11.2.5 Phase Space Diagrams

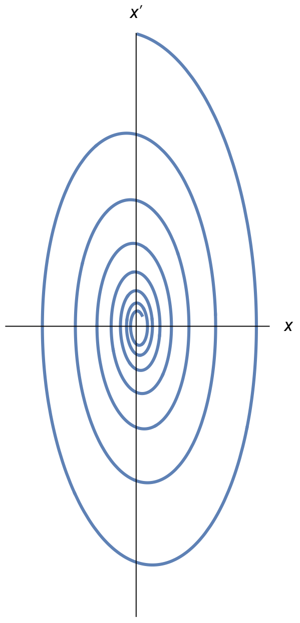
The phase space of a dynamical system, whose degree of freedom is f, is a 2f-dimensional space. It is made up of the f pairs of dependent variables describing the state of the system. To each degree there correponds a pair of variables, the position coordinate and the corresponding (generalized) velocity or the corresponding (canonical) momentum. The solutions obtained for x(t), x'(t), y(t), y'(t), ...give a curve in this 2f-dimensional space, the phase curve. A point on this curve, the phase point, describes the state of the system corresponding to the time t. In this and the next section it is shown how one may draw phase space diagrams from numeric solutions of the equations of motion. In this section such diagrams for f = 1 are treated. If f > 1 then the dimension of phase space is too large to permit one to draw the full space. One is restricted toprojections. An important representation of this type is the Poincaré Map treated in the next section.

11.2.5.1 Damped Harmonic Oscillator

```
deq = x''[t] + .3 x'[t] + 5 x[t] ; tmin = 0; tmax = 20;
dosc = NDSolve[ {deq == 0, x[0] == 0, x'[0] == 1}, x, {t, tmin, tmax}]
```

```
{{x → InterpolatingFunction[ Domain: {{0., 20.}}
                             Output: scalar ]}}
```

```
ParametricPlot[Evaluate[{x[t], x′[t]} /.dosc], {t, tmin, tmax}, ImageSize → 150,
 PlotRange → {{-0.45`, 0.45`}, {-1, 1}}, AxesLabel → {x, x′}, Ticks → None]
```

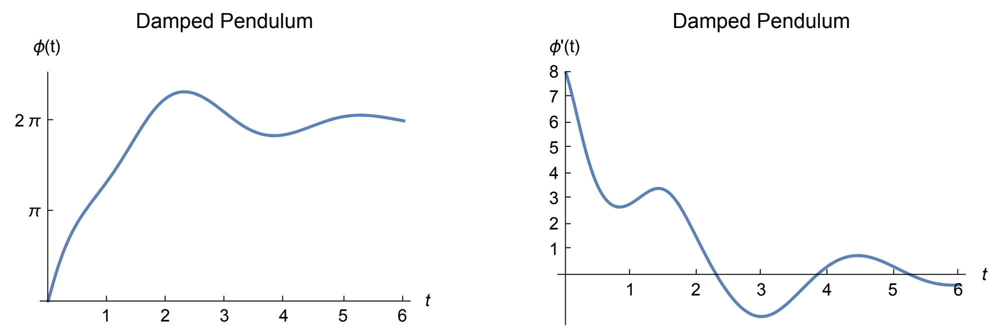


11.2.5.2 Damped Mathematical Pendulum

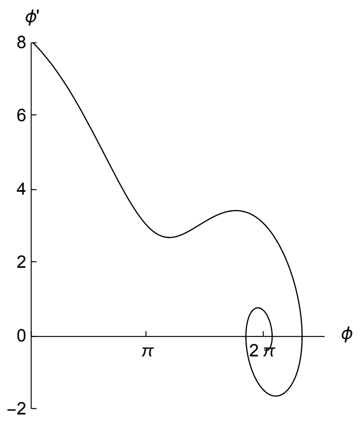
```
deq = p''[t] + p'[t] + 5 Sin[p[t]] ; tmin = 0; tmax = 6;
dap = NDSolve[ {deq == 0, p[0] == 0, p'[0] == 8}, p, {t, tmin, tmax}];
```

```
dapp = Plot[Evaluate[p[t] /.dap],
   {t, tmin, tmax}, AxesLabel → {t, "ϕ(t)"}, PlotRange → {0, 8},
   Ticks → {{1, 2, 3, 4, 5, 6}, {0, π, 2 π}}, PlotLabel → "Damped Pendulum"];
dappp = Plot[Evaluate[p′[t] /.dap], {t, tmin, tmax},
   AxesLabel → {t, "ϕ'(t)"}, PlotRange → {-2, 8},
   Ticks → {{1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6, 7, 8}}, PlotLabel → "Damped Pendulum"];
```

```
Show[GraphicsRow[{dapp, dappp}], ImageSize → 550]
```



```
plc = ParametricPlot[Evaluate[{p[t], p′[t]} /.dap], {t, tmin, tmax},
   AxesLabel → {"ϕ", "ϕ'"}, Ticks → {{0, π, 2 π}, {-2, 0, 2, 4, 6, 8}},
   PlotStyle → {GrayLevel[0], Thickness[0.004]},
   PlotRange → {{0, 8}, {-2, 8}}, ImageSize → 180 ]
```



The first two figures show the angle and the angular velocity as functions of time. The full phase curve is shown just before.

But the phase space diagram does not contain information on time. This can be done by adding points on the phase curve corresponding to equal time steps. The method to achieve this is shown at first for a small number of intervalls.

