# Computational Examples for Physics Classes

## SC 12 Salt Lake City, November 12-16

Richard Gass
Department of Physics, University of Cincinnati
Cincinnati, OH 45221-0011
gass@physics.uc.edu

## Introduction to *Mathematica*

### A Very Brief Introduction

All built-in *Mathematica* commands and functions start with a capital letter. Function arguments are enclosed in square brackets. Parentheses control the order of operations and curly brackets are used for lists. *Mathematica* commands can be nest to arbitrary depth. Use return (RET) for a new line and enter( ENTER) or shift return to run code. Multiplication is denoted by using a space or a *. Use ^ to raise something to a power. Use ESC p ESC to get $\pi$.

### Examples

```
3 * 4 ^ (1 / 2)
```

6

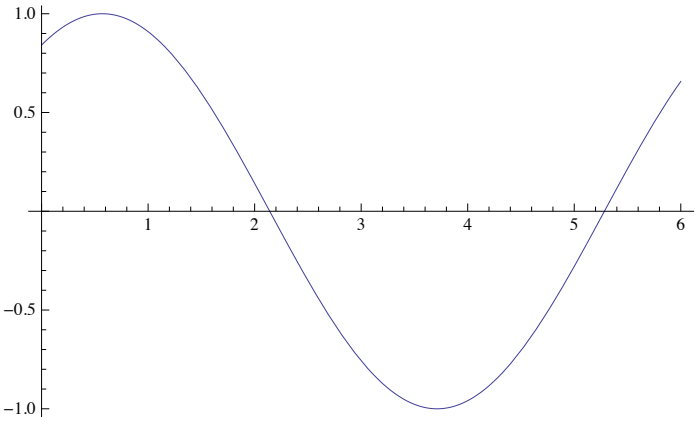
```
π
```

$\pi$

```
N[π]
```

3.14159

```
Plot[Sin[x + 1], {x, 0, 6}]
```




```
Table[RandomReal[], {i, 1, 10}]
```

{0.435173, 0.822377, 0.391235, 0.240684, 0.625791, 0.299427, 0.728945, 0.937551, 0.994226, 0.822635}

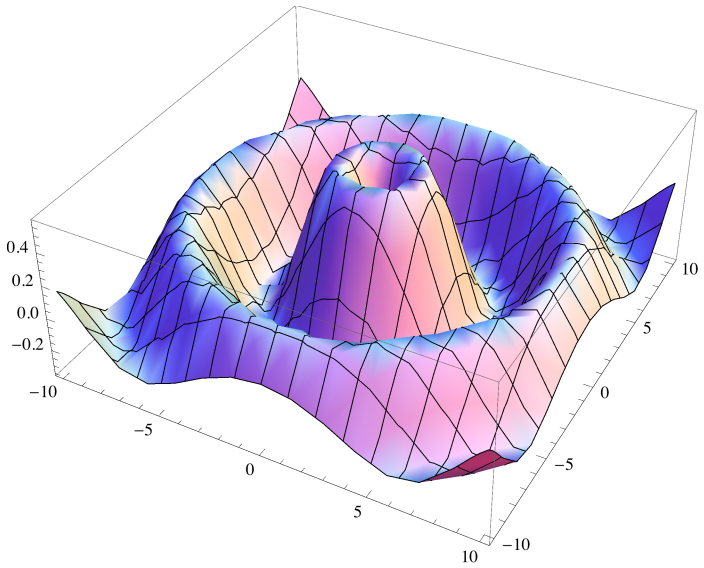
```
Integrate[Sin[x^2], x]
```

$$\sqrt{\frac{\pi}{2}}\, S\left(\sqrt{\frac{2}{\pi}}\, x\right)$$

```
Integrate[Sin[x^2], {x, 0, 1}]
```

$$\sqrt{\frac{\pi}{2}}\, S\left(\sqrt{\frac{2}{\pi}}\right)$$

```
Plot3D[BesselJ[1, Sqrt[x^2 + y^2]], {x, -10, 10}, {y, -10, 10}]
```




```
NIntegrate[BesselY[3 / 7, x^2], {x, 0, 5}]
```

−5.856

You may also want to look at the Wolfram Research training videos on getting stated with Mathematica.

### ■ A Brief Introduction

A more comprehensive introduction is given in the notebook Workshop.nb. The same material is also available as a cdf (computable document format) or a pdf.

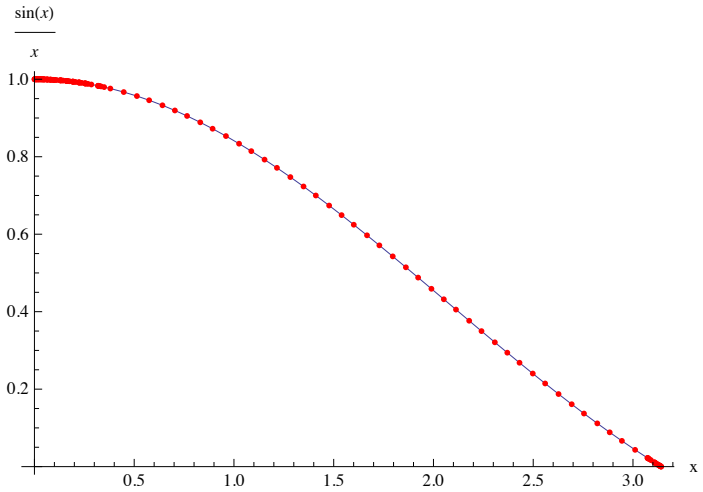
## Looking under the Hood

*Mathematica* has a number of high level numerical functions. The most useful of these may be Plot, Plot3D, NIntegrate, NDSolve and FindRoot. These functions all use sophisticated algorithms but hide the details from the user. There are times however when you need to see the details. The documentation will usually tell you what algorithms the function uses by default and how to change the method if you need to do but it often useful to be able to look at the performance of the algorithm visually. The functions defined below allow you to do this. These are not built-in functions so you must run the code defining them before they will work.

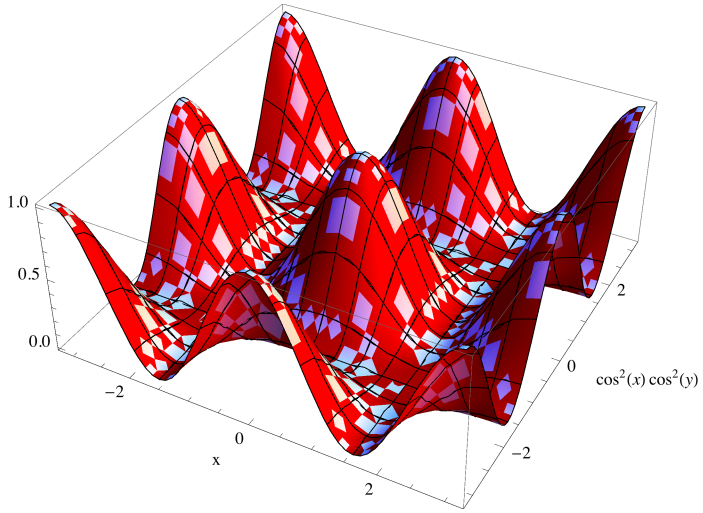
The function **ShowPlotPoints** will show you where the plotting routines sampled the function for both two and three dimensional plots.

```
ShowPlotPoints[f_, {x_, xmin_, xmax_}, options___] :=
 Module[{result, points}, {result, points} =
   Reap[Plot[f, {x, xmin, xmax}, EvaluationMonitor :→ Sow[{x, f}], options]];
  Show[Plot[f, {x, xmin, xmax}, AxesLabel → {ToString[x], ToString[TraditionalForm[f]]}],
   ListPlot[points, PlotStyle → Red]]]
ShowPlotPoints[f_, {x_, xmin_, xmax_}, {y_, ymin_, ymax_}, options___] :=
 Module[{result, points}, {result, points} = Reap[Plot3D[f, {x, xmin, xmax},
     {y, ymin, ymax}, EvaluationMonitor :→ Sow[{x, y, f}], options]];
  Show[Plot3D[f, {x, xmin, xmax}, {y, ymin, ymax},
    AxesLabel → {ToString[x], ToString[TraditionalForm[f]]}, options],
   ListPlot3D[points, PlotStyle → Red]]]
```

```
ShowPlotPoints[Sin[x] / x, {x, 0, π}]
```




```
ShowPlotPoints[Cos[y]^2 Cos[x]^2 , {x, -π, π}, {y, -π, π}, PlotRange → All]
```

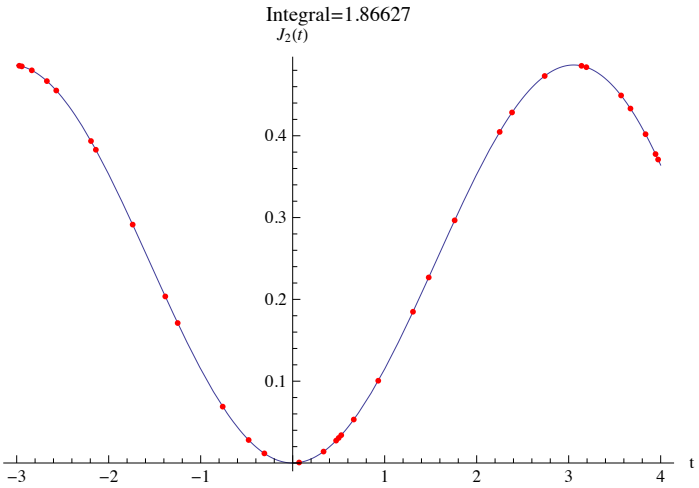



The function **ShowIntegrationPoints** will show you the points at which the numerical integration routines sampled the function. **ShowIntegrationPoints** works for one and two dimensional numerical integrals.

```
ShowIntegrationPoints[f_, {x_, xmin_, xmax_}, options___] :=
 Module[{result, points}, {result, points} =
   Reap[NIntegrate[f, {x, xmin, xmax}, EvaluationMonitor :> Sow[{x, f}], options]];
  Show[Plot[f, {x, xmin, xmax}, AxesLabel → {ToString[x], ToString[TraditionalForm[f]]},
    PlotLabel → StringJoin["Integral=", ToString[result]]],
   ListPlot[points, PlotStyle → Red]]]
ShowIntegrationPoints[f_, {x_, xmin_, xmax_}, {y_, ymin_, ymax_}, options___] :=
 Module[{result, points}, {result, points} = Reap[NIntegrate[f, {x, xmin, xmax},
     {y, ymin, ymax}, EvaluationMonitor :> Sow[{x, y, f}], options]];
  Show[Plot3D[f, {x, xmin, xmax}, {y, ymin, ymax}, PlotStyle → Opacity[.25],
    AxesLabel → {ToString[x], ToString[y], ToString[TraditionalForm[f]]}, PlotLabel →
     StringJoin["Integral=", ToString[result]]], ListPlot3D[points, PlotStyle → Red]]]
```

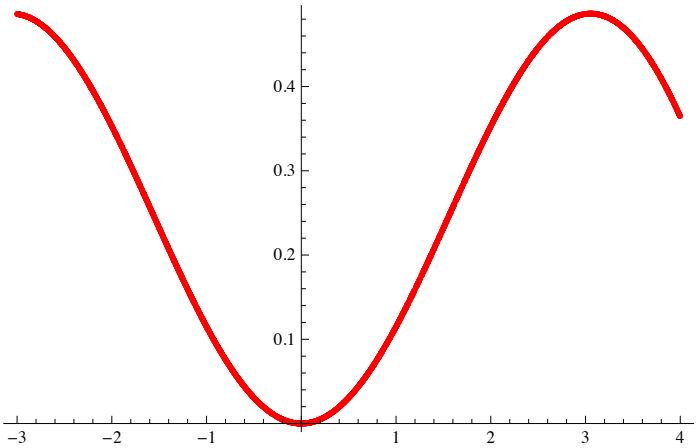
As an example we integrate $J_2(t)$ from $t = 2$ to $t = 3$. The points where the Bessel function was sampled are shown as red dots.

```
ShowIntegrationPoints[BesselJ[2, t], {t, -3, 4}]
```



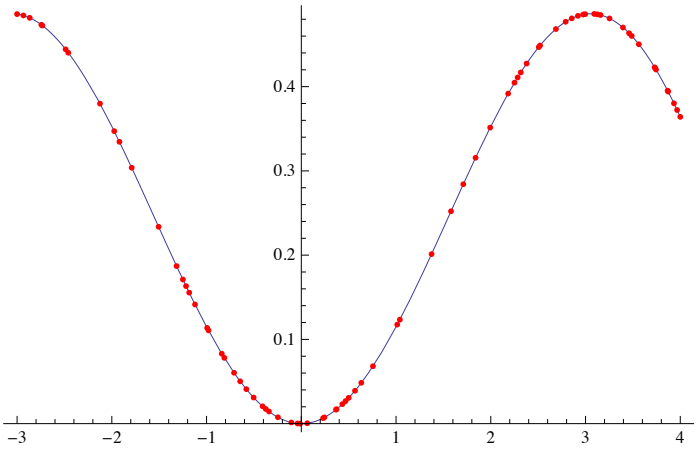

One can change the integration method using the method option. Using a Monte Carlo forces many more points to be sampled showing why this is a terrible method for this integral.

```
ShowIntegrationPoints[BesselJ[2, t], {t, -3, 4}, Method -> "MonteCarlo"]
```




Another example of changing the method.

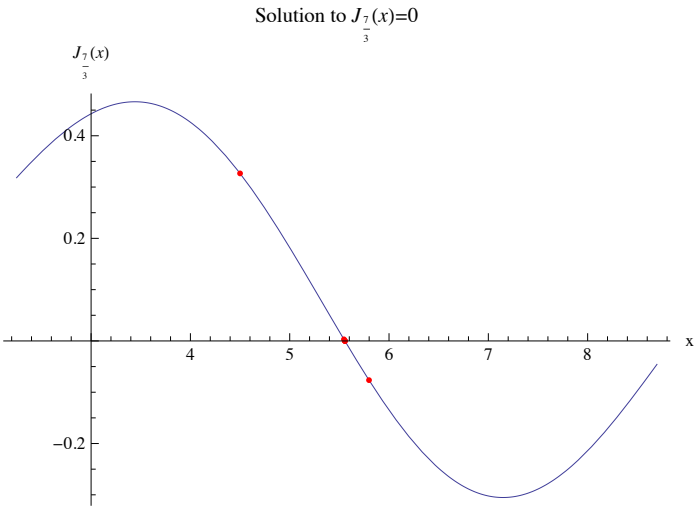
```
ShowIntegrationPoints[BesselJ[2, t], {t, -3, 4}, Method -> "ClenshawCurtisRule"]
```




The function **ShowFindRootPoints** shows how the root finder **FindRoot** converges or fails to converge.

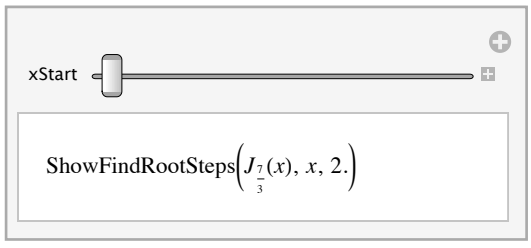
```
ShowFindRootSteps[f_, x_, xStart_] := Module[{},
  points = Reap[FindRoot[f == 0, {x, xStart}, EvaluationMonitor :> Sow[{x, f}]]][[2]];
  xValues = Chop[points /. {z_, y_} → z];
  Show[Plot[f,
    {x, Min[1.5 Min[xValues], .5 Min[xValues]], Max[1.5 Max[xValues], .5 Max[xValues]]},
    AxesLabel → {ToString[x], ToString[TraditionalForm[f]]},
    PlotLabel → StringJoin["Solution to ", ToString[TraditionalForm[f]], "=0"] ],
   ListPlot[Chop[points], PlotStyle → Red]]]
```

```
ShowFindRootSteps[BesselJ[7 / 3, x], x, 4.5]
```




This can be combined with Manipulate to examine the dependence on the initial starting point.

```
Manipulate[ShowFindRootSteps[BesselJ[7 / 3, x], x, xStart],
 {xStart, 2, 7}, SaveDefinitions → True]
```



The same methods allow one to monitor the solution for a differential equation using NDSolve. You can look not only at where the where the function is sampled but also for monitor the energy of the system. Details are given in the notebook EvaluationsMonitors. CDF and PDF versions are here(CDF) and here(PDF).

