

PHYSICS 115/242
Homework 3, Solutions

1. BISECTION METHOD

SOURCE CODE

```
f(x) = x**3 - 5*x + 3

print '(A,$)', "xmin = ? xmax = ? "
read *, xmin, xmax
print '(A, f10.4, 2x, A, f10.4)', "xmin = ", xmin, "xmax = ", xmax

isignl = 1
if (f(xmin) < 0) isignl = -1
isignu = 1
if (f(xmax) < 0) isignu = -1
if (isignu == isignl) stop "root not bracketed"

do while (abs(xmax - xmin) > 1.e-4)
  xnew = 0.5*(xmin + xmax)
  isignnew = 1
  if (f(xnew) < 0) isignnew = -1
  if (isignnew == isignl) then
    xmin = xnew
  else
    xmax = xnew
  end if
  print '(2f12.5)', xnew, f(xnew)
end do
print '(/A, f10.5)', "root = ", xnew

end
```

Output:

From a sketch, done in class, the two positive roots lie between 0 and 1, and 1 and 2.

```
xmin = ? xmax = ?
xmin =    0.0000  xmax =    1.0000
  xnew      f(xnew)
  0.50000   0.62500
  0.75000  -0.32813
  0.62500   0.11914
  0.68750  -0.11255
  0.65625   0.00137
```

0.67188	-0.05608
0.66406	-0.02747
0.66016	-0.01308
0.65820	-0.00586
0.65723	-0.00225
0.65674	-0.00044
0.65649	0.00047
0.65662	0.00002
0.65668	-0.00021

root = 0.65668

xmin = ? xmax = ?

xmin = 1.0000 xmax = 2.0000

xnew	f(xnew)
1.50000	-1.12500
1.75000	-0.39063
1.87500	0.21680
1.81250	-0.10815
1.84375	0.04892
1.82813	-0.03096
1.83594	0.00865
1.83203	-0.01124
1.83398	-0.00132
1.83496	0.00366
1.83447	0.00117
1.83423	-0.00007
1.83435	0.00055
1.83429	0.00024

root = 1.83429

2. NEWTON-RAPHSON

SOURCE CODE

```

real*8 f, fp, x
f(x) = x**3 - 5*x + 3
fp(x) = 3*x**2 - 5

print '(A,$)', "x = ? "
read *, x
print '(2f20.15)', x, f(x)

do while (abs(f(x)) > 1.d-15)

```

```

    x = x - f(x)/fp(x)
    print '(2f20.15)', x, f(x)
end do

print '(/A, f20.15)', "root = ", x

```

END

Output:

Feed in the results from Qu. 1 for the initial guess.

```

x = ? 0.65668
    0.6566800000000000    -0.000220788242368
    0.656620429161032    0.000000006990840
    0.656620431047110    0.000000000000000

```

```

root = 0.656620431047110

```

```

x = ? 1.83429
    1.8342900000000000    0.000238460462589
    1.834243186681593    0.000000012059364
    1.834243184313922    0.000000000000000

```

```

root = 1.834243184313922

```

Notice how fast one gets the answer to machine precision.

3. We keep track of the current value, x , the previous value, x_p , and the new value, x_n . We stop when $|x_n - x| < EPS$, the desired precision. We took the first two values to be 0.5 and 1. We see from the output that successive values converge rapidly to the root at $x = 0.957504024077$. Note that everything is in double precision.

SOURCE CODE:

```

REAL(8) xp, xn, x, f, EPS
f(x) = x - tanh (2 * x)
EPS = 1.0d-12

print '(A,$)', "x0 = ? x1 = ? "
read *, xp, x
print '(A, f10.4, 2x, A, f10.4)', "x0 = ", xp, "x1 = ", x

xn = 1.e+6
do

```

```

        xn = x - f(x) * (x - xp) / (f(x) - f(xp))
        print '(f16.12)', xn
        if (abs(xn - x) < EPS) exit
        xp = x
        x = xn
    enddo
print '(/A, f16.12)', "root = ", xn