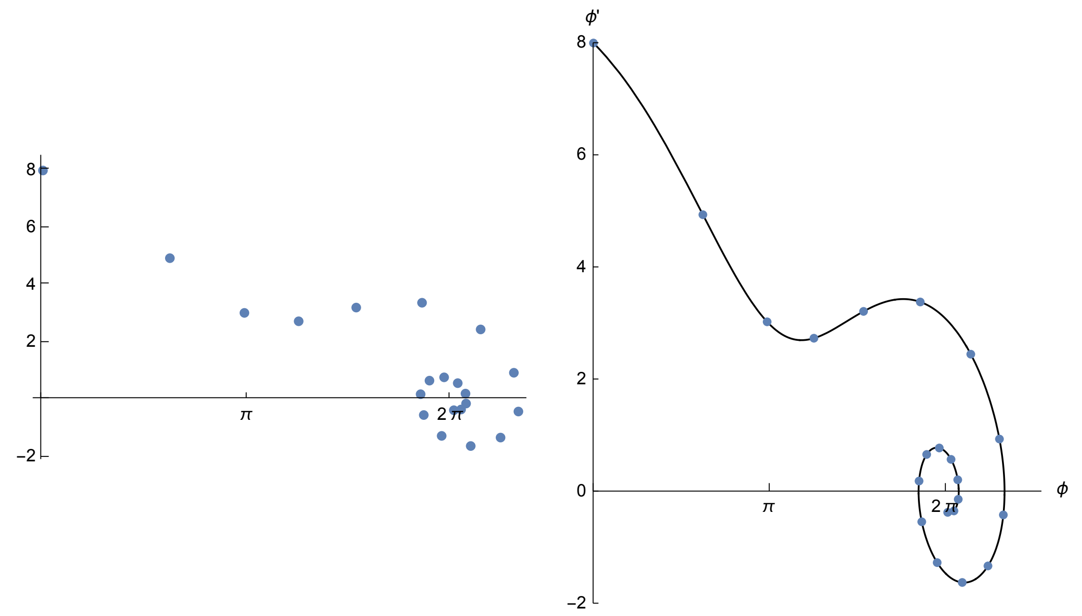
```
np = 5; tk := k (tmax - tmin) / np;
Print[{"k", " time", "       ϕ   ", "    ϕ'       "}] Do[Print[
   Column[{{k, N[tk + 0.001`], Evaluate[{p[tk], p′[tk]} /.dap]}}, Center]], {k, 0, np}]
```

```
{k,  time,       ϕ   ,    ϕ'       }

{0, 0.001, {{0., 8.}}}

{1, 1.201, {{4.81754, 3.21412}}}

{2, 2.401, {{7.31137, -0.415265}}}

{3, 3.601, {{5.85627, -0.53808}}}

{4, 4.801, {{6.37852, 0.575126}}}

{5, 6.001, {{6.31851, -0.370106}}}
```

Null^2

```
np  =  20; tk  :=  k  (tmax  -  tmin)/np;
lp  =  Table[Flatten[Evaluate[{p[tk],p'[tk]}  /.  dap]],  {k,0,np}];

plp = ListPlot[lp, Ticks → {{0, π, 2 π}, {-2, 0, 2, 4, 6, 8}}];
```

```
plt = Show[plc, plp, Prolog → PointSize[0.01`],
   PlotRegion → {{0.01`, 0.99`}, {0.01`, 0.99`}}];
Show[GraphicsRow[{plp, plt}], ImageSize → 550]
```



11.2.6 Poincaré Maps

The phase space of a system of two degrees of freedom is 4-dimensional; so there is no means to represent it in full. The situation is even worse if f > 2. One must recur to projections of the phase space. A particularly important representation is the Poincaré Map. The phase curve (cf. preceeding section) intersects an appropriately chosen plane (a subspace of the full phase space) frequently. Each intersection is marked by a dot. This pattern of dots is called a Poincaré Map. It displays important information on a dynamical system. If there are isolating integrals of the motion (as e.g. energy or angular momentum) these will restrict the subspace available to the phase curve considerrably. Some of such integrals may not be valid globally (.i.e in the whole phase space) but only locally (in a restricted subdomain of phase space). The action of such local integrals may be rendered visible indirectly in Poincaré Maps. They lead to rather regular shapes of the trajectories in coordinate and in phase space and to regular patterns in the Poincaré Maps; whereas the lack of such integrals leads to complex phase curves and to chaotic distributions of the points in the map.

This is shown with a specific example developed by the French astronomer Hénon describing the motion of a star of unit mass in the field of the other stars making up a galaxis. The equations of motion are 2-dimensional; the force on the star is nonlinear

$$\mathbf{F} = (-x - 2xy,\ -y + y^2 - x^2).$$

There is a potential V(x,y); so energy is conserved.

$$V\{x,y\} = (x^2 + y^2)/2 + x^2 y - y^3/3\,; \qquad E = (\dot{x}^2 + \dot{y}^2)/2 + V(x,y)\,.$$

For small values of total energy there exists an additional local integral of the motion. At first the equations of motion are solved by **`NDSolve`**. It is advantageous to represent the equations of motions as a system of first order differential equations. For, on some machines the solution process runs faster than that for the system of second order equations. In any case the velocities are needed for the map. Application of **`NDSolve`** gives an (internal) representation of the functions x(t), y(t), vx(t), vy(t). This can be used to calculate the tranjectory in coordinate space or phase space. For
the Poincaré map the instants must be found for which the phase curve intersects the plane (say x = 0) selected as the subspace of the phase space for the particular map. Each of these times inserted into the solution gives a dot in the y, vy-plane.

Experience (s. end of 11.2.4.) shows that Mathemtica is not very well suited for preparing Poincaré maps for non-linear systems displaying chaotic behavior. It requires too much time and space on central memory and disks for calculations giving reliable results. So the examples given below should be regarded as crude illustrations.

