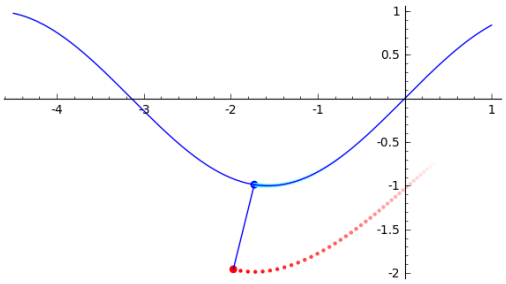
Problems in Physics

(with SageMath)



Marcin Kostur, Jerzy uczka

Institute of Physics
University of Silesia
Poland

August 20, 2019

Download Juptyer Notebook files, pdf and html files of this book from
https://github.com/marcinofulus/Mechanics_with_SageMath

Contents

1 Preface

Solving problems with both CAS as well as powerfull numerical methods is fun!

- link do Mechaniki wstpy
- jak pisa w Markdown

2 Three faces of classical mechanics

- 1687 - edition of Principia Mathematics by Isaac Newton
- 1788 - edition of the Mécanique analytique by Joseph Louis Lagrange (Giuseppe Lodovico Lagrangia)
- 1833 - formulation of mechanics by William Rowand Hamilton

2.1 Harmonic Oscillator

There are three approaches to describe classical mechanical systems. To explain them, we consider one of the simplest example: a one-dimensional harmonic oscillator (a particle of mass m) moving along the x-axis and characterised by the position $x(t)$ at time t. We present it in a trivialized way.

2.1.1 The Newton mechanics

In the Newton description we have to know all forces F which act on the particle of mass m and its dynamics is determined by the Newton second law (the equation of motion):

$$ma = F,$$

where $a = a(t)$ is an acceleration of the particle which is a time-derivative of the particle velocity $v = v(t)$, which in turn in a time-derivative of the particle position (coordinate) $x = x(t)$:

$$a(t) = \dot{v}(t) = \frac{dv(t)}{dt}, \qquad v(t) = \dot{x}(t) = \frac{dx(t)}{dt}, \qquad F = F(x, v, t).$$

Taking into account the above relations the Newton equation $ma = F$ is in fact a second-order differential equation
in the form

$$m\frac{d^2x}{dt^2} = F(x, \dot{x}, t)$$

Therefore two initial conditions have to be imposed. Usually, it is the initial position $x_0 = x(0)$ and the initial velocity $v_0 = v(0)$ of the particle.

In a general case, the force F depends on the particle position x, its velocity $\dot{x} = v$ and explicitly on time t (in the case of an external time-dependent driving as e.g. $A\cos(Bt)$). When $F = F(x)$ then it is conservative system. In such a case we can define a potential energy $U(x)$ of the system by the relations:

$$U(x) = -\int F(x)dx \qquad \text{or} \qquad F(x) = -\frac{dU(x)}{dx}$$

For the harmonic oscillator, the force F is proportional to the partice displacement x, namely,

$$F = F(x) = -kx$$

where k is a positive parameter (e.g. k is a measure of the stiffness of the spring for the mass-spring oscillator). The corresponding potential energy $E_p = U(x)$ is

$$U(x) = \frac{1}{2}kx^2$$

i.e. it is a parabola. The Newton equation for the harmonic oscillator is rewritten in the standadrd form as

$$\ddot{x} + \omega_0^2 x = 0, \qquad \text{where} \qquad \omega_0^2 = \frac{k}{m}$$

Its analysis in presented in the next notebook.

2.1.2 The Langange mechanics

In the Lagrange approach, the mechanical system is described by the scalar function L called the Lagrangian function. In the case of conservative systems it is a difference between the kinetic energy E_k and the potential energy E_p of the system, i.e.,

$$L = L(x, v) = E_k - E_p = \frac{mv^2}{2} - U(x)$$

For the harmonic oscillator it reads

$$L = \frac{mv^2}{2} - \frac{1}{2}kx^2$$

Dynamics of the system is determined by the Euler-Lagrange equation:

$$\frac{d}{dt}\frac{\partial L}{\partial v} - \frac{\partial L}{\partial x} = 0$$

It is a counterpart of the Newton equation. For the harmonic oscillator

$$\frac{\partial L}{\partial v} = mv = p$$

is the (Newton) momentum p of the particle and

$$\frac{\partial L}{\partial x} = -kx = F$$

is the force F acting on the oscillator. As a result, the Euler-Lagrange equation leads to the equation of motion

$$\frac{dp}{dt} = -kx$$

and is the same as the Newton equation because $p = mv = m\dot{x}$.

2.1.3 The Hamilton mechanics

In the Hamilton approach, the mechanical system is described by the scalar function H called the Hamilton function. In the case of conservative systems, it is a total energy of the system, i.e.,

$$H = H(x, p) = E_k + E_p = \frac{p^2}{2m} + U(x)$$

where the kinetic energy E_k is expressed by the CANONICAL MOMENTUM p !

For the harmonic oscillator it reads

$$H = \frac{p^2}{2m} + \frac{1}{2}kx^2$$

Dynamics of the system is determined by the Hamilton equations:

$$\frac{dx}{dt} = \frac{\partial H}{\partial p}, \qquad \frac{dp}{dt} = -\frac{\partial H}{\partial x}$$

It is a counterpart of the Newton equation or the Euler-Lagrange equation. For the harmonic oscillator we obtain

$$\frac{\partial H}{\partial p} = \frac{p}{m}$$

is the velocity v of the particle and

$$\frac{\partial H}{\partial x} = kx = -F$$

is proportional to the force F acting on the oscillator. As a result, the Hamilton equations lead to the equation of motion in the form

$$\frac{dx}{dt} = \dot{x} = \frac{p}{m}, \qquad \frac{dp}{dt} = \dot{p} = -kx$$

and are equivalent to the Newton equation or the Euler-Lagrange equation. Indeed, if we differentiate the first equation we get $\ddot{x} = \dot{p}/m$. Next we insert to it the second equation for $\dot{p}$ and finally we obtain $\ddot{x} = -kx/m = -\omega_0^2 x$.

REMARK: We want to stress that in a general case, the construction of the Lagrange function or the Hamilton function can be a more complicated task.

Exercise Solve the equation of motion for the harmonic oscillator with the mass $m = 2$ and the eigen-frequency $\omega_0 = 1$. Find $x(t)$ and $v(t)$ for a given initial conditions: $x(0) = 1$ and $v(0) = 0.5$. Next, depict the time-evolution of the kinetic energy $E_k(t)$, potential energy $E_p(t)$ and the total energy $E(t) = E_k(t) + E_p(t)$:

$$E_k(t) = \frac{1}{2}mv^2(t), \qquad E_p(t) = \frac{1}{2}m\omega_0^2 x^2(t)$$

What is the main conclusion regarding the total $E(t)$.

Solution In SageMath we can write easile solve any system of ODE numerically. First we write Newton equation in a following form:

$$\begin{cases} \dfrac{dx}{dt} = v \\ \dfrac{dv}{dt} = -\omega_0^2 x \end{cases} \tag{1}$$

Then we use `desolve_odeint` to obtain a numerical solution.

[1]:
```
m = 2
omega0 = 1
var('x,v')
times = srange(0,4,0.01)
xv = desolve_odeint([ v, -omega0^2*x ], [1,0.5] , times, [x,v])
```

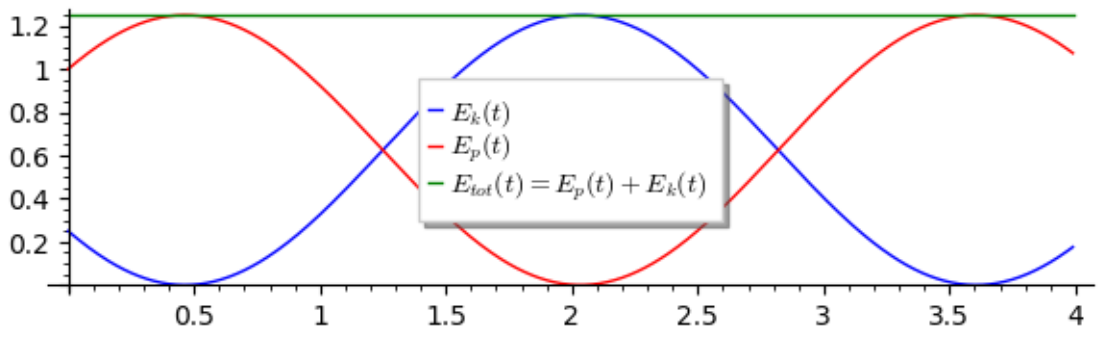
We can compute E_k and E_p:

[2]:
```
Ek = 1/2*m*xv[:,1]^2
Ep = 1/2*m*omega0^2*xv[:,0]^2
```

And plot the results:

[3]:
```
p_Ek = line( zip( times, Ek),\
            legend_label=r'$E_k(t)$', figsize=(6,2))
p_Ep = line( zip( times,Ep ),\
            color='red',legend_label=r'$E_p(t)$')
p_Etot = line( zip(times,Ek + Ep),\
             color='green',legend_label=r'$E_{tot}(t) = E_p(t)+E_k(t)$')
p_Ek + p_Ep + p_Etot
```

[3]:



2.2 Damped harmonic oscillator

2.2.1 The Newton mechanics

A system which interacts with its environment is dissipative (it losses its energy) due to friction. It can be conveniently described by the Newton mechanics. For small velocity of the particle, the friction force is proportional to its velocity (the Stokes force) $F = -\gamma v = -\gamma \dot{x}$, where γ is a friction or damping coefficient. The Newton equation for the damped harmonic oscillator has the form

$$m\ddot{x} = -\gamma \dot{x} - kx.$$

Its analysis in presented in the next notebook.

2.2.2 The Langange mechanics

In the Lagrange approach, now the Lagrange function is not a difference between the kinetic energy and the potential energy but is constructed in such an artificial way in order to obtain the correct equation of motion presented above. Let us propose the following function:

$$L = L(x, v, t) = e^{\gamma t/m} \left[\frac{mv^2}{2} - \frac{1}{2} kx^2 \right]$$

Then in the Euler-Lagrange equation:

$$\frac{\partial L}{\partial v} = mve^{\gamma t/m}$$

Its time derivative is

$$\frac{d}{dt} \frac{\partial L}{\partial v} = m\dot{v}e^{\gamma t/m} + \gamma ve^{\gamma t/m}$$

The second part of the Euler-Lagrange equation is

$$\frac{\partial L}{\partial x} = -kxe^{\gamma t/m}$$

As a result, the final form of the Euler-Lagrange equation is

$$m\ddot{x} + \gamma \dot{x} + kx = 0$$

and is the same as in the Newton approach.

2.2.3 The Hamilton mechanics

In the Hamilton approach, the Hamilton function is in the form

$$H = H(x, p, t) = \frac{p^2}{2m} e^{-\gamma t/m} + \frac{1}{2} kx^2 e^{\gamma t/m}$$

The partial derivatives are

$$\frac{\partial H}{\partial p} = \frac{p}{m} e^{-\gamma t/m}$$

and

$$\frac{\partial H}{\partial x} = kxe^{\gamma t/m}$$

As a result, the Hamilton equations lead to the equation of motion in the form

$$\frac{dx}{dt} = \frac{p}{m} e^{-\gamma t/m}, \qquad \frac{dp}{dt} = -kxe^{\gamma t/m}$$

One can show that they are equivalent to the Newton equation or the Euler-Lagrange equation. However, there is one important remark: From the first Hamilton equation it follows that

$$mv = pe^{-\gamma t/m}$$

The left side is the Newton momentum of the particle. In the right side, p is the canonical momentum. The above equations of motion are investigated in detail in the next notebook.

Exercise Solve the equation of motion for the damped harmonic oscillatora with the mass $m = 2$, the friction coefficient $\gamma = 1$ and the eigen-frequency $\omega_0 = 1$. Find $x(t)$ and $v(t)$ for a given initial conditions: $x(0) = 1$ and $v(0) = 0.5$. Next depict the time-evolution of the kinetic energy $E_k(t)$, potential energy $E_p(t)$ and the total energy $E(t) = E_k(t) + E_p(t)$:

$$E_k(t) = \frac{1}{2} m v^2(t), \qquad E_p(t) = \frac{1}{2} m \omega^2 x^2(t)$$

Compare the results with the frictionless case, $\gamma = 0$.

Solution In this case the system of ODEs 2 takes a following form: :

$$\begin{cases} \dfrac{dx}{dt} = v \\ \dfrac{dv}{dt} = -\omega_0^2 x - \gamma v \end{cases} \tag{2}$$

Then we use `desolve_odeint` to obtain a numerical solution.

[4]:
```
m = 2
omega0 = 1
gamma_ = 1
var('x,v')
times = srange(0,4,0.01)
xv = desolve_odeint([ v, -x-gamma_*v ], [1,0.5] , times, [x,v])
```

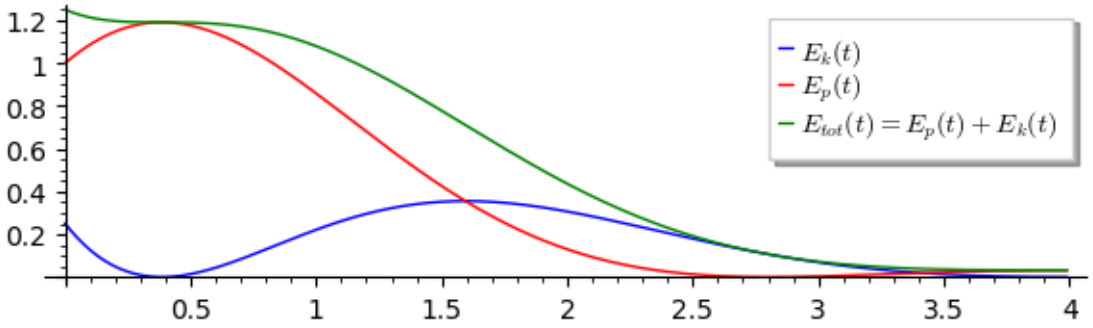
We can compute E_k and E_p:

[5]:
```
Ek = 1/2*m*xv[:,1]^2
Ep = 1/2*m*omega0^2*xv[:,0]^2
```

And plot the results:

[6]:
```
p_Ek = line( zip( times, Ek),\
            legend_label=r'$E_k(t)$', figsize=(6,2))
p_Ep = line( zip( times,Ep ),\
            color='red',legend_label=r'$E_p(t)$')
p_Etot = line( zip(times,Ek + Ep),\
             color='green',legend_label=r'$E_{tot}(t) = E_p(t)+E_k(t)$')
p_Ek + p_Ep + p_Etot
```

[6]:



3 Harmonic oscillator with computer algebra

3.1 Harmonic oscillator

This material includes the derivation of harmonic oscillator solutions using Sage in the following classic cases:

- free oscillator
- free damped oscillator
- forced oscillator with damping: resonance phenomenon

3.1.1 What is a harmonic oscillator?

The harmonic oscillator is a point particle with mass m in the force field, which depends linearly on the position. The restoring force to the equilibrium position depends linearly on the amount of the deflection. An example of such a force may be the reaction of a flexible body applying Hook's law. In other words, we have a material point with mass m in a square potential.

Consider this last definition, let's have potential:

$$U(x) = \frac{1}{2}kx^2$$

then the Newton equation for a material point will be:

$$ma = -\frac{\partial U(x)}{\partial x} = -kx,$$

and because acceleration a is the second derivative of the position after the time we have finally:

$$m\ddot{x} = -kx.$$

This is the equation of motion for the harmonic oscillator. Another method of obtaining it is to substitute the force derived from the linear span - that is the Hook's law $\vec{F} = -k\vec{x}$, to equate $m\vec{a} = \vec{F}$.

3.1.2 Approximation of small vibrations

The harmonic oscillator is an extremely important model that appears in many real situations. Note that if we have any potential, which in the $x = 0$ point has a minimum, then we can write its development of Taylor:

$$U(x) = U(0) + U'(0)x + \frac{1}{2}U''(0)x^2 + ...$$

Since we have assumed that in $x = 0$ we have a minimum, the first derivative disappears $U'(0) = 0$ and we get:

$$U(x) = U(0) + \frac{1}{2}U''(0)x^2 + ...$$

Taylor's series can be broken while keeping the word with the lowest non-flammable power of x or the other. Permanent word can be omitted (as mathematicians say, without limiting the generality of friendships, that it is equal to zero). We will then receive an approximation that is right for small x:

$$U(x) = \frac{1}{2}U''(0)x^2.$$

The obtained potential is identical to the square potential, if only the second derivative of the potential at the point of its minimum U ” (0) is identified with the elastic constant k. So we see that the movement in every minimum “almost” of any potential, for small deviations, can be approximated by a harmonic oscillator. This fact is the reason why the harmonic oscillator is such a frequently used model in physics.

3.2 Free oscillator

Consider the equation of motion for a harmonic oscillator that does not contain friction or any external force. It has a form:

$$m\ddot{x} = -kx.$$

It is conveniet to use dimensionless variables, in which we have:

$$\ddot{x} = -\omega_0^2 x,$$

where $\omega_0 = \sqrt{\frac{k}{m}}$ is a positive number.

It is a linear second order ordinary differential equation with constant coefficients. Equations from this class can be easily solved - assuming the form of a solution and substituting it for the equation. With the computer algebra system included in SageMath, we can simplify this procedure using the desolve function:

[1]:
```
load('cas_utils.sage')
```

[2]:
```
var('omega0 x0')
assume(omega0>0)

var('t')
X = function('X')(t)

osc = diff(X,t,2) == -omega0^2*X
showmath( osc )
```

[2]:

$$\frac{\partial^2}{(\partial t)^2} X(t) = -\omega_0^2 X(t)$$

Because we use a set of variables for operations on expressions containing derivatives, the above cell will give us a mathematical formula of the differential equation representing the harmonic oscillator.

With this expression, we can use computer algebra to solve the differential equation:

[3]:
```
phi_anal = desolve(osc,dvar=X,ivar=t,show_method=True)
showmath(phi_anal)
```

[3]:

$$[K_2 \cos(\omega_0 t) + K_1 \sin(\omega_0 t), \texttt{constcoeff}]$$

First, we see that we must assume that ω_0 is non-zero. Otherwise, the solution would have a different form.

Secondly, we see that we have been scrubbing the so-called a general solution that depends on two constants. We can determine these two constants knowing the position and speed of the oscillator at some point in time. Because equations of motion do not depend on time, without the limitation of generality, $t = 0$ can be used as an initial moment.

