БЪЛГАРСКА АКАДЕМИЯ НА НАУКИТЕ • BULGARIAN ACADEMY OF SCIENCES

ПРОБЛЕМИ НА ТЕХНИЧЕСКАТА КИБЕРНЕТИКА И РОБОТИКАТА, **62**
PROBLEMS OF ENGINEERING CYBERNETICS AND ROBOTICS, **62**

София • 2010 • Sofia

Using Implicit Runge-Kutta Methods for Multi Precision Solving of Nonlinear Differential Equations

Velichko Djambov

Institute of Information and Communication Technologies, 1113 Sofia
E-mail: vili_jambov@abv.bg

I. General data about the implicit Runge-Kutta schemes

Some basic notions and results are presented herein, needed for the construction of a procedure of the type of Runge-Kutta implicit scheme. For more thorough analysis of the notions and methods, connected with stability, you may see [1, 2].

Let the system $\dot{y} = f(t, y(t))$ be given, for which we would like to solve Cauchy's problem for $0 \le t \le T$ and $y(0) = y_0$. We assume that $y(t)$ is a real m-dimensional vector.

I.1. Structure of Runge-Kutta methods

The single-step Runge-Kutta methods are presented as an implicit relation (between y_{n+1} and y_n, which denotes the approximate values for the successive steps)

(1) $$y_{n+1} = y_n + \tau\,\Phi[\tau, y_n, y_{n+1}], \quad \tau = t_{n+1} - t_n \quad (n \ge 1),$$

where y_n denotes the approximation of $y(t)$ for t_n. The length of step τ can be altered by n.

One single-step Runge-Kutta method is described with the help of the formula

(2) $$y_{n+1} = y_n + \tau \sum_{i=1}^{s} b_i k_i\,,$$

where

(3) $$k_i = f(t_n + c_i\tau, y_n + \tau\sum_{j=1}^{s} a_{ij}k_j)\,.$$

Formula (2) is called *s*-staged and it is based on *s* calculations of function *f* for the derivative of the solution. In the case when a_{ij} = 0 for $j \geq i$, the coefficients k_i may be explicitly calculated from the values of k_1, …, k_{i-1}. Similar formulae are called *explicit*. In case a_{ij} = 0 at $j > i$, but $a_{ii} \neq 0$, each k_i is implicitly defined by the equation

(4) $$k_i = f(t_n + c_i\tau, y_n + \tau\sum_{j=1}^{i-1} a_{ij}k_j + \tau a_{ii}k_i)\,.$$

This requires calculation of approximate values for k_i. Such Runge-Kutta methods are called *diagonally implicit.* The execution of one step for such method requires the solution of *s* nonlinear systems of algebraic equations of *m*-th order with the help of an iterative procedure. The methods, which are not explicit, or are diagonally implicit, are called *implicit.* All k_{i} must be simultaneously calculated in implicit methods. Hence, one step of an implicit method requires the solution of a nonlinear system of algebraic equations of m_s order. The coefficients in formulae (2) and (3) usually satisfy the condition

(5) $$c_i = \sum_{j=1}^{s} a_{ij} \quad (1 \leq i \leq s)\,.$$

In order to present a Runge-Kutta method (by its coefficients), the so called Butcher table is used:

$$\frac{c\,|\,A}{|\,b^{\mathrm{T}}} = \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}.$$

I.2. Definition of an approximation degree. Simplifying conditions

If the following is assumed for a local error in discretization

(6) $$\hat{l}_{n+1} = y(t_{n+1}) - \hat{y}_{n+1}\,, \quad \hat{y}_{n+1} = y(t_n) + \tau\,\Phi[\tau, y(t_n), \hat{y}_{n+1}]\,,$$

the approximation degree of the method is defined as the biggest non-negative integer number *p*, for which

(7) $$\hat{l}_{n+1} = O(\tau^{p+1}), \quad \tau \to 0\,.$$

For the methods considered the approximation degree may be determined with the help of Butcher simplifying conditions.

It is known that the *s*-stage method of Runge-Kutta meets the condition

(8) $$A(\xi), \quad \text{if} \quad \hat{l}_{n+1} = O(\tau^{\xi+1}),$$

(9) $$B(\xi), \quad \text{if} \quad \sum_{i=1}^{s} b_i c_i^{k-1} = \frac{1}{k} \quad (1 \leq k \leq \xi),$$

(10) $$C(\xi),\quad \text{if}\quad \sum_{j=1}^{s} a_{ij}c_j^{k-1} = \frac{1}{k}c_i^k \quad (1 \le k \le \xi,\ 1 \le i \le s),$$

(11) $$D(\eta),\quad \text{if}\quad \sum_{i=1}^{s} b_i c_i^{l-1} a_{ij} = \frac{1}{l} b_j (1 - c_j^l) \quad (1 \le j \le s,\ 1 \le l \le \eta),$$

