

On Dormand-Prince Method

Toshinori Kimura*

September 24, 2009

Abstract

Although Runge-Kutta-Fehlberg method works pretty well even for problems that need changing calculation intervals automatically, it is a little old method. It was used 1960's. Recently, people use a method called Dormand-Prince method. It is also a method categorised in Adaptive Step Method. It is more accurate than the Runge-Kutta-Fehlberg method. It is used in present Matlab.

In this paper, we look what the Dormand-Prince method is. We apply the method to the same problem we treated in Runge-Kutta-Fehlberg method and we see how accurate the Dormand-Prince method is compared to the Runge-Kutta-Fehlberg method.

1 Dormand-Prince Method

The one step calculation in the Dormand-Prince method is done as the following.

$$k_1 = hf(t_k, y_k), \quad (1)$$

$$k_2 = hf(t_k + \frac{1}{5}h, y_k + \frac{1}{5}k_1), \quad (2)$$

$$k_3 = hf(t_k + \frac{3}{10}h, y_k + \frac{3}{40}k_1 + \frac{9}{40}k_2), \quad (3)$$

$$k_4 = hf(t_k + \frac{4}{5}h, y_k + \frac{44}{45}k_1 - \frac{56}{15}k_2 + \frac{32}{9}k_3), \quad (4)$$

$$k_5 = hf(t_k + \frac{8}{9}h, y_k + \frac{19372}{6561}k_1 - \frac{25360}{2187}k_2 + \frac{64448}{6561}k_3 - \frac{212}{729}k_4), \quad (5)$$

$$k_6 = hf(t_k + h, y_k + \frac{9017}{3168}k_1 - \frac{355}{33}k_2 - \frac{46732}{5247}k_3 + \frac{49}{176}k_4 - \frac{5103}{18656}k_5), \quad (6)$$

$$k_7 = hf(t_k + h, y_k + \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6). \quad (7)$$

Then the next step value y_{k+1} is calculated as

$$y_{k+1} = y_k + \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6. \quad (8)$$

This is a calculation by Runge-Kutta method of order 4. We have to be aware that we do not use k_2 , though it is used to calculate k_3 and so on.

Next, we will calculate the next step value z_{k+1} by Runge-Kutta method of order 5 as

$$z_{k+1} = y_k + \frac{5179}{57600}k_1 + \frac{7571}{16695}k_3 + \frac{393}{640}k_4 - \frac{92097}{339200}k_5 + \frac{187}{2100}k_6 + \frac{1}{40}k_7. \quad (9)$$

We calculate the difference of the two next values $|z_{k+1} - y_{k+1}|$.

$$|z_{k+1} - y_{k+1}| = |\frac{71}{57600}k_1 - \frac{71}{16695}k_3 + \frac{71}{1920}k_4 - \frac{17253}{339200}k_5 + \frac{22}{525}k_6 - \frac{1}{40}k_7|. \quad (10)$$

*Senior Volunteer of Japan International Cooperative Association - Japan Malaysia Technical Institute

This is considered as the error in y_{k+1} . We calculate the optimal time interval h_{opt} as,

$$s = \left(\frac{\epsilon h}{2|z_{k+1} - y_{k+1}|} \right)^{\frac{1}{5}}, \quad (11)$$

$$h_{opt} = sh, \quad (12)$$

where h in the right side is the old time interval. In practical programming, this new h_{opt} will be used in the next step of the calculation, though the author thinks it should be also used in the present calculation when it is very small, half or smaller for example.

2 Programming Guidance

The general programming procedure to solve the equation

$$y' = f(t, y) \quad (13)$$

$$y(t_0) = y_0 \quad (14)$$

will be as the following. We express the program as in Language C style, though we don't obey to the rigorous Language C syntax.

```
...
eps=0.000001; //error allowance in one step calculation.
t0=0; //initiallization of variables.
y0=0;
h0=0.1;
while(t0<=tf) //tf is the ending time of the calculation.
{
    k1=h*f(t0,y0);
    ...
    k7=h*f(t0+h,...);
    y1=y0 + 35/384*k1 + ...

    z1=y0 + 5179/57600*k1 + ... //to estimate the error.
    err=abs(z1-y1); //error estimation. .... (*)
    s=pow(eps*h0/(2*err),1/5);
    h1=s*h0; //optimal time interval. used in the next step.
    if(h1<hmin)h1=hmin; //confine time interval between hmin and hmax.
    else if(h1>hmax)h1=hmax;

    t0=t0+h0; //renew variables
    y0=y1;
    h0=h1;
}
...
```