### Quantum Mechanics

## Bound States

One way to find bound states is to use an eigenvalue expansion. Although this methods scales badly with dimension and particle number it has the advantage of being conceptually simple and relatively straightforward to program.

One method of solving eigenvalue problems is by expansion in normal modes or eigenfunctions. Suppose we want to solve

$$A\,\psi = \lambda\,\psi \tag{1}$$

where $A$ is an operator such as $\frac{-\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)$and the $\lambda$'s are the eigenvalues of $A$. In our case the $\lambda$'s will be the energies of the bound states of a quantum well. In general we will not be able to solve equation exactly so we need to find an approximate solution.

The strategy is to expand $\psi$ as

$$\psi = \sum_n \mathrm{a_n}\,\phi_n \tag{2}$$

where the $\phi_n$ are known functions and form a complete orthonormal basis This means that

$\int \phi^*{}_m\,\phi_n\,dV = \delta_{n,m} = \begin{cases} 1 \text{ if } n = m \\ 0 \text{ if } n \neq m \end{cases}$ where the $dV$ tells us to integrate over the whole space.

From equation (1) we have upon multiplying by $\psi_n{}^*$ and integrating

$$\int \psi_m{}^*\,A\,\psi_n\,dV \;=\; \lambda \int \psi_m{}^*\,\psi_n\,dV \tag{3}$$

Using the expansion given by equation (2) in equation (3) gives

$\sum_{m,n} a_m{}^*\,a_n \int \phi_m{}^*\,A\,\phi_n\,dV = \lambda \sum_{m,n} a_m{}^*\,a_n\left(\int \phi_m{}^*\,\phi_n\,dV\right) \quad \Rightarrow \sum_{m,n} a_m{}^*\,a_n \int \phi_m{}^*\,A\,\phi_n\,dV = \lambda \sum_{m,n} a_m{}^*\,\delta_{m,n}$ or $\sum_{m,n} a_m{}^*\,a_n\left(\int \phi_m{}^*\,A\,\phi_n\,dV - \lambda\,\delta_{n,m}\right) = 0.$

Which shows that the eigenvalues $\lambda_n$ are given by the eigenvalues of the matrix $\boldsymbol{M}$ defined by $M_{m,n} = \int \phi_m{}^*\,A\,\phi_n\,dV$. Note that $\boldsymbol{M}$ is infinite dimensional. The functions $\psi_n$ are given by the inner (or dot) product of the eigenfunctions of $\boldsymbol{M}$ with the basis functions $\phi$. At this point the solution is exact but the price is an infinite dimensional matrix $\boldsymbol{M}$. The approximation we will make is to truncate $\boldsymbol{M}$ so that it is a finite dimensional matrix.

## The code

The algorithm is implemented by the code below. The wave functions are returned in the list called functions and the energy levels in the list called states.

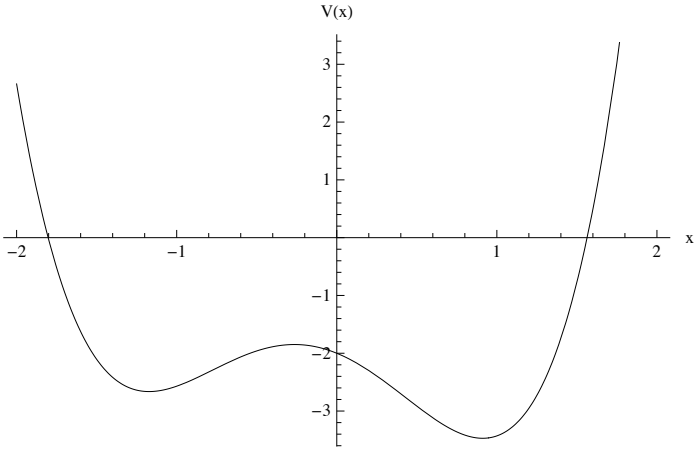
```
waveFunctionSolver[V_, x_, edge_, n_] := Module[{ϕ, L, Hψ, a, M, vector, levels},
  Clear[ϕ, Hψ, a, L, M, states, vectors, levels, functions];
  L = edge;
  ϕ[i_, x] = (√(2 / L) Sin[i π (x + L / 2) / L]);
  Hψ[i_, x] = -ħ^2 / (2 m) D[ϕ[i, x], {x, 2}] + V ϕ[i, x];
  a[j_, i_] := NIntegrate[ϕ[j, x] Hψ[i, x], {x, -L / 2, L / 2}, AccuracyGoal → 3];
numberOfBasisFunctions = n;
  ħ = 1;
  m = 1;

  M = Table[a[j, k], {j, 1, numberOfBasisFunctions}, {k, 1, numberOfBasisFunctions}];
  states = Eigenvalues[Chop[N[M]]];
  basisFunctions = Table[ϕ[k, x], {k, 1, numberOfBasisFunctions}];
vectors = Eigenvectors[Chop[N[M]]];
wavefunctions = vectors.basisFunctions;
levels = Flatten[Map[Position[states, #] &, Sort[states]]];
functions = Map[wavefunctions[[#]] &, levels];]
```

## Example

Suppose we have the following asymmetric potential

```
Vpot[x_] = x^4 + .7 x^3 - 2 x^2 - 1.13 x - 2;
Vplot = Plot[Vpot[x], {x, -2, 2}, AxesLabel → {"x", "V(x)"}, PlotStyle → Black]
```

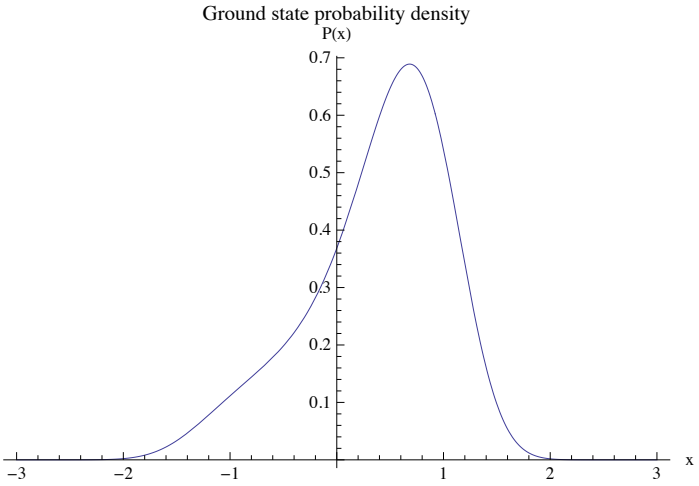



We now solve for the eigenvalues and eigenvectors with $L = 30$ and 30 basis functions.

```
waveFunctionSolver[x^4 + .7 x^3 - 2 x^2 - 1.13 x - 2, x, 6, 30]
```

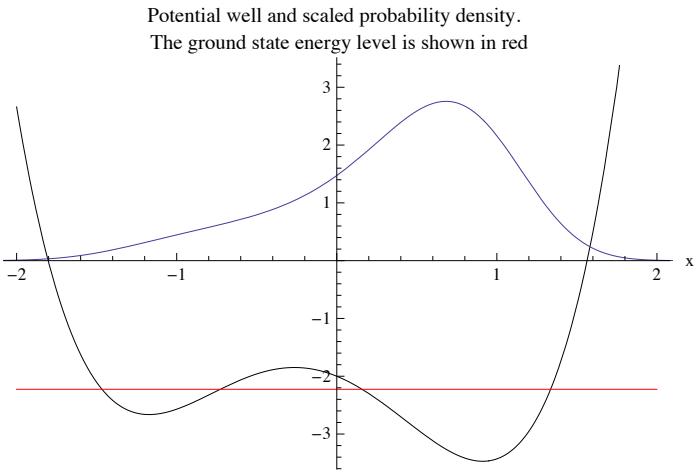
The wave functions and states are returned in order so the ground state probability density is

```
plot1 = Plot[functions[[1]]^2, {x, -3, 3},
  AxesLabel → {"x", "P(x)"}, PlotLabel → "Ground state probability density"]
```




Notice that the probability density is very asymmetric. We can see why if we plot the probability density, energy of the ground state and the potential together. Of course the vertical axis has units of energy for the well and the ground state level and units of a probability density for the square of the wave function. I have scaled the probability density by a factor of four for visual clarity.

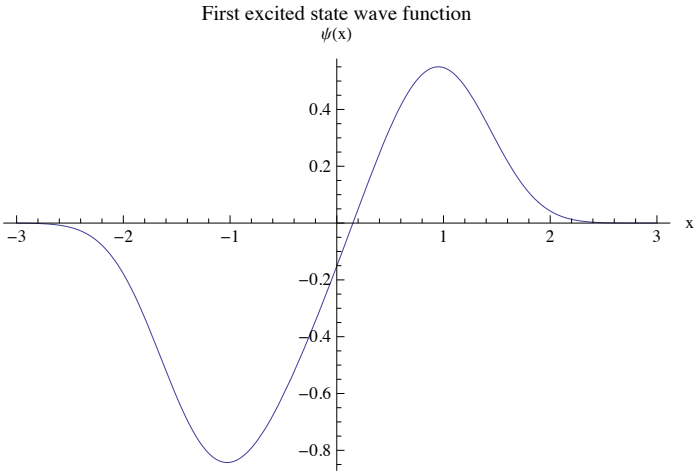
```
Show[Plot[Tooltip[Vpot[x]], {x, -2, 2}, AxesLabel → {"x", None}, PlotStyle → Black],
 Graphics[{Red, Line[{{-2, Sort[states][[1]]}, {2, Sort[states][[1]]}}]}],
 Plot[4 functions[[1]]^2, {x, -3, 3}],
 PlotLabel → "Potential well and scaled probability density.
    \n The ground state energy level is shown in red"]
```



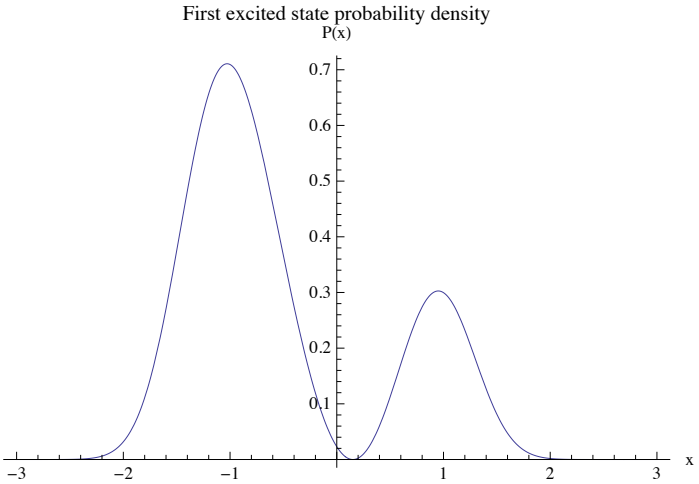

Looking at this we can understand why the wave function is so asymmetric.The ground state energy lies below "hump" in the well so to get from the right hand side of the well to the left hand side the particle must tunnel into the classically forbidden region. Notice that the probability density decays exponentially at the classical turning points which occur at the points were the red line intersects the potential.

We now turn our attention to the first exited state, which is the second element in the list "functions". Both the wave function and the probability density are plotted below. Once again the probability density is highly asymmetric but for a much different reason.

```
Plot[functions[[2]], {x, -3, 3}, AxesLabel → {"x", "ψ(x)"},
 PlotLabel → "First excited state wave function"]
```

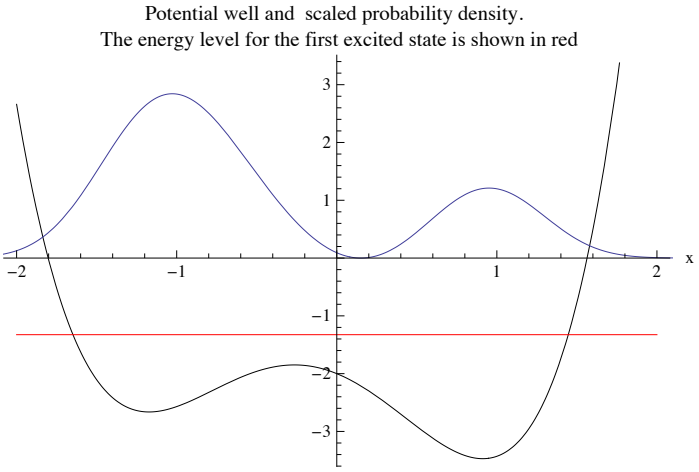



```
plot2 = Plot[functions[[2]]^2, {x, -3, 3}, AxesLabel → {"x", "P(x)"},
  PlotLabel → "First excited state probability density"]
```




Plotting the probability density and well as before (with the same scaling of the probability density) we see that the particle is more likely to found in the shallower well. This may seem counter intuitive but if we think about the particle's kinetic energy we can see why. The particle has less kinetic energy in the left well than it does in the right well. Since the widths of the wells are about the same the particle will spend more of it's time in the left hand side and is thus more likely to be found there when we look for it and thus localize the wave function.

```
Show[Plot[Tooltip[Vpot[x]], {x, -2, 2}, AxesLabel → {"x", None}, PlotStyle → Black],
 Graphics[{Red, Line[{{-2, Sort[states][[2]]}, {2, Sort[states][[2]]}}]}],
 Plot[4 functions[[2]]^2, {x, -3, 3}],
 PlotLabel → "Potential well and  scaled probability density.
    \n The energy level for the first excited state is shown in red"]
```




It's easy to compute the particle's average position. We just compute $< x >= \int_{-L/2}^{L/2} \psi^* x \psi \, dx$. Thus for the ground state we have

```
NIntegrate[x functions[[1]]^2, {x, -3, 3}]
```

0.374176

and for the first excited state we get

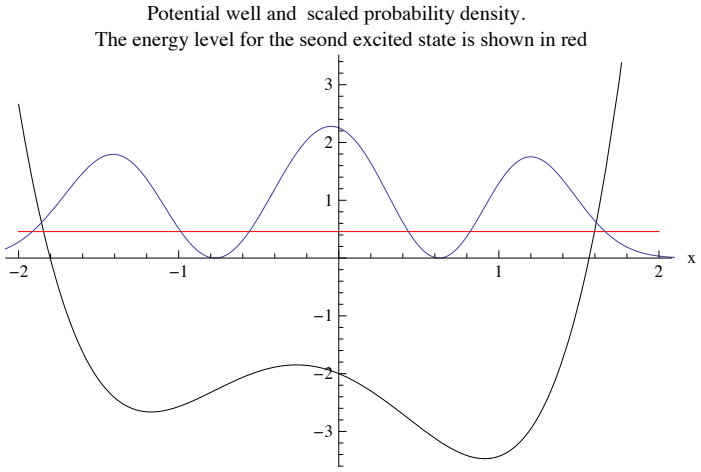
```
xAverage = NIntegrate[x functions[[2]]^2, {x, -3, 3}]
```

−0.518599

notice that I have done the integral numerically. I could have done the integral analytically but since we used 30 basis functions our integrand has $30^2 = 900$ terms so doing the integral analytically take a while but the numerical integration is fast.

