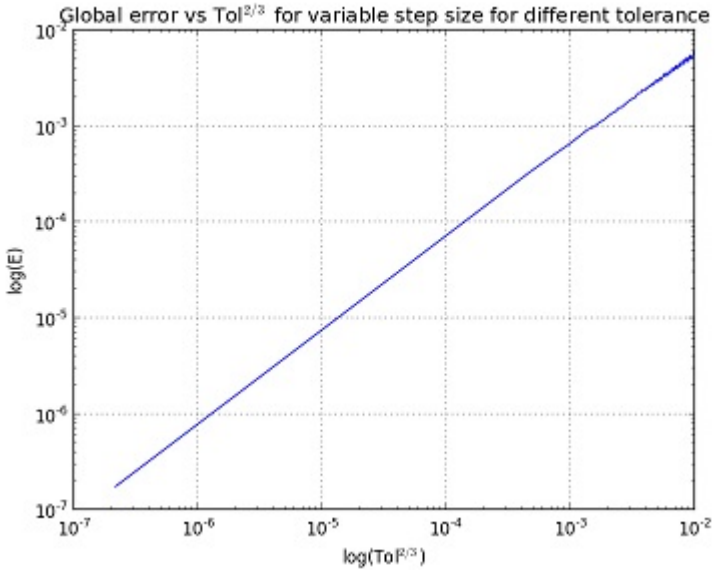


Figure 5: Graph with logarithmic x-axis and y-axis. Problem (6) was solved with variable step-sizes by using 8 different values of the tolerance. Note that the slope is 1.

5.3 Variable step-size: error vs step-size

We want to repeat what we did for fixed step-size in figure 4, but now for variable step size. In this test we are going to change the tolerance but plot the global error vs the average of h,

$$H = ||h||_1/\mathrm{m}.$$

The tolerance goes from 10^{-4} to 10^{-12}(8 trials). The parameter perc was set to 0.8, which means that at each step h is allowed to change between $0.8h$ and $1.2h$.

If we plot the global error against H in a log log plot, we get a line with slope two as shown in figure 6. This means that the order of the method is preserved when the step-sizes are varied.

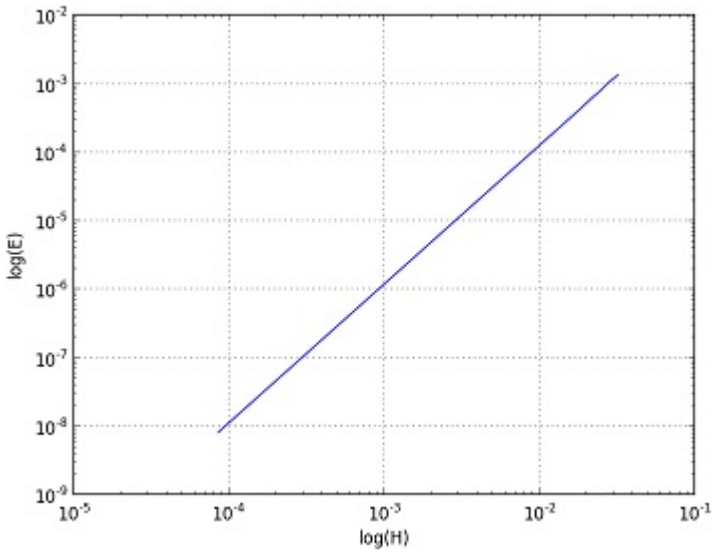


Figure 6: Graph of the global error vs average h for variable step-size. Note that the slope is two, as expected.

5.4 Solution with different 2-step explicit methods

By chosing different θ in the interval $(\frac{\pi}{4}, \pi)$ we get different methods, and we want to see if all θ give similar results when problem (6) was solved in the interval $[0, 2]$. Table 1 shows the number of steps required by each method and the root mean square of the global error of the solution.

θ	Number of steps	Mean global error
1.1	1507	7.1282e-06
1.3	1378	6.4996e-06
$\frac{\pi}{2}$	1288	6.0567e-06
2.9	1042	4.8526e-06
4.2	1551	7.3447e-06
5.1	1211	5.6815e-06

Table 1: Numbers of steps required to solve problem (6), and global error at the endpoint at the interval.