The trajectorys in x,y-space, x(t) and y(t), finally the Poincaré map, will be shown for two cases:

1. For a small energy eno = .01, where motion is still well ordered;

2. for an energy **enc = .161**, which is slightly below the limit **enl = 1/6 = .166666...** for bounded motion, **where motion is chaotic.**

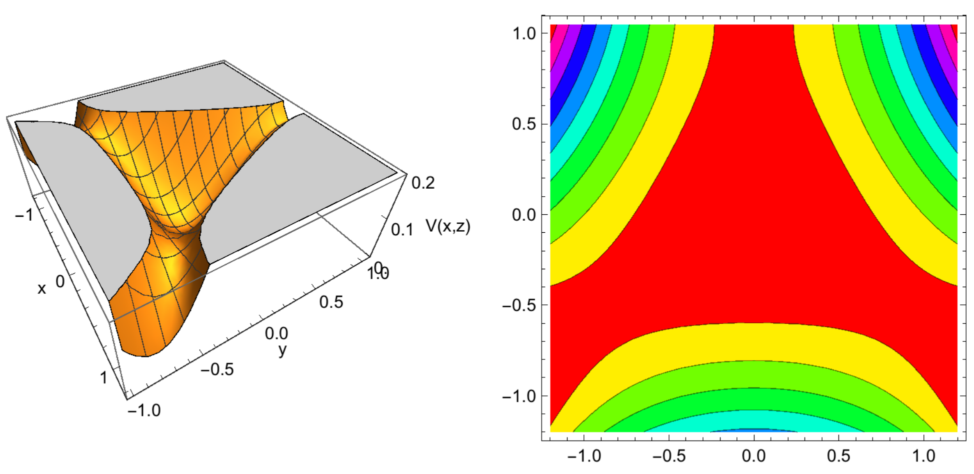
The potential of this system is shown in perspective and by a contour drawing below. V = 1/6 corresponds to the three saddle points.

Plotting the potential

```
v[x_, y_] = (x^2 + y^2) / 2 + x^2 y - y^3 / 3;
```

```
plv = Plot3D[v[x, y], {x, -1.2, 1.2`}, {y, -1`, 1.05`}, PlotPoints → 30,
   PlotRange → {0, 0.2`}, AxesLabel → {"x", "y", "    V(x,z)"},
   ViewPoint → {1, -0.6`, 1}, Ticks → {Automatic, Automatic, {0.`, 0.1`, 0.2`}}];
plc = ContourPlot[v[x, y], {x, -1.2, 1.2`}, {y, -1.2`, 1.05`}, ColorFunction → Hue];
```

```
Show[GraphicsRow[{plv, plc}], ImageSize → 500]
```



Equations of motion in Hamiltonian form

```
r[t_] = {x[t], y[t]}; v[t_] = {vx[t], vy[t]};
force =
    {-x[t] - 2 x[t] y[t], -y[t] + y[t]^2 - x[t]^2};
sys = D[Join[r[t], v[t]], t] == Join[v[t], force] // Thread
```

```
{x′[t] == vx[t], y′[t] == vy[t], vx′[t] == -x[t] - 2 x[t] y[t], vy′[t] == -x[t]^2 - y[t] + y[t]^2}
```

```
en = (vx[t]^2 + vy[t]^2)/2 + (x[t]^2 + y[t]^2)/2 +
      x[t]^2 y[t] - y[t]^3/3;
```

Orderly motion

At first the initial data are chosen such that the mortion is orderly.

```
x0 = 0.;  y0 = .01;  vx0 = .141;  vy0 = 0.;
tmax = 251 Pi//N
```

```
788.54
```

```
eno = en /.
  {x[t] -> x0, y[t] -> y0, vx[t] -> vx0, vy[t] -> vy0}
```

```
0.00999017
```

```
anf = {x[0] == x0, y[0] == y0, vx[0] == vx0, vy[0] == vy0}
```

```
{x[0] == 0., y[0] == 0.01, vx[0] == 0.141, vy[0] == 0.}
```

```
solo = NDSolve[Join[sys, anf], {x, y, vx, vy}, {t, 0, tmax}, MaxSteps → 6500] // Flatten
```

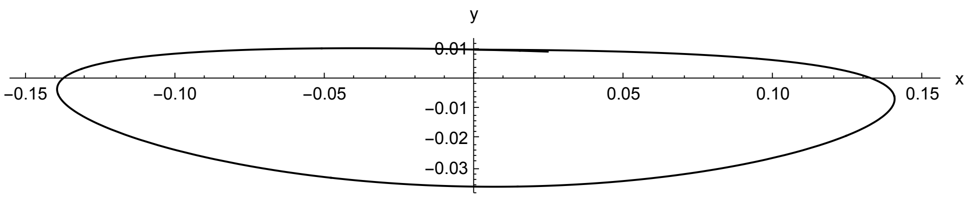
```
{x → InterpolatingFunction[ Domain: {{0., 789.}}
                            Output: scalar ],

 y → InterpolatingFunction[ Domain: {{0., 789.}}
                            Output: scalar ],

 vx → InterpolatingFunction[ Domain: {{0., 789.}}
                             Output: scalar ],

 vy → InterpolatingFunction[ Domain: {{0., 789.}}
                             Output: scalar ]}
```

```
solo2 =
 NDSolve[Join[sys, anf], {x, y, vx, vy}, {t, 0, 2 tmax}, MaxSteps → 15 000] // Flatten
```

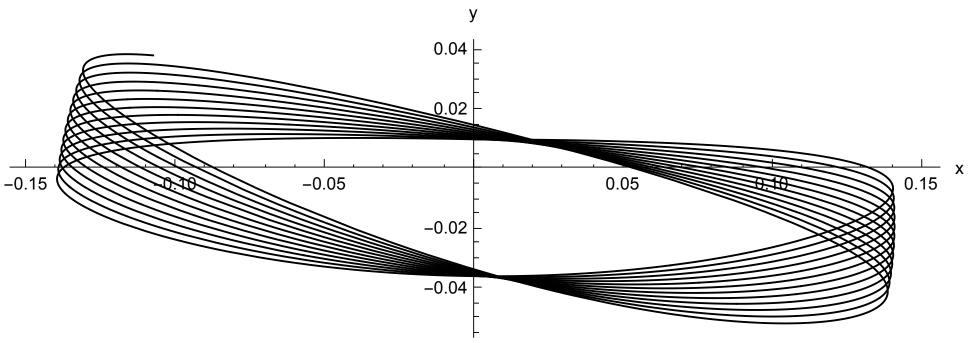
```
{x → InterpolatingFunction[ Domain: {{0., 1580.}}
                            Output: scalar ],

 y → InterpolatingFunction[ Domain: {{0., 1580.}}
                            Output: scalar ],

 vx → InterpolatingFunction[ Domain: {{0., 1580.}}
                             Output: scalar ],

 vy → InterpolatingFunction[ Domain: {{0., 1580.}}
                             Output: scalar ]}
```

The nonlinear terms lead to a nonclosed trajectory which rotates slowly.The ellipse of linear motion is shifted and slightly deformed. The figure below showing the trajectory for a longer time intervall displays the threefold symmetry of the potential well.