```

END

OUTPUT:

```

x0 = ? x1 = ? 0.5 1
x0 =      0.5000  x1 =      1.0000
  0.939555677889
  0.957223306557
  0.957506007248
  0.957504023864
  0.957504024077
  0.957504024077

```

```

root =      0.957504024077

```

4. Sqrt(2)

SOURCE CODE

(This is virtually the same as for Qu. 2)

```

real*8 f, fp, x
f(x) = x*x - 2
fp(x) = 2*x

print '(A,$)', "x = ? "
read *, x
print '(2f20.15)', x, f(x)

do while (abs(f(x)) > 1.d-15)
    x = x - f(x)/fp(x)
    print '(2f20.15)', x, f(x)
end do

print '(/A, f20.15)', "root = ", x

end

```

OUTPUT

x = ? 1.5

1.5000000000000000	0.2500000000000000
1.4166666666666667	0.0069444444444445
1.414215686274510	0.000006007304883
1.414213562374690	0.000000000004511
1.414213562373095	0.0000000000000000

root = 1.414213562373095

5. Use of RK2:

The analytic solution is easily obtained from separating the variables. It is $y = \tan(x)$, so $y(\pi/4) = 1$.

This problem is easier to code than a more general problem because the expression for dy/dx is independent of x .

I give a code in C!

SOURCE CODE

```
double f(double y)
{
    return 1 + y*y;           // the RHS of the equation
}

main()
{
    double f(), x, y, h, k1, k2, xn, x0, y0, PI04;
    int nx, i, idouble;

    PI04 = atan(1.0);
    x0 = 0;
    xn = PI04;
    y0 = 0;
    h = xn - x0;
    nx = round(1 / h);
    printf ("      h          error          error/h^2 \n");

                                // Double no. of intervals 8 times
    for (idouble = 0; idouble < 8; idouble++)
    {
        h = h / 2;
        nx = 2 * nx;
```

```

y = y0;

for (i = 0; i < nx; i++)                // Do RK2 for a given n
{
    k1 = f(y);
    k2 = f(y + h * k1);
    y = y + 0.5 * h * (k1 + k2);
}

printf (" %10.5f %14.8f %10.5f \n", h, y - 1, (y-1)/(h*h));
}
}

```

OUTPUT

h	error	error/h ²
0.39270	0.00491811	0.03189
0.19635	0.00112970	0.02930
0.09817	0.00032808	0.03404
0.04909	0.00009464	0.03928
0.02454	0.00002582	0.04286
0.01227	0.00000677	0.04493
0.00614	0.00000173	0.04604
0.00307	0.00000044	0.04661

Here the error is $y(1) - \pi/4$. Clearly the error goes down like h^2 for small h .

6. (a) This question requires a bit more thought. We first of all need to decide when the particle has completed one oscillation. I did this by determining the time when the momentum changes sign. This is half the period so I multiplied this time by 2. (I did a linear extrapolation between the last two times to determine more precisely the time at which the momentum vanished).

The other issue is to estimate whether the time step is small enough that reasonable accuracy has been achieved. I did this by checking whether the value of x after a half period is equal to $-x_0$ (where x_0 is the initial displacement, i.e. the amplitude of the oscillation) to within some tolerance, which I took to be 10^{-4} . The source code below incorporates these features and uses 2nd order Runge Kutta (RK2) to integrate the motion.

Note: I checked the code for the simple harmonic oscillator, by setting $f(x) = -x$ and checking that the period was (close to) 2π for any initial displacement.

