Semi-Classical Quantization of Molecular Vibrations

Michael Papasimeon
12 August 1997

I. Quadrature

A. Aims

THE main aim is to write a FORTRAN program to numerically integrate the integral:

$$\int_0^1 \frac{\ln(1+x)}{x} dx = \frac{\pi^2}{12} \tag{1}$$

- using the Trapezoidal rule,
- using Simpson's method,
- and to investigate the accuracy of both the Trapezoidal rule and Simpson's method for different step sizes.

B. Procedure

A FORTRAN program was written (`quad.f`, source code in Appendix A), which implemented both the Trapezoidal and Simpson's rules for the quadrature of equation 1. The program calls the functions to calculate the integral a number of times each with a diiferent number of divisions n, and hence step size h. Both the Trapezoidal and Simpson's rules are evaluated at number of divisions starting from $n = 10$ to $n = 10^6$, with each new evaluation increasing by a single order of magnitude. This allows the investigation of the accuracy of the different techniques as the number of divisions increases.

The error is calculated is the absolute error:

$$Error = \text{abs}(Result - Answer)$$

where Result is the value produced by the numerical technique and $Answer$, is the analytic solution to the integral known to be $\frac{\pi^2}{12} \simeq 0.822467033$.

Since the integrand is not defined at $x = 0$, the FORTRAN function used to evaluate this returns a value of 1.0 when $x = 0$, this is because

$$\lim_{x \to 0} \frac{\ln(1+x)}{x} = 1$$

.

C. Results

1) Trapezoidal Rule: The output from the program `quad.f` using the Trapezoidal rule is shown below in in Table I

2) Simpson's Rule: The output from the program `quad.f` using Simpson's rule is shown below in Table II.

TABLE I
Results of using Trapezoidal rule.

n	h	$Result$	$Error$
10	.1000000000	.8227225585	.255525E-03
100	.0100000000	.8224695905	.255709E-05
1000	.0010000000	.8224670590	.255711E-07
10000	.0001000000	.8224670337	.255709E-09
100000	.0000100000	.8224670334	.255251E-11
1000000	.0000010000	.8224670334	.263123E-13

TABLE II
Results using Simpson's rule.

n	h	$Result$	$Error$
10	.1000000000	.8224677660	.732614E-06
100	.0100000000	.8224670335	.744940E-10
1000	.0010000000	.8224670334	.799361E-14
10000	.0001000000	.8224670334	.310862E-14
100000	.0000100000	.8224670334	.310862E-14
1000000	.0000010000	.8224670334	.643929E-14

D. Discussion

Looking at the results for the Trapezoidal rule calculations in Table 1, we see that as we increase the number of divisions n by one order of magnitude, the error decreases by 2 orders. Therefore, we see that when using the Trapezoidal rule for quadrature, the error is of order $O(n^2)$, as expected.

Making a comparison with the results for the Simpson's rule calculations shown in Table 2, we see that that as the number of divisions n is increased by an order of magnitude, the error decreases by 4 orders. Therefore, when using Simpson's rule, the error is of order $O(n^4)$. The only problem is that the magnitude of the error stops decreasing after n = 10000. The most likely explanation for this is that we have reached the floating point limit on the number of decimal places which the computer which ran the program can handle.

Therefore we see that that Simpson's rule is more accurate than the trapezoidal rule for quadrature.

II. Root Finding

A. Aims

The main aim is to write a FORTRAN program which makes use of the false position method to calculate the non zero root of the equation

$$\int_0^x t^2 dt = x \tag{2}$$

and to compare with the analytic solution. The program will make use of the Simpson's rule to evaluate the integral.

Procedure

Firstly equation 2 must be solved analytically.

$$\begin{aligned} \int_0^x t^2 dt &= x \\ \left[\frac{t^3}{2}\right]_0^x - x &= 0 \\ \frac{x^3}{3} - x &= 0 \\ x\left(\frac{x^2}{3} - 1\right) &= 0 \end{aligned}$$

Therefore equation 2 has three roots

$$x = 0, \pm\sqrt{3}$$

We are interested in finding the positive root, $x = +\sqrt{3} \simeq 1.732050808$, of equation 2 numerically.

The procedure involved writing a FORTRAN program to find the roots of the equation

$$\int_0^x t^2 dt - x = 0$$

. The algorithm used to find the root of this equation was the false position method. Simpson's rule was also used in the program to evaluate the integral in the equation.

The program outputs a table of results, with varying number of divisions n for the Simpson rule calculations and varying $Tolerance$ for the false position method. The values of n range from $n = 10$ to $n = 10^7$ with each one being an order of magnitude of the previous value. The value for the tolerance of the false position algorithm is simply $1/n$.

The source code for the program `root.f` can be found in Appendix B.

B. Results

The results from the `root.f` program using the false position method is shown in Table III

TABLE III
RESULTS USING THE FALSE POSITION METHOD.

n	$Tolerance$	$Result$	$Error$
10	.1000000000	1.6831581899	.488926E-01
100	.0100000000	1.7301619523	.188886E-02
1000	.0010000000	1.7318112238	.239584E-03
10000	.0001000000	1.7320371397	.136679E-04
100000	.0000100000	1.7320476140	.319354E-05
1000000	.0000010000	1.7320506340	.173520E-06
10000000	.0000001000	1.7320507671	.404475E-07

C. Discussion

Looking at the results in Table 3, we see that as the number of divisions n (for Simpson's rule) is increased in order of magnitude and the tolerance (for false position method) is made smaller, the error decreases by order $O(n)$.