Sage, can also do determine constants, so if in the moment $t = 0$, the oscillator was in the rest state: $x(0) = x_0$ and $v(0) = 0$, then we have:

[4]:
```
phi_anal = desolve(osc,dvar=X,ivar=t,ics=[0,x0,0])
showmath(phi_anal)
```

[4]:

$$x_0 \cos\left(\omega_0 t\right)$$

and if, at $t = 0$, the oscillator was at $x(0) = 0$ and $v(0) = v_0$, then we have:

[5]:
```
var('v0 x0')
phi_anal = desolve(osc,dvar=X,ivar=t,ics=[0,0,v0])
showmath(phi_anal)
```

[5]:

$$\frac{v_0 \sin\left(\omega_0 t\right)}{\omega_0}$$

at the moment of $t = 0$, the oscillator was at $x(0) = x_0$ and $v(0) = v_0$, then we have:

[6]:
```
var('v0 x0')
phi_anal = desolve(osc,dvar=X,ivar=t,ics=[0,x0,v0])
showmath(phi_anal)
```

[6]:

$$x_0 \cos\left(\omega_0 t\right) + \frac{v_0 \sin\left(\omega_0 t\right)}{\omega_0}$$

We can also set specific numerical values:

[7]:
```
phi_anal = desolve(osc,dvar=X,ivar=t,ics=[0,2,3])
showmath(phi_anal)
```

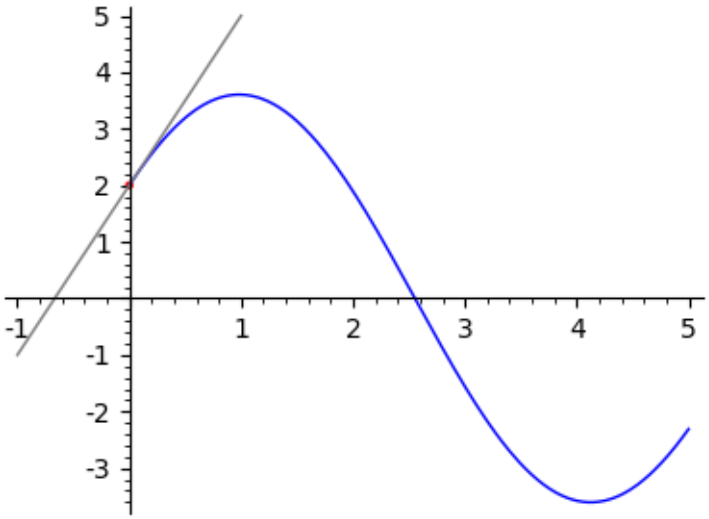
[7]:

$$\frac{3 \sin\left(\omega_0 t\right)}{\omega_0} + 2 \cos\left(\omega_0 t\right)$$

Let's plot of this solution with initial condition $x_0 = 2$ $v_0 = 3$ for $\omega_0 = 1$. We see that the solution is tangent to $3/\omega_0$ line that passes point $(0, 2)$:

[8]:
```
plot(phi_anal.subs(omega0==1),(t,0,5),figsize=4)+\
 plot( 3*t+2,(t,-1,1),color='gray')+\
 point([0,2],color='red')
```

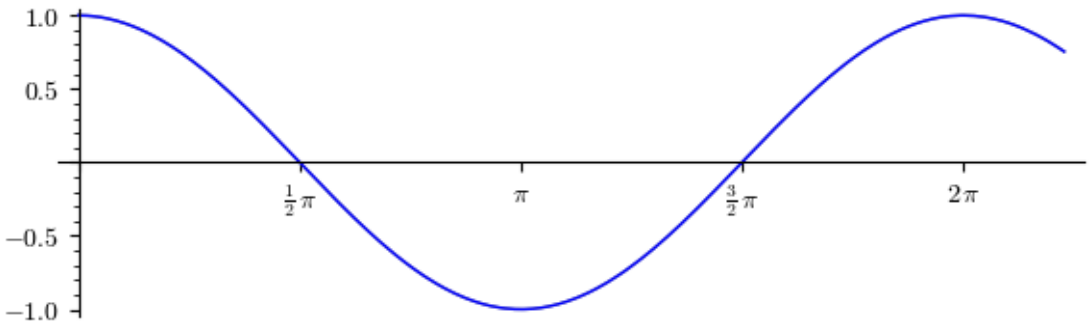
[8]:

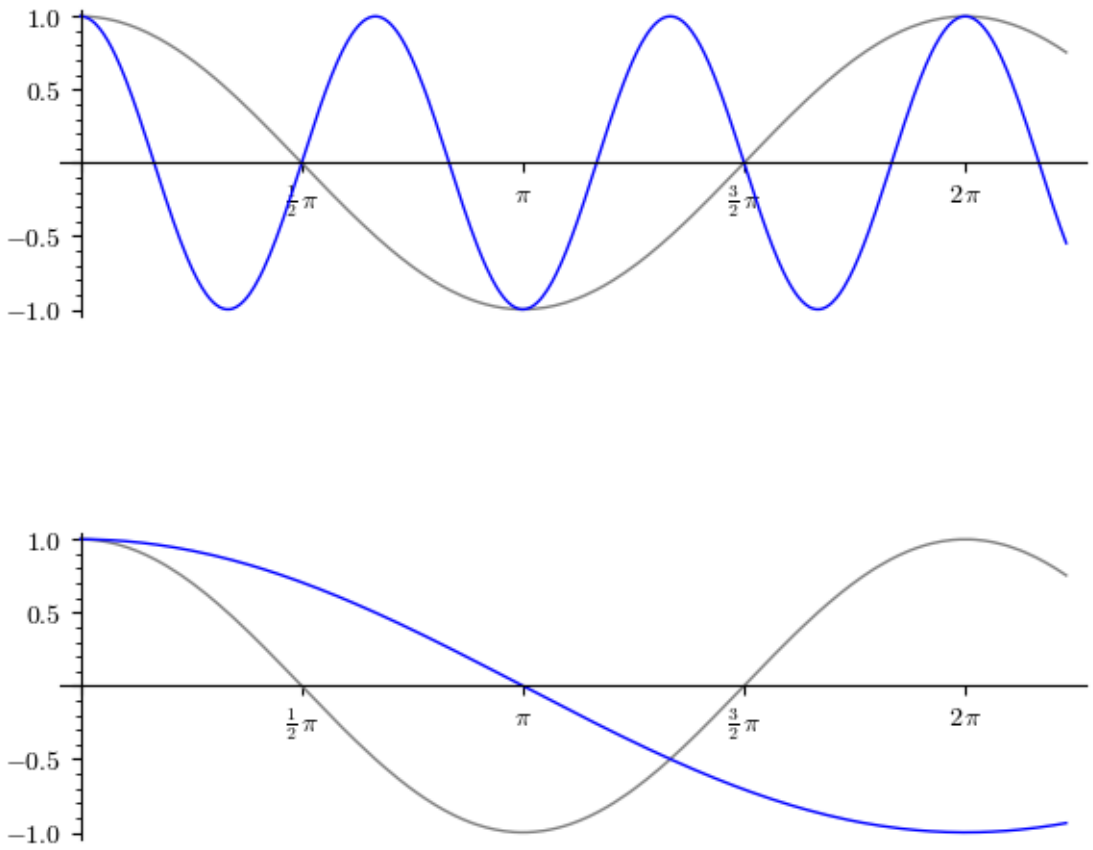


We can examine how the solution depends on the parameters ω_0 and x_0 (with $v_0 = 0$).

[9]:
```
#@interact
def free_oscilator_xt(w0=slider(0.1,3,0.1,default=1),x0=slider(0.1,3,0.
↪1,default=1)):
    phi_anal = desolve(osc,dvar=X,ivar=t,ics=[0,1,0])
    p = plot(phi_anal.subs({omega0:1}),(t,0,7),color='gray')
    phi_anal = desolve(osc,dvar=X,ivar=t,ics=[0,x0,0])
    p += plot(phi_anal.subs({omega0:w0}),(t,0,7))
    p.show( ticks=[[0,pi/2,pi,3/2*pi,2*pi],1/
↪2],tick_formatter="latex",figsize=(6,2))
```

[10]:
```
free_oscilator_xt(w0=1, x0=1)
free_oscilator_xt(w0=3, x0=1)
free_oscilator_xt(w0=1/2, x0=1)
```





3.3 Damped oscillator

As we have seen from previous considerations, the free hamon oscillator performs vibrations infinitively long. In practice, every classic harmonic oscillator is subjected to certain damping forces, which cause the vibration to decay in time. It is so called damped oscillator. A special case of the friction force, which is taken into account when testing the motion of the oscillator - is the friction directly proportional to the speed:

$$\vec{F}_D = -\gamma\vec{v}.$$

Such friction occurs, for example, when the ball is in the liquid with small Reynolds numbers and is called viscous friction. If the Reynolds number is large, then the friction depends more on the speed - in aerodynamics a quadratic relationship is a good model.

Consideration of linearly rate-dependent friction has the basic advantage that, despite adding an additional member to the equation of motion, we still deal with a linear equation with constant coefficients and a full mathematical analysis of solutions can be carried out. So we have the equation:

$$m\ddot{x} + \Gamma\dot{x} + kx = 0.$$

Let's divide this equation by m and enter the new $\gamma = \Gamma/m$ designation, and for k/m, put ω_0^2 as in the previous case:

$$\ddot{x} + \gamma\dot{x} + \omega_0^2 x = 0.$$

From the theory of linear differential equations, we know that we should consider the roots of a characteristic polynomial for a given equation and depending on their nature we have a different solution. It can be easily done assuming that solution to the equation in an exponent e^{kt} and substitute this form into ODE:

[11]:
```
var('k t')
var('omega omega0')
var('g', latex_name='\gamma')

f = exp(k*t)
eq = f.diff(t,2)+g*f.diff(t)+omega0^2*f
showmath(eq)
```

[11]:

$$\gamma k e^{(kt)} + k^2 e^{(kt)} + \omega_0^2 e^{(kt)}$$

Now we can factorize this expression:

[12]:
```
eq.factor().show()
```

```
(g*k + k^2 + omega0^2)*e^(k*t)
```

It is zero for all t only when $\gamma k + k^2 + \omega_0^2 = 0$. This is quadratic equation which has follwing solutions:

[13]:
```
showmath( (eq.factor()*exp(-k*t)).solve(k) )
```

[13]:

$$\left[k = -\frac{1}{2}\gamma - \frac{1}{2}\sqrt{\gamma^2 - 4\,\omega_0^2}, k = -\frac{1}{2}\gamma + \frac{1}{2}\sqrt{\gamma^2 - 4\,\omega_0^2}\right]$$

There are three different values of determinant:

$$\Delta = \sqrt{\gamma^2 - 4\omega_0^2}$$

for which soltuion have qualitatively different properties:

- Two complex roots - damped oscillations. - Two real roots - damped oscillation, no oscillation. - One degenerate root ($\Delta = 0$ discriminant) - critical vibrations, no oscillations, but it can have one maximum.

3.3.1 Two complex roots:

Since both γ and ω_0 are positive and the expression for Δ factorizes:

$$\gamma^2 - 4\omega_0^2 = (\gamma - 2\omega_0)(\gamma + 2\omega_0),$$

the expression under square root will be negative if and only if

$$\gamma - 2\omega_0 < 0 \tag{3}$$

[14]:
```
var('omega omega0')
var('g', latex_name='\gamma')

forget()
assume(g-2*omega0<0)
assume(g>0)
assume(omega0>0)
show( assumptions() )

osc = diff(X,t,2) == -g*diff(X,t)-omega0^2*X

showmath(osc)
```

```
[g - 2*omega0 < 0, g > 0, omega0 > 0]
```

[14]:

$$\frac{\partial^2}{(\partial t)^2} X(t) = -\omega_0^2 X(t) - \gamma \frac{\partial}{\partial t} X(t)$$

[15]:
```
phi_anal = desolve(osc,dvar=X,ivar=t)
showmath(phi_anal)
```

[15]:

$$\left(K_2 \cos\left(\frac{1}{2}\sqrt{-\gamma^2 + 4\,\omega_0^2}\,t\right) + K_1 \sin\left(\frac{1}{2}\sqrt{-\gamma^2 + 4\,\omega_0^2}\,t\right)\right) e^{\left(-\frac{1}{2}\,\gamma t\right)}$$

In Sage above expression contain constants which have no "handles" in global namespace:

[16]:
```
phi_anal.variables()
```

[16]:
```
(_K1, _K2, g, omega0, t)
```

It means that expession like `.subs({_K1:2})` will throw an exception.

We can however define symbolic variables for all constants in solution. Note that those constants start with underscore character, thus we can do it automatically:

[17]:
```
[var(str(s)) for s in phi_anal.variables() if str(s).startswith("_")]
```

[17]:
```
[_K1, _K2]
```

Now we have `_K1` variable available, and we can use it in substitutions:

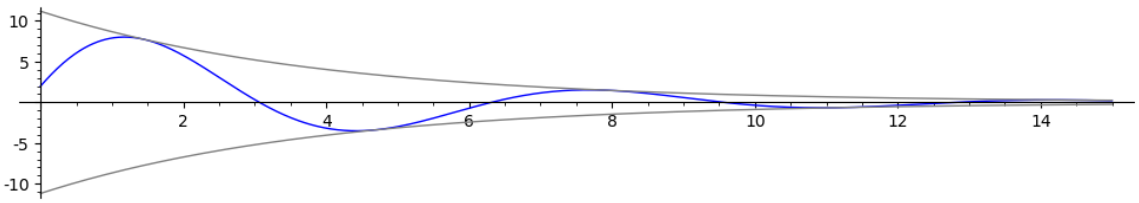
[18]:
```
_K1
```

[18]:
```
_K1
```

Since the solution is in the form of a periodic function multiplied by exponent, we might want to draw the envelope:

[19]:
```
var('t')
pars={_K1:11,_K2:2,omega0:1,g:.51}
A = sqrt(_K1^2+_K2^2).subs(pars)
plot( phi_anal.subs(pars), (t,0,15), figsize=(10,2)) + \
    plot( (A*exp(-1/2*g*t)).subs(pars), (t,0,15),color='gray' ) +\
    plot( (-A*exp(-1/2*g*t)).subs(pars), (t,0,15),color='gray' )
```

[19]:



Where did the formula for A come from?

We have formula which allow to add sin and cos functions with the same frequency:

$$\sqrt{2}\sin\left(\frac{1}{4}\pi + x\right) = \cos(x) + \sin(x)$$

$$\sqrt{a^2+b^2}\sin\left(x + \arctan\left(\frac{b}{a}\right)\right)$$

We have:

[20]:
```
phi_osc = phi_anal.coefficient(e^(-1/2*g*t))
showmath(phi_osc)
```

[20]:

$$K_2\cos\left(\frac{1}{2}\sqrt{-\gamma^2+4\,\omega_0^2}\,t\right) + K_1\sin\left(\frac{1}{2}\sqrt{-\gamma^2+4\,\omega_0^2}\,t\right)$$

We can transform the linear combination of the sin and cos functions into one function with a different amplitude. In order to accomplish this in Sage, we can use wildcard substitutions.

[21]:
```
w0 = SR.wild(0)
w1 = SR.wild(1)
w2 = SR.wild(2)
sub3 =  { w1*sin(w0)+w2*cos(w0):sqrt(w1^2+w2^2)*sin(w0+arctan(w2/w1)) }
```

[22]:
```
showmath(phi_osc.subs(sub3))
```

[22]:

$$\sqrt{K_1^2+K_2^2}\sin\left(\frac{1}{2}\sqrt{-\gamma^2+4\,\omega_0^2}t+\arctan\left(\frac{K_2}{K_1}\right)\right)$$

Alternatively, you can match the solution to a bit more complicated pattern:

[23]:
```
w0 = SR.wild(0)
w1 = SR.wild(1)
w2 = SR.wild(2)
w3 = SR.wild(3)
sub4 =  { w3*(w1*sin(w0)+w2*cos(w0)):w3*sqrt(w1^2+w2^2)*sin(w0+arctan(w2/w1)) }
```

[24]:
```
showmath(phi_anal.subs(sub4))
```

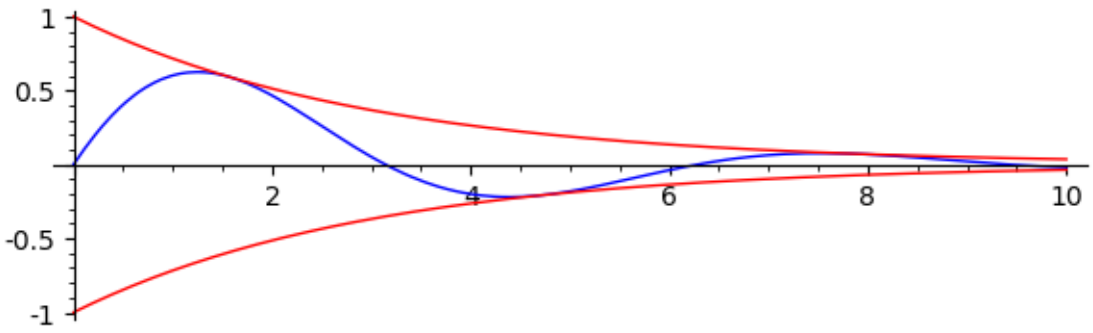
[24]:

$$\sqrt{K_1^2+K_2^2}e^{\left(-\frac{1}{2}\gamma t\right)}\sin\left(\frac{1}{2}\sqrt{-\gamma^2+4\,\omega_0^2}t+\arctan\left(\frac{K_2}{K_1}\right)\right)$$

On the other hand if have expression in the form of $\sin(x+\phi)e^{-x}$, then the exponent term is an envelope:

[25]:
```
plot(sin(x)*exp(-x/3) ,(x,0,10),figsize=(6,2))+\
 plot([exp(-x/3),-exp(-x/3)],(x,0,10),color='red')
```

[25]:

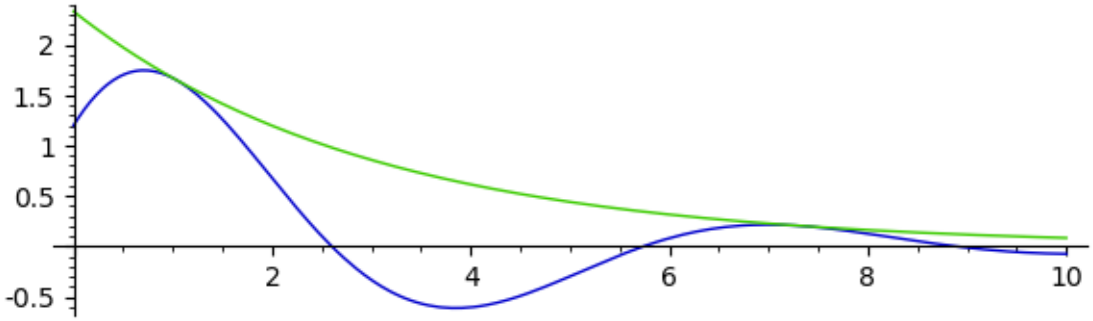


we have therefore:

[26]:
```
a,b = 2,1.2
plot([(a*sin(x)+b*cos(x))*exp(-x/3),sqrt(a^2+b^2)*exp(-x/3)],\
```

```
    (x,0,10), figsize=(6,2))
```

[26]:



3.3.2 Two real roots

If the determinant Δ is positive (but non-zero), i.e.:

$$\gamma - 2\omega_0 > 0, \tag{4}$$

then we have:

[27]:
```
var('omega omega0')
var('g', latex_name='\gamma')
forget()
assume(g-2*omega0>0)
assume(g>0)
assume(omega0>0)
show( assumptions() )
osc = diff(X,t,2) == -g*diff(X,t)-omega0^2*X
showmath(osc)
```

```
[g - 2*omega0 > 0, g > 0, omega0 > 0]
```

[27]:

$$\frac{\partial^2}{(\partial t)^2} X(t) = -\omega_0^2 X(t) - \gamma \frac{\partial}{\partial t} X(t)$$

[28]:
```
phi_anal = desolve(osc,dvar=X,ivar=t)
showmath(phi_anal)
```

[28]:

$$K_2 e^{\left(-\frac{1}{2}\left(\gamma+\sqrt{\gamma^2-4\omega_0^2}\right)t\right)} + K_1 e^{\left(-\frac{1}{2}\left(\gamma-\sqrt{\gamma^2-4\omega_0^2}\right)t\right)}$$

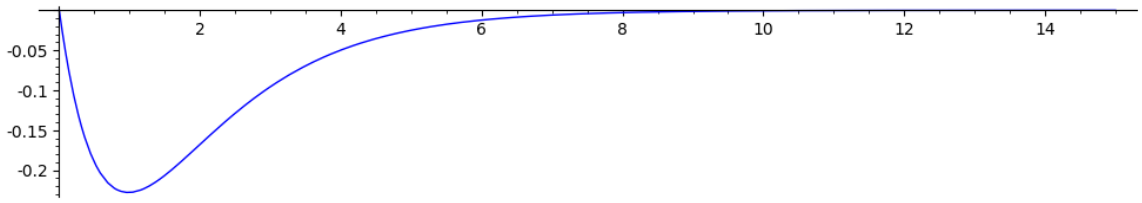
We see that the solution does not contain periodic functions but only is the sum of two exponents, with negative exponents.

Task: Prove that the exponent $k_1 e^{\left(-\frac{1}{2}\left(\gamma-\sqrt{\gamma^2-4\,\omega_0^2}\right)t\right)}$, for $t > 0$, it's negative.

In this case, solutions are decaying without oscillations, however, for certain parameters it is possible that a single extremum will occur:

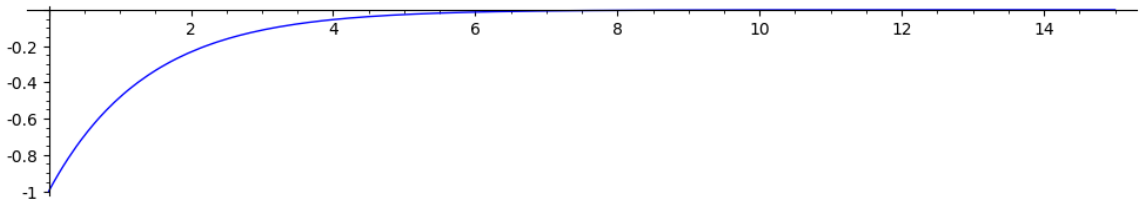
[29]:
```
var('t')
plot( phi_anal.subs({_K1:-1,_K2:1,omega0:1,g:2.1}), (t,0,15), figsize=(10,2))
```

[29]:



[30]:
```
var('t')
plot( phi_anal.subs({_K1:-1,_K2:0,omega0:1,g:2.1}), (t,0,15), figsize=(10,2))
```

[30]:



3.3.3 Degenerate case

Consider the case when the characteristic equation disappears. It is fullfilled when:

$$\gamma - 2\omega_0 < 0 \tag{5}$$

Let us have a look how the general solution looks:

[31]:
```
var('omega omega0')
var('g', latex_name='\gamma')

forget()
```

```
assume(g-2*omega0==0)
assume(g>0)
assume(omega0>0)
show( assumptions() )
osc = diff(X,t,2) == -g*diff(X,t)-omega0^2*X
showmath(osc)
phi_anal = desolve(osc, dvar=X, ivar=t)
showmath(phi_anal)
```

```
[g - 2*omega0 == 0, g > 0, omega0 > 0]
```

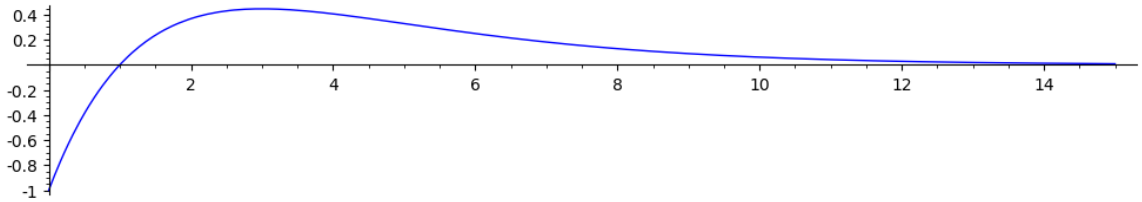
[31]:

$$(K_2 t + K_1) e^{\left(-\frac{1}{2}\gamma t\right)}$$

This case is called critically damped.