We can see the meanings of important steps in the program as comments. As we wrote in the former section, we can add the repetition of the calculation of the step when $h1$ is too small to accept the result as an accurate one. We use $z1$ only to calculate the err in (*), so we can omit the calculation of $z1$ by replacing (*) as

```
err=abs(71/57600*k1 - ...);
```

References

- [1] <http://reference.wolfram.com/mathematica/tutorial/NDSolveExplicitRungeKutta.html>
- [2] <http://documents.wolfram.com/v5/Built-inFunctions/AdvancedDocumentation/DifferentialEquations/NDSolve/ExplicitRungeKutta.html>
- [3] http://en.wikipedia.org/wiki/Dormand-Prince_method From Wikipedia to confirm the contents of the above materials, especially to confirm the coefficients.

3 Results of the Dormand-Prince Method

Let us apply the Dormand-Prince method to the problems we treated by Runge-Kutta-Fehlberg method in the former paper. (See "Toshinori Kimura, Sep.16,2009, On Runge-Kutta-Fehlberg Method") The problems are

- Stationary Satellite,
- Satellite with 1/10 of the Stationary Velocity at Stationary Distance,
- Satellite with 1/100 of the Stationary Velocity at Stationary Distance.

We compare the result with those of the Runge-Kutta-Fehlberg method. The results are in the following subsections.

Each subsection contains results by Runge-Kutta-Fehlberg method, former part, from the former paper, and Dormand-Prince method, latter part, with their input data correspondently. The program used for Dormand-Prince method is in the appendix. Some comments are added in the input and output results by the auther. Input and output formats are same for all cases. Therefore comments are mainly added in the first result.

3.1 Stationary Satellite

The following results are for a Stationary Satellite.

```
----- Runge-Kutta-Fehlberg -----
(Input)
10000.0
42242276.53890282602184866499414568877931 <-- Initial radius
3071.94503809087027757155147883394003751 <-- Initial speed
90.0 <----- Angle between initial radius and velocity
0.0 <----- Initial time
86400.0 <----- Ending time (One day in seconds)
0.000000001
2.0 <----- Initial calculation time interval
10.0
0.0000000000001 <-- Error tolerance
(Output)
(Counter)      (Time)      (X-address)      (Y-address)      (X-velocity)      (Y-velocity)
18715  86382.0177874306(  42242240.419898990491  -55240.35293223393699  4.0171918276299428069  3071.9424114427891508)err= 0.000000000000230789830574485575
18716  86386.6335840421(  42242256.582627758778  -41060.88871299291112  2.9860320909862636381  3071.94358682959379 )err= 0.000000000000230789830574485559
18717  86391.2493806536(  42242267.985732087578  -26881.419867237460128  1.9548720178929610981  3071.9444160867416483)err= 0.000000000000230789830574485569
18718  86395.865177265 (  42242274.629210692052  -12701.947992631911724  0.92371172453546376914  3071.9448992141392894)err= 0.000000000000230789830574485549
18719  86400      (  42242276.53890282602  2.3051574558912487181e-09  -1.6763867977934957343e-13  3071.9450380908702775)err= 0.00000000000011723210811630357 <-- (1)

----- Dormand-Prince -----
(Input)
10000.0
42242276.53890282602184866499414568877931
3071.94503809087027757155147883394003751
90.0
0.0
86400.0
0.000000001
2.0
10.0
0.0000000000001
(Output)
16677  86381.6709821408(  42242239.013281141992  -56305.718792764813197  4.0946674193158032157  3071.9423091506525937)err= 0.000000000000258997395505153714
16678  86386.8509300509(  42242257.22635333788  -40393.214029957284599  2.9374774178563419377  3071.9436336426386364)err= 0.000000000000258997395505153707
16679  86392.030877961 (  42242269.445240445475  -24480.703535344289896  1.7802869995682245458  3071.9445222251845883)err= 0.000000000000258997395505153681
16680  86397.2108258711(  42242275.66994074319  -8568.1895669143752576  0.62309632865701188566  3071.9449748981643591)err= 0.000000000000258997395505153712
16681  86400      (  42242276.538902826023  -5.2500577096363695983e-12  3.817950510355943447e-16  3071.9450380908702775)err= 0.0000000000000011723210811630357 <-- (1)
```

We can see that the Dormand-Prince method is about 500 times more accurate than a Runge-Kutta-Fehlberg method for the same input data, notwithstanding the iteration number is 10 percent lesser than that of the Runge-Kutta-Fehlberg method.