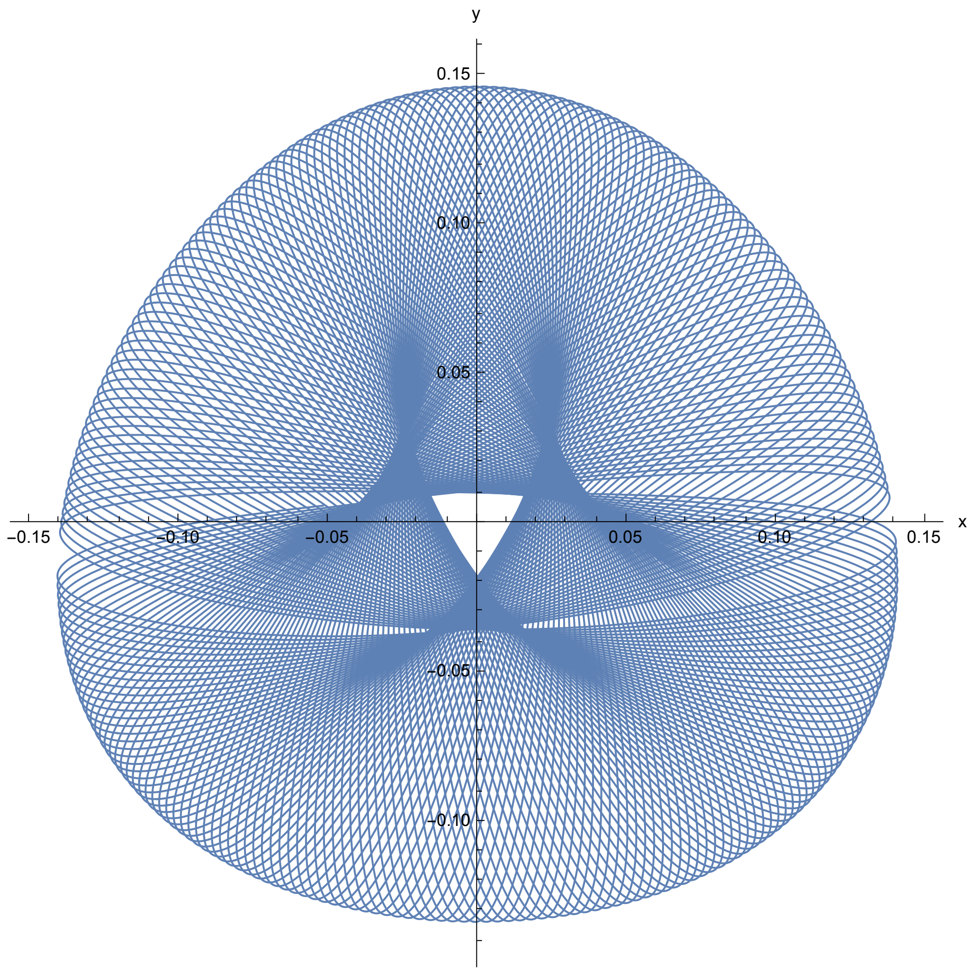
```
ParametricPlot[Evaluate[r[t] /.solo], {t, 0, 6.5`},
 AxesLabel → {"x", "y"}, AspectRatio → Automatic,
 PlotStyle → {AbsoluteThickness[1`], GrayLevel[0]}, ImageSize →  500]
```



```
ParametricPlot[Evaluate[r[t] /.solo], {t, 0, 75.`},
 AxesLabel → {"x", "y"}, AspectRatio → Automatic,
 PlotStyle → {AbsoluteThickness[1`], GrayLevel[0]}, ImageSize →  500]
```

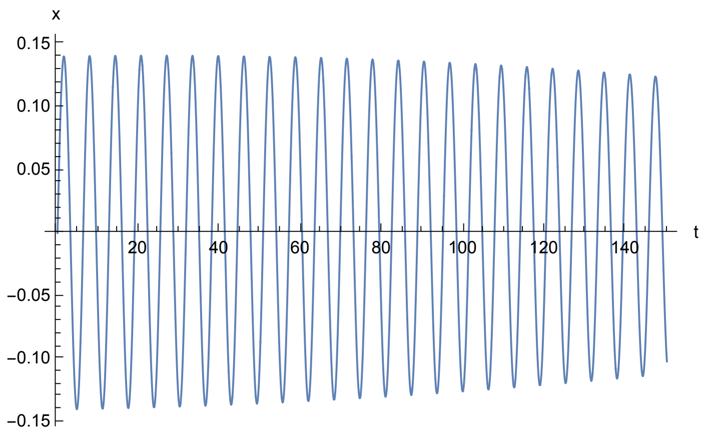


```
ParametricPlot[Evaluate[r[t] /.solo], {t, 0, tmax},
 AxesLabel → {"x", "y"}, AspectRatio → Automatic,
 PlotStyle → AbsoluteThickness[1], PlotPoints → 3000, ImageSize →  500]
```

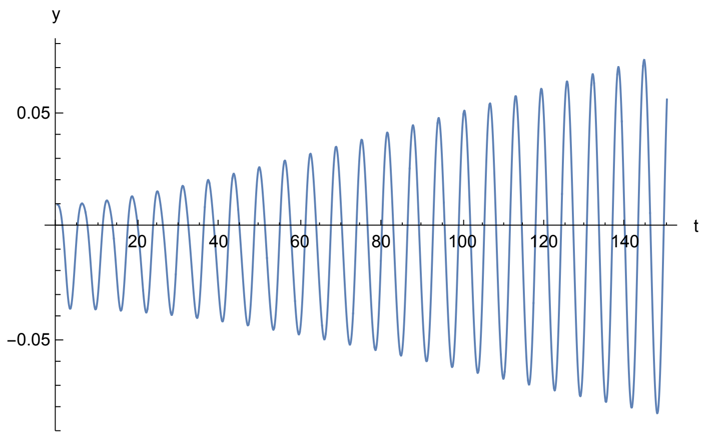


The coordinates as functions of time show also a rather regular behavior. They show oscillations with a regular modulation of the amplitude.