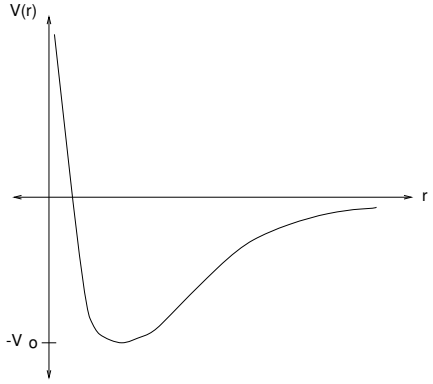
III. MOLECULAR VIBRATIONS

A. Background

When two atoms are bound together in a molecular structure such as the two Hydrogen atoms in a H_2 molecule, they vibrate. Since the nuclei (protons in this case), are much heavier than the electrons the approximation that the protons are infinitely heavier than the electrons can be made. Therefore the potential between the protons depends only on the distance between them.

At large distances, the potential is attractive to a van der Waals interaction and repulsive at short distances due to the Coulombic electrostatic repulsion of like charges, and because of the Pauli exclusion principle which states that no two fermions can occupy the same quantum state.

The interaction of the potential for a diatomic molecule such as H_2 can be summarised in the graph below.



For a quantum system such as this one, Schrodinger's Equation is usually used to solve for the allowed energies E_n of the molecular system.

$$\left[\frac{\hbar}{2m}\frac{d^2}{dr^2} + V(r)\right] = E_n\psi_n \tag{3}$$

Due to the fact that the mass of the protons is approximated to be infinite, the problem can be solved using classical mechanics and then applying quantisation rules of the "old" quantum theory.

The total energy in classical mechanics is given by the sum of the kinetic and potential energies.

$$E = \frac{p^2}{2m} + V(r) \tag{4}$$

Solving for the momentum p we get:

$$p(r) = \pm\sqrt{2m(E - V(r)} \tag{5}$$

To quantize this classical motion, we consider the potential in phase space. The area enclosed by the phase space trajectory is called the "action" is given by $S(E)$, where E is the energy. According the "old" quantum theory quantisation rules, for a given energy E_n, the action must be only half integral multiples of π.

$$S(E_n) = \oint \frac{p(r)}{\hbar} dr \tag{6}$$

$$S(E_n) = 2\sqrt{\frac{2m}{\hbar^2}} \int_{r_{in}}^{r_{out}} \sqrt{E_n - V(r)} dr \tag{7}$$

$$S(E_n) = \left(n + \frac{1}{2}\right)\pi \tag{8}$$

B. Aims

The main aim is to solve the scaled action equation, to find the quantised scaled energy levels ϵ_n for the different quantum levels n.

$$s(\epsilon_n) = \gamma \int_{x_{in}}^{x_{out}} [\epsilon_n - v(x)]^{1/2} dx = \left(n + \frac{1}{2}\right)\pi. \tag{9}$$

The actual energy levels can then be calculated from $E = V\epsilon$. The aim is to solve the equation with a quadratic potential $v(x)$ both analytically and numerically. Then the potential is to be replaced with the Morse potential which can only be solved numerically. The numerical results of the energy then need to be compared with the experimental results obtain for the quantised energy levels of the H_2 molecule.

C. Quadratic Potential Procedure

Using a quadratic potential $V(r)$, equation 9 needs to be solved.

$$V(r) = 4V_0\left(\frac{r}{a} - 1\right)\left(\frac{r}{a} - 2\right)$$

We need to find the roots of $\epsilon_n - v(x) = 0$, $x_{in}(\epsilon_n)$ and $x_{out}(\epsilon_n)$, where $x = r/a$, and $v(x) = V(x)/V_0$ and therefore $v(x) = 4(x-1)(x-2)$.

$$\begin{aligned} \epsilon_n - v(x) &= 0 \\ \epsilon_n - 4(x-1)(x-2) &= 0 \\ -4x^2 + 12x - 8 + \epsilon_n &= 0 \end{aligned}$$

The roots of a quadratic are given by

$$x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$\begin{aligned} x_\pm &= \frac{-12 \pm \sqrt{144 + 16(\epsilon_n - 8)}}{-8} \\ x_\pm &= \frac{-3 \pm \sqrt{\epsilon_n + 1}}{-2} \end{aligned}$$

Therefore the roots are given by:

$$x_- = x_{in}(\epsilon_n) = \frac{3 - \sqrt{\epsilon_n + 1}}{2}$$

$$x_+ = x_{out}(\epsilon_n) = \frac{3 + \sqrt{\epsilon_n + 1}}{2}$$

With the roots of the $\epsilon_n - v(x) = 0$, know available, equation 9 can be solved analytically. Alternatively the a convenience equation can be used to solve equation 9.

$$\int_{x_-}^{x_+} \sqrt{ax^2 + bx + c}\, dx = \frac{(4ac - b^2)\pi}{8a\sqrt{-a}}$$