One strange thing is that the errors in Dormand-Prince method are 10 percent larger than those of Runge-Kutta-Fehlberg method, notwithstanding the final result is far accurate than that of Runge-Kutta-Fehlberg method. Readers can research in it if they are interested in it.

3.1.1 Dormand-Prince Method is 5th Order.

We can confirm that the Dormand-Prince method is a 5th order method by calculating the stationary orbit in various accuracy. Because in these cases, the time intervals seem to be same. The following list is the result.

```
(Input)
10000.0
42242276.53890282602184866499414568877931
3071.94503809087027757155147883394003751
90.0
0.0
86400.0
0.000000001
2.0
10.0
0.000000000000001
(Output)
29658 86389.5734915917( 42242264.395865046021 -32029.657700276795445 2.3292624381524382448 3071.9441550242433814) err= 0.000000000000014564493868221272
29659 86392.4863903654( 42242270.23300664691 -23081.394686695289189 1.6785263884798242566 3071.9445795131593841) err= 0.000000000000014564493868221293
29660 86395.399289139 ( 42242274.174619889953 -14133.130637387296887 1.02779026348705472 3071.9448661553632645) err= 0.000000000000014564493868221284
29661 86398.3121879127( 42242276.220704598279 -5184.8659538863566344 0.37705409237447253892 3071.9450149508421604) err= 0.000000000000014564493868221275
29662 86400 ( 42242276.538902826022 -2.9573918968781521165e-13 2.1506762109128427153e-17 3071.9450380908702776) err= 0.00000000000000951237356179873 <-- (2)

(Input)
10000.0
42242276.53890282602184866499414568877931
3071.94503809087027757155147883394003751
90.0
0.0
86400.0
0.000000001
2.0
10.0
0.000000000000001
(Output)
52742 86394.1278142915( 42242272.687232317327 -18039.031201666057053 1.3118353680794246488 3071.9447579894866164) err= 0.00000000000000819021678388020
52743 86395.7658576483( 42242274.536366154826 -13007.052382371836953 0.94589954188233752854 3071.9448924622940097) err= 0.00000000000000819021678388042
52744 86397.4039010051( 42242275.786081231514 -7975.0733785072798424 0.57996370226291673182 3071.9449833441390098) err= 0.00000000000000819021678388030
52745 86399.0419443618( 42242276.436377529658 -2943.0942614762596316 0.21402785441379850204 3071.945030635020327 ) err= 0.00000000000000819021678388033
52746 86400 ( 42242276.538902826022 -1.6648598137446318734e-14 1.2107201840692255341e-18 3071.9450380908702776) err= 0.000000000000005605597928619 <-- (3)

(Input)
10000.0
42242276.53890282602184866499414568877931
3071.94503809087027757155147883394003751
90.0
0.0
86400.0
0.000000001
2.0
10.0
0.000000000000001
(Output)
93792 86396.7828923171( 42242275.382841786823 -9882.777893431113361 0.71869588951534255267 3071.9449540197390774) err= 0.00000000000000046056973600741
93793 86397.7040317891( 42242275.95008450873 -7053.0881203695078499 0.5129150422239418794 3071.9449952707938909) err= 0.00000000000000046056973600745
93794 86398.6251712611( 42242276.327774368482 -4223.3983156587468124 0.30713419263094966692 3071.9450227371755745) err= 0.00000000000000046056973600742
93795 86399.5463107331( 42242276.515911364383 -1393.7084919964011554 0.1013533416597593291 3071.9450364188840049) err= 0.00000000000000046056973600747
93796 86400 ( 42242276.538902826022 -9.3870014192987959649e-16 6.8264168547105138691e-20 3071.9450380908702776) err= 0.0000000000000001334937578747 <-- (4)
```

If we use the result (1) in Section 3.1 with above (2), (3), (4), we can summarize the result as the next table.

| Case | Interval(h) | h^5 | h^5 ratio | Error | Error ratio |
|------|-------------|---------|-------------|--------------------------|-------------|
| (1) | 5.1799 | 3729.1 | 1 | 5.2501×10^{-12} | 1 |
| (2) | 2.9129 | 209.71 | 0.056236 | 2.9574×10^{-13} | 0.056330 |
| (3) | 1.6380 | 11.792 | 0.0031622 | 1.6649×10^{-14} | 0.0031712 |
| (4) | 0.92114 | 0.66318 | 0.00017784 | 9.3870×10^{-16} | 0.00017880 |

We can see in the table that the Error in Dormand-Prince method is proportional to h^5 . It means that the Dormand-Prince method is a 5th order method.

