Computational Physics with Maxima or R: Example 3, Time Independent Schroedinger's Equation in 1D *

Edwin (Ted) Woollett

August 24, 2015

Contents

1 Transforming Schroedinger's Time Independent Equation to Dimensionless Form **3**

2 The Finite Rectangular Potential Well: Energy Levels and Wave Functions **4**
2.1 Finite Well Analytic Solution . . . 5
2.1.1 Analytic Energies and Wave Functions using Maxima . . . 6
2.1.2 Analytic Energies and Wave Functions using R . . . 12
2.2 Numerical Runge Kutta Finite Well Solution . . . 19
2.2.1 Numerical Energies and Wave Functions using Maxima . . . 19
2.2.2 Numerical Energies and Wave Functions using R . . . 32

3 The Numerov Integration Method **45**
3.1 Classical Simple Harmonic Oscillator Test Case . . . 45
3.1.1 Classical SHO Numerov Method Using Maxima . . . 46
3.1.2 Classical SHO Numerov Method Using R . . . 48

4 The Lennard Jones 6-12 Potential Well: Energy Levels and Wave Functions **50**
4.1 The Numerov Method Using Maxima . . . 50
4.2 The Numerov Method Using R . . . 64

*The code examples use **R ver. 3.0.2** and **Maxima ver. 5.31** using **Windows 7**. This is a live document which will be updated when needed. Check **http://www.csulb.edu/~woollett/** for the latest version of these notes. Send comments and suggestions for improvements to **woollett@charter.net**

Code files which accompany Example 3 are
1. FW.mac finite rectangular well
2. FW.R
3. LJ6-12.mac Lennard Jones 6/12 potential
4. LJ6-12.R

The code files from Chapter 3, cp3.mac and cp3.R, are
also needed to run the examples.

A very useful reference is the new textbook:
Computational Physics: A Practical Introduction to Computational Physics
and Scientific Computing, Athens, 2014,
by Konstantinos Anagnostopoulos.
Free copies in various formats are available on the webpage:
http://www.physics.ntua.gr/~konstant/ComputationalPhysics/

The author's webpage is:
http://www.physics.ntua.gr/~konstant

Feedback from readers is the best way for this series of notes to become more helpful to users of **R** and **Maxima**. *All* comments and suggestions for improvements will be appreciated and carefully considered.

1 Transforming Schroedinger's Time Independent Equation to Dimensionless Form

We assume a scalar particle (spin zero). Using ordinary units, the wave function $\psi(x)$, for one dimensional problems, is a solution of the eigenvalue equation

$$\frac{\hbar^2}{2\,m}\,\frac{d^2}{dx^2}\,\psi(x) + (E - V(x))\;\psi(x) = 0 \tag{1.1}$$

in which E is the energy of the particle in ergs, $V(x)$ is the potential energy in ergs, and x is a length in centimeters. The coordinate space wave function $\psi(x)$ is assumed to be normalized according to (integrating over the region in which $\psi(x)$ is nonzero)

$$\int |\psi(x)|^2\,dx = 1, \tag{1.2}$$

since $dx\,|\psi(x)|^2$ is (in the absence of any other information) the probability that the particle described by $\psi(x)$ will be found in the interval $(x, x + dx)$, and the sum of the probabilities of mutually exclusive outcomes must add up to unity.

For Schroedinger's time independent equation in 1D, we can usually take $\psi(x)$ to be a real function of x, and from (1.2) we see that the dimension of $\psi(x)$ is $1/\sqrt{cm.}$. Quoting Landau and Lifshitz, Quantum Mechanics, third revised edition reprinted 2003, p. 55,

> Schroedinger's equation for the wave functions ψ of stationary states is real, as are the conditions imposed on its solution. Hence its solutions can always be taken as real (These assertions are not valid for systems in a magnetic field; it is assumed that the potential energy does not depend explicitly on the time: the system is either closed or in a constant (non-magnetic) field).
>
> The eigenfunctions of non-degenerate values of the energy are automatically real, apart from an unimportant phase factor. ... The wave functions corresponding to the same degenerate energy level need not be real, but by a suitable choice of linear combinations of them we can always obtain a set of real functions.

We assume there exists a quantity L with the dimensions of $cm.$, chosen for convenience, in terms of which we can define a dimensionless coordinate $\tilde{x}$

$$\tilde{x} = x/L. \tag{1.3}$$

We also assume there exists an energy V_0 (with dimensions of $ergs$) associated with the potential energy in terms of which we can define a dimensionless potential energy $\tilde{V}(\tilde{x})$

$$\tilde{V}(\tilde{x}) = \frac{V(x)}{V_0} \tag{1.4}$$

and a dimensionless energy $\tilde{E}$

$$\tilde{E} = \frac{E}{V_0}. \tag{1.5}$$

We also define a dimensionless coordinate space wave function $\tilde{\psi}(\tilde{x})$

$$\tilde{\psi}(\tilde{x}) = \sqrt{L}\,\psi(x), \tag{1.6}$$

in terms of which we have the transformed Schroedinger's equation

$$\frac{d^2\,\tilde{\psi}(\tilde{x})}{d\,\tilde{x}^2} + \gamma^2\,\left(\tilde{E} - \tilde{V}(\tilde{x})\right)\,\tilde{\psi}(\tilde{x}) = 0, \tag{1.7}$$

where

$$\gamma = \sqrt{\frac{2\,m\,L^2\,V_0}{\hbar^2}}, \tag{1.8}$$

and we have a normalization condition in terms of $\tilde{x}$ and $\tilde{\psi}(\tilde{x})$:

$$\int \tilde{\psi}(\tilde{x})^2 \, d\,\tilde{x} = 1. \tag{1.9}$$

The "uncertainty relation" is a condition on the product of the uncertainty Δx in the position of the particle, and the uncertainty Δp_x in the simultaneous x component of the mechanical momentum of the particle:

$$\Delta x \, \Delta p_x \geq \frac{\hbar}{2} \tag{1.10}$$

in which, for a property A,

$$\Delta A = \sqrt{(\Delta A)^2} \tag{1.11}$$

and

$$(\Delta A)^2 = \left\langle (A - < A >)^2 \right\rangle = \left\langle A^2 \right\rangle - < A >^2 . \tag{1.12}$$

We define a dimensionless x component of momentum $\tilde{p}_x$

$$\tilde{p}_x = \frac{L}{\hbar} \, p_x, \tag{1.13}$$

in terms of which (1.10) becomes

$$\Delta\tilde{x} \, \Delta\tilde{p}_x \geq \frac{1}{2}. \tag{1.14}$$

The expectation value of p_x^n

$$\langle p_x^n \rangle = \int \psi^*(x) \, \left(\frac{\hbar}{i} \, \frac{d}{dx} \right)^n \, \psi(x) \, dx \tag{1.15}$$

becomes

$$\langle \tilde{p}_x^n \rangle = \int \tilde{\psi}^*(\tilde{x}) \, \left(\frac{1}{i} \, \frac{d}{d\tilde{x}} \right)^n \, \tilde{\psi}(\tilde{x}) \, d\tilde{x}. \tag{1.16}$$

When we can assume $\psi(x)$ is real (most of the time), $< p_x >$ is either zero or a pure imaginary number, since

$$< p_x >= \int \psi(x) \, \frac{\hbar}{i} \, \frac{d\psi(x)}{dx} \, dx. \tag{1.17}$$

But $< p_x >$ must be real, and thus we conclude it must also be zero. For consistency, this implies that

$$\int \psi(x) \, \frac{d\psi(x)}{dx} \, dx = 0. \tag{1.18}$$

2 The Finite Rectangular Potential Well: Energy Levels and Wave Functions

We assume a finite well such that $V(x) = V_0$ for $x \leq 0$ and also for $x \geq L > 0$, while $V(x) = 0$ for $0 < x < L$. Transforming to dimensionless units as described in the previous section, we then have $\tilde{V}(\tilde{x}) = 1$ for $\tilde{x} \leq 0$ and for $\tilde{x} \geq 1$, and $\tilde{V}(\tilde{x}) = 0$ for $0 < \tilde{x} < 1$. In the following, we drop the tildes, with $\tilde{x} \to x$, $\tilde{V}(\tilde{x}) \to V(x)$, $\tilde{E} \to E$, $\tilde{p}_x \to p_x$, and $\tilde{\psi}(\tilde{x}) \to y(x)$, so we are seeking energy eigenvalues E and the associated energy eigenfunctions $y(x)$ such that $y(x)$ is a real continuous function satisfying the equation

$$\frac{d^2}{dx^2} \, y(x) + \gamma^2 \, \left(E - V(x) \right) \, y(x) = 0 \tag{2.1}$$

and such that $y(x) \to 0$ as $x \to \pm\infty$ in such a way that we can satisfy the normalization condition

$$\int_{-\infty}^{\infty} y(x)^2 \, dx = 1, \tag{2.2}$$

and we also satisfy the basic uncertainty relation $\Delta x \, \Delta p_x \geq \frac{1}{2}$.

Using **R** graphics methods, we can make a simple plot of the dimensionless rectangular well (which now looks like a finite square well), and add a hypothetical energy level (in red).

```
> setwd("c:/k3")
> source("cp3.R")
> plot(0, type="n",xlim=c(-2,2),ylim=c(-2,2),xlab="x",ylab="V(x)")
> lines( c(-2,0), c(1,1), lwd = 3,col = "blue" )
> lines( c(0,0), c(0,1), lwd = 3, col = "blue" )
> lines( c(0,1), c(0,0), lwd = 3, col = "blue" )
> lines( c(1,1), c(0,1), lwd = 3, col = "blue" )
> lines( c(1,2), c(1,1), lwd = 3, col = "blue" )
> mygrid()
> lines( c(0,1), c(0.3,0.3), lwd = 3, col = "red" )
```

which produces the plot

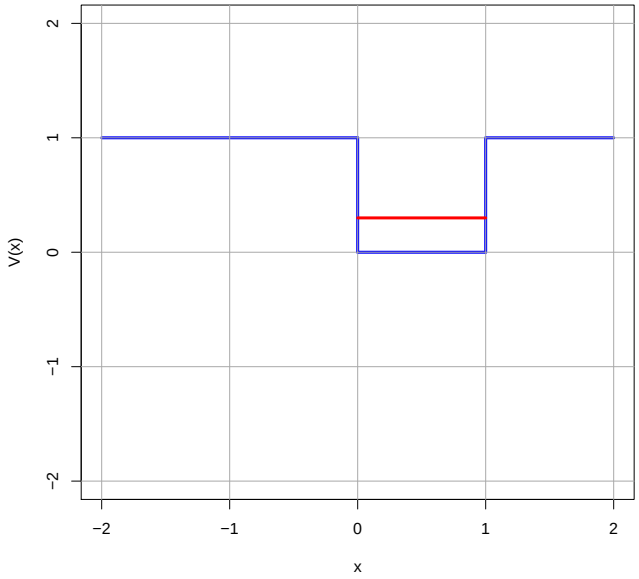

Figure 1: Dimensionless Finite Well

2.1 Finite Well Analytic Solution

We first seek an analytic solution of the finite well problem. In the regions in which $V(x) = 1$, the solutions of (2.1) which vanish for large values of $|x|$ are

$$y(x) = B_1\, e^{k_1\, x}, \quad x \le 0, \tag{2.3}$$

and

$$y(x) = A_2\, e^{-k_1\, x}, \quad x \ge 1, \tag{2.4}$$

in which

$$k_1 = \gamma\, \sqrt{1 - E}, \tag{2.5}$$

and B_1 and A_2 are constants.

The general solution in the region in which $V(x) = 0$ can be written in the form

$$y(x) = A\, \sin(k\, x + \delta), \quad 0 \le x \le 1, \tag{2.6}$$

in which

$$k = \gamma\, \sqrt{E}. \tag{2.7}$$

This last equation can be solved for E:

$$E = \frac{k^2}{\gamma^2}. \tag{2.8}$$

We can also then express k_1 in terms of k:

$$k_1 = \sqrt{\gamma^2 - k^2}. \tag{2.9}$$

The required continuity of both y and y' implies the continuity of the ratio y'/y. We have

$$\frac{y'(x)}{y(x)} = k_1, \quad x \leq 0, \tag{2.10}$$

and

$$\frac{y'(x)}{y(x)} = -k_1, \quad x \geq 1, \tag{2.11}$$

and

$$\frac{y'(x)}{y(x)} = k\, \cot(k\, x + \delta), \quad 0 \leq x \leq 1. \tag{2.12}$$

Then the continuity of y'/y at $x = 0$ implies

$$\tan(\delta) = \frac{k}{k_1} = \frac{k}{\sqrt{\gamma^2 - k^2}}. \tag{2.13}$$

And the continuity of y'/y at $x = 1$ implies

$$\tan(k + \delta) = -\frac{k}{k_1} = -\tan(\delta). \tag{2.14}$$

We expand the left hand side of this last equation, using the identity

$$\tan(A + B) = \frac{\tan(A) + \tan(B)}{1 - \tan(A)\, \tan(B)}, \tag{2.15}$$

and use again (2.13) to get

$$k = \frac{(2\, k^2 - \gamma^2)}{2\, \sqrt{\gamma^2 - k^2}}\, \tan(k). \tag{2.16}$$

This equation will involve real numbers provided we have

$$0 < k < \gamma. \tag{2.17}$$

We can search for values of k which satisfy both (2.16) and (2.17), which will then imply corresponding energy eigenvalues using (2.8). A graphical search can be achieved by plotting the left and right hand sides of (2.16) on the same plot and looking for intersections.

2.1.1 Analytic Energies and Wave Functions using Maxima

Our function **`kroot_plot(kkmin,kkmax,ymn,ymx)`** (available for use once the code file **`FW.mac`** is loaded) is designed to partially automate such a graphical search.

```
Frhs(k) := (float( (2*k^2 - gam2)*tan(k)/2/sqrt(gam2 - k^2)) )$

kroot_plot(kkmin,kkmax, ymn, ymx) :=
         ( plot2d([kk, Frhs(kk)], [kk, kkmin, kkmax],
                  [y, ymn, ymx],  [style, [lines,2]], [xlabel, "k"],
               [legend, false], [ylabel, ""],
                [gnuplot_preamble," set grid"]))$
```

After loading **FW.mac**, **gam2** is a global parameter which stands for γ^2, and in this code file we use $\gamma = 50$.

```
(%i1) load(cp3);
(%o1) "c:/k3/cp3.mac"
(%i2) load(FW);
  gam =  50     gam2 =  2500
 h =  0.01 ,   xdecay =  0.5 ,  ypleft =  1.0E-8   ypright =  1.0E-8
(%o2) "c:/k3/FW.mac"
(%i3) kroot_plot(1,49,0,60)$
plot2d: some values were clipped.
```

which produces the plot

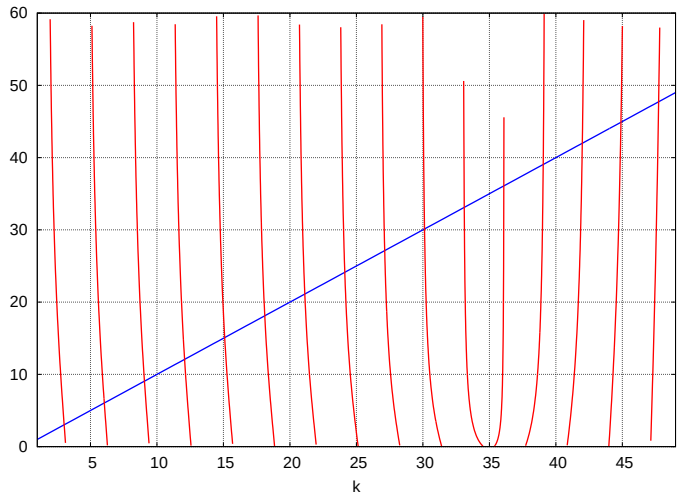

Figure 2: Graphical Search for k Roots

Placing the cursor over the intersections of the curves, we see that k roots occur approximately at values
k = 3.01, 6.03, 9.06,....

Fa(k) is a function, also defined in **FW.mac**, which is zero at these special values of **k**. Also defined is **ktoE(k)** which converts **k** to **E**.

```
Fa(k) := (float(k - (2*k^2 - gam2)*tan(k)/2/sqrt(gam2 - k^2)) )$

ktoE (kv) := (kv^2/gam2)$
```

This function **Fa(k)** (of one variable) can then be directly used with the core Maxima function **find_root** to find the ground state energy **E0**.

```
(%i4) Fa(2.9);
(%o4) -3.2290071
(%i5) Fa(3.1);
(%o5) 2.0655923
(%i6) k0 : find_root(Fa,2.9,3.1);
(%o6) 3.0206914
(%i7) E0 : ktoE(k0);
(%o7) 0.00364983
(%i8) plot_analytic(E0)$
  E  =   0.00364983
 x_mean =  0.5
 delx =  0.18802
 ydy_sum =  -1.78676518E-16
 delp =  2.9619274   delx*delp =  0.556902
 number of nodes =  0
```

which produces the plot of the normalized analytic ground state wave function with zero nodes.

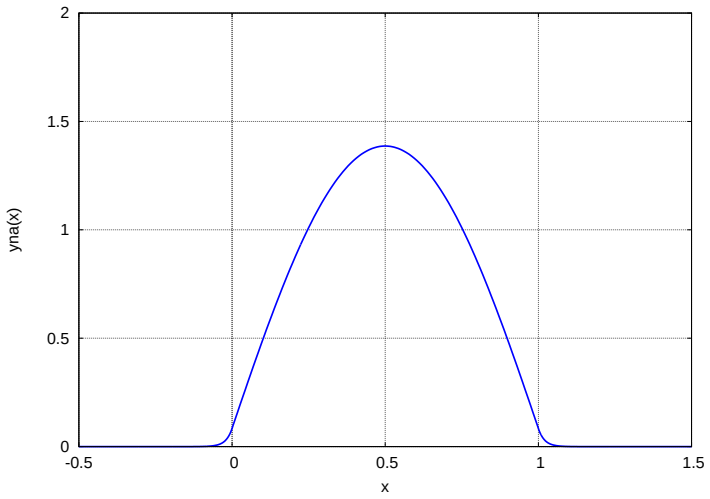

Figure 3: Analytic Normalized Ground State Wave Function

The function **plot_analytic(E)** (see below) calls **analytic_wf(E)** (see code file **FW.mac**), and the latter file calculates **delx**, **delp**, **x_mean**, and **ydy_sum**. The calculated value of **ydy_sum** corresponds to the integral (1.18) which should be zero.

The function **find_root** also appears to find a spurious root at or near $\pi/2$ which is not a physical solution. The function

$$F_a(k) = k - \frac{(2\,k^2 - \gamma^2)}{2\,\sqrt{\gamma^2 - k^2}}\,\tan(k). \tag{2.18}$$

is not a continuous function at $k = n\,\frac{\pi}{2}, \quad n = 1, 3, 5, \ldots$ where $\tan(k)$ is not continuous, and **find_root** cannot be trusted at such points. For example,

```
(%i9) Fa(1.55);
(%o9) 1201.7787
(%i10) Fa(1.59);
(%o10) -1298.109
(%i11) k0s : find_root(Fa,1.55,1.59);
(%o11) 1.5707963
(%i12) Fa(k0s);
(%o12) 4.07689764E+17
(%i13) E0s : ktoE(k0s);
(%o13) 9.8696044E-4
(%i14) plot_analytic(E0s)$
  E  =   9.8696044E-4
 x_mean =  0.700405
 delx =  0.212493
 ydy_sum =  2.22044605E-16
 delp =  6.6877608*(-1)^0.5   delx*delp =  1.4210999*(-1)^0.5
 number of nodes =  0
```

which produces the plot of a discontinuous zero node wave function:

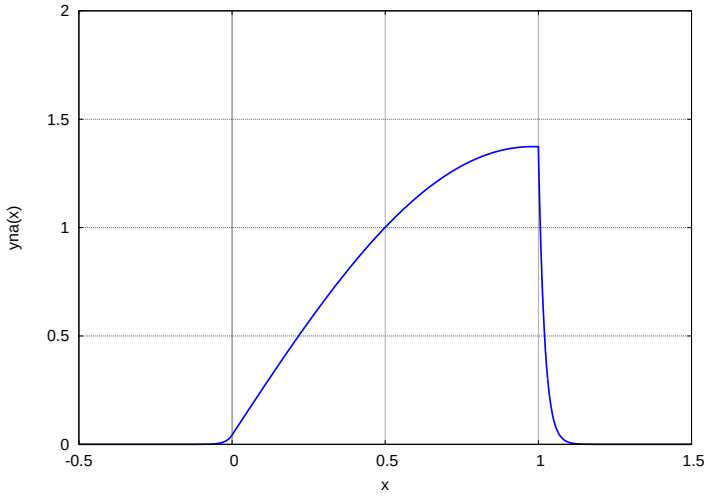

Figure 4: Discontinuous Spurious Root Wave Function

Note also the calculation produced a purely imaginary value for **delp**. Note also the large value of **Fa(k0s)**, which should be a very small number close to a physical root.

The function **analytic_wf(E)** creates global analytic normalized wave functions **yn1(x)**, **yn2(x)**, **yn0(x)**, and **yna(x)** (and also computes and prints analytic values of Δx, represented by **delx**, and Δp, represented by **delp**). **yn1(x)** is the wave function for $x < 0$, **yn2(x)** is the wave function for $x > 1$, **yn0(x)** is the wave function for $0 \le x \le 1$. **yna(x)** is the wave function for all x, usable by **plot2d**, but not by **integrate**, created by the line

```
yna(x) :=
  ( if x < 0 then yn1(x)
     else if x > 1 then yn2(x)
     else  yn0(x) ),
```

Ignoring normalization, one can write a continuous $y_u(x)$ as

$$\begin{aligned} y_u(x) &= \sin(\delta)\, e^{k_1\, x}, \quad x \le 0, \quad k_1 = \gamma\sqrt{1-E} \\ &= \sin(k\, x + \delta), \quad 0 \le x \le 1, \quad k = \gamma\sqrt{E} \\ &= \sin(k + \delta)\, e^{k_1}\, e^{-k_1\, x}, \quad x \ge 1. \end{aligned}$$

Normalization then requires calculating

$$D = \int_{-\infty}^{\infty} y_u^2(x)\, dx. \tag{2.19}$$

and a normalized wave function $y_n(x)$ is then

$$y_n(x) = \frac{y_u(x)}{\sqrt{D}}. \tag{2.20}$$

The function **plot_analytic(E)** used above calls **analytic_wf(E)** and **nodes_analytic(ddx)**, prints out the number of nodes, and makes a plot of **yna(x)**.

```
plot_analytic(E) :=
block ( [ddx : 0.001, xvL, ynL, ymn, ymx,
              xmn: -0.25, xmx : 1.25,  numer], numer:true,
     analytic_wf(E),
     xvL : makelist(x,x,0,1,ddx),
     ynL : map(yn0, xvL),
     ymn : floor( lmin(ynL) ),
```

```
     ymx : 1 + floor( lmax (ynL) ),
     print(" number of nodes = ", nodes_analytic(ddx) ),
     plot2d(yna(x), [x,xmn,xmx], [ylabel, "yna(x)"],[y,ymn,ymx],
           [style,[lines,2]],[gnuplot_preamble,"set grid"]))$
```

The function **nodes_analytic(dx)** counts the number of nodes implied by the function **yn0(x)** created by **analytic_wf(E)**. The count is naturally restricted to the region $0 \leq x \leq 1$ and the argument **dx** is the step size used.

```
nodes_analytic ( dx ) :=
block( [num:0, xv:0, xnew, f1, f2, numer], numer:true,
    do (
         f1 : yn0(xv),
         xnew : xv + dx,
         if xnew > 1  then return(),
         f2 :  yn0(xv + dx),
         if f1*f2 < 0 then num : num + 1,
         xv : xnew),
        num)$
```

A function **levels_analytic(kmax)** returns a list of analytic eigen energies related to k via $k = \gamma\sqrt{E}$. The corresponding eigenvalues of k are separated by roughly $dk = 3$ and lie in the middle of the intervals $[n_1 \frac{\pi}{2}, n_2 \frac{\pi}{2}]$, where n_1 and n_2 are adjacent odd integers with $n_2 > n_1$. The ground state (zero node soln) corresponds to $k = 3.02, n_1 = 1, n_2 = 3$.

```
levels_analytic(kmax) :=
block( [ka,kb,kv,level:0, nmax, Elist : [],  numer], numer:true,
     nmax : ceiling(2*kmax/%pi),
     print("    nodes      E    "),
     print( "  "),
     /* make nmax an odd integer */
     if evenp (nmax) then nmax : nmax + 1,
     /* analytic kv using n*pi/2 + 0.1 with n odd  */
     for j:1 step 2  thru nmax do (
        [ka, kb] : bracket_basic( Fa,   j*%pi/2 + 0.1,  0.01, 0.005),
        kv : find_root( Fa, ka, kb),
        Ev : ktoE(kv),
        Elist : cons(Ev, Elist),
        print( "   ", level,  "   ", Ev ),
        level : level + 1),
     reverse(Elist) )$
```