Before leaving this example let's look at the second excited state. Notice that the probability density is only slightly asymmetric. As we move to high states and thus move up the energy landscape the particles position is no longer sensitive to the low lying features of the well and the wave function is most determined by the steep sides of the well.

```
Show[Plot[Tooltip[Vpot[x]], {x, -2, 2}, AxesLabel → {"x", None}, PlotStyle → Black],
 Graphics[{Red, Line[{{-2, Sort[states][[3]]}, {2, Sort[states][[3]]}}]}],
 Plot[4 functions[[3]]^2, {x, -3, 3}],
 PlotLabel → "Potential well and  scaled probability density.
    \n The energy level for the seond excited state is shown in red"]
```




# Wave Packet Dynamics

- **Wave packet scattering off barriers**

The standard textbook treatment of barriers considers a single plane wave incident on a barrier and computes the probability of reflection or transmission. This is time-independent. Here I want to consider the more complicated problem of a wave packet scattering off a barrier or barriers and calculate the time evolution of the packet. The first computer generated movies of wave packet scattering were done by Goldberg, Schey and Schwartz [1]. The example I want to consider is the scattering of a Gaussian wave packet of a set of two barriers. The time-dependent Schrödinger equation is coded below.

```
Clear[V, ħ, m, ψ]
eqn = i ħ D[ψ[x, t], t] == -(ħ^2 / 2 m) D[ψ[x, t], {x, 2}] + V[x] ψ[x, t]
```

$$i\,\hbar\,\psi^{(0,1)}[x,t] == V[x]\,\psi[x,t] - \frac{1}{2}\,m\,\hbar^2\,\psi^{(2,0)}[x,t]$$

The initial wave shape is taken to be a Gaussian wave packet centered at $x_0$with average momentum of $\hbar k_0$ and a width determined by $\sigma_0$. Note that the factor of $e^{i k_0 x}$ makes this a right-moving wave packet. For simplicity we will work with a unit mass particle and in "natural units" where $\hbar = 1$. The initial wave shape provides the initial condition that we need to solve the Schrödinger equation but we also need two boundary conditions. Physically we want the wave function to vanish at $\pm\infty$. To implement this condition numerically we will put our system in a box of length $L$ and require that the wave function vanish at the boundaries of the box.

```
InitialWaveFunction = Exp[i k0 x] Exp[-(x - x0)^2 / (2 σ0^2)]
```

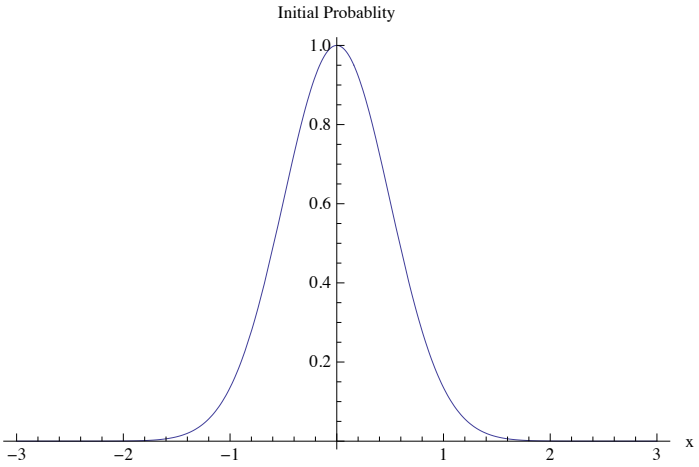
$$e^{i\,k_0\,x - \frac{(x-x_0)^2}{2\,\sigma_0^2}}$$

```
Clear[ψ, k0]
m = 1;
ħ = 1;
k0 = 1;
x0 = 0;
σ0 = 1 / √2 ;
L = 30;
```

It is important to realize that what we start with is a wave function. To get the initial probability distribution we have to multiply the wave function by its complex conjugate. The initial probability distribution is plotted below.

```
InitialWaveShape = InitialWaveFunction (InitialWaveFunction /. i → -i);

Plot[InitialWaveShape, {x, -3, 3}, AxesLabel → {"x", "Initial Probablity"}]
```

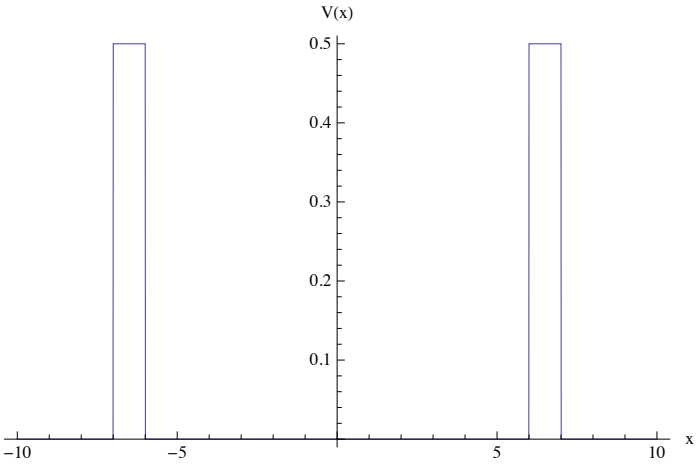



The next step is to define the barriers. I have chosen a barrier height of 1/2. To understand what this means physically, recall that the average momentum of the initial wave packet is $p = \hbar k_0$ so the average energy is $E = p^2/2m = 1/2$. The barriers are plotted below.

```
V[x_] =
 Piecewise[{{0, x < -7}, {1 / 2, -7 ≤ x ≤ -6}, {0, -6 ≤ x ≤ 6}, {1 / 2, 6 ≤ x ≤ 7}, {0, x > 7}}]
```

$$\begin{cases} 0 & x < -7 \\ \frac{1}{2} & -7 \le x \le -6 \\ 0 & -6 \le x \le 6 \\ \frac{1}{2} & 6 \le x \le 7 \end{cases}$$

```
well = Plot[V[x], {x, -10, 10}, Exclusions → None, AxesLabel → {"x", "V(x)"}]
```




We are now ready to solve the time-dependent Schrödinger equation. We can just call NDSolve with the appropriate initial condition and boundary conditions. Remember that the double bracket notation is used to ask for a part of an expression, so [[1,1,2]] gives the second sub-part of the first sub-part of the first part. I have also used the function TimeUsed to see how long this computation took. TimeUsed[] gives the amount of CPU time used in the current kernel session. On my machine the calculation takes 24 seconds. NDSolve uses the method of lines for the solution.

```
Clear[result]
a = TimeUsed[];
result[x_, t_] = NDSolve[{eqn, ψ[x, 0] == InitialWaveFunction, ψ[-L, t] == 0, ψ[L, t] == 0},
    ψ[x, t], {x, -L, L}, {t, 0, 15}][[1, 1, 2]]
TimeUsed[] -
 a
```

```
InterpolatingFunction[{{-30., 30.}, {0., 15.}}, <>][x, t]
```

```
24.1734
```

We can now multiply the result with its complex conjugate to get the probability density as a function of both space and time. I have used **Chop** to get rid of any small imaginary terms that might arise due to numerical noise.

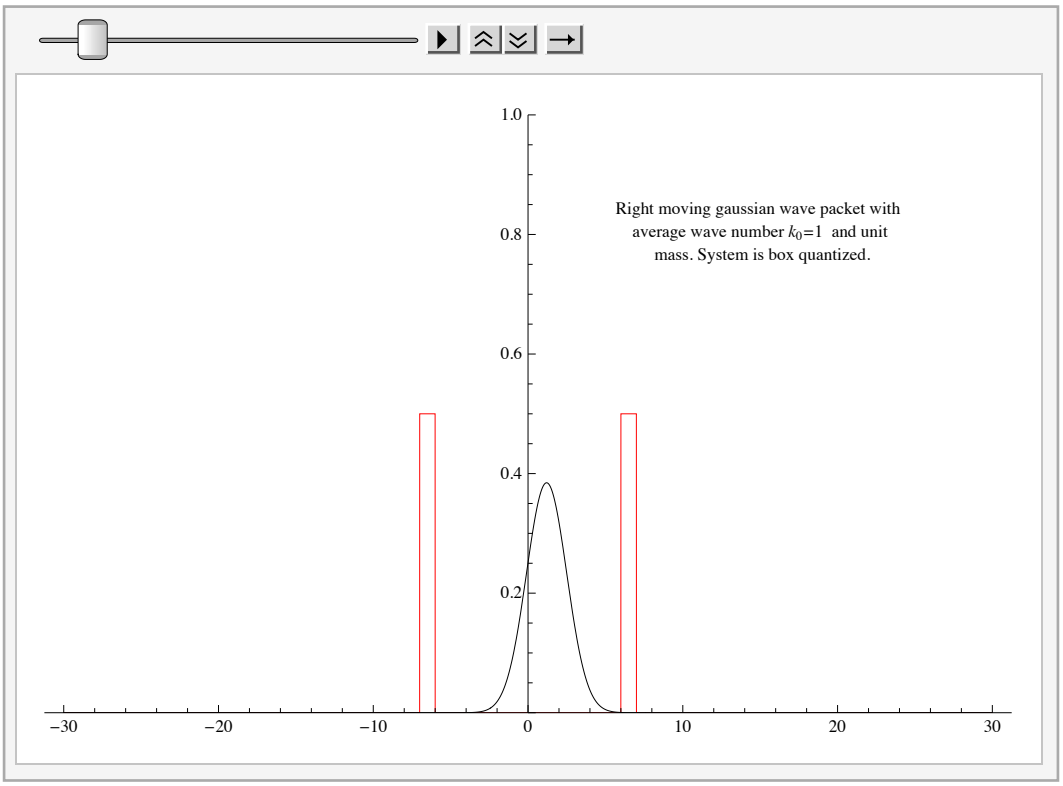
```
Clear[a, b]
wave[x_, t_] :=
 Chop[Times[result[x, t], (result[x, t] /. Complex[a_, b_] → Complex[a, -b])]]
```

The best way to understand the physics of this result is to make an animation of the wave packet. I first generate a plot caption

```
caption = Graphics[Text[
     StyleForm["Right moving gaussian wave packet with \n average wave number k0=1  and
        unit \n mass. System is box quantized.", FontFamily -> "Times"], {15, .8}]];
```

and then animate the motion, showing both the wave packet and the barriers on the same plot.

```
ListAnimate[Table[Show[Plot[{V[x], wave[x, t]}, {x, -30, 30}, Exclusions → None,
    PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 0, 0], RGBColor[0, 0, 0]},
    PlotRange → {0, 1}, PlotPoints → 200], caption, ImageSize → 7 × 72], {t, 0, 15, .15}]]
```




As you watch the animation you will see the wave packet move to the right (it's a right moving wave after all) and spread out. As it strikes the right barrier we see that there is some probability of reflection and some probability of transmission. When the packet reaches the right wall of the box the wave function is forced vanish by the boundary conditions. This is what causes the sinusoidal ripples!

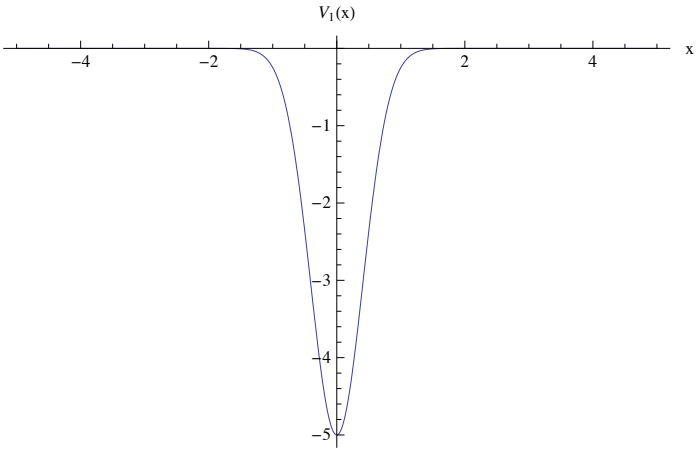
## ■ Wave packet splitting

An interesting wave packet phenomena of current interest is wave packet splitting. A simple method for splitting a wave packet was recently proposed by Zhang *et. al.* [2]. Wave packet splitting using a double well has recently been experimentally realized by Hall *et. al.* [3] . We can fairly easily numerically simulate wave packet splitting.

We first define a static potential.

```
V1[x_] := V0 Exp[-3 x^2]
V0 = -5;
```

```
Plot[V1[x], {x, -5, 5}, PlotRange → All, AxesLabel → {"x", "V1(x)"}]
```

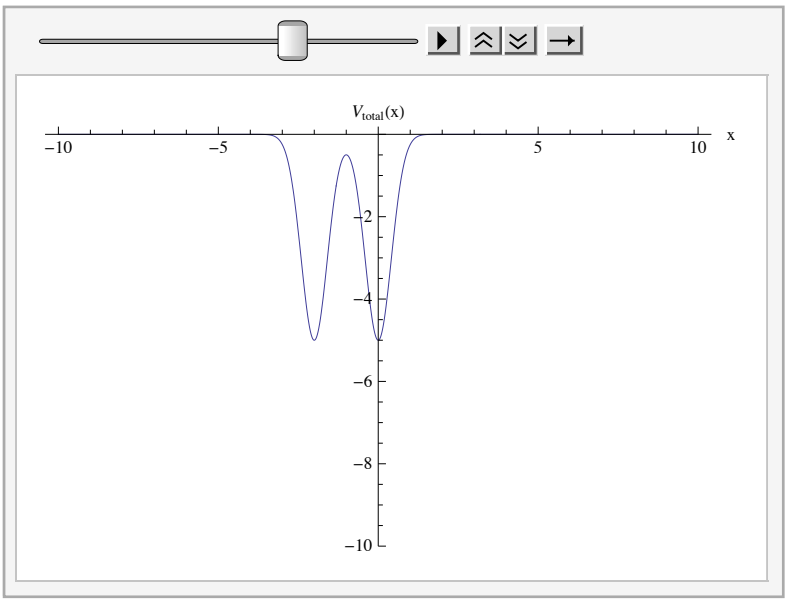



We now define a second potential which is time-dependent. This potential is left moving and its speed is controlled by $\omega$. At $t = 0$ the potential is centered around $x = 5$ so that the two potentials are distinct at $t = 0$.

```
V2[x_, t_] := V0 Exp[-3 (x - 5 + ω t)^2]
```

The full potential is $V_1 + V_2$ which I have plotted below for $\omega = 1$.

```
ω = 1;
ListAnimate[Table[Plot[V1[x] + V2[x, t], {x, -10, 10},
    PlotRange → {0, -10}, AxesLabel → {"x", "Vtotal(x)"}], {t, 0, 10, 0.5}]]
```




We start our wave packet in an eigenstate of $V_1$, the stationary well. To find the eigenstates we use the module Eigen-Solver which we have used before. The code is repeated below for convenience.

```
waveFunctionSolver[V_, x_, edge_, n_] := Module[{ϕ, L, Hψ, a, M, vector, levels},
  Clear[ϕ, Hψ, a, L, M, states, vectors, levels, functions];
  L = edge;
  ϕ[i_, x] = (√(2 / L) Sin[i π (x + L / 2) / L]);
  Hψ[i_, x] = -ħ^2 / (2 m) D[ϕ[i, x], {x, 2}] + V ϕ[i, x];
  a[j_, i_] := NIntegrate[ϕ[j, x] Hψ[i, x], {x, -L / 2, L / 2}, AccuracyGoal → 3];
numberOfBasisFunctions = n;
  ħ = 1;
  m = 1;

  M = Table[a[j, k], {j, 1, numberOfBasisFunctions}, {k, 1, numberOfBasisFunctions}];
  states = Eigenvalues[Chop[N[M]]];
  basisFunctions = Table[ϕ[k, x], {k, 1, numberOfBasisFunctions}];
vectors = Eigenvectors[Chop[N[M]]];
wavefunctions = vectors.basisFunctions;
levels = Flatten[Map[Position[states, #] &, Sort[states]]];
functions = Map[wavefunctions[[#]] &, levels];]
```