(12) $$E(\xi,\eta),\quad \text{if}\quad \sum_{i,\,j=1}^{s} b_i c_i^{l-1} a_{ij} c_j^{k-1} = \frac{1}{k(l+k)} \quad (1 \le l \le \eta,\ 1 \le k \le \xi).$$

The condition $A(\xi)$ means that the method has an approximation degree not smaller than ξ. The application of Runge-Kutta method to the differential equation $\dot{y}(t) = f(t),\ y(t_n) = 0$ gives the quadrature formula

(13) $$y_n = \tau \sum_{i=1}^{s} b_i f(t_n + c_i \tau),$$

the right side of which approximates the integral

$$y(t_{n+1}) = \tau \int_0^1 f(t_n + x\tau)dx.$$

The method of digital integration (13) is defined entirely by the abscissas c_i and the weights b_i of Runge-Kutta method. Hence, condition $B(\xi)$ means that the quadrature formula is exact in case f is a polynomial of a degree not higher than $\xi - 1$. This is equivalent to the affirmation that the quadrature formula (13) is of degree ξ.

The following results reflecting the relations among the conditions are in power:

Theorem 1 (Butcher). If for a given s-stage Runge-Kutta method all the abscissas $c_1, \ldots, c_s$ are different, and all the weights $b_1, \ldots, b_s$ are different from zero, then the following logical relations are valid:

(14) $$A(\xi) \Rightarrow B(\xi),$$
(15) $$A(\eta + \xi) \Rightarrow E(\eta, \xi),$$
(16) $$B(\eta + \xi) \wedge C(\xi) \Rightarrow E(\eta, \xi),$$
(17) $$B(\eta + \xi) \wedge D(\eta) \Rightarrow E(\eta, \xi),$$
(18) $$B(s + \xi) \wedge E(s, \xi) \Rightarrow C(\xi),$$
(19) $$B(\eta + s) \wedge E(\eta, s) \Rightarrow D(\eta),$$
(20) $$B(p) \wedge C(\xi) \wedge D(\eta) \Rightarrow A(p) \quad (p \le \min(\xi + \eta + 1, 2\xi + 2)).$$

Theorem 2 (Butcher). Let for a given s-stage method all the abscissas be different, and the weights – different from zero. Then the following relations are valid:

(21) $$A(2s) \Rightarrow B(2s) \wedge C(s) \wedge D(s),$$
(22) $$B(2s) \wedge C(s) \Rightarrow D(s),$$
(23) $$B(2s) \wedge D(s) \Rightarrow C(s),$$
(24) $$B(2s) \wedge C(s) \wedge D(s) \Rightarrow A(2s).$$

The next result (refer to [1]) enables the calculation of the coefficients of A in Butcher table:

Theorem 3. Let s different abscissas $c_1, \ldots, c_s$ be given. Then conditions $B(s)$ and $C(s)$ define unambiguously a Runge-Kutta scheme. The same is also true for conditions $B(s)$ and $D(s)$. The method defined by conditions $B(s)$ and $C(s)$ has an approximation degree not smaller than s.

The approximation degree can be greater than s at appropriate choice of the abscissas. More concretely, this is so when the abscissas are selected to be nodes of the qadrature formula of high order. We shall discuss the case of an s-stage method of Gauss-Legendre, the abscissas of which are roots of the modified polynomial of Legendre $P_s^*(x)$ of s order, determined in the interval [0, 1] (refer to [3]). If matrix V is defined as

$$V = \begin{bmatrix} 1 & c_1 & . & . & . & c_1^{s-1} \\ 1 & c_2 & . & . & . & c_2^{s-1} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ 1 & c_s & . & . & . & c_s^{s-1} \end{bmatrix},$$

and the diagonal matrix S as

$$S = \begin{bmatrix} 1 & & & & \\ & \frac{1}{2} & & & \\ & & . & & \\ & & & . & \\ & & & & \frac{1}{s} \end{bmatrix},$$

then the weights of this method are defined by the condition $B(s)$ as follows:

$$V^{\mathrm{T}} b = Se,$$

where b is the weights vector, and $e = [1, 1, \ldots, 1]^{\mathrm{T}}$.

The remaining parameters are defined by condition $C(s)$. The method thus constructed satisfies condition $B(2s)$ and it follows from Theorem 2 that the s-stage method of Gauss-Legendre has an approximation degree of $2s$.

II. The *mpmath* library

It is obvious from the previous chapter that theoretically an implicit Runge-Kutta scheme may be designed of an arbitrary approximation degree. With a sufficiently small step and a method with an approximation degree of 20 (for example the 10-stage method of Gauss-Legendre) we would achieve accuracy of the order of $1.0e^{-40}$. A necessary condition is that the calculations be accomplished with sufficient accuracy; both in calculating the coefficients for the method step, and also for probable alteration of the step length under condition of convergence absence in the iterative procedure, determining these coefficients. The *mpmath* library provides this possibility (as well as other built-in tools). It contains a built-in method for solving systems of differential equations, but based on another approach, which is not rectilinear in parameters setting for high accuracy. There is not a realized adaptive scheme for step length alteration in the illustrating program, given below. However, the user realizes what happens at convergence lack and he/she may realize the respective logic of step length alteration. Only the library elements, used in the program are explained here. For more detail information [4] is to be seen.