[32]:
```
var('t')
plot( phi_anal.subs({_K1:-1,_K2:1,g:1}), (t,0,15), figsize=(10,2))
```

[32]:



3.3.4 Power dissipation

Having analytical solution one can easily describe how energy is dissipated in the system. For given $omega_0$ we might expect that power remains constant in two cases: $\gamma = 0$ and $\gamma \to \infty$. Let's check it out - first we can obtain analytical solutions in oscilating and damped regimes. We will take initial condition $x_0 = 1$ and $v_0 = 1$:

[33]:
```
forget()
assume(g>2*omega0)
assume(g>0)
assume(omega0>0)
x_damped = desolve(osc, dvar=X, ivar=t, ics=[0,1,0] )

forget()
assume(g<2*omega0)
assume(g>0)
```

```
assume(omega0>0)
x_oscil = desolve(osc, dvar=X, ivar=t, ics=[0,1,0] )
```

Now, let's define functions which define dissipated power:

$$P = F_{friction} v = (\gamma v) v = \gamma v^2 \tag{6}$$

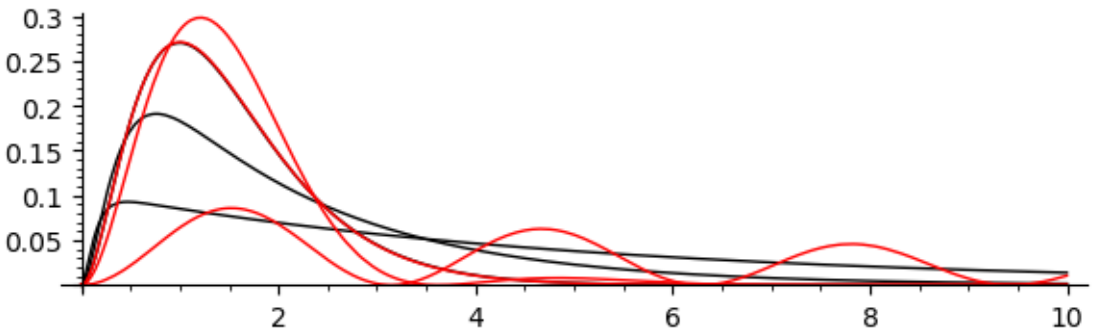
and total energy (for $m = 1$):

$$E_{tot} = \frac{1}{2} v^2 + \frac{1}{2} \omega_0 x^2 \tag{7}$$

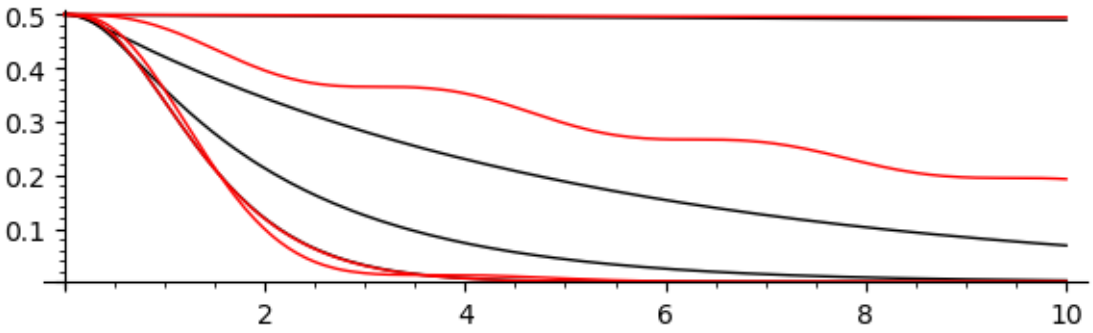
[34]:
```
E = lambda x: 1/2 * x.diff(t)^2 + 1/2*omega0*x^2
P = lambda x: g * (x.diff(t))^2
```

Now we can plot power energy and trajectory as a function of time:

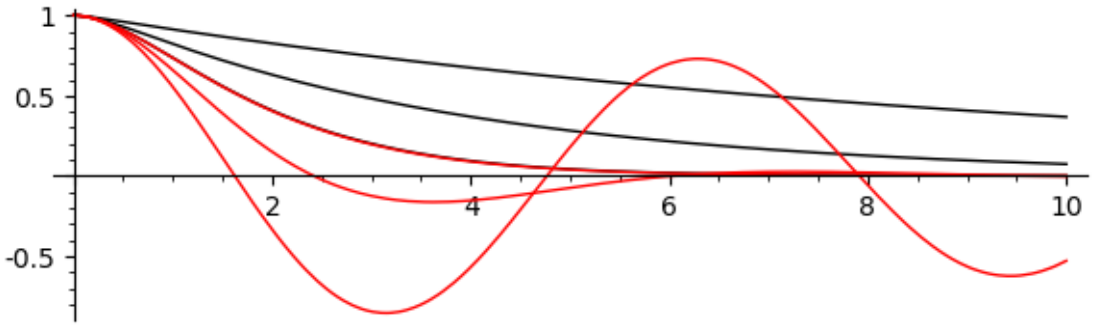
[35]:
```
plt_power = plot([P(x_damped).subs({omega0:1,g:g_}) for g_ in [2.01,4,10]],\
     (t,0,10), color='black')
plt_power += plot([P(x_oscil).subs({omega0:1,g:g_}) for g_ in [.1,1,1.99]],\
      (t,0,10), color='red')
plt_power.show(figsize=(6,2))
```



[36]:
```
plt_E = plot([E(x_damped).subs({omega0:1,g:g_}) for g_ in [2.01,4,10,1e3]],\
             (t,0,10),color='black')
plt_E += plot([E(x_oscil).subs({omega0:1,g:g_}) for g_ in [1e-3,.1,1,1.99]],\
              (t,0,10),color='red')
plt_E.show(figsize=(6,2))
```



[37]:
```
plt_x = plot([(x_damped).subs({omega0:1,g:g_}) for g_ in [2.01,4,10]],\
             (t,0,10),color='black')
plt_x += plot([(x_oscil).subs({omega0:1,g:g_}) for g_ in [.1,1,1.99]],\
              (t,0,10),color='red')
plt_x.show(figsize=(6,2))
```



Note that for damping near the critical (i.e. $\Delta = 0$) regime, the power dissipation is maxiamal.

3.4 Forced harmonic oscillator

If on a harmonic oscillator with damping a certain force depends on time of $\sin(\omega t)$ type, then we are dealing with a forced harmonic oscillator. In this case, the equation of motion contains a term independent of the position $x(t)$, but explicitely dependent on time. Such an equation is a linear non-homogeneous differential equation and we can give its analytical solutions. Let's see how they look:

[38]:
```
var('a omega omega0')
var('g', latex_name='\gamma')
forget()
assume(g-2*omega0<0)
```

```
Phi = function('Phi')(t)
assume(g>0)
assume(omega0>0)
osc = diff(Phi,t,2)+ g*diff(Phi,t) + omega0^2*Phi -a*sin(omega*t)

showmath(osc)
```

[38]:

$$\omega_0^2\Phi\left(t\right)-a\sin\left(\omega t\right)+\gamma\frac{\partial}{\partial t}\Phi\left(t\right)+\frac{\partial^2}{(\partial t)^2}\Phi\left(t\right)$$

[39]:
```
phi_anal,method = desolve(osc,dvar=Phi,ivar=t,show_method=True)
print(method)
showmath(phi_anal)
```

```
variationofparameters
```

[39]:

$$\left(K_2\cos\left(\frac{1}{2}\sqrt{-\gamma^2+4\,\omega_0^2}t\right)+K_1\sin\left(\frac{1}{2}\sqrt{-\gamma^2+4\,\omega_0^2}t\right)\right)e^{\left(-\frac{1}{2}\gamma t\right)}-\frac{a\gamma\omega\cos\left(\omega t\right)+\left(a\omega^2-a\omega_0^2\right)\sin\left(\omega t\right)}{\gamma^2\omega^2+\omega^4-2\,\omega^2\omega_0^2+\omega_0^4}$$

It can be seen that we have a solution in the form of a general solution of a free damped oscillator, hence a homogeneous equation (contains k_1 and k_2 constants) and a solution that does not contain free constants. It is called a special solution to the heterogeneous equation. It can be noted that this solution is the one that survives in the $t\to\infty$ limit.

[40]:
```
r_sz = phi_anal.operands()[1]
showmath(r_sz)
```

[40]:

$$-\frac{a\gamma\omega\cos\left(\omega t\right)+\left(a\omega^2-a\omega_0^2\right)\sin\left(\omega t\right)}{\gamma^2\omega^2+\omega^4-2\,\omega^2\omega_0^2+\omega_0^4}$$

Let's transform a special solution to the form:

$$A\sin(\omega t+\phi).$$

To do this, we extract the numerator and denominator:

[41]:
```
expr_denom = r_sz.denominator()
expr = r_sz.numerator()
showmath([expr,expr_denom])
```

[41]:

$$\left[-a\gamma\omega\cos\left(\omega t\right)-a\omega^2\sin\left(\omega t\right)+a\omega_0^2\sin\left(\omega t\right),\gamma^2\omega^2+\omega^4-2\,\omega^2\omega_0^2+\omega_0^4\right]$$

The numerator is a sum of sin and cos with different amplitudes. The formula:

$$a\sin\left(x\right)+b\cos\left(x\right)=\sqrt{a^2+b^2}\sin\left(x+\arctan\left(\frac{b}{a}\right)\right)$$

can be used Use for this purpose to put with wildcards (an. Wildcards):

[42]:
```
w0 = SR.wild(0)
w1 = SR.wild(1)
w2 = SR.wild(2)
w3 = SR.wild(3)
sub3 =  { w1*sin(w0)+w2*cos(w0):sqrt(w1^2+w2^2)*sin(w0+arctan(w2/w1)) }
```

Let's check how the substitution works on the formula:

[43]:
```
var('a b x')
assume(a>0)
(a*sin(x)+b*cos(x)).subs(sub3).show()
```

```
sqrt(a^2 + b^2)*sin(x + arctan(b/a))
```

and expand the obtained formula to check:

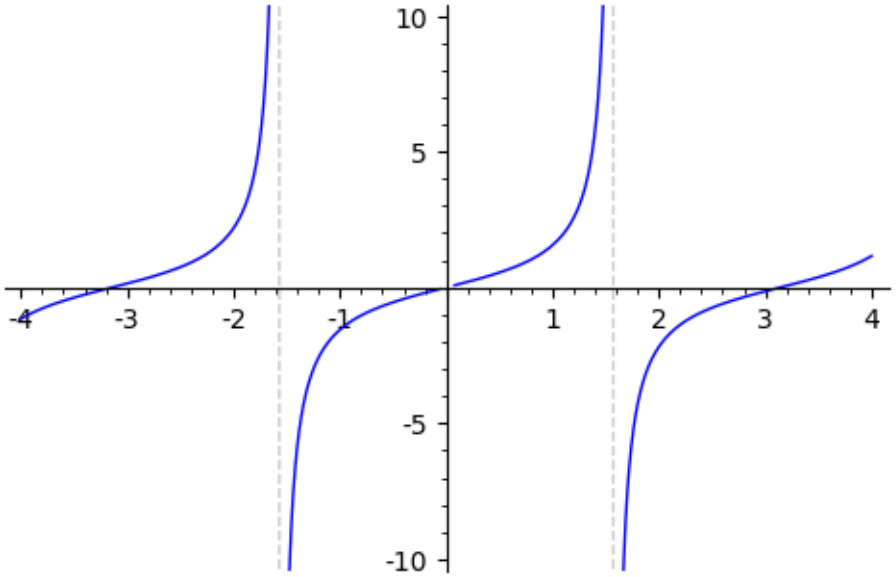
[44]:
```
assume(a>0)
(a*sin(x)+b*cos(x)).subs(sub3).full_simplify().show()
```

```
b*cos(x) + a*sin(x)
```

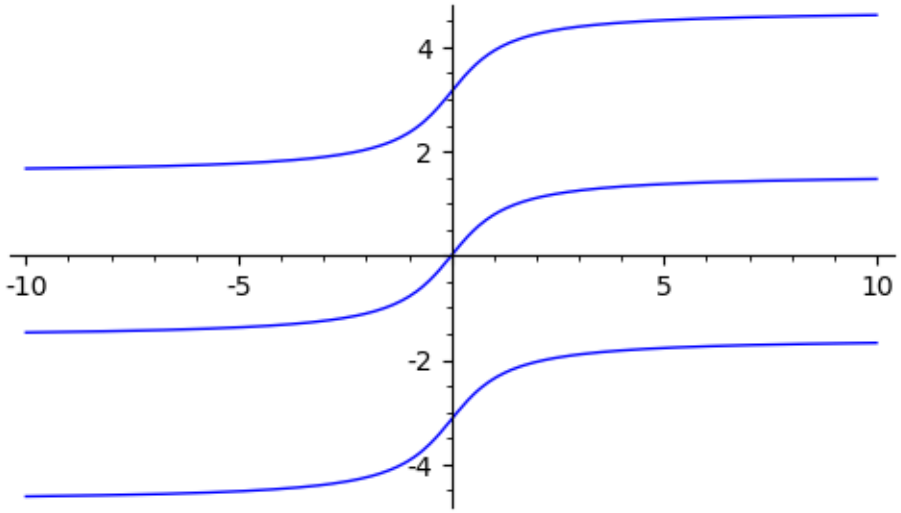
Everything looks nice, but we have a classic problem to choose the function's branch. Let's look at the graph of the $\tan(x)$ function:

[45]:
```
print (arctan(10)-pi).n(),(arctan(-10)-pi).n()

plot(tan(x),(x,-4.,4),detect_poles='show',figsize=5,ymin=-10,ymax=10).show()
sum([plot(arctan(x)+n*pi,(x,-10.,10),figsize=(5,3)) for n in range(-1,2)])
```

```
-1.67046497928606 -4.61272032789353
```



[45]:



We should take the branch $y \in -\pi..0$ and the function in Sage takes $y \in -\pi..\pi$. Therefore, it is better to use the two-argument `arctan2`:

[46]:
```
w4 = SR.wild(4)
w5 = SR.wild(5)
sub3a = {arctan(w4/w5):(arctan2(w4,w5)-pi)}
```

Returning to our oscillator, we can use the substitution:

Note `collect (sin (omega * t))` is needed for the pattern to be in a form in which the substitution can be used.

[47]:
```
expr = r_sz.numerator().collect(sin(omega*t))
showmath(expr)
```

[47]:

$$-a\gamma\omega\cos(\omega t) - \left(a\omega^2 - a\omega_0^2\right)\sin(\omega t)$$

[48]:
```
expr = expr.subs(sub3)
showmath(expr)
```

[48]:

$$\sqrt{a^2\gamma^2\omega^2 + \left(a\omega^2 - a\omega_0^2\right)^2}\sin\left(\omega t + \arctan\left(\frac{a\gamma\omega}{a\omega^2 - a\omega_0^2}\right)\right)$$

[49]:
```
expr = expr.subs(sub3a)
showmath(expr)
```

[49]:

$$\sqrt{a^2\gamma^2\omega^2 + \left(a\omega^2 - a\omega_0^2\right)^2}\sin\left(-\pi + \omega t + \arctan\left(a\gamma\omega, a\omega^2 - a\omega_0^2\right)\right)$$

and in all its glory our formula is presented as:

[50]:
```
r_szczegolne = (expr/expr_denom).canonicalize_radical()
showmath(r_szczegolne)
```

[50]:

$$-\frac{a\sin\left(\omega t + \arctan\left(a\gamma\omega, a\omega^2 - a\omega_0^2\right)\right)}{\sqrt{\gamma^2\omega^2 + \omega^4 - 2\,\omega^2\omega_0^2 + \omega_0^4}}$$

If for some reason we would like to pick its coefficients then we have:

[51]:
```
w0 = SR.wild(0)
w1 = SR.wild(1)
m = r_szczegolne.match(w0*sin(w1))
showmath(m)
```

[51]:

$$\left\{\$1 : \omega t + \arctan\left(a\gamma\omega, a\omega^2 - a\omega_0^2\right), \$0 : -\frac{a}{\sqrt{\gamma^2\omega^2 + \omega^4 - 2\,\omega^2\omega_0^2 + \omega_0^4}}\right\}$$

check:

[52]:
```
showmath(w0.subs(m)*sin(w1.subs(m)))
```

[52]:

$$-\frac{a\sin\left(\omega t + \arctan\left(a\gamma\omega, a\omega^2 - a\omega_0^2\right)\right)}{\sqrt{\gamma^2\omega^2 + \omega^4 - 2\,\omega^2\omega_0^2 + \omega_0^4}}$$

Thus, the amplitude of the oscillator in the limit $t \to \infty$ is:

[53]:
```
A = -w0.subs(m)
showmath(A)
```

[53]:

$$\frac{a}{\sqrt{\gamma^2\omega^2 + \omega^4 - 2\,\omega^2\omega_0^2 + \omega_0^4}}$$

It is a famous formula that contains the phenomenon of resonance. If the damping is weak then in the case of $\omega_0 \to \omega$, the amplitude of the special solution will increase to very large values, and will be infinite if the system is not damped. This means that the energy from the external source can be pumped in particularly efficiently if the frequency of force is consistent with the own frequency of the system.

The strength of this phenomenon was seen by soldiers passing over the bridge in Angers (http://en.wikipedia.org/wiki/Angers_Bridge), which had just its own frequency equal to the frequency of the military march. The bridge during the march of the column of the army, stared to swing to such an amplitude that it was destroyed.

[54]:
```
phase = w1.subs(m)-omega*t-pi
showmath(phase)
```

[54]:

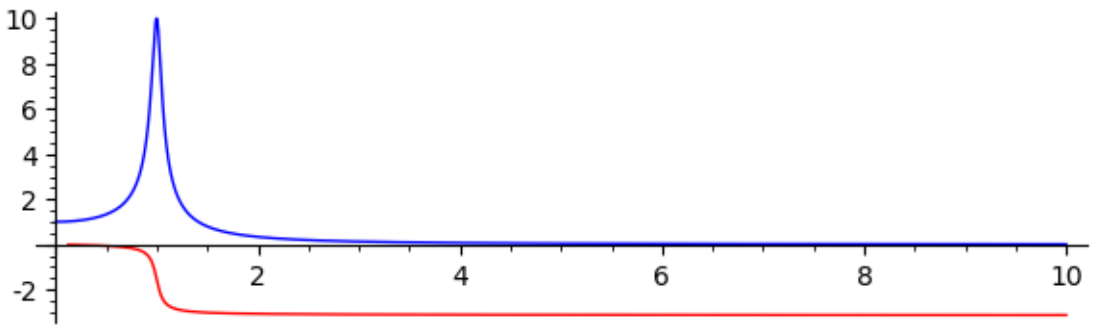
$$-\pi + \arctan\left(a\gamma\omega, a\omega^2 - a\omega_0^2\right)$$

The maximal amplitude depends on the attenuation coefficient:

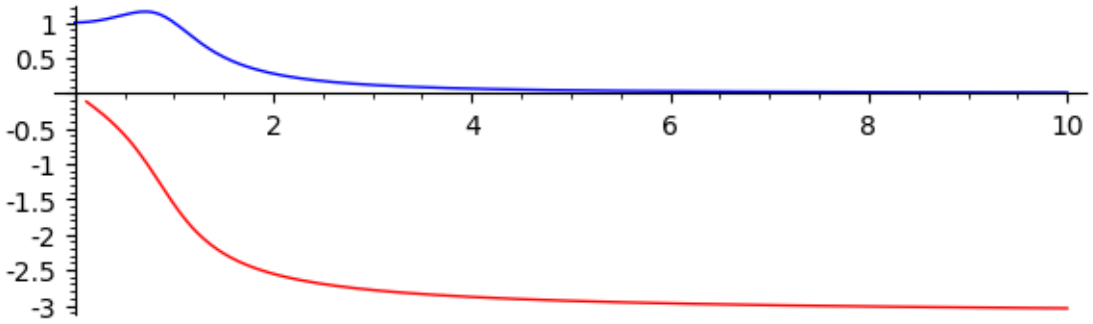
[55]:
```
#@interact
def plot_A_phase(g_=slider(0.01,1,0.01,label="$\gamma$",default=0.2)):
    pars = {g:g_,a:1,omega0:1}
    print ( g^2*omega^2 + omega^4 - 2*omega^2*omega0^2 + omega0^4 ).subs(pars)
    p = plot(A.subs(pars),(omega,0,10))
    p += plot(phase.subs(pars),(omega,0,10),color='red',detect_poles="show")
    p.show(figsize=(6,2))
```

[56]:
```
plot_A_phase(g_=0.1),plot_A_phase(g_=1)
```

```
omega^4 - 1.99000000000000*omega^2 + 1
```



```
omega^4 - omega^2 + 1
```



[56]:
```
(None, None)
```

3.4.1 Numerical analysis

We can also numerically integrate the equation of motion for the harmonic oscillator. It requires to know all parameters and initial conditions numerically before, and it does not allow for easy analysis of the generic properties of the solutions. On the other hand numerical procedure can be applied easily to system of ODE which do not have analytical solution. Here, let us compare results obtained from numerical solution with $t \to \infty$ formula:

[57]:
```
var('phid',latex_name=r'\dot\varphi')
var('phi',latex_name=r'\varphi')
```