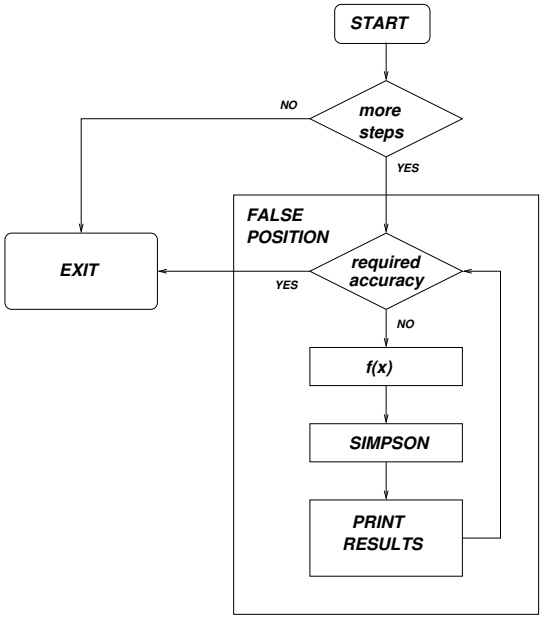


Fig. 1. Flow chart of quadratic integration algorithm

$$\begin{aligned} s(\epsilon_n) = \gamma \int_{x_{in}}^{x_{out}} [\epsilon_n - v(x)]^{1/2} dx &= \left(n + \frac{1}{2}\right)\pi \\ \gamma \int_{x_{in}}^{x_{out}} \sqrt{-4x^2 + 12x - 8 + \epsilon_n}\, dx &= \left(n + \frac{1}{2}\right)\pi \\ \gamma\left[\frac{4(-4)(\epsilon_n - 8) - 144}{-64}\right] &= \left(n + \frac{1}{2}\right)\pi \\ \gamma\left[\frac{\epsilon_n + 1}{4}\right] &= \left(n + \frac{1}{2}\right) \\ \epsilon_n &= \frac{4\left(n + \frac{1}{2}\right)}{\gamma} - 1 \end{aligned}$$

Therefore the analytic solution for the energy ϵ_n is given by

$$\epsilon_n = \frac{4\left(n + \frac{1}{2}\right)}{\gamma} - 1$$

Using this information, a FORTRAN program was written, to solve equation 9 for the quadratic potential $v(x) = 4(x-1)(x-2)$. The source code for the program `quadratic.f` can be found in Appendix C. The program takes the quantum number n, and a value for the constant γ as input.

Figure 1 shows a very high level architectural design of the FORTRAN program `quadratic.f` shown in Appendix C.

D. Quadratic Potential Results

The table below shows the results of the `quad.f` program for the values of $n = 0, 1, 2, 3, 4$, and with $\gamma = 1$. Since the accuracy of this method was investigated for different step sizes and tolerance values in the previous section, the results here all have 10^6 divisions for Simpson's rule and a tolerance of 10^{-6} for the false position algorithm.

The column labelled ϵ_n is the analytic solution to equation 9 for $v(x)$ given by equation 10, whereas the column labelled ϵ_n^* is the solution given by the FORTRAN program `quadratic.f`.

To look at the numerical stability of the program, we vary the number of divisions N in the Simpson quadrature algorithm,

TABLE IV
VALUES OF ϵ_n FOR A QUADRATIC POTENTIAL WITH $\gamma = 1$.

n	ϵ_n	ϵ_n^*	$Error$
0	1	1.0000000008	.826897E-09
1	5	5.0000000025	.248070E-08
2	9	9.0000000041	.413422E-08
3	13	13.0000000058	.578873E-08
4	17	17.0000000074	.744295E-08

and the tolerance in the false position algorithm. Taking just one of the values of n, we have the following results for $n = 1$ and $\gamma = 1$, in the table below.

TABLE V
ϵ_n FOR $n = 1, \gamma = 1$, FOR VARYING TOLERANCES FOR QUADRATURE AND ROOT FINDING ALGORITHMS.

N	$Tolerance$	ϵ_n	$Error$
10	.1000000	5.0801607150	.801607E-01
100	.0100000	5.0024839897	.248399E-02
1000	.0010000	5.0000784582	.784582E-04
10000	.0001000	5.0000024808	.248084E-05
100000	.0000100	5.0000000785	.784505E-07
1000000	.0000010	5.0000000025	.248070E-08

E. Quadratic Potential Discussion

Looking at Table IV, we see good agreement with the values calculated analytically for ϵ_n and those calculated numerically, with an error of approximately 10^{-8} for all cases except for when the $n = 0$, when the error is approximately 10^{-9}. The technique can then be used solve the problem of finding the quantized energy levels with more complicated and more physically realistic potentials such as the Morse potential, which there exists no analytic solution and must therefore be solved numberically.

Looking at Table V, we see the results for varying tolerances for both the quadrature (Simpson's) algorithm, N, and the root finding algorithm (false position), $Tolerance$, for $n = 1$ and $\gamma = 1$. We see that the algorithms used result in numerically stable solutions by oberving that the error consistently decreases, as N is increased and as the $Tolerance$ is decreased.

F. Morse Potential Procedure

The procedure was to modify the program used to calculate ϵ_n from using a quadratic potential, to a potential that was closer in shape to what had been observed in experiments. This is the Morse potential given by

$$V_{Morse}(r) = V_0 \left[\left(1 - e^{-(r - r_{min})/a} \right)^2 - 1 \right] \qquad (10)$$

The Morse potential can be normalised ($x = r/a$ and $v(x) = V(r)/V_0$) to get

$$v(x) = \left(1 - e^{-(x - x_{min})} \right)^2 - 1 \qquad (11)$$

The next step is to analytically find the turning points $x_{in}(\epsilon_n)$ and $x_{out}(\epsilon_n)$ of the equation $\epsilon_n - v(x) = 0$.