Using waveFunctionSolver we can easily find the ground state. The wave function is returned as the list functions. The ground state wave function is the first element of the list..

```
waveFunctionSolver[V₁[x], x, 10, 40]
```

Rather than deal with the long sum of trig functions it is faster to sample the function once and make an interpolating function.

```
f[x_] = functions[[1]];
wavefunction = Interpolation[Table[{x, f[x]}, {x, -5, 5, .1}]]
```

InterpolatingFunction[( −5.  5. ), <>]

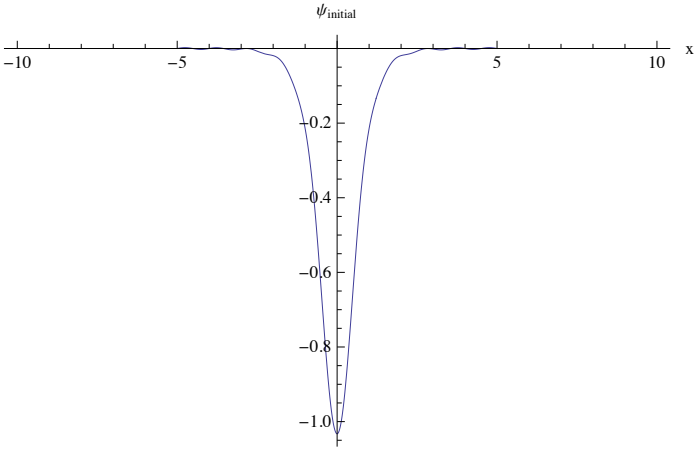
We need the wave function over a larger interval than is returned by waveFunctionSolver, but the value of the wave function is so small outside of the interval $-5 < x < 5$ that we can just set it to zero.

```
ψinitial[x_] = Piecewise[{{0, x < -5}, {wavefunction[x], -5 ≤ x ≤ 5}, {0, x > 5}}]
```

$$\begin{cases} 0 & x < -5 \\ \text{InterpolatingFunction}[(-5.\ \ 5.), <>](x) & -5 \le x \le 5 \end{cases}$$

The initial wave function is plotted below.

```
Plot[ψinitial[x], {x, -10, 10}, PlotRange → All, AxesLabel → {"x", ψinitial}]
```




Now that we have the initial wave function we can solve the wave equation with the full time-dependent potential.

```
Clear[ħ, m, ψ, V]
eqn = i ħ D[ψ[x, t], t] == - (ħ^2 / 2 m) D[ψ[x, t], {x, 2}] + V[x, t] ψ[x, t]
```

$$i\,\hbar\,\psi^{(0,1)}(x,t) = V(x,t)\,\psi(x,t) - \frac{1}{2}\,m\,\hbar^2\,\psi^{(2,0)}(x,t)$$

As usual we will solve the equation in a box and force the wave function to zero at the box boundaries. The box runs from $-L$ to $L$ and I have taken $L = 8$ in units in which $m = 1$ and $\hbar = 1$.

```
m = 1;
ħ = 1;
L = 8;
V[x_, t_] := V1[x] + V2[x, t]
```

Since we don't want the wave packet to hit the boundaries of the box I have tracked the solution only until $t = 8$. I have also used AccuracyGoal → 2. This gives a low accuracy solution (the default to AccuracyGoal → 6) but it's fast and does not take much memory. We will have to check to see if our low accuracy solution is good enough.

```
Clear[packet]
packet[x_, t_] =
 NDSolve[{eqn, ψ[x, 0] == ψinitial[x], ψ[-L, t] == 0, ψ[L, t] == 0}, ψ[x, t], {x, -L, L},
     {t, 0, 10}, MaxSteps → 100 000, AccuracyGoal → 2][[1, 1, 2]] // Timing
```

NDSolve::mxsst : Using maximum number of grid points 10000
allowed by the MaxPoints or MinStepSize options for independent variable $x$. ≫

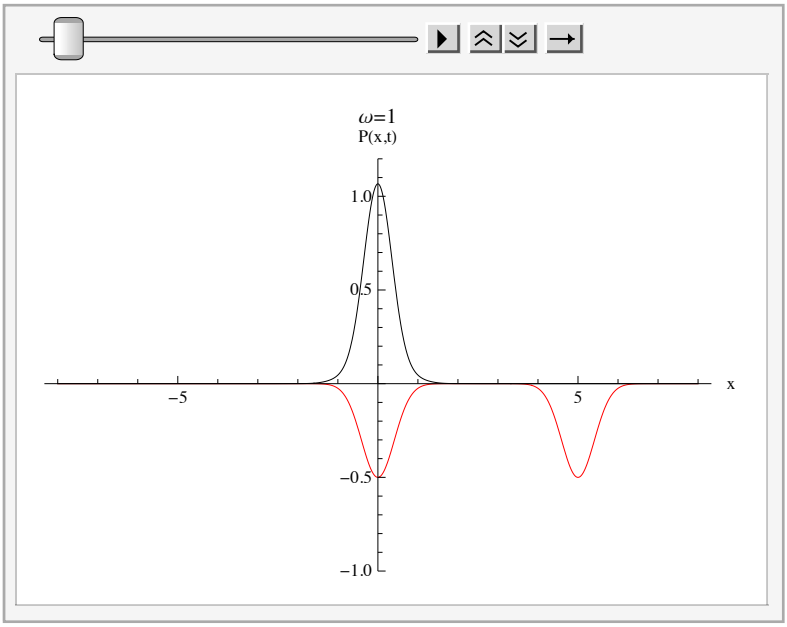
$$\left\{37.1851,\ \text{InterpolatingFunction}\left[\begin{pmatrix} -8. & 8. \\ 0. & 10. \end{pmatrix}, <>\right](x,t)\right\}$$

We can now form the wave packet

```
wave[x_, t_] := Chop[Times[packet[x, t][[2]], Conjugate[packet[x, t][[2]]]]]
```

and plot the time evolution.

```
ListAnimate[
 Table[Plot[Evaluate[{V[x, t] / 10, wave[x, t]}], {x, -L, L}, PlotRange → {-1, 1.2},
   PlotLabel → "ω=1", PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 0, 0]},
   AxesLabel → {"x", "P(x,t)"}], {t, 0, 10, .1}]]
```




Wave packet splitting. The wave packet is shown in black. The potential is shown in red.

It is important to understanding what the animation really means. Remember that we are looking at a probability distribution so we don't split the wave packet. We split the probability distribution. There is a chance that the wave packet stays in the stationary well and a chance that it moves off with the moving potential. Notice that for our potential, the probability of moving the wave packet is not very high.

To check the accuracy of our solution we can re-solve the equation with an accuracy goal of 3 and compare the solutions.

```
Clear[HPpacket]
HPpacket[x_, t_] =
 NDSolve[{eqn, ψ[x, 0] == ψinitial[x], ψ[-L, t] == 0, ψ[L, t] == 0}, ψ[x, t], {x, -L, L},
    {t, 0, 10}, MaxSteps → 100 000, AccuracyGoal → 3][[1, 1, 2]] // Timing
```

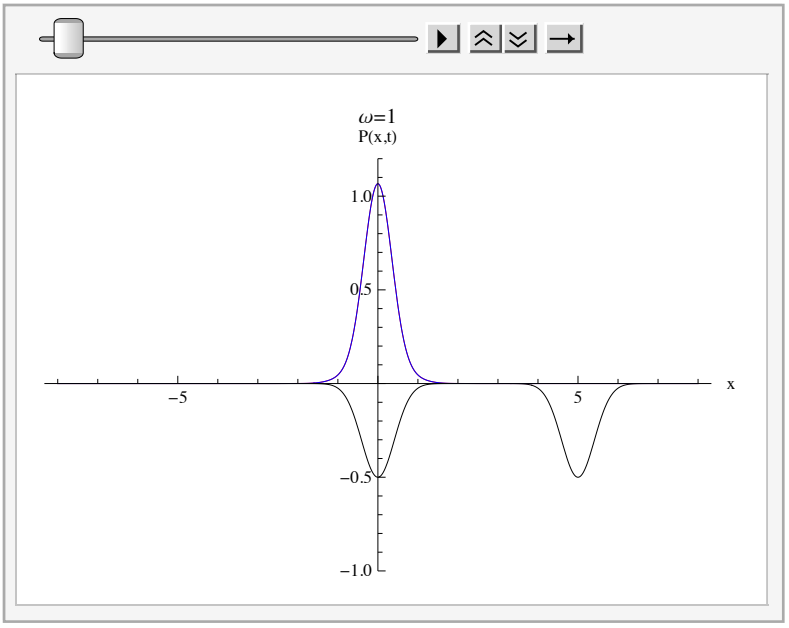
NDSolve::mxsst : Using maximum number of grid points 10000
allowed by the MaxPoints or MinStepSize options for independent variable $x$. ≫

$$\left\{49.3391, \text{InterpolatingFunction}\left[\begin{pmatrix} -8. & 8. \\ 0. & 10. \end{pmatrix}, <>\right](x, t)\right\}$$

```
HPwave[x_, t_] := Chop[Times[HPpacket[x, t][[2]], Conjugate[HPpacket[x, t][[2]]]]]
```

Plotting shows that the two solutions are very similar. You can barely see the difference between the two curves.

```
ListAnimate[Table[Plot[Evaluate[{V[x, t] / 10, HPwave[x, t], wave[x, t]}],
   {x, -L, L}, PlotRange → {-1, 1.2}, PlotLabel → "ω=1",
   PlotStyle → {Black, Red, Blue}, AxesLabel → {"x", "P(x,t)"}], {t, 0, 10, .1}]]
```



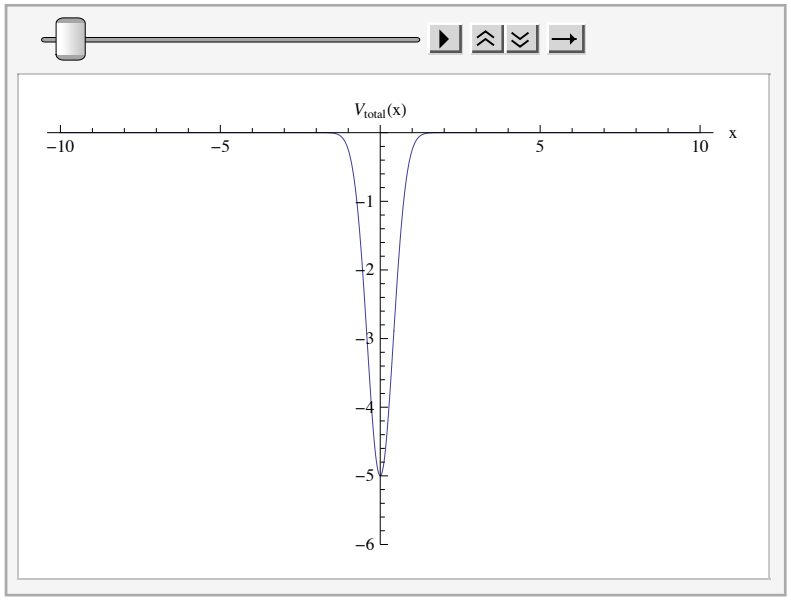

Comparing our two solutions. The two wave packet solutions are shown in red (AccuracyGoal→3) and blue (AccuracyGoal→2). Note that there is little difference between them. The potential is shown in black.

Another way to split a wave packet into two is to start from a single well and split it into two wells. This has not yet been done experimentally but it allows you to split the packet with equal probability. Suppose we have the following potential

```
V[x_, t_] := (V0 / 2) Exp[-3 (x + ω t)^2] + (V0 / 2) Exp[-3 (x - ω t)^2]
```

This potential is centered at $x = 0$ at $t = 0$ and is composed of equal parts of a left moving and right moving wave. The potential's time-dependence is plotted below for $V_0 = -5$ and $\omega = 1$.

```
ω = 1;
V0 = -5;
ListAnimate[Table[Plot[V[x, t], {x, -10, 10},
    PlotRange → {0, -6}, AxesLabel → {"x", "Vtotal(x)"}], {t, 0, 10, 0.5}]]
```




Notice that at $t = 0$ the potential is the same as our original static potential so that we can use the same initial wave shape. This makes it easy to re-solve the wave equation.

```
Clear[packet]
packet[x_, t_] =
 NDSolve[{eqn, ψ[x, 0] == ψinitial[x], ψ[-L, t] == 0, ψ[L, t] == 0}, ψ[x, t], {x, -L, L},
    {t, 0, 10}, MaxSteps → 100 000, AccuracyGoal → 2][[1, 1, 2]] // Timing
```

NDSolve::mxsst : Using maximum number of grid points 10000
allowed by the MaxPoints or MinStepSize options for independent variable $x$. ≫

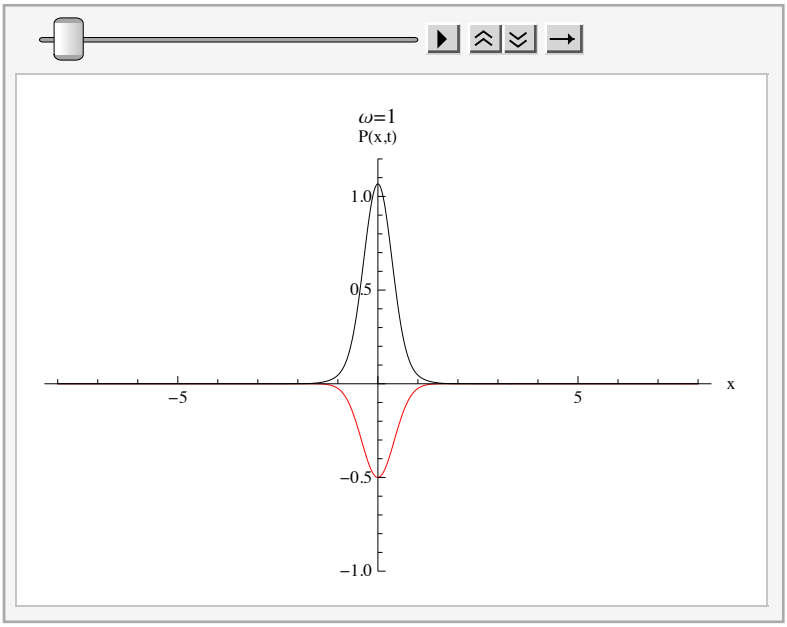
$$\left\{10.5814, \text{InterpolatingFunction}\left[\begin{pmatrix} -8. & 8. \\ 0. & 10. \end{pmatrix}, <>\right](x, t)\right\}$$

We now form the probability density

```
wave[x_, t_] := Chop[Times[packet[x, t][[2]], Conjugate[packet[x, t][[2]]]]]
```

and plot the result. To avoid effects from the box boundary we plot only until $t = 5$.

```
ListAnimate[
 Table[Plot[Evaluate[{V[x, t] / 10, wave[x, t]}], {x, -L, L}, PlotRange → {-1, 1.2},
   PlotLabel → "ω=1", PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 0, 0]},
   AxesLabel → {"x", "P(x,t)"}], {t, 0, 5, .1}]]
```




Wave packet splitting with equal probabilities. The wave packet is shown in black. The potential is shown in red.