We observe that the different methods will use different number of steps, but they are of the same magnitude. The number of steps is dependent on the method, but also on the problem solved. The global error at the final point is similar for all methods we tried.

6 How to use the program

To be able to run the code for your initial value problem you need to have Python installed on your computure. If we assume that you have imported the code listed in the appendix the code should not give any error messeges. This code can be downloaded from

`http://www.maths.lu.se/staff/carmen-arevalo/downloads/`

To be able to solve an IVP the first thing you should do is to define the right-hand side of your IVP. Say that we want to define the following equation

$$y' = \begin{bmatrix} -1 & 1 \\ 1 & -3 \end{bmatrix} y.$$

Then you write

```
def function(y,t):
        return array([-y[0] +1*y[1],y[0]-3*y[1]])
```

The $y[0]$ is the first column, $y[k-1]$ is the k:th column. To switch row you need to have a comma as seen in the example above.

After this you need to start the class IVP_Solver. The class has the inputs

function	Your right-hand side function given by the IVP
y_condition	The initial conditions given by your IVP
theta	For which method you want to run the solver
TOL	What tolerance you want to have
perc	Percentage of h allowed to change
t_start	The starting value of the time
t_end	The end value of the time

Say that we want to solve the IVP given by section 3.2. To solve this IVP with the program you need to change the function to.

```
def function(y,t):
    return array([6*y[0]+3*y[1],-2*y[0]+1*y[1]])
```

then run the program. Then you have to initiate the class by writing

```
some_variable=IVP_Solver(function,y_condition,theta,TOL,perc)
some_variable(t_start,t_end)
```

Where some_variable can be any object for example a letter or a word.

For this IVP it would be the following

```
some_variable=IVP_Solver(function,array([-1,-1],pi/2,1.e-8,0.8)
some_variable(20,22)
```

The class returns the solution of your IVP and the amount of steps it took. If you want to see the values of the solution to your IVP you write

```
some_variable.y_finale
```

Notice that theta, TOL and perc are variables that you can change to your demand.

7 Conclusion

We have developed a working Python code for the family of all variable step-size 2-step explicit methods of order two. The method are not based on extending fixed step size formulas but rather interpolating and collecting information to calculate the next point.

We have shown that the order of the methods is 2 when the solver is used for fixed and variable step-size where the global error depends linearly on some tolerance. We have shown for which θ the solver is zero-stable and which conditions they must satisfy. Solving differential equations with different methods requires different amount of steps, but they are of the same magnitude.

The program can easly be extended to construct higher order of variable step-size multistep methods.

References

[1] Bashforth, Francis, Adams,J.C, *An attempt to test the theories of capillary action by comparing the theoretical and measured forms of drops of fluid. With an explanation of the method of integration employed in constructing the tables which give the theoretical forms of such drops*, Cambridge, university Press, 1883.

[2] Skeel.R.D, *Construction of variables-stepsize multistep formulas*, Math.Comp., 47(176)(1986, pp503-510,S45-S52)

[3] `http://en.wikipedia.org/wiki/Linear_multistep_method`

[4] Arévalo.C, *Parametric variable step-size formulation of multistep methods.* Unpublished Paper,2015

[5] Süli, Endre; Mayers, David (2003), *An Introduction to Numerical Analysis*, Cambridge University Press, ISBN 0-521-00794-1.