The library presents several digital types, from which the following two are used: mpf – for real numbers with a floating point and matrix – for matrices. Copies of mpf class can be created by strings (representing numbers), integers, floating, or of another mpf copy. mpmath library uses the global operating accuracy of the calculations. The execution of arithmetic operations or the invoking of mpf() rounds the result up to the given operating accuracy. This accuracy is set through the number of decimal characters or the number of bits for the global mp object. For example:

```
mp.dps = 100    # number of decimal characters, by default it is 15,
mp.prec = 333   # number of bits, by default it is 53.
```

If higher accuracy is needed for certain calculations, it may be changed. For example:

```
mp.dps + = 10,
# Calculations of increased accuracy are done
mp.dps – = 10.
```

The matrices in *mpmath* can be created setting the number of rows and columns (one parameter for the design of a quadrature matrix) or by a list. By default the elements type is mpf, but it can be separately given. The matrices are realized as Python lists. An example:

$$A = \text{matrix}(2),$$
$$b = [1, 2, 3],$$
$$B = \text{matrix}(b).$$

A is a quadratic matrix 2×2 with null elements; *B* is a single-row matrix. The access to a specific element is with the help of the syntax *A*[*i*, *j*], where *i* denotes the row and *j* – the column.

lu_solve(*A*, *b*) gives back the solution of the linear system ($Ax = b$, in a matrix record). The function polyroots() is also used in the program, that computes all the roots of a polynomial. The function polyroots accepts the list with polynomial coefficients as an argument. The value eps presents the current accuracy, and nprint prints out with a given number of decimal characters (the second argument).

III. Example

The program is a realization of the implicit Runge-Kutta scheme of Gauss-Legendre type. Its kernel consists of a solver module. A test module for the solver is also provided. The solver is realized as a class, considering the scheme order and the computing accuracy as parameters and it computes the nodes and the weights of Gauss-Legendre quadrature formula with the specified parameters when creating a copy of the class. It contains a method for additional initialization init, which accepts as parameters the system of differential equations to be solved (in the form of a function), the step and the initial conditions of Cauchy problem. The method *step* accomplishes the calculations for one step, using *iterate* method. Method *iterate* uses a simple iteration to find the coefficients for a given step, using as an initial point the coefficients from the previous step. With such selection of an initial point, the existence of a step with a positive value, for which the simple iteration converges, is guaranteed (refer to [1]), but in *iterate* and *step* no built-in adaptive scheme is available for alteration of the step at convergence absence.