3.2 1/10 of the Stationary Velocity at Stationary Distance

The following results are for a satellite whose initial distance from the center of the Earth is same with a Stationary Satellite though the initial velocity is 1/10 of the Stationary Satellite.

```

----- Runge-Kutta-Fehlberg -----
(Input)
10000.0
42242276.53890282602184866499414568877931
307.194503809087027757155147883394003751
90.0
0.0
15388.7779177688999563149786986655582348 (Half of the cycle)
0.000000001
0.1
1.0
0.000000000001
(Output)
30471 15388.773207974 ( -212272.64830618783554 287.91775134215767149 -41.666540700918006214 -61131.67800643828119) err= 0.0000000000000000611747448814844
30472 15388.774431464 ( -212272.69266329043457 213.12374758092669446 -30.842596882537118837 -61131.69077488588809) err= 0.0000000000000000611746564145237
30473 15388.7756549532 ( -212272.72377738551917 138.32977862276120991 -20.018652385971009045 -61131.699735325113206) err= 0.0000000000000000611745777729708
30474 15388.7768784418 ( -212272.74164849273919 63.535837121182613381 -9.1947081536148627746 -61131.704881963839669) err= 0.0000000000000000611745089567641
30475 15388.7779177689 ( -212272.7464266473865 -1.4054790473810616087e-09 2.0316753626305444887e-10 -61131.706258008314348) err= 0.0000000000000000270602826765095 <-- (1)

(Input)
10000.0
42242276.53890282602184866499414568877931
307.194503809087027757155147883394003751
90.0
0.0
30777.55583553779991262995739733311164696 (One cycle)
0.000000001
0.1
1.0
0.000000000001
(Output)
60957 30773.5826513158 ( 42242274.775599681179 -1220.5403386418256608 0.88760201845929504944 307.19449098598419013) err= 0.00000000000000004885529886160
60958 30774.5826513158 ( 42242275.551502618241 -913.34584469919546356 0.66420385701191643455 307.1944966285099404 ) err= 0.00000000000000004883974346970
60959 30775.5826513158 ( 42242276.10400739751 -606.15134592633827203 0.44080570248443916949 307.19450064643821235) err= 0.00000000000000004882753575690
60960 30776.5826513158 ( 42242276.433114024741 -298.95684394785147484 0.2174075525494252356 307.19450303976917406) err= 0.00000000000000004881867822954
60961 30777.5558355378 ( 42242276.538902824598 -3.8960276984704254866e-10 -1.7858511798951323993e-13 307.19450380908703816) err= 0.00000000000000004261006322550 <-- (2)

----- Dormand-Prince -----
(Input)
10000.0
42242276.53890282602184866499414568877931
307.194503809087027757155147883394003751
90.0
0.0
15388.7779177688999563149786986655582348 (Half of the cycle)
0.000000001
0.1
1.0
0.000000000001
(Output)
29712 15388.7731456702 ( -212272.64569303046746 291.72649148596424288 -42.217728939024950874 -61131.677248090233335) err= 0.0000000000000000636927223940551
29713 15388.774419517 ( -212272.6922941803321 213.85409228272026056 -30.948290012110520371 -61131.690668587189117) err= 0.0000000000000000636925982985074
29714 15388.7756933627 ( -212272.72453976568336 135.98174202265018144 -19.678851951991907886 -61131.699954880347873) err= 0.0000000000000000636924862426213
29715 15388.7769672076 ( -212272.74242981495462 68.109431252636931581 -8.4094156541670962699 -61131.705106974167643) err= 0.0000000000000000636923862262494
29716 15388.7779177689 ( -212272.74642664736695 -3.5540264946138800438e-13 5.1178878598501297188e-14 -61131.706258008318523) err= 0.0000000000000000147368807952994 <-- (3)

(Input)
10000.0
42242276.53890282602184866499414568877931
307.194503809087027757155147883394003751
90.0
0.0
30777.55583553779991262995739733311164696 (One cycle)
0.000000001
0.1
1.0
0.000000000001
(Output)
59443 30774.1148845162 ( 42242275.216369703136 -1057.0412306936826671 0.76870209903181278324 307.19449419135445393) err= 0.00000000000000000933761077068
59444 30775.1148845162 ( 42242275.87337272286 -749.84673397801458589 0.54530394155715611885 307.19449896921546393) err= 0.00000000000000000929565302232
59445 30776.1148845162 ( 42242276.306977588144 -442.65223329678419866 0.32190578976366097807 307.19450212247909353) err= 0.00000000000000000926403209970
59446 30777.1148845162 ( 42242276.517184303505 -135.45773027458881423 0.098507641323889498894 307.19450365114547464) err= 0.00000000000000000924285408030
59447 30777.5558355378 ( 42242276.538902826042 -3.3538089515393454569e-12 2.443904942527998422e-15 307.19450380908702761) err= 0.0000000000000000015390298251 <-- (4)