Here is an example of the use of **levels_analytic**:

```
(%i15) levels_analytic(20);
    nodes      E

    0     0.00364983
    1     0.0145973
    2     0.032836
    3     0.0583552
    4     0.0911389
    5     0.131165
    6     0.178405
(%o15) [0.00364983,0.0145973,0.032836,0.0583552,0.0911389,0.131165,0.178405]
(%i16) E4 : %[5];
(%o16) 0.0911389
(%i17) plot_analytic(E4)$
  E  =   0.0911389
 x_mean =  0.5
 delx =  0.297122
 ydy_sum =  -8.32667268E-17
 delp =  14.787571   delx*delp =  4.3937127
 number of nodes =  4
```

which produces the plot

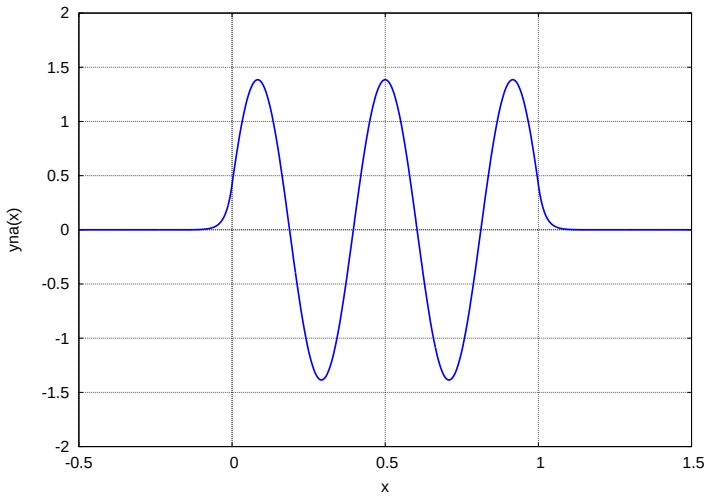

Figure 5: Analytic Four Node Wave Function $E = 0.0911389$

The function **levels_analytic** calls **bracket_basic(func,x,dx,xacc)** which looks for a sign change in **func**, starting with **x**, and increasing **x** by **dx** each step. If a sign change is found, then we back up to the previous **x** and search with new **dx** value one half of the previous value.

```
bracket_basic(func,xx,dxx,xacc) :=
block([ x:xx, dx:dxx,x1,x2,it:0,itmax:1000],
    do (
          it : it + 1,
          if it > itmax then (
          print(" can't find change in sign "),
              return([0, 0 ])),
          x1 : x,
          x2 : x + dx,
          if debug then print(" it = ",it," x1 = ",x1," x2 = ",x2," dx = ", dx),

          if func(x1)*func(x2) < 0 then (
               if abs(dx) < xacc then return([x1,x2]),
                   x : x - dx,
                   dx : dx/2)
          else x : x2))$
```

Here is an example using **bracket_basic** with **func = sin**:

```
(%i18) [xa,xb] : bracket_basic(sin,3,0.01,0.001);
(%o18) [3.14125,3.141875]
(%i19) xv : find_root(sin,xa,xb);
(%o19) 3.1415927
```

Here is an example of locating the zero node ground state value of **k** for the finite potential well problem using **func = Fa** (see (2.18)).

```
(%i20) [ka,kb] : bracket_basic(Fa,1.6,0.1,0.05);
(%o20) [3.0,3.025]
(%i21) kv : find_root(Fa,ka,kb);
(%o21) 3.0206914
```

2.1.2 Analytic Energies and Wave Functions using R

We can search for values of k which satisfy both (2.16) and (2.17), which will then imply corresponding energy eigenvalues using (2.8). A graphical search can be achieved by plotting the left and right hand sides of (2.16) on the same plot and looking for intersections. Our function **`kroot_plot(kkmin,kkmax,ymn,ymx)`** is designed to partially automate such a graphical search.

```
Frhs  =  function (k) {  (2*k^2 - gam2)*tan(k)/2/sqrt(gam2 - k^2) }

kroot_plot  =   function (kkmin,kkmax, ymn, ymx) {
        curve (Frhs,kkmin,kkmax, n=200, col = "red",lwd = 3,ylim = c(ymn,ymx),
                                  xlab = "k", ylab = "")
        lines( c(kkmin,kkmax), c(kkmin,kkmax),col = "blue",lwd = 3 ) }
```

After loading **`FW.R`**, **`gam2`** is a global parameter which stands for γ^2, and in this code file we use $\gamma = 50$.

```
> source("cp3.R")
> source("FW.R")
  gam =  50     gam2 =  2500
 h =  0.01 ,   xdecay =  0.5 ,  ypleft =  1e-08   ypright =  1e-08
> kroot_plot(1,49,0,60)
> mygrid()
```

which produces the plot

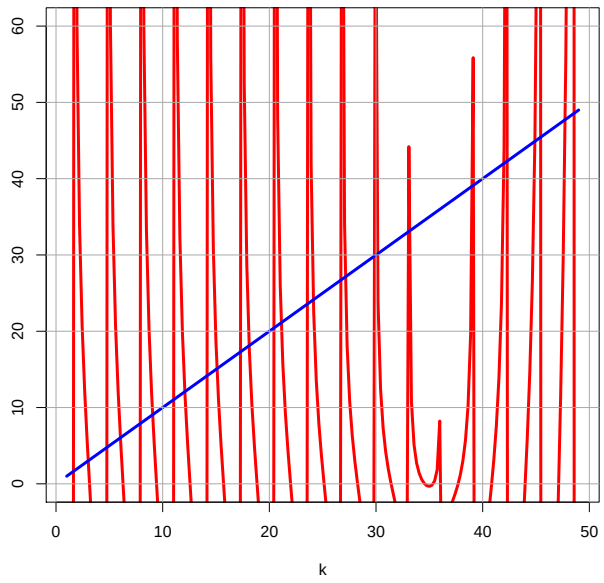

Figure 6: Graphical Search for k Roots

Let's zoom in on the beginning of this plot and add a custom grid.

```
>  kroot_plot(1,10,0,10)
> abline( v = seq(1,10,by = 1),h = seq(0,10,by = 1) )
```

which produces the plot

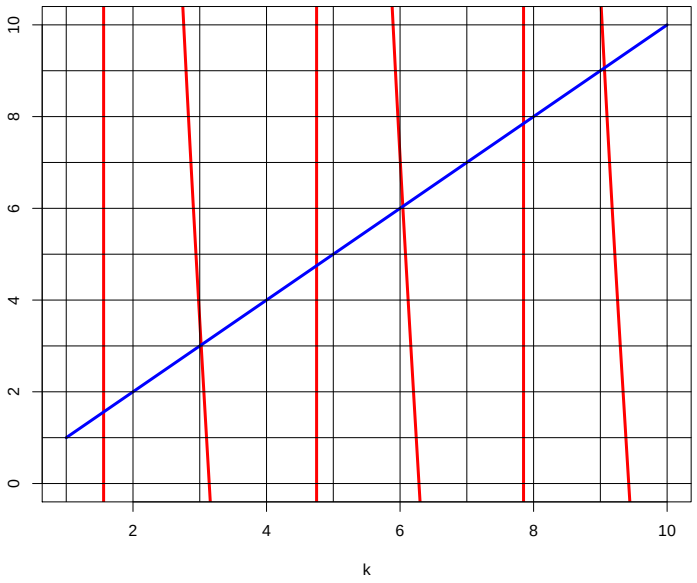

Figure 7: Graphical Search for k Roots

Physically valid k roots occur roughly at values slightly larger than $k = 3, 6, 9, \ldots$. The vertical red lines correspond to values of $k = \pi/2, 3\pi/2, \ldots$ where $tan(k)$ both changes sign and has an arbitrarily large magnitude. The intersections with the vertical red lines are not physically valid roots, as we will see.

An function which should be zero at a physical root is called **Fa(k)** and we also define **ktoE(k)**, which converts **k** to **E**.

```
Fa =  function(k) { k - (2*k^2 - gam2)*tan(k)/2/sqrt(gam2 - k^2) }

ktoE =  function (k)  k^2/gam2
```

This function **Fa(k)** (of one variable) can then be directly used with **uniroot** to find the ground state energy **E0**.

```
> Fa(2.9)
[1] -3.22901
> Fa(3.1)
[1] 2.06559
> k0 = uniroot(Fa, c(2.9, 3.1), tol=1e-16)$root
> k0
[1] 3.02069
> E0 = ktoE(k0)
> E0
[1] 0.00364983
> plot_analytic(E0)
  E  =   0.00364983
 x_mean =  0.5
 delx =  0.18802
 ydy_sum =  -1.78677e-16
 delp =  2.96193   delx*delp =  0.556902
 ymn =  0   ymx =  2
 number of nodes =  0
```

which produces the plot of the normalized analytic ground state wave function with zero nodes.

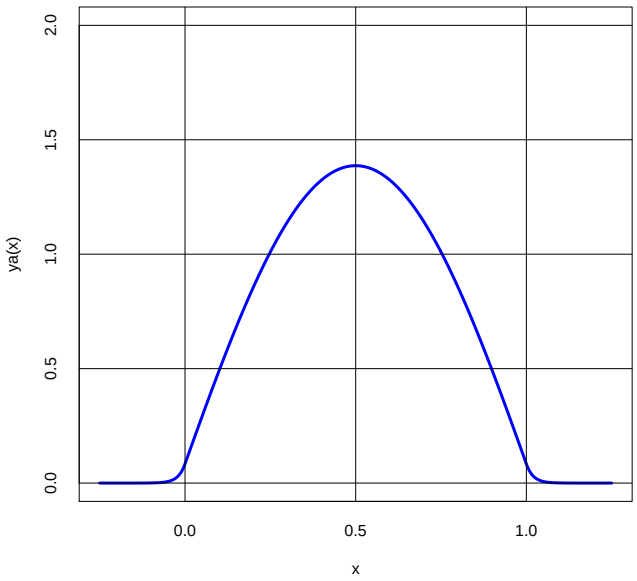

Figure 8: Analytic Normalized Ground State Wave Function

The function **uniroot** also appears to find a spurious root at or near $\pi/2$ which is not a physical solution. The function

$$F_a(k) = k - \frac{(2\,k^2 - \gamma^2)}{2\,\sqrt{\gamma^2 - k^2}}\,\tan(k). \tag{2.21}$$

is not a continuous function at $k = n\,\frac{\pi}{2}, \quad n = 1, 3, 5, \ldots$ where $\tan(k)$ is not continuous, and **uniroot** cannot be trusted at such points. For example,

```
> Fa(1.55)
[1] 1201.78
> Fa(1.59)
[1] -1298.11
> k0s = uniroot(Fa,c(1.55,1.59),tol=1e-16)$root
> k0s
[1] 1.5708
> Fa(k0s)
[1] -3.01869e+16
> E0s = ktoE(k0s)
> E0s
[1] 0.00098696
> plot_analytic(E0s)
  E  =   0.00098696
 x_mean =  0.700405
 delx =  0.212493
 ydy_sum =  1.11022e-16
 delp =  NaN   delx*delp =  NaN
 ymn =  0   ymx =  2
 number of nodes =  0
Warning message:
In sqrt(delp2) : NaNs produced
```

which produces the plot of a discontinuous zero node wave function:

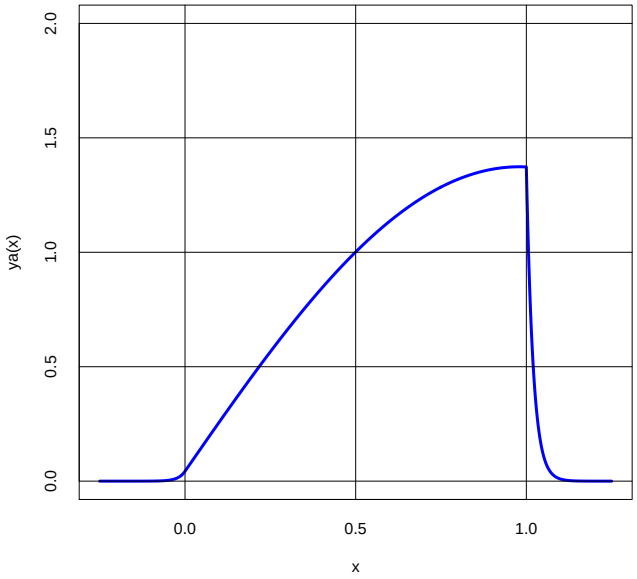

Figure 9: Discontinuous Spurious Root Wave Function

Note also the calculation produced a purely imaginary value for **delp**, which **R** writes as **NaN** (not a number). Note also the large value of **Fa(k0s)**, which should be a very small number close to a physical root.

The function **analytic_wf(E)** creates global analytic normalized wave functions **yn1(x)**, **yn2(x)**, **yn0(x)**, and **yna(x)** (and also computes and prints analytic values of Δx, represented by **delx**, and Δp, represented by **delp**). **yn1(x)** is the wave function for $x < 0$, **yn2(x)** is the wave function for $x > 1$, **yn0(x)** is the wave function for $0 \le x \le 1$. **yna(x)** is the wave function for all x, usable by **sapply**, but not by **curve**, created by the line

```
yna  <<-  function (x)  { if (x < 0)  yn1(x) else if (x > 1) yn2(x) else  yn0(x) }
```

Ignoring normalization, one can write a continuous $y_u(x)$ as

$$\begin{aligned} y_u(x) &= \sin(\delta)\, e^{k_1\, x}, \quad x \le 0, \quad k_1 = \gamma\sqrt{1-E} \\ &= \sin(k\, x + \delta), \quad 0 \le x \le 1, \quad k = \gamma\sqrt{E} \\ &= \sin(k + \delta)\, e^{k_1}\, e^{-k_1\, x}, \quad x \ge 1. \end{aligned}$$

Normalization then requires calculating

$$D = \int_{-\infty}^{\infty} y_u^2(x)\, dx. \tag{2.22}$$

and a normalized wave function $y_n(x)$ is then

$$y_n(x) = \frac{y_u(x)}{\sqrt{D}}. \tag{2.23}$$

The function **plot_analytic(E)** used above calls **analytic_wf(E)** and **nodes_analytic(ddx)**, prints out the number of nodes, and makes a plot of **yna(x)**.

```
plot_analytic =  function (E)   {
    ddx =   0.001
    xmn =   - 0.25
    xmx =   1.25
```

```
    analytic_wf(E)
    xL = seq(xmn,xmx ,by = ddx)
    yL =  sapply(xL, yna)
    ymn = floor( min(yL) )
    ymx = 1 + floor( max (yL) )
    cat (" ymn = ",ymn,"  ymx = ",ymx, "\n")
    cat (" number of nodes = ", nodes_analytic(ddx), "\n"  )
    plot(xL, yL, type = "l", lwd = 3, col = "blue", xlab = "x",
                         ylab = "ya(x)",tck=1, ylim = c(ymn,ymx) ) }
```

The function **nodes_analytic(dx)**, used above, counts the number of nodes implied by the function **yn0(x)** created by **analytic_wf(E)**. This function looks in the region $0 \leq x \leq 1$, using intervals of size **dx**.

```
nodes_analytic =  function ( dx )   {
    num =  0
    xv  =  0
    repeat  {
         f1 = yn0(xv)
         xnew = xv + dx
         if  (xnew > 1)   break
         f2 =  yn0(xv + dx)
         if  (f1*f2 < 0 )   num = num + 1
         xv = xnew }
    num }
```

A function **levels_analytic(kmax)** returns a list of analytic energy eigenvalues related to k via $k = \gamma\sqrt{E}$. The corresponding eigenvalues of k are separated by roughly $dk = 3$ and lie in the middle of the intervals $[n_1 \frac{\pi}{2}, n_2 \frac{\pi}{2}]$, where n_1 and n_2 are adjacent odd integers with $n_2 > n_1$. The ground state solution corresponds to $k = 3.02, n_1 = 1, n_2 = 3$.

```
levels_analytic  =  function (kmax)   {
     rmax = 20
     EL = rep(NA, rmax)
     level = 0
     nmx = ceiling(2*kmax/pi)
     ##           make nmx an odd integer
     if ( is.even (nmx) )  nmx = nmx + 1
     cat ("    nodes      E     \n ")
     cat ( "   \n ")

##     analytic kv using n*pi/2 + 0.1 with n odd
     for ( j  in seq(1, nmx, by=2) )  {
        out = bracket_basic( Fa,   j*pi/2 + 0.1,  0.01,0.005)
        kv = uniroot( Fa, out, tol = 1e-16)$root
        Ev = ktoE(kv)
        EL[ j ] = Ev
        cat ( "   ", level,  "   ", Ev, "\n" )
        level = level + 1 }
     EL[!is.na(EL)]   }
```

Here is an example of the use of **levels_analytic**:

```
> EL = levels_analytic(20)
    nodes      E

    0     0.00364983
    1     0.0145973
    2     0.032836
    3     0.0583552
    4     0.0911389
    5     0.131165
    6     0.178405
```

```
> EL
[1] 0.00364983 0.01459726 0.03283598 0.05835517 0.09113890 0.13116525 0.17840502
> E4 = EL[5]
> E4
[1] 0.0911389
> plot_analytic(E4)
  E  =   0.0911389
 x_mean =  0.5
 delx =  0.297122
 ydy_sum =  -8.32667e-17
 delp =  14.7876   delx*delp =  4.39371
 ymn =  -2   ymx =  2
 number of nodes =  4
```

which produces the plot

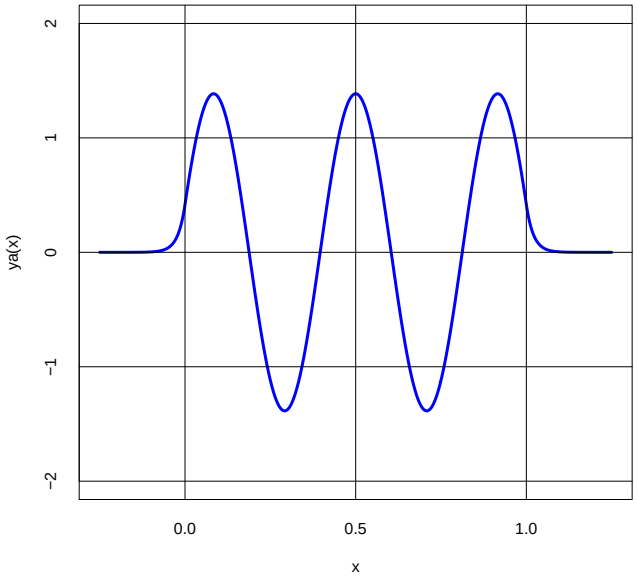

Figure 10: Analytic Four Node Wave Function $E = 0.0911389$

The function **levels_analytic** calls **bracket_basic(func,x,dx,xacc)** which looks for a sign change in **func**, starting with **x**, and increasing **x** by **dx** each step. If a sign change is found, then we back up to the previous **x** and search with new **dx** value one half of the previous value. Note that **debug = FALSE** is set when loading the file **FW.R**.

```
bracket_basic =  function (func,xx,dxx,xacc)  {
     x = xx
     dx = dxx
     it = 0
     itmax = 1000
     anerror = FALSE

    repeat  {
          it = it + 1
          if (it > itmax)  {
          cat (" can't find change in sign  \n")
                  anerror = TRUE
                  break}

          x1 = x
          x2 = x + dx
          if ( debug )  cat (" it = ",it," x1 = ",x1," x2 = ",x2," dx = ", dx, "\n")
```

```
        if  (func(x1)*func(x2) < 0 )    {
           if  ( abs(dx) < xacc )   break
           x = x - dx
           dx = dx/2 }  else x = x2 }

    if (anerror ) c(0,0)  else c(x1,x2)           }
```

Here is an example using **bracket_basic** with **func = sin**:

```
> out = bracket_basic(sin,3,0.01,0.001)
> out
[1] 3.14125 3.14187
> uniroot(sin,out,tol=1e-16)$root
[1] 3.14159
```

Here is an example of locating the zero node ground state value of **k** for the finite potential well problem using **func = Fa** (see (2.21)).

```
> out =  bracket_basic(Fa,1.6,0.1,0.05)
> out
[1] 3.000 3.025
> kv = uniroot(Fa,out,tol = 1e-16)$root
> kv
[1] 3.02069
> Ev = ktoE(kv)
> Ev
[1] 0.00364983
```

2.2 Numerical Runge Kutta Finite Well Solution

It is easiest to use Runge-Kutta methods for a case in which the potential $V(x)$ is discontinuous at one or more values of x. In this example $V(x)$ is discontinuous at $x = 0$ and $x = 1$. The Runge-Kutta method automatically supplies the first derivatives at the spatial grid points **xL**.

2.2.1 Numerical Energies and Wave Functions using Maxima

We use our homemade **rk4** routine for the Runge-Kutta integration. When the file **FW.mac** is loaded, a number of global parameters are defined. The top the the file **FW.mac** has the lines:

```
/* FW.mac uses Runge-Kutta
   for finite well.

   dimensionless units
   V = 1 for x < 0 and x > 1
   V = 0 for 0 < x < 1
   y''(x) + gam2*(E - V(x))*y(x) = 0
   gam2 = gam^2 = 2500
   gam = 50 = sqrt(2*m*L^2*V0/ hbar^2)

*/

 /*     initial global parameters:      */

 (  N : 100,
    h : 0.01,
    gam : 50,
    gam2 : gam^2,
    xdecay : 0.5,  /* start yL1 integration at x = -xdecay  */
       /* start yR integration at x = 1 + xdecay */
    ypleft : 1e-8,
    ypright : 1e-8,
    print("  gam = ",gam, "    gam2 = ", gam2),
    print(" h = ", h, ",   xdecay = ", xdecay, ",  ypleft = ",
                                  ypleft,"  ypright = ", ypright))$
```

We integrate from a point **x = -xdecay** chosen so that we can assume **y(-xdecay) = 0** to the point $x = 0$, thus defining a grid **xL1** of integration points, a list **yL1** of values of $y(x)$ at these grid points, and a list **ypL1** of values of $y'(x)$ at these grid points where $V = 1$.

We assume an arbitrary small value **ypleft** for the first derivative y' at this starting point. The resulting wave function, the solution of a homogeneous equation, can be later normalized, which will, in effect, amount to choosing the correct initial first derivative at the left starting point.

The final value of y and y' thus generated become the initial values of y and y' for integration through the region where $V = 0$, $0 \leq x \leq 1$, thus generating a grid **xL2** of integration points, a list **yL2** of values of $y(x)$ at these grid points, and a list **ypL2** of values of $y'(x)$ at these grid points where $V = 0$.

The integration in the region $x > 1$ is done by starting at a location **x = 1 + xdecay** where we can assume $y = 0$ and we again assign an arbitrary (but negative) first derivative **- ypright**. We then integrate toward smaller values of x until we reach $x = 1$. Since we are hence integrating in the direction in which the physical solution is growing, we avoid integration instability problems produced by small roundoff and integration algorithm errors.

We then multiply the lists **yL1**, **ypL1**, **yL2**, and **ypL2** by a factor which assures us that the final value of $y(x)$ produced by the independent rightward and leftward integrations agree at the matching point $x = 1$. The value of $y(x)$ can be made to agree at the matching point for any energy E. However, the resulting wave function values will still be discontinuous

because the first derivatives will not agree at the matching point.

The crucial step, then, is to design a function $F(E)$, say, that is zero (to within numerical errors) when the derivatives agree at the matching point. We can then look for the locations of sign changes in $F(E)$ to locate the energy eigenvalues.