SOURCE CODE

```

real*8 x, p, t, h, k1x, k2x, k1p, k2p, x0, pold, xold, thalf

```

```

logical half_orbit

f(x) = -x**3 !This is the force
half_orbit = .FALSE.

print '(A, $)', "x0 = ? h = ? "
read *, x0, h
print '(A, f10.5, 4x, A, f10.5)', "x0 = ", x0, "h = ", h

x = x0
p = 0
t = 0

do while ( .NOT. half_orbit )      !Do one time step
  pold = p
  k1x = p
  k1p = f(x)
  k2x = p + h * k1p
  k2p = f(x + h * k1x)
  x = x + 0.5d0 * h * (k1x + k2x)
  p = p + 0.5d0 * h * (k1p + k2p)
  t = t + h
  if (p > 0) then !We've done half an orbit
    half_orbit = .TRUE.
    thalf = t - h * p / (p - pold) !linear interpolation to determine the
                                   !time where p went through zero
    if ( abs(x+x0) > 1.e-4) then
      stop "error in x too great; need to reduce h"
    end if
  end if
end do

print '(A, f10.5)', "period = ", 2 * thalf

end

```

Output:

Here is some sample output, which also shows the error message generated if the time step h is not small enough.

```

x0 = ? h = ? 0.1 0.1
x0 = 0.10000 h = 0.10000
period = 74.16066

```

```

x0 = ? h = ? 1 0.1

```

```
x0 = 1.00000 h = 0.10000
STOP error in x too great; need to reduce h
```

```
x0 = ? h = ? 1 0.01
x0 = 1.00000 h = 0.01000
period = 7.41607
```

```
x0 = ? h = ? 10 0.001
x0 = 10.00000 h = 0.00100
period = 0.74161
```

The first, third and fourth output give the period for the initial displacements 0.1, 1 and 10. The second output gives an error message because h was too big to give the desired accuracy.

- (b) For the simple harmonic oscillator the force and (hence the velocity) increase as the amplitude increases. Hence, although the particle has further to go, it goes more quickly and these two effects compensate in the determination of the period. For the x^4 potential the force increases very fast as the amplitude increases, so the increase in velocity is greater than the increase in distance, so the period goes down (apparently like $1/\text{amplitude}$) as the amplitude increases.

7. (242 students)

SOURCE CODE

```
IMPLICIT NONE
```

```
REAL(8) a, func, al, ag, amid, eps
```

```
a = 0.5
```

```
print '(A, f8.5,2x, A, f10.5)', " For a = ", a, "the integral is", func(a)
```

```
a = 1.0
```

```
print '(A, f8.5,2x, A, f10.5)', " For a = ", a, "the integral is", func(a)
```

```
print '(A)', "Hence we see that the solution is bracketed by 0.5 and 1.0"
```

```
print '(A)', "Now improve this by bisection"
```

```
eps = 1.0d-5
```

```
al = 0.5
```

```
ag = 1.0
```

```
do while (abs(ag - al) > eps)
```

```
    amid = 0.5 * (ag + al)
```

```
    if (func(amid) - 0.5 < 0) then
```

```
        al = amid
```

```
    else
```

```

        ag = amid
    end if
enddo

print '(A, f10.5)', " The value of a is ", 0.5d0*(a1 + ag)

END

REAL(8) FUNCTION func(a) ! This does the integral by the trapezium rule
IMPLICIT NONE
REAL(8) a, x, TWO_over_PI, f, sum, oldsum, h
INTEGER idouble, n, j

f(x) = exp(-x**2 / 2)
TWO_over_PI = 2 / acos(-1.d0)

n = 1
do idouble = 1, 8
    n = 2 * n
    h = a / n
    sum = 0.5d0 * (f(a) + f(0))
    do j = 1, n - 1
        x = j*h
        sum = sum + f(x)
    end do
    sum = sum * h * sqrt(TWO_OVER_PI)
    if (idouble > 1) then
        if (abs(sum - oldsum) < 1.d-5) exit ! Precision of 10(-5) required
    endif
    oldsum = sum
end do
func = sum
END FUNCTION func

```

Output:

```

For a = 0.50000 the integral is 0.38292
For a = 1.00000 the integral is 0.68269
Hence we see that the solution is bracketed by 0.5 and 1.0
Now improve this by bisection
The value of a is 0.67449

```

8. (242 students)
 From Qu. (5) the period is $T = 7.41607$.
 Firstly we use RK4.