```

We see in (1), (2), (3), (4) that the Dormand-Prince method is roughly 100 times or more accurate than the Runge-Kutta-Fehlberg method notwithstanding the iteration is 3 percent smaller than that of the Runge-Kutta-Fehlberg method. The calculation time intervals are almost same for both method even at around the perigee. (See (1), (3))

The errors in Dormand-Prince method are in the same order with that of the Runge-Kutta-Fehlberg method as it was in stationary case.

3.3 1/100 of the Stationary Velocity at Stationary Distance

The following results are for a satellite whose initial distance from the center of the Earth is same with a Stationary Satellite though the initial velocity is 1/100 of the Stationary Satellite.

```

----- Runge-Kutta-Fehlberg -----
(Input)
10000.0
42242276.53890282602184866499414568877931
30.7194503809087027757155147883394003751
90.0
0.0
15274.65205821368691354466399621963063851 (Half of the cycle)
0.000000001
0.1
1.0
0.000000000001
(Output)
109914 15274.6520575902 ( -2112.2194205505561778 0.38304038831379610027 -55.708180195315361447 -614358.28311659376756) err= 0.0000000000000000080707048123
109915 15274.6520577516 ( -2112.2194283786518589 0.28387430770086006911 -41.285779848017109454 -614358.28539346676412) err= 0.0000000000000000080707046194

```

```

109916 15274.652057913 ( -2112.2194338787689157 0.18470822783775143253 -26.863379473275901629 -614358.28699322596402)err= 0.00000000000000000080707044479
109917 15274.6520580744( -2112.219437050907404 0.085542148572472906837 -12.440979096667829144 -614358.28791587137292)err= 0.00000000000000000080707042976
109918 15274.6520582137( -2112.2194379170372367 -1.5133909203982008156e-08 2.2010317224353568029e-06 -614358.28816779313426)err= 0.00000000000000000038547558784 <-- (1)

(Input)
10000.0
42242276.53890282602184866499414568877931
30.7194503809087027757155147883394003751
90.0
0.0
30549.30411642737382708932799243926127702 (One cycle)
0.000000001
0.1
1.0
0.000000000001
(Output)
219861 30545.8237083759( 42242275.185865095469 -106.91622130122564696 0.77751672720002241432 30.719449396951876102)err= 0.000000000000000000519194536136
219862 30546.8237083759( 42242275.85168274313 -76.196771648637318306 0.55411856929616936487 30.719449881148156546)err= 0.000000000000000000505557942587
219863 30547.8237083759( 42242276.294102236014 -45.477321593082580102 0.33072041725234613359 30.719450202884698048)err= 0.000000000000000000494958992457
219864 30548.8237083759( 42242276.513123578801 -14.75787129702116391 0.10732226870602605971 30.719450362161514219)err= 0.000000000000000000487595810790
219865 30549.3041164274( 42242276.538902819792 1.1770364682355309713e-09 -7.6663091195437848983e-13 30.71945038090870731 )err= 0.00000000000000000012370540860 <-- (2)

----- Dormand-Prince -----
(Input)
10000.0
42242276.53890282602184866499414568877931
30.7194503809087027757155147883394003751
90.0
0.0
15274.65205821368691354466399621963063851 (Half of the cycle)
0.000000001
0.1
1.0
0.000000000001
(Output)
100839 15274.6520576357( -2112.2194229916320687 0.35510085582089930776 -51.644743191510331989 -614358.2838266029805 )err= 0.00000000000000000087563874905
100840 15274.6520578108( -2112.2194306658841422 0.24750968090356267703 -35.997023787974433685 -614358.28605872916622)err= 0.00000000000000000087563872351
100841 15274.6520579859( -2112.219435599786502 0.13991850686017837572 -20.349304371452029072 -614358.28749379966105)err= 0.00000000000000000087563870061
100842 15274.6520581611( -2112.2194377933392161 0.032327333506473418539 -4.7015849760400290869 -614358.28813181447366)err= 0.00000000000000000087563868034
100843 15274.6520582137( -2112.219437917037153 -3.5057110288895679001e-12 5.0985962015474715084e-10 -614358.28816779314681)err= 0.0000000000000000000012370540860 <-- (3)

(Input)
10000.0
42242276.53890282602184866499414568877931
30.7194503809087027757155147883394003751
90.0
0.0
30549.30411642737382708932799243926127702 (One cycle)
0.000000001
0.1
1.0
0.000000000001
(Output)
201717 30545.8697362361( 42242275.221415891145 -105.50227077889209709 0.76723418785268492104 30.719449422805137045)err= 0.000000000000000000175575201561
201718 30546.8697362361( 42242275.876950999629 -74.78282110418934485 0.54383603027042601524 30.719449899523742942)err= 0.000000000000000000150850910192
201719 30547.8697362361( 42242276.309087953603 -44.063371033997857186 0.32043787843945287427 30.719450213782610676)err= 0.000000000000000000129236862186
201720 30548.8697362361( 42242276.517826757638 -13.343920730777365046 0.097039729997241849755 30.719450365581753538)err= 0.000000000000000000112539580042
201721 30549.3041164274( 42242276.538902825874 2.4647785203979219144e-12 -1.7921943959999136459e-14 30.719450380908702884)err= 0.000000000000000000001592814271 <-- (4)