The first step needed, in order to be able to design such a function **F(E)**, is to design a function **wf(E)** which uses Runge Kutta methods to find a un-normalized wave function corresponding to a given total energy **E**. Here is our code for such a wave function integrator, as listed in **FW.mac**.

```
/*  wf(E)    creates ** un-normalized **  numerical wave functions
     using Runge-Kutta routine rk4.
     The wave functions are stored in global xL1, yL1,ypL1, xL2, yL2, ypL2, xR, yR, ypR .
      Program also defines **global**   nleft, nright, ncenter
      the global xL1 grid extends from -xdecay to 0   and
      the global xL2 grid extends from 0 to 1 and
      the global  xR grid extends from  1 to 1 + xdecay

*/

wf(E)  :=
block([  glr,gc, outL,  fac,  numer], numer : true,

     if (E < 0) or (E > 1) then (
         print(" need 0 < E < 1 "),
         return(false)),
     ncenter : N,
     if not integerp(ncenter) then (
         print (" ncenter = ",ncenter,"  is not an integer"),
         return(false)),
     nleft : round(xdecay/h),
     nright : nleft,
     if wfdebug then print(" nleft = ",nleft," ncenter = ",ncenter," nright = ",nright),

     glr : gam2*(E - 1),          /* g(x) for x < 0 and x > 1   */
     gc : gam2*E,                 /* g(x) for 0 < x < 1  */
     if wfdebug then  print(" glr = ",glr,"   gc = ", gc),

     /* construct xL1, yL1, and ypL1 for -xdecay < x < 0  */

     outL : rk4([’y2, - glr* ’y1], [’y1, ’y2], [ 0, ypleft], [’x, -xdecay, 0, h] ),
     xL1 : take(outL,1),
     yL1 : take(outL,2),
     ypL1 : take(outL,3),

    /*  construct xL2, yL2, and ypL2 for 0 < x < 1  */

     outL : rk4([’y2, - gc* ’y1], [’y1, ’y2], [ last(yL1), last(ypL1)], [’x, 0, 1, h] ),
     xL2 : take(outL,1),
     yL2 : take(outL,2),
     ypL2 : take(outL,3),

    /*  construct xR, yR, and ypR for 1 < x < 1 + xdecay   */

     outL : rk4([’y2, - glr* ’y1], [’y1, ’y2], [ 0, -ypright], [’x, 1 + xdecay, 1, -h] ),
     xR : take(outL,1),
     yR : take(outL,2),
     ypR : take(outL,3),

     xR : reverse(xR),
     yR : reverse(yR),
     ypR : reverse(ypR),
     if wfdebug then print(" yR(1) = ", first(yR)),
```

```
    fac : first(yR)/last(yL2),
    if wfdebug then print(" fac = ",fac),

    yL1 : fac*yL1,
    ypL1 : fac*ypL1,
    yL2 : fac*yL2,
    ypL2 : fac*ypL2,

    done)$
```

The second step needed to design **F(E)** is to create a function **dy_diff()** which returns a normalized difference of the first derivatives $y'_L(x) - y'_R(x)$ evaluated at the matching point $x = 1$. We return this difference divided by the value of $y(x = 1)$.

```
/*  dy_diff() uses global ypL2, ypR, and yR,
     returns a normalized difference of derivatives
         (yL'(1) - yR'(1)/ yR(1)
*/

dy_diff() :=
block([dy_left, dy_right,  numer],numer:true,
     dy_left : last(ypL2),
     dy_right : first(ypR),
     (dy_left - dy_right)/abs(first(yR)) )$
```

For example,

```
(%i1) load(cp3);
(%o1) "c:/k3/cp3.mac"
(%i2) load(FW);
  gam =  50     gam2 =  2500
 h =  0.01 ,   xdecay =  0.5 ,  ypleft =  1.0E-8   ypright =  1.0E-8
(%o2) "c:/k3/FW.mac"
(%i3) wf(0.5);
(%o3) done
(%i4) dy_diff();
(%o4) 35.071714
(%i5) last(ypL2);
(%o5) -0.00190471
(%i6) first(ypR);
(%o6) -0.237432
(%i7) first(yR);
(%o7) 0.00671558
```

Here, finally, is our code for **F(E)**:

```
/* energy eigenvalue if global function F(E) = 0 .
  F(E) calls wf(E) then returns dy_diff(), but
    returns false if E > 0.
 */

F(E) :=
block( [ numer],numer:true,
    if E < 0 or E > 1 then (
         print(" in F(E), E = ",E,"  should be between 0 and 1 "),
         return(false)),
    wf(E),
    dy_diff())$
```

And here is an example of using **F(E)** to produce a rough graphical survey of the possibilities:

```
(%i8) EL : makelist(e,e,0.1,0.9,0.01)$
(%i9) FL : map(F, EL)$
(%i10) plot2d([discrete,EL,FL],[xlabel,"E"],[ylabel,"F(E)"])$
```

which produces the plot

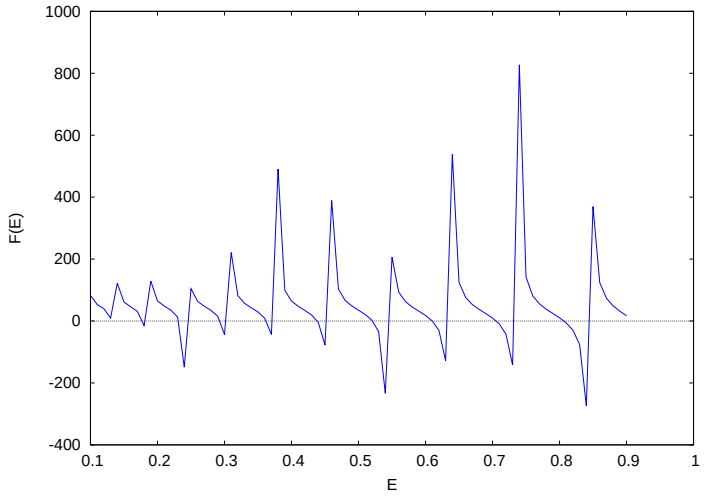

Figure 11: Crude Plot of F(E)

We can use **F(E)** to search for energy eigenvalue candidates. We do this with a function **bracket(func,x,dx,xacc)** which attempts to return two values of **x** at which **func** has the opposite sign.

```
/*  bracket is a modified version of bracket_basic, designed to work with
    the function F(E) or F1(k) which can return 'false'.

    bracket looks for a sign change in func,
    starting with xx,  and increasing xx by dxx each step.
    If sign change is found,  then we back up to the previous xx
    and search with new  dxx value one half of the previous value.
    normally returns [ea,eb] or [ka,kb], but if can't find change in sign,
    then returns [0,0], and if func returns false, then
    bracket returns false.
*/

bracket(func,xx,dxx,xacc) :=
block([f1,f2, x:xx, dx:dxx,xx1,xx2,it:0,itmax:1000],

     do (
         it : it + 1,
         if debug then print(it),
         if it > itmax then (
              print(" can't find change in sign "),
              return([0, 0 ])),
         xx1 : x,
         xx2 : x + dx,

         if debug then print(" it = ",it," xx1 = ",xx1," xx2 = ",xx2," dx = ", dx),
         f1 : func(xx1),
         if not f1 then (
               print(" in bracket, f1 = false , xx1 = ",xx1, "  dx = ", dx),
               return(f1)),

         f2 : func(xx2),
```

```
        if not f2 then (
             print(" in bracket, f2 = false ,  xx2 = ",xx2, "  dx = ", dx),
             return(f2)),
        if debug then print (" f1 = ",f1,"  f2 = ",f2),
        if f1*f2 < 0 then (
              if abs(dx) < xacc then return([xx1,xx2]),
              x : x - dx,
              dx : dx/2)
        else x : xx2) )$
```

Here is an example of using **bracket** with the function **F(E)**. This example produces the zero node ground state case, and we plot the un-normalized wave function pieces produced by **wf(E)**.

```
(%i11) [ea,eb] : bracket(F,0.0005,0.0001,0.00005);
(%o11) [0.003625,0.00365]
(%i12) e : find_root(F,ea,eb);
(%o12) 0.00364983
(%i13) wf(e);
(%o13) done
(%i14) plot2d([[discrete,xL1,yL1],[discrete,xL2,yL2],[discrete,xR,yR]],
            [xlabel,"x"],[ylabel,"y(x)"], [legend,false])$
```

which produces a plot of the un-normalized ground state wave function with zero nodes.

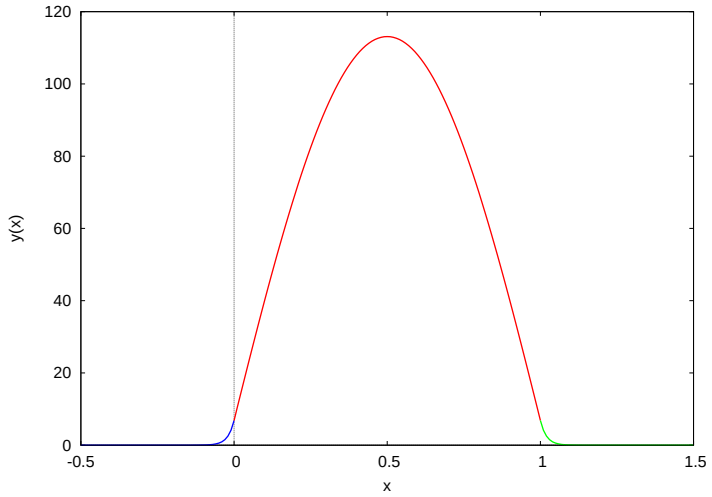

Figure 12: Numerical Un-normalized Ground State Wave Function

We can check the normalized difference in slopes at the matching point for the solution produced by **wf(E)**:

```
(%i15) dy_diff();
(%o15) -2.49565076E-14
```

We can check the number of nodes with the function **num_nodes()**.

```
/*  count the number of nodes in yL2  */

num_nodes() :=
block([ n, numer], numer:true,
        n : 0,
        for j thru (length(yL2) - 1) do
            if yL2[ j ] * yL2[ j + 1 ] < 0 then n : n + 1,
        n)$
```

For the numerical ground state solution generated above by **wf(E)** we get:

```
(%i16) num_nodes();
(%o16) 0
```

A function **normalize()** uses the current wave function pieces produced by **wf(E)** and uses our Simpson's rule function **simp** to produce global normalized wave function (lists) **xn** and **yn**. **normalize()** uses our function **merge(aL1,aL2)** to combine the separate lists into one list. After calling **normalize()**, one can check the normalization:

```
(%i17) normalize()$
 AA =  6652.6824
 x_mean =  0.5
 delx =  0.18802
(%i18) simp(xn,yn^2);
(%o18) 1.0
```

Here is the code for **normalize()**.

```
 /* normalize() uses the current global xL1,yL1, xL2,yL2, xR, yR and
  the  utility functions  merge and simp  to define global
  xn and yn, with the latter being normalized.
  */

normalize() :=
block ( [AA,x_mean,x2_mean,delx,delx2, numer ], numer:true,

     xn :  merge( xL1, merge( rest(xL2), rest(xR))),
     yn :  merge( yL1, merge( rest(yL2), rest(yR))),

     /* we need xn to have odd # of elements to use simp */
     if evenp ( length (xn) ) then (
          xn : rest (xn),
          yn : rest (yn)),
     if debug then print ( " fll(xn) = ", fll(xn) ),
     if debug then print( " fll(yn) = ", fll(yn) ),
     AA : simp(xn,yn^2),
     print( " AA = ",AA),
     yn : yn/sqrt(AA),
     if debug then print( " fll(yn) = ", fll(yn) ),

     x_mean : simp(xn, xn * yn^2),
     print(" x_mean = ", x_mean),
     x2_mean : simp(xn, xn^2 * yn^2),

     delx2 : x2_mean - x_mean^2,   /* this should be positive!  */

     delx : sqrt(delx2),

     print(" delx = ", delx),
     done)$
```

Once we have used **normalize()**, we can use the function **yn_plot_current()**, which uses the current lists **xn** and **yn**.

```
(%i19) yn_plot_current()$
 ymax =  1.3867012
```

which produces the plot

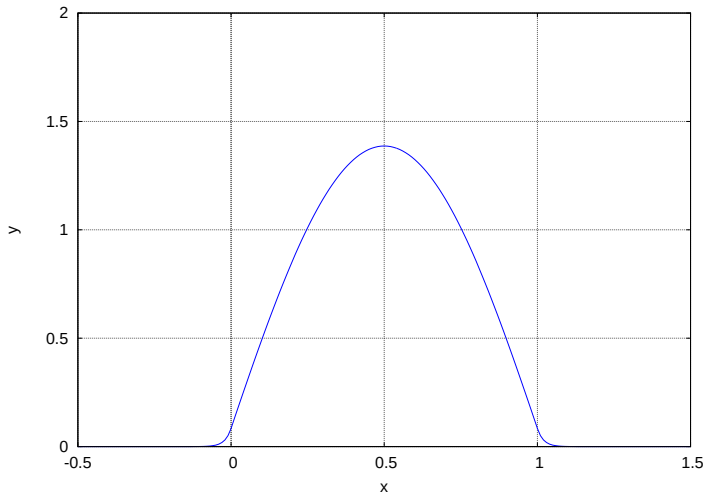

Figure 13: Numerical Normalized Ground State Wave Function

Here is our code for **yn_plot_current()**:

```
/*   yn_plot_current() uses the currently defined normalized set (xn,yn)
*/

yn_plot_current() :=
block([ymn, ymx, numer],numer:true,
         ymn : float(floor( lmin(yn) )),
         ymx : float( 1 + floor( lmax (yn) ) ),
         print(" ymax = ", lmax(yn) ),
         plot2d( [discrete, xn, yn], ['y,ymn,ymx],
                 [ylabel,"y"],  [xlabel,"x"],
                 [style,[lines,3]], [legend, false],  [gnuplot_preamble,"set grid"]))$
```

The more versatile function **yn_plot(E,xmin,xmax)** does three tasks in succession, first calling **wf(E)** to create the un-normalized wave function pieces, then calling **normalize()** to create the normalized wave function lists **xn** and **yn**, and finally making a plot of the normalized wave function, using **xmin** and **xmax** to control the horizontal display.

Here is an example dealing with the first excited (one node) state.

```
(%i20) [ea,eb] : bracket(F,0.01,0.005,0.001);
(%o20) [0.014375,0.015]
(%i21) e : find_root(F,ea,eb);
(%o21) 0.0145973
(%i22) yn_plot(e,-0.5,1.5)$
 E =  0.0145973
 number of nodes =  1 ,       dy_diff =  -7.59213486E-14
 AA =  1278.4068
 x_mean =  0.5
 delx =  0.276562
 normalized ymax =  1.3865517
```

which produces the plot

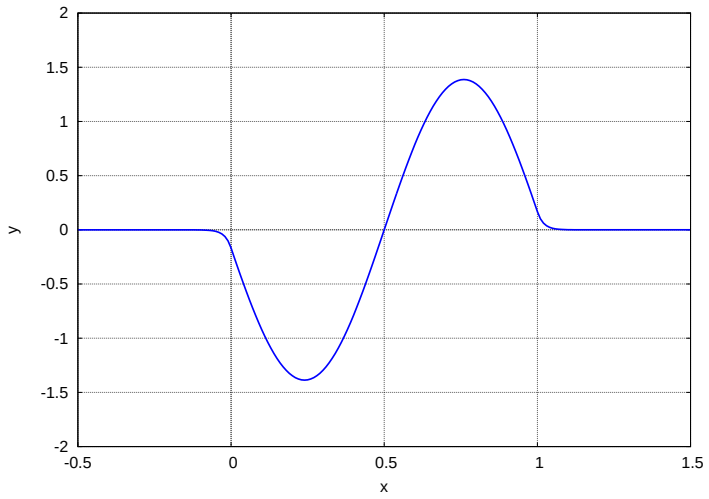

Figure 14: Numerical Normalized First Excited State Wave Function

Here is our code for **yn_plot**:

```
/*   yn_plot(E,xmin,xmax) first calls wf(E) to create
     un-normalized wave functions corresponding to the
         given energy E. Then normalizes those wave functions
         to produce the lists xn and yn. Finally makes a plot
         of yn over only the region (xmn, xmx)   */

yn_plot(E,xmn,xmx) :=
block([ymn, ymx, numer],numer:true,
     wf(E),
     print(" E = ",E ),
     print(" number of nodes = ",num_nodes(),",       dy_diff = ",dy_diff() ),
     normalize(),
     print(" normalized ymax = ", lmax(yn) ),
     ymn : floor( lmin(yn) ),
     ymx : 1 + floor( lmax (yn) ),
     plot2d( [discrete, xn, yn], ['x,xmn, xmx],
          ['y,ymn,ymx],  [ylabel,"y"],  [xlabel,"x"],
            [style, [lines, 3] ], [legend, false],  [gnuplot_preamble,"set grid"]))$
```

We now want to construct a function **levels(...)** which will produce a list of the energy levels, found using our numerical Runge-Kutta methods, starting with the ground state energy, and continuing up to some maximum energy. Some experimentation shows that instead of using **F(E)** with a succession of small values of **E**, it is easier to use a function **F1(k)**, since the **k** values corresponding to the energy eigenvalues are larger numbers of order $O(1)$. Here is such a function which we call **F1(k)**.

```
/* energy eigenvalue if global function F1(k) = 0 .
  F1(k) calls wf(E) then returns dy_diff(), but
    returns false if k <=  0 or >= gam .
 */

F1(k) :=
block( [ numer],numer:true,
     if k <= 0 or k >= gam then (
          print(" in F1(k), k = ",k,"  k should be greater than 0 and less than ",gam),
          return(false) ),
     wf(k^2/gam2),
     dy_diff())$
```

Here is an example of use of **F1(k)** to find the ground state energy:

```
(%i23) F1(2.9);
(%o23) 34.29503
(%i24) F1(3.05);
(%o24) -49.882755
(%i25) [ka,kb] : bracket(F1,2.9,0.05,0.02);
(%o25) [3.0125,3.025]
(%i26) k0 : find_root(F1,ka,kb);
(%o26) 3.0206914
(%i27) E0 : ktoE(k0);
(%o27) 0.00364983
(%i28) wf(E0);
(%o28) done
(%i29) dy_diff();
(%o29) -1.49739046E-13
(%i30) num_nodes();
(%o30) 0
```

which reveals a zero node wave function with a very small value of **dy_diff()**, a signal of a good wave function.

We can make a crude plot of **F1(k)** versus **k**

```
(%i31) kL : makelist(k,k,1,49,0.5)$
(%i32) F1L : map(F1, kL)$
(%i33) time(%);
(%o33) [6.23]
(%i34) plot2d([discrete,kL,F1L],[xlabel,"k"],[ylabel,"F1(k)"])$
```

which produces the plot

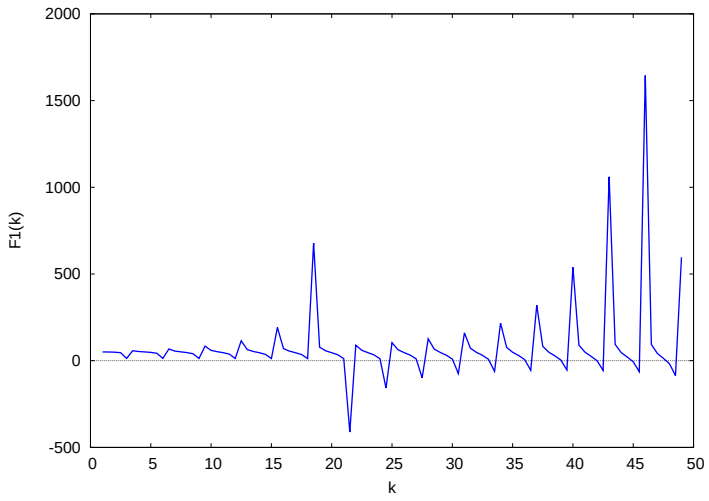

Figure 15: Crude Plot of F1(k) versus k

Let's use **F1(k)** to look at the region **2.8 <= k <= 3.3**:

```
(%i315 kL : makelist(k,k,2.8,3.3,0.05)$
(%i36) F1L : map(F1, kL)$
(%i37) plot2d([discrete,kL,F1L],[xlabel,"k"],[ylabel,"F1(k)"])$
```

which produces the plot

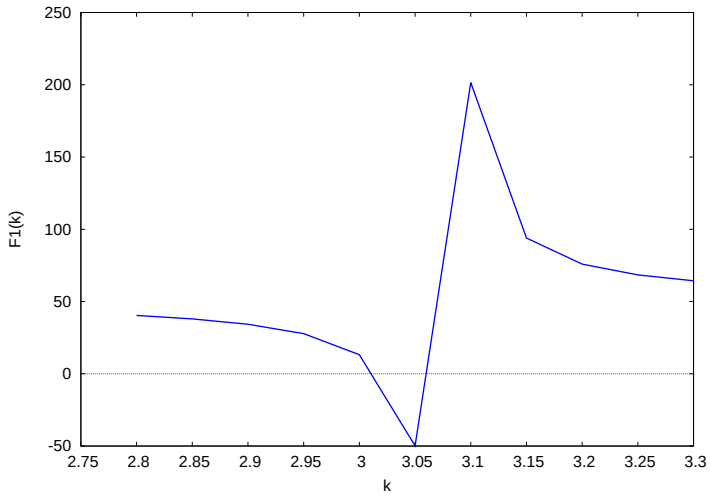

Figure 16: F1(k) Zoom

Placing the cursor on the **`F1 = 0`** intersections shows roots at roughly **`k = 3.01`** and **`k = 3.06`**. The first root is close to the valid ground state case as shown above. We can use **`bracket`** with **`F1(k)`** to refine the second root.

```
(%i38) [ka,kb] : bracket(F1,3.03,0.01,0.005);
(%o38) [3.0775,3.08]
(%i39) kv : find_root(F1,ka,kb);
(%o39) 3.0799546
(%i40) Ev : ktoE(kv);
(%o40) 0.00379445
(%i41) wf(Ev);
(%o41) done
(%i42) num_nodes();
(%o42) 0
(%i43) dy_diff();
(%o43) -1.60000886E+16
```

The large value of **`dy_diff()`** shows that this second, slightly larger root, is an un-physical solution. Since we already have a valid zero node solution at the lower energy, there cannot be a second zero node solution at another, higher, energy. This pattern persists, with the physical root being smaller, and the un-physical root being slightly larger. This pattern provides the rationale for our code for **`levels(...)`**. Once we have found a solution with a given number of nodes, we reject all solutions with higher energy but the same number of nodes. The unphysical roots correspond to a sudden change in which (left or right) integration function has the larger slope magnitude at the matching point.

Here is our code for **`levels(kmin,kmax,dk, kacc )`**.

```
/* levels(kmin,kmax,dk, kacc ) returns a list  [Ea, Eb,...] of energy levels with
     increasingly larger number of nodes in energy range (Emin, Emax)
     according to Emin = kmin^2/gam^2, and Emax = kmax^2/gam^2.
     uses F1(k) (inside bracket) to find  roots, and calls wf(E), num_nodes() and
         dy_diff() for each root found,
     Uses bracket and find_root.
     The arguments (dk, kacc) are used to call bracket, and do not describe
        the accuracy of the energy levels found.
     Once a good energy e.v. is found we look for the region of
         energies with one more node and search there.
     Includes an interactive continue or stop input.
     Searching for the k eigenvalues via F1(k) is easier than searching
          directly for the E eigenvalues via F(E) for the case gam = 50 which we
          consider in our examples.
*/
```

```
levels(kmin,kmax,dk, kacc ) :=
block([ k,knext, kroot,eroot, eL, ka, kb, nn, nlast : -1, r, numer], numer:true,
     k : kmin,
     eL : [ ],     /*    list eL will hold energy eigenvalues found  */
     do (
          if k > kmax then return(),    /* exit do loop  */
          print("---------------- levels ------------------"),
          print(" nlast = ", nlast),
          print(" kstart = ", k," dk = ", dk ),
          [ka, kb] : bracket(F1,k, dk, kacc),
          print(" ka = ",ka," kb = ",kb),
          if float(ka) = 0.0 then (
               print(" can't find bracket interval "),
               print(" k = ",k),
               return() ),
          kroot : find_root(F1, ka, kb),
          print(" kroot = ", kroot),
          eroot : kroot^2/gam2,
          print(" eroot = ", eroot),
          wf(eroot),
          nn : num_nodes(),
          print(" number of nodes = ", nn),
          print(" dy_diff at x = 1 is    ", dy_diff() ),
          eL : cons(eroot, eL),
          nlast : nn,
          r : read (" input c; or s; "),
          if string(r) = "s" then return(),        /* exit do loop  */

     /* search for a k value greater than kb which produces
                 a wave function with nn + 1 nodes */
          knext : kb + dk,
          do (
               wf(knext^2/gam2),
               if num_nodes() > nlast then (
                    k : knext,
                    return() )
               else knext : knext + dk)),

     reverse(eL) )$
```

Here is an example of using **`levels`**. To continue to the next energy eigenvalue, one enters **`c;`** at the prompt (for "continue"). Actually, any letter except **`s;`** will cause the program to continue.

```
(%i44) EL : levels(1,8,0.05,0.02);
---------------- levels ------------------
 nlast =  -1
 kstart =  1  dk =  0.05
 ka =  3.0125  kb =  3.025
 kroot =  3.0206914
 eroot =  0.00364983
 number of nodes =  0
 dy_diff at x = 1 is     -2.49565076E-14
 input c; or s;
c;
---------------- levels ------------------
 nlast =  0
 kstart =  3.125  dk =  0.05
 ka =  6.0375  kb =  6.05
 kroot =  6.040956
 eroot =  0.0145973
 number of nodes =  1
 dy_diff at x = 1 is     2.18273877E-13
 input c; or s;
c;
```

```
---------------- levels ------------------
 nlast =  1
 kstart =  6.2  dk =  0.05
 ka =  9.05  kb =  9.0625
 kroot =  9.0603555
 eroot =  0.032836
 number of nodes =  2
 dy_diff at x = 1 is     7.10273285E-14
 input c; or s;
c;
(%o44) [0.00364983,0.0145973,0.032836]
```

We can then use **yn_plot(E,xmin,xmax)** to both construct the lists **xn** and **yn** of the normalized wave function and make a plot. We can also use **makelist** to construct one list **xyn**, say, which combines the lists **xn** and **yn** into one. In the following, we do not show the plots produced by the calls to **yn_plot**.

```
(%i45) yn_plot(EL[1],-0.5,1.5)$
 E =  0.00364983
 number of nodes =  0 ,       dy_diff =  -2.49565076E-14
 AA =  6652.6824
 x_mean =  0.5
 delx =  0.18802
 normalized ymax =  1.3867012
(%i46) xyn0 : makelist([xn[j],yn[j]],j,1,length(xn))$
(%i47) fll(xyn0);
(%o47) [[-0.5,0.0],[1.5,0.0],201]
```

We continue in this manner with the first excited state and the second excited state.

```
(%i48) yn_plot(EL[2],-0.5,1.5)$
 E =  0.0145973
 number of nodes =  1 ,       dy_diff =  -2.05936658E-12
 AA =  1278.4068
 x_mean =  0.5
 delx =  0.276562
 normalized ymax =  1.3865517
(%i49) xyn1 : makelist([xn[j],yn[j]],j,1,length(xn))$
(%i50) yn_plot(EL[3],-0.5,1.5)$
 E =  0.032836
 number of nodes =  2 ,       dy_diff =  -2.00652203E-12
 AA =  365.3835
 x_mean =  0.5
 delx =  0.290108
 normalized ymax =  1.385693
(%i51) xyn2 : makelist([xn[j],yn[j]],j,1,length(xn))$
```

We can then combine the plots for the wave functions of these three lowest lying states.

```
(%i52) plot2d([[discrete,xyn0],[discrete,xyn1],[discrete,xyn2]],
          [xlabel,"x"],[ylabel,"y"],[style,[lines,2]],
            [legend,"E0","E1","E2"])$
```

which produces the plot

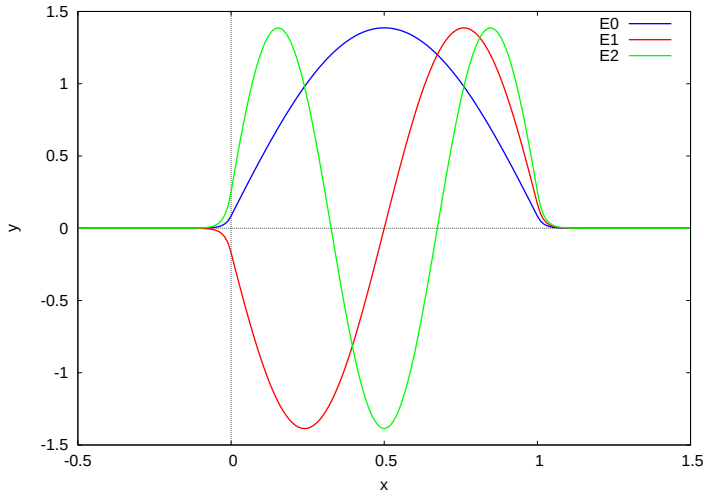

Figure 17: Finite Well: Lowest Three Energy Level Wave Functions

Just for practice, we can write these three wave functions to a file in the local folder, and then restart Maxima, read in the file contents, and make a plot based on the file contents. We have discussed some details of such read and write actions within Maxima in Chapter 2 of Maxima by Example.