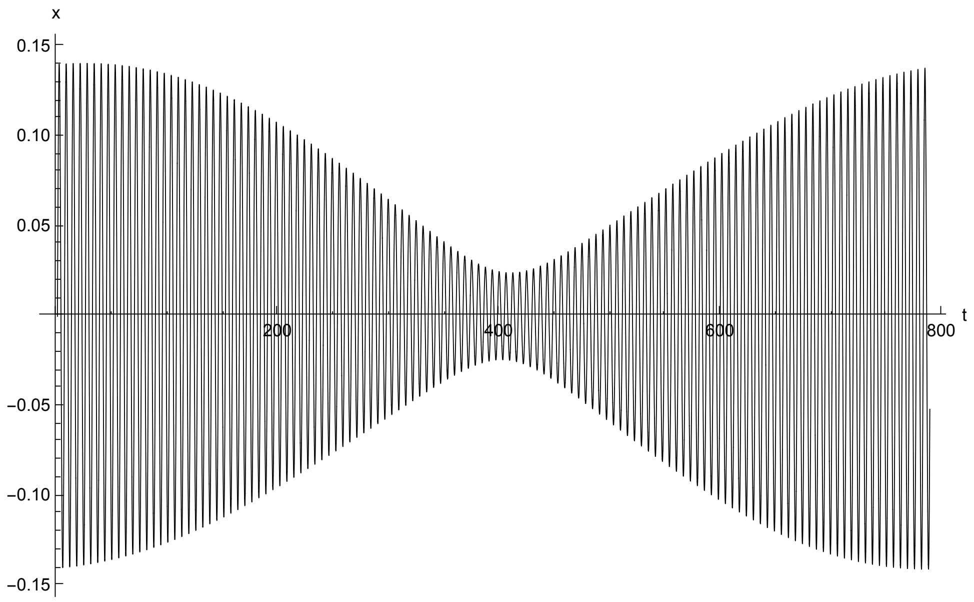
```
Plot[Evaluate[x[t] /.solo], {t, 0, 150.`}, PlotPoints → 2500,
 PlotStyle → AbsoluteThickness[1], AxesLabel → {"t", "x"}]
```



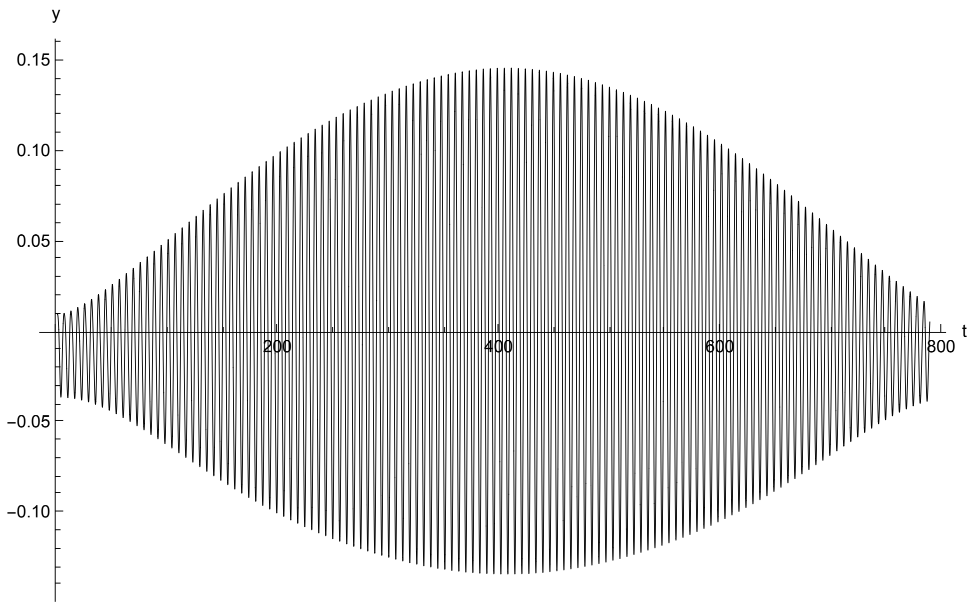
```
Plot[Evaluate[y[t] /.solo], {t, 0, 150.`}, PlotPoints → 2500,
 PlotStyle → AbsoluteThickness[1], AxesLabel → {"t", "y"}]
```



```
Plot[Evaluate[x[t] /.solo], {t, 0, tmax},
 PlotPoints → 3500, PlotStyle → {Thickness[0.001`], GrayLevel[0]},
 ImageSize →  500, AxesLabel → {"t", "x"}]
```



```
Plot[Evaluate[y[t] /.solo], {t, 0, tmax},
 PlotPoints → 3500, PlotStyle → {Thickness[0.001`], GrayLevel[0]},
 ImageSize →  500, AxesLabel → {"t", "y"}]
```