$$\epsilon_n - \left[\left(1 - e^{-(x - x_{min})} \right)^2 - 1 \right] = 0$$

Let $z = e^{-(x - x_{min})}$

$$\begin{aligned} \epsilon_n - [(1 - z)^2 - 1] &= 0 \\ \epsilon_n - (z^2 - 2z) &= 0 \\ z^2 - 2z - \epsilon_n &= 0 \end{aligned}$$

Solve for z using

$$\begin{aligned} z_\pm &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ z_\pm &= \frac{2 \pm \sqrt{4 - 4(1)(-\epsilon_n)}}{2} \\ z_\pm &= 1 \pm \sqrt{1 + \epsilon_n} \end{aligned}$$

Now substitute z back in to get

$$\begin{aligned} e^{-(x_\pm - x_{min})} &= 1 \pm \sqrt{1 + \epsilon_n} \\ -(x_\pm - x_{min}) &= \ln\left(1 \pm \sqrt{1 + \epsilon_n}\right) \\ x_\pm &= x_{min} - \ln\left(1 \pm \sqrt{1 + \epsilon_n}\right) \end{aligned}$$

Therefore $x_{in}(\epsilon_n) = x_{min} - \ln(1 + \sqrt{1 - \epsilon_n})$ and $x_{out}(\epsilon_n) = x_{min} - \ln(1 + \sqrt{1 + \epsilon_n})$.

The following procedure was then followed:

- The `quadratic.f` program was modified to change the potential from quadratic to Morse and to add the new turning points calculated above.
- The source code for the new program `morse.f` can be found in Appendix D.
- For the case of $n = 0$, the program was run with a number of different values of a, until a value was found such that the value given for the energy $E_n = V_0\epsilon_n$ was very close to value of $E_0 = -4.477$ as given in Table 1.5 of Koonin (experimental result).
- Using the final value of a, the first four quantised energy levels were calculated with the program and compared with the experimental results for the H_2 spectrum.
- The choice of starting values for ϵ_n, is restricted by the turning points, $x_\pm = x_{min} - \ln\left(1 \pm \sqrt{1 + \epsilon_n}\right)$. We can see from this equation that $-2 < \epsilon_n \leq 1$.

G. Morse Potential Results

1) Determining the parameter **a***:* The table below shows the results of inputing different values of a into the `morse.f` program. We have the following values:

- $\gamma = 33.6567a$
- $r_{min} = 0.74166$ Å, $x_{min} = 0.74166a$ Å
- $V_0 = 4.747$ eV, $(E_n = V_0\epsilon_n)$
- $n = 0$
- $N = 1000$ (Number of divisions in Simpson's Rule)
- $T = 10_{-3}$ (Tolerance in false position method)

From experimental results looking at the spectrum of the Hydrogen molecule, we know that for the lowest energy state (when

TABLE VI
VALUES OF ϵ_0 FOR DIFFERENT VALUES OF THE PARAMETER a.

a	ϵ_0
0.50	-.9414586271
0.60	-.9510928378
0.55	-.9467075966
0.52	-.9436775546
0.51	-.9425895401
0.515	-.9431387536

$n = 0$), we have $E_0 = -4.477\text{eV}$. Therefore, for a given value of a, we are looking for $\epsilon_0 = -4.477/4.747 \simeq -0.943121971$.

We see from Table VI that we get closest to the value of ϵ_0 when $a = 0.515$.

Using this value of a, we can then run the program for the first 4 values of $n = 0, 1, 2, 3$ and compare to the experimental results. Since the experimental results are only available to 3 decimal places, it is meaningless to attempt to calculate solutions with more accuracy since we don't have more accurate experimental data to compare with. In all cases the calculations were done with 1000 divisions in Simpson's rule and a tolerance of 10^{-3} for the root finding false position algorithm.

The results of the running the program for the first four energy levels are shown in Table VII, where E_n^* is the numerical result calculated using the program and E_n is the experimental result from Table 1.5 of Koonin.

TABLE VII
FIRST FOUR QUANTIZED ENERGY LEVELS OF THE SPECTRUM OF THE H_2 MOLECULE.

n	ϵ_n	E_n^*	E_n	$Error$
0	-.9431387536	-4.4770796631	-4.477	.796631E-04
1	-.8344090650	-3.9609398314	-3.962	.106017E-02
2	-.7323362062	-3.4763999710	-3.475	.139997E-02
3	-.6369210780	-3.0234643571	-3.017	.646436E-02

H. Morse Potential Discussion

As can be seen from Table 7, the values of the energy levels of the Hydrogen molecule for quantum levels $n = 0, 1, 2, 3$, have a good agreement with the experimental results. The error was of magnitude 10_{-2} for $n = 1, 2, 3$ and of magnitude 10_{-4} for $n = 0$.

The agreement between calculation and experiment is good, especially considering the use of the old quantum theory in the numerical calculations.

More insight could have been gained into the accuracy of the numerical techniques used to do the calculations if the experimental data presented for the Hydrogen molecule spectrum was more accurate.