The simplest approach is to use **`save(filePath,a1,a2,a3,...)`**, where objects **`a1`**, **`a2`**, etc are object names bound to quantities known to Maxima. The names and the objects bound to the names are stored in Lisp language format in the file requested (which is created if it does not yet exist, and overwritten if it already exists). You can, of course, open that file with a text editor to see the contents, written in Lisp.

One can use **`load`** to load in that file into a new Maxima session, and the names and objects will then be available for use in your new Maxima session. If you don't remember the names you used in your previous session, you can use **`values;`** to generate a list of currently known object names.

```
(%i53) E0 : EL[1];
(%o53) 0.00364983
(%i54) E1 : EL[2];
(%o54) 0.0145973
(%i55) E2 : EL[3];
(%o55) 0.032836
(%i56) save("c:/k3/fw1.dat",E0,xyn0,E1,xyn1,E2,xyn2);
(%o56) "c:/k3/fw1.dat"
```

If we look at the top of the file **`c:/k3/fw1.dat`** with a text editor (such as $Notepad++$) we see:

```
;;; -*- Mode: LISP; package:maxima; syntax:common-lisp; -*-
(in-package :maxima)
(DSKSETQ |$e0| 0.0036498306077588829)
(ADD2LNC '|$e0| $VALUES)
(DSKSETQ $XYN0
         '((MLIST SIMP) ((MLIST SIMP) -0.5 0.0)
           ((MLIST SIMP) -0.48999999999999999 1.2769296242295217E-12)
           ((MLIST SIMP) -0.47999999999999998 2.8785287263561221E-12)
           ((MLIST SIMP) -0.46999999999999997 5.2117339500682875E-12)
           ((MLIST SIMP) -0.46000000000000002 8.8694267352155839E-12)
           ((MLIST SIMP) -0.45000000000000001 1.4781088055950433E-11)
etc., etc.
```

We now restart Maxima and load in **`cp3.mac`**, **`FW.mac`**, and the data file created using **`save`** above.

```
(%i1) load(cp3);
(%o1) "c:/k3/cp3.mac"
(%i2) load(FW);
  gam =  50     gam2 =  2500
 h =  0.01 ,   xdecay =  0.5 ,  ypleft =  1.0E-8   ypright =  1.0E-8
(%o2) "c:/k3/FW.mac"
(%i3) load("c:/k3/fw1.dat");
(%o3) "c:/k3/fw1.dat"
(%i4) values;
(%o4) [mydate,_binfo%,N,h,gam,gam2,xdecay,ypleft,ypright,E0,xyn0,E1,xyn1,E2,xyn2]
(%i5) E0;
(%o5) 0.00364983
(%i6) E1;
(%o6) 0.0145973
(%i7) E2;
(%o7) 0.032836
(%i8) fll(xyn0);
(%o8) [[-0.5,0.0],[1.5,0.0],201]
(%i9) head(xyn0);
(%o9) [[-0.5,0.0],[-0.49,1.27692962E-12],[-0.48,2.87852873E-12],
       [-0.47,5.21173395E-12],[-0.46,8.86942674E-12],[-0.45,1.47810881E-11]]
(%i10) plot2d([[discrete,xyn0],[discrete,xyn1],[discrete,xyn2]],
          [xlabel,"x"],[ylabel,"y"],[style,[lines,2]],
            [legend,"E0","E1","E2"])$
```

and we get the same plot as above.

2.2.2 Numerical Energies and Wave Functions using R

We use our homemade **myrk4** routine for the Runge-Kutta integration. When the file **FW.R** is loaded, a number of global parameters are defined. The top the the file **FW.R** has the lines:

```
## FW.R uses Runge-Kutta  for finite well.

##   dimensionless units
##   V = 1 for x < 0 and x > 1
##   V = 0 for 0 < x < 1
##   y''(x) + gam2*(E - V(x))*y(x) = 0
##   gam2 = gam^2 = 2500
##   gam = 50 = sqrt(2*m*L^2*V0/ hbar^2)


##     initial global parameters:

    N = 100
    h = 0.01
    gam = 50
    gam2 = gam^2
    xdecay = 0.5      ## start yL1 integration at x = -xdecay
       ## start yR integration at x = 1 + xdecay
    ypleft = 1e-8
    ypright = 1e-8
    debug = FALSE
    wfdebug = FALSE
    cat ("  gam = ",gam, "    gam2 = ", gam2,"\n")
    cat (" h = ", h, ", xdecay = ", xdecay, ",ypleft = ", ypleft,"  ypright = ",ypright,"\n")
```

We integrate from a point **x = -xdecay** chosen so that we can assume **y(-xdecay) = 0** to the point **x = 0**, thus defining a grid vector **xL1** of integration points, a vector **yL1** of values of $y(x)$ at these grid points, and a vector **ypL1** of values of $y'(x)$ at these grid points where $V = 1$.

We assume an arbitrary small value **ypleft** for the first derivative y' at this starting point. The resulting wave function, the solution of a homogeneous equation, can be later normalized, which will, in effect, amount to choosing the correct

initial first derivative at the left starting point.

The final value of y and y' thus generated become the initial values of y and y' for integration through the region where $V = 0$, $0 \leq x \leq 1$, thus generating a grid vector **xL2** of integration points, a vector **yL2** of values of $y(x)$ at these grid points, and a vector **ypL2** of values of $y'(x)$ at these grid points where $V = 0$.

The integration in the region $x > 1$ is done by starting at a location **x = 1 + xdecay** where we can assume $y = 0$ and we again assign an arbitrary (but negative) first derivative **-ypright**. We then integrate toward smaller values of x until we reach $x = 1$. Since we are hence integrating in the direction in which the physical solution is growing, we avoid integration instability problems produced by small roundoff and integration algorithm errors.

We then multiply the vectors **yL1**, **ypL1**, **yL2**, and **ypL2** by a factor which assures us that the final value of $y(x)$ produced by the independent rightward and leftward integrations agree at the matching point $x = 1$. The value of $y(x)$ can be made to agree at the matching point for any energy E. However, the resulting wave function values will still be discontinuous because the first derivatives will not agree at the matching point.

The crucial step, then, is to design a function $F(E)$, say, that is zero (to within numerical errors) when the derivatives agree at the matching point. We can then look for the locations of sign changes in $F(E)$ to locate the energy eigenvalues.

The first step needed, in order to be able to design such a function **F(E)**, is to design a function **wf(E)** which uses Runge Kutta methods to find a un-normalized wave function corresponding to a given total energy **E**. Here is our code for such a wave function integrator, as listed in **FW.R**.

```
##       wf(E)    creates ** un-normalized **  numerical wave functions
##       using Runge-Kutta routine myrk4.
##       The wave functions are stored in global vectors
##              xL1, yL1,ypL1, xL2, yL2, ypL2, xR, yR, ypR .
##       Program also defines **global**   nleft, nright, ncenter.
##       the global xL1 grid extends from -xdecay to 0   and
##       the global xL2 grid extends from 0 to 1 and
##       the global  xR grid extends from  1 to 1 + xdecay

wf =  function (E)    {
     if  ( (E < 0) | (E > 1) ) {
          cat (" need 0 < E < 1  \n")
          return(false) }
     ncenter = N
     if ( round(ncenter) != ncenter) {
          cat (" ncenter = ",ncenter,"  is not an integer \n")
          return(false) }
     nleft = round(xdecay/h)
     nright = nleft
     if (wfdebug)  cat (" nleft = ",nleft," ncenter = ",ncenter," nright = ",nright, "\n")
     glr =  gam2*(E - 1)          ##   g(x) for x < 0 and x > 1
     gc =  gam2*E                 ##    g(x) for 0 < x < 1
     if (wfdebug) cat ("  glr = ",glr,"   gc = ", gc, "\n")
     ##     construct xl1, yl1, and ypl1 for -xdecay < x < 0
     derivs.decay = function (x,y) { c (y[2], - glr*y[1]) }
     xl1 = seq (- xdecay, 0, h)
     outL = myrk4( c(0,ypleft), xl1, derivs.decay)
     yl1 = outL[[1]]
     ypl1 = outL[[2]]
     ##     construct xl2, yl2, and ypl2 for 0 < x < 1
     derivs01 = function (x,y) { c (y[2], - gc*y[1]) }
     xl2 = seq(0, 1, h)
     outL = myrk4( c(last(yl1), last(ypl1)), xl2, derivs01)
     yl2 = outL[[1]]
     ypl2 = outL[[2]]
```

```
     ##     construct xr, yr, and ypr for 1 < x < 1 + xdecay
     xr = seq( 1 + xdecay, 1, -h)
     outL = myrk4( c(0, -ypright), xr, derivs.decay)
     yr = outL[[1]]
     ypr = outL[[2]]
     xr = rev(xr)
     yr = rev(yr)
     ypr = rev(ypr)
     if (wfdebug)   cat (" yr(1) = ", yr[1], "\n" )
     fac =  yr[1] / last(yl2)
     if (wfdebug)   cat (" fac = ",fac, "\n")
     yl1 = fac*yl1
     ypl1 = fac*ypl1
     yl2 = fac*yl2
     ypl2 = fac*ypl2
     ##   make global xL1,xL2,xR,yL1,yL2,yR,ypL1,ypL2,ypR
     xL1 <<-  xl1
     xL2 <<-  xl2
     xR <<-  xr
     yL1 <<-  yl1
     yL2 <<-  yl2
     yR <<-   yr
     ypL1 <<-  ypl1
     ypL2 <<-  ypl2
     ypR <<-  ypr  }
```

The second step needed to design **F(E)** is to create a function **dy_diff()** which returns a normalized difference of the first derivatives $y'_L(x) - y'_R(x)$ evaluated at the matching point $x = 1$. We return this difference divided by the value of $y(x = 1)$.

```
##    dy_diff() uses global ypL2, ypR, and yR,
##       returns a normalized difference of derivatives
##           (yL'(1) - yR'(1)/ yR(1)

dy_diff = function ()  {
     dy_left = last(ypL2)
     dy_right = ypR[1]
     (dy_left - dy_right)/abs(  yR[1]   ) }
```

For example,

```
> wf(0.5)
> dy_diff()
[1] 35.0717
> last(ypL2)
[1] -0.00190471
> ypR[1]
[1] -0.237432
> yR[1]
[1] 0.00671558
```

Here is our code for **F(E)**:

```
##   energy eigenvalue if global function F(E) = 0 .
##     F(E) calls wf(E) then returns dy_diff(), but
##       returns FALSE  if E < 0 or > 1 .
F =  function (E)    {
     if  (E < 0 | E > 1)    {
          cat (" in F(E), E = ",E,"  should be between 0 and 1   \n ")
          return(FALSE)  }
     wf(E)
     dy_diff()   }
```

Here is an example of using **F(E)** to produce a rough graphical survey of the possibilities:

```
> EL = seq(0.1,0.9,0.01)
> FL = sapply(EL, F)
> fll(FL)
  82.3613   17.2303   81
> plot(EL, FL, type = "l", lwd = 2, col = "blue", xlab = "E",ylab = "F(E)" )
> mygrid()
```

which produces the plot

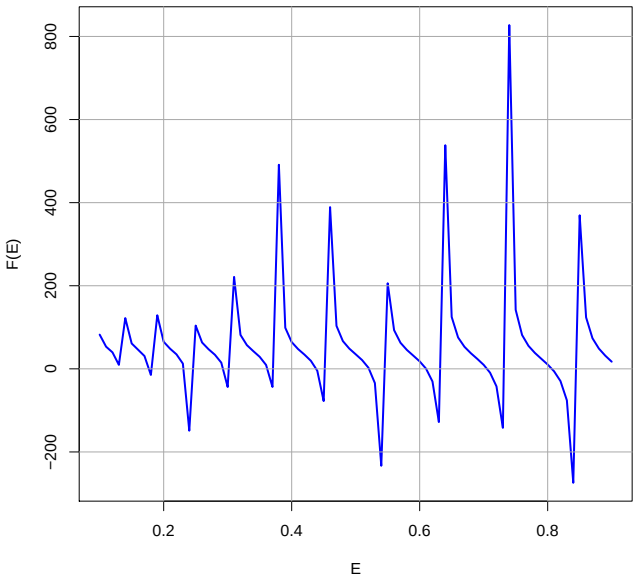

Figure 18: Crude Plot of F(E)

We can use **F(E)** to search for energy eigenvalue candidates. We do this with a function **bracket(func,x,dx,xacc)** which attempts to return two values of **x** at which **func** has the opposite sign.

```
##  bracket is a modified version of bracket_basic, designed to work with
##    the functions F(E) or F1(k) which can return FALSE.
##    bracket looks for a sign change in func,
##    starting with xx,  and increasing xx by dxx each step.
##    If sign change is found,  then we back up to the previous xx
##    and search with new  dxx value one half of the previous value.
##    normally returns [ea,eb], but if can't find change in sign,
##       then returns [0,0], and if func returns FALSE, then
##       bracket returns FALSE.

bracket =  function (func,xx,dxx,xacc)  {
     x = xx
     dx = dxx
     it = 0
     itmax = 1000
     anerror = FALSE
     anerror2 = FALSE

     repeat  {
         it = it + 1
         if (it > itmax)  {
             cat (" can't find change in sign  \n")
```

```
            anerror = TRUE
            break}
        x1 = x
        x2 = x + dx
        if ( debug )  cat (" it = ",it," x1 = ",x1," x2 = ",x2," dx = ", dx, "\n")
        f1 = func(x1)
        if ( f1 == FALSE)   {
            cat (" in bracket, f1 = FALSE , x1 = ",x1, "  dx = ", dx, " \n ")
            anerror2 = TRUE
            break }
        f2 = func(x2)
        if ( f2 == FALSE)   {
            cat (" in bracket, f2 = FALSE , x2 = ",x2, "  dx = ", dx, " \n ")
            anerror2 = TRUE
        break }
        if  ( f1 * f2  < 0 )    {
            if  ( abs(dx) < xacc )   break
            x = x - dx
            dx = dx/2 }  else x = x2 }
    if (anerror) c(0,0) else if (anerror2) FALSE  else c(x1,x2) }
```

Here is an example of using **bracket** with the function **F(E)**. This example produces the zero node ground state case, and we plot the un-normalized wave function pieces produced by **wf(E)**.

```
> out = bracket(F,0.0005,0.0001,0.00005)
> out
[1] 0.003625 0.003650
> e = uniroot(F,out, tol = 1e-16)$root
> e
[1] 0.00364983
> wf(e)
> plot(0, type = "n", xlim = c(min(xL1), max(xR)), ylim = c(0, max(yL2)),
+               xlab = "x", ylab = "y" )
> lines (xL1, yL1, lwd = 3, col = "blue")
> lines (xL2, yL2, lwd = 3, col = "red")
> lines (xR, yR, lwd = 3, col = "green")
> mygrid()
```

which produces a plot of the un-normalized ground state wave function with zero nodes.

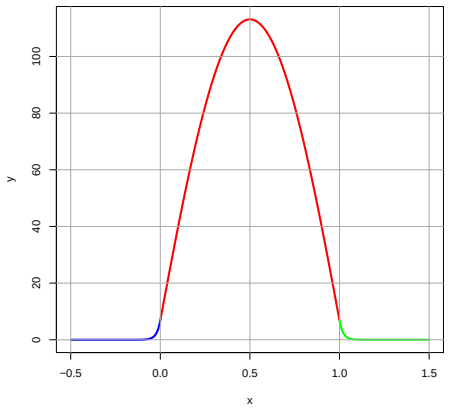

Figure 19: Numerical Un-normalized Ground State Wave Function

We can check the normalized difference in slopes at the matching point for the solution produced by **wf(E)**:

```
> dy_diff()
[1] -3.42736e-12
```

We can check the number of nodes with the function **num_nodes()**.

```
num_nodes  =  function ()    {
        n = 0
        for ( j  in 1 :  (length(yL2) - 1) )   { if (yL2[ j ] * yL2[ j + 1 ] < 0)  n = n + 1 }
        n  }
```

For the numerical ground state solution generated above by **wf(E)** we get:

```
> num_nodes()
[1] 0
```

A function **normalize()** uses the current wave function pieces produced by **wf(E)** and uses our Simpson's rule function **simp** to produce global normalized wave function (vectors) **xn** and **yn**. We use the function **rest** defined in **cp3.R** in **normalize()**.

Before we use **normalize()**, let's show interactively the route we follow in the beginning of **normalize()**:

```
> xn = c (xL1, c ( rest(xL2), rest(xR) ) )
> fll(xn)
  -0.5   1.5   201
> yn = c (yL1, c ( rest(yL2), rest(yR) ) )
> fll(yn)
  0   0   201
> head(xn)
[1] -0.50 -0.49 -0.48 -0.47 -0.46 -0.45
> head(yn)
[1] 0.00000e+00 1.04151e-10 2.34784e-10 4.25090e-10 7.23426e-10 1.20560e-09
> simp(xn,yn^2)
[1] 6652.68
```

Here we call **normalize()**, and then check the normalization interactively using Simpson's rule **simp**.

```
> normalize()
 AA =  6652.68
 x_mean =  0.5
 delx =  0.18802
> simp(xn,yn^2)
[1] 1
```

Here is the code for **normalize()**.

```
##   normalize() uses the current global xL1,yL1, xL2,yL2, xR, yR and
##    the  utility functions  rest and simp  to define global
##    xn and yn, with the latter being normalized.

normalize  =  function ()   {

     xn = c (xL1, c ( rest(xL2), rest(xR) ) )
     yn = c (yL1, c ( rest(yL2), rest(yR) ) )
     ##   we need xn to have odd number of elements to use simp
     if ( is.even ( length (xn) ) )   {
         xn = rest (xn)
         yn = rest (yn)    }
     AA = simp(xn,yn^2)
     cat ( " AA = ",AA, "\n" )
     yn = yn/sqrt(AA)
     x_mean = simp(xn, xn * yn^2)
     cat (" x_mean = ", x_mean, "\n" )
     x2_mean = simp(xn, xn^2 * yn^2)
     delx2 = x2_mean - x_mean^2      ## this should be positive!
     delx = sqrt(delx2)
     cat (" delx = ", delx, "\n" )
     xn <<-  xn
     yn <<-  yn }
```

After using **normalize()**, one can plot the normlized wave function (the current vectors **xn** and **yn**) using the function **yn_plot_current()**:

```
> yn_plot_current()
 ymax =  1.3867
```

which produces the plot

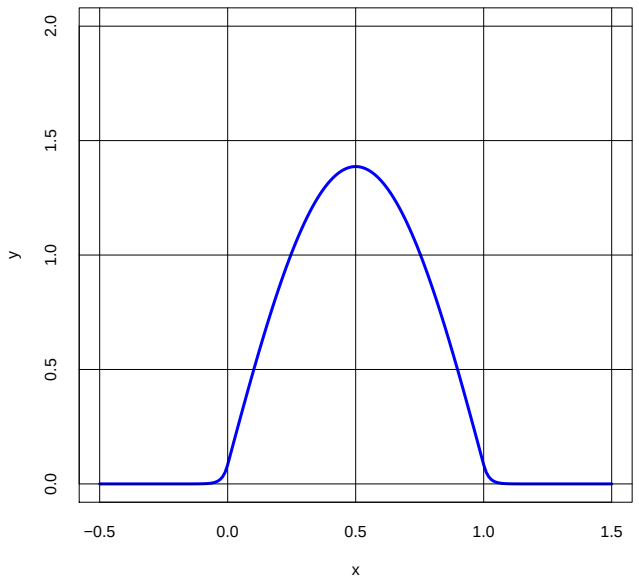

Figure 20: Numerical Normalized Ground State Wave Function

The function **yn_plot_current()** uses the current global vectors **xn** and **yn** created by **normalize()**.

```
##     yn_plot_current() uses the currently defined normalized set (xn,yn)
yn_plot_current  =  function ()    {
         ymn = floor( min(yn) )
         ymx =  1 + floor( max (yn) )
         cat (" ymax = ", max(yn), "\n"  )
         plot(xn, yn, type = "l", lwd = 3, col = "blue", ylim = c(ymn, ymx),
                  xlab = "x",  ylab = "y", tck = 1) }
```

The more versatile function **yn_plot(E,xmin,xmax)** does three tasks in succession, first calling **wf(E)** to create the un-normalized wave function pieces, then calling **normalize()** to create the normalized wave function vectors in the form of **xn** and **yn**, and finally making a plot of the normalized wave function, using **xmin** and **xmax** to control the display.

Here is an example dealing with the first excited (one node) state.

```
> out = bracket(F,0.01,0.005,0.001)
> out
[1] 0.014375 0.015000
>  e = uniroot(F,out, tol = 1e-16)$root
> e
[1] 0.0145973
> yn_plot(e,-0.5,1.6)
 E =  0.0145973
 number of nodes =  1 ,       dy_diff =  1.61333e-13
 AA =  1278.41
 x_mean =  0.5
 delx =  0.276562
 normalized ymax =  1.38655
```

which produces the plot

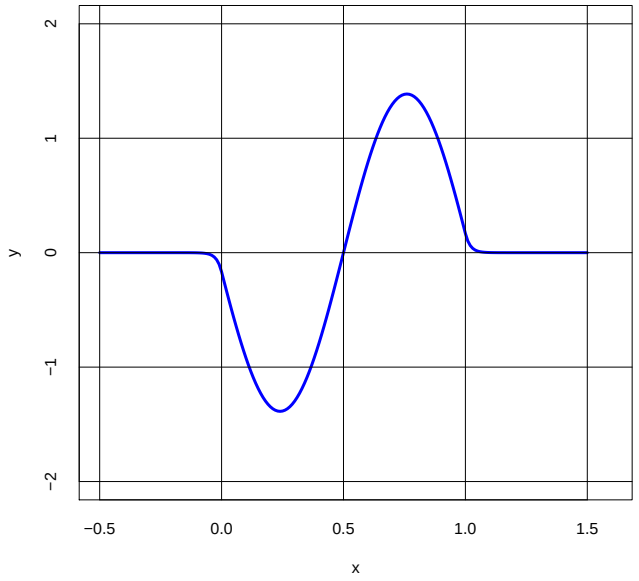

Figure 21: Numerical Normalized First Excited State Wave Function

Here is our code for **yn_plot**:

```
##    yn_plot(E,xmin,xmax) first calls wf(E) to create
##    un-normalized wave functions corresponding to the
##    given energy E. Then normalizes those wave functions
##    to produce the vectors xn and yn. Finally makes a plot
##    of yn over only the region (xmn, xmx)

yn_plot  =  function (E,xmn,xmx)   {
     wf(E)
     cat (" E = ",E, "\n" )
     cat (" number of nodes = ",num_nodes(),", dy_diff = ",dy_diff(), "\n" )
     normalize()
     cat (" normalized ymax = ", max(yn), "\n" )
     ymn = floor( min(yn) )
     ymx =  1 + floor( max (yn) )
     plot(xn, yn, type = "l", lwd = 3, col = "blue", xlim = c(xmn,xmx), ylim = c(ymn, ymx),
                 xlab = "x",  ylab = "y", tck = 1) }
```

We now want to construct a function **levels(...)** which will produce a vector of the energy levels, found using our numerical Runge-Kutta methods, starting with the ground state energy, and continuing up to some maximum energy. Some experimentation shows that instead of using **F(E)** with a succession of small values of **E**, it is easier to use a function **F1(k)**, since the **k** values corresponding to the energy eigenvalues are larger numbers of order $O(1)$. Here is such a function, which we call **F1(k)**.

```
##     F1(k):  energy eigenvalue if global function F1(k) = 0 .
##    F1(k) calls wf(E) then returns dy_diff(), but
##         returns false if k <= 0 or k >=  gam.
F1  =  function (k)    {
     if  (k <= 0 | k >= gam)   {
         cat (" in F1(k), k = ",k,"  k should be greater than 0 and less than ",gam, "\n")
         return(FALSE)   }
     wf(k^2/gam2)
     dy_diff()   }
```

Here is an example of use of **F1(k)** to find the ground state energy:

```
> F1(2.9)
[1] 34.295
> F1(3.05)
[1] -49.8828
> out = bracket(F1,2.9,0.05,0.02)
> out
[1] 3.0125 3.0250
> k0 = uniroot(F1, out, tol = 1e-16)$root
> k0
[1] 3.02069
> E0 = ktoE(k0)
> E0
[1] 0.00364983
> wf(E0)
> dy_diff()
[1] 3.41072e-13
> num_nodes()
[1] 0
```

which reveals a zero node wave function with a very small value of **dy_diff()**, a signal of a good wave function.