Chaotic motion

Now the trajectory of chaotic motion is computed. In order to get reliable data the various precision options must be increased as discussed in 11.2.4. The trajectory is rather irregular. The same applies to the coordinates as function of time as shown in the last two figures of 11.2.4.

```
x0 = 0.`; y0 = .1`; vx0 = .15`; vy0 = 0.54`;
anf = {x[0] == x0, y[0] == y0, vx[0] == vx0, vy[0] == vy0}
```

```
{x[0] == 0., y[0] == 0.1, vx[0] == 0.15, vy[0] == 0.54}
```

```
enc = en /. {x[t] → x0, y[t] → y0, vx[t] → vx0, vy[t] → vy0}
```

```
0.161717
```

```
solc = NDSolve[Join[sys, anf], {x, y, vx, vy}, {t, 0, tmax}, AccuracyGoal → 15,
    WorkingPrecision → 25, PrecisionGoal → 15, MaxSteps → 26 250] // Flatten
```

NDSolve::precw : The precision of the differential equation ({{x′[t] == vx[t], y′[t] == vy[t], vx′[t] == −x[t] − 2 x[t] y[t], vy′[t] == −x[t]^2 − y[t] + y[t]^2, x[0] == 0., y[0] == 0.1, vx[0] == 0.15, vy[0] == 0.54}, {}, {}, {}, {}}) is less than WorkingPrecision (25.`). ≫

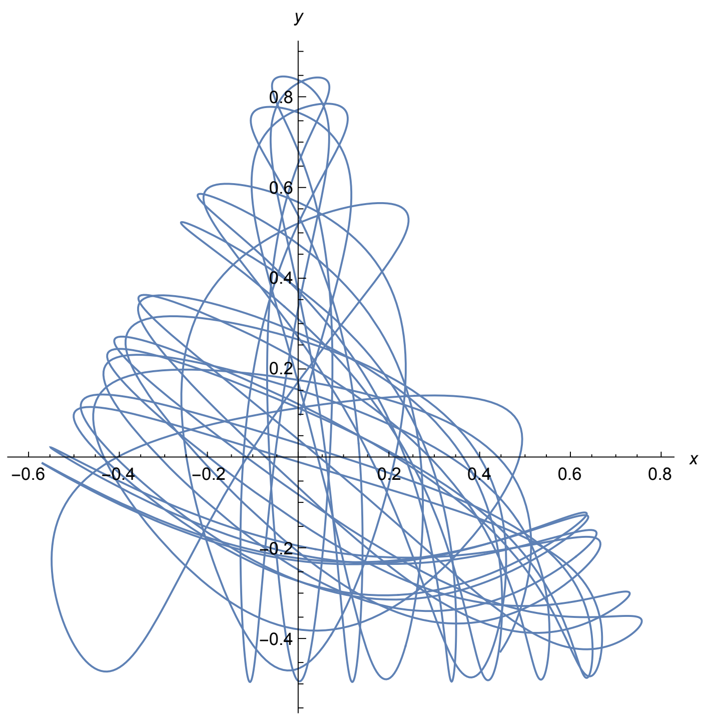
```
{x → InterpolatingFunction[  Domain: {{0, 788.5397560510381254061940}}
                             Output: scalar  ],
 y → InterpolatingFunction[  Domain: {{0, 788.5397560510381254061940}}
                             Output: scalar  ],
 vx → InterpolatingFunction[  Domain: {{0, 788.5397560510381254061940}}
                              Output: scalar  ],
 vy → InterpolatingFunction[  Domain: {{0, 788.5397560510381254061940}}
                              Output: scalar  ]}
```

```
solc2 = NDSolve[Join[sys, anf], {x, y, vx, vy}, {t, 0, 2 tmax}, AccuracyGoal → 15,
    WorkingPrecision → 25, PrecisionGoal → 15, MaxSteps → 50 250] // Flatten
```

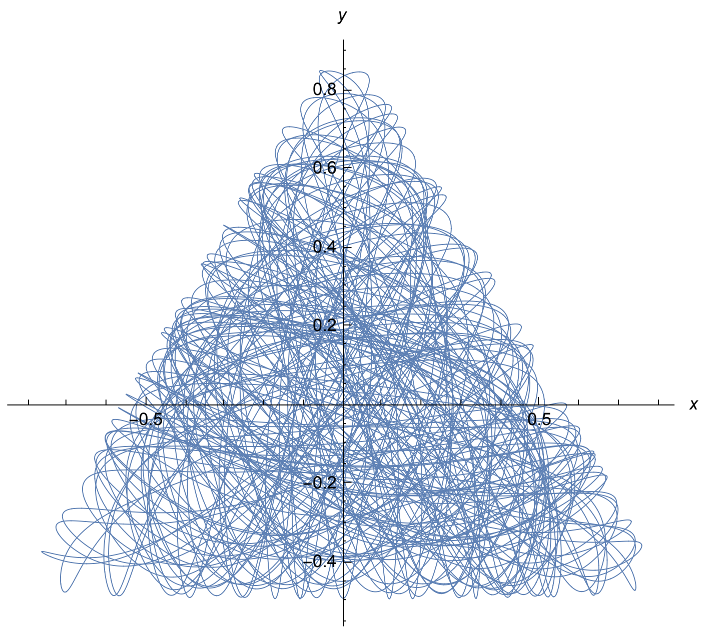
NDSolve::precw : The precision of the differential equation ({{x′[t] == vx[t], y′[t] == vy[t], vx′[t] == −x[t] − 2 x[t] y[t], vy′[t] == −x[t]^2 − y[t] + y[t]^2, x[0] == 0., y[0] == 0.1, vx[0] == 0.15, vy[0] == 0.54}, {}, {}, {}, {}}) is less than WorkingPrecision (25.`). ≫

```
{x → InterpolatingFunction[  Domain: {{0, 1577.079512102076250812388}}
                             Output: scalar  ],
 y → InterpolatingFunction[  Domain: {{0, 1577.079512102076250812388}}
                             Output: scalar  ],
 vx → InterpolatingFunction[  Domain: {{0, 1577.079512102076250812388}}
                              Output: scalar  ],
 vy → InterpolatingFunction[  Domain: {{0, 1577.079512102076250812388}}
                              Output: scalar  ]}
```

```
ParametricPlot[Evaluate[{x[t], y[t]} /.solc], {t, 0, 150.`},
 AxesLabel → {x, y}, PlotStyle → AbsoluteThickness[1], AspectRatio → Automatic]
```



```
ParametricPlot[Evaluate[{x[t], y[t]} /.solc], {t, 0, tmax}, AxesLabel → {x, y},
 PlotStyle → AbsoluteThickness[0.5], PlotPoints → 15 000, AspectRatio → Automatic]
```