Notice that we get equal probabilities of moving the wave packet to the right or the left.

- **Paradoxical Reflection**

An interesting wave packet phenomena is called paradoxical reflection or anti-tunneling. A notebook on this can be found here and in cdf format here and in pdf form here.

# GPU Computing

Eigenfunction expansions can also be made to run a GPU. A *Mathematica* notebook with a GPU implementation can be found her, the cdf file is here and the pdf version is here.

## Dynamics

# Chaotic Systems

- **Three Body Problems**
- The Restricted Three Body Problem

A problem of great interest is the restricted three body problem [4,5]. In the restricted three body problem, one of the masses is infinitesimal and all the masses are restricted to move in the xy-plane. We will take $m_3$ to be the infinitesimal mass. The sum of the masses $m_1$ and $m_2$ can be taken without loss of generality to be one. This leaves only one free mass parameter: $\mu = m_2/(m_1 + m_2)$. The two massive particles will rotate about their center of mass with an angular frequency of $\omega^2 = \mathrm{GM}/L^3$ where L is the separation between the two massive

planets. Recall that in the discussion of the Trojan points, the rotation of the three masses about their center of mass was independent of the masses of the particles. In particular, the result still holds when one of the masses is zero. Thus two particles will rotate about their center of mass with angular velocity $\omega^2 = \mathrm{GM}/L^3$ and in the restricted problem we can take M=1. This suggests working in a synodic or rotating coordinate system which rotates along with the center of mass of the system. The synodic coordinates are related to the sidereal (inertial) coordinates by

$\bar{x} = x\cos(\omega t) - y\sin(\omega t)$,

$\bar{y} = x\sin(\omega t) + y\cos(\omega t)$,

where the bared coordinates are coordinates in the sidereal system and x and y are coordinates in the synodic coordinate system. The equations of motion in the synodic coordinate system are

$\ddot{x} - 2\dot{y} = \frac{\partial \Omega}{\partial x}$

and

$\ddot{y} + 2\dot{x} = \frac{\partial \Omega}{\partial y}$

where

$\Omega = \frac{1}{2}\left[(1-\mu)\,r_1^2 + \mu r_2^2\right] + \frac{(1-\mu)}{r_1} + \frac{\mu}{r_2}$,

and $r_1$ and $r_2$ are the distances from $m_1$ and $m_2$ respectively to the infinitesimal mass. The dots denote derivatives with respect to a dimensionless time $t = \omega t'$ where $t'$ is the normal time. The distances $x$ and $y$ are dimensionless and are $x = x'/\mathrm{L}$ and $y = y'/\mathrm{L}$ where $x'$ and y' are the normal (that is with dimensions) distances. The distances $r_1$ and $r_2$ are

$r_1 = \left[(x-\mu)^2 + y^2\right]^{1/2}$

and

$r_2 = \left[(x-\mu+1)^2 + y^2\right]^{1/2}$.

The equations of motion admit a first integral known as the Jacobian integral. This first integral is

$\dot{x}^2 + \dot{y}^2 = 2\,\Omega - C$

where C is the Jacobian constant of integration which is constant throughout the motion.

## ■ Periodic Orbits

We want to look for periodic orbits in the restricted three body problem. The task of finding periodic orbits has received extensive attention [4, 5] and remains an area of active research [6]. We will specialize to the Copenhagen problem which is the $\mu$=1/2 case. There are many periodic orbits for the Copenhagen problem but none of them are stable, which makes finding them difficult. We will work in the synodic coordinate system in dimensionless coordinates. Since we will be interested in both the orbits and the phase space plots, as well as the Jacobian constant, we will solve for $x$, $y$, $v_x$ and $v_y$. The two primaries (massive bodies) are at $\{x=-1/2, y=0\}$ and $\{x=1/2, y=0\}$. The *Mathematica* code to generate and solve the equations is straightforward. We look at two orbits from Moulton's family of periodic orbits. These orbits are retrograde orbits around the $L_4$ and $L_5$. The first members of this family were found by Moulton [7] in 1920. The orbits were further investigated by Strömgren [8] and the family of orbits was established by Szebehely and Van Flandern [9] in 1966. The starting conditions for these orbits are $x(0)=0$, $y(0) = y_0$, $v_x(0) = v_{x_0}$ and $v_y(0)=0$. Each individual orbit in the family is characterized by the value of the Jacobian constant.

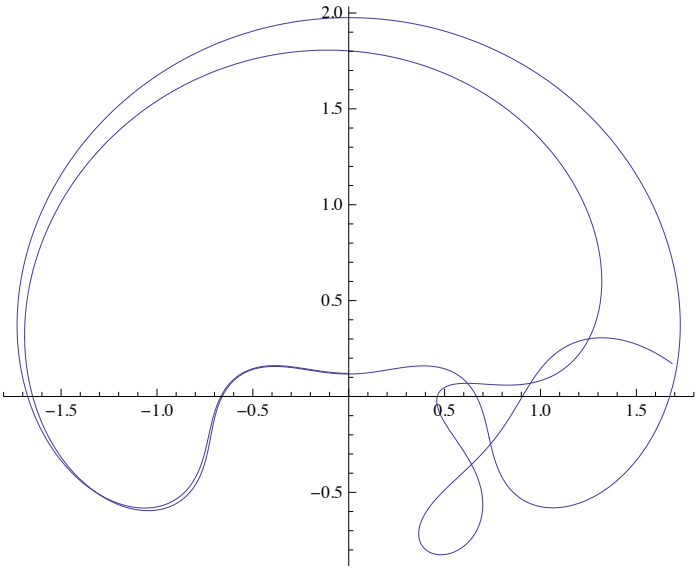
```
Clear[Omega,r1,r2,X,Y,mu,Vx,Vy]
Omega=(1/2)((1-mu)r1^2+mu r2^2)+mu/r2+(1-mu)/r1;
r1=((X[t]-mu)^2+Y[t]^2)^(1/2);
r2=((X[t]+1-mu)^2+Y[t]^2)^(1/2);
VxEquation=Simplify[
    D[Vx[t],t]-2Vy[t]-D[Omega,X[t]]];
VyEquation=Simplify[
    D[Vy[t],t]+2Vx[t]-D[Omega,Y[t]]];
xEquation=D[X[t],t]-Vx[t];
yEquation=D[Y[t],t]-Vy[t];
```

Having generated the equations of motion we can use the starting values $v_{x_0}$ and $y_0$ that Szebehely and Van Flandern [9] give for the C=3.02919 orbit. The starting values are $v_{x_0}$=-1.06006 and $y_0$=0.12067. The solution of the differential equation is found using NDSolve. Plotting the orbits gives a perhaps unexpected result.

```
Clear[X, Y, Vx, Vy]
```

$$\text{mu} = \frac{1}{2};$$

```
InitialConditions = {X[0] == 0, Vx[0] == -1.06006`, Y[0] == 0.12067`, Vy[0] == 0};
EquationList = Join[{VxEquation == 0}, {VyEquation == 0},
  {xEquation == 0}, {yEquation == 0}, InitialConditions];
Orbit = NDSolve[EquationList, {X[t], Y[t], Vx[t], Vy[t]}, {t, 0, 20}, MaxSteps → 6000];
X[t_] = First[X[t] /. Orbit]; Y[t_] = First[Y[t] /. Orbit];
Vx[t_] = First[Vx[t] /. Orbit]; Vy[t_] = First[Vy[t] /. Orbit];
ParametricPlot[{X[t], Y[t]}, {t, 0, 20}, AspectRatio → Automatic, PlotPoints → 100]
```



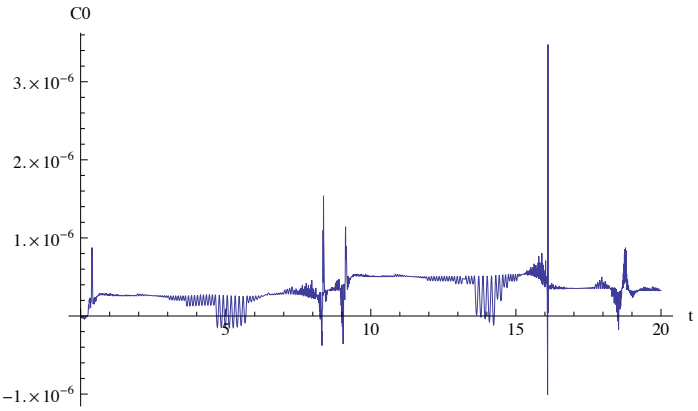

Notice that the orbit is outside both of the primaries, and that something has gone wrong the second time we traverse the orbit. The orbit itself is both amazing and beautiful. It is in a sense typical of this family of orbits which all lie outside the primaries. It is the simplest member of the family and we will find a more elaborate orbit shortly. Before we can look for more complicated orbits, we must address the issue of what went wrong on the second traversal of the orbit. The root cause of the problem is easy to identify: we are integrating a chaotic system, and if we integrate it for long enough time intervals, we are certain to run into trouble. Readers with an interest in the numerical analysis of chaotic systems should consult Parker and Chua [10].

We need to investigate the problem more closely and see if we can maintain accuracy for two orbits. If we want to integrate some of the more complicated orbits (which have longer periods) for a whole period, we need to be able to follow this orbit for two periods. There are two sources of possible trouble: our integrator might not be accurate enough or our initial conditions may not be accurate enough.

We can check the accuracy of our integrator by looking at the change in the Jacobian constant *C*. The Jacobian

constant is a constant of the motion and should in theory not change at all. However, no numerical integration is exact and so $C$ will not, in practice, be exactly conserved. Let $C_0$=$C$ at time t=0 and $\delta C = (C - C_0)/C_0$. Plotting $\delta C$ gives:

```
C0 = 2 Omega - Vx[t]^2 - Vy[t]^2 /. t → 0;
deltaC = (2 Omega - Vx[t]^2 - Vy[t]^2 - C0)/C0;
Plot[Evaluate[deltaC], {t, 0, 20}, PlotRange → All, AxesLabel → {"t", "C0"}]
```




The plot shows that $\delta C$ is not controlled very well, and ranges from about $4 \times 10^{-6}$ to $-2 \times 10^{-6}$. But $\delta C$ does not grow with time and is thus unlikely to be the culprit. We must therefore turn our attention to the initial conditions. Before looking at the initial conditions, a few remarks about $\delta C$ are in order. The control of $\delta C$ that we get by using NDSolve is adequate for this example, but long term integrations will require better control of $\delta C$. This can be achieved by writing a custom integrator that uses an adaptive method to control $\delta C$. Using an Adams – Bashforth – Moulton method [11] with a variable step size controlled by $\delta C$ will yield $\delta C$'s on the order of $10^{-8}$. Using an adaptive Bulirsch-Stoer method[12,13] yields $\delta C$'s on the order of $10^{-12}$. Although these methods are more accurate than NDSolve, they are also slower. However, for very long term integrations the Bulirsch-Stoer method, along with other new methods such as the Bettis method [14], are the methods of choice.

The initial conditions that Szebehely and Van Flandern give are in fact not accurate enough to use in computing a second orbit. In fact, Szebehely and Van Flandern did not use them to compute even a single orbit. The orbit is symmetric with respect to reflection about the y-axis, so it is sufficient to compute the orbit for half a period and then reflect it about the y-axis to get the entire orbit. This very symmetry enables us to find better initial conditions. In order for the orbit to be symmetric with respect to reflection about the y-axis, the $y$ component of the velocity must be zero at $x$=0. To find better initial conditions, we keep $y(0)$ fixed and vary $v_x(0)$ so as to make vy zero at x=0. The function "FindVy" computes the value of $v_y$ at $x$=0 given a value of $v_x(0)$. This is done by finding the orbit for the given starting conditions and then using FindRoot to find the time at which the orbit crosses the y-axis. This time is then substituted into $v_y(t)$ to find $v_y$ when x=0. An initial guess for t is needed. It is called "tStart". An initial guess of $t$=4 is used to ensure that the time found is not $t$=0 (when of course, $x$ is also zero). This particular orbit has a period of T=8.7645 so $t$=4 is a good guess [9].

```
SetSystemOptions["EvaluateNumericalFunctionArgument" → False];
```

```
FindVy[VxStart_,tGuess_]:=Module[{},
Clear[X,Y,Vx,Vy];
mu=1/2;
InitialConditions={X[0]==0,Vx[0]==VxStart,
    Y[0]==0.12067,Vy[0]==0};
EquationList=Join[{VxEquation==0},{VyEquation==0},
    {xEquation==0},{yEquation==0},
    InitialConditions];
Orbit=NDSolve[EquationList,{X[t],Y[t],Vx[t],Vy[t]},
    {t,0,5},
    MaxSteps->6000];
X[t_]=First[X[t]/.Orbit];
Y[t_]=First[Y[t]/.Orbit];
Vx[t_]=First[Vx[t]/.Orbit];
Vy[t_]=First[Vy[t]/.Orbit];
solution=FindRoot[X[t]==0,{t,tGuess}];
Vy=Vy[t]/.solution]
```

FindRoot can now be used to find a "VxStart" that makes $v_y$(x=0) zero. On a PowerMac 8500 this takes a little over 1.8 seconds,

```
N[FindRoot[FindVy[VxStart,4]==0,
    {VxStart,-1.06007,-1.060075},AccuracyGoal->15,WorkingPrecision->30],15]//Timing
```

FindRoot::brdig : The root has been bracketed as closely as possible with

30. working digits but the function value exceeds the absolute tolerance $1.\times 10^{-15}$. ≫

{1.64291, {VxStart → −1.06007120731063}}

and yields a $v_y$(x=0) of $-6\,\text{x}10^{-14}$

```
FindVy[-1.06007120731063,4]
```

$-6.19643\times 10^{-14}$

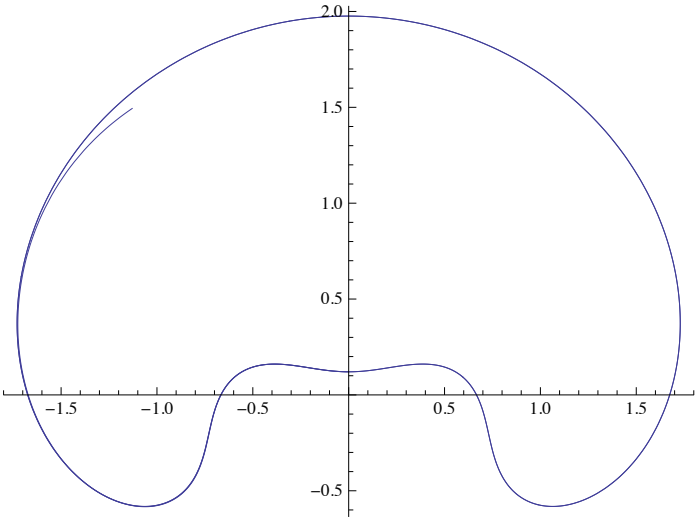
compared to a value of $4\,\text{x}10^{-5}$ obtained by using Szebehely and Van Flandern's value for $v_x$(0).

```
FindVy[-1.06007,4]
```

−0.000042784

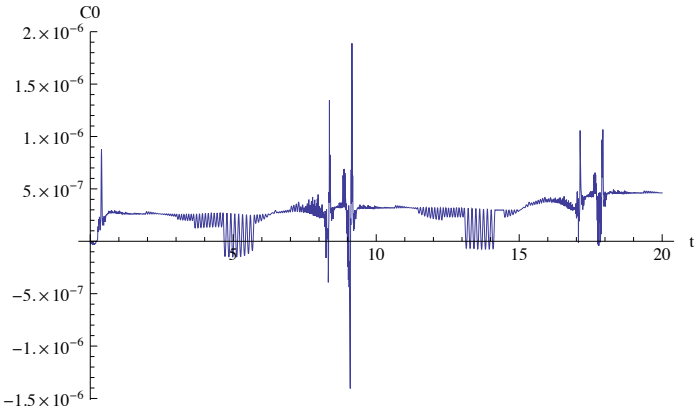
Replotting the orbit with our new initial conditions gives a much better result. We are now accurate over two orbits but not three.

```
Clear[X, Y, Vx, Vy]
mu = 1/2; InitialConditions =
 {X[0] == 0, Vx[0] == -1.06007120731063, Y[0] == 0.12067`, Vy[0] == 0};
EquationList = Join[{VxEquation == 0}, {VyEquation == 0},
  {xEquation == 0}, {yEquation == 0}, InitialConditions];
Orbit = NDSolve[EquationList, {X[t], Y[t], Vx[t], Vy[t]}, {t, 0, 20}, MaxSteps → 6000];
X[t_] = First[X[t] /. Orbit]; Y[t_] = First[Y[t] /. Orbit];
Vx[t_] = First[Vx[t] /. Orbit]; Vy[t_] = First[Vy[t] /. Orbit];
ParametricPlot[{X[t], Y[t]}, {t, 0, 21}, AspectRatio → Automatic, PlotPoints → 100]
```




The variation of the Jacobian constant is about the same.

```
C0 = 2 Omega - Vx[t]^2 - Vy[t]^2 /. t → 0; deltaC = (2 Omega - Vx[t]^2 - Vy[t]^2 - C0)/C0;
Plot[Evaluate[deltaC], {t, 0, 20}, PlotRange → All, AxesLabel → {"t", "C0"}]
```




## ■ Electrostatic Traps?

Suppose we have four equal charges at the corners of a square of length d.