We can make a crude plot of **F1(k)** versus **k**

```
> kL = seq(1,49,by=0.5)
> head(kL)
[1] 1.0 1.5 2.0 2.5 3.0 3.5
> F1L = sapply (kL, F1)
> head (F1L)
[1] 50.6042 50.0387 48.9460 46.2163 13.2072 57.5400
> plot(kL, F1L, type = "l", lwd = 2,col = "blue",xlab = "k",ylab = "F1(k)")
> mygrid()
```

which produces the plot

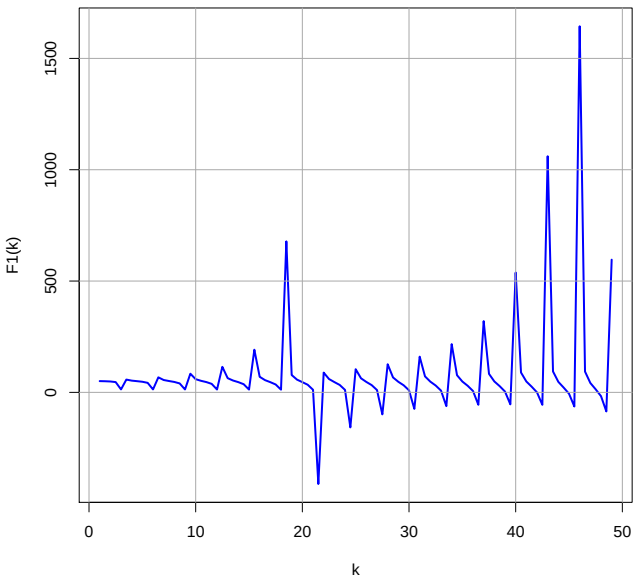

Figure 22: Crude Plot of F1(k) versus k

Let's use **F1(k)** to look at the region **2.8 <= k <= 3.3**:

```
> kL = seq (2.8, 3.3, by = 0.05)
> head(kL)
[1] 2.80 2.85 2.90 2.95 3.00 3.05
> F1L =  sapply( kL, F1)
> head(F1L)
[1]  40.3843  37.9920  34.2950  27.7891  13.2072 -49.8828
> plot(kL, F1L, type = "l", lwd = 2,col = "blue",xlab = "k",ylab = "F1(k)")
> mygrid()
```

which produces the plot

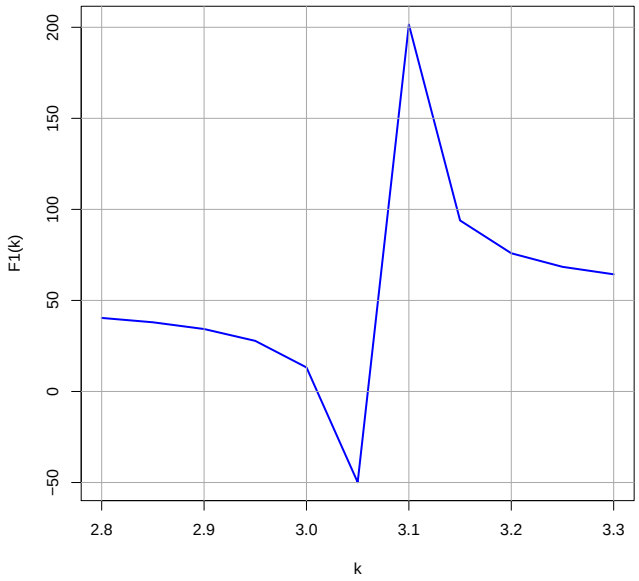

Figure 23: F1(k) Zoom

This zoom plot shows roots at roughly **k = 3.01** and **k = 3.06**. The first root is close to the valid ground state case as shown above. We can use **bracket** with **F1(k)** to refine the second root.

```
> out = bracket(F1,3.03,0.01,0.005)
> out
[1] 3.0775 3.0800
> kv = uniroot(F1, out, tol = 1e-16)$root
> kv
[1] 3.07995
> Ev = ktoE(kv)
> Ev
[1] 0.00379445
> wf(Ev)
> num_nodes()
[1] 0
> dy_diff()
[1] -2.64829e+15
```

The large value of **dy_diff()** shows that this second, slightly larger root, is an un-physical solution. Since we already have a valid zero node solution at the lower energy, there cannot be a second zero node solution at another, higher, energy. This pattern persists, with the physical root being smaller, and the un-physical root being slightly larger. This pattern provides the rationale for our code for **levels(...)**. Once we have found a solution with a given number of nodes, we reject all solutions with higher energy but the same number of nodes. The unphysical roots correspond to a sudden change in which (left or right) integration function has the larger slope magnitude at the matching point.

Here is an example of using **levels**.

```
> EL = levels(1,8,0.05,0.02)
---------------- levels -------------------
 nlast =  -1
 kstart =  1  dk =  0.05
 ka =  3.0125  kb =  3.025
 kroot =  3.02069
 eroot =  0.00364983
 number of nodes =  0
 dy_diff at x = 1 is     3.41072e-13
 input c or s
 c
---------------- levels -------------------
 nlast =  0
 kstart =  3.125  dk =  0.05
 ka =  6.0375  kb =  6.05
 kroot =  6.04096
 eroot =  0.0145973
 number of nodes =  1
 dy_diff at x = 1 is     1.61333e-13
 input c or s
 c
---------------- levels -------------------
 nlast =  1
 kstart =  6.2  dk =  0.05
 ka =  9.05  kb =  9.0625
 kroot =  9.06036
 eroot =  0.032836
 number of nodes =  2
 dy_diff at x = 1 is     -1.36136e-13
 input c or s
 c
> EL
[1] 0.00364983 0.01459726 0.03283602
```

We can then use **yn_plot(E,xmin,xmax)** to both construct the vectors **xn** and **yn** of the normalized wave function and make a plot. We save the normalized wave functions by assignment statements such as **xn0 = xn**, and **yn0 = yn**, before another call to **yn_plot** defines the wave functions corresponding to a different energy. In the following, we do not show the plots produced by the calls to **yn_plot**.

```
> yn_plot(EL[1],-0.5,1.5)
 E =  0.00364983
 number of nodes =  0 ,       dy_diff =  3.41072e-13
 AA =  6652.68
 x_mean =  0.5
 delx =  0.18802
 normalized ymax =  1.3867
> xn0 = xn
> fll(xn0)
  -0.5   1.5   201
> yn0 = yn
> fll(yn)
  0   0   201
```

We continue in this manner with the first excited state and the second excited state.

```
> yn_plot(EL[2],-0.5,1.5)
 E =  0.0145973
 number of nodes =  1 ,       dy_diff =  1.61333e-13
 AA =  1278.41
 x_mean =  0.5
 delx =  0.276562
 normalized ymax =  1.38655
> xn1 = xn
> fll(xn1)
  -0.5   1.5   201
> yn1 = yn
> fll(yn1)
  0   0   201
```

```
> yn_plot(EL[3],-0.5,1.5)
 E =  0.032836
 number of nodes =  2 ,       dy_diff =  -1.36136e-13
 AA =  365.384
 x_mean =  0.5
 delx =  0.290108
 normalized ymax =  1.38569
> xn2 = xn
> yn2 = yn
```

We can then combine the plots for the wave functions of these three lowest lying states.

```
> plot(0,type = "n",xlim = c(-0.5,1.5),ylim = c(-1.5,1.5),xlab="x",ylab="y")
> lines(xn0,yn0,lwd=2,col="blue")
> lines(xn1,yn1,lwd=2,col="red")
> lines(xn2,yn2,lwd=2,col="green")
> mygrid()
> legend("bottomright",col = c("blue","red","green"),
+              legend = c("E0", "E1", "E2"), lwd=2,cex=1.5)
```

which produces the plot

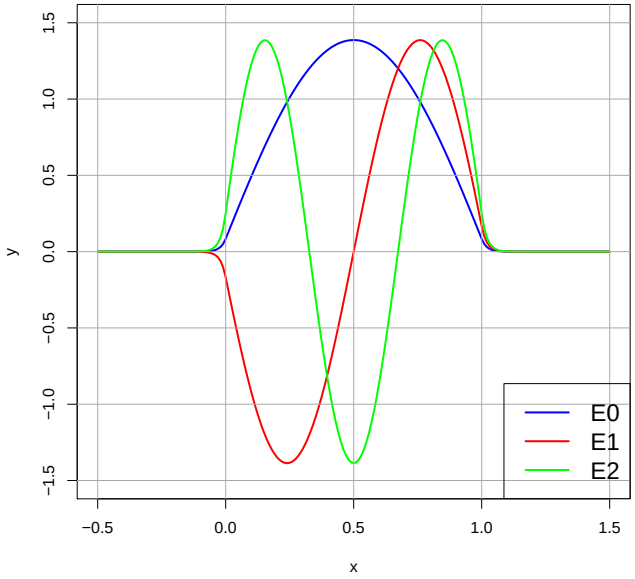

Figure 24: Finite Well: Lowest Three Energy Level Wave Functions

Just for practice, we can write these three wave functions to a file in the local folder, and then restart **R**, read in the file contents, and make a plot based on the file contents.

The simplest approach is to use **save(filePath,a1,a2,a3,...)**, where objects **a1**, **a2**, etc are object names bound to quantities known to R. The names and the objects bound to the names are stored in a binary file format in the file requested (which is created if it does not yet exist, and overwritten if it already exists).

One can use **load** to load in that file into a new session, and the names and objects will then be available for use in your new Maxima session. In the following, we first save the wave function files to **xy.rda**. We then use **rm** to remove knowledge of those objects from the current session. We then use **load** to recover knowledge of those objects, which can then be used as before, for example to make plots and make calculations.

```
> save(xn0,yn0,xn1,yn1,xn2,yn2, file = "xy.rda")
> rm(xn0,yn0,xn1,yn1,xn2,yn2)
```

```
> fll(xn0)
Error in fll(xn0) : object 'xn0' not found
> load("xy.rda")
> fll(xn0)
  -0.5   1.5   201
```

For example, we can remake the plot of all three wave functions

```
> plot(0,type = "n",xlim = c(-0.5,1.5),ylim = c(-1.5,1.5),xlab="x",ylab="y")
> lines(xn0,yn0,lwd=2,col="blue")
> lines(xn1,yn1,lwd=2,col="red")
> lines(xn2,yn2,lwd=2,col="green")
> mygrid()
> legend("bottomright",col = c("blue","red","green"),
+              legend = c("E0", "E1", "E2"), lwd=2,cex=1.5)
```

and we get the same plot as above.

Here is the code for **`levels`**:

```
##    levels(kmin,kmax,dk, kacc ) returns a vector  c( Ea, Eb,...)  of energy levels with
##    increasingly larger number of nodes in energy range (Emin, Emax)
##    according to Emin = kmin^2/gam^2, and Emax = kmax^2/gam^2.
##    uses F1(k) (inside bracket) to find  roots, and calls wf(E), num_nodes() and
##        dy_diff() for each root found,
##    Uses bracket and uniroot.
##    The arguments (dk, kacc) are used to call bracket, and do not describe
##        the accuracy of the energy levels found.
##    Once a good energy e.v. is found we look for the region of
##        energies with one more node and search there.
##    Code has interactive continue or stop.
##    Searching for the k eigenvalues via F1(k) is easier than searching
##        directly for the E eigenvalues via F(E) for the case gam = 50 we
##         consider in this example.
levels  =  function (kmin,kmax,dk, kacc )   {
     rmax = 20
     eL = rep(NA, rmax)         ##  vector  eL will hold energy eigenvalues found
     k = kmin
     nlast =  -1
     j = 1
     repeat {
         if  (k > kmax  |  j > rmax)  break    ## exit do loop
         cat ("---------------- levels ------------------\n")
         cat (" nlast = ", nlast,"\n")
         cat (" kstart = ", k," dk = ", dk, "\n" )
         out =  bracket(F1,k, dk, kacc)
         cat (" ka = ",out[1]," kb = ",out[2],"\n")
         if (out[1] == 0)  {
             cat (" can't find bracket interval \n")
             cat (" k = ",k, "\n")
             break  }
         kroot = uniroot(F1, out, tol = 1e-16)$root
         cat (" kroot = ", kroot, "\n")
         eroot = kroot^2/gam2
         cat (" eroot = ", eroot, "\n")
         wf(eroot)
         nn = num_nodes()
         cat (" number of nodes = ", nn, "\n")
         cat (" dy_diff at x = 1 is    ", dy_diff(), "\n" )
         eL[ j ]  = eroot
         nlast = nn
         j = j + 1
```

```
        r = readline (" input c or s \n ")
        if (r == "s")  break            ## exit do loop
        ##    search for a k value greater than kb which produces
        ##      a wave function with nn + 1 nodes
        knext  =  out[2] + dk
        repeat  {
            wf(knext^2/gam2)
            if  (num_nodes() > nlast)  {
                k = knext
                break }  else knext =  knext + dk }  }   ## end of outer repeat loop
    ##   remove NA's at end of eL
    eL[!is.na(eL)]  }
```

3 The Numerov Integration Method

Numerov's method was developed by the Russian astronomer Boris Vasil'evich Numerov in the years 1924-1927. Numerov's algorithm is a simple and efficient method for integrating linear second order ode's which do not contain a first order derivative term and is especially useful for homogeneous ode's, such as Schroedinger's equation.

Corresponding to a grid of equally spaced values x_n of the independent variable x, will be values y_n of the dependent variable. A numerical solution of the ode

$$y''(x) + g(x)\,y(x) = S(x) \tag{3.1}$$

can then be constructed using the following Numerov three term recursion relation

$$\left(1 + \frac{h^2}{12}\,g_{n+1}\right) y_{n+1} - 2\left(1 - \frac{5h^2}{12}\,g_n\right) y_n + \left(1 + \frac{h^2}{12}\,g_{n-1}\right) y_{n-1} = \frac{h^2}{12}\,(S_{n+1} + 10S_n + S_{n-1}) + O(h^6) \tag{3.2}$$

Solving this linear equation for either y_{n+1} or y_{n-1} then provides a recursion relation for integrating either forward or backward in x, with a local error $O(h^6)$. The Numerov scheme is more efficient than the Runge-Kutta method, as each step of the Numerov method requires the computation of g and S at only the grid points (and not at intermediate points). However, the Runge-Kutta method provides both $y(x)$ and $y'(x)$ at the grid points, whereas the Numerov method only provides $y(x)$ at the grid points.

For a derivation of the Numerov method, see

```
https://en.wikipedia.org/wiki/Numerov's_method
```

3.1 Classical Simple Harmonic Oscillator Test Case

Let's try out the Numerov method for a classical simple harmonic oscillator with unit period, defined by

$$\frac{d^2y}{dx^2} = -4\,\pi^2\,y(x), \quad y(0) = 1, \quad y'(0) = 0 \tag{3.3}$$

over the domain $0 \le x \le 1$. This corresponds to (3.1) with $S(x) = 0$ and $g(x) = 4\,\pi^2$, in which case the Numerov algorithm (3.2) takes the form (integrating in the direction of increasing x):

$$y_{n+1} = A\,y_n - y_{n-1} \tag{3.4}$$

with

$$A = \frac{2\left(1 - \frac{5\,h^2\,\pi^2}{3}\right)}{\left(1 + \frac{h^2\,\pi^2}{3}\right)} \tag{3.5}$$

The analytic solution for the initial conditions assumed in (3.3) is $y(x) = \cos(2\,\pi\,x)$. Given $y_0 = y(x = 0)$ and $y'_0 = y'(x = 0)$ we can calculate $y_1 = y(h)$ using a Taylor series expansion about $x = 0$. In order to calculate $y_2 = y(2\,h)$ with an accuracy $O(h^6)$ we need y_1 with this same accuracy. It is sufficient, however, to calculate y_1 with an accuracy $O(h^5)$ because the global error of Numerov's method is $O(h^5)$ and we calculate y_1 just once. Using the expansion

$$y_1 = y(h) = y_0 + h\,y'_0 + \frac{h^2}{2}y''(0) + \frac{h^3}{3!}y'''(0) + \frac{h^4}{4!}y''''(0) + O(h^5) \tag{3.6}$$

we get

$$y_1 = y_0 + h\,y'_0 - 2\,\pi^2\,h^2\,y_0 - \frac{2}{3}\pi^2\,h^3\,y'_0 + \frac{2}{3}\pi^4\,h^4\,y_0 \tag{3.7}$$

3.1.1 Classical SHO Numerov Method Using Maxima

We have written two Maxima "do loop" versions of Numerov's method to integrate our classical simple harmonic oscillator example. These two versions are called **sho(h,y0,yp0)** and **sho2(h,y0,yp0)** and they are in the file **cp3.mac**. Here is the first version, which builds up a list of lists **rL** using the Maxima function **cons**.

```
/*   sho(h,y0,yp0)

integrates simple harmonic oscillator with unit period
   d^2y/dx^2 = - 4 pi^2 y(x) over [x,0,1]
   using the numerov method.
   Input: h = step size, y0 = y(0), yp0 = y'(0)
   Output: list [[0,y0],[h,y1],[2 h, y2],...]
*/

sho(h,y0,yp0) :=
block([A,y1,N,ym,yz,yp,rL,x,
               xmin:0,xmax:1,numer],numer:true,
     A : 2*(1 - 5*%pi^2*h^2/3)/(1 + %pi^2*h^2/3),
     y1 : y0 + h*yp0 - 2*%pi^2*h^2*y0 - 2*%pi^2*h^3*yp0/3 +
                  2*%pi^4*h^4*y0/3,
     N :        round( (xmax - xmin)/h ),
     rL : [[xmin+h,y1], [xmin,y0]],
     x : xmin + 2*h,
     ym : y0,
     yz : y1,
     for j thru N - 1  do (
         yp : A*yz - ym,
         rL : cons ( [x,yp], rL),
         ym : yz,
         yz : yp,
         x : x + h),
     reverse(rL))$
```

This code uses **yp** ("y plus") to represent y_{n+1}, **yz** (" y zero") to represent y_n, and **ym** ("y minus") to represent y_{n-1}, hence the line **yp : A*yz - ym,** in the loop, with values being "rolled" at the end of the loop. If we use $h = 0.01$, we get good agreement with the exact solution:

```
(%i1) load(cp3);
(%o1) "c:/k3/cp3.mac"
(%i2) soln : sho(0.01, 1, 0)$
(%i3) xL : take(soln,1)$
(%i4) fll(xL);
(%o4) [0,1.0,101]
(%i5) head(xL);
(%o5) [0,0.01,0.02,0.03,0.04,0.05]
(%i6) tail(xL);
(%o6) [0.95,0.96,0.97,0.98,0.99,1.0]
(%i7) xL[1];
(%o7) 0
(%i8) yL : take(soln,2)$
```

```
(%i9) fll(yL);
(%o9) [1,1.0,101]
(%i10) head(yL);
(%o10) [1,0.998027,0.992115,0.982287,0.968583,0.951057]
(%i11) yL[1];
(%o11) 1
(%i12) plot2d([['discrete,xL,yL],cos(2*%pi*x)],['x,0,1],['xlabel,"x"],
         ['ylabel,""],['legend,"numerov","analytic"],
          [gnuplot_preamble,"set key bottom"])$
(%i13) plot2d([['discrete,soln],cos(2*%pi*x)],['x,0,1],['xlabel,"x"],
         ['ylabel,""],['legend,"numerov","analytic"],
          [gnuplot_preamble,"set key bottom"])$
```

Either of the above plot2d statements produce the plot

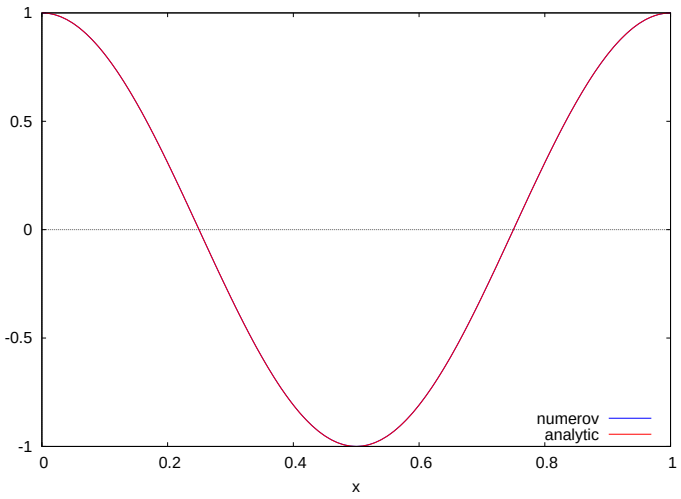

Figure 25: Classical SHO Numerov Solution with $h = 0.01$

The second Maxima version **sho2** uses 'hashed arrays' called **xL** and **yL** in the code. By using a line in the code of the form **local(xL,yL)** these hashed arrays are not known at the global level, and we can reuse the same names globally as shown in the example below. Note that the form of the output of **sho2** is different than that of **sho**.

```
sho2(h,y0,yp0) :=
block([A,y1,xmin,xmax,N,x,ym,yz,yp, numer], numer:true,
    local(xL,yL),
    A : 2*(1 - 5*%pi^2*h^2/3)/(1 + %pi^2*h^2/3),
    y1 : y0 + h*yp0 - 2*%pi^2*h^2*y0 - 2*%pi^2*h^3*yp0/3 +
                 2*%pi^4*h^4*y0/3,
    print(" A = ",A," y1 = ", y1),
    xmin : 0,
    xmax : 1,
    N :        round ( (xmax - xmin)/h),
    print(" N = ", N),

    xL[1] : xmin,
    xL[2] : xmin + h,
    yL[1] : y0,
    yL[2] : y1,
    x : xmin + 2*h,
    ym : y0,
    yz : y1,
    for j : 3   thru  N + 1 do (
        yp : A*yz - ym,
        xL[j] : x,
        yL[j] : yp,
        ym : yz,
```

```
        yz : yp,
        x : x + h ),
    [ listarray(xL), listarray(yL) ] )$
```

We use the same input parameters as before in this example. To emphasize the fact that the code line **local(xL,yL)** hides the values assigned to the hashed arrays, we first use **remvalue** to remove the definition of the lists **xL** and **yL** produced in the example above.

```
(%i14) remvalue(xL, yL);
(%o14) [xL,yL]
(%i15) xL[1];
(%o15) xL[1]
(%i16) yL[1];
(%o16) yL[1]
(%i17) soln : sho2(0.01, 1, 0)$
 A =  1.9960535  y1 =  0.998027
 N =  100
(%i18) xL[1];
(%o18) xL[1]
(%i19) xL : soln[1]$
(%i20) fll(xL);
(%o20) [0,1.0,101]
(%i21) head(xL);
(%o21) [0,0.01,0.02,0.03,0.04,0.05]
(%i22) yL[1];
(%o22) yL[1]
(%i23) yL : soln[2]$
(%i24) fll(yL);
(%o24) [1,1.0,101]
(%i25) plot2d([['discrete,xL,yL],cos(2*%pi*x)],['x,0,1],['xlabel,"x"],
         ['ylabel,""],['legend,"numerov","analytic"],
          [gnuplot_preamble,"set key bottom"])$
```

which produces exactly the same plot as produced using **sho**.

3.1.2 Classical SHO Numerov Method Using R

The file **cp3.R** contains a function **sho(h,y0,yp0)** with the same input syntax as the Maxima version. But the R function returns a R list: **list(xL, yL)** in which **xL** is a vector containing the x positions, and **yL** is a vector containing the corresponding y values produced by the Numerov code.

```
##     sho(h,y0,yp0)

##  integrates simple harmonic oscillator with unit period
##     d^2y/dx^2 = - 4 pi^2 y(x) over [x,0,1]
##     using the numerov method.
##     Input: h = step size, y0 = y(0), yp0 = y'(0)
##     Output: list( xL, yL)


sho =  function(h,y0,yp0) {
    A = 2*(1 - 5*pi^2*h^2/3)/(1 + pi^2*h^2/3)
    y1 = y0 + h*yp0 - 2*pi^2*h^2*y0 - 2*pi^2*h^3*yp0/3 +
                 2*pi^4*h^4*y0/3
    xmin = 0
    xmax = 1
    N =        (xmax - xmin)/h
    yL = vector(length = N + 1)
    xL = vector(length = N + 1)
    xL[1] = xmin
    xL[2] = xmin + h
    yL[1] = y0
    yL[2] = y1
```

```
    x = xmin + 2*h
    ym = y0
    yz = y1
    for (j in 3:(N + 1)) {
        yp = A*yz - ym
        xL[j] = x
        yL[j] = yp
        ym = yz
        yz = yp
        x = x + h }
    list( xL, yL) }
```

We can then compare the Numerov method with the analytic solution with the same value of h as used in our Maxima work above. The R function **`fll`** (also in **`cp3.R`**) prints out the first, last, and length of a R vector.

```
> source("cp3.R")
> soln = sho(0.01,1,0)
 A =  1.99605  y1 =  0.998027
 N =  100
> xL[1]
Error: object 'xL' not found
> xL = soln[[1]]
> fll(xL)
  0   1   101
> yL = soln[[2]]
> fll(yL)
  1   1   101
> plot(xL,yL,type="l",lwd=2,xlab="x",ylab = "")
> curve(cos(2*pi*x),0,1,add=TRUE,col="red",lwd=2)
> mygrid()
> legend("bottomright",col=c("black","red"),cex=1.5,
+            legend=c("Numerov","Analytic"),lwd=2)
```

which produces a plot which shows agreement:

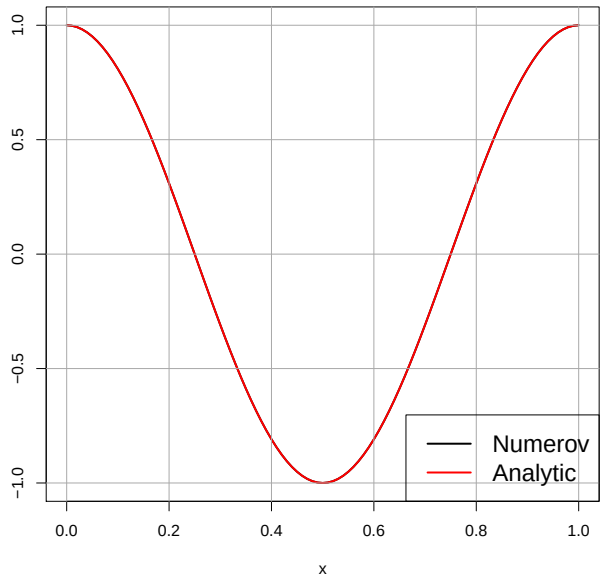

Figure 26: Numerov Solution for Classical SHO with $h = 0.01$ using R

4 The Lennard Jones 6-12 Potential Well: Energy Levels and Wave Functions

The Lennard Jones 6-12 potential (energy) can be defined by

$$V(r) = 4\,V_0\,\left[\left(\frac{a}{r}\right)^{12} - \left(\frac{a}{r}\right)^{6}\right], \tag{4.1}$$

in which a is an adjustable length, V_0 is an adjustable energy and $r \geq 0$. We used the quasi-classical WKB method in Example 1 of this series to estimate the lowest molecular energy levels associated with this potential.