```

We see in (1), (2), (3), (4) that the Dormand-Prince method is 500 times or more accurate than the Runge-Kutta-Fehlberg method notwithstanding the iteration is 8 percent smaller than that of the Runge-Kutta-Fehlberg method. The calculation time intervals are almost same for both method even at around the perigee. (See (1), (3))

The errors in Dormand-Prince method are in the same order with that of the Runge-Kutta-Fehlberg method as it was in stationary case and in the case immediately before.

4 Appendix (Program for Dormand-Prince Method)

```

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define M_PIL 3.1415926535897932384626433832795028841968L

int giogrv(long double m,long double x[3],long double f[3]);
int f(long double t,long double r[6],long double rd[6]);
int initd(long double *m1,long double r[6],long double *t0,long double *tf,long double *hmin,long double *h,long double *hmax,long double *eps);
int onestep(long double h0,long double t0,long double r0[6],long double hmin,long double hmax,long double eps,
long double *t1,long double r1[6],long double *h1,long double *erra);
int print(long double t,long double r[6]);

long double m1; //mass 1

int main()
{
    long double r0[6],t0,r1[6],eps,t1,tf,h0,h1,erra,hmin,hmax;
    int i;
    long j;

    initd(&m1,r0,&t0,&tf,&hmin,&h0,&hmax,&eps);

    j=0;
    while(t0<tf)
    {
        if((t0+h0) >= tf){h0=tf-t0;}
        onestep(h0,t0,r0,hmin,hmax,eps,&t1,r1,&h1,&erra);
        j++;
        printf("%7ld ",j);print(t1,r1);printf("err=%33.30Lf\n",erra);
    }
}

```

```

        if(h1/h0 > 0.5L)
        {
            t0=t1;
            for(i=0;i<6;i++)r0[i]=r1[i];
            h0=h1;
        }
        else
        {
            h0=h1;
        }
    }
    return 0;
}

/*
gravitational force by earth.
m: mass of the object.(receive)
x[3]: position vector.(receive)
f[3]: gravitational force vector.(return)
giogrv=0: normal.(return)
jan.27.2009,toshinori kimura
*/
int giogrv(long double m,long double x[3],long double *f)
{
    long double G=6.67259e-11; //gravitational constant.
    long double M=5.9742e+24; //mass of the earth.
    long double r; //radius from the center of the earth to the object.
    long double GMmovR3;
    int i;
    for(r=0,i=0;i<3;i++)r+=x[i]*x[i];
    r=sqrtl(r);
    GMmovR3=G*M*m/(r*r*r);
    for(i=0;i<3;i++)f[i]=-GMmovR3*x[i];
    return 0;
}

/*
runge kutta function
m1: mass of the object at x1.(external)
t: time.(receive)
r[6]: position vector,x1,x1d.(receive)
rd[6]: derivertive of position vector.(return)
f=0: normal.(return)
feb.3.2009,toshinori kimura
*/
int f(long double t,long double r[6],long double rd[6])
{
    long double *x1; //position x1
    long double *x1d; //velocity x1

    long double *ox1d; //output velocity x1
    long double *ox1dd; //output acceleration x1

    int i;

    long double fx1[3]; //force for x1;

    x1 =r+0;
    x1d=r+3;

    ox1d =rd+0;
    ox1dd=rd+3;

    for(i=0;i<3;i++)ox1d[i]=x1d[i];
    giogrv(m1,x1,fx1);
    for(i=0;i<3;i++)ox1dd[i]=fx1[i]/m1;
    return 0;
}

/*
initial data input
m1:mass 1.(return)
h:time interval.(return)
t0:initial time.(return)
tf:final time.(return)
r:initial vector for runge kutta,x1,x1d.(return)
eps:error tolerance.(return)
initd=0:normal.(return)
feb.13.2009,toshinori kimura
*/
int initd(long double *m1,long double r[6],long double *t0,long double *tf,long double *hmin,long double *h0,long double *hmax,long double *eps)
{
    char initdfname[50];
    FILE *fp;
    long double R1,V1,V1phy;
    printf("m1:mass 1.\n");
    printf("R1:radius of m1 from center of earth.\n");
    printf("  define x,y,z axis as\n");
    printf("  x:direction of R1.\n");
    printf("  y:orthogonal to x, in R1-V1 plane,(V1.y)>0.(see V1 below)\n");
    printf("  z:direction x,y,z make right handed coordinate system.\n");
    printf("V1:velocity of m1.\n");
    printf("V1phy:angle of V1 from x.\n");
    printf("t0:starting time.\n");
    printf("tf:final time.\n");
    printf("hmin:h minimum.\n");
    printf("h0:initial calculation interval.\n");
    printf("hmax:h maximum.\n");
    printf("eps:error tolerance");

    printf("initial data file name:");