8 Appendix

```
from __future__ import division
from scipy import *
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
```

```
""" Hi and welcome to this IVP solver program. To be able to run this program
you have to have an IVP problem (or system of them) in the following form
y´=f(t,y) with y0=f(t0). This code is a 2-step explicit method. This code uses
piecewise polynomials of degree two to illustarte the desire result.
"""

""" The first thing you have to do to run this program is to write your
function. To solve your problem you need to write your IVP below in the
constructed code. Your function should be an array. The dimension of the array
will depend on how your function looks like. If you just have one equation to
solve write it in the follwing way "array([-1*y[0]])". The y[0] tells the program that to
call that specific y. If you have a system of equationfor example a 2x2 matrix
of the form
 |-y y |
 |y -3y|
 then you write "array([-y[0] +1*y[1],y[0]-3*y[1]])"
 Now you are half way to calculate your function"""
def function(y,t):
    """ Give your IVP problem here."""

    #return array([0*y[0]+1*y[1],-4*y[0]+0*y[1]])
    return array([y[0]])


class IVP_Solver(object):


    """ To run the program and get the answear for your problem you have to
call the
program ´a=IVP_solver(function,y_condition,theta,TOL,perc)´ then a(t_start.t_end)

The program returns a plot of your answear in your interval and how many points
it took to get there.

"""

    def __init__(self,function,y_condition,theta,TOL,perc):

        self.y_condition=y_condition
        self.function=function
        self.theta=theta
        self.TOL=TOL
        self.perc=perc
```

```
def __call__(self,t_start,t_end):

    return self.Explicit_2_step_method_with_splines(self.function,
           self.y_condition,t_start,t_end,self.theta,self.TOL,self.perc)




def error_const(self,theta,eta):
    """Calculates the error constant.
        """
    self.theta=theta
    self.eta=eta
    self.K=0
    self.k=len(theta) +1

    if self.k==2:
        self.cx= cos(theta)
        self.sx= sin(theta)
        self.cy = cos(eta)
        self.sy = sin(eta)
        self.Kx=(2*self.cx-5*self.sx)/(self.cx-2*self.sx)/6
        self.Ky=(2*self.cy-5*self.sy)/(self.cy-2*self.sy)/6
        self.K= self.Kx/(self.Ky-self.Kx)

    return self.K

def pol(self,theta,h,y,yp):
    """Returns a points which helps with constructing a piecewise polynomial between two
    given points
   """

    self.h=h
    self.y=y
    self.yp=yp
    self.kk= len(theta) +1
    if self.kk ==2:
        self.c=cos(theta)
        self.s=sin(theta)
        self.p = (self.c*(y[-2]-y[-1]+h[-2]*yp[-1])+self.s*h[-2]*
                  (yp[-2]-yp[-1]))/(h[-2]**2*(self.c-2*self.s))
    return self.p



def angle2(self,theta):
    """Returns eta, eta depends what your theta is. If theta=pi/2 the method
will be Adam Bashforth."""

    self.theta=theta
    self.k= len(theta)+1
    self.eta=theta.copy()
    for j in range(self.k-1):
        if theta[j] <= 5*pi/8:
            self.eta[j]=13*pi/16
        else:
            self.eta[j]=7*pi/16
    return self.eta
```

```
def start_(self,theta,t,h,y,yp):
    """This function helps you to start with the 2-step explicit method.
Since in your problem you are just given you one point(we need two points to
start the process) this program calculates the second point with help of the
inbuild function "odeint" (Runge-Kutta). It also gives you the
second point in time you need to have."
"""
    self.theta=theta
    self.t=t
    self.h=h
    self.y=y
    self.yp=yp

    self.k = len(theta)+1
    for j in range(self.k-1):
        t.append(t[-1]+h[-1])
        self.Y=odeint(self.function,y[0],t)
        y.append(self.Y[-1])
        self.f=self.function(y[-1],t[-1])
        yp.append(self.f)

    return t,y,yp
```

```
def controller(self,parvec,theta,k, TOL, est, r, perc,w):
    """The function controller choses how big the next stepsize should be.
    If the given function grows fast or slow in an interval the controller will