We define a dimensionless coordinate $\tilde{x} = r/a$, a dimensionless potential energy $\tilde{V} = V/V_0$ and energy $\tilde{E} = E/V_0$ and a dimensionless wave function $\tilde{\psi}(\tilde{x}) = \sqrt{a}\,\psi(x)$, in terms of which we have a dimensionless potential (energy)

$$\tilde{V}(\tilde{x}) = 4\,\left[\frac{1}{\tilde{x}^{12}} - \frac{1}{\tilde{x}^{6}}\right] \tag{4.2}$$

and Schroedinger's equation takes the form

$$\frac{d^2\,\tilde{\psi}(\tilde{x})}{d\,\tilde{x}^2} + \gamma^2\,\left(\tilde{E} - \tilde{V}(\tilde{x})\right)\,\tilde{\psi}(\tilde{x}) = 0, \tag{4.3}$$

in which the dimensionless parameter

$$\gamma = \left[\frac{2\,m\,a^2\,V_0}{\hbar^2}\right]^{1/2}. \tag{4.4}$$

The wave function normalization condition becomes

$$\int_0^\infty \tilde{\psi}(\tilde{x})^2\,d\,\tilde{x} = 1. \tag{4.5}$$

In Example 1 we used the case $\gamma = 50$, which we will also use here.

In the following, we omit the tildes and use $y(x)$ to represent $\tilde{\psi}(\tilde{x})$.

4.1 The Numerov Method Using Maxima

The file **LJ6-12.mac** contains a group of Maxima functions designed to explore the energy levels and wave functions associated with a quantum particle in the Lennard-Jones 6/12 potential introduced in the previous section. The dimensionless parameter γ takes on the same value as used for our quasi-classical limit approach in Example 1 of this series, and **gam2** in our code represents γ^2.

The dimensionless potential (energy) $V(x)$ does not depend on the value of γ and we can make a simple plot and explore its shape. After defining a function which is based on the form of $V(x)$, we define the value of x, called **xm**, where $V(x)$ takes on its minimum value, and show that $V(xm) = -1$, $V(1) = 0$, and $V(x)$ approaches large positive values as x approaches 0, and small negative values for x very large. Recall that now x represents a non-negative dimensionless number.

```
(%i1) V(z) := ( 4*(z^(-12) - z^(-6)) )$
(%i2) xm : float(2^(1/6));
(%o2) 1.122462048309373
(%i3) V(xm);
(%o3) -1.0
(%i4) V(1);
(%o4) 0
(%i5) V(0.5);
(%o5) 16128.0
(%i6) V(3);
(%o6) -2912/531441
(%i7) float(%);
```

```
(%o7) -0.0054794417442388
(%i8) V(0.1);
(%o8) 3.9999959999999946E+12
(%i9) V(10.0);
(%o9) -3.9999959999999997E-6
%i10) limit(V(x),x,inf);
(%o10) 0
(%i11) limit( V(x),x,0,plus );
(%o11) inf
(%i12) plot2d([ [discrete,[ [0.8,0],[2,0]]], V(x)],
          [x,0.8,2],[y,-1.5,2], [xlabel,"x"],[ylabel,"V(x)"],
               [style, [lines,3]],[legend,false],[gnuplot_preamble,"set grid"])$
plot2d: some values were clipped.
```

which produces the dimensionless potential (energy) plot:

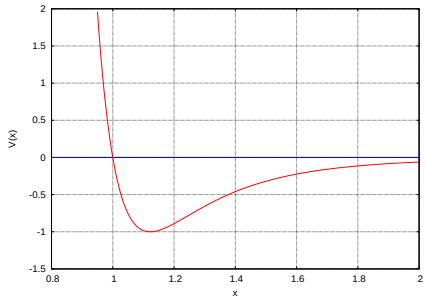

Figure 27: Dimensionless Lennard Jones Potential (Energy)

The bound state energy eigenvalues lie in the range $-1 < E < 0$ and the classical turning points are defined by the equation $E = V(x)$. Setting $y = x^6$, one obtains a quadratic equation in y which is easily solved for y_{tp}. One can then obtain the classical turning points as a function of E from $x_{tp} = y_{tp}^{1/6}$. Bearing in mind that $E < 0$, we can write the turning points in Maxima code as

```
(%i13) xin(E) :=  xm*(sqrt(E+1)/E-1/E)^(1/6)$
(%i14) xout(E) := xm*(sqrt(E+1)+1)^(1/6)/(-E)^(1/6)$
```

and then one can add a hypothetical energy level line to our potential energy plot:

```
(%i15) plot2d([ [discrete,[ [0.8,0],[2,0]]], V(x),
          [discrete, [ [xin(-0.5),-0.5],[xout(-0.5),-0.5] ]] ],
          [x,0.8,2],[y,-1.5,2], [xlabel,"x"],[ylabel,"V(x)"],
               [style, [lines,3]],[legend,false],[gnuplot_preamble,"set grid"])$
plot2d: some values were clipped.
```

which produces the plot

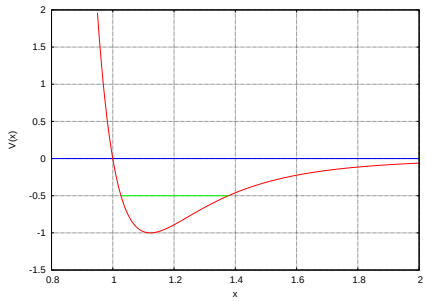

Figure 28: Adding a Hypothetical Energy Level

Loading in the file **LJ6-12.mac** defines a set of global parameters and functions defined at the top of the file:

```
/*     initial global parameters:      */

(  h : 0.01,
   h2 : h^2/12,    /* Numerov constants */
   h52 : 5*h2,
   gam2 : 2500,  /* square of gam = 50  */
   x1decay : 0.3,   /*  start yL integration at (x1 - x1decay)   */
   x2decay : 1,    /* start yR integration at (x2 + x2decay)   */
   y2left : 1e-19,     /*  y(x_left + h)  value chosen from E = -0.9 case */
   y2right : 1e-16,  /*      y(x_right - h)  value chosen from E = -0.9 case  */
   print(" gam2 = ", gam2),
   print(" h = ", h, ",   x1decay = ", x1decay, ",  x2decay = ", x2decay),
   print(" y2left = ",y2left, "   y2right = ", y2right ),
   xm : float(2^(1/6)),   /* this is where V(x) =  -1 = minimum value  */
   /* for given energy -1 < E < 0 , these are the turning points  */
   xin(E) :=  xm*(sqrt(E+1)/E-1/E)^(1/6),
   xout(E) := xm*(sqrt(E+1)+1)^(1/6)/(-E)^(1/6),
   /* dimensionless potential V(x) for Lennard-Jones 6/12 potential */
   V(z) := ( 4*(z^(-12) - z^(-6)) ) )$
```

The grid size **h** and two Numerov method constants **h2** and **h52** which depend on **h** are then globally available. We call the left classical turning point **x1** and we start the rightward Numerov integration at position **x1 - x1decay** with $y = 0$. Likewise we call the right classical turning point **x2** and start the leftward Numerov integration at position **x2 + x2decay** with $y = 0$.

```
(%i1) load(cp3);
(%o1) "c:/k3/cp3.mac"
(%i2) load("LJ6-12.mac");
 gam2 =  2500
 h =  0.01 ,   x1decay =  0.3 ,  x2decay =  1
 y2left =  1.0E-19    y2right =  1.0E-16
(%o2) "c:/k3/LJ6-12.mac"
```

Let us ignore, at first, some slight refinements we have in the code, and give a simplified version.

We first define a grid point **x2c** which is close to the right classical turning point **x2**.

We next generate a list called **yL** which contains the values of $y(x)$ from the grid point **x_left : x1 - x1decay**, with $y(x_{left}) = 0$, and $y(x_{left} + h) =$ **y2left**, and further points generated using the Numerov method, continuing to the grid point **x2c + h**.

Next we generate a list called **yR** which contains the values of $y(x)$ from **x2 + x2decay**, with $y(x_{right}) = 0$, and $y(x_{right} - h) =$ **y2right**, and further points generated using the Numerov method, continuing to the grid point **x2c - h**.

We then multiply all values of **yL** by a common factor which ensures that **yL** and **yR** agree on the value of **y(x = x2c)**.

The **x** grid values **xL** and **xR**, and the corresponding **y(x)** grid values contained in the lists **yL** and **yR** are available as global quantities.

Here is code for **wf(E)** from **LJ6-12.mac** that generates the un-normalized wave functions in the form of the global lists **xL**, **yL**, **xR**, **yR**. In the function **wf(E)** we use the hashed arrays **xl**, **yl**, **xr**, and **yr**. Their names are included in a **local** statement of the form **local(g,xl,yl,xr,yr)**, so that the hashed arrays are not available at the global level. At the end of **wf(E)** we have statements such as **xL : listarray(xl)** and **yL : listarray(yl)**, which make the contents of the hashed arrays available as ordinary lists globally.

```
/*  wf(E)    creates ** un-normalized **  wave functions
     for the Lennard-Jones 6/12 potential case.
     limits of numerical integration, x_left and x_right are
     determined by energy E and xdecay values.
     The wave functions are stored in global xL, yL, xR, yR.
     Program also defines **global**  x2c, nleft, nright, x_left, x_right.
     See example run at end.

     nleft = the number of steps from x_left to x2c =  grid point nearest to
     x1 = xin(E) =  classical turning point < xm = 1.122462
     x2 = xout(E) =  classical turning point > xm.
     the global xL grid extends from x_left to x2c + h   and
     the global  xR grid extends from  x2c - h to   x_right,
     so we can compute y'(x2c) using a 3 pt. symmetric formula.
*/

wf(E) := block( [x1,x2,x, fac,numer],numer:true,
     if (E > 0) or (E < -1) then (
         print(" need -1 < E < 0 "),
         return(false)),
     local(g,xl,yl,xr,yr),
     g(z) :=  gam2*( E - V(z) ),  /*  coeff. func. in ode:  y''(x) + g(x) y(x) = 0  */
     x1 : xin(E),  /* classical turning point for x < xm   */
     x2 : xout(E),  /* classical turning point for x> xm   */
     if wfdebug then print("  x1 = ", x1, "   x2 = ", x2),
     x_left : x1 - x1decay,
     nleft : round ( (x2 - x_left)/h ),  /* number of steps from x_left to x2c = match point */
     x2c : x_left + h*nleft,
     if wfdebug then print(" x_left = ",x_left," nleft = ",nleft,"  x2c = ", x2c),
     if wfdebug then print(" y2left = ", y2left, " y2right = ", y2right),
     nright : round ( (x2 + x2decay - x2c)/h ),   /* number of steps from x2c to x_right  */
     x_right : x2c + h*nright,
     if wfdebug then print("  nright = ",nright," x_right = ",x_right),
     /* find yL for x_left <=  x <= x2c + h using Numerov algorithm  */
     xl[1] : x_left,
     xl[2] : x_left + h,
     yl[1] : 0,
     yl[2] : y2left,
     for j:2 thru nleft + 1 do (
         x : x_left + j*h,
         xl[j+1] : x,
         yl[j+1] : ( 2*(1 - h52*g(x-h)) * yl[j] - (1 + h2*g(x-2*h)) * yl[j-1] )/ (1 + h2*g(x)) ),
     /* find yR for  x2c - h  <= x  <= x_right using Numerov method */
     xr[nright + 2 ] : x_right,
     xr[nright + 1] : x_right - h,
     yr[nright + 2] : 0,
     yr[nright + 1] : y2right,
     for j:nright+1  step -1  thru 2 do (
         x : x2c + h*(j -3),
         xr[j-1] : x,
         yr[j-1] : (2*(1-h52*g(x+h))*yr[j]  - (1 + h2*g(x+2*h))*yr[j+1] )/(1 + h2*g(x)) ),
     fac : yr[2]/yl[nleft + 1],     /*   yR(x2c) / yL(x2c)    */
     for j thru nleft+2 do   yl[ j ] : fac * yl[ j ],
     xL : listarray(xl),
     yL : listarray(yl),
     xR : listarray(xr),
     yR : listarray(yr),
     done)$
```

An example of the use of **wf(E)** for a randomly chosen energy is

```
(%i3) wf(-0.5);
(%o3) done
(%i4) fll(xL);
(%o4) [0.726743,1.3867425,67]
(%i5) fll(xR);
(%o5) [1.3667425,2.3767425,102]
(%i6) head(yL);
(%o6) [0,1.75654725E-29,-3.97814273E-28,1.06730999E-26,-3.66490209E-25,
       1.85515641E-23]
(%i7) tail(yL);
(%o7) [1.46282504E-4,4.50817194E-4,7.49059762E-4,0.00104033,0.00132673,0.00161295]
(%i8) dy_diff();
(%o8) 33.121756
(%i9) num_nodes();
(%o9) 3
```

The function **dy_diff()** used above is designed to return the difference of the approximate numerical first derivatives at the matching point **x2c** implied by **yL** and **yR**, divided by the value of **y(x2c)**. Here is our code for **dy_diff()**.

```
/*  dy_diff() uses global yL, yR,nleft, h
     computes numerical y'(x2c)  using
     symmetric three point method for
     both yL and yR, and returns the difference
     divided by y(x2c)
*/

dy_diff() :=
block([ypL, ypR,  numer],numer:true,
     ypL : ( last(yL) - yL[nleft] ) / (2*h),
     ypR : (yR[3] - first(yR)) / (2*h),
     (ypL - ypR)/ abs (yR[2]) )$
```

The function **num_nodes()** has the definition:

```
/*  count the number of nodes in yL
     ignore region where elements of yL are
     tiny in magnitude.
     uses position(...)   */

num_nodes() :=
block([x11, j0, yLm2, n, numer], numer:true,
     x11 : x_left + x1decay,
     j0 : position(x11, xL),
     yLm2 : rest(yL,-2),  /* ignore y(x2c) and y(x2c+h) values in count */
     n : 0,
     for j : j0  thru (length(yLm2) - 1) do
         if yLm2[j] * yLm2[j + 1] < 0 then n : n + 1,
     n)$
```

and uses **position(...)**:

```
/* position(xv, aL) is designed to be used with xL to locate
    position of first element which is equal to or greater than x1,
    since in this package xL has just increasing positive numbers in it
*/

position(xv, aL) := ( first (sublist_indices(aL,lambda([x], x >= xv))))$
```

We search for energy eigenvalues by seeking energies **E** such that the value returned by **dy_diff()** is zero to within numerical errors. A function **F(E)** allows us to scan energy ranges for energy eigenvalues.

```
/* energy eigenvalue if global function F(E) = 0 .
     F(E) calls wf(E) then returns dy_diff(), but
     returns false if E > 0.
 */

F(E) :=
block( [ numer],numer:true,
     if E > 0 then (
         print(" in F(E), E = ",E,"  should be negative "),
         return(false)),
     wf(E),
     dy_diff())$
```

Here is an example of using **F(E)**.

```
(%i10) EL : makelist(e,e,-0.91,-0.85,0.01);
(%o10) [-0.91,-0.9,-0.89,-0.88,-0.87,-0.86,-0.85]
(%i11) FL : map(F,EL);
(%o11) [7.199669,2.035782,-5.228712,-24.50145,-58.60732,232.5781,61.42046]
(%i12) F(-0.91);
(%o12) 7.1996689
(%i13) F(-0.88);
(%o13) -24.501452
(%i14) e : find_root(F, -0.91,-0.88);
(%o14) -0.896404
```

A function **wf_plot(E)** generates a non-normalized numerical solution using **wf(E)**, makes a plot and prints out the energy and maximum y value, the number of nodes, and the value of **dy_diff()** corresponding to the chosen energy **E**.

```
(%i15) wf_plot(e);
 E =  -0.896404 ,      ymax =  27.422334
 number of nodes =  0 ,      dy_diff =  7.94875172E-14
```

which produces the plot

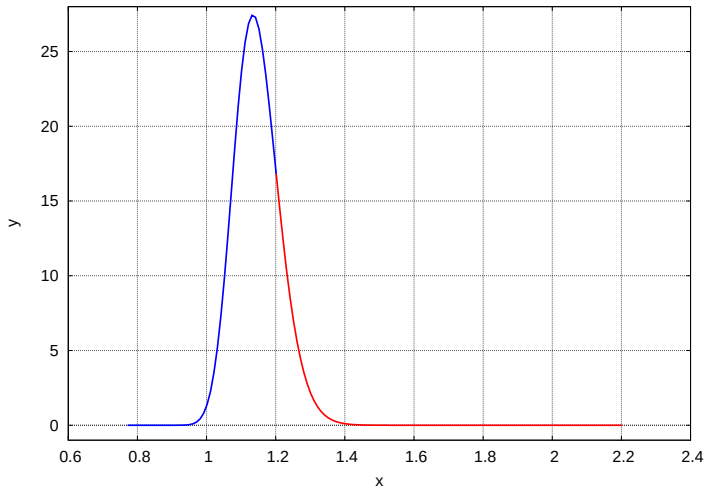

Figure 29: Zero Nodes Un-normalized Eigenfunction

The code for **wf_plot(E)** is

```
/*
     wf_plot (E) calls wf(E) and plot2d
     creates ** un-normalized **  wave functions
     stored in global xL, yL, xR, yR.
     prints out number of nodes in yL
     prints out dy_diff .
*/


wf_plot(E) :=
block([xxL, yyL, ymn,ymx, numer], numer:true,
     wf(E),
     xxL : rest(xL, -1),
     yyL : rest(yL, -1),
     ymn : float( floor ( lmin(yyL))),
     ymx : float(1 + floor ( lmax(yyL))),
     print( " E = ", E, ",      ymax = ", lmax(yyL) ),
     print(" number of nodes = ",num_nodes(), ",      dy_diff = ",dy_diff() ),
     plot2d([ [discrete, xxL, yyL],  [discrete, rest(xR,1), rest(yR,1)] ],
         [y,ymn,ymx],  [ylabel,"y"],  [xlabel,"x"], [style, [lines,3]],
         [legend, false],  [gnuplot_preamble,"set grid"]))$
```

We then create (from **yL** and **yR**) a global normalized wave function list **yn** corresponding to a global grid list **xn** created from **xL** and **xR** using the function **normalize()**. This function also computes and prints the value of the quantum mechanical particle position uncertainty Δx implied by the wave function.

```
(%i16) normalize();
 AA =  85.006868
 x_mean =  1.1406875
 delx =  0.0455039
(%o16) done
(%i17) fll(xn);
(%o17) [0.781455,2.2014549,143]
(%i18) fll(yn);
(%o18) [-3.14183197E-20,0,143]
(%i19) lmax(yL);
(%o19) 27.422334
(%i20) lmax(yn);
(%o20) 2.9742495
```

The function **normalize** uses our utility functions **simp** (Simpson's one third integration rule) and **merge**.

```
/* normalize() uses the current global xL,yL, xR, yR and
     the  utility functions  merge and simp  to define global
     xn and yn, with the latter being normalized.
  */

normalize() :=
block ( [AA,x_mean,x2_mean,delx,delx2, numer ], numer:true,
     xn : merge( rest(xL,-1), rest(xR, 2) ),
     yn : merge( rest(yL,-1), rest(yR, 2) ),
     /* we need xn to have odd # of elements to use simp */
     if evenp ( length (xn) ) then (
         xn : rest (xn),
         yn : rest (yn)),
     AA : simp(xn,yn^2),
     print( " AA = ",AA),
     yn : yn/sqrt(AA),
     x_mean : simp(xn, xn * yn^2),
     print(" x_mean = ", x_mean),
     x2_mean : simp(xn, xn^2 * yn^2),
     delx2 : x2_mean - x_mean^2,   /* this should be positive!  */
```

```
    delx : sqrt(delx2),
    print(" delx = ", delx),
    done)$
```

Once **normalize()** has been used to create **xn** and **yn** from the current un-normalized wave function, we can use **yn_plot_current()** to see the current normalized wave function.

```
(%i21) yn_plot_current()$
 ymax =  2.9742495
```

which produces the plot

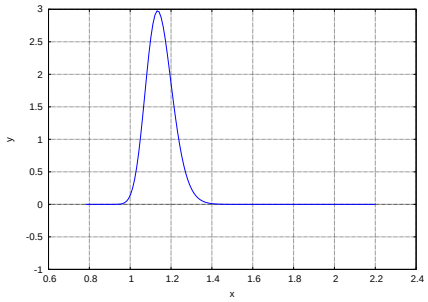

Figure 30: Zero Node Normalized Eigenfunction

A function **yn_plot(E, xmin, xmax)** goes from a given energy **E** to a call to **wf(E)** and **normalize()** and then makes a plot of the resulting normalized wave function in one step, with control over the region of the **x** axis for the plot. Thus

```
(%i22) yn_plot(e,0.8, 1.6)$
 E =  -0.896404
 number of nodes =  0 ,       dy_diff =  7.94875172E-14
 AA =  85.006868
 x_mean =  1.1406875
 delx =  0.0455039
 normalized ymax =  2.9742495
```

produces the plot

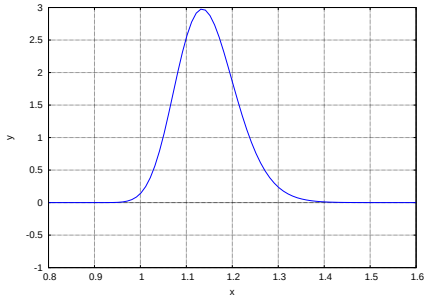

Figure 31: Using yn_plot(E,xmin,xmax) for $E = -0.896404$

Here is an example of using **yn_plot** for an energy which is not an energy eigenvalue. The value of **dy_diff()** reported is based on the non-normalized wave function produced by **wf(E)**.

```
(%i23) yn_plot(-0.95,0.8, 1.6)$
 E =  -0.95
 number of nodes =  0 ,       dy_diff =  16.398098
 AA =  139.33333
 x_mean =  1.1545679
 delx =  0.0420213
 normalized ymax =  3.2673444
```

which produces the plot

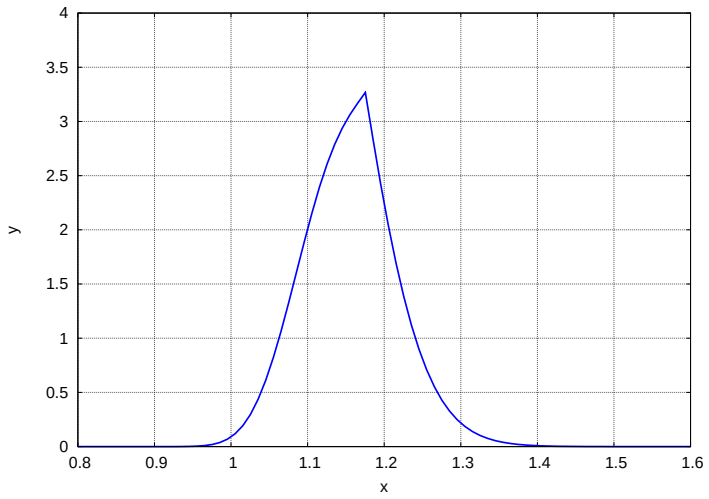

Figure 32: Using yn_plot(E,xmin,xmax) for $E = -0.95$

We see a large discontinuity in the slope of the normalized wave function, reflected also in the large value of **dy_diff()** reported in the non-normalized wave function.

Here is the code for **yn_plot**:

```
yn_plot(E,xmn,xmx) :=
block([ymn, ymx, numer],numer:true,
     wf(E),
     print(" E = ",E ),
     print(" number of nodes = ",num_nodes(),",       dy_diff = ",dy_diff() ),
     normalize(),
     print(" normalized ymax = ", lmax(yn) ),
     ymn : floor( lmin(yn) ),
     ymx : 1 + floor( lmax (yn) ),
     plot2d( [discrete, xn, yn], ['x,xmn, xmx],
         ['y,ymn,ymx],  [ylabel,"y"],  [xlabel,"x"],
         [style, [lines, 3] ], [legend, false],  [gnuplot_preamble,"set grid"]))$
```

A plot of the values of **F(E)** over a wider energy range will show other candidate energies for excited states having energies greater than the ground state (zero node state with energy $E_0 = -0.896404$ found above). However, use of the function **bracket(Estart,dE,Eacc)** is an easier way to find candidate energy eigenvalues. **bracket** looks for the first sign change in **F(E)**, and (in a normal exit) returns a pair of energies for which **F(E)** has the opposite sign. Applying this approach to the ground state energy found above,

```
(%i23) [ea,eb]:bracket(F,-0.96,0.02,0.01);
(%o23) [-0.9,-0.895]
(%i24) e: find_root(F,ea,eb);
(%o24) -0.896404
(%i25) yn_plot(e,0.8,1.6)$
 E =  -0.896404
 number of nodes =  0 ,       dy_diff =  -1.33606678E-13
 AA =  85.006868
 x_mean =  1.1406875
 delx =  0.0455039
 normalized ymax =  2.9742495
```

which results in the same plot as we displayed above for the normalized ground state.