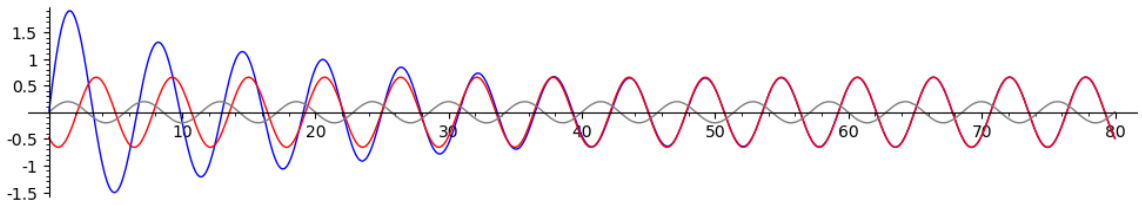
[57]:
```
phi
```

[58]:
```
rhs = solve(osc,diff(Phi(t), t, 2))[0].rhs()
#@interact
def plot_A_traj(g_=slider(0.01,2.2,0.01,label="$\gamma$",default=0.2),\
                omega_=slider(0.01,5.134,0.01,label="$\omega$",default=1.4)):
    pars = {g:g_,a:1,omega0:1}
    print ( g^2*omega^2 + omega^4 - 2*omega^2*omega0^2 + omega0^4 ).subs(pars)
    p = plot(A.subs(pars),(omega,0,10),figsize=(10,2))
    p += plot(phase.subs(pars),(omega,0,10),color='red',detect_poles="show")
    pars = {omega:omega_,omega0:1,a:1,g:g_}
```

[59]:
```
pars = {omega:1+0.1,omega0:1,a:0.2,g:.2}
ode = [phid,rhs.subs({Phi:phi,diff(Phi,t):phid}).subs(pars)]
```

```
times = srange(0,80,0.001)
ics = [0.0,2.1]
sol = desolve_odeint(ode,ics,times,[phi,phid])
line( zip(times,sol[::1,0]),figsize=(10,2) ) + \
 plot( r_szczegolne.subs(pars), (t,0,80),color='red') +\
 plot( (a*sin(omega*t)).subs(pars), (t,0,80),color='gray')
```

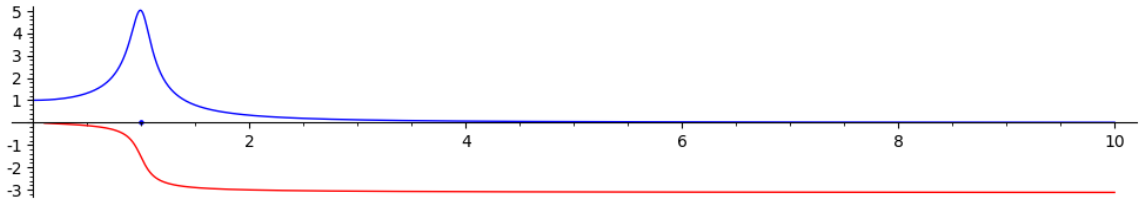
[59]:

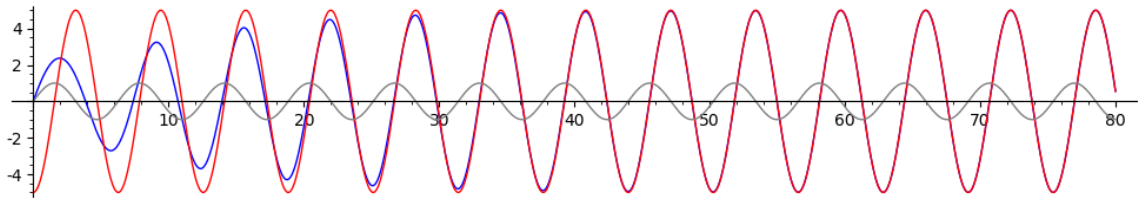


[60]:
```
rhs = solve(osc,diff(Phi(t), t, 2))[0].rhs()
#@interact
def plot_A_traj(g_=slider(0.01,2.2,0.01,label="$\gamma$",default=0.2),\
                omega_=slider(0.01,5.134,0.01,label="$\omega$",default=1.4)):
    pars = {g:g_,a:1,omega0:1}
    print ( g^2*omega^2 + omega^4 - 2*omega^2*omega0^2 + omega0^4 ).subs(pars)
    p = plot(A.subs(pars),(omega,0,10),figsize=(10,2))
    p += plot(phase.subs(pars),(omega,0,10),color='red',detect_poles="show")
    pars = {omega:omega_,omega0:1,a:1,g:g_}
    ode = [phid,rhs.subs({Phi:phi,diff(Phi,t):phid}).subs(pars)]

    times=srange(0,80,0.001)
    ics=[0.0,2.1]
    sol = desolve_odeint(ode,ics,times,[phi,phid])
    r_szczegolne2 = a*sin(omega*t +pi- \
                          arctan2(-g*omega,(omega^2 - omega0^2)))/\
          sqrt(g^2*omega^2+ omega^4 - 2*omega^2*omega0^2 + omega0^4)
    p2 = line( zip(times,sol[::1,0]),figsize=(10,2) ) + \
     plot( r_szczegolne.subs(pars), (t,0,80),color='red') +\
     plot( (a*sin(omega*t)).subs(pars), (t,0,80),color='gray')
    p += point([omega_,0])
    p.show()
    p2.show()
```

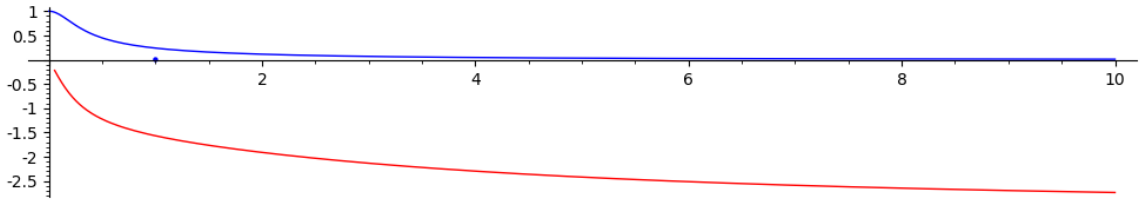
[61]:
```
plot_A_traj(g_=0.2,omega_=1.)
plot_A_traj(g_=4.2,omega_=1.)
```

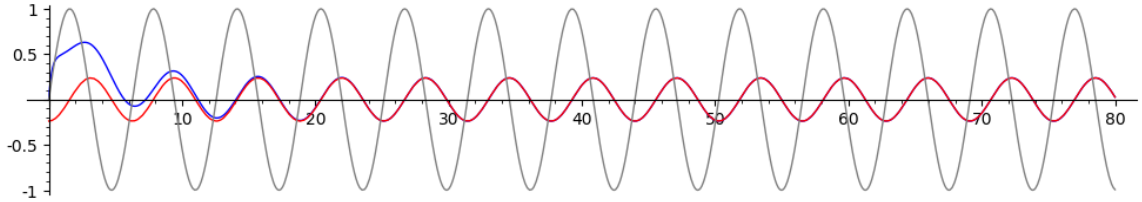
```
omega^4 - 1.96000000000000*omega^2 + 1
```






```
omega^4 + 15.6400000000000*omega^2 + 1
```





We can further analyze the formula for resonance and e.g. find its maximum in *omega*:

[62]:
```
showmath(A.diff(omega))
```

[62]:

$$-\frac{\left(\gamma^2\omega+2\,\omega^3-2\,\omega\omega_0^2\right)a}{\left(\gamma^2\omega^2+\omega^4-2\,\omega^2\omega_0^2+\omega_0^4\right)^{\frac{3}{2}}}$$

[63]:
```
sol = solve(A.diff(omega), omega)
showmath(sol)
```

[63]:

$$\left[\omega=-\sqrt{-\frac{1}{2}\,\gamma^2+\omega_0^2},\omega=\sqrt{-\frac{1}{2}\,\gamma^2+\omega_0^2},\omega=0\right]$$

[64]:
```
omega_max = sol[1].rhs()
showmath(omega_max)
```

[64]:

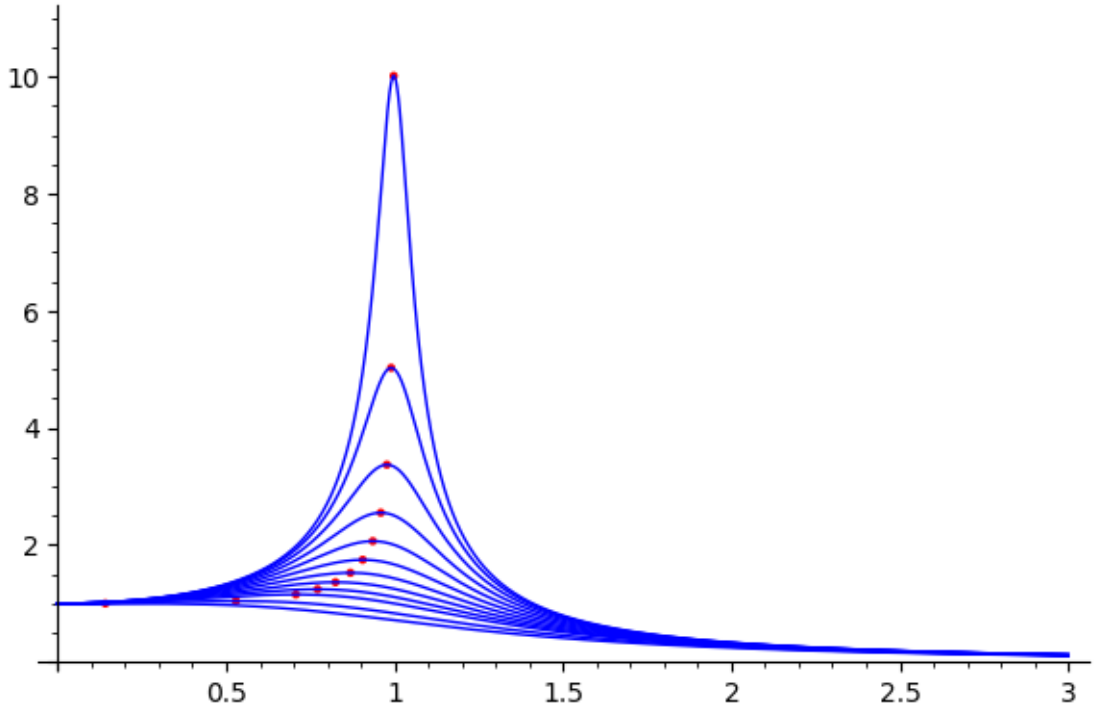
$$\sqrt{-\frac{1}{2}\,\gamma^2+\omega_0^2}$$

[65]:
```
showmath( (omega_max^2).factor() )
```

[65]:

$$-\frac{1}{2}\,\gamma^2+\omega_0^2$$

[66]:
```
p=[]
for g_ in srange(0.1,1,0.1)+srange(1,1.41,.2):
    pars = {g:g_,a:1,omega0:1}
    omega_max_v =  omega_max.subs(pars).n()
    if omega_max_v.is_real():
        p.append( point( (omega_max_v,A.subs(pars).subs(omega==omega_max_v)␣
 ↪),color='red' ) )
    p.append(  plot(A.subs(pars),(omega,0,3)) )
sum(p).show(ymax=11)
```



In above formula we see that in the weak damping limit the resonance frequency tends to internal frequency of the oscillator. But with increased damping we position moves towards lower values of frequency. Also it can be easily seen that for

$$-\gamma^2 + 2\omega_0^2 < 0$$

because:

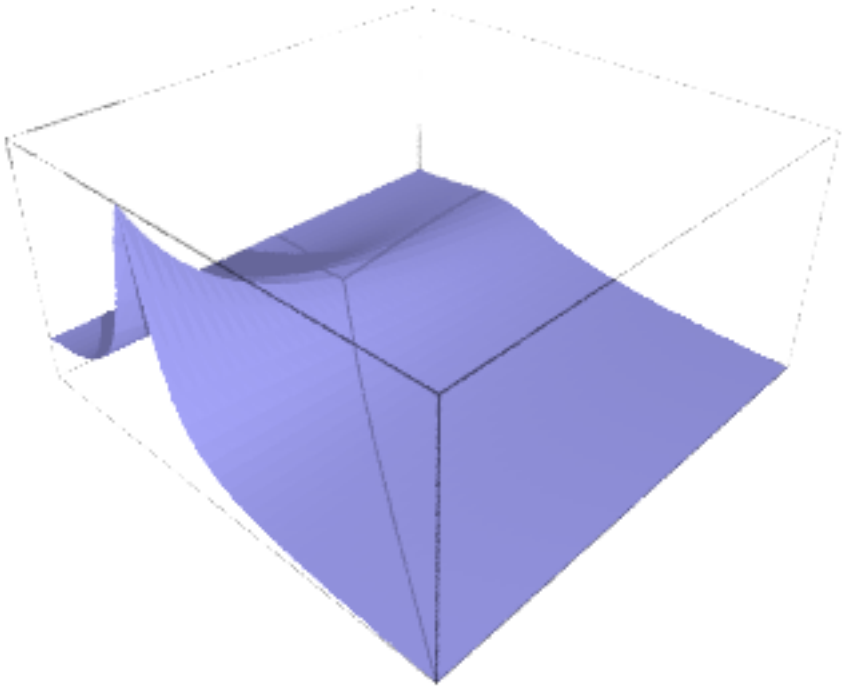
$$-\gamma^2 + 2\omega_0^2 = (\sqrt{2}\omega_0 - \gamma)(\sqrt{2}\omega_0 + \gamma)$$

for

$$\sqrt{2}\omega_0 < \gamma,$$

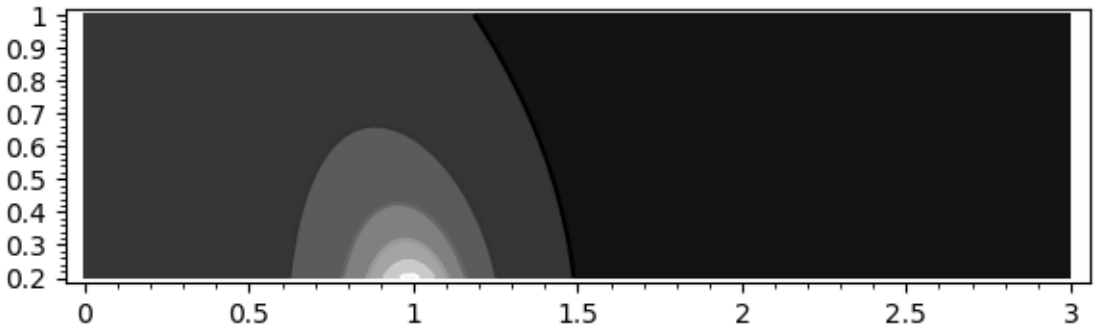
the resonance disappears - i.e. there is not maxima in $A(\omega)$ dependence.

[67]:
```
pars = {a:1,omega0:1}
plot3d( A.subs(pars), (omega,0,3),(g,0.2,1) ).show(viewer='tachyon',figsize=4)
```



[68]:
```
pars = {a:1,omega0:1}
contour_plot( A.subs(pars), (omega,0,3),(g,0.2,1) )
```

[68]:



3.4.2 Power absorbed

We can also analyze how much power is absorbes by periodically driven harmonic oscillator. We can use the solution $t \to \infty$ which has a form:

[69]:
```
showmath( r_szczegolne )
```

[69]:

$$-\frac{a\sin\left(\omega t+\arctan\left(a\gamma\omega, a\omega^2-a\omega_0^2\right)\right)}{\sqrt{\gamma^2\omega^2+\omega^4-2\,\omega^2\omega_0^2+\omega_0^4}}$$

Power dissipation is force times the velocity, so:

[70]:
```
P = g*(r_szczegolne.diff(t))^2
```

We want to integrate the power oven one period of oscillations

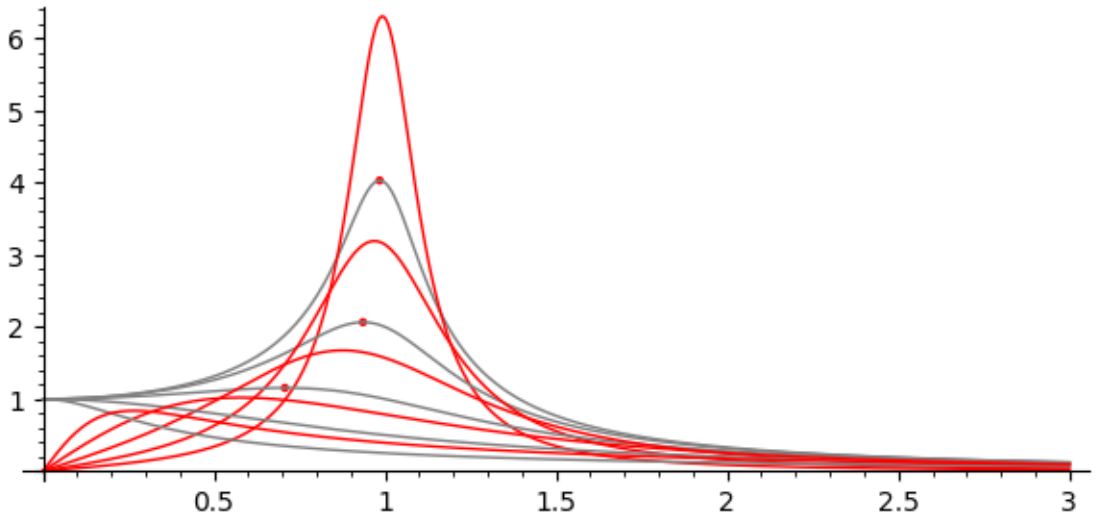
[71]:
```
showmath( P.integrate(t,0,2*pi) )
```

[71]:

$$\frac{a^2\gamma\omega^2\left(\frac{4\,\pi\omega-2\,\arctan\left(-\frac{\gamma\omega}{\omega^2-\omega_0^2}\right)-\sin\left(-4\,\pi\omega+2\,\arctan\left(-\frac{\gamma\omega}{\omega^2-\omega_0^2}\right)\right)}{\omega}+\frac{2\,\arctan\left(-\frac{\gamma\omega}{\omega^2-\omega_0^2}\right)+\sin\left(2\,\arctan\left(-\frac{\gamma\omega}{\omega^2-\omega_0^2}\right)\right)}{\omega}\right)}{4\left(\gamma^2\omega^2+\omega^4-2\,\omega^2\omega_0^2+\omega_0^4\right)}$$

We can plot this formula together with an expression for amplitude.

[72]:
```
p=[]
for g_ in [0.25,0.5,1,2,4]:
    pars = {g:g_,a:1,omega0:1}
    assume(omega>0)
    omega_max_v =  omega_max.subs(pars).n()
    if omega_max_v.is_real():
        p.append( point( (omega_max_v,A.subs(pars).subs(omega==omega_max_v)␣
↪),color='red' ) )
    p.append(  plot(A.subs(pars),(omega,0,3), color='gray') )
    p.append(  plot(P.subs(pars).integrate(t,0,2*pi/omega)/
↪2,(omega,0,3),color='red') )
sum(p).show(figsize=(6,3))
```


6
5
4
3
2
1
0.5
1
1.5
2
2.5
3

4 Particle in one-dimensional potential well

4.1 General remarks

We consider the motion of a point particle in a one-dimensional potential well. Further, we assume that there are no dissipative forces and the energy is conserved.

Instead solving Newton eqaution we will use conservation of energy (which is an the integral of equation of motion)

$$\frac{m}{2}\left(\frac{dx}{dt}\right)^2 + U(x) = E, \tag{8}$$

First, if we treat velocity $v = \frac{dx}{dt}$ as independent variable we have a parametric equation of the orbit in **phase space** (x, v) of a point particle with energy E:

$$\frac{m}{2}v^2 + U(x) = E$$

Secondly we can solve this equation for time:

$$t = \sqrt{\frac{m}{2}} \int_{-x1}^{x1} \frac{dx}{(\sqrt{E - U(x)}} \tag{9}$$

This is a formula which allows to calculate period of oscillation and similar quantities.

In this section we will consider potentials in the form of

$$U(x) = A|x|^n, \tag{10}$$

where n can be positive integer or a rational number.

Those functions are bounded from below and have only one minimum. It is, therefore a family of potential wells in which frictionless point particle can oscillate. We will use computer algebra and numerical methods to investigate properties of motion is such potential wells.

[1]:
```
load('cas_utils.sage')
```

[2]:
```
x = var('x')
m = var('m')
A = var('A')
assume(A > 0)
assume(m > 0)
y = function('y')(x)
de = m*diff(y,x,2) + 2*A*y == 0
showmath( desolve(de, y,ivar=x) )
```

[2]:

$$K_2 \cos\left(\frac{\sqrt{2}\sqrt{A}x}{\sqrt{m}}\right) + K_1 \sin\left(\frac{\sqrt{2}\sqrt{A}x}{\sqrt{m}}\right)$$

4.2 Particle in potential $|x|^2$

For $n = 2$ we have a harmonic oscillator:

$$U(x) = Ax^2.$$

[3]:
```
#reset()
var('m A x E')
forget()
assume(A > 0)
assume(E > 0)
assume(E,'real')
```

To obtain the integration limits in the formula for the period of oscillation, we must solve the equation:

$$U(x) = E$$

So for the Ax^2 potential, we have:

[4]:
```
U(A,x) = A*x^2
xextr = solve (U(A=A,x=x)==E,x)
showmath(xextr)
```

[4]:

$$\left[x = -\sqrt{\frac{E}{A}}, x = \sqrt{\frac{E}{A}}\right]$$

These formulas describe the values of the oscillator's extremes positions for a given energy. Let's put then in the formula for the period is oscillations 9: :

[5]:
```
period = 2*sqrt(m/2)*integrate( 1/sqrt(E-U(A,x)),(x,x.subs(xextr[0]),x.
↪subs(xextr[1])))
period = period.canonicalize_radical()
showmath(period)
```

[5]:

$$\frac{\sqrt{2}\pi\sqrt{m}}{\sqrt{A}}$$

We see that the period does not depend on energy - means it does not depend on the initial condition. This is a known characteristic of the harmonic oscillator.