III.1. Solver code

```
    # module irk_solver
    """
    Generating the coefficients of an implicit Runge-
Kutta scheme of a given order.
    For the moment <= 68.
    The coefficients are set with an arbitrary accuracy,
user defined.
    One step of the method is executed.
    """
    from __future__ import division
    from mpmath import mp, mpf, matrix, lu_solve,
factorial, polyroots, eps, sqrt

    class irk(object):
        #__slots__ =
['__init__','init','iterate','step','rang','acc','r','b'
,'a','size','f','t','h','yb','ks','tn','y','yn']
        def __init__(self,rang=10,acc=100):
            # Generates the coefficients of Legendre
polynomial of  n-th order.
            # acc is the number of decimal characters of
the coefficients.
            # self.cf is the list with coefficients.
            self.rang = rang
            self.acc = mp.dps = acc
            cn = mpf(0.0)
            k = mpf(0)
            n = mpf(rang)
            m = mpf(n/2)
            cf = []
            for k in range(n+1):
                cn = (-
1)**(n+k)*factorial(n+k)/(factorial(n-
k)*factorial(k)*factorial(k))
                cf.append(cn)
            cf.reverse()
            # Generates the coefficients of of the
implicit Runge-Kutta scheme of Gauss-Legendre type.
            # acc is the number of the decimal
characters of the coefficients.
            # Gives back the cortege (r,b,a), the terms
of which correspond to Butcher scheme
            #
            # r1 | a11 . . . a1n
            #  . |  .         .
            #  . |  .         .
            #  . |  .         .
            # rn | an1 . . . ann
            # ---+--------------
            #    |  b1 . . .  bn
            self.r  = polyroots(cf)
            A1 = matrix(rang)
            for  j in range(n):
                for k in range(n):
                    A1[k,j] = self.r[j]**k
            bn = []
            for j in range(n):
                bn.append(mpf(1.0)/mpf(j+1))
            B = matrix(bn)
            self.b = lu_solve(A1,B)
            self.a = matrix(rang)
            for i in range(1,n+1):
                A1 = matrix(rang)
                cil = []
                for l in range(1,n+1):
                    cil.append(mpf(self.r[i-
1])**l/mpf(l))
                    for j in range(n):
                        A1[l-1,j] = self.r[j]**(l-1)
                Cil = matrix(cil)
                an = lu_solve(A1,Cil)
                for k in range(n):
                    self.a[i-1,k] = an[k]

        def init(self,f,t,h,initvalues):
            self.size = len(initvalues)
            self.f = f
            self.t = t
            self.h = h
            self.yb = matrix(initvalues)
            self.ks = matrix(self.size,self.rang)
            for k in range(self.size):
                for i in range(self.rang):
                    self.ks[k,i] = self.r[i]
            self.tn = matrix(1,self.rang)
            for i in range(self.rang):
                self.tn[i] = t + h*self.r[i]
            self.y = matrix(self.size,self.rang)
            for k in range(self.size):
                for i in range(self.rang):
                    self.y[k,i] = self.yb[k]
                    temp = mpf(0.0)
                    for j in range(self.rang):
                        temp += self.a[i,j]*self.ks[k,j]
                    self.y[k,i] += temp
            self.yn = matrix(self.yb)

        def iterate(self,tn,y,yn,ks):
            # Generates the coefficients of the implicit
    Runge-Kutta scheme for the given step
            # with the method of the simple iteration
    with an initial value, coinciding with the coefficients,
            # calculated at the previous step. At
    sufficiently small step this must
            # work. There exists such a value of the
    step, Under which convergence is guaranteed.
            # No automatic re-setup of the step is
    foreseen in this procedure.
            mp.dps = self.acc
            y0 = matrix(yn)
            norme = mpf(1.0)
            #eps0 = pow(eps,mpf(3.0)/mpf(4.0))
            eps0 = sqrt(eps)
            ks1 = matrix(self.size,self.rang)
            yt = matrix(1,self.size)

            count = 0
            while True:
                count += 1
                for i in range(self.rang):
                    for k in range(self.size):
                        yt[k] = y[k,i]
                    for k in range(self.size):
                        ks1[k,i] = self.f(tn,yt)[k]
                norme = mpf(0.0)
```

```
                for k in range(self.size):
                    for i in range(self.rang):
                        norme += (ks1[k,i]-
ks[k,i])*(ks1[k,i]-ks[k,i])
                norme = sqrt(norme)
                for k in range(self.size):
                    for i in range(self.rang):
                        ks[k,i] = ks1[k,i]
                for k in range(self.size):
                    for i in range(self.rang):
                        y[k,i] = y0[k]
                        for j in range(self.rang):
                            y[k,i] +=
self.h*self.a[i,j]*ks[k,j]
                if norme <= eps0:
                    break
                if count >= 100:
                    print unicode('No convergence','UTF-
8')
                    exit(0)

            return ks1

        def step(self):
            mp.dps = self.acc
            self.ks =
self.iterate(self.tn,self.y,self.yn,self.ks)
            for k in range(self.size):
                for i in range(self.rang):
                    self.yn[k] +=
self.h*self.b[i]*self.ks[k,i]
            for k in range(self.size):
                for i in range(self.rang):
                    self.y[k,i] = self.yn[k]
                    for j in range(self.rang):
                        self.y[k,i] +=
self.a[i,j]*self.ks[k,j]
            self.t += self.h
            for i in range(self.rang):
                self.tn[i] = self.t + self.h*self.r[i]
            return self.yn
```