Here is our code for **bracket**:

```
/* this function implementation of bracket, designed to work with
    the package LJ6-12.mac,  looks for a sign change in func,
    starting with xx,  and increasing xx by dxx each step.
    If sign change is found,  then we back up to the previous xx
    and search with new  dxx value one half of the previous value.
    normally returns [ea,eb], but if can't find change in sign,
    then returns [0,0], and if func returns false, then
    bracket returns false.
*/

bracket(func,xx,dxx,xacc) :=
block([f1,f2, x:xx, dx:dxx,xx1,xx2,it:0,itmax:1000],
do (
    it : it + 1,
    if debug then print(it),
    if it > itmax then (
        print(" can't find change in sign "),
        return([0, 0 ])),
    xx1 : x,
    xx2 : x + dx,
    f1 : func(xx1),
    if not f1 then (
        print(" in bracket, f1 = false , xx1 = ",xx1, "  dx = ", dx),
        return(f1)),
    f2 : func(xx2),
    if not f2 then (
        print(" in bracket, f2 = false ,  xx2 = ",xx2, "  dx = ", dx),
        return(f2)),
    if f1*f2 < 0 then (
        if abs(dx) < xacc then return([xx1,xx2]),
        x : x - dx,
        dx : dx/2)
    else x : xx2))$
```

Let's use **bracket** to find a candidate energy for the first excited state, which should have one node and be a continuous function.

```
(%i26) [ea,eb]:bracket(F,e + 0.01,0.02,0.01);
(%o26) [-0.866404,-0.861404]
(%i27) e: find_root(F,ea,eb);
(%o27) -0.865689
(%i28) yn_plot(e,0.8,1.6)$
 E =  -0.865689
 number of nodes =  0 ,       dy_diff =  -1.17696426E+16
 AA =  5.59079278E+30
 x_mean =  1.1192306
 delx =  0.0350225
 normalized ymax =  3.2891799
```

which produces the plot

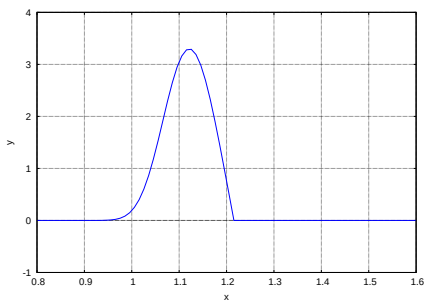

Figure 33: False Energy Eigenvalue for $E = -0.865689$

The plot shows a discontinuous zero node wave function and the reported value of **dy_diff()** is not a tiny number, as **dy_diff()** should be for a valid energy eigenvalue case. We also note that we had already found a valid zero node energy eigenvalue.

Continuing with a search for energy eigenvalues using **bracket**:

```
(%i29) [ea,eb]:bracket(F,e + 0.01,0.02,0.01);
(%o29) [-0.710689,-0.705689]
(%i30) e : find_root(F,ea,eb);
(%o30) -0.71066
(%i31) yn_plot(e,0.8,1.6)$
 E =  -0.71066
 number of nodes =  1 ,       dy_diff =  8.68128554E-15
 AA =  0.0295798
 x_mean =  1.1806289
 delx =  0.0807403
 normalized ymax =  2.6000666
```

which produces a valid one node wave function

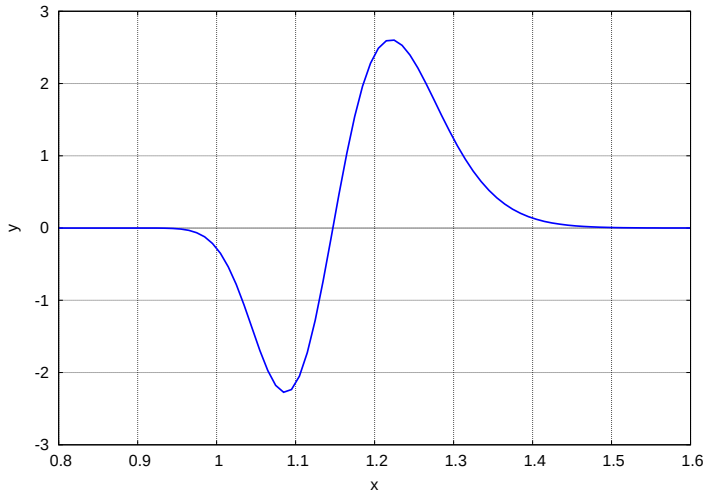

Figure 34: One Node Wave Function for $E = -0.71066$

This one node wave function is continuous and the reported value of **dy_diff()** is a tiny number.

We will again find a spurious one node solution at a slightly higher energy. However, we must reduce the size of the **dE** argument to **bracket**.

```
(%i32) [ea,eb]:bracket(F,e + 0.01,0.02,0.01);
(%o32) [-0.71066,-0.70566]
(%i33) [ea,eb]:bracket(F,e + 0.02,0.02,0.01);
(%o33) [-0.71066,-0.70566]
(%i34) [ea,eb]:bracket(F,e + 0.02,0.01,0.005);
(%o34) [-0.68566,-0.68316]
(%i35) e : find_root(F,ea,eb);
(%o35) -0.684654
(%i36) yn_plot(e,0.8,1.6)$
 E =  -0.684654
 number of nodes =  1 ,       dy_diff =  4.23941273E+16
 AA =  3.63949121E+28
 x_mean =  1.1531148
 delx =  0.0706492
 normalized ymax =  2.5594982
```

which shows a spurious one node wave function (note the very large value of **dy_diff**):

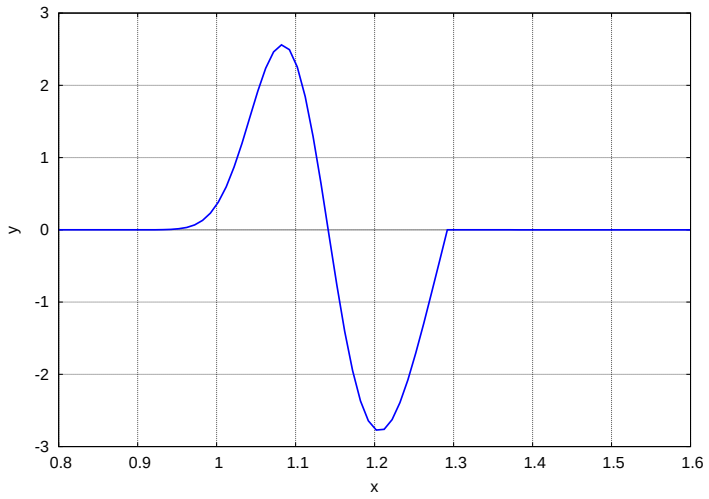

Figure 35: False Energy Eigenvalue for $E = -0.684654$

Now that we see the pattern of valid energy eigenvalues, we can write a function **levels(Emin,Emax,dE, Eacc)**, which uses **bracket** with the added filter that no sign change of **F(E)** is taken seriously unless the associated wave function has one more node than the last energy eigenvalue found.

```
/* levels(Emin,Emax,dE, Eacc ) returns a list  [Ea, Eb,...] of energy levels with
        increasingly larger number of nodes in energy range (Emin, Emax).
    uses F(E) to find  roots, and calls wf, num_nodes() and dy_diff() for each root found,
    Uses bracket and find_root.
    The arguments (dE, Eacc) are used to call bracket, and do not describe
         the accuracy of the energy levels found.
    Once a good energy e.v. is found we look for the region of
         energies with one more node and search there.
    Code inclues an interactive continue or stop prompt.
*/


levels(Emin,Emax,dE, Eacc ) :=
block([ e,enext, eroot, eL, ea, eb, nn, nlast : -1, r, numer], numer:true,
    e : Emin,
    eL : [ ],     /*    list eL will hold energy eigenvalues found  */
    do (  if e > Emax then return(),    /* exit do loop  */
        print("---------------- levels -------------------"),
        print(" nlast = ", nlast),
        print(" Estart = ", e," dE = ", dE ),
        [ea, eb] : bracket(F,e, dE, Eacc),
        print(" ea = ",ea," eb = ",eb),
        if float(ea) = 0.0 then (
            print(" can't find bracket interval "),
            print(" e = ",e),
            return() ),
        eroot : find_root(F, ea, eb),
        print(" eroot = ", eroot),
        wf(eroot),
        nn : num_nodes(),
        print(" number of nodes = ", nn),
        print(" dy_diff at x2c  = ", dy_diff() ),
        eL : cons(eroot, eL),
        nlast : nn,
        r : read (" input c; or s; "),
        if string(r) = "s" then return(),        /* exit do loop  */
```

```
        /* search for an energy greater than eb which produces
                       a wave function with nn + 1 nodes */
        enext : eb + dE,
        do (
            wf(enext),
            if num_nodes() > nlast then (
                e : enext,
                return() )
            else enext : enext + dE)),
    reverse(eL) )$
```

Here is an example of use:

```
(%i37) levels(-0.95, -0.6, 0.02,0.01);
---------------- levels ------------------
 nlast =  -1
 Estart =  -0.95  dE =  0.02
 ea =  -0.9  eb =  -0.895
 eroot =  -0.896404
 number of nodes =  0
 dy_diff at x2c  =  -5.41191607E-14
 input c; or s;
c;
---------------- levels ------------------
 nlast =  0
 Estart =  -0.835  dE =  0.02
 ea =  -0.715  eb =  -0.71
 eroot =  -0.71066
 number of nodes =  1
 dy_diff at x2c  =  1.73625711E-14
 input c; or s;
c;
---------------- levels ------------------
 nlast =  1
 Estart =  -0.67  dE =  0.02
 ea =  -0.555  eb =  -0.55
 eroot =  -0.551436
 number of nodes =  2
 dy_diff at x2c  =  -7.17260262E-15
 input c; or s;
c;
(%o37) [-0.896404,-0.71066,-0.551436]
(%i38) [E0,E1,E2] : %;
(%o38) [-0.896404,-0.71066,-0.551436]
```

We then can use **`yn_plot`** for the ground state energy to create the normalized wave function and make a plot, and using **`xyn0 = makelist(...)`** to save these wave function definitions under a unique name.

```
(%i39) yn_plot(E0,0.8,1.6)$
 E =  -0.896404
 number of nodes =  0 ,       dy_diff =  -5.41191607E-14
 AA =  85.006868
 x_mean =  1.1406875
 delx =  0.0455039
 normalized ymax =  2.9742495
(%i40) xyn0 : makelist([xn[j],yn[j]],j,1,length(xn))$
(%i41) fll(xyn0);
(%o41) [[0.781455,-3.14183197E-20],[2.2014549,0],143]
```

We can then continue with the first and second excited states.

```
(%i42) yn_plot(E1,0.8,1.6)$
 E =  -0.71066
 number of nodes =  1 ,       dy_diff =  1.98222687E-13
 AA =  0.0295798
 x_mean =  1.1806289
 delx =  0.0807403
 normalized ymax =  2.6000666
(%i43) xyn1 : makelist([xn[j],yn[j]],j,1,length(xn))$
```

```
(%i44) fll(xyn1);
(%o44) [[0.754761,2.83787145E-25],[2.2747607,0],153]
(%i45) yn_plot(E2,0.8,1.6)$
 E =  -0.551436
 number of nodes =  2 ,       dy_diff =  2.03223741E-13
 AA =  1.24980723E-5
 x_mean =  1.2265982
 delx =  0.108671
 normalized ymax =  2.4452266
(%i46) xyn2 : makelist([xn[j],yn[j]],j,1,length(xn))$
(%i47) fll(xyn2);
(%o47) [[0.730536,0],[2.3505359,0],163]
```

We can finally make a plot of the three normalized wave functions corresponding to the lowest three energy levels in the Lennard-Jones potential.

```
(%i48) plot2d([[discrete,xyn0],[discrete,xyn1],[discrete,xyn2]],
          [x,0.8,1.6],[xlabel,"x"],[ylabel,"y"],[style,[lines,2]],
           [legend,"E0","E1","E2"])$
```

which produces the plot

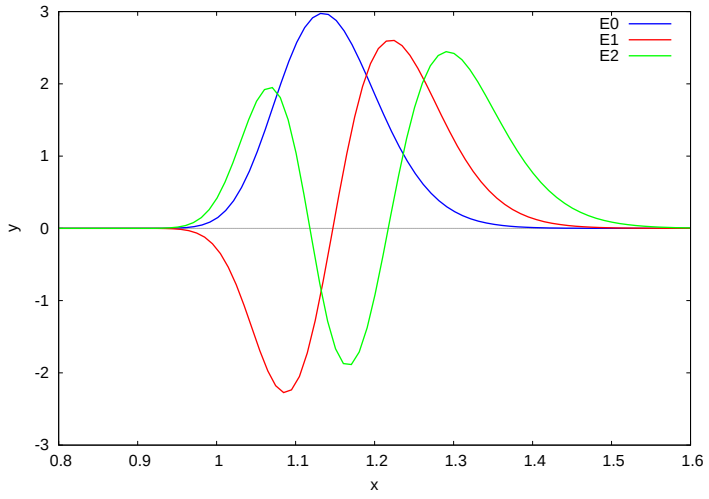

Figure 36: Lennard-Jones: Wave Functions for Lowest Three Energy Levels

As we did with the finite potential well case, we can save the energies and wave functions using **`save(filename,a1,a2,...)`** for use in a later, different Maxima session.

```
(%i49) save("c:/k3/LJ1.dat",E0,xyn0,E1,xyn1,E2,xyn2);
(%o49) "c:/k3/LJ1.dat"
```

The top of this data file, viewed with a text editor, is written in Lisp, and looks like:

```
;;; -*- Mode: LISP; package:maxima; syntax:common-lisp; -*-
(in-package :maxima)
(DSKSETQ |$e0| -0.89640379037496043)
(ADD2LNC '|$e0| $VALUES)
(DSKSETQ $XYN0
         '((MLIST SIMP)
           ((MLIST SIMP) 0.78145488500192117 -3.1418319670117798E-20)
           ((MLIST SIMP) 0.79145488500192118 7.7629000010676559E-18)
           ((MLIST SIMP) 0.80145488500192119 8.9446105712810272E-16)
           ((MLIST SIMP) 0.8114548850019212 3.8907747472815292E-14)
etc., etc., ...
```

If we start a new session of Maxima, we can use **load** to load in these saved definitions, and continue to use them as usual.

```
(%i1) load(cp3);
(%o1) "c:/k3/cp3.mac"
(%i2) load("LJ6-12.mac");
 gam2 =  2500
 h =  0.01 ,   x1decay =  0.3 ,  x2decay =  1
 y2left =  1.0E-19    y2right =  1.0E-16
(%o2) "c:/k3/LJ6-12.mac"
(%i3) load("LJ1.dat");
(%o3) "c:/k3/LJ1.dat"
(%i4) values;
(%o4) [mydate,_binfo%,h,h2,h52,gam2,x1decay,x2decay,y2left,y2right,xm,E0,xyn0,
       E1,xyn1,E2,xyn2]
(%i5) E0;
(%o5) -0.896404
(%i6) fll(xyn0);
(%o6) [[0.781455,-3.14183197E-20],[2.2014549,0],143]
(%i7) plot2d([[discrete,xyn0],[discrete,xyn1],[discrete,xyn2]],
           [x,0.8,1.6],[xlabel,"x"],[ylabel,"y"],[style,[lines,2]],
            [legend,"E0","E1","E2"])$
```

and we get the same plot as we did in the previous session.

A proper exploration of the likely accuracy of the energy levels found in this approach would involve experimenting with the values of **x1decay**, **x2decay**, and the grid size **h**. One can modify the code so that the use of a five point symmetric formula for the first derivative is used, instead of the present three point symmetric formula.

Increased integration accuracy can also be sought by writing a Numerov integration routine which uses big float arithmetic in the Maxima language to use 20 digit arithmetic, for example, rather than the default 16 digit arithmetic.

4.2 The Numerov Method Using R

The file **LJ6-12.R** contains a group of R functions designed to explore the energy levels and wave functions associated with a quantum particle in the Lennard-Jones 6/12 potential, using R. The dimensionless parameter $\gamma = 50$ takes on the same value as used for our quasi-classical limit approach in Example 1 of this series, and **gam2** in our code represents γ^2.

The dimensionless potential (energy) $V(x)$ does not depend on the value of γ and we can make a simple plot and explore its shape. After defining a function which is based on the form of $V(x)$, we define the value of x, called **xm**, where $V(x)$ takes on its minimum value, and show that $V(xm) = -1$, $V(1) = 0$, and $V(x)$ approaches large positive values as x approaches 0, and small negative values for x very large. Recall that now x represents a non-negative dimensionless number.

```
> V  =  function (x)     4*(x^(-12) - x^(-6))
> xm = 2^(1/6); xm
[1] 1.12246
> V(xm)
[1] -1
> V(1)
[1] 0
> V(0.5)
[1] 16128
> V(3)
[1] -0.00547944
> V(0.1)
[1] 4e+12
> curve(V,0.8,2,lwd=3,col="red",ylim = c(-1.5,2),xlab="x",ylab="y(x)")
> lines(c(0.8,2),c(0,0),lwd=3,col="blue")
> mygrid()
```

which produces the dimensionless potential (energy) plot:

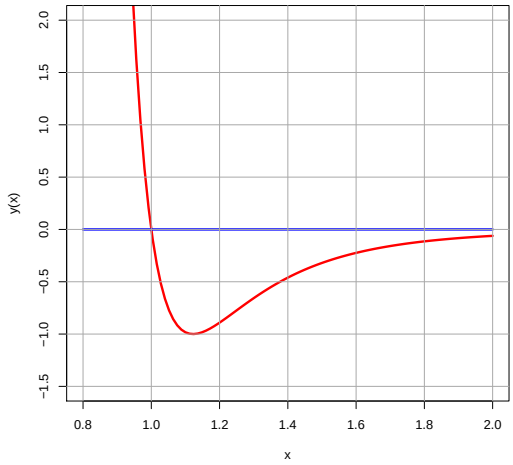

Figure 37: Dimensionless Lennard Jones Potential (Energy)

The bound state energy eigenvalues lie in the range $-1 < E < 0$ and the classical turning points are defined by the equation $E = V(x)$. Setting $y = x^6$, one obtains a quadratic equation in y which is easily solved for y_{tp}. One can then obtain the classical turning points as a function of E from $x_{tp} = y_{tp}^{1/6}$. Bearing in mind that $E < 0$, we can write the turning points in R code as

```
> xin =  function (E)    xm*(sqrt(E+1)/E-1/E)^(1/6)
> xout  =  function (E)    xm*(sqrt(E+1)+1)^(1/6)/(-E)^(1/6)
```

and then one can add a hypothetical energy level line to our potential energy plot:

```
> lines(c(xin(-0.5),xout(-0.5)), c(-0.5,-0.5), lwd = 3, col = "green")
```

which produces the plot

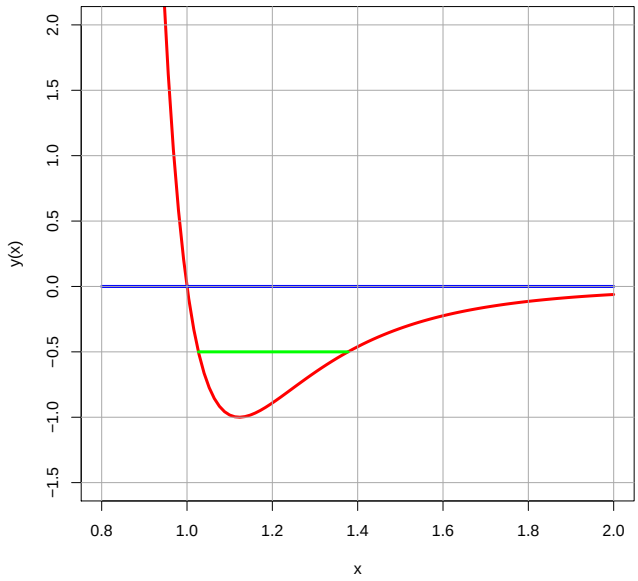

Figure 38: Adding a Hypothetical Energy Level

Loading in the file **LJ6-12.mac** defines a set of global parameters and functions defined at the top of the file:

```
##       initial global parameters :

    h = 0.01
    h2 = h^2/12    ##   Numerov constants
    h52 = 5*h2
    gam2 = 2500   ##   square of gam = 50
    x1decay = 0.3   ##   start yL integration at (x1 - x1decay)
    x2decay = 1    ##   start yR integration at (x2 + x2decay)
    y2left = 1e-19     ##  y(x_left + h)  value chosen from E = -0.9 case
    y2right = 1e-16    ##      y(x_right - h)  value chosen from E = -0.9 case
    cat (" gam2 = ", gam2, "\n")
    cat (" h = ", h, ",   x1decay = ", x1decay, ",  x2decay = ", x2decay, "\n")
    cat (" y2left = ",y2left, "   y2right = ", y2right, "\n" )
    xm = 2^(1/6)      ##     this is where V(x) =  -1 = minimum value
    ##    for given energy -1 < E < 0 , these are the turning points
    xin =  function (E)    xm*(sqrt(E+1)/E-1/E)^(1/6)
    xout  =  function (E)    xm*(sqrt(E+1)+1)^(1/6)/(-E)^(1/6)
    ##    dimensionless potential V(x) for Lennard-Jones 6/12 potential
    V  =  function (x)     4*(x^(-12) - x^(-6))
    wfdebug = FALSE
```

The grid size **h** and two Numerov method constants **h2** and **h52** which depend on **h** are then globally available. We call the left classical turning point **x1** and we start the rightward Numerov integration at position **x1 - x1decay** with $y = 0$. Likewise we call the right classical turning point **x2** and start the leftward Numerov integration at position **x2 + x2decay** with $y = 0$.

```
> source("cp3.R")
> source("LJ6-12.R")
 gam2 =  2500
 h =  0.01 ,   x1decay =  0.3 ,  x2decay =  1
 y2left =  1e-19    y2right =  1e-16
```

Let us ignore, at first, some slight refinements we have in the code, and give a simplified version.

We first define a grid point **x2c** which is close to the right classical turning point **x2**.

We next generate a vector called **yl** which contains the values of $y(x)$ from the grid point **x_left : x1 - x1decay**, with $y(x_{left}) = 0$, and $y(x_{left} + h) =$ **y2left**, and further points generated using the Numerov method, continuing to the grid point **x2c + h**.

Next we generate a vector called **yr** which contains the values of $y(x)$ from **x2 + x2decay**, with $y(x_{right}) = 0$, and $y(x_{right} - h) =$ **y2right**, and further points generated using the Numerov method, continuing to the grid point **x2c - h**.

We then multiply all values of **yl** by a common factor which ensures that **yl** and **yr** agree on the value of **y(x = x2c)**.

The **x** grid values **xl** and **xr**, and the corresponding **y(x)** grid values contained in the vectors **yl** and **yr** are finally made available as the global quantities **xL**, **xR**, **yL**, and **yR** respectively.

Here is code for **wf(E)** from **LJ6-12.R** that generates the un-normalized wave functions in the form of the global vectors **xL**, **yL**, **xR**, **yR**.

```
##  wf(E)    creates ** un-normalized **  wave functions
##    for the Lennard-Jones 6/12 potential case.
##    limits of numerical integration, x_left and x_right are
##    determined by energy E and xdecay values.
##    The wave functions are stored in global xL, yL, xR, yR.
##    Program also defines **global**  x2c, nleft, nright, x_left, x_right.
##    See example run at end.
##    nleft = the number of steps from x_left to x2c =  grid point nearest to
##    x1 = xin(E) =  classical turning point < xm = 1.122462
##    x2 = xout(E) =  classical turning point > xm.
```

```
##    the global xL grid extends from x_left to x2c + h   and
##    the global  xR grid extends from  x2c - h to   x_right,
##    so we can compute y'(x2c) using a 3 pt. symmetric formula.

wf  =  function (E)   {
    if  ((E > 0) | (E < -1))  {
        cat (" need -1 < E < 0   \n")
        return(false)   }
    g  =  function(x) gam2*( E - V(x) )  ## coeff. func. in ode: y''(x) + g(x) y(x) = 0
    x1 = xin(E)      ## classical turning point for x < xm
    x2 = xout(E)     ##  classical turning point for x> xm
    x_left = x1 - x1decay
    nleft = round ( (x2 - x_left)/h )    ##   number of steps from x_left to x2c = match point
    x2c = x_left + h*nleft
    nright =  round ( (x2 + x2decay - x2c)/h )   ##   number of steps from x2c to x_right
    x_right = x2c + h*nright
    ##      find yL for x_left <=  x <= x2c + h using Numerov algorithm
    xl = vector( mode = "numeric",  length = nleft + 2)
    yl = vector( mode = "numeric",  length = nleft + 2)
    xl[1] = x_left
    xl[2] = x_left + h
    yl[1] = 0
    yl[2] = y2left
    for( j  in  2: (nleft + 1) )   {
        x = x_left + j*h
        xl[j+1] = x
        yl[j+1] = (2*(1-h52*g(x-h))*yl[j] - (1+h2*g(x-2*h))*yl[j-1] )/ (1+h2*g(x)) }
    ##     find yR for  x2c - h  <= x  <= x_right using Numerov method
    xr = vector( mode = "numeric",  length = nright + 2)
    yr = vector( mode = "numeric",  length = nright + 2)
    xr[nright + 2 ] = x_right
    xr[nright + 1] =  x_right - h
    yr[nright + 2] = 0
    yr[nright + 1] = y2right
    for ( j  in   (nright+1):2 )  {
        x = x2c + h*(j -3)
        xr[ j - 1] = x
        yr[j - 1] = (2*(1-h52*g(x+h))*yr[j] - (1+h2*g(x+2*h))*yr[j+1] )/(1+h2*g(x)) }
    fac = yr[2]/yl[nleft + 1]       ##     yR(x2c) / yL(x2c)
    yl = fac*yl
    ##  create globally known stuff
    x2c <<-   x2c
    nleft <<-   nleft
    nright <<-   nright
    x_left <<-   x_left
    x_right <<-   x_right
    xL <<-  xl
    yL <<-  yl
    xR <<-  xr
    yR <<-  yr }
```

An example of using **`wf(E)`** for some arbitrarily chosen negative energy:

```
> setwd("c:/k3")
> source("cp3.R")
> source("LJ6-12.R")
 gam2 =  2500
 h =  0.01 ,   x1decay =  0.3 ,  x2decay =  1
 y2left =  1e-19    y2right =  1e-16
> wf(-0.5)
> fll(xL)
  0.726743   1.38674   67
> head(yL)
[1]  0.00000e+00  1.75655e-29 -3.97814e-28  1.06731e-26 -3.66490e-25  1.85516e-23
```

```
> fll(yL)
  0   0.00161295   67
> fll(xR)
  1.36674   2.37674   102
> head(yR)
[1] 0.001479596 0.001326728 0.001173346 0.001024722 0.000884658 0.000755663
> dy_diff()
[1] 33.1218
> num_nodes()
[1] 3
> nleft
[1] 65
> nright
[1] 100
> x_left
[1] 0.726743
> x2c
[1] 1.37674
> plot(0,type="n",xlim=c(min(xL),max(xR)),ylim=c(min(yL),max(yL)),
+                xlab = "x",ylab = "y")
> lines(xL[1:(nleft+1)],yL[1:(nleft+1)],lwd=2,col="blue")
> lines(xR[2:(nright+2)],yR[2:(nright+2)],lwd=2,col="red")
> mygrid()
```

shows an un-normalized wave function with two pieces with a large change in slope at the matching point, as indicated by the large value of **dy_diff()**.