For drawing the Poincar› map the times t_i must be found for which $x(t_i) = 0$. This is done with the help of **FindRoot**. In order to get two starting values for this command, at first a list of values $x(t_k)$ is computed. The difference $t_{k+1} - t_k$ must be chosen small enough; it can be read off from the figures showing x(t) given above. This list is scanned for changes of sign; two adjacent values having different signs are used as starting values for **FindRoot**.This gives a list of the t_i .Then a list of values { $y(t_i)$, $vy(t_i)$ } is computed and plotted. At first ths is done for a short running time to show the working of the method.

```
solo2[[1]]
```

```
x → InterpolatingFunction[ Domain: {{0., 1580.}}
                           Output: scalar ]
```

```
fx[t_] = x[t] /. solo2[[1]];
fy[t_] = y[t] /. solo2[[2]];
fpx[t_] = vx[t] /. solo2[[3]];
fpy[t_] = vy[t] /. solo2[[4]];
```

Presenting the method of finding roots for a short running time

```
pas = 1;
tx = Table[ Evaluate[fx[t]], {t,0,10,pas}]//Chop
```

```
{0, 0.118292, 0.127593, 0.0228717, -0.10176, -0.136002,
 -0.045322, 0.087703, 0.139773, 0.0661968, -0.0651117}
```

The changes of sign are found be multiplying adjacent values and evaluating the sign of this product:

erstes und letztes element der liste wird weggeschnitten, anschl. elementweise multiplikation, signum-Fkt, negativer wert bei vorzeichenwechsel

```
Sign[Drop[Chop[tx],1] Drop[Chop[tx],-1]]
```

```
{0, 1, 1, -1, 1, 1, -1, 1, 1, -1}
```

The positions of -1 in the list above give the positions of the value of x preceeding a change in sign. This is used as a starting value for FindRoot[] searching for the value t_i of the zero x[t_i] = 0.

```
li=Flatten[Position[
Sign[Drop[tx,1] Drop[tx,-1]],-1]]
```

```
{4, 7, 10}
```

```
Table[FindRoot[Evaluate[fx[t]] == 0.,{t,li[[k]]}],
{k,Length[li]}]
```

```
{{t → 3.16814}, {t → 6.32746}, {t → 9.50443}}
```

An approximate value t0 of the zero is given by the point where the straight line joining the neighbouring points (t_i, x_i) and (t_{i+1}, x_{i+1}) crosses the t-axis. The corresponding formula is shown with t_i= t1, x_i= x1 and t_{i+1}= t2, x_{i+1}= x2 :

$$t0 = t1 - x1 \frac{t2 - t1}{x2 - x1} = \frac{t1\, x2 - x1\, t2}{x2 - x1}$$

```
tax =
(pas*(li-1)*fx[pas*li]-pas*li*fx[pas*(li-1)])/
(fx[pas*li]-fx[pas*(li-1)]) ;
```

```
fx[tax]
```

```
{-0.00209993, 0.00186699, 0.0000405414}
```

Due to the large step width **pas** the approximate values of the zeros $t_i$given in the list **tax** are still somwhat crude and the corresponding values x(t_i) are small, but not very small. But this improves with a smaller step width.

```
pas = .1;
tx = Table[ Evaluate[fx[t]], {t,0,10,pas}]//Chop ;
```

```
li=Flatten[Position[
Sign[Drop[tx,1] Drop[tx,-1]],-1]];
```

```
tax =
(pas*(li-1)*fx[pas*li]-pas*li*fx[pas*(li-1)])/
(fx[pas*li]-fx[pas*(li-1)])
```

```
{3.16813, 6.32747, 9.50443}
```

```
fx[tax]
```

```
{1.66514 × 10^-6, 2.15412 × 10^-6, -8.16642 × 10^-7}
```

```
{fy[tax],fpy[tax]}//Transpose
```

```
{{-0.035794, 0.00170438}, {0.0100406, -0.00309606}, {-0.0357616, 0.00511471}}
```

The list above gives the pairs (y, vy) for the three times given in **tax**.

Another method for finding the zeros t_i has been given by C. Berger:

```
tx = Table[ Evaluate[x[t] /. solo] /. t -> k, {k,0,10}];
ts = Module[{l = {}},  Do[ If[ tx[[k]] tx[[k+1]] <= 0, l =
                        Append[l, (FindRoot[ Evaluate[x[t] /. solo] == 0., {t,k,k-1}])] ],{k,Length[tx]-1} ]; l]
```

Drawing the Poincaré map in the y, py-plane for oderly motion

```
pas = .1;
tx = Table[ Evaluate[fx[t]], {t,0,2tmax,pas}]//Chop ;
```

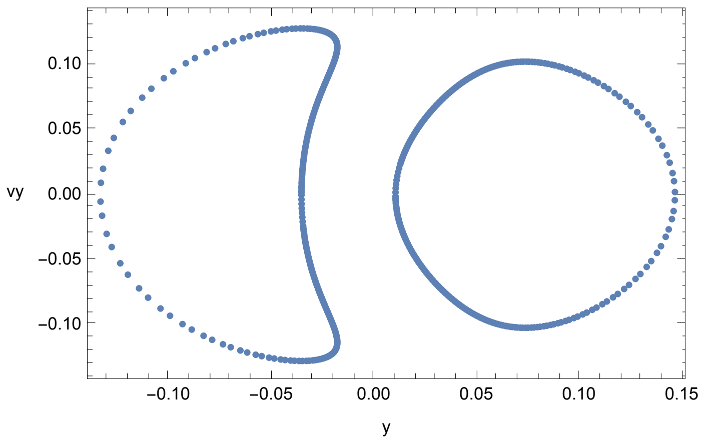
```
li=Flatten[Position[
Sign[Drop[tx,1] Drop[tx,-1]],-1]];
```

```
tax =
(pas*(li-1)*fx[pas*li]-pas*li*fx[pas*(li-1)])/
(fx[pas*li]-fx[pas*(li-1)]) ;
```

```
Max[Abs[fx[tax]]]
```

```
2.27312 × 10^-6
```

```
py = ListPlot[Transpose[{fy[tax], fpy[tax]}], Axes → None,
  RotateLabel → False, Frame → True, FrameLabel → {"y", "vy"}]
```