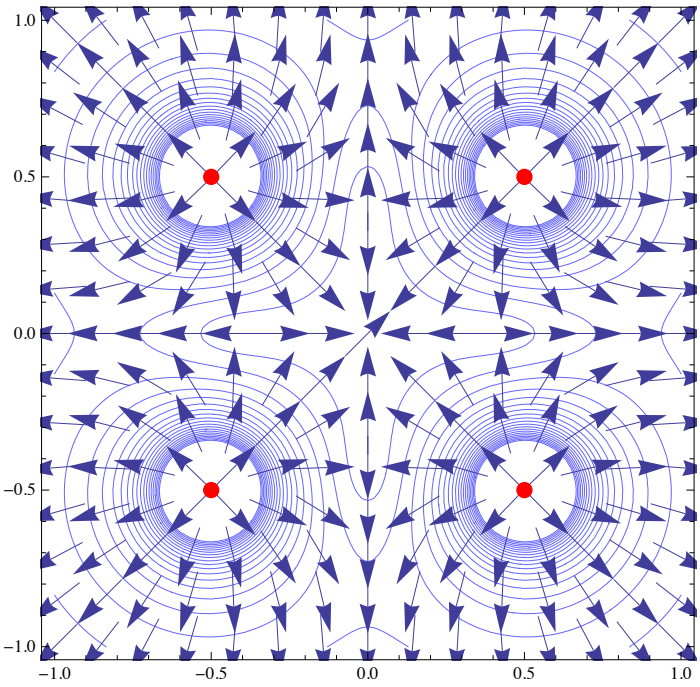
```
Clear[q1, q2, q3, q4, d]
FourCharges = {{q1, {-d/2, -d/2}}, {q2, {d/2, -d/2}}, {q3, {-d/2, d/2}}, {q4, {d/2, d/2}}};
q1 = 4 π ϵ0;
q2 = q1;
q3 = q1;
q4 = q1;
d = 1;
```

The code below compute the electric field and potential from an arbitrary configuration of charges. The function ShowElectricFiled can then be used to visualize the electric field and the charges.

```
Clear[DrawCharges]
Clear[TotalEfield]
Clear[MagnitudeEfield]
Clear[Distance]
Distance[x_, y_, points_List] := Table[Sqrt[(x - points[[i, 1]]) ^ 2 +
    (y - points[[i, 2]]) ^ 2], {i, 1, Length[points]}]
Clear[SinesAndCosines]
SinesAndCosines[x_, y_, points_List] := Table[
    {(x - points[[i, 1]]) / Distance[x, y, points][[i]],
    (y - points[[i, 2]]) / Distance[x, y, points][[i]]},
        {i, 1, Length[points]}]
Clear[Efields]
Efields[x_, y_, charges_List] :=
 Table[(charges[[i, 1]] SinesAndCosines[x, y, {charges[[i, 2]]}]) /
   ((4 π ϵ0) Distance[x, y, {charges[[i, 2]]}]^2), {i, 1, Length[charges]}]
MagnitudeEfield[x_, y_, charges_List] :=
    Sqrt[TotalEfield[x, y, charges][[1]] ^ 2 +
    TotalEfield[x, y, charges][[2]] ^ 2]
TotalEfield[x_, y_, charges_List] :=
    {Sum[Efields[x, y, charges][[i, 1, 1]], {i, 1, Length[charges]}],
        Sum[Efields[x, y, charges][[i, 1, 2]],
            {i, 1, Length[charges]}]}
EFieldDirection[x_, y_, charges_List] := ArcTan[
    TotalEfield[x, y, charges][[2]] / TotalEfield[x, y, charges][[1]]]
Clear[Potential]
Potential[x_, y_, charges_List] := First[
  Sum_{j=1}^{Length[charges]} Table[charges[[i, 1]] / ((4 π ϵ0) Distance[x, y, {charges[[i, 2]]}]), {i, 1, Length[charges]}][[j]]]
DrawCharges[charges_List, options___] := Show[Graphics[
   Table[{PointSize[0.025], If[Evaluate[N[charges[[i, 1]] /. ϵ0 → 1]] > 0, RGBColor[1, 0, 0],
      RGBColor[0, 1, 0]], Point[charges[[i, 2]]]}, {i, 1, Length[charges]}]], options]
Clear[ShowElectricField]
ShowElectricField[Charges_List, {x_, xmin_, xmax_}, {y_, ymin_, ymax_}, options___] :=
 Module[{GrayColorFunction, plot1, plot2, plot3},
  GrayColorFunction[z_] = RGBColor[.4, .4, .4];
  plot1 = VectorPlot[Evaluate[TotalEfield[x, y, Charges] / MagnitudeEfield[x, y, Charges]],
    {x, xmin, xmax}, {y, ymin, ymax}, ColorFunction → GrayColorFunction]; plot2 =
   ContourPlot[Evaluate[MagnitudeEfield[x, y, Charges]], {x, xmin, xmax}, {y, ymin, ymax},
    options, ContourShading → None, ContourStyle → Lighter[Blue, .4], Contours → 20];
  plot3 = DrawCharges[Charges]; Show[plot2, plot1, plot3]]
```

Here is what the field from our four charge configuration looks like.

```
plot1 = ShowElectricField[FourCharges, {x, -1, 1}, {y, -1, 1}]
```



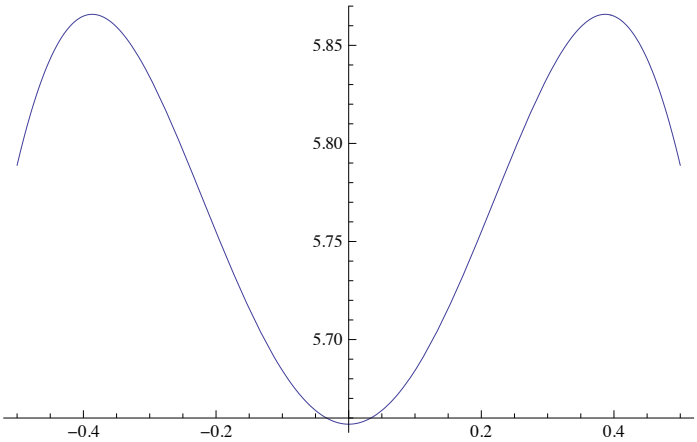

Notice that the electric field in the center is zero. Can we use this to make an electrostatic trap? The potential that a test charge would see is

```
Well = Potential[x, y, FourCharges]
```

$$\frac{1}{\sqrt{\left(x+\frac{1}{2}\right)^2+\left(y-\frac{1}{2}\right)^2}}+\frac{1}{\sqrt{\left(x-\frac{1}{2}\right)^2+\left(y+\frac{1}{2}\right)^2}}+\frac{1}{\sqrt{\left(x+\frac{1}{2}\right)^2+\left(y+\frac{1}{2}\right)^2}}+\frac{1}{\sqrt{\left(x-\frac{1}{2}\right)^2+\left(y-\frac{1}{2}\right)^2}}$$

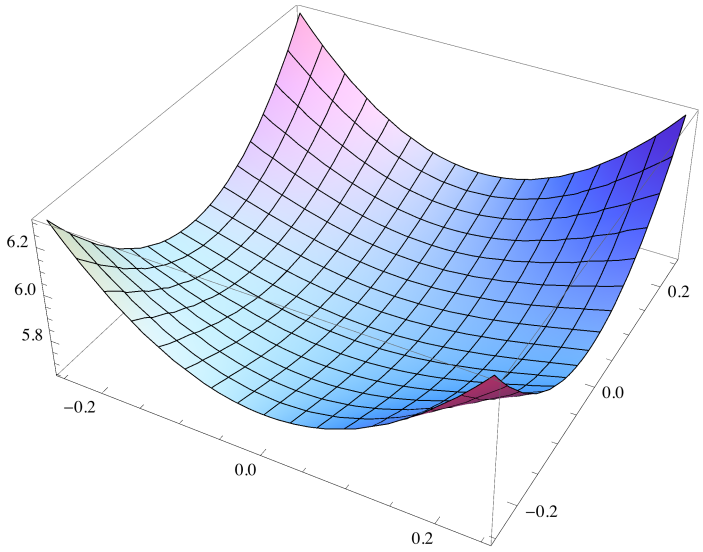
Near the center it looks like a harmonic oscillator potential. Below is a plot of the potential along the x-axis.

```
Plot[Well /. y -> 0, {x, -1 / 2, 1 / 2}]
```



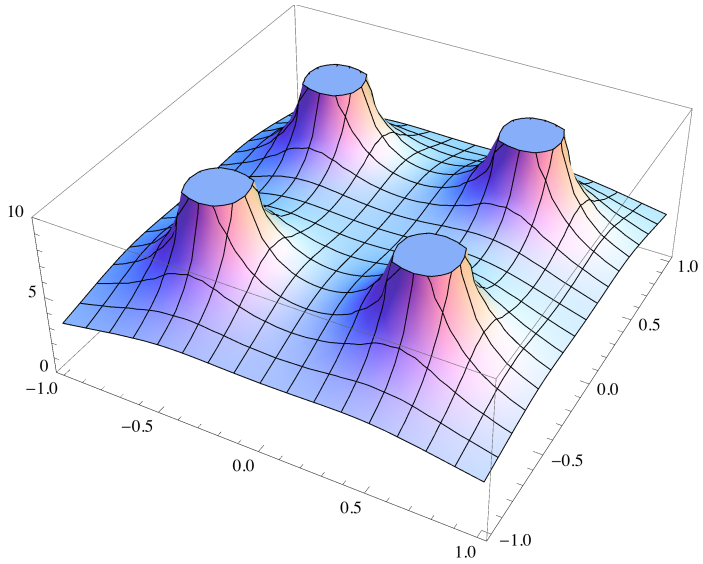

In two dimensions the potential looks near the center looks like

```
Plot3D[Well, {x, -1 / 4, 1 / 4}, {y, -1 / 4, 1 / 4}]
```




but globally the potential is much more complicated.

```
Plot3D[Well, {x, -1, 1}, {y, -1, 1}, PlotRange → {0, 10}]
```




It looks like a nice stable well (at least near the center) and so it would appear that we have a nice electrostatic trap. But we don't and we can't. You cannot make an electrostatic trap from fixed arrangement of charges. If the trapped charge is to be in equilibrium the electric field on it must be zero. However, if the equilibrium is to be stable all the lines of flux into a small sphere (Gaussian surface) surrounding the charge must flow inward toward the charge. But by Gauss' law if all the flux lines are inward the electric field cannot be zero. That is the general argument but how does our particular case fail? It fails because we have looked only at motion in the x-y plane. What about motion in the z plane (that is into or out of the computer screen). For motion in the z-plane our trap is not stable. As soon as the particle moves away from the z=0 plane it feels a positive electric field and is thus not in equilibrium.

From looking at the potential we expect that if a test charge is near the center it should undergo simple harmonic motion but that if the excursions from the center are large enough the motion will be complicated. The force on a test charge is q**E** so the force on a test charge in our four charge electric field is

```
qE = (qtest Simplify[
TotalEfield[x, y, FourCharges]]) /. {x -> x[t], y -> y[t]}
```

$$\left\{\text{qtest}\left(\frac{x(t)-\frac{1}{2}}{\left(\left(x(t)-\frac{1}{2}\right)^2+\left(y(t)-\frac{1}{2}\right)^2\right)^{3/2}}+\frac{x(t)-\frac{1}{2}}{\left(x(t)^2-x(t)+y(t)^2+y(t)+\frac{1}{2}\right)^{3/2}}+\right.\right.$$
$$\left.\frac{x(t)+\frac{1}{2}}{\left(x(t)^2+x(t)+y(t)^2-y(t)+\frac{1}{2}\right)^{3/2}}+\frac{x(t)+\frac{1}{2}}{\left(x(t)^2+x(t)+y(t)^2+y(t)+\frac{1}{2}\right)^{3/2}}\right),$$
$$\text{qtest}\left(\frac{y(t)-\frac{1}{2}}{\left(\left(x(t)-\frac{1}{2}\right)^2+\left(y(t)-\frac{1}{2}\right)^2\right)^{3/2}}+\frac{y(t)-\frac{1}{2}}{\left(x(t)^2+x(t)+y(t)^2-y(t)+\frac{1}{2}\right)^{3/2}}+\right.$$
$$\left.\left.\frac{y(t)+\frac{1}{2}}{\left(x(t)^2-x(t)+y(t)^2+y(t)+\frac{1}{2}\right)^{3/2}}+\frac{y(t)+\frac{1}{2}}{\left(x(t)^2+x(t)+y(t)^2+y(t)+\frac{1}{2}\right)^{3/2}}\right)\right\}$$

where we have made x and y functions of t by using a replacement rule. The equations of motion are

```
Xequation = D[x[t], {t, 2}] == qE[[1]]
```

$$x''(t)=\text{qtest}\left(\frac{x(t)-\frac{1}{2}}{\left(\left(x(t)-\frac{1}{2}\right)^2+\left(y(t)-\frac{1}{2}\right)^2\right)^{3/2}}+\right.$$
$$\left.\frac{x(t)-\frac{1}{2}}{\left(x(t)^2-x(t)+y(t)^2+y(t)+\frac{1}{2}\right)^{3/2}}+\frac{x(t)+\frac{1}{2}}{\left(x(t)^2+x(t)+y(t)^2-y(t)+\frac{1}{2}\right)^{3/2}}+\frac{x(t)+\frac{1}{2}}{\left(x(t)^2+x(t)+y(t)^2+y(t)+\frac{1}{2}\right)^{3/2}}\right)$$

and

```
Yequation = D[y[t], {t, 2}] == qE[[2]]
```

$$y''(t)=\text{qtest}\left(\frac{y(t)-\frac{1}{2}}{\left(\left(x(t)-\frac{1}{2}\right)^2+\left(y(t)-\frac{1}{2}\right)^2\right)^{3/2}}+\right.$$
$$\left.\frac{y(t)-\frac{1}{2}}{\left(x(t)^2+x(t)+y(t)^2-y(t)+\frac{1}{2}\right)^{3/2}}+\frac{y(t)+\frac{1}{2}}{\left(x(t)^2-x(t)+y(t)^2+y(t)+\frac{1}{2}\right)^{3/2}}+\frac{y(t)+\frac{1}{2}}{\left(x(t)^2+x(t)+y(t)^2+y(t)+\frac{1}{2}\right)^{3/2}}\right)$$