The function **dy_diff()** used above is designed to return the difference of the approximate numerical first derivatives at the matching point **x2c** implied by **yL** and **yR**, divided by the value of **y(x2c)**. Here is our code for **dy_diff()**.

```
##    dy_diff() uses global yL, yR,nleft, h
##    computes numerical y'(x2c)  using
##    symmetric three point method for
##    both yL and yR, and returns the difference
##    divided by y(x2c)

dy_diff  =  function()   {
    ypL =   ( last(yL) - yL[nleft] ) / (2*h)
    ypR =   (yR[3] - yR[1]) / (2*h)
    (ypL - ypR)/ abs (yR[2])  }
```

The function **num_nodes()** has the definition:

```
##    num_nodes()
##    count the number of nodes in yL
##    ignore region where elements of yL are
##    tiny in magnitude.
##    takes advantage of the fact that xL elements steadily increase

num_nodes  =  function ()    {
    x11 = x_left + x1decay
    j0 = which(xL > x11) [1]   ## position in xL where x > x11
    n = 0
    for (j in  j0: (length(yL) - 3) )  { if (yL[ j ] * yL[ j + 1 ] < 0)   n = n + 1 }
    n  }
```

We search for energy eigenvalues by seeking energies **E** such that the value returned by **dy_diff()** is zero to within numerical errors. A function **F(E)** allows us to scan energy ranges for energy eigenvalues.

```
##    F(E)
##    energy eigenvalue if global function F(E) = 0 .
##    F(E) calls wf(E) then returns dy_diff(), but
##     returns false if E > 0.
```

```
F  =  function (E)    {
    if (E > 0)   {
        cat (" in F(E), E = ",E,"  should be negative   \n")
        return(FALSE)   }
    wf(E)
    dy_diff()  }
```

Here is an example of using **F(E)**.

```
> EL = seq(-0.91,-0.85,by = 0.01)
> FL = sapply(EL, F)
> head(FL)
[1]   7.19967   2.03578  -5.22871 -24.50145 -58.60732 232.57810
> head(EL)
[1] -0.91 -0.90 -0.89 -0.88 -0.87 -0.86
> F(-0.91)
[1] 7.19967
> F(-0.88)
[1] -24.5015
> e = uniroot(F, c(-0.91,-0.88),tol = 1e-16)$root; e
[1] -0.896404
```

A function **wf_plot(E)** generates a non-normalized numerical solution using **wf(E)**, makes a plot and prints out the energy and maximum y value, the number of nodes, and the value of **dy_diff()** corresponding to the chosen energy **E**.

```
> wf_plot(e)
 E =  -0.896404 ,      ymax =  27.4223
 number of nodes =  0 ,      dy_diff =  7.44138e-14
```

which produces the plot

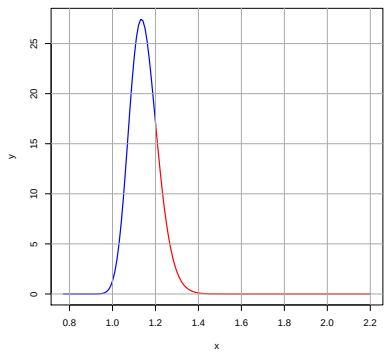

Figure 39: Zero Nodes Un-normalized Eigenfunction

The code for **wf_plot(E)** is

```
##    wf_plot (E) calls wf(E) and plot2d
##    creates ** un-normalized **  wave functions
##    stored in global xL, yL, xR, yR.
##    prints out number of nodes in yL
##    prints out dy_diff .

wf_plot  =  function(E)  {
    wf(E)
    cat ( " E = ", E, ",      ymax = ", max(yL), "\n" )
    cat (" number of nodes = ",num_nodes(), ",      dy_diff = ",dy_diff(), "\n" )
    plot(0,type="n",xlim=c(min(xL),max(xR)),ylim=c(min(yL),max(yL)),
                        xlab = "x",ylab = "y")
    lines(xL[1:(nleft+1)],yL[1:(nleft+1)],lwd=2,col="blue")
    lines(xR[2:(nright+2)],yR[2:(nright+2)],lwd=2,col="red")
    mygrid()        }
```

We then create (from **yL** and **yR**) a global normalized wave function vector **yn** corresponding to a global grid vector **xn** created from **xL** and **xR** using the function **normalize()**. This function also computes and prints the value of the quantum mechanical particle position uncertainty Δx implied by the wave function.

```
> normalize()
 AA =  85.0069
 x_mean =  1.14069
 delx =  0.0455039
> fll(xn)
  0.781455   2.20145   143
> fll(yn)
  -3.14183e-20   0   143
> max(yn)
[1] 2.97425
> max(yL)
[1] 27.4223
```

The function **normalize** uses our utility functions **simp** (Simpson's one third integration rule) and **merge**.

```
##    normalize() uses the current global xL,yL, xR, yR and
##    the  utility function simp (Simpson's 1/3 rule)  to define global
##    xn and yn, with the latter being normalized.

normalize  =  function()    {
    xn = c ( xL[1:(length(xL) - 1)], xR[3:length(xR)] )
    yn = c ( yL[1:(length(yL) - 1)], yR[3:length(yR)] )
    ##   we need xn to have odd number of elements to use simp
    if (is.even ( length (xn) ) )  {
        xn = xn[2 : length(xn)]
        yn = yn[2 : length(yn)]    }
    AA = simp(xn,yn^2)
    cat ( " AA = ",AA, "\n")
    yn = yn/sqrt(AA)
    x_mean = simp(xn, xn * yn^2)
    cat (" x_mean = ", x_mean, "\n")
    x2_mean = simp(xn, xn^2 * yn^2)
    delx2 = x2_mean - x_mean^2     ##   this should be positive!
    delx = sqrt(delx2)
    cat (" delx = ", delx, "\n")
    xn  <<-   xn
    yn  <<-   yn         }
```

Once **normalize()** has been used to create **xn** and **yn** from the current un-normalized wave function, we can use **yn_plot_current()** to see the current normalized wave function.

```
> yn_plot_current()
 ymax =  2.97425
```

which produces the plot

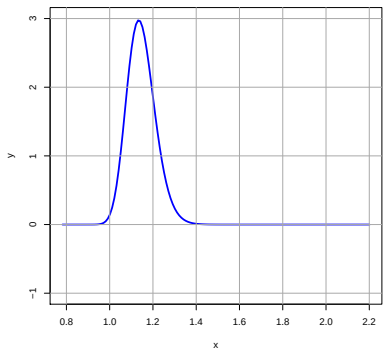

Figure 40: Zero Node Normalized Eigenfunction

A function **yn_plot(E, xmin, xmax)** goes from a given energy **E** to a call to **wf(E)** and **normalize()** and then makes a plot of the resulting normalized wave function in one step, with control over the region of the **x** axis for the plot. Thus

```
> yn_plot(e,0.8, 1.6)
 E =  -0.896404
 number of nodes =  0 ,       dy_diff =  7.44138e-14
 AA =  85.0069
 x_mean =  1.14069
 delx =  0.0455039
 normalized ymax =  2.97425
```

produces the plot

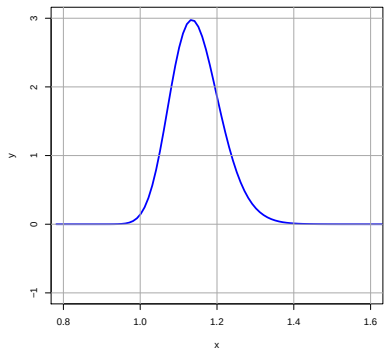

Figure 41: Using yn_plot(E,xmin,xmax) for $E = -0.896404$

Here is an example of using **yn_plot** for an energy which is not an energy eigenvalue. The value of **dy_diff()** reported is based on the non-normalized wave function produced by **wf(E)**.

```
> yn_plot(-0.95,0.8, 1.6)
 E =  -0.95
 number of nodes =  0 ,       dy_diff =  16.3981
 AA =  139.333
 x_mean =  1.15457
 delx =  0.0420213
 normalized ymax =  3.26734
```

which produces the plot

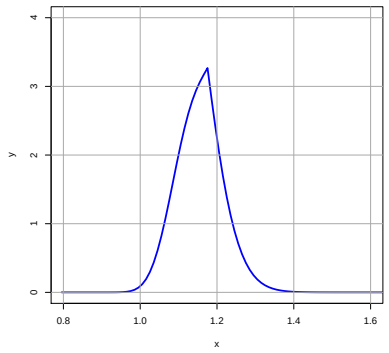

Figure 42: Using yn_plot(E,xmin,xmax) for $E = -0.95$

We see a large discontinuity in the slope of the normalized wave function, reflected also in the large value of **dy_diff()** reported in the non-normalized wave function.

Here is the code for **yn_plot**:

```
##    yn_plot(E,xmin,xmax) first calls wf(E) to create
##    un-normalized wave functions corresponding to the
##    given energy E. Then normalizes those wave functions
##    to produce the vectors xn and yn. Finally makes a plot
##    of yn over only the region (xmn, xmx)   */

yn_plot =  function (E,xmn,xmx)   {
    wf(E)
    cat (" E = ",E, "\n" )
    cat (" number of nodes = ",num_nodes(),",       dy_diff = ",dy_diff(), "\n" )
    normalize()
    cat (" normalized ymax = ", max(yn), "\n" )
    ymn = floor( min(yn) )
    ymx =  1 + floor( max (yn) )
    plot(xn,yn,type = "l",ylim = c(ymn,ymx), xlim = c(xmn, xmx),
                lwd=3,col="blue",xlab="x",ylab="y")
    mygrid() }
```

A plot of the values of **F(E)** over a wider energy range will show other candidate energies for excited states having energies greater than the ground state (zero node state with energy $E_0 = -0.896404$ found above). However, use of the function **bracket(Estart,dE,Eacc)** is an easier way to find candidate energy eigenvalues. **bracket** looks for the first sign change in **F(E)**, and (in a normal exit) returns a pair of energies for which **F(E)** has the opposite sign. Applying this approach to the ground state energy found above,

```
> out = bracket(F,-0.96,0.02,0.01)
> out
[1] -0.900 -0.895
> e = uniroot(F, out, tol = 1e-16)$root
> e
[1] -0.896404
> yn_plot(e,0.8,1.6)
 E =  -0.896404
 number of nodes =  0 ,       dy_diff =  -1.86035e-13
 AA =  85.0069
 x_mean =  1.14069
 delx =  0.0455039
 normalized ymax =  2.97425
```

which results in the same plot as we displayed above for the normalized ground state.

Here is our code for **bracket**:

```
##    bracket is a modified version of bracket_basic, designed to work with
##        the function F(E)  which can return FALSE.
##    bracket looks for a sign change in func,
##        starting with xx, and increasing xx by dxx each step.
##    If sign change is found, then we back up to the previous xx
##        and search with new dxx value one half of the previous value.
##    normally returns [ea,eb], but if can't find change in sign,
##        then returns [0,0], and if func returns FALSE, then
##        bracket returns FALSE.

bracket =  function (func,xx,dxx,xacc)  {
    x = xx
    dx = dxx
    it = 0
    itmax = 1000
    anerror = FALSE
    anerror2 = FALSE
```

```
    repeat  {
        it = it + 1
        if (it > itmax)  {
            cat (" can't find change in sign  \n")
            anerror = TRUE
            break}
        x1 = x
        x2 = x + dx
        f1 = func(x1)
        if ( f1 == FALSE)   {
            cat (" in bracket, f1 = FALSE , x1 = ",x1, "  dx = ", dx, " \n ")
            anerror2 = TRUE
            break }
        f2 = func(x2)
        if ( f2 == FALSE)   {
            cat (" in bracket, f2 = FALSE , x2 = ",x2, "  dx = ", dx, " \n ")
            anerror2 = TRUE
            break }
        if  ( f1 * f2  < 0 )    {
            if  ( abs(dx) < xacc )   break
            x = x - dx
            dx = dx/2 }  else x = x2 }
    if (anerror) c(0,0) else if (anerror2) FALSE  else c(x1,x2)          }
```

Let's use **`bracket`** to find a candidate energy for the first excited state, which should have one node and be a continuous function.

```
> out = bracket(F,e + 0.01,0.02,0.01)
> out
[1] -0.866404 -0.861404
> e = uniroot(F, out, tol = 1e-16)$root
> e
[1] -0.865689
> yn_plot(e,0.8,1.6)
 E =  -0.865689
 number of nodes =  0 ,       dy_diff =  -2.32077e+15
 AA =  2.17377e+29
 x_mean =  1.11923
 delx =  0.0350225
 normalized ymax =  3.28918
```

which produces the plot

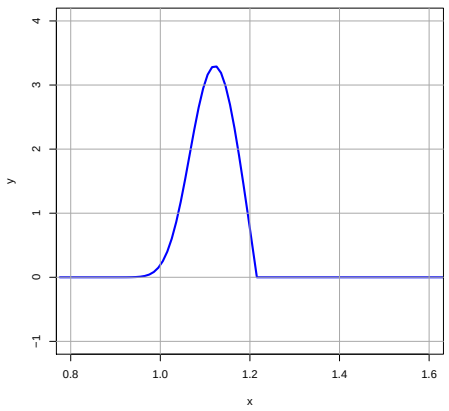

Figure 43: False Energy Eigenvalue for $E = -0.865689$

The plot shows a discontinuous zero node wave function and the reported value of **`dy_diff()`** is not a tiny number, as **`dy_diff()`** should be for a valid energy eigenvalue case. We also note that we had already found a valid zero node energy eigenvalue.

Continuing with a search for energy eigenvalues using **bracket**:

```
> out = bracket(F,e + 0.01,0.02,0.01)
> out
[1] -0.710689 -0.705689
> e = uniroot(F, out, tol = 1e-16)$root
> e
[1] -0.71066
> yn_plot(e,0.8,1.6)
 E =  -0.71066
 number of nodes =  1 ,       dy_diff =  -1.63498e-13
 AA =  0.0295798
 x_mean =  1.18063
 delx =  0.0807403
 normalized ymax =  2.60007
```

which produces a valid one node wave function

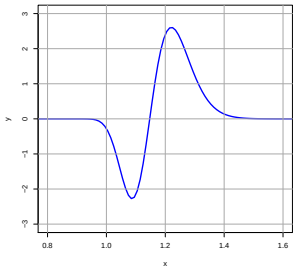

Figure 44: One Node Wave Function for $E = -0.71066$

This one node wave function is continuous and the value of **dy_diff()** is a tiny number.

We will again find a spurious one node solution at a slightly higher energy.

```
> out = bracket(F,e + 0.01,0.02,0.01); out
[1] -0.71566 -0.71066
> out = bracket(F,e + 0.02,0.02,0.01); out
[1] -0.68566 -0.68066
> e = uniroot(F, out, tol = 1e-16)$root; e
[1] -0.684654
> yn_plot(e,0.8,1.6)
 E =  -0.684654
 number of nodes =  1 ,       dy_diff =  1.353e+15
 AA =  3.70704e+25
 x_mean =  1.15311
 delx =  0.0706492
 normalized ymax =  2.5595
```

which shows a spurious one node wave function

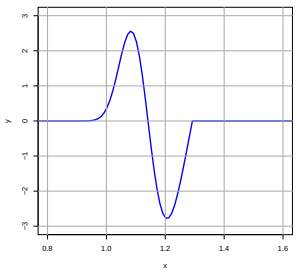

Figure 45: False Energy Eigenvalue for $E = -0.684654$

The plot shows again a discontinuous wave function, corresponding to the very large value of **dy_diff()**.

Now that we see the pattern of valid energy eigenvalues, we can write a function **levels(Emin,Emax,dE, Eacc)**, which uses **bracket** with the added filter that no sign change of **F(E)** is taken seriously unless the associated wave function has one more node than the last energy eigenvalue found.

```
##    levels(Emin,Emax,dE, Eacc ) returns a vector  of energy levels with
##         increasingly larger number of nodes in energy range (Emin, Emax).
##    uses F(E) to find  roots, and calls wf, num_nodes() and dy_diff() for each root found,
##    Uses bracket and uniroot.
##    The arguments (dE, Eacc) are used to call bracket, and do not describe
##        the accuracy of the energy levels found.
##    Once a good energy e.v. is found we look for the region of
##        energies with one more node and search there.
##    Code contains an interactive continue or stop decision.

levels  =  function (Emin,Emax,dE, Eacc )   {
    rmax = 20
    eL = rep(NA, rmax)         ##  vector  eL will hold energy eigenvalues found
    e = Emin
    nlast =  -1
    j = 1
    repeat {
        if  (e > Emax  |  j > rmax)  break    ## exit do loop
        cat ("---------------- levels ------------------\n")
        cat (" nlast = ", nlast,"\n")
        cat (" Estart = ", e," dE = ", dE, "\n" )
        out =  bracket(F,e, dE, Eacc)
        cat (" ea = ",out[1]," eb = ",out[2],"\n")
        if (out[1] == 0)    {
            cat (" can't find bracket interval \n")
            cat (" e = ",e, "\n")
            break  }
        eroot = uniroot(F, out, tol = 1e-16)$root
        cat (" eroot = ", eroot, "\n")
        wf(eroot)
        nn = num_nodes()
        cat (" number of nodes = ", nn, "\n")
        cat (" dy_diff at x = x2c is    ", dy_diff(), "\n" )
        eL[ j ]  = eroot
        nlast = nn
        j = j + 1
        r = readline (" input c or s \n ")
        if (r == "s")  break            ## exit do loop
        ##    search for an  e value greater than eb which produces
        ##      a wave function with nn + 1 nodes
        enext  =  out[2] + dE
        repeat  {
            wf(enext)
            if  (num_nodes() > nlast)   {
                e = enext
                break }  else enext =  enext + dE }  }   ## end of outer repeat loop
    ##   remove NA's at end of vector eL
    eL[!is.na(eL)]  }
```

Here is an example of use:

```
> EL = levels(-0.95, -0.6, 0.02,0.01)
---------------- levels ------------------
 nlast =  -1
 Estart =  -0.95  dE =  0.02
 ea =  -0.9  eb =  -0.895
 eroot =  -0.896404
 number of nodes =  0
 dy_diff at x = x2c is     -1.86035e-13
```

```
 input c or s
 c
---------------- levels ------------------
 nlast =  0
 Estart =  -0.835  dE =  0.02
 ea =  -0.715  eb =  -0.71
 eroot =  -0.71066
 number of nodes =  1
 dy_diff at x = x2c is     -1.63498e-13
 input c or s
 c
---------------- levels ------------------
 nlast =  1
 Estart =  -0.67  dE =  0.02
 ea =  -0.555  eb =  -0.55
 eroot =  -0.551436
 number of nodes =  2
 dy_diff at x = x2c is     1.72142e-13
 input c or s
 c
> EL
[1] -0.896404 -0.710660 -0.551436
```

We can then use **yn_plot(E,xmin,xmax)** to both construct the vectors **xn** and **yn** of the normalized wave function and make a plot. We save the normalized wave functions by assignment statements such as **xn0 = xn**, and **yn0 = yn**, before another call to **yn_plot** defines the wave functions corresponding to a different energy. In the following, we do not show the plots produced by the calls to **yn_plot**.

```
> E0 = EL[1]; E0
[1] -0.896404
> yn_plot(E0,0.8,1.6)
 E =  -0.896404
 number of nodes =  0 ,       dy_diff =  -1.86035e-13
 AA =  85.0069
 x_mean =  1.14069
 delx =  0.0455039
 normalized ymax =  2.97425
> xn0 = xn; fll(xn0)
  0.781455   2.20145   143
> yn0 = yn; fll(yn0)
  -3.14183e-20   0   143
```

We continue in this manner with the first excited state and the second excited state.

```
> E1 = EL[2]; E1
[1] -0.71066
> yn_plot(E1,0.8,1.6)
 E =  -0.71066
 number of nodes =  1 ,       dy_diff =  -1.63498e-13
 AA =  0.0295798
 x_mean =  1.18063
 delx =  0.0807403
 normalized ymax =  2.60007
> xn1 = xn; fll(xn1)
  0.754761   2.27476   153
> yn1 = yn; fll(yn1)
  2.83787e-25   0   153
> E2 = EL[3]; E2
[1] -0.551436
> yn_plot(E2,0.8,1.6)
 E =  -0.551436
 number of nodes =  2 ,       dy_diff =  1.72142e-13
 AA =  1.24981e-05
 x_mean =  1.2266
 delx =  0.108671
 normalized ymax =  2.44523
> xn2 = xn; fll(xn2)
  0.730536   2.35054   163
> yn2 = yn; fll(yn2)
  0   0   163
```

We can then combine the plots for the wave functions of these three lowest lying states.

```
> plot(0,type = "n",xlim = c(0.8,1.6),ylim = c(-3,3),xlab="x",ylab="y")
> lines(xn0,yn0,lwd=2,col="blue")
> lines(xn1,yn1,lwd=2,col="red")
> lines(xn2,yn2,lwd=2,col="green")
> mygrid()
> legend("bottomright",col = c("blue","red","green"),
+          legend = c("E0", "E1", "E2"), lwd=2,cex=1.5)
```

which produces a plot of the numerical normalized wave functions describing the lowest three energy levels.

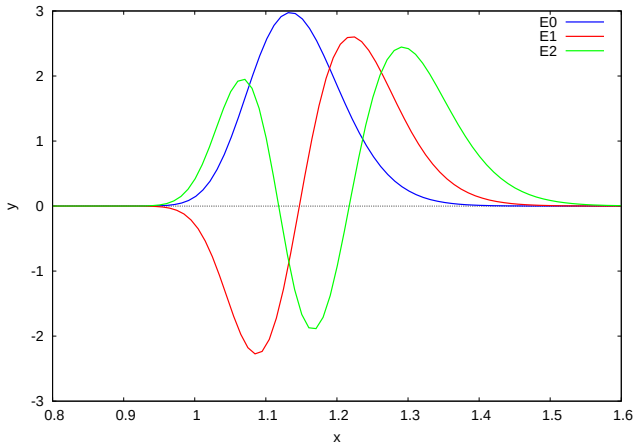

Figure 46: Lennard-Jones: Wave Functions for Lowest Three Energy Levels

Just for practice, we can write these three wave functions to a file in the local folder, and then restart **R**, read in the file contents, and make a plot based on the file contents.

The simplest approach is to use **`save(filePath,a1,a2,a3,...)`**, where objects **`a1`**, **`a2`**, etc are object names bound to quantities known to R. The names and the objects bound to the names are stored in a binary file format in the file requested (which is created if it does not yet exist, and overwritten if it already exists).

One can use **`load`** to load in that file into a new session, and the names and objects will then be available for use in your new Maxima session. In the following, we first save the wave function files to **`xy.rda`**. We then use **`rm`** to remove knowledge of those objects from the current session. We then use **`load`** to recover knowledge of those objects, which can then be used as before, for example to make plots and make calculations.

```
> save(xn0,yn0,xn1,yn1,xn2,yn2, file = "xy.rda")
> rm(xn0,yn0,xn1,yn1,xn2,yn2)
> fll(xn0)
Error in fll(xn0) : object 'xn0' not found
> load("xy.rda")
> fll(xn0)
  0.781455   2.20145   143
```

For example, we can remake the plot of all three wave functions

```
> plot(0,type = "n",xlim = c(0.8,1.6),ylim = c(-3,3),xlab="x",ylab="y")
> lines(xn0,yn0,lwd=2,col="blue")
> lines(xn1,yn1,lwd=2,col="red")
> lines(xn2,yn2,lwd=2,col="green")
> mygrid()
> legend("bottomright",col = c("blue","red","green"),
+         legend = c("E0", "E1", "E2"), lwd=2,cex=1.5)
```

and we get the same plot as above.

A proper exploration of the likely accuracy of the energy levels found in this approach would involve experimenting with the values of **`x1decay`**, **`x2decay`**, and the grid size **`h`**. One can also modify the code so that the use of a five point symmetric formula for the first derivative is used, instead of the present three point symmetric formula.