With all the data now available we can make a plot of the equation

$$V_{Morse}(r) = V_0 \left[\left(1 - e^{-(r - r_{min})/a} \right)^2 - 1 \right].$$

Using the values for γ, a, r_{min}, and V_0, shown in the previous section we can plot the Morse potential for the Hydrogren molecule in Figure 2

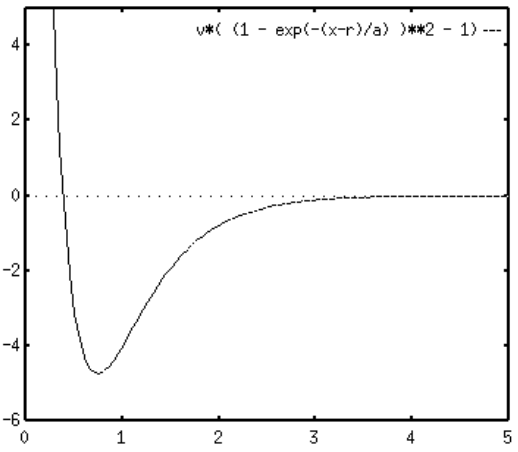


Fig. 2. Morse Potential for Hydrogen Molecule

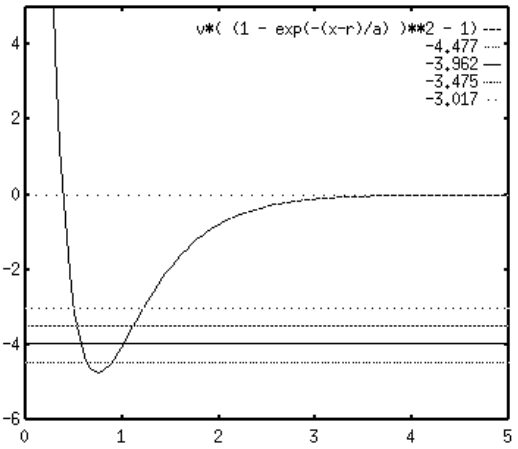


Fig. 3. First Four Quantized Energy Levels

Using the data for the first four quantized energy levels, we can plot these on top of the Morse potential plot, shown in Figure 3

APPENDIX
CODE LISTING: QUAD.F

```
c------------------------------------------------------------
c PROGRAM : QUADRATURE - TRAPEZOIDAL AND SIMPSON'S RULES
c
c SUBJECT   : 640-364 COMPUTATIONAL PHYSICS
c NAME      : MICHAEL PAPASIMEON
c DATE      : 30/07/1997
c
c PURPOSE   : This program does a numerical integration
c           : between 0 and 1 for the function ln(1+x)/x
c           : using both the Trapezoidal and Simpson's
c : rules for quadrature.
c------------------------------------------------------------

      program main
            call all_trapezoidals
            call all_simpsons
      end

c------------------------------------------------------------
c FUNCTION : f
c PURPOSE : Given a real number x, returns the result of
c : evaluating ln(1+x)/x.
c : Care is taken when x = 0, since this causes
c : a division by zero error, and hence a value
c : of 1 is returned in this case.
c------------------------------------------------------------

      double precision function f(x)
      implicit none
      real*8 x
      if ( x .eq. 0.0d0 ) then
            f = 1.0
      else
            f = dlog(1+x)/x
      endif
      return
      end

c------------------------------------------------------------
c SUBROUTINE : all_trapezoidals
c PURPOSE : Calls the trapezoidal subroutine 6 each time
c : with an increase in the number of divisions
c : (increased by a factor of 10). The result
c : is a table of the result of the quadrature
c : including an output of the percentage error.
c------------------------------------------------------------

      subroutine all_trapezoidals
      implicit none
      integer*4 n, i
      real*8 actual, pi, a, b

      a = 0.0d0
      b = 1.0d0
      pi = 4*atan(1.0d0)
      actual = (pi**2)/dfloat(12)

      write(*,*) '------------------------------------------------'
      write(*,*) '                TRAPEZOIDAL RULE'
      write(*,*) '------------------------------------------------'
      write(*,*) 'Lower Limit : a = ',a
      write(*,*) 'Upper Limit : b = ',b
      write(*,*) 'Actual Result = ',actual
      write(*,*) '------------------------------------------------'
      write(*,*) '        n              h        Result         Error'
```

```
      write(*,*) '-------------------------------------------------'

      n = 10
      do 10 i = 0, 5, +1
           call trapezoidal(n,a,b,actual)
           n = n*10
10 continue
      write(*,*) '-------------------------------------------------'

      end

c-------------------------------------------------------------
c SUBROUTINE : trapezoidal
c PURPOSE : Performs numerical quadrature using the
c : trapezoidal rule for the function f
c : between a and b and prints out the result
c : for the number of divisions given as a
c : parameter.
c-------------------------------------------------------------

      subroutine trapezoidal(n, a, b, answer)
      implicit none
      integer*4 n
      real*8 a, b, answer
        integer*4 i
      real*8 error, h, sum, x, f

        i = 0
      sum = 0.0d0
      x = 0.0d0
      h = (b-a)/dfloat(n)

        do 100 i = 1, (n-1), +1
               x = a + i*h
               sum = sum + 2.0d0*f(x)
100 continue
        sum = sum + f(a) + f(b)
      sum = (0.5d0)*h*sum
      error = abs(sum - answer)

      write(*,200) n, h, sum, error
200 format(i10, f14.10, f14.10, g14.6)

      end

c-------------------------------------------------------------
c SUBROUTINE : all_simpsons
c PURPOSE : Calls the simpson subroutine 6 each time
c : with an increase in the number of divisions
c : (increased by a factor of 10). The result
c : is a table of the result of the quadrature
c : including an output of the percentage error.
c-------------------------------------------------------------

      subroutine all_simpsons
      implicit none
      integer*4 n, i
      real*8 actual, pi, a, b

      a = 0.0d0
      b = 1.0d0
      pi = 4*atan(1.0d0)
      actual = (pi**2)/dfloat(12)

      write(*,*) '-------------------------------------------------'
      write(*,*) '               SIMPSONS RULE'
      write(*,*) '-------------------------------------------------'
      write(*,*) 'Lower Limit : a = ',a
      write(*,*) 'Upper Limit : b = ',b
```

```
      write(*,*) 'Actual Result = ',actual
      write(*,*) '------------------------------------------------'
      write(*,*) '        n             h         Result         Error'
      write(*,*) '------------------------------------------------'

      n = 10
      do 10 i = 0, 5, +1
            call simpson(n,a,b,actual)
            n = n*10
10 continue
      write(*,*) '------------------------------------------------'

      end

c-------------------------------------------------------------
c SUBROUTINE : simpson
c PURPOSE :
c-------------------------------------------------------------

      subroutine simpson(n, a, b, answer)
      implicit none
      integer*4 n, i, factor
      real*8 a, b, answer
      real*8 error, h, sum, x, f

      i = 0
      factor = 4
      sum = 0.0d0
      x = 0.0d0
      h = (b-a)/dfloat(n)

        do 300 i = 1, (n-1), +1
               x = a+i*h
               if (factor .eq. 2) then
                      sum = sum + 2*f(x)
                      factor = 4
               else
               sum = sum + 4*f(x)
                      factor = 2
               endif
300 continue
      sum = sum + f(a) + f(b)
      sum = (h*sum)/3.0d0
      error = dabs(sum - answer)

      write(*,400) n, h, sum, error
400 format(i10, f14.10, f14.10, g14.6)

      end
```