These differential equations can not be solved exactly but we can solve them numerically. *Mathematica* has a function called **NDSolve** which solves differential equations numerically. To use NDSolve you give it a list of equations which must include the initial conditions as well as the differential equation. You must then give **NDSolve** a second list containing the dependent variables. The last step is to specify the independent variable and the range over which you want to solve the differential equation. in the example below we start our particle at the origin with a small initial velocity.

```
m = 1;
qtest = 1;
solution = NDSolve[{Xequation, Yequation, x[0] == 0,
    x'[0] == .1, y[0] == 0, y'[0] == .1}, {x, y}, {t, 0, 40}];
```

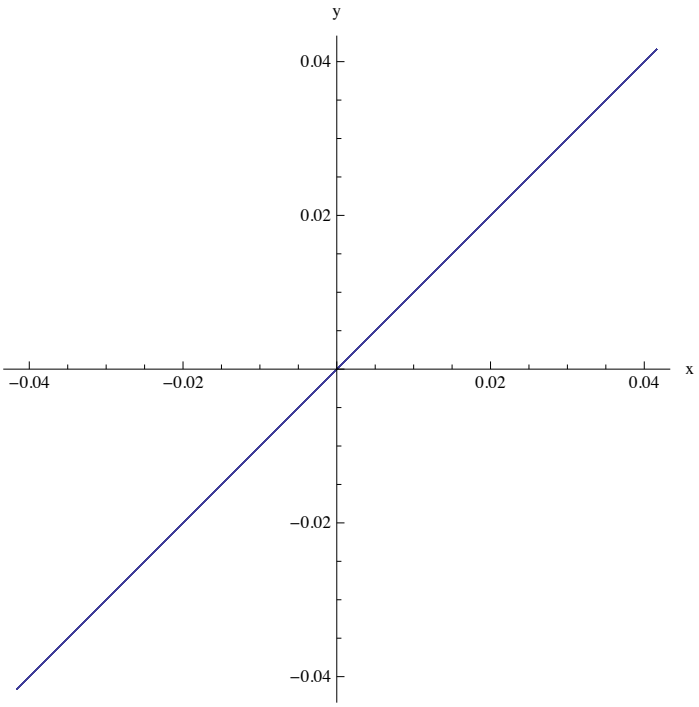
The result is returned as a set of interpolating functions. You can use the interpolating functions just as you would any other *Mathematica* function.

```
motion[t_] = First[{x[t], y[t]} /. solution]
```

{InterpolatingFunction[{ 0.  40. }, <>](*t*), InterpolatingFunction[{ 0.  40. }, <>](*t*)}

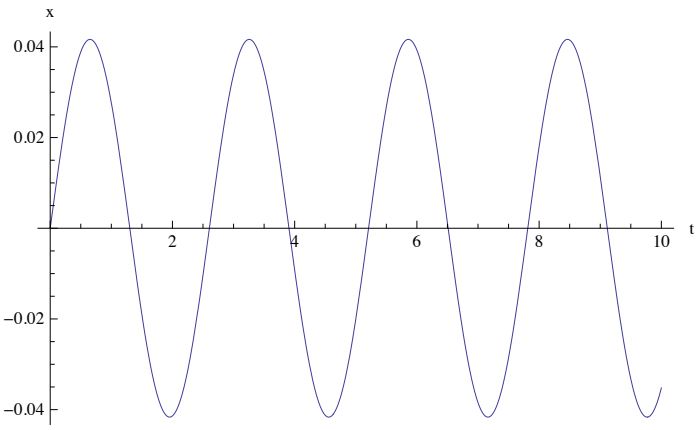
We can use Parametric plot to plot the motion as a function of time.

```
ParametricPlot[Evaluate[motion[t]], {t, 0, 40},
    AxesLabel -> {"x", "y"}]
```

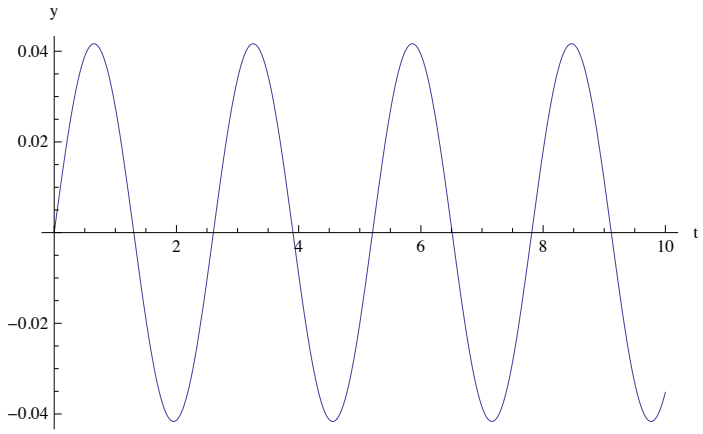



The particle just moves back and forth along the line. In fact it undergoes simple harmonic motion as you can see by plotting the x and y components against time.

```
Plot[motion[t][[1]], {t, 0, 10}, AxesLabel -> {"t", "x"}]
```



```
Plot[motion[t][[2]], {t, 0, 10}, AxesLabel -> {"t", "y"}]
```

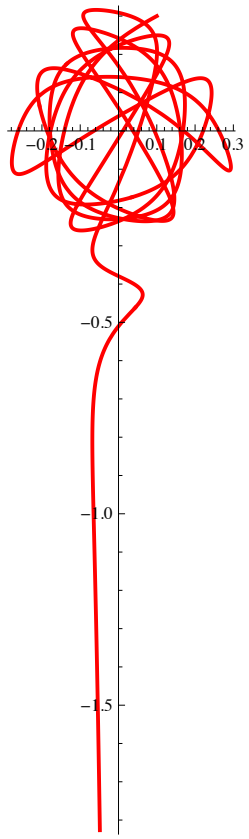



Suppose we start our particle with just enough energy to make out of the well.

```
m = 1;
qtest = 1;
solution = NDSolve[{Xequation, Yequation, x[0] == .1,
    x'[0] == 0, y[0] == .3, y'[0] == 0}, {x, y}, {t, 0, 40},
    MaxSteps -> 5000];
motion[t_] = First[{x[t], y[t]} /. solution];
```

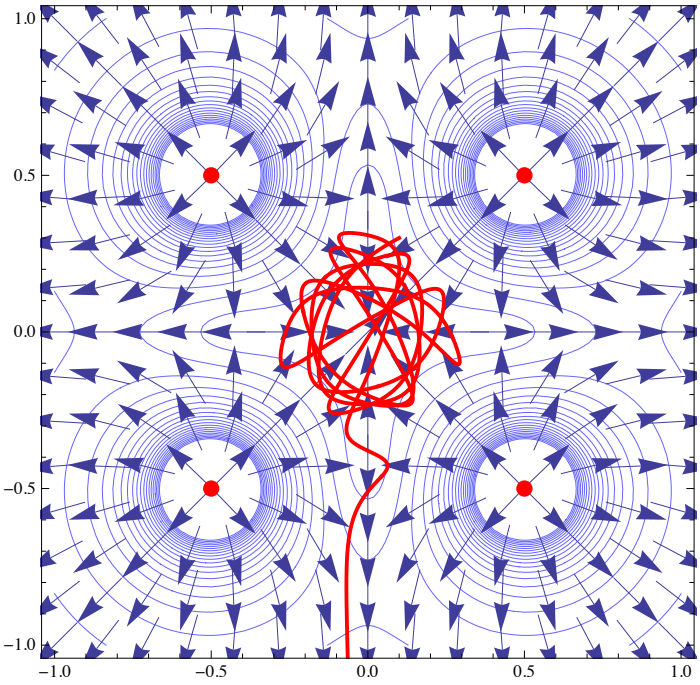
Let's look at the motion for the first 19.5 seconds. It's very complicated.

```
plot2 = ParametricPlot[Evaluate[motion[t]],
  {t, 0, 20}, PlotRange -> All, PlotStyle → {Thick, Red}]
```



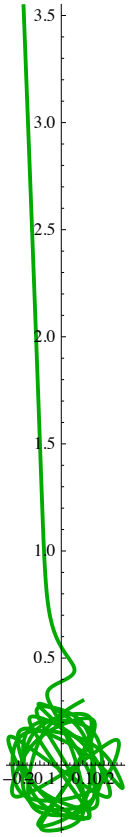
Here we see the particle's motion along with the charges and the electric field. The particle has enough energy to escape but it has to find its way out. It rattles around before it finds a gap in the charges.

```
Show[plot1, plot2]
```



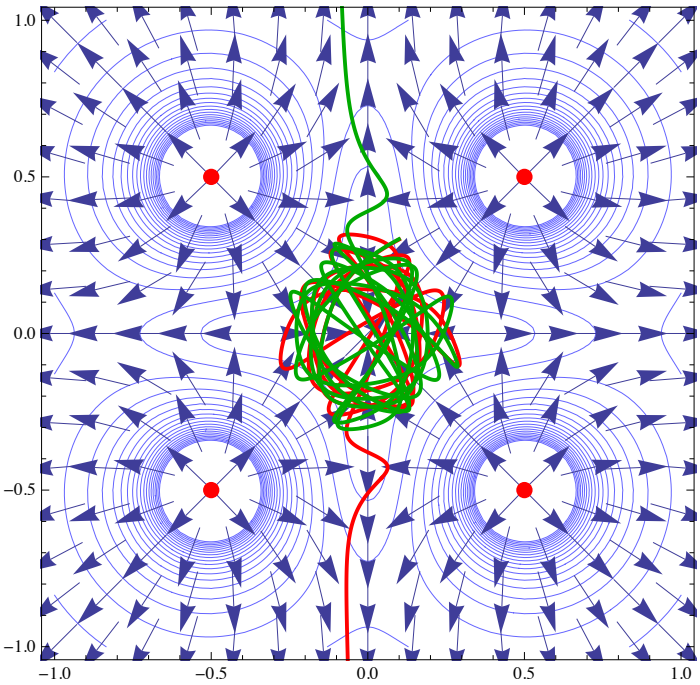

Now lets start our particle with slightly different starting conditions. Instead of x=0.1 I have used x=0.099 and instead of y=0.3 I have used y=0.301.

```
m = 1;
qtest = 1;
solution = NDSolve[{Xequation, Yequation, x[0] == .099,
    x'[0] == 0, y[0] == .301, y'[0] == 0}, {x, y}, {t, 0, 40},
    MaxSteps -> 5000];
motion[t_] = First[{x[t], y[t]} /. solution];
plot3 = ParametricPlot[Evaluate[motion[t]], {t, 0, 29},
PlotRange -> All, PlotStyle → {Thick, Darker[Green]}]
```




The motion is completely different! Lets put both trajectories together on the same plot.

```
Show[plot1,plot2,plot3]
```

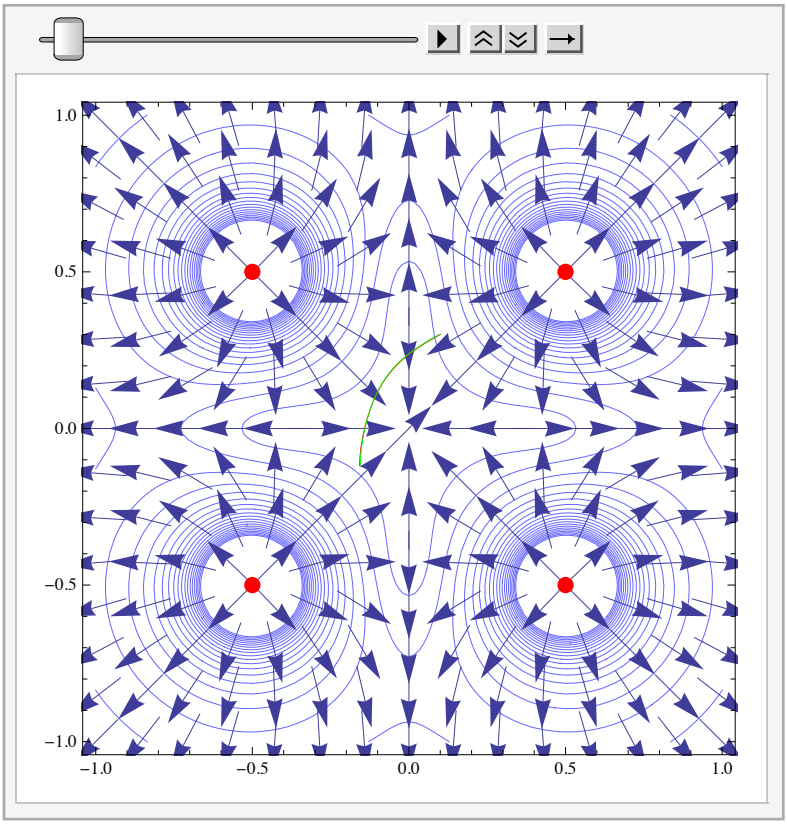



Both particles have almost the same starting conditions and yet after a long time they have completely different motions. We can get a better feel for this with an animation.

```
m = 1;
qtest = 1;
solution2 = NDSolve[{Xequation, Yequation, x[0] == .099,
    x'[0] == 0, y[0] == .301, y'[0] == 0}, {x, y}, {t, 0, 40},
    MaxSteps -> 5000];
motion2[t_] = First[{x[t], y[t]} /. solution2];
```