4.3 Particle in the $|x|^n$ potential

If $n \neq 2$, the general formula for the period can be written as:

$$T = 4\sqrt{\frac{m}{2}}\frac{1}{\sqrt{E}}\int_0^{x_1}\frac{dx}{\sqrt{1-A/Ex^n}}$$

the integral can be brought to a dimensionless form by substitution

$$\frac{A}{E}x^n = y^n.$$

Is is, in fact a linear relationship between x and y:

$$\left(\frac{A}{E}\right)^{\frac{1}{n}} x = y.$$

Therefore, we can change the variables on y in the sub-expression. To do this, can transform with Sage the expression under integral in the following way:

[6]:
```
var('dx dy A E x y')
var('n',domain='integer')
assume(n>=0)
assume(A>0)
assume(E>0)
ex1 = dx/sqrt(1-A/E*x^n)
showmath(ex1)
```

[6]:

$$\frac{dx}{\sqrt{-\frac{Ax^n}{E}+1}}$$

and we substitute:

[7]:
```
ex2 = ex1.subs({x:(E/A)^(1/n)*y,dx:dy*(E/A)^(1/n)})
showmath( ex2.canonicalize_radical().full_simplify() )
```

[7]:

$$\frac{E^{\left(\frac{1}{n}\right)}dy}{A^{\left(\frac{1}{n}\right)}\sqrt{-y^n+1}}$$

Let's take out the expression that depends on the parameters A and E:

[8]:
```
expr2 = (ex2/dy*sqrt(-y^n + 1)).full_simplify()
showmath( expr2.canonicalize_radical() )
```

[8]:

$$\frac{E^{\left(\frac{1}{n}\right)}}{A^{\left(\frac{1}{n}\right)}}$$

[9]:
```
prefactor = expr2*sqrt(m/2)*4*1/sqrt(E)
showmath( prefactor.canonicalize_radical() )
```

[9]:

$$\frac{2\sqrt{2}\sqrt{m}}{A^{\left(\frac{1}{n}\right)}E^{\frac{n-2}{2n}}}$$

that is, we obtained:

$$T = 4\sqrt{\frac{m}{2}}\frac{1}{A^{1/n}}E^{\frac{1}{n}-\frac{1}{2}}\int_0^1 \frac{dy}{\sqrt{1-y^n}}$$

For $n = 2$, dependence on E disappears, as we already have seen in previous case.

We still need to calculate the integration limit of the y variable. In the integral, the upper limit is the position in which all energy is potential energy:

$$U(x) = E,$$

in our case

$$Ax^n = E.$$

By changing the variables we get:

[10]:
```
solve( (A*x^n == E).subs({x:(E/A)^(1/n)*y}), y)
```

[10]:
```
[y == 1]
```

that is, the integration limit is $y \in (0, 1)$.

The scaled integral can be expressed by the beta function of Euler http://en.wikipedia.org/wiki/Beta_function. We can calculate it:

[11]:
```
var('a')
assume(a,'integer')
assume(a>0)
showmath( assumptions() )
```

[11]:

$$[A > 0, E > 0, \texttt{E is real}, \texttt{n is integer}, n \geq 0, \texttt{a is integer}, a > 0]$$

[12]:
```
integrate(1/sqrt(1-x^(a)),(x,0,1) )
```

[12]:
```
beta(1/2, 1/a)/a
```

We get a formula containing the beta function. It can be evaluated numerically for any values of the a parameter.

[13]:
```
(beta(1/2,1/a)/a).subs({a:2}).n()
```

[13]:
```
1.57079632679490
```

Let's examine this formula numerically. You can use the β function, or numerically estimate the integral. This second approach allows you to explore any potential, not just $U(x) = ax^n$.

[14]:
```
def beta2(a,b):
    return gamma(a)*gamma(b)/gamma(a+b)

a_list = srange(0.02,5,0.1)
a_list2 = [1/4,1/3,1/2,1,2,3,4,5]

integr_list = [ integral_numerical( 1/sqrt(1-x^a_) ,0,1,␣
 ↪algorithm='qng',rule=2)[0] \
               for a_ in a_list ]
integr_list_analytical = [ beta2(1/2, 1/a_)/a_ for a_ in a_list2 ]
```

we obtain some analytically simple formulas:

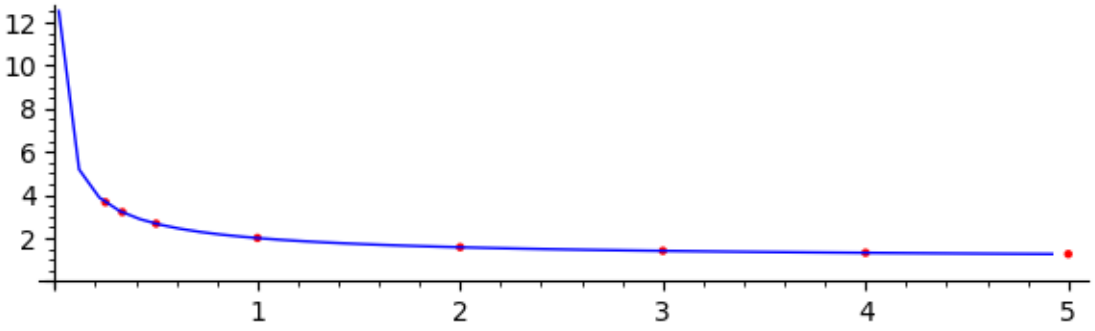
[15]:
```
showmath( integr_list_analytical )
```

[15]:

$$\left[\frac{128}{35}, \frac{16}{5}, \frac{8}{3}, 2, \frac{1}{2}\,\pi, \frac{\sqrt{\pi}\,\Gamma\left(\frac{1}{3}\right)}{3\,\Gamma\left(\frac{5}{6}\right)}, \frac{\sqrt{\pi}\,\Gamma\left(\frac{1}{4}\right)}{4\,\Gamma\left(\frac{3}{4}\right)}, \frac{\sqrt{\pi}\,\Gamma\left(\frac{1}{5}\right)}{5\,\Gamma\left(\frac{7}{10}\right)}\right]$$

Not we can compare those analytical numbers with numerical results, for example on the plot:

[16]:
```
plt_num = list_plot(zip( a_list,integr_list), plotjoined=True )
plt_anal = list_plot(zip( a_list2,integr_list_analytical),color='red')
(plt_num + plt_anal).show(ymin=0,figsize=(6,2))
```



Having an analytical solution, you can examine the asymptotics for large n:

[17]:
```
var('x')
asympt = limit( beta2(1/2, 1/x)/x,x=oo )
asympt
```

[17]:
```
1
```

[18]:
```
plt_asympt = plot(asympt,(x,0,5),linestyle='dashed',color='gray')
```

Let's add a few points for which the integral takes exact values

[19]:
```
l = zip(a_list2[:5],integr_list_analytical[:5])
showmath(l)
```

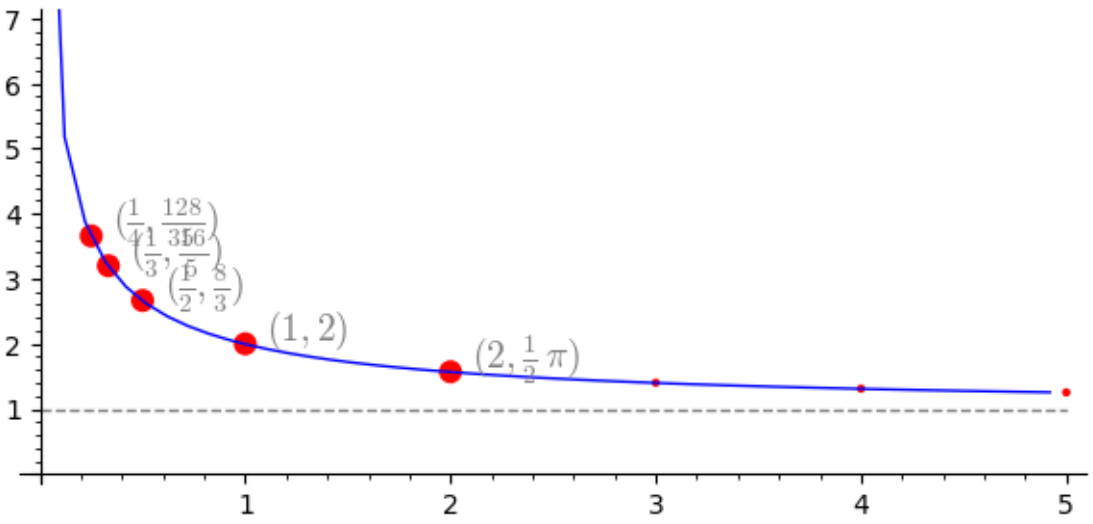
[19]:

$$\left[\left(\frac{1}{4}, \frac{128}{35}\right), \left(\frac{1}{3}, \frac{16}{5}\right), \left(\frac{1}{2}, \frac{8}{3}\right), (1,2), \left(2, \frac{1}{2}\,\pi\right)\right]$$

[20]:
```
def plot_point_labels(l):
    p=[]
    for x,y in l:
        p.append(  text( "$("+latex(x)+", "+latex(y)+")$" ,(x+0.1,y+0.2) ,␣
↪fontsize=14,horizontal_alignment='left',color='gray') )
        p.append( point ( (x,y),size=75,color='red' ) )
    return sum(p)
```

[21]:
```
some_points = plot_point_labels(l)
```

[22]:
```
plt_all = plt_num+plt_anal+plt_asympt+some_points
plt_all.show(figsize=(6,3),ymin=0,ymax=7)
```



4.4 Numerical convergence

The integral

$$\int_0^1 \frac{dx}{\sqrt{1-x^n}}$$

seems to be divergent for n :

[23]:
```
showmath( numerical_integral( 1/sqrt(1-x^(0.25)) , 0, 1)  )
```

[23]:

$$\left(3.6571428158276147, 2.159628070816712 \times 10^{-06}\right)$$

However, choosing the right algorithm gives the correct result:

[24]:
```
a_ = 1/4. # exponent in integral
integral_numerical( 1/sqrt(1-abs(x)^a_) , 0, 1, algorithm='qags')
```

[24]:
```
(3.6571428571462925, 1.611218136687853e-08)
```

lets check it out with an exact formula:

[25]:
```
(beta(1/2,1/a)/a).subs({a:a_}).n()
```

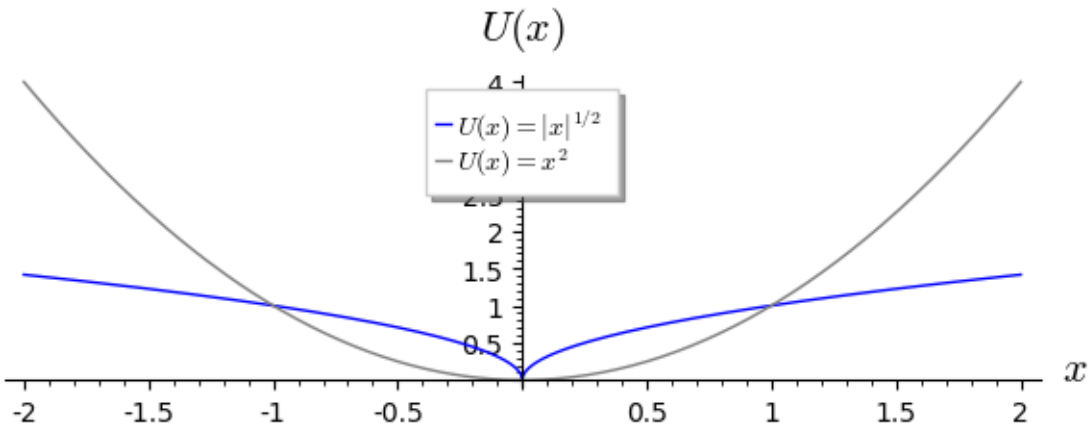
[25]:
```
3.65714285714286
```

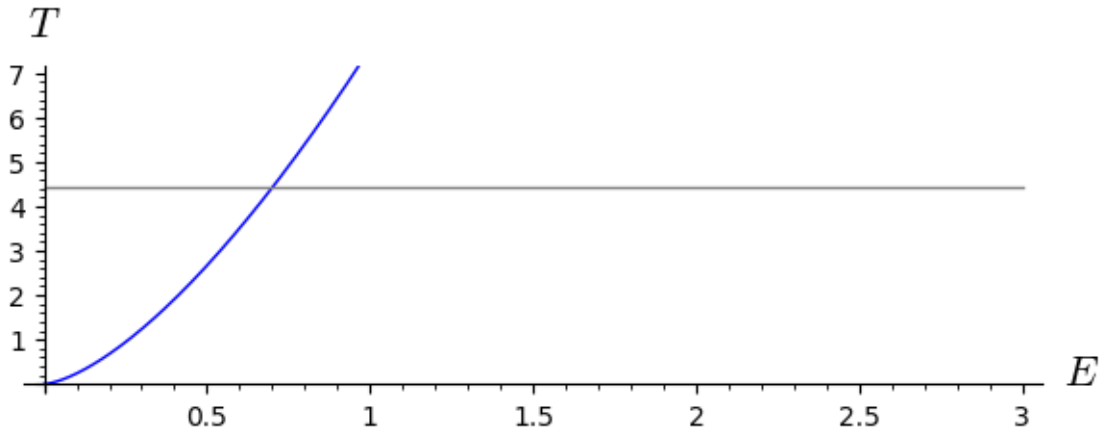
indeed, we see that carefull numerical integration give finite result.

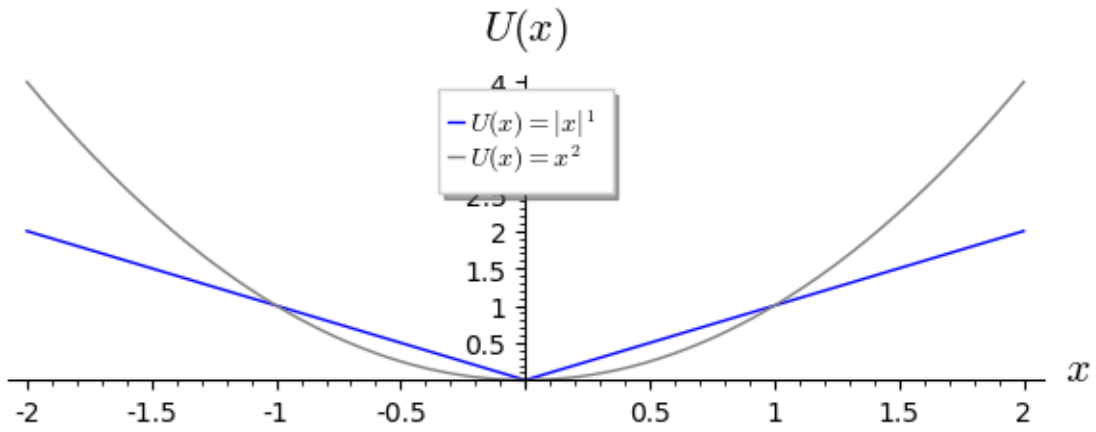
4.5 The dependence of the period on particle energy for different n.

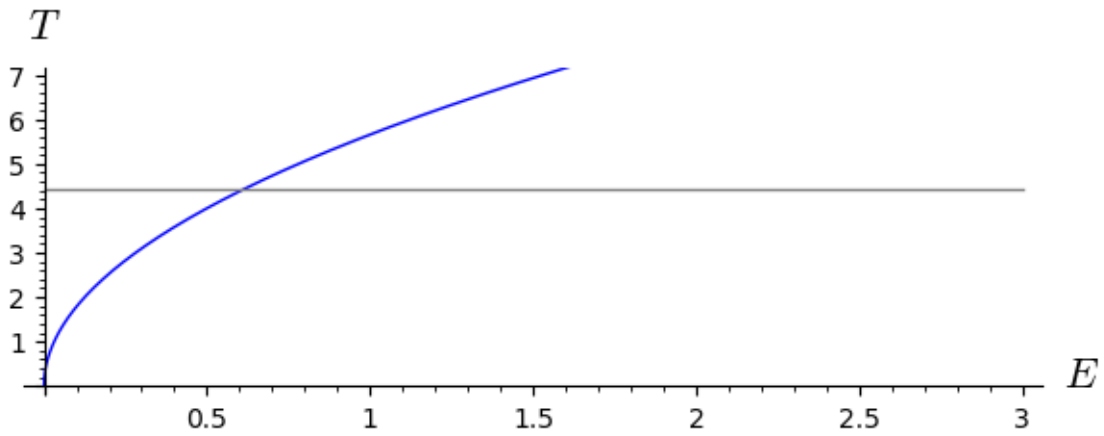
[26]:
```
var('E x n')
def draw_E(n,figsize=(6,2.5)):
    p = []
    p2 = []
    p.append( plot(abs(x)^n,(x,-2,2),\
                   ymin=0,ymax=4,legend_label=r"$U(x)=|x|^{%s}$" % n )   )
    p.append( plot( (x)^2,(x,-2,2),\
                   color='gray',legend_label=r"$U(x)=x^{2}$",\
                   axes_labels=[r"$x$",r"$U(x)$"]  ))

    p2.append(  plot(  4/sqrt(2)*(beta(1/2, 1/n)/n)* E^(1/n-1/2),\
                     (E,0.00,3),ymin=0,ymax=7,axes_labels=[r"$E$",r"$T$"]  ) )
    p2.append(  plot(  4/sqrt(2)*(beta(1/2, 1/2)/2),\
                     (E,0.00,3) ,color='gray' ) )

    show( sum(p), figsize=figsize )
    show( sum(p2), figsize=figsize )
```

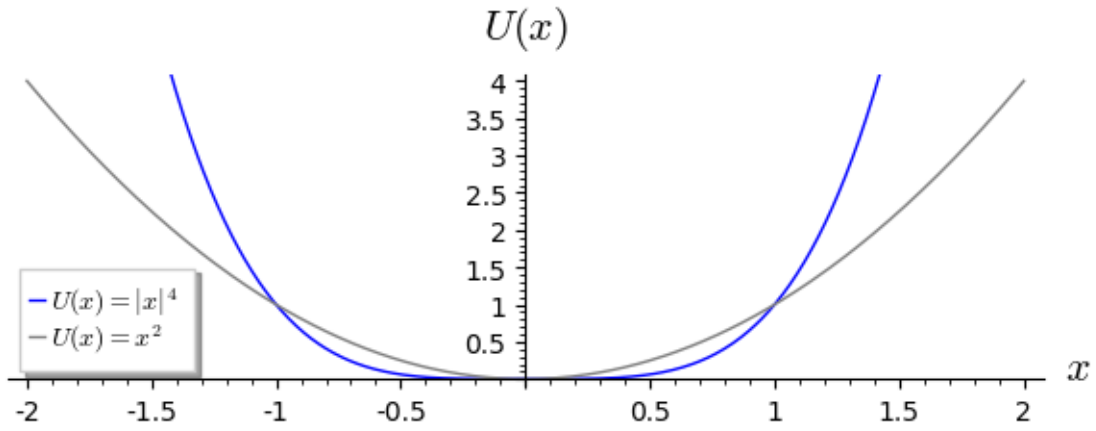
[27]:
```
draw_E(1/2)
draw_E(1)
draw_E(4)
```

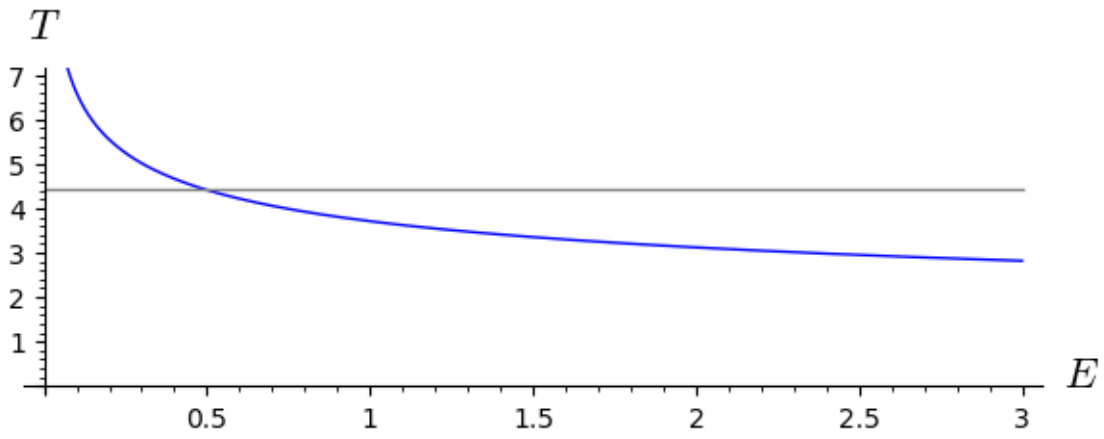





$U(x)$
$U(x) = |x|^1$
$U(x) = x^2$
2
1.5
1
0.5
x
-2
-1.5
-1
-0.5
0.5
1
1.5
2


T
7
6
5
4
3
2
1
E
0.5
1
1.5
2
2.5
3


$U(x)$
$U(x) = |x|^4$
$U(x) = x^2$
4
3.5
3
2.5
2
1.5
1
0.5
x
-2
-1.5
-1
-0.5
0.5
1
1.5
2

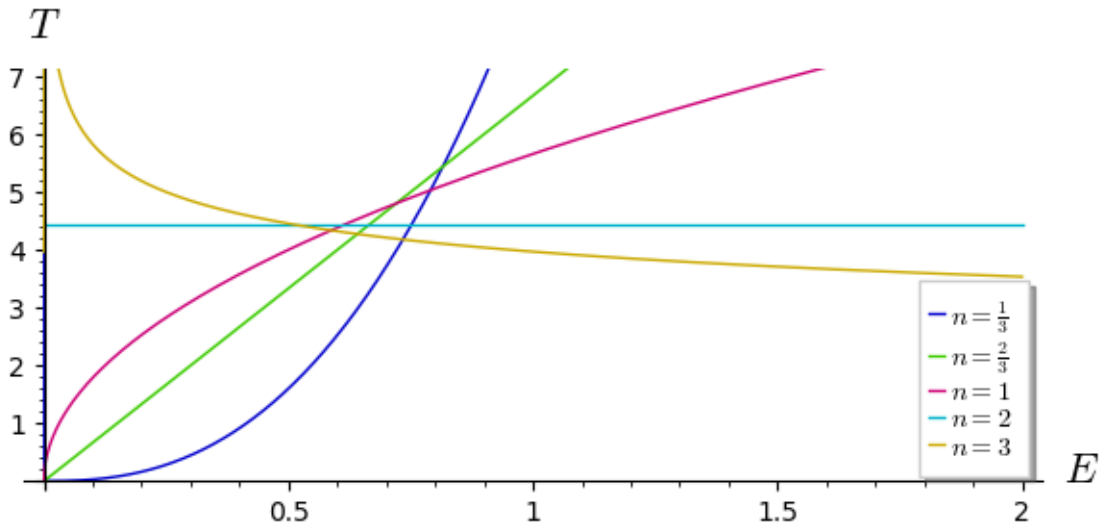


We can plot the dependende of period on energy (i.e. amplitude) $T(E)$ for different values of n. In figure belowe we see that if $n > 2$ then oscillations are faster as energy grow. On the other hand if $n < 1$, oscillations are getting slower with griwing energy.