APPENDIX
CODE LISTING: ROOT.F

```
c-------------------------------------------------------------
c PROGRAM  : ROOT FINDING - FALSE POSITION METHOD
c
c SUBJECT : 640-364 COMPUTATIONAL PHYSICS
c NAME : MICHAEL PAPASIMEON
c DATE : 06/08/1997
c
c PURPOSE : This program does a numerical integration
c : between 0 and 1 for the function ln(1+x)/x
c : using both the Trapezoidal and Simpson's
c          : rules for quadrature.
c-------------------------------------------------------------

        program main
      implicit none
      integer*4 n, i
      real*8 result, false_position, tolx, answer, error

      answer = dsqrt(3.0d0)

      n = 10
      tolx = 1.0d0/dfloat(n)
      do 10 i = 0, 6, +1
            result = false_position(n, 1.0d0, tolx)
            error = dabs(result - answer)
            write(*,15)n, tolx, result, error
15          format(i10, f14.10, f14.10, g14.6)
            n = n*10
            tolx = 1.0d0/dfloat(n)
10    continue
        end

c-------------------------------------------------------------
c FUNCTION    : g
c PURPOSE     : returns t**2
c-------------------------------------------------------------

        double precision function g(t)
        implicit none
        real*8 t
        g = t**2
        return
        end

c-------------------------------------------------------------
c SUBROUTINE  : simpson
c PURPOSE     : calculates the integral between a and b
c             : for the function g(x)
c-------------------------------------------------------------

        double precision function simpson(n, a, b)
      implicit none
      integer*4 n, i, factor
      real*8 a, b
      real*8 h, sum, x, g

      i = 0
      factor = 4
      sum = 0.0d0
      x = 0.0d0
      h = (b-a)/dfloat(n)

      do 300 i = 1, (n-1), +1
            x = a+i*h
            if (factor .eq. 2) then
```

```
                    sum = sum + 2.0d0*g(x)
                    factor = 4
                else
                    sum = sum + 4.0d0*g(x)
                    factor = 2
                endif
300     continue
        sum = sum + g(a) + g(b)
        sum = (h*sum)/3.0d0

        simpson = sum
        return

        end

c-------------------------------------------------------------
c FUNCTION  : f
c PURPOSE   : calculates and returns the value of the function
c           : of the integral between a and b of the function
c           : g(x) determined by the simpson function minus
c           : c. When a = 0 and b = x, we have
c           : f = Integrate[t**2 dt,0,x] - x
c-------------------------------------------------------------

        double precision function f(n,a,b)
        implicit none
        integer*4 n
        real*8 a, b, simpson
        f = simpson(n,a,b) - b
        return
        end

c-------------------------------------------------------------
c FUNCTION  : false_position
c PURPOSE   : calculates and returns the root of the function
c           : f, after position x1, to a tolerance of tolx
c           : using the false position method. The parameter
c           : n determines the accuracy to which the function
c           : f can be calculated to.
c-------------------------------------------------------------

      double precision function false_position(n, x1, tolx)
      implicit none
      integer*4 n
      real*8 x1, tolx
      real*8 f
      real*8 x2, f1, f2, x3, f3, h, a

      a = 0.0d0
      h = 0.33

      x2 = x1 + h
      f1 = f(n,a,x1)
      f2 = f(n,a,x2)

      do while (f1*f2 .ge. 0.0d0)
          x2 = x2 + h
          f2 = f(n,a,x2)
      end do

      x3 = x2 - f2*(x2-x1)/(f2-f1)
      f3 = f(n,a,x3)

      do while (dabs(f3) .gt. tolx)
          if (f1*f3 .lt. 0.0d0) then
              x2 = x3
          else
              x1 = x3
          endif
```

```
            f1 = f(n,a,x1)
            f2 = f(n,a,x2)
            x3 = x2 - f2*(x2-x1)/(f2-f1)
            f3 = f(n,a,x3)
        end do

        false_position = x3
        return

    end
```