SOURCE

```
REAL*8 T, x, p, t, h, k1x, k2x, k1p, k2p, k3p, k3x, k4p, k4x, energy, enex, de
REAL*8 en
```

```
energy(x, p) = 0.5d0*p**2 + 0.25d0*x**4
T = 7.41607 ! the period
```

```
print '(A, $)', "number of periods = "
read *, n_period
```

```
h = 0.02d0 * T
nt = nint(T * n_period / h)
```

```
de = 0
x = 1
p = 0
t = 0
en = energy(x, p)
enex = energy(x, p)
```

```
do i = 1, nt
  k1x = p
  k1p = -x**3
  k2x = p + 0.5d0 * h * k1p
  k2p = -(x + 0.5d0 * h * k1x)**3
  k3x = p + 0.5d0 * h * k2p
  k3p = -(x + 0.5d0 * h * k2x)**3
  k4x = p + h * k3p
  k4p = -(x + h * k3x)**3
  x = x + h * (k1x + 2*k2x + 2*k3x + k4x)/6.d0
  p = p + h * (k1p + 2*k2p + 2*k3p + k4p)/6.d0
  en = energy(x, p)
  if(abs(en - enex) > de) de = en - enex
  t = t + h
end do
```

```
print '(A, e14.5)', "Max deviation of energy is ", de
```

END

OUTPUT

```
number of periods = 1
Max deviation of energy is -0.80003E-05
```

```
number of periods = 1000
Max deviation of energy is -0.76956E-02
```

We see that the maximum deviation of the energy from the correct value increases about 1000 times when we increase the time by a factor of 1000. Hence we infer that the deviation of the energy increases proportional to time using RK4.

Next we use the PEFRL 4-th order symplectic algorithm described in the handout.

SOURCE

```
IMPLICIT NONE
REAL(8) x, v, f, h, half, third, en_exact, xi_h, chi_h, lambda_h, t, en
REAL(8) T, maxerr, half_one_m_two_lambda_h, one_m_two_chi_plus_xi_h
REAL(8) xi, lambda, chi
INTEGER i, nstep, nperiod

f(x) = -x**3
en(x, v) = x**4/4 + v**2/2
print '(A, $)', " no. of periods = ? "
read *, nperiod

T = 7.41607
h = T * 0.02
half = 1.0_8 / 2.0_8
xi = 0.1786178958448091
lambda = -0.2123418310626054
chi = -0.06626458266981849

xi_h = xi * h
half_one_m_two_lambda_h = half * (1 - 2 * lambda) * h
chi_h = chi * h
lambda_h = lambda * h
one_m_two_chi_plus_xi_h = (1 - 2 * (chi + xi)) * h

maxerr = 0
x = 1; v = 0
t = 0
en_exact = en(x, v)

do while (t <= nperiod)
    x = x + xi_h * v
    v = v + half_one_m_two_lambda_h * f(x)
    x = x + chi_h * v
    v = v + lambda_h * f(x)
    x = x + one_m_two_chi_plus_xi_h * v
```

```

    v = v + lambda_h * f(x)
    x = x + chi_h * v
    v = v + half_one_m_two_lambda_h * f(x)
    x = x + xi_h * v
    t = t + h / T
    if (abs(en(x, v) - en_exact) > maxerr) maxerr = abs(en(x, v) - en_exact)
enddo
print '(A, e14.5)', "Max deviation of energy is ", maxerr

END

```

```

OUTPUT
no. of periods = ? 1
Max deviation of energy is      0.22970E-06

no. of periods = ? 1000
Max deviation of energy is      0.22970E-06

```

We see that, unlike for RK4, the maximum error in the energy does *not* continue to grow with time. This is because the symplectic nature of the algorithm gives it a global stability. Even for 1 period, the PEFRL is a little better than RK4. Nonetheless PEFRL is not significantly more complicated than RK4. (The main difficulty is that the values of χ, ξ and λ have to be cut and pasted into the code.)

Note: for the Forrest Ruth algorithm, the max error in energy is 0.7822×10^{-4} , both after 1 period and 1000 periods.