III.2. Code of the test module

The test module below given is for illustration. Lorenz system is used, at parameter $r = 28$ ($r > 24.74$), i.e., description of a chaotic mode of Lorenz attractor. The initial point is selected close to the attractor (in order to avoid the time for entering it). Description and studies of this system can be found at many places. For more details about nonlinear dynamics [5, 6] may be referred to. 100 steps are computed with a value of 0.01 according to the 10- and 12-stage scheme, in order to obtain accuracy evaluation. The calculations are with an accuracy up to 100 decimal characters. After 100 steps the differences, obtained from both schemes are smaller than $1.0e^{-33}$. At module beginning the object *odefun* of *mpmath* library is used to realize computations in the same interval and with the same operating accuracy and without setting an additional parameter for the part of Tayler series used (for *odefun*). The results are printed out with 20 characters and set after the code. A convenient possibility is provided to enter some parameters from the command line. It is interesting to establish after what time the trajectory “splits” – i.e., the error accumulated leads to trajectory split, described by schemes of different order. The experiment (the results are given in the next chapter) shows that for schemes of a relatively high order, 10 for example, thousand iterations are needed.

```
    # Test for the module irk_solver.py
    from __future__ import division
    #from mpmath import mp, mpf, linspace, zeros, nprint
    from mpmath import *
    from irk_solver import irk
    import sys
    import getopt
    def usage():
        print '\n'
        print  u'Използване:  '  +  sys.argv[0]  +  u'
[options]'
        print '\n'
        print u'Options:'
        print u'  -h, --help                      Prints
out this message and quits the program'
        print  u'    -p  ПАРАМЕТЪР,  --param=ПАРАМЕТЪР
Parameter r in Lorenz system'
        print u'  -r РАНГ, --rang=РАНГ              Rank
of the implicit scheme'
        print   u'      -a   ТОЧНОСТ,   --acc=ТОЧНОСТ
Accuracy of calculation (in decimal characters)'
        print u'  -s СТЪПКА, --step=СТЪПКА          Step
of integration'
        print   u'      -m   МНОЖИТЕЛ,   --mlt=МНОЖИТЕЛ
Multiplier for the number of the points'
        print '\n'
    mp.dps = 100
    print '=== odefun ==='
    ff = odefun(lambda t, y: [mpf(10.0)*(y[1]-
y[0]),mpf(28.0)*y[0]-y[1]-y[0]*y[2],-
mpf(mpf(8.0)/mpf(3.0))*y[2]+y[0]*y[1]], 0,
[mpf(10.6451),mpf(4.06125),mpf(36.057)])
    nprint(ff(1),20)
    acc = 100
    try:
        options, args = getopt.getopt(sys.argv[1:],
'p:r:s:a:m:h',
['rang=','step=','acc=','mlt=','param=','help'])
    except getopt.GetoptError, err:
        print '\n' + str(err)
        usage()
        sys.exit(2)
    rang = 10
    t0 = mpf(0.0)
    h  = mpf(0.01)
    mlt = 1
    disp = 'xz'
    r = mpf(28.0) # 24.74
    b0 = mpf(mpf(8.0)/mpf(3.0))
    s = mpf(10.0)
    for option, value in options:
        if option in ('-r', '--rang'):
            rang = int(value)
        if option in ('-p', '--param'):
            r = mpf(value)
        if option in ('-s', '--step'):
            h = mpf(value)
        if option in ('-a', '--acc'):
            acc = mpf(value)
        if option in ('-m', '--mlt'):
            mlt = int(value)
        if option in ('-h', '--help'):
            usage()
            sys.exit()
    def F(t,y):
        global s
        global b0
        global r
        y0 = mpf(y[0])
        y1 = mpf(y[1])
        y2 = mpf(y[2])
        res0 = mpf(s*(y1-y0))
        res1 = mpf(r*y0-y1-y0*y2)
        res2 = mpf(-b0*y2+y0*y1)
        return (res0,res1,res2)
    yb = [mpf(10.6451),mpf(4.06125),mpf(36.057)]
    solver = irk(rang,acc)
    solver.init(F,t0,h,yb)
    solver1 = irk(rang+2,acc)
    solver1.init(F,t0,h,yb)
    num_points = mlt*int(1.0/h + 0.1) + 1
    tpa = linspace(0,h*(num_points-1),num_points)
    for k in range(num_points-1):
        solver.yn = solver.step()
        solver1.yn = solver1.step()
    print '===== IRK ====='
    print 'x = ',
    nprint(solver.yn[0],20)
    print 'y = ',
    nprint(solver.yn[1],20)
    print 'z = ',
    nprint(solver.yn[2],20)
    print '===== ERR ====='
    nprint(solver1.yn[0]-solver.yn[0],20)
    nprint(solver1.yn[1]-solver.yn[1],20)
    nprint(solver1.yn[2]-solver.yn[2],20)
```

III.3. Test results

```
    === odefun ===
    [-0.10454605687628689048, -1.2345223788685477043,
20.029753956718710011]
    ===== IRK =====
    x =  -0.10454605687628712 57
    y =  -1.2345223788685476 96
    z =  20.029753956718708902
    ===== ERR =====
    8.8113415875398021481e-35
    3.6191682314597705644e-34
    -8.9322105203614438561e-34
```