APPENDIX
CODE LISTING: QUADRATIC.F

```
c-------------------------------------------------------------
c PROGRAM : main
c PURPOSE :
c-------------------------------------------------------------

       program main
       implicit none
      integer i, n, step
      real*8 gamma, en, answer, error
       real*8 result, tol, false_position

      write(*,*)'Enter␣n,␣gamma,␣answer'
      read(*,*) n, gamma, answer

      en = -0.1d0
      step = 10
      tol = 1.0d0/step

      write(*,*) '-------------------------------------------'
      write(*,*) 'Molecular␣Vibrations␣:␣Quadratic␣Potential'
      write(*,*) '␣'
      write(*,*) 'step␣=␣step␣size␣for␣simpson␣integration'
      write(*,*) 'tol␣=␣tolerance␣for␣false␣position␣method'
      write(*,*) 'n␣=␣',n
      write(*,*) 'gamma␣=␣',gamma
      write(*,*) 'Energy␣=␣Quantised␣energy␣level␣En'
      write(*,*) '-------------------------------------------'
      write(*,*) '␣␣␣␣step␣␣␣␣␣tol␣␣␣␣␣␣␣␣Energy␣␣␣␣␣␣␣␣␣Error'
      write(*,*) '-------------------------------------------'

      do 100 i = 0, 5, +1
           result = false_position(n, gamma, step, en, tol)
           error = dabs(answer - result)
           write(*,20) step, tol, result, error
20 format(i8, f9.7, f14.10, e14.6)
           step = step*10
           tol = 1.0d0/step
100 continue

      write(*,*) '-------------------------------------------'

       end

c-------------------------------------------------------------
c FUNCTION : xin
c-------------------------------------------------------------

       double precision function xin(en)
       implicit none
       real*8 en

       xin = (3.0d0-dsqrt(en + 1.0d0))/2.0d0
       return

       end

c-------------------------------------------------------------
c FUNCTION : xout
c-------------------------------------------------------------

       double precision function xout(en)
       implicit none
       real*8 en

       xout = (3.0d0+dsqrt(en + 1.0d0))/2.0d0
```

```
        return

        end

c-------------------------------------------------------------
c FUNCTION : f
c-------------------------------------------------------------

        double precision function f(en, n, gamma, step)
        implicit none
        integer*4 n, step
        real*8 xin, xout, simpson
        real*8 en, gamma, xi, xo, pi

        pi = 4*atan(1.0d0)
        xi = xin(en)
        xo = xout(en)

        f = gamma*simpson(step,xi,xo,en) - pi*(dfloat(n) + 0.5d0)
        return

        end

c-------------------------------------------------------------
c FUNCTION : g
c PURPOSE : integrand
c-------------------------------------------------------------

        double precision function g(en, x)
        implicit none
        real*8 en, x, v, arg

      arg = en - v(x)

      if (dabs(arg) .lt. 1.0E-14) then
           g = 0.0d0
        else
           g = dsqrt(arg)
      endif
        return

        end

c-------------------------------------------------------------
c FUNCTION : v
c PURPOSE : potential
c-------------------------------------------------------------


        double precision function v(x)
        implicit none
        real*8 x

        v = 4*(x-1)*(x-2)
        return

        end

c-------------------------------------------------------------
c FUNCTION : simpson
c-------------------------------------------------------------

      double precision function simpson(step, a, b, en)
      implicit none
      integer*4 step, i, factor
      real*8 a, b, en
      real*8 h, sum, x, g

      i = 0
```

```
      factor = 4
      sum = 0.0d0
      x = 0.0d0
      h = (b-a)/dfloat(step)

      do 300 i = 1, (step-1), +1
            x = a+i*h
            if (factor .eq. 2) then
                  sum = sum + 2.0d0*g(en,x)
                  factor = 4
            else
                  sum = sum + 4.0d0*g(en,x)
                  factor = 2
            endif
300 continue
      sum = sum + g(en,a) + g(en,b)
      sum = (h*sum)/3.0d0

      simpson = sum
      return

      end

c-------------------------------------------------------------
c FUNCTION : false_position
c PURPOSE : calculates and returns the root of the function
c : f, after position x1, to a tolerance of tolx
c : using the false position method. The parameter
c : step determines the accuracy to which the function
c : f can be calculated to.
c-------------------------------------------------------------

      double precision function false_position(n,gamma,step,
     > start,tolx)
      implicit none
      integer*4 step, n
      real*8 start, tolx, gamma
      real*8 f
      real*8 x1, x2, f1, f2, x3, f3, h

      h = 0.3d0

      x1 = start
      x2 = x1 + h
      f1 = f(x1,n,gamma,step)
      f2 = f(x2,n,gamma,step)

      do while (f1*f2 .ge. 0.0d0)
            x2 = x2 + h
            f2 = f(x2,n,gamma,step)
90 format(f15.8, f15.8)
      end do

      x3 = x2 - f2*(x2-x1)/(f2-f1)
      f3 = f(x3,n,gamma,step)

      do while (dabs(f3) .gt. tolx)
            if (f1*f3 .lt. 0.0d0) then
                  x2 = x3
            else
                  x1 = x3
            endif
            f1 = f(x1,n,gamma,step)
            f2 = f(x2,n,gamma,step)
            x3 = x2 - f2*(x2-x1)/(f2-f1)
            f3 = f(x3,n,gamma,step)
      end do

      false_position = x3
```

```
204         return
205
206         end
```