Drawing the Poincaré map in the y, py-plane for chaotic motion

```
fx[t_] = x[t] /. solc2[[1]];
fy[t_] = y[t] /. solc2[[2]];
fpx[t_] = vx[t] /. solc2[[3]];
fpy[t_] = vy[t] /. solc2[[4]];
```

```
pas = .1;
tx = Table[ Evaluate[fx[t]], {t,0,2tmax,pas}]//Chop ;
```

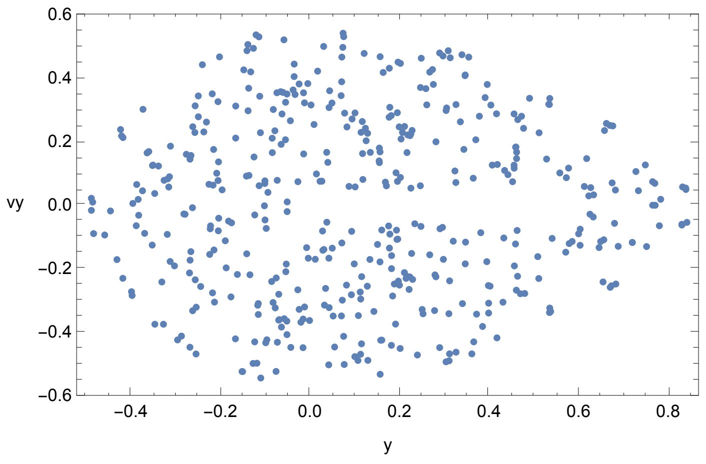
```
li=Flatten[Position[
Sign[Drop[tx,1] Drop[tx,-1]],-1]];
```

```
tax =
(pas*(li-1)*fx[pas*li]-pas*li*fx[pas*(li-1)])/
(fx[pas*li]-fx[pas*(li-1)]) ;
```

```
Max[Abs[fx[tax]]]//ScientificForm
```

1.2588×10^{-5}

```
py = ListPlot[Transpose[{fy[tax], fpy[tax]}], Axes → None,
  RotateLabel → False, Frame → True, FrameLabel → {"y", "vy"}]
```



11.2.7 Finding Eigenvalues by the Shooting Method

The shooting method is a means to compute eigenvalues of a boundary problem defined by a one-dimensional differential equation and boundary conditions fixed at the end of the interval whereinthe equation is defined.
At first a trial value is inserted for the eigenvalue and the differential equationis solved by numeric integration for an initial problem with initial data fulfilling the boundary condition(s) at one end. The integration goes as far as the other end. In most cases inspection of the solution shows that there,the boundary condition(s) is (are) not fulfilled. But the result gives a hint how to improve the trial value for the eigenvalue. This procedure is repeated till a solution fulfilling all boundary conditions is obtained. The trial value then used is an approximation to an eigenvalue of the boundary value problem.

This method is displayed for the eigenvalue problem of the one-dimensional harmonic oscillator in quantum mechanics. The differential equation expressed in dimensionless variables is :

$$\frac{\partial^2 y}{\partial x^2} + \left(2 v + 1 - x^2\right) y = 0$$

v is the eigenvalue parameter; it is related to energy E by E = $\hbar\omega$ (2 v + 1)/2. The boundary cond-tions are:

$$x = \pm\infty: \ y = 0.$$

Analytic solution leads to E/$\hbar\omega$ = 1/2, 3/2, ..., i.e. v = 0, 1, ... ; see § 11.1.1 This is now shown by the shooting method. In place of x = ± ∞ a value of sufficient magnitude is chosen; experience shows that for low v x = 5 suffices; the value of the first derivative must be small, the specific value is uncritical. The end value xe is determined empirically such that the resulting curve fits into the picture.

```
Clear[x, y, sh, nu, xe]
```

11.2.7.1 Showing the shooting method in detail for n = 0

```
sh[nu_,xe_] :=
NDSolve[ {y''[x]  + (2 nu + 1 - x^2) y[x] == 0,
    y[-5] == 4 10^-6, y'[-5] == 5 10^-5}, y[x], {x,-5,xe}]

SetOptions[Plot, PlotStyle -> Thickness[.005]];
dd = PlotStyle -> Dashing[{.01}];
dt = PlotStyle -> Dashing[{.02,.01,.0025,.01}];

so = Table[0, {5}]; pp = so;

so⟦1⟧ = sh[-1.5`, xe = -0.5`];
pp⟦1⟧ = Plot[Evaluate[y[x] /.so⟦1⟧], {x, -5, xe}, Evaluate[dd]];
so⟦2⟧ = sh[-0.5`, xe = 1];
pp⟦2⟧ = Plot[Evaluate[y[x] /.so⟦2⟧], {x, -5, xe}, Evaluate[dd]];
so⟦3⟧ = sh[-0.01`, xe = 6];
pp⟦3⟧ = Plot[Evaluate[y[x] /.so⟦3⟧], {x, -5, xe}, Evaluate[dd]]; so⟦4⟧ = sh[0, xe = 7];
pp⟦4⟧ = Plot[Evaluate[y[x] /.so⟦4⟧], {x, -5, xe}]; so⟦5⟧ = sh[0.01`, xe = 6];
pp⟦5⟧ = Plot[Evaluate[y[x] /.so⟦5⟧], {x, -5, xe}, Evaluate[dt]];
```