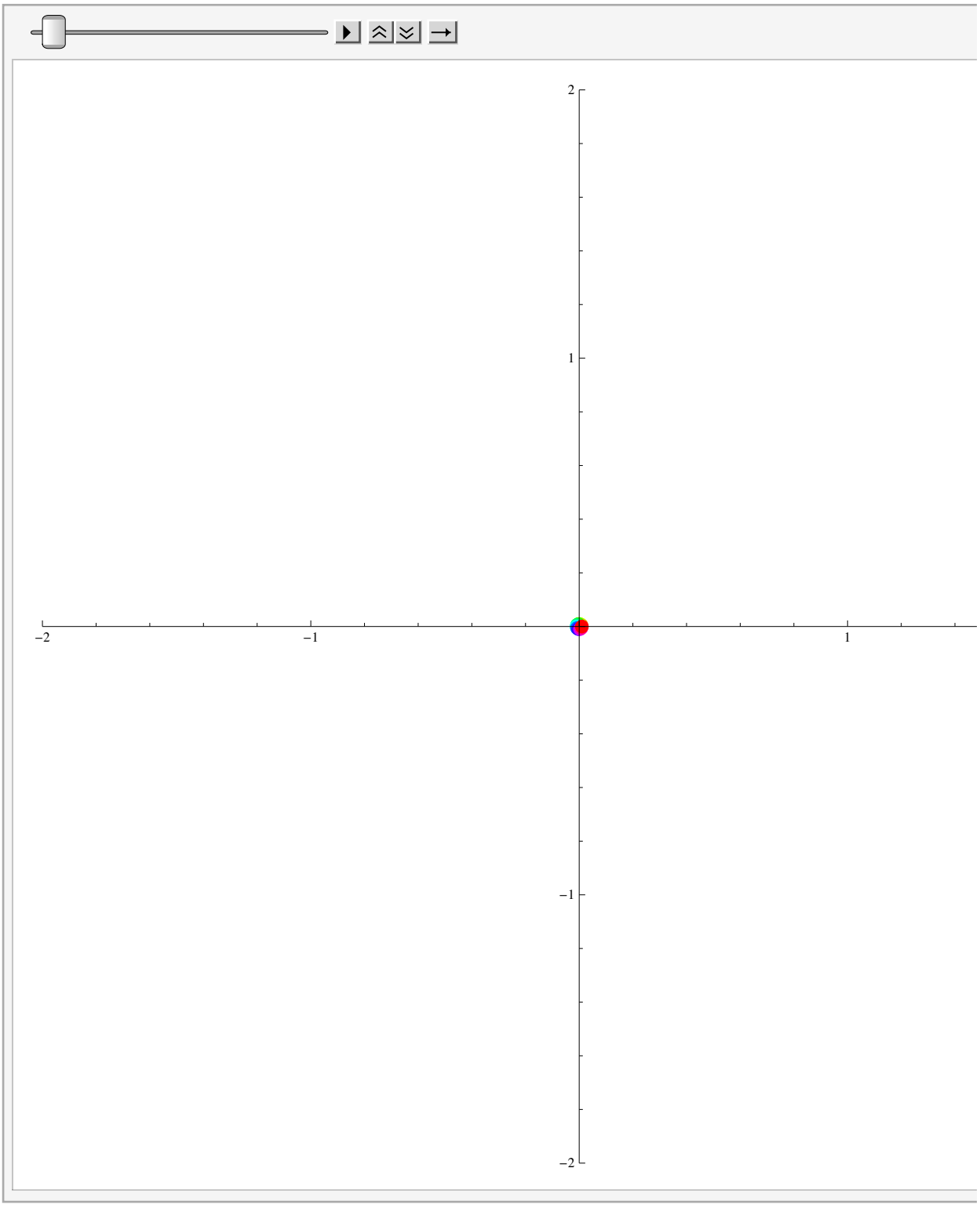
```
solution = NDSolve[{Xequation, Yequation, x[0] == 0.1`, x'[0] == 0, y[0] == 0.3`, y'[0] == 0},
   {x, y}, {t, 0, 40}, MaxSteps → 5000];
motion[t_] = First[{x[t], y[t]} /.solution];
```

```
ListAnimate[Table[Show[plot1,
    ParametricPlot[Evaluate[motion[t]], {t, 0, tmax}, PlotRange → {{-0.5, 0.5}, {-0.5, 0.5}},
     PlotStyle → Red], ParametricPlot[Evaluate[motion2[t]], {t, 0, tmax},
     PlotRange → {{-0.5, 0.5}, {-0.5, 0.5}}, PlotStyle → Green]], {tmax, 1, 19.5, 0.1}]]
```




## ■ Pitfalls

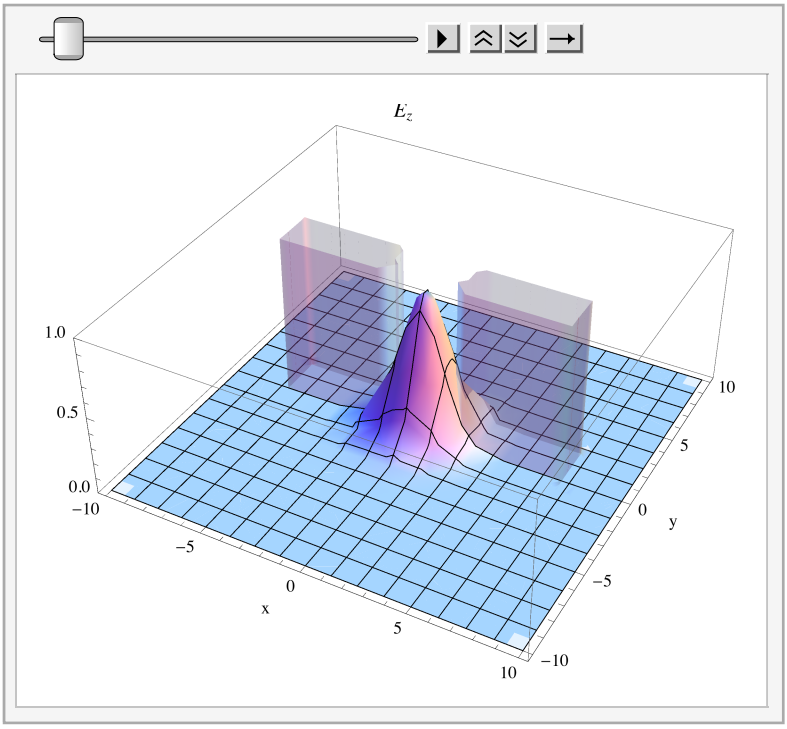
This was handed in by a student in computational physics. Ten masses equally spaced around a ring are thrown out outward. The mass then move under the influence of gravity except that the potential is cut-off to prevent it from going to infinity. That is $1/r \to 1/(r+\epsilon)$. The initial motion is correct but the system is chaotic and small errors in the integration lead to disaster. Changing the accuracy goal on the ODE solver gives a completely different result. This is inevitable in a chaotic system but the student did not realize what was going on.


2
1
−1
−2
−2
−1
1

Electrodynamics

# FDTD Methods

- **Scattering through a dielectric slit**




The full details for this problem and other FDTD examples can be found here (as a *Mathematica* notebook), here (as a cdf document) and here (as a pdf).

# Solutions of $\nabla^2 u = \rho / \epsilon_0$

Here we implement a finite difference method for solving the Laplace and Poisson equations.

- **The Laplace Equation**

The Laplacian can be written as

```
laplace[i_, j_] = u[i + 1, j] + u[i - 1, j] + u[i, j + 1] + u[i, j - 1] - 4 u[i, j]
```

$u(i-1,\, j) + u(i,\, j-1) - 4\,u(i,\, j) + u(i,\, j+1) + u(i+1,\, j)$

With the Laplacian written in discrete form Laplace's equation $\nabla^2 U = 0$ becomes a set of coupled algebraic equations. These are generated by the code below.

```
Clear[equations, boundaryConditions, mesh, variables]
equations[mesh_] := Flatten[Table[laplace[i, j] == 0, {i, 1, mesh}, {j, 1, mesh}]]
boundaryConditions[mesh_] :=
 {u[0, k_] → -100., u[mesh + 1, k_] → 100., u[k_, 0] → 0, u[k_, mesh + 1] → 0}
variables[mesh_] := Flatten[Table[u[i, j], {i, 1, mesh}, {j, 1, mesh}]]
```

## ■ The Poisson Equation

The generalization to the Poisson equation is straightforward.

```
poisson[i_, j_] = laplace[i, j] - ρ[i, j]
```

$$-\rho(i,\, j) + u(i-1,\, j) + u(i,\, j-1) - 4\, u(i,\, j) + u(i,\, j+1) + u(i+1,\, j)$$

```
Clear[PoissonEquations, PoissonBoundaryConditions, variables]
PoissonEquations[mesh_] := Flatten[Table[poisson[i, j] == 0, {i, 1, mesh}, {j, 1, mesh}]]
PoissonBoundaryConditions[mesh_] :=
 {u[0, k_] → 0., u[mesh + 1, k_] → 0., u[k_, 0] → 0, u[k_, mesh + 1] → 0}
variables[mesh_] := Flatten[Table[u[i, j], {i, 1, mesh}, {j, 1, mesh}]]
```

One of the lab our first year physics majors do it to paint shapes on conducting carbon paper with silver paint. One of the shapes we have them paint is shown below. They then apply voltage to the leads and measure the equal potential lines.

```
Clear[ρ]
Table[ρ[i, 50] = 1, {i, 45, 75}];
Table[ρ[i, 51] = 1, {i, 45, 75}];
Table[ρ[60, i] = 1, {i, 40, 50}];
Table[ρ[60, i] = -1, {i, 75, 80}];
Table[If[(i - 60)^2 + (j - 70)^2 < 25, ρ[i, j] = -1], {i, 0, mesh}, {j, 0, mesh}];
ρ[i_, j_] = 0;

ρdata = Table[ρ[i, j], {j, 1, mesh}, {i, 1, mesh}];

device =
 Show[Graphics[{RGBColor[0.85, 0.77, 0.76], Map[Point, Position[ρdata, _?(# ≠ 0 &)]]}]]
```

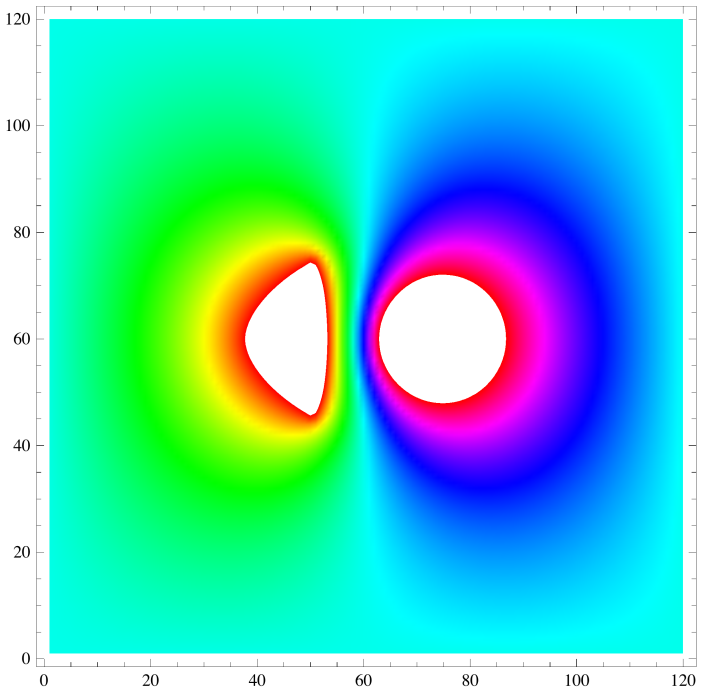


It is straightforward to simulate the result.

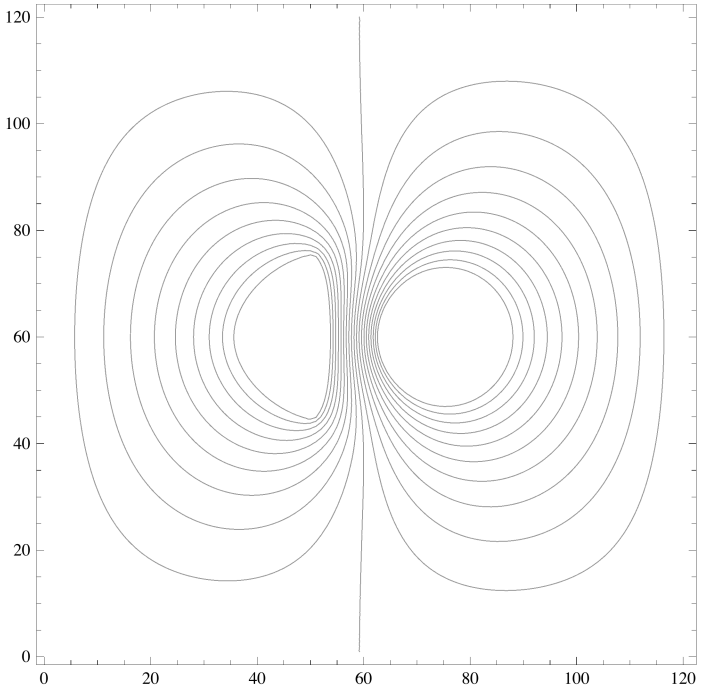
```
A = TimeUsed[];
result = First[Solve[First[
      (PoissonEquations[mesh] /. {PoissonBoundaryConditions[mesh]})], variables[mesh]]];
data = Partition[variables[mesh] /. result, mesh];
TimeUsed[] - A
```

and to plot the result.

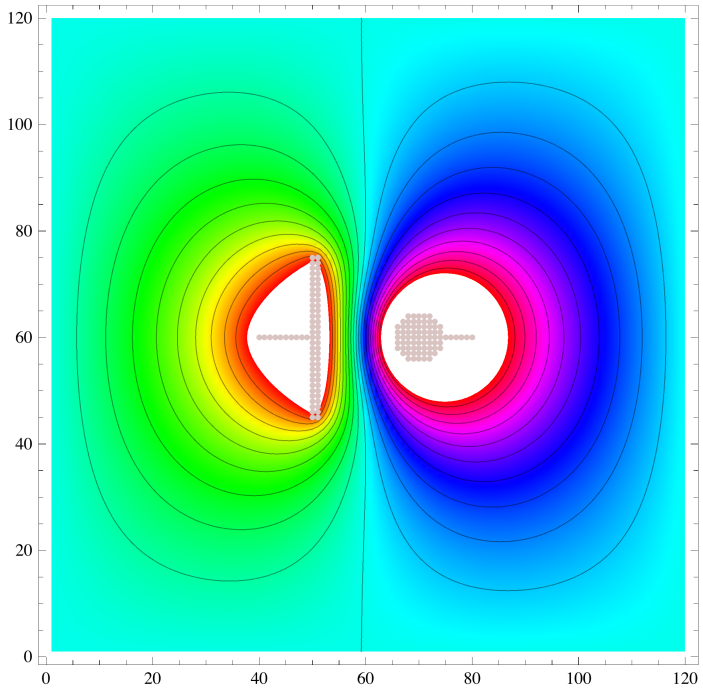
```
plot1 =
 ListDensityPlot[data, ColorFunction -> Hue, Mesh → False, ColorFunctionScaling → True]
```




```
plot2 = ListContourPlot[data, Contours → 20, ContourShading → False]
```



```
Show[plot1, plot2, device]
```




# References

[1] Abraham Goldberg, Harry M.Schey and Judo Schwartz, Am.J.Phys .35, 177 (1967).
[2] M. Zhang, P. Zhang. M.S. Chapman and L. You, PRL 97, 070403 (2006).
[3] B. V. Hall, S. Whitlock, R. Anderson, P. Hannaford and A. I. Sidorov, PRL 98, 030402 (2007).
[4] Alexander Bruno, The Restricted 3-Body Problem, Walter de Gruyter, Berlin, 1994.
[5] V. Szebehely, Theory of Orbits, Academic Press, New York, 1967.
[6] See for example V.V. Markellos, Predictability, Stability, and Chaos in N-Body Dynamical Systems, NATO ASI Series B 272, page 413, Ed. A. Roy, Plenum Press, New York, 1990.
[7] F.R. Moulton, *Periodic Orbits*, Carnegie Institution of Washington, Washington D.C., 1920.
[8] E. Strömgren, Forms of periodic motion in the restricted problem and in the general problem of three bodies, according to researches executed at the Observatory of Copenhagen, Copenhagen Obs. Publ. 39, 1922.
[9] V. Szebehely and T. Van Flandern, *A Family of Retrograde Orbits around the Triangular Equalibrium Points*, Astron. J. 72, 373 (1967).
[10] T.S. Parker and L.O. Chua, Practical Numerical Algorithms for Chaotic Systems, Springer-Verlag, New York, 1989.
[11] T.S. Parker and L.O. Chua, Practical Numerical Algorithms for Chaotic Systems, page 89, Springer-Verlag, New York, 1989.
[12] R. Bulirsch and J. Stoer, *Numerical Treatment of Ordinary Differential Equations by Extrapolation Methods*, Num. Math. 8,1 (1966).
[13] P. Deuflhard, *Recent Progress in Extrapolation Methods for Ordinary Differential Equations*, Num. Math. 27, 505 (1985).
[14] J. Ferrándiz and S. Novo, *Improved Bettis Methods for Long-Term Predication,* in Predictability, Stability, and Chaos in N-Body Dynamical Systems, NATO ASI Series B 272, page 515, Ed. A. Roy, Plenum Press, New York, 1990.