    take an smaller h, if the function does not change much in an interval the
    controller will take a bigger h. This function estimates the error at easch
    step, and tries to keep it equal to the tolerance"""

    self.ord_ = len(theta) + 1
    if self.k== self.ord_ +1:
        self.cerrold=1
    else:
        self.cerrold= (TOL/est[-2])**(1/(self.ord_+1))
    if est[-1]==0:
        est[-1]=1e-18
    self.cerr= (TOL/(est[-1]))**(1/(self.ord_+1))
    self.convec =array([self.cerr,self.cerrold,r])

    r= prod(array([self.convec[0]**self.parvec[0],self.convec[1]**
        self.parvec[1],self.convec[2]**self.parvec[2]]))

    if r<perc:
        r=perc
        w=w+1
    elif r > 2-perc:
        r=2-perc
        w=w+1
    return r,w
```

```
def Explicit_2_step_method_with_splines(self,function,y_condition,t_start,
                                        t_end,theta,TOL,perc):
    """This program uses other programs below. The program returns a plot of
the numerical answear for your ODE problem and returns how many points it
took to get there. This process is an explicit method, the process stopiterating
until the given time interval is reached. Changing theta will give different
explicit methods, some are zero-stable some are not. "theta=array([pi/2]" is
Adams-Bashforth method.
"""
    self.function=function
    self.y_condition=y_condition
    self.t_start=t_start
    self.t_end=t_end
    self.parvec=array([1,1,0])/6
    self.theta=array([theta])
    self.ord_=len(self.theta)+1
    self.TOL = TOL
    self.perc=perc
    self.w=0
    self.eta=self.angle2(self.theta)
    self.K=self.error_const(self.theta,self.eta)
    self.h=[self.TOL**(2/(self.ord_+1))]
    self.t=[t_start]
    self.y=[y_condition]
    self.yp=[self.function(y_condition,self.t[0])]


    self.t,self.y,self.yp=self.start_(self.theta,self.t,self.h,self.y,self.yp)

    self.est=[0]


    self.r=1
    self.k=self.ord_



    while self.t[self.k-1]<t_end:

        self.k=self.k+1

        self.h.append(self.r*self.h[self.k-3])

        self.t.append(self.t[self.k-2]+self.h[self.k-2])


        self.px = self.pol(self.theta,self.h,self.y,self.yp)

        self.py = self.pol(self.eta,self.h,self.y,self.yp)
        self.y.append(self.px*self.h[self.k-2]**2 +self.yp[self.k-2]*
                      self.h[self.k-2]+self.y[self.k-2])
        self.yp.append(self.function(self.y[self.k-1],self.t[self.k-1]))


        self.y_1= self.py*self.h[self.k-2]**2 +self.yp[self.k-2]*\
                            self.h[self.k-2]+self.y[self.k-2]
        self.est.append(norm(self.K*(self.y_1-self.y[self.k-1])/
                               (abs(self.y[self.k-1])+1e-3)))


        self.r,self.w=self.controller(self.parvec,self.theta,self.k,
                            self.TOL, self.est, self.r, self.perc,self.w)
```

```
        if len(self.t) >=100000:
            return "To big interval or wrong function!"


    if self.t[-1]>t_end:
        self.t[-1] = t_end


        self.h[-1] = self.t[-1]-self.t[-2]

        self.y=self.y[:-1]

        self.yp=self.yp[:-1]
        self.px = self.pol(self.theta,self.h,self.y,self.yp)
        self.py = self.pol(self.eta,self.h,self.y,self.yp)


        self.s = self.yp[-1]*self.h[-1]+self.y[-1]


    for j in range(2,self.ord_+1):
        self.s = self.s + self.px[self.ord_-j]*self.h[-1]**j



    self.y.append(array(self.s))


    self.yp.append(self.function(self.y[-1],self.t[-1]))
    self.z = self.yp[-1]*self.h[-1]+self.y[-1]
    for j in range(2,self.ord_+1):
        self.z = self.z + self.py[self.ord_-j]*self.h[-1]**j


    self.y_finale=zeros(shape=(len(self.t),len(self.y_condition)))
    for j in range(len(self.t)):
        for k in range(len(self.y_condition)):
            self.y_finale[j,k]=self.y[j][k]




    for i in range(len(self.y_condition)):
        plt.plot(array(self.t),self.y_finale[:,i])
        plt.ylabel('Y-VALUES')
        plt.xlabel('TIME')
        plt.title('Result of your IVP')
        plt.show()

    return len(self.t)
```