IV. More detailed results and illustrations

Since increased accuracy is on account of time, it would be nice to preserve the calculations results. With the help of (lorenz_save.py) module the computation results for 10 000 iterations were stored as ordinary text files from one and the same initial point (0.01) for order of the scheme 10 and 12 respectively.

```
    # Results saving in a file
    from __future__ import division
    from mpmath import mp, mpf, nprint, matrix
    from irk_solver import irk
    from pylab import *
    import sys
    import getopt
    def usage():
        print '\n'
        print u'Използване: ' + sys.argv[0] + u'
[options]'
        print '\n'
        print u'Options:'
        print u'  -h, --help                      Prints
out this message and quits the program'
        print u'  -p ПАРАМЕТЪР, --param=ПАРАМЕТЪР
Parameter r in Lorenz system '
        print u'  -r РАНГ, --rang=РАНГ            Rank
of the implicit scheme'
        print u'  -a ТОЧНОСТ, --acc=ТОЧНОСТ
Computing accuracy (in decimal characters)'
        print u'  -s СТЪПКА, --step=СТЪПКА        Step
of integration'
        print u'  -m МНОЖИТЕЛ, --mlt=МНОЖИТЕЛ
Multiplier for the number of points'
        print u'  -f ЗАПИСВАНЕ, --file=ЗАПИСВАНЕ  Name
of the file, where the results are saved'
        print '\n'
    mp.dps = 100
    acc = 100
    rang = 10
    t0 = mpf(0.0)
    h  =
mpf('0.0100000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000')
    mlt = 1
    fname = ""
    try:
        options, args = getopt.getopt(sys.argv[1:],
'p:r:s:a:m:f:h',
['rang=','step=','acc=','mlt=','param=','file=','help'])
    except getopt.GetoptError, err:
        print '\n' + str(err)
        usage()
        sys.exit(2)
    r = mpf(28.0) # 24.74
    b0 = mpf(mpf(8.0)/mpf(3.0))
    s = mpf(10.0)
    for option, value in options:
        print (option,value)
        if option in ('-r', '--rang'):
            rang = int(value)
        if option in ('-p', '--param'):
            r = mpf(value)
        if option in ('-s', '--step'):
            h = mpf(value)
        if option in ('-a', '--acc'):
            acc = mpf(value)
        if option in ('-m', '--mlt'):
            mlt = int(value)
        if option in ('-f', '--file'):
```

```
        fname = value
    if option in ('-h', '--help'):
        usage()
        sys.exit()
def F(t,y):
    global s
    global b0
    global r
    y0 = mpf(y[0])
    y1 = mpf(y[1])
    y2 = mpf(y[2])
    res0 = mpf(s*(y1-y0))
    res1 = mpf(r*y0-y1-y0*y2)
    res2 = mpf(-b0*y2+y0*y1)
    return (res0,res1,res2)
ybf = [10.6451,4.06125,36.057]
yb = [mpf(ybf[0]),mpf(ybf[1]),mpf(ybf[2])]
zero = mpf(0.0)
solver = irk(rang,acc)
solver.init(F,t0,h,yb)
num_points = mlt*int(1.0/h + 0.1) + 1
tmp = []
for k in range(num_points):
    tmp.append(zero)
tpa = tmp[:]
for k in xrange(num_points-1):
    tpa[k+1] = tpa[k] + h
xpa = tmp[:]; xpa[0] = ybf[0]
ypa = tmp[:]; ypa[0] = ybf[1]
zpa = tmp[:]; zpa[0] = ybf[2]
for k in xrange(num_points-1):
    solver.yn = solver.step()
    #print 'step: ',
    #print k+1
    xpa[k+1] = solver.yn[0]
    ypa[k+1] = solver.yn[1]
    zpa[k+1] = solver.yn[2]
fsave = open(fname,'w')
for k in xrange(num_points):
    tstr = str(tpa[k])
    xstr = str(xpa[k])
    ystr = str(ypa[k])
    zstr = str(zpa[k])
    line = "%s %s %s %s \n" % (tstr,xstr,ystr,zstr)
    fsave.write(line)
fsave.close()
print u'Натисни ENTER за излизане',
raw_input()
```

Deriving information with the purpose to visualize and save in graphical format may be realized very easy, for example in the following way (the first parameter for the program is the name of the file with the results saved, and the second one is the plane, where projected on):

```
    #Visualization of the data stored and saving of a
graphical file
    from mpmath import mp, mpf
    from pylab import *
    import sys
    fname = sys.argv[1]
    disp = sys.argv[2]
    taf = []
    xaf = []
    yaf = []
    zaf = []
    fread = open(fname,'r')
    for line in fread.readlines():
        point = line.split(' ')
        taf.append(float(mpf(point[0])))
        xaf.append(float(mpf(point[1])))
        yaf.append(float(mpf(point[2])))
        zaf.append(float(mpf(point[3])))
    fig = figure()
    xlabel(disp[0].upper()+' Axis')
    ylabel(disp[1].upper()+' Axis')
    if disp=='xz':
        plot(xaf[0:],zaf[0:],'b-')
    elif disp=='xy':
        plot(xaf[0:],yaf[0:],'b-')
    elif disp=='yz':
        plot(yaf[0:],zaf[0:],'b-')
    elif disp=='tx':
        plot(taf[0:],xaf[0:],'b-')
    elif disp=='ty':
        plot(taf[0:],yaf[0:],'b-')
    elif disp=='tz':
        plot(taf[0:],zaf[0:],'b-')
    fig.show()
    fig.savefig(disp+'_10000.png', format='png')
    raw_input()
```

Fig. 1 shows the result for the case of a second parameter *XZ*.