```

```

scanf("%s",initdfname);
fp=fopen(initdfname,"r");
if(fscanf(fp,"%Lg %Lg %Lg %Lg %Lg %Lg %Lg %Lg %Lg",m1,&R1,&V1,&V1phy,t0,tf,hmin,h0,hmax,eps))
{
    fclose(fp);

    V1phy =V1phy *(M_PIl)/180;

    printf("m1      =%Lg\n",*m1 );
    printf("R1      =%Lg\n",R1 );
    printf("V1      =%Lg\n",V1 );
    printf("V1phy    =%Lg\n",V1phy );
    printf("t0      =%Lg\n",*t0 );
    printf("tf      =%Lg\n",*tf );
    printf("hmin     =%Lg\n",*hmin );
    printf("h0      =%Lg\n",*h0 );
    printf("hmax     =%Lg\n",*hmax );
    printf("eps      =%Lg\n",*eps );

}
else
{
    fclose(fp);
    printf("fscanf ng\n");
    return 1;
}

r[0]=R1;
r[1]=0;
r[2]=0;
r[3]=V1*cos1(V1phy);
r[4]=V1*sin1(V1phy);
r[5]=0;

return 0;
}

/*
one step calculation
h:time interval.(receive)
t0:original time.(receive)
r0:original vector for runge kutta.(receive)
t1:resulting time.(return)
r1:resulting vector for runge kutta.(return)
onestep=0:normal.(return)
feb.14.2009,toshinori kimura
*/
int onestep(long double h,long double t0,long double r0[6],long double hmin,long double hmax,long double eps,
long double *t1,long double r1[6],long double *h1,long double *erra)
{
    long double k1[6],k2[6],k3[6],k4[6],k5[6],k6[6],k7[6];
    long double a2=1.0L/ 5,b21= 1.0L/ 5;
    long double a3=3.0L/10,b31= 3.0L/ 40,b32= 9.0L/ 40;
    long double a4=4.0L/ 5,b41= 44.0L/ 45,b42= -56.0L/ 15,b43= 32.0L/ 9;
    long double a5=8.0L/ 9,b51=19372.0L/ 6561,b52=-25360.0L/2187,b53=64448.0L/ 6561,b54=-212.0L/ 729;
    long double a6=1.0L ,b61= 9017.0L/ 3168,b62= -355.0L/ 33,b63=46732.0L/ 5247,b64= 49.0L/ 176,b65= -5103.0L/ 18656;
    long double a7=1.0L ,b71= 35.0L/ 384, b73= 500.0L/ 1113,b74= 125.0L/ 192,b75= -2187.0L/ 6784,b76= 11.0L/ 84;
    long double c1= 35.0L/ 384, c3= 500.0L/ 1113, c4= 125.0L/ 192, c5= -2187.0L/ 6784, c6= 11.0L/ 84;
//long double d1= 5179.0L/57600, d3= 7571.0L/16695, d4= 393.0L/ 640, d5=-92097.0L/339200, d6=187.0L/2100,d7= 1.0L/40;
    long double e1= 71.0L/57600, e3= -71.0L/16695, e4= 71.0L/1920, e5=-17253.0L/339200, e6= 22.0L/ 525,e7=-1.0L/40;
    long double tk0 ,tk1 ,tk2 ,tk3 ,tk4 ,tk5 ,tk6;
    long double yk0[6],yk1[6],yk2[6],yk3[6],yk4[6],yk5[6],yk6[6];
    long double rz[6],err[6];

    long double s;

    int i;

// printf("a2=%40.30Lf c5=%40.30Lf\n",a2,c5);
tk0=t0;
for(i=0;i<6;i++)yk0[i]=r0[i];
f(tk0,yk0,k1);
for(i=0;i<6;i++)k1[i]*=h;

tk1=t0+a2*h;
for(i=0;i<6;i++)yk1[i]=r0[i] + b21*k1[i];
f(tk1,yk1,k2);
for(i=0;i<6;i++)k2[i]*=h;

tk2=t0+a3*h;
for(i=0;i<6;i++)yk2[i]=r0[i] + b31*k1[i] + b32*k2[i];
f(tk2,yk2,k3);
for(i=0;i<6;i++)k3[i]*=h;

tk3=t0+a4*h;
for(i=0;i<6;i++)yk3[i]=r0[i] + b41*k1[i] + b42*k2[i] + b43*k3[i];
f(tk3,yk3,k4);
for(i=0;i<6;i++)k4[i]*=h;

tk4=t0+a5*h;
for(i=0;i<6;i++)yk4[i]=r0[i] + b51*k1[i] + b52*k2[i] + b53*k3[i] + b54*k4[i];
f(tk4,yk4,k5);
for(i=0;i<6;i++)k5[i]*=h;

tk5=t0+a6*h;
for(i=0;i<6;i++)yk5[i]=r0[i] + b61*k1[i] + b62*k2[i] + b63*k3[i] + b64*k4[i] + b65*k5[i];
f(tk5,yk5,k6);
for(i=0;i<6;i++)k6[i]*=h;

tk6=t0+a7*h;
for(i=0;i<6;i++)yk6[i]=r0[i] + b71*k1[i] + b73*k3[i] + b74*k4[i] + b75*k5[i] + b76*k6[i];