Another interesting observation of that in potentials with $n > 1$ and $n < 1$ oscillations will become infinitely slow and fast, respectively, if $E \to 0$. For $n > 1$ potential well is *shallow* at the bottom and *steep* far away from the equilibrium point and for $n < 1$ the opposite is true.

[28]:
```
n_s  = [1/3,2/3, 1, 2, 3]
plot( [4/sqrt(2)*(beta(1/2, 1/n_)/n_)* E^(1/n_-1/2) \
                  for n_ in  n_s],\
    (E,0.00,2),axes_labels=[r"$E$",r"$T$"],\
     legend_label=['$n='+str(latex(n_))+'$' for n_ in n_s],\
     ymax=7, figsize=(6,3) )
```

[28]:

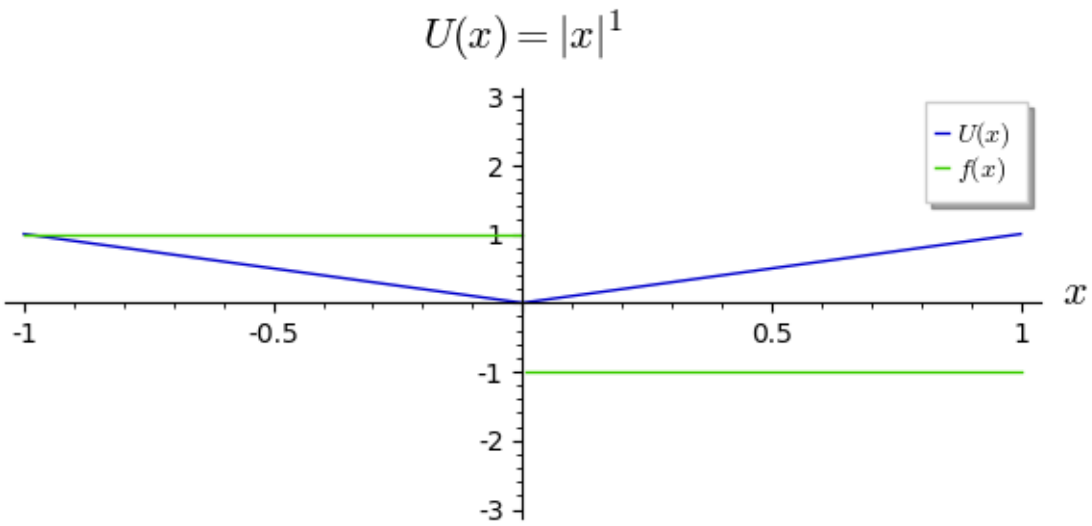


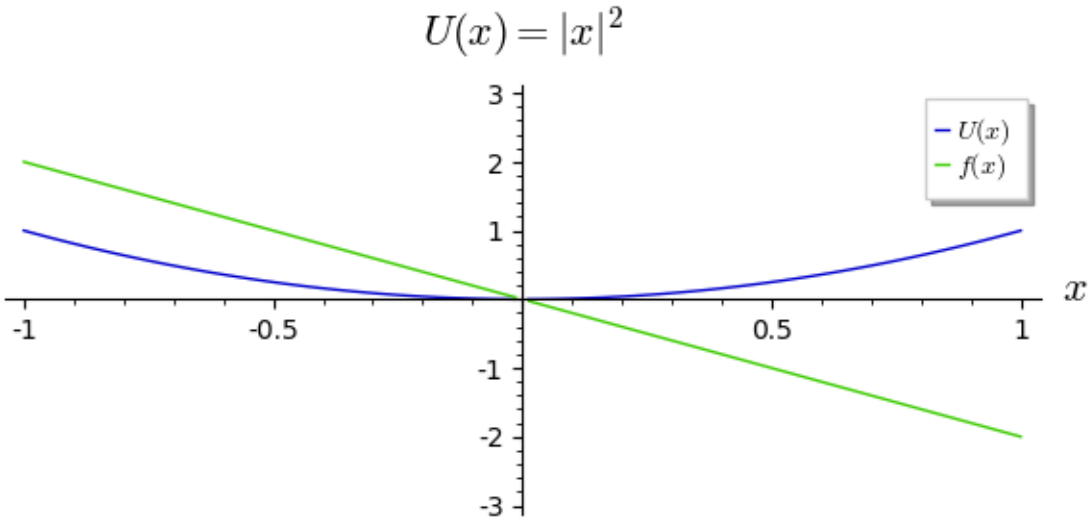
4.6 Numerical integration of equations of motion

Here we will investigate numerically period of motion T and compare with analytical formula. First, let's have a closer look how potential and force behave for different exponents n:

[29]:
```
# @interact
def plot_Uf(n_):
    U(x) = abs(x)^(n_)
    plt = plot( [U(x),-diff( U(x),x)],(x,-1,1),\
               detect_poles='show',ymin=-3,ymax=3,
              legend_label=[r"$U(x)$",'$f(x)$'])
    plt.axes_labels([r"$x$",r"$U(x)=|x|^%s$"%latex(n_)])
    show(plt,figsize=(6,3))
```

[30]:
```
plot_Uf(1),plot_Uf(2)
```

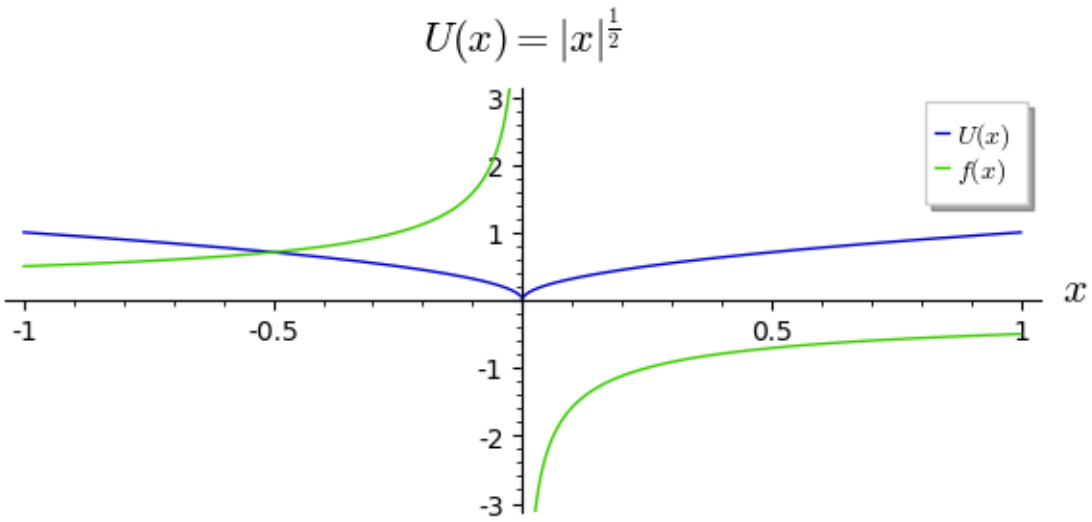




[30]: `(None, None)`

We can see that for $n >= 1$ force and potential is continous. If $n = 1$ then force has finite jump (discontinuity). Both those cases should not be a problem for numerical integration. However for $n < 1$ we have infinite force at $x = 0$:

[31]:
```
plot_Uf(1/2)
```



There is a possibility that if the ODE integrator comes too close, it will blow out!

We can fix this problem by softening the potential singularity by adding small number:

$$|x| \to |x| + \epsilon.$$

[32]:
```
var('x',domain='real')
var('v t')
eps = 1e-6
U(x) = (abs(x)+eps)^(1/2)
showmath( U.diff(x).expand().simplify() )
```

[32]:

$$x \mapsto \frac{x}{2\sqrt{|x| + 1 \times 10^{-06}|x|}}$$

to make sure that Sage will not leave $x/|x|$ unsimplified we can do:

[33]:
```
w0 = SR.wild(0)
w1 = SR.wild(1)
f = -U.diff(x).subs({w0*w1/abs(w1):w0*sign(w1)})
```

[34]:
```
showmath( f(x) )
```

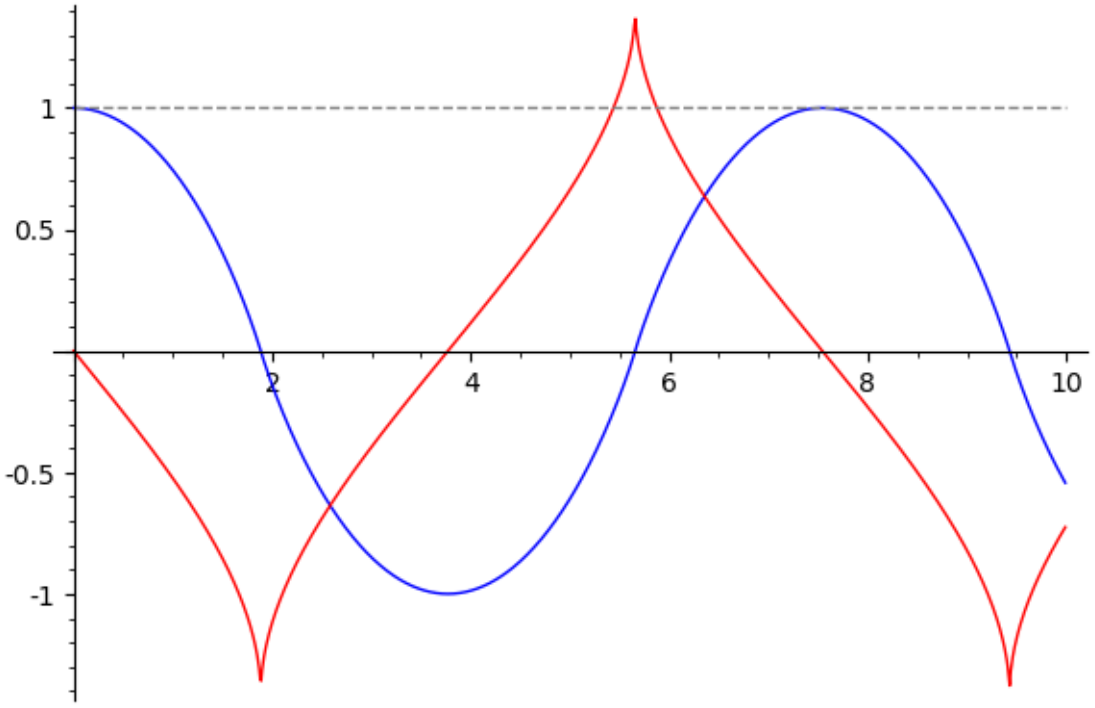
[34]:

$$-\frac{\mathrm{sgn}\,(x)}{2\sqrt{|x| + 1.00000000000000 \times 10^{-6}}}$$

[35]:
```
ode_pot = [v,f(x)]

t_lst = srange(0,10,0.01)
sol = desolve_odeint(ode_pot,[1,.0],t_lst,[x,v])
```

[36]:
```
p = line(zip(t_lst, sol[:,0])) + line(zip(t_lst, sol[:,1]), color='red')
p.axes_labels(['$t$','$x(t),v(t)$'])
p + plot(1,(x,0,10),linestyle='dashed',color='gray')
```

[36]:



We can from the trajectory computer numerically the period. For this purpose one might need an interpolation of numerical table returned by `desolve_odeint`:

[37]:
```
import numpy as np
def find_period(x,t):
    zero_list=[]
    x0 = x[0]
    for i in range(1,len(x)):
        if x[i]*x[i-1] < 0:
            zero_list.append( - (t[i-1]*x[i] - t[i]*x[i-1])/(x[i-1] - x[i]) )
    lnp = np.array(zero_list)
    return 2*(  (lnp-np.roll(lnp,1))[1:] ).mean()
```

[38]:
```
var('x1 x2 t1 t2 a b ')
showmath(  (-b/a).subs(  solve([a*t1+b==x1,a*t2+b==x2],[a,b],␣
 ↪solution_dict=True)[0] ) )
```

[38]:

$$\frac{t_2 x_1 - t_1 x_2}{x_1 - x_2}$$

We find numerically a period of trajectory:

[39]:
```
find_period( sol[:,0],t_lst)
```

[39]:
```
7.54250742200179
```

Exact results for comparison:

[40]:
```
# for n=2 2*pi/sqrt(2)==(2*pi/sqrt(2)).n()
table( [["n","T"]]+[ [n_,((4/sqrt(2)*(beta(1/2, 1/n_)/n_)* E^(1/n_-1/2)).subs({E:
↪1})).n()]
                    for n_ in [1/4,1/3,1/2,2/3,1,2,3,4,5] ] )
```

[40]:

n	T
1/4	10.3439620562146
1/3	9.05096679918781
1/2	7.54247233265651
2/3	6.66432440723755
1	5.65685424949238
2	4.44288293815837
3	3.96596990053623
4	3.70814935460274
5	3.54608570635798

4.7 Using the formula for the period to reproduce the trajectory of movement

We take $m = 1$ and $A = 1$ (then $x = E$), then we can reproduce the trajectory reversing the formula for $T(E)$.

[41]:
```
var('x')
U(A,x) = A*x^2
A = 1/2
E = 1
m = 1.
x1=0.1
showmath( solve(E-U(A,x), x) )
```

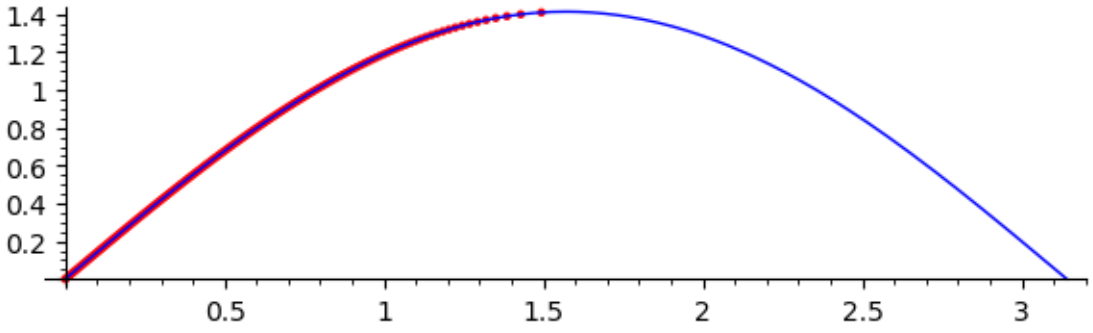
[41]:

$$\left[x = -\sqrt{2}, x = \sqrt{2}\right]$$

[42]:
```
t_lst  = [ (sqrt(m/2.)*integrate( 1/sqrt(E-U(A,x)),(x,0,x1)).n(),x1)  \
          for x1 in srange(0,sqrt(2.)+1e-10,1e-2)]
```

[43]:
```
point(t_lst ,color='red')+\
 plot(sqrt(2)*sin(x),(x,0,pi),figsize=(6,2))
```

[43]:



Interestingly, we if we known the dependence of $T(E)$ then we can calculate exactly the potential!

5 Particle in a multistable potential

5.1 Phase portrait for a one-dimensional system

Equations of motion for a point particle in a 1d potential can be written as a system of two ODEs:

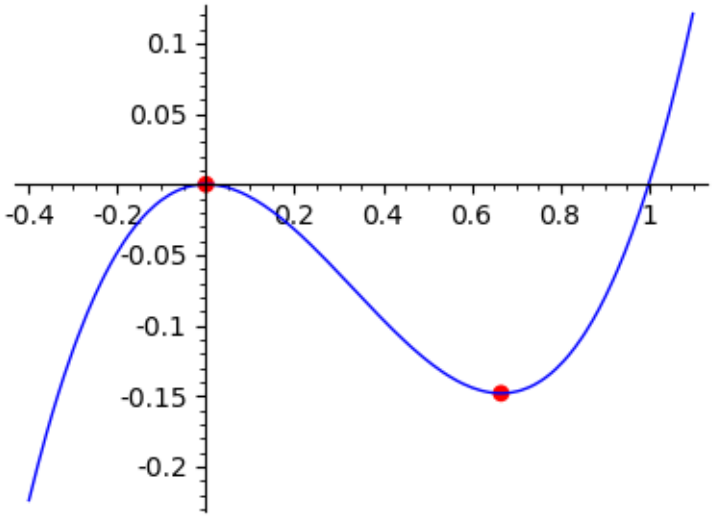
$$\begin{cases} \dot{x} = v \\ \dot{v} = -\frac{1}{m}U'(x) \end{cases}$$

We can draw a phase portrait, i.e. parametric solutions $(x(t), v(t))$ and a vector field defined by the right hand sides in the (x, v) phase space.

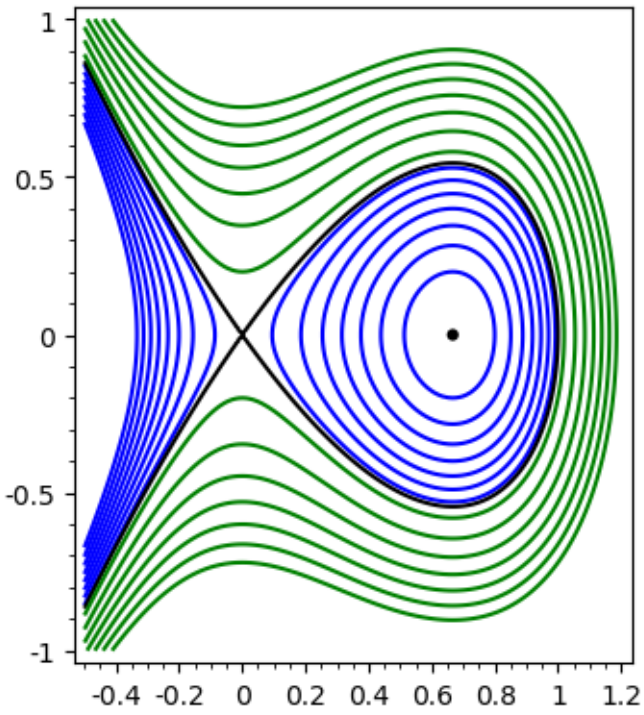
5.2 Example: motion in the $U(x) = x^3 - x^2$ potential

[1]:
```
var('v')
m = 1
U(x) = x^3-x^2
xmax,xmin = sorted([s_.rhs() for s_ in solve(U.diff(x)==0,x)])
Emin = U(xmin)
Etot = 1/2*m*v^2 + U(x)

plot(U(x),(x,-0.4,1.1),figsize=4) +\
 point([xmin,U(xmin)],color='red',size=40)+\
 point([xmax,U(xmax)],color='red',size=40)
```

[1]:

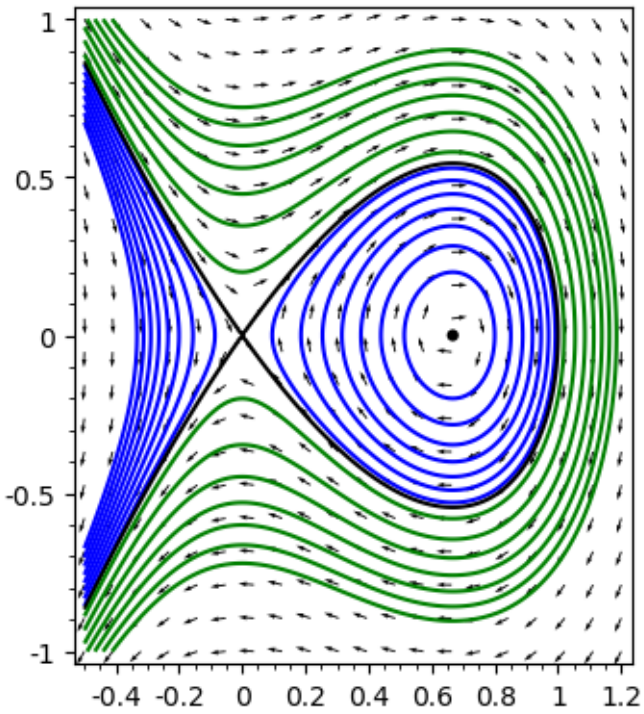


[2]:
```
pkt = point((xmin,0),size=20,color='black')
plt =sum([ implicit_plot(Etot==E0,(x,-1/2,1.2),(v,-1,1),color='blue')\
          for E0 in srange(Emin,0.0,0.02)])
plt +=implicit_plot(Etot==0,(x,-1/2,1.2),(v,-1,1),color='black') +pkt
plt +=sum([ implicit_plot(Etot==E0,(x,-1/2,1.2),(v,-1,1),color='green')\
           for E0 in srange(0.02,-2*Emin,0.04)])
plt.show()
```



[3]:
```
vector_field = vector([v,-U.diff(x)])
plt  + plot_vector_field(vector_field.normalized(),(x,-1/2,1.2),(v,-1,1))
```

[3]:



5.3 Harmonic oscillation limit a one-dimensional system

Consider a conservative one-dimensional system. In this case, the force can always be represented as a potential gradient:

$$f(x) = -\frac{\partial U(x)}{dx}.$$

Consider a certain potential that has a minium. The necessary condition for the minimum function is the disappearance of the first derivative, ie the force in the minimum potential is zero. Let's expand the potential in the Taylor series around the minimum. we have:

$$U(x) = U(x_0) + \underbrace{U'(x_0)(x - x_0)}_{=0} + \frac{1}{2}U''(x_0)(x - x_0)^2 + ...$$

With accuracy as to the constant, we have effective traffic in the potential of the type:

$$U(x) = \frac{1}{2}kx^2,$$

if only x would be close enough to x_0 to cut off Taylor's series, it was justified. The Newton equation for such a motion is as follows:

$$m\ddot{x} = ma = F = -U'(x) = -kx$$

This is the already known equation for the harmonic oscillator, which describes the arrangement with any potential cavity with approximately small vibrations.