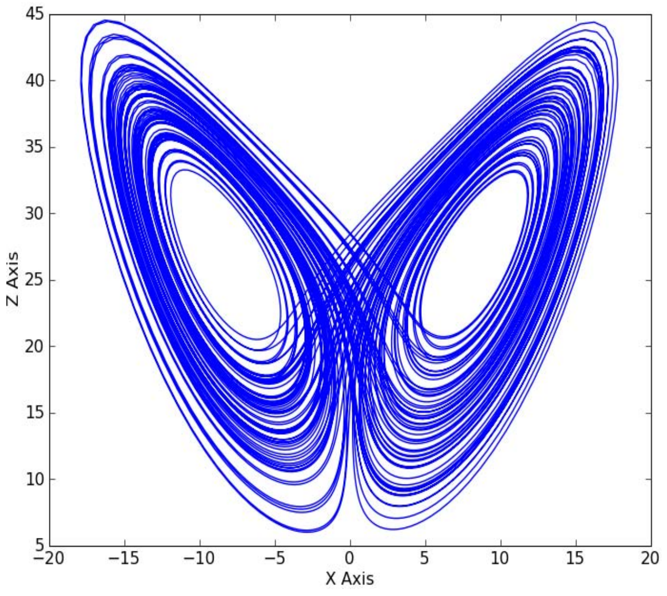

Fig. 1

The chaotic state may be illustrated visualizing the value of any of the coordinates in time duration. For example (Fig. 2) with the second parameter *TZ*.

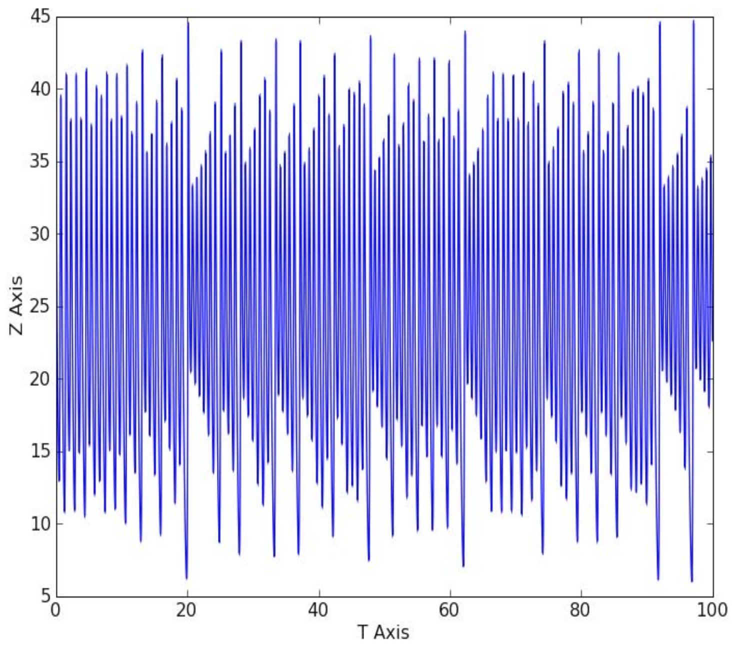

Fig. 2

The illustrations obtained are classical, but at first glance the contribution of high accuracy is not clear, since the qualitative image would be the same using other methods. As above mentioned, it is interesting to trace up to what moment the increased accuracy of the computations keeps down the error accumulation. Here is an example code, using two apriori saved files that will demonstrate this.

```
from mpmath import mp, mpf
from pylab import *
import sys
fname1 = sys.argv[1]
fname2 = sys.argv[2]
disp = sys.argv[3]
taf = []
xaf1 = []
yaf1 = []
zaf1 = []
xaf2 = []
yaf2 = []
zaf2 = []
file1 = open(fname1,'r')
for line in file1.readlines():
    point = line.split(' ')
    taf.append(float(mpf(point[0])))
    xaf1.append(float(mpf(point[1])))
    yaf1.append(float(mpf(point[2])))
    zaf1.append(float(mpf(point[3])))
file1.close()
file2 = open(fname2,'r')
for line in file2.readlines():
    point = line.split(' ')
    xaf2.append(float(mpf(point[1])))
    yaf2.append(float(mpf(point[2])))
    zaf2.append(float(mpf(point[3])))
file2.close()
fig = figure()
xlabel('T Axis')
ylabel(disp[0].upper()+' Axis')
scnd = []
dim = len(taf)
if disp=='x':
    for k in xrange(dim):
        scnd.append(xaf2[k] - xaf1[k])
elif disp=='y':
    for k in xrange(dim):
        scnd.append(yaf2[k] - yaf1[k])
elif disp=='z':
    for k in xrange(dim):
        scnd.append(zaf2[k] - zaf1[k])
plot(taf[0:],scnd[0:],'b-')
fig.show()
name = disp + '_diff.png'
fig.savefig(name,format='png')
raw_input()
```

The first two parameters are for the names of the two files with apriori written results, and the third one is for difference in time along a given axis. The results are similar for the three axes. The difference along *x* axis (x_diff.png) is as shown in Fig. 3.