```



```

f(tk6,yk6,k7);
for(i=0;i<6;i++)k7[i]*=h;

*t1=t0 + h;
for(i=0;i<6;i++) r1[i]=r0[i] + c1*k1[i] + c3*k3[i] + c4*k4[i] + c5*k5[i] + c6*k6[i];
//for(i=0;i<6;i++) rz[i]=r0[i] + d1*k1[i] + d3*k3[i] + d4*k4[i] + d5*k5[i] + d6*k6[i] + d7*k7[i];

for(i=0;i<6;i++)err[i]= e1*k1[i] + e3*k3[i] + e4*k4[i] + e5*k5[i] + e6*k6[i] + e7*k7[i];
*erra=0.0L;
for(i=0;i<6;i++)*erra += (err[i])*(err[i]);
*erra=sqrt1(*erra);

// for(i=0;i<6;i++)err[i]=rz[i]-r1[i];
// errb=0.0L;
// for(i=0;i<6;i++)errb += err[i]*err[i];
// errb=sqrt1(errb);

// printf("erra=%40.30Lf, errb=%40.30Lf\n",erra, errb);
// printf("tel h/2.0erra=%40.30Lf\n",tol*h/(2.0L*erra));

if(*erra > 0.0L)
{
s=powl(eps*h/(2.0L*(*erra)),0.2L);
}
else
{
s=4.0L;
}

// printf("s=%40.30Lf\n",s);

if(s<0.25L)
{
s=0.25L;
}
else if(s>4.0L)
{
s=4.0L;
}

// printf("s=%40.30Lf\n",s);

*h1=s*h;

if(*h1 < hmin)
{
*h1=hmin;
}
else if(*h1 > hmax)
{
*h1=hmax;
}

return 0;
}

/*
print data
t:time.(receive)
r:array of runge-kutta.(receive)
print=0:normal.(return)
feb.14.2009,toshinori kimura
*/
int print(long double t,long double r[6])
{
long double *x1,*x1d;
int i;
x1 =&r[0];
x1d=&r[3];

printf("%18.15Lg",t);
printf("(");
for(i=0;i<2;i++)printf(" %25.20Lg",x1[i]);
for(i=0;i<2;i++)printf(" %25.20Lg",x1d[i]);
printf(")");

return 0;
}

```