[4]:
```
var('x v')
Etot = 1/2*v^2 + U(x)
Elin = 1/2*v^2 + U(xmin)+1/2*U.diff(x,2).subs(x==xmin)*(x-xmin)^2
show(Etot)
show(Elin)
```

```
x^3 + 1/2*v^2 - x^2
```

```
x |--> 1/2*v^2 + 1/9*(3*x - 2)^2 - 4/27
```

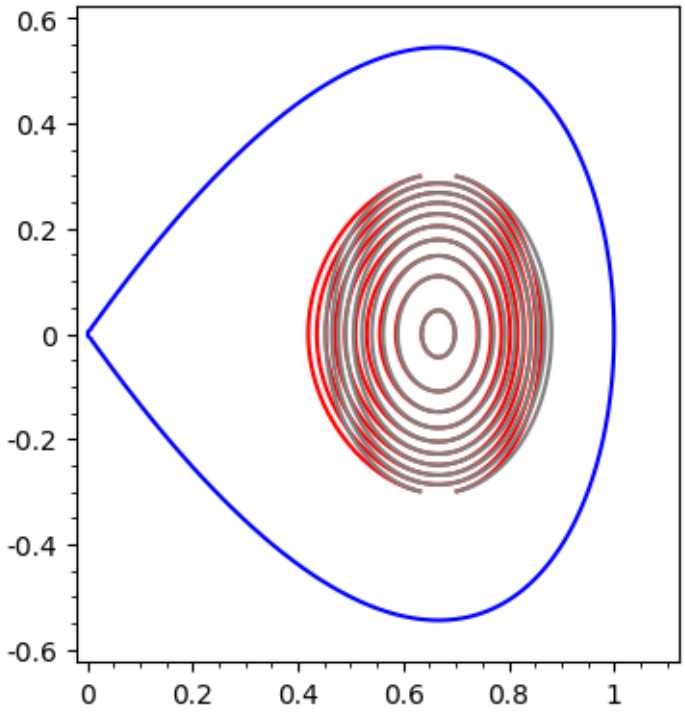
[5]:
```
Emin = Etot(x=xmin,v=0)
Emin
```

[5]:
```
-4/27
```

Let's have a look at the trajectories for small deflections of linearized and exact system. The blue line below is a separatrix - i.e. a solution with $E = 0$

[6]:
```
plt = sum([ implicit_plot(Etot==E0,(x,.4,.91),(v,-.3,.3),color='red') \
          for E0 in srange(Emin+1e-3,-0.1,0.005)])
plt += implicit_plot(Etot==0.00,(x,0,1.1), (v,-.6,.6),color='blue')

plt_lin =sum([ implicit_plot(Elin==E0+1e-3,(x,.4,.91),(v,-.3,.3),color='gray') \
              for E0 in srange(Emin,-0.1,0.005)])
plt+plt_lin
```

[6]:



For larger ones, there is a growing discrepancy:

- for the nonlinear system, above certain energy, there are open trajectories - motion in an linearized system is always an ellipse. The period does not depend on the amplitude.

5.4 Time to reach the hill

Without the loss of generality we can assume that the top of the potential hill is at the beginning of the coordinate system (x, E). Then we examine the limit $E \to 0$ boundary

Near zero, we can approximate the potential by the reverse parabola. Then the time to reach the hill from a certain point (for example $x = 1$) reads:

[7]:
```
var('E')
assume(E>0)
integrate(-1/sqrt(E+x^2),x,1,0)
```

[7]:
```
arcsinh(1/sqrt(E))
```

This result is divergent for $E \to 0$:

[8]:
```
limit( arcsinh(1/x),x=0)
```

[8]:
```
Infinity
```

It means that time needed to climb a hill with *just* enough kinetic energy is **infinite**. It is valid only for potential hills which have zero derivative at the top. On the other hand for potential barriers which do not have this property, for example, $U(x) = -|x|$, the particle can reach the top with just enought energy in finite time.

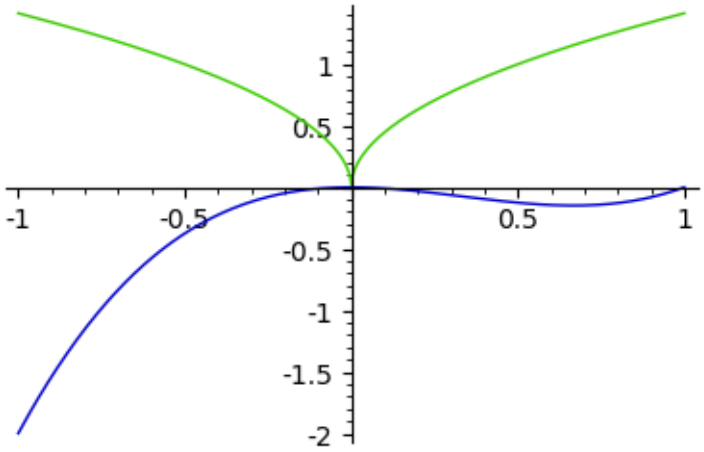
Let analyze it:

[9]:
```
U1(x) = -abs(x)
E0 = 0
```

we can plot velocity and potential:

[10]:
```
plot([U(x), sqrt(2*(E0 - U1(x)))],(x,-1,1),figsize=4)
```

[10]:



the time of travel from $x = -1$ to $x = 0$ is given by:

$$t = \sqrt{\frac{m}{2}} \int_{-x1}^{x1} \frac{dx}{(\sqrt{(E - U(x)})}$$

which in this case is:

[11]:
```
sqrt(m/2.)*integrate(1/sqrt((E0- U1(x))),x,-1,0).n()
```

[11]:
```
1.41421356237310
```

In the case of potentials which behave like $|x|^{\alpha}$, for $\alpha > 1$ we can calculate time of travel if we particle total energy is by dE larger than potential barrier.

[12]:
```
dE = 0.01
```

[13]:
```
E0 = U(xmin)+dE
E0
```

[13]:
```
-0.138148148148148
```

[14]:
```
_, x1,x2 = sorted( [s_.rhs().n().real() for s_ in solve(U(x)==E0,x)])
x1,x2
```

[14]:
```
(0.560919215938882, 0.762206802325432)
```

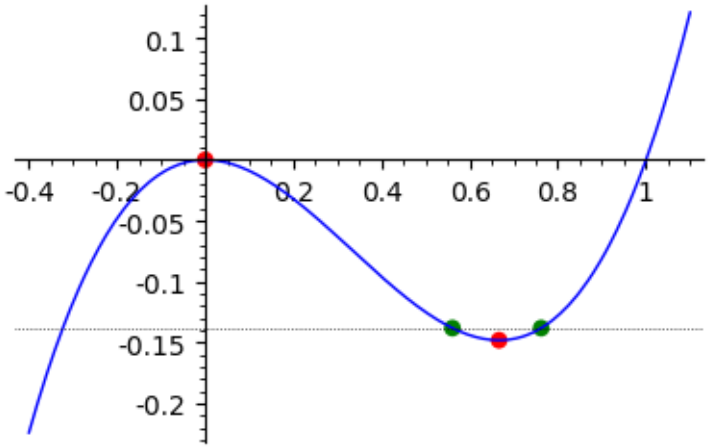
[15]:
```
period = 2*sqrt(m/2.)*\
  integral_numerical(sqrt(E-U(x)).subs(E==E0) , x1,x2, algorithm='qags')[0]
period
```

[15]:
```
0.0223211406255025
```

[16]:
```
U(x) = x^3-x^2
xmax,xmin = sorted([s_.rhs() for s_ in solve(U.diff(x)==0,x)])
print(xmin)
plot(U(x),(x,-0.4,1.1),figsize=4,gridlines=[None,[E0]])+\
 point([xmin,U(xmin)],color='red',size=40)+\
 point([xmax,U(xmax)],color='red',size=40)+\
 point([x1,U(x1)],color='green',size=40)+\
 point([x2,U(x2)],color='green',size=40)
```

```
2/3
```

[16]:



[17]:
```
integral_numerical( 1/sqrt(E0-U(x)) , x1,x2, algorithm='qag')
```

[17]:
```
(3.1721596145817488, 2.320464914499202e-06)
```

[18]:
```
def T(E0):
    m = 1

    _, x1,x2 = sorted( [s_.rhs().n().real() for s_ in solve(U(x)==E0,x)])

    integral, error = \
      integral_numerical(1/sqrt(E0-U(x)), x1,x2, algorithm='qags')
    # print(":::",x1,x2,error)
    m = 1
    period = 2*sqrt(m/2.) * integral
    return period
```

[19]:
```
period_num = T(U(xmin)+dE)
period_num
```

[19]:
```
4.48611131059499
```

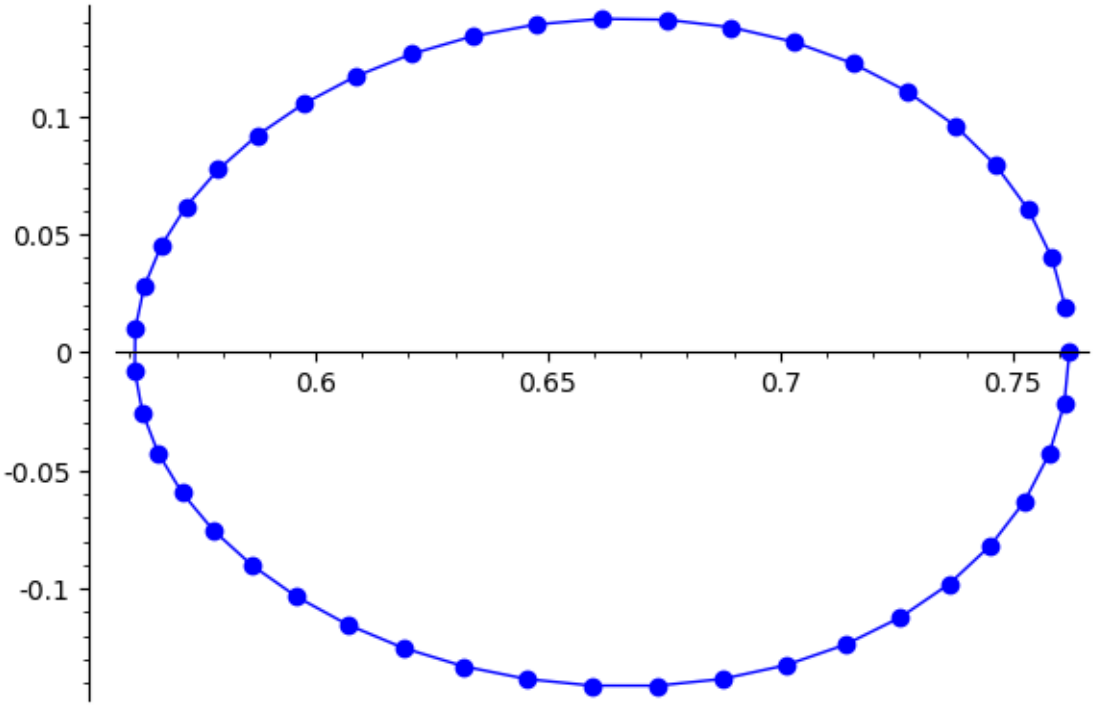
[20]:
```
omega = sqrt(U(x).diff(x,2).subs(x==xmin.n()))
period_harm = 2*pi.n()/omega
period_harm
```

[20]:
```
4.44288293815837
```

[21]:
```
t_lst = srange(0, period_num, 0.01, include_endpoint=True)
sol = desolve_odeint([v,-U.diff(x)],[x2,.0],t_lst,[x,v])
```

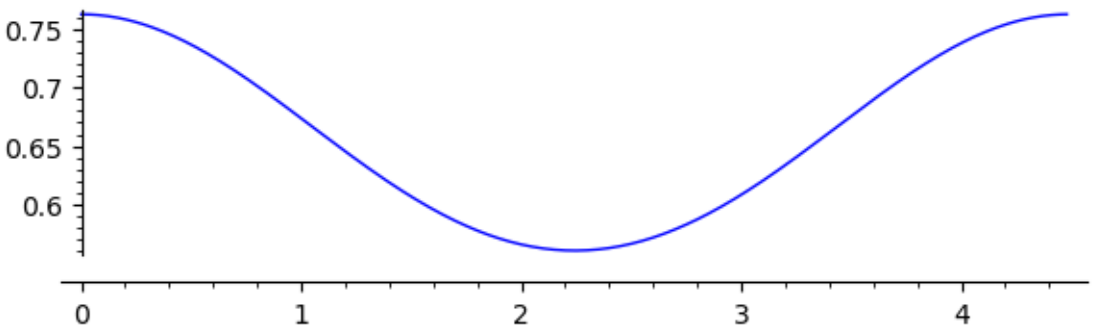
[22]:
```
line(sol[::10,:],marker='o')
```

[22]:



[23]:
```
line(zip(t_lst,sol[:,0]),figsize=(6,2))
```

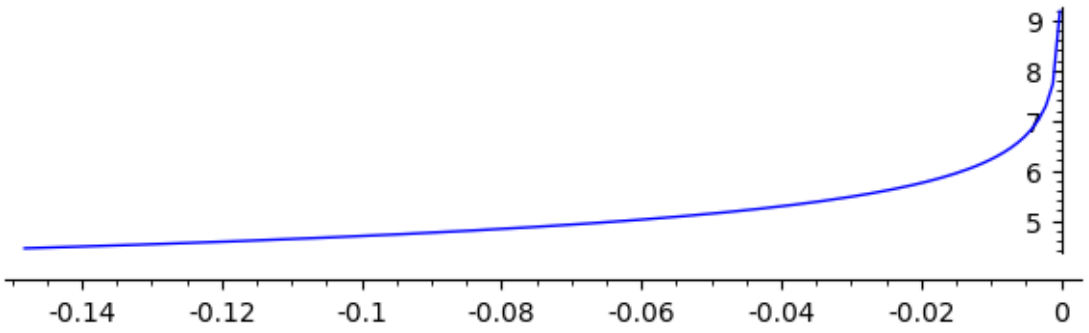
[23]:



[24]:
```
TonE = [(E_,T(E_)) for E_ in srange(U(xmin)+1e-6,-1e-5,0.001)]
```

[25]:
```
line(TonE, figsize=(6,2))
```

[25]:



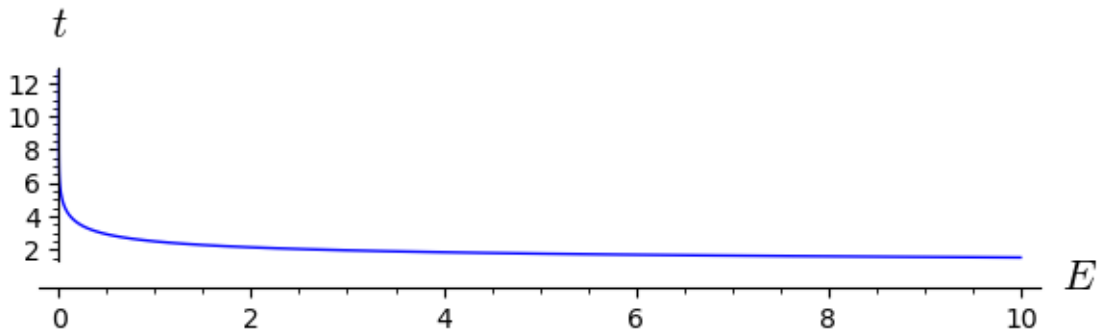
[26]:
```
def t_hill(E0):
    m = 1

    x2, = [s_.rhs().n().real() for s_ in solve(U(x)==E0,x) if s_.rhs().n().imag().
 ↪abs()<1e-6]

    integral, error = \
      integral_numerical(1/sqrt(E0-U(x)), 0,x2, algorithm='qags')
    m = 1
    period = 2*sqrt(m/2.) * integral
    return period
```

[27]:
```
t_hill(9.1)
```

[27]:
```
1.53679467633445
```

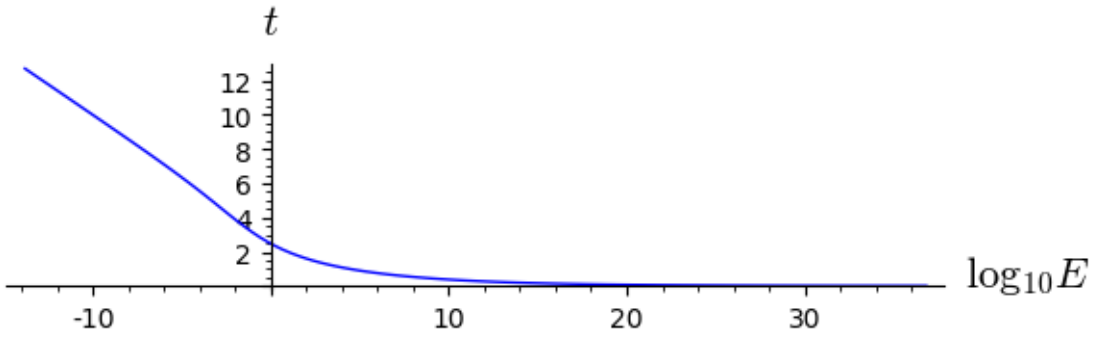
[28]:
```
import numpy as np
t_E = [(E_,t_hill(E_)) for E_ in np.logspace(-6,1,120)]
line(t_E,axes_labels=['$E$','$t$'], figsize=(6,2))
```

[28]:



[29]:
```
t_E = [(log(E_),t_hill(E_)) for E_ in np.logspace(-6,16,120)]
line(t_E,axes_labels=['$\log_{10} E$','$t$'], figsize=(6,2))
```

[29]:



5.5 Excercise 1

Analyze numerically or analytically how the period from the deflection in the nonlinear system depends

5.6 Excercise 2

Examine in a similar way the system corresponding to the movement in the $U(x) = -\cos(x)$ potential - this is a physical pendulum.

6 d'Alembert with computer algebra system

6.1 d'Alembert principle

d'Alembert principle states that the sum of the differences between the forces acting on a system of mass particles and the time derivatives of the momenta of the system itself projected onto any virtual displacement consistent with the constraints of the system is zero.

It can be written as following,

$$\sum_i (\mathbf{F}_i - m_i \ddot{\mathbf{x}}_i) \cdot \delta \mathbf{r}_i = 0, \tag{1}$$

where:

- i enumerates particles,
- $\mathbf{F_i}$ $\ddot{\mathbf{x}}_\mathbf{i}$ are forces and accelerations of i-th particle,
- $\delta \mathbf{r}_i$ is virtual displacement of i-th particle.

We consider N particles in 3 dimensional physical space, subjected to p holonomous constraints in the form:

$$f_k(x, t) = 0 \quad k = 1, 2, ..., p.$$

The virtual displacements of each coordinates: δx_j, can be arbitrary numbers fulfilling:

$$\sum_{j=1}^{3N} \frac{\partial f_k}{\partial x_j} \delta x_j = 0, \quad k = 1, 2, ..., p. \tag{2}$$

This is a homogenoues system of p linear equation for $3N$ δx_j, thus p displacements can be expressed by remaining $3N - p$ which are arbitrary.