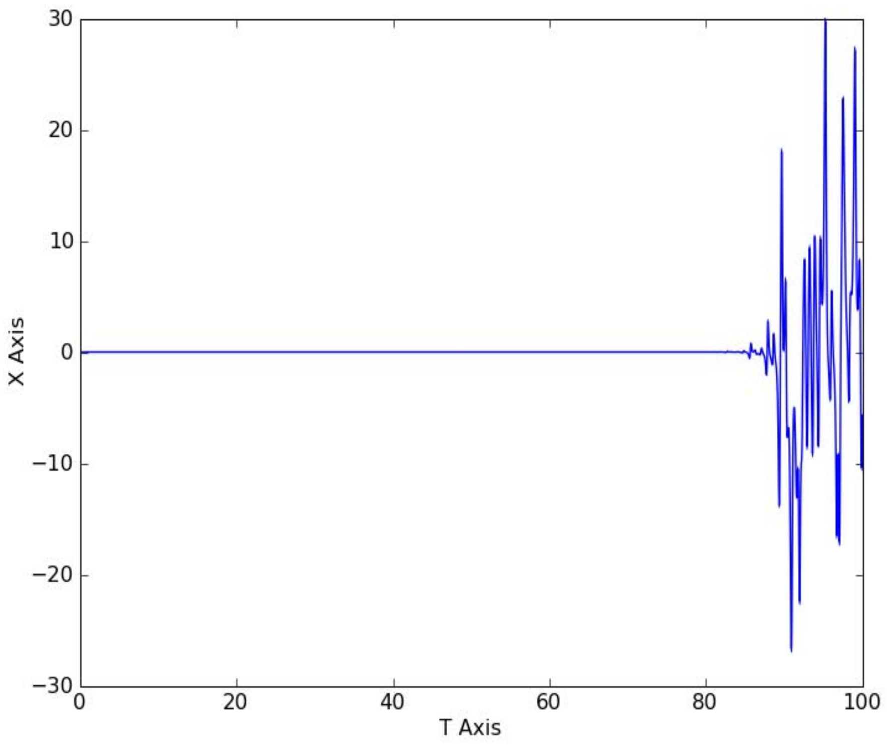

Fig. 3

Having in mind that in the case discussed one unit along the axis of time corresponds to 100 iterations, for the parameters given the results could be considered as exact for more than 8000 iterations.

V. Conclusion

The idea of designing irk_solver.py came after visiting the site of Pavel Holoborodko. And particularly, the part, concerning digital integration (at current address **http://www.holoborodko.com/pavel/?page_id=679**,). The challenge was how this will be done nowadays, compared to 15 years ago. Every one, who has written a library for computations with arbitrary accuracy, would evaluate the labour consumption of a similar task. But today the situation is different. It is not absolutely necessary to create the tools from the lowest level. This refers to visualization as well. The researcher who wants to focus on solving and/or study of a given type of mathematic models, can design the corresponding tool much faster, using already existing tools as built-in blocks. This predetermined the choice of Python, which is an excellent environment for the purpose considered. The straight-forward code above given may be still improved in many ways, including the application of some idioms, typical for Python. However, the utility and efficiency of Python are of no doubt. Efficiency means mainly the expressing power and time saving of the design. The efficiency, concerning execution, is another aspect (yet, it is an interpreted language, though written, as well as the packages, in C), but Python has the possibility to integrate with C in several approaches, which is another topic.

R e f e r e n c e s

1. D e k k e r, K., J. G. V e r w e r. Stability of Runge–Kutta Methods for Stiff Nonlinear Differenrial Equations. North-Holland, 1984. Превод на руски: К. Деккер, Я. Вервер. Устойчивость методов Рунге–Куты для жестких нелинейных дифференциальных уравнений. М., Мир, 1988.
2. B u t c h e r, J. C. Numerical Methods for Ordinary Differential Equations. Second Ed. John Wiley & Sons, Ltd., 2008.
3. K y t h e, P., K. Mi c h e l, R. S c h ä f e r k o t t e r. Handbook of Computational Methods for Integration. Chapman & Hall/CRC Press, 2005.
4. **http://mpmath.googlecode.com/svn/tags/0.13/doc/build/index.html**
5. T a b o r, M. Chaos and Integrability in Nonlinear Dynamics. An Introduction. John Wiley & Sons, 1989.
6. O t t, E. Chaos in Dynamical Systems. Cambridge University Press, 1993.
7. L a n g t a n g e n, H. P. Python Scripting for Computational Science. Third Edition. Springer, 2008.

Использование неявных методов Рунге–Куты для прецизного вычисления нелинейных дифференциальных уравнений

Величко Джамбов

Институт информационных и коммутационных технологий, 1113 София
E-mail: vili_jambov@abv.bg

(Р е з ю м е)

В работе обсуждается метод вычисления нелинейных дифференциальных уравнений с повышеной точностью. Реализация основана на неявной схеме Рунге–Кута. Коэффициенты, необходимые для въчислений, генерируются во времени выполнения задачи при помощи библиотеки *mpmath* в среде Python.