APPENDIX
CODE LISTING: MORSE.F

```
c-------------------------------------------------------------
c PROGRAM : main
c-------------------------------------------------------------

      program main
        implicit none
      integer i, n, step
      real*8 gamma, en, answer, error, xmin, a
        real*8 result, tol, false_position

      write(*,*)'Enter n, a'
      read(*,*) n, a

      answer = -0.943121971d0
      gamma = 33.6567d0*a

      en = -1.0d0
      step = 10
      tol = 1.0d0/step
      xmin = 0.74166d0*a

      write(*,*) '-------------------------------------------'
      write(*,*) 'Molecular Vibrations : Quadratic Potential'
      write(*,*) ' '
      write(*,*) 'step = step size for simpson integration'
      write(*,*) 'tol = tolerance for false position method'
      write(*,*) 'n = ',n
      write(*,*) 'gamma = ',gamma
      write(*,*) 'Energy = Quantised energy level En'
      write(*,*) '-------------------------------------------'
      write(*,*) '    step      tol        Energy        Error'
      write(*,*) '-------------------------------------------'

      do 100 i = 0, 5, +1
           result = false_position(n, gamma, step, en, tol, xmin)
           error = dabs(answer - result)
           write(*,20) step, tol, result, error
20 format(i8, f9.7, f14.10, e14.6)
           step = step*10
           tol = 1.0d0/step
100 continue

      write(*,*) '-------------------------------------------'

        end

c-------------------------------------------------------------
c FUNCTION : xin
c-------------------------------------------------------------

        double precision function xin(en, xmin)
        implicit none
        real*8 en, xmin

        xin = xmin - dlog(1.0d0 + dsqrt(en + 1.0d0))
        return

        end

c-------------------------------------------------------------
c FUNCTION : xout
c-------------------------------------------------------------

        double precision function xout(en, xmin)
        implicit none
```

```
      real*8 en, xmin

c write(*,*)'[xout]:[en]␣',en
      xout = xmin - dlog(1.0d0 - dsqrt(en + 1.0d0))
      return

      end

c-----------------------------------------------------------------
c FUNCTION : f
c-----------------------------------------------------------------

      double precision function f(en, n, gamma, step, xmin)
      implicit none
      integer*4 n, step
      real*8 xin, xout, simpson
      real*8 en, xmin, gamma, xi, xo, pi

      pi = 4*atan(1.0d0)
      xi = xin(en, xmin)
      xo = xout(en, xmin)

      f = gamma*simpson(step,xi,xo,en,xmin) - pi*(dfloat(n) + 0.5d0)
      return

      end

c-----------------------------------------------------------------
c FUNCTION : g
c PURPOSE : integrand
c-----------------------------------------------------------------

      double precision function g(en, x, xmin)
      implicit none
      real*8 en, xmin, x, v, arg

     arg = en - v(x, xmin)

     if (dabs(arg) .lt. 1.0E-14) then
          g = 0.0d0
       else
          g = dsqrt(arg)
     endif
       return

       end

c-----------------------------------------------------------------
c FUNCTION : v
c PURPOSE : potential
c-----------------------------------------------------------------


      double precision function v(x, xmin)
      implicit none
      real*8 x, xmin

      v = (1 - dexp(-(x - xmin)))**2.0d0 - 1.0d0
      return

      end

c-----------------------------------------------------------------
c FUNCTION : simpson
c-----------------------------------------------------------------

     double precision function simpson(step, a, b, en, xmin)
     implicit none
     integer*4 step, i, factor
```

```
      real*8 a, b, en, xmin
      real*8 h, sum, x, g

      i = 0
      factor = 4
      sum = 0.0d0
      x = 0.0d0
      h = (b-a)/dfloat(step)

      do 300 i = 1, (step-1), +1
            x = a+i*h
            if (factor .eq. 2) then
                  sum = sum + 2.0d0*g(en,x,xmin)
                  factor = 4
            else
                  sum = sum + 4.0d0*g(en,x,xmin)
                  factor = 2
            endif
300 continue
      sum = sum + g(en,a,xmin) + g(en,b,xmin)
      sum = (h*sum)/3.0d0

      simpson = sum
      return

      end

c-------------------------------------------------------------
c FUNCTION : false_position
c PURPOSE : calculates and returns the root of the function
c : f, after position x1, to a tolerance of tolx
c : using the false position method. The parameter
c : step determines the accuracy to which the function
c : f can be calculated to.
c-------------------------------------------------------------

      double precision function false_position(n,gamma,step,
     > start,tolx,xmin)
      implicit none
      integer*4 step, n
      real*8 start, tolx, gamma
      real*8 f, xmin
      real*8 x1, x2, f1, f2, x3, f3, h

      h = 0.00001d0

      x1 = start
      x2 = x1 + h
      f1 = f(x1,n,gamma,step,xmin)
      f2 = f(x2,n,gamma,step,xmin)

      do while (f1*f2 .ge. 0.0d0)
            x2 = x2 + h
            f2 = f(x2,n,gamma,step,xmin)
      end do

      x3 = x2 - f2*(x2-x1)/(f2-f1)
      f3 = f(x3,n,gamma,step,xmin)

      do while (dabs(f3) .gt. tolx)
            if (f1*f3 .lt. 0.0d0) then
                  x2 = x3
            else
                  x1 = x3
            endif
            f1 = f(x1,n,gamma,step,xmin)
            f2 = f(x2,n,gamma,step,xmin)
            x3 = x2 - f2*(x2-x1)/(f2-f1)
            f3 = f(x3,n,gamma,step,xmin)
```

```
204         end do
205
206         false_position = x3
207         return
208
209         end
```