We can substitute this solution to the original d'Alembert equation (1) and we will obtain $3N - p$ second order differential equations. Together with p constraints (2) they can allow to determine evolution of all variables. Let us note that this is system of differential-algebraic equations. It can be solved for example by differentiating algebraic equations and solvin the system od ODEs in classical manner.

Better possibility, used in most of texbook problems, is to find equation of motion in $3N - p$ independent generalized coordinates which are compliant with the constraints. Then we need to transform d'Alembert principle (1) into those coordinates and it leads to a system of $3N - p$ ODEs.

6.2 How to use CAS with d'Alembert principle.

One of the problems which prohibit direct use of CAS is the need to treat symbols as independent variables and as functions of time, depending on the context. One possible solution to this is to define for each symbolic variable the correspoding Sage symbolic function and variables which would represent first and second derivative.

- coordinate - small letter: `a`
- its time derivatives as independent symbols: $\dot{a}$ i $\ddot{a}$ - `ad` i `add`
- explicit function of time $a(t)$: `A`
- virtual displacement δa - `da`

6.2.1 Example - step by step

Let a denote some generalized coordinate in out dynamical system:

[1]:
```
var('t')
var('a')
```

[1]:
```
a
```

We add symbols representing its derivatives with nice LaTeX representation:

[2]:
```
var('ad',latex_name=r'\dot a')
var('add',latex_name=r'\ddot a')
show([a,ad,add])
```

```
[a, ad, add]
```

We define with capital `A` function of time.

[3]:
```
A = function('A')(t)
show(A)
```

```
A(t)
```

Now, we can do following:

[4]:
```
show(1+A.diff())
show ( (1+A.diff()).subs({A.diff():ad}) )
```

```
diff(A(t), t) + 1
```

```
ad + 1
```

Let us calculate second time derivative of (1+a)^3:

[5]:
```
expr = (1+a)^3
```

we change variables to explicit function of time:

[6]:
```
expr  = expr.subs({a:A})
show(expr)
```

```
(A(t) + 1)^3
```

and calculate derivative:

[7]:
```
expr = expr.diff(t,2)
show(expr)
```

```
6*(A(t) + 1)*diff(A(t), t)^2 + 3*(A(t) + 1)^2*diff(A(t), t, t)
```

we can now convert to the form containing symbols: `ad` and `add`

[8]:
```
expr  = expr.subs({A:a,A.diff():ad,A.diff(2):add})
show(expr)
```

```
6*(a + 1)*ad^2 + 3*(a + 1)^2*add
```

And calculate derivative ove $\dot{a}$:

[9]:
```
expr = expr.diff(ad)
show(expr)
```

```
12*(a + 1)*ad
```

6.2.2 Automatic definitions

We can now easily for each variable, construct two symbols representing time derivatives and explicit time function and also dictionaries for converting from one form to the another.

Let us define list of variables and their LaTeX representations in a list of pairs: `xy_wsp`. Then we cen write:

[10]:
```
# %load cas_utils.sage
from IPython.display import Math
def showmath(expr):
    return Math(latex(expr))

def sanitize_namelist(lst):
    new_lst = []
    for x_ in lst:
```

```
        if isinstance(  x_ , str):
            v, lv = x_, x_
        elif isinstance( x_, tuple):
            v, lv = x_
        else:
            raise ValueError,'Wrong name: ' + str(type(x_))
        new_lst.append((v, lv))
    return new_lst

def make_symbols(xy_names, uv_names=[], verbose=False):
    """
    Make a variables for CAS manipulation of
    expressions, including derivatives and pretty typing.

    params:

    A list of coordinated with their latex_names, must be lower case

    - ``xy_coords = [('x','x'),... ]``
    - ``uv_coords = [('phi',r'\varphi')]``

    For example for variable ``phi``:
     - a function ``Phi(t)``
     - variables: ``dphi``, ``phid`` and ``phidd``
     will be injected into global namespace.

    To dictionaries will be returned

    - to_fun - for substitution variables to functions,
                  and their 1st and 2dn derivative
    - to_var - for substitution functions and their
                1st and 2dn derivativeto variables
    """
    xy_names = sanitize_namelist(xy_names)
    uv_names = sanitize_namelist(uv_names)

    var('t',domain='real')

    for v,lv in uv_names + xy_names:

        var("%s"%v,latex_name=r'%s'%lv)
        globals()[v.capitalize()] = function(v.capitalize())(t)
        var("%sdd"%v, latex_name=r'\ddot %s'%lv)
        var("%sd"%v, latex_name=r'\dot %s'%lv)
        var("d%s"%v, latex_name=r'\delta %s'%lv)
        print v, " :: has been  processed"

    uv = [globals()[v] for v,lv in uv_names]
    xy = [globals()[v] for v,lv in xy_names]
```

```
    to_fun = dict()

    for v,lv in uv_names + xy_names:
        to_fun[globals()[v]] = globals()[v.capitalize()]
        to_fun[globals()[v+"d"]] = globals()[v.capitalize()].diff()
        to_fun[globals()[v+"dd"]] = globals()[v.capitalize()].diff(2)

    to_var = dict((v,k) for k,v in to_fun.items())
    if verbose:
        print 'we have dictionaries:'
        show( table([ [v,r'$\iff$',k] for k,v in to_var.iteritems()]) )
    return to_fun, to_var


def transform_virtual_displacements(xy_names, uv_names, verbose=False,␣
 ↪suffix='_polar'):
    """
    Transform vistual displacements using
    chain rule of differentiation.



    """
    xy_names = sanitize_namelist(xy_names)
    uv_names = sanitize_namelist(uv_names)

    uv = [globals()[v] for v,lv in uv_names]
    xy = [globals()[v] for v,lv in xy_names]

    new_variations = []
    for w in xy:
        globals()['d'+repr(w)+suffix] = \
         sum([w.subs(x2u).diff(w2)*globals()['d'+repr(w2)]\
                                              for w2 in uv])
        new_variations.append( globals()['d'+repr(w)+suffix] )
        if verbose:
            print 'd'+repr(w)+suffix+' : is added to namespace'
            show([globals()['d'+repr(w)],globals()['d'+repr(w)+suffix]])


    return new_variations
```

[11]:
```
var('t')
xy_wsp = [('x','x'),('y','y')]
to_fun, to_var = make_symbols(xy_wsp)
```

```
x  :: has been  processed
y  :: has been  processed
```

[12]:
```
show(to_var)
```

```
{X(t): x,
 Y(t): y,
 diff(X(t), t, t): xdd,
 diff(X(t), t): xd,
 diff(Y(t), t): yd,
 diff(Y(t), t, t): ydd}
```

[13]:
```
show(to_fun)
```

```
{ydd: diff(Y(t), t, t),
 xdd: diff(X(t), t, t),
 x: X(t),
 yd: diff(Y(t), t),
 xd: diff(X(t), t),
 y: Y(t)}
```

Let's experiment with examples:

[14]:
```
show( (1+x^2*y) )
show( (1+x^2*y).subs(to_fun))
show( (1+x^2*y).subs(to_fun).diff(t,2) )
show( (1+x^2*y).subs(to_fun).diff(t,2).subs(to_var) )
```

```
x^2*y + 1
```

```
X(t)^2*Y(t) + 1
```

```
2*Y(t)*diff(X(t), t)^2 + 2*X(t)*Y(t)*diff(X(t), t, t) + 4*X(t)*diff(X(t), t)*diff(Y(t), t) +
```

```
2*xd^2*y + 2*x*xdd*y + 4*x*xd*yd + x^2*ydd
```

[15]:
```
show( (1+x^2*y).subs(to_fun).diff(t,2).subs(to_var).diff(xd).diff(x) )
```

```
4*yd
```

[16]:
```
x.subs(to_fun).diff().subs(to_var).subs(to_fun)
```

[16]:
```
diff(X(t), t)
```

6.3 Example: mathematical pendulum in cartesian coordinates in 2d

We consider in 2d a point with mass m in Earth gravitation subjected to constraints: $x^2 + y^2 - l^2 = 0$. l is a lenght of the pendulum.

Position of the mass is (x, y), thus:

[17]:
```
var('t')
var('l g')
xy_wsp = [('x','x'),('y','y')]

for v,lv in xy_wsp:
    var("%s"%v,latex_name=r'%s'%lv)
    vars()[v.capitalize()] = function(v.capitalize())(t)
    var("%sdd"%v,latex_name=r'\ddot %s'%lv)
    var("%sd"%v,latex_name=r'\dot %s'%lv)
    var("d%s"%v,latex_name=r'\delta %s'%lv)

xy = [vars()[v] for v,lv in xy_wsp]
dxy = [vars()['d'+repr(zm)] for zm in xy]


to_fun=dict()
for v,lv in xy_wsp:
    to_fun[vars()[v]]=vars()[v.capitalize()]
    to_fun[vars()[v+"d"]]=vars()[v.capitalize()].diff()
    to_fun[vars()[v+"dd"]]=vars()[v.capitalize()].diff(2)
to_var = dict((v,k) for k,v in to_fun.items())
```

[18]:
```
show(xy),show(dxy),
```

```
[x, y]


[dx, dy]
```

[18]:
```
(None, None)
```

[19]:
```
var('t')
var('l g')
xy_wsp = ['x','y']
to_fun, to_var = make_symbols(xy_wsp)
```

```
x  :: has been  processed
y  :: has been  processed
```

After this we have in our namespace following variables:

[20]:
```
show([x,xd,xdd,dx])
```

```
[x, xd, xdd, dx]
```

[21]:
```
xy = [vars()[v] for v in xy_wsp]
dxy = [vars()['d'+repr(zm)] for zm in xy]
show(xy)
```

```
[x, y]
```

[22]:
```
show(dxy)
```

```
[dx, dy]
```

Having constraints, one can obtain its differential form:

$$\frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial y}\delta y = 0$$

[23]:
```
f = x^2+y^2-l^2
constr =sum([dz*f.diff(z) for z,dz in zip(xy,dxy)])
show( constr)
```

```
2*dx*x + 2*dy*y
```

d'Alembert principle reads:

[24]:
```
dAlemb = (X.diff(t,2))*dx + (Y.diff(t,2)+g)*dy
show(dAlemb.subs(to_var))
```

```
dy*(g + ydd) + dx*xdd
```

First equation we obtain by substituting e.g. δx from the differential contraints equation to d'Alembert principle:

[25]:
```
eq1 = (dAlemb.subs(constr.solve(dx)[0])).expand().coefficient(dy).subs(to_var)
show(eq1)
```

```
g - xdd*y/x + ydd
```

The second equation can be obtained by differentiating contraints over time two times:

[26]:
```
eq2 = f.subs(to_fun).diff(t,2).subs(to_var)
show(eq2)
```

```
2*xd^2 + 2*x*xdd + 2*yd^2 + 2*y*ydd
```

We have to solve for $\ddot{x}$ i $\ddot{y}$ and we get equation of motion:

[27]:
```
sol = solve( [eq1,eq2],[xdd,ydd])
show( sol[0] )
```

```
[xdd == (g*x*y - (xd^2 + yd^2)*x)/(x^2 + y^2),
 ydd == -(g*x^2 + (xd^2 + yd^2)*y)/(x^2 + y^2)]
```

We can easily solve it with `desolve_odeint` numerically. Interestingly, the lenght of the pendulum must be taken into the account inside initial conditions, as l was removed from the above system by diferentiation.

Having access to right hand sides:

[28]:
```
sol[0][0].rhs()
```

[28]:
```
(g*x*y - (xd^2 + yd^2)*x)/(x^2 + y^2)
```

[29]:
```
sol[0][1].rhs()
```

[29]:
```
-(g*x^2 + (xd^2 + yd^2)*y)/(x^2 + y^2)
```

We solve the system of four first order ODEs (we treat x and velocity: $\dot{x}$ as independent variables):

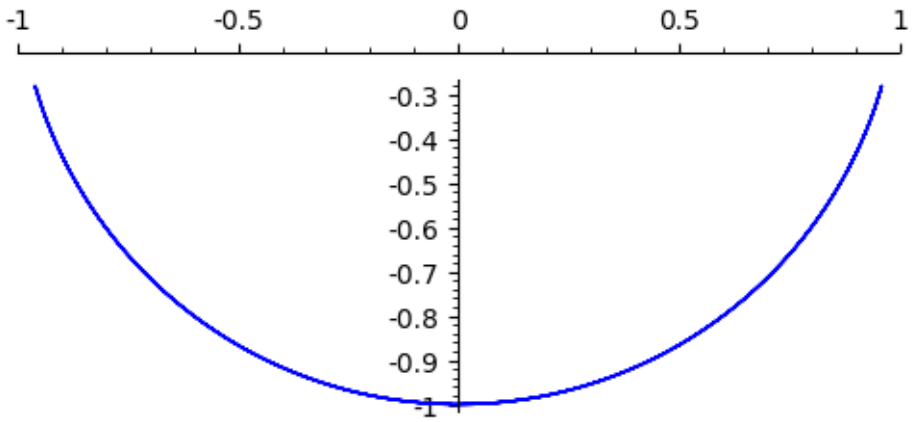
$$\frac{dx}{dt} = \dot{x} \tag{11}$$

$$\frac{dy}{dt} = \dot{y} \tag{12}$$

$$\frac{d\dot{x}}{dt} = \frac{gxy - \left(\dot{x}^2 + \dot{y}^2\right)x}{x^2 + y^2} \tag{13}$$

$$\frac{d\dot{y}}{dt} = -\frac{gx^2 + \left(\dot{x}^2 + \dot{y}^2\right)y}{x^2 + y^2} \tag{14}$$

[30]:
```
ode=[xd,yd,sol[0][0].rhs().subs({g:1}),sol[0][1].rhs().subs({g:1})]
times = srange(0,14,0.01)
numsol=desolve_odeint(ode,[0,-1,1.2,0],times,[x,y,xd,yd])
p=line(zip(numsol[:,0],numsol[:,1]),figsize=5,aspect_ratio=1)
p.show()
```



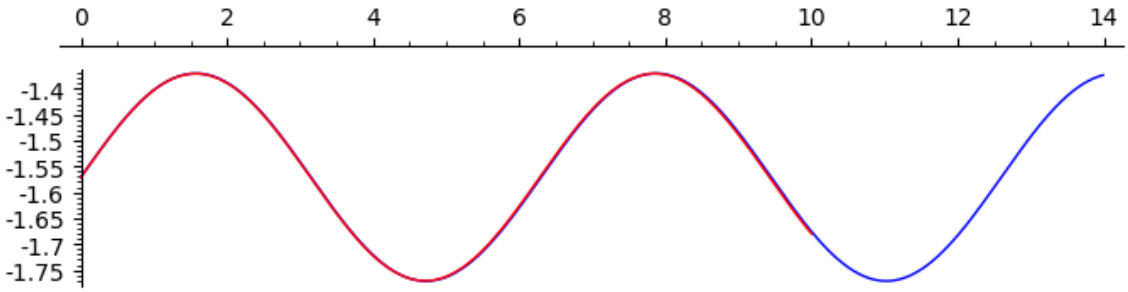
We can compare this numerical solution with small amplitude approximation. Suppose that the pendulum starts at its lowest position, $\phi = \arctan(y/x) = -\pi/2$ with linear velocity $\dot{x}(0) = 0.2$. The analytical solution in that case reads:

$$\phi = -\pi/2 + 0.2\sin(\omega_0 t),$$

where $\omega_0 = \sqrt{g/l} = 1$

[31]:
```
times = srange(0,14,0.01)
numsol = desolve_odeint(ode,[0,-1,.2,0],times,[x,y,xd,yd])

import numpy as np

line(zip( times,np.arctan2(numsol[:,1],numsol[:,0]) ),figsize=(7,2))+\
 plot(0.2*sin(t)-pi/2,(t,0,10),color='red')
```

[31]:



We can also check if contraints, which are the lenght of the pendulum, are fulfilled during the simulation:

[32]:
```
print "initial l:",numsol[0,0]**2+numsol[0,1]**2,
print "final l:",numsol[-1,0]**2+numsol[-1,1]**2
```

```
initial l: 1.0 final l: 0.9999999900787478
```

6.3.1 Solution in generalized coordinaes.

Clearly, the derived system od DAE is not the best approach to describe mathematical pendulum. The better idea is to use coordinates which fulfill automatically the constraint. In the case of mathematical pendulum one can use the angle ϕ.

We will need two sets of coordinates: (x, y) and ϕ:

[33]:
```
var('x y t')
var('l g')

xy_wsp = [('x','x'),('y','y')]
uv_wsp = [('phi','\phi')]

to_fun, to_var = make_symbols(xy_wsp+uv_wsp)

x2u = {x:l*cos(phi),y:l*sin(phi)}
```

```
x  :: has been  processed
y  :: has been  processed
phi  :: has been  processed
```

[34]:
```
uv = [vars()[v] for v,lv in uv_wsp]
xy = [vars()[v] for v,lv in xy_wsp]
```

We have to express virtual displacements in new coordinates:

$$\delta x = \frac{\partial x(r,\phi)}{\partial \phi}\delta\phi$$

$$\delta y = \frac{\partial y(r,\phi)}{\partial \phi}\delta\phi$$

Despite the fact that we have only one element on `uv`, i.e. one new coordinate, we will use general formula below:

[35]:
```
for w in xy:
    vars()['d'+repr(w)+'_polar']=\
```

```
       sum([w.subs(x2u).diff(w2)*vars()['d'+repr(w2)] for w2 in uv])
show([dx_polar,dy_polar])
```

```
[-dphi*l*sin(phi), dphi*l*cos(phi)]
```

d'Alembert principle in new coordinates reads:

[36]:
```
dAlemb = (x.subs(x2u).subs(to_fun).diff(t,2))*dx_polar + \
         (y.subs(x2u).subs(to_fun).diff(t,2)+g)*dy_polar
dAlemb = dAlemb.subs(to_var)
```

[37]:
```
show(dAlemb)
```

```
-(l*phid^2*sin(phi) - l*phidd*cos(phi) - g)*dphi*l*cos(phi) + (l*phid^2*cos(phi) + l*phidd*s
```

Above expression is zero when coefficient at $\delta\phi$ is zero:

[38]:
```
for v in uv:
    show(dAlemb.expand().coefficient(vars()['d'+repr(v)]).trig_simplify())
```

```
l^2*phidd + g*l*cos(phi)
```

We finally arrive at known and expected equation:

[39]:
```
show( dAlemb.expand().coefficient(dphi).trig_simplify().solve(phidd) )
```

```
[phidd == -g*cos(phi)/l]
```

Stable point is $\phi = -\frac{\pi}{2}$, we can expand in this point the right hand side and obtain harmonic oscilator in ϕ:

[40]:
```
taylor(-g/l*cos(phi),phi,-pi/2,1).show()
```

```
-1/2*(pi + 2*phi)*g/l
```

one can redefine ϕ, so it is zero at lowest point, and we recognize the classical formula:

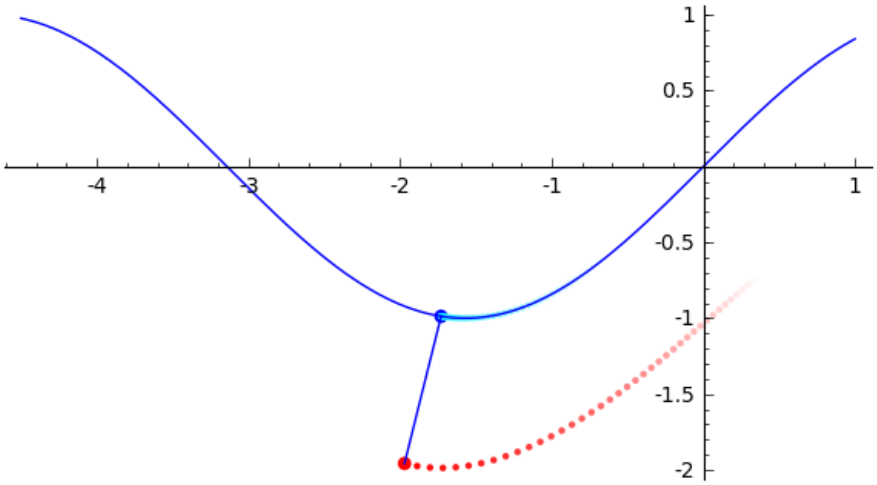
[41]:
```
taylor(-g/l*cos(phi),phi,-pi/2,1).subs({phi:phi-pi/2}).expand().show()
```

```
-g*phi/l
```

7 Pendulum on $\sin(x)$.

7.1 System definition

Consider a pendulum with mass m_2 hanging from a rod of length l. The support point has a mass m_1, and can move without friction along a curve given by formula $y = f(x)$



Pendulum

We will need some helpers for the algebra:

[1]:
```
load("cas_utils.sage")
```

[2]:
```
var('t')
var('l g m1 m2')
xy_wsp = [('x1','x_1'),('y1','y_1'),('x2','x_2'),('y2','y_2')]

uv_wsp = [('phi','\phi'),('x','x')]

to_fun, to_var = make_symbols(xy_wsp + uv_wsp)

uv = [vars()[v] for v,lv in uv_wsp]
xy = [vars()[v] for v,lv in xy_wsp]
```

```
x1  :: has been  processed
y1  :: has been  processed
x2  :: has been  processed
y2  :: has been  processed
phi  :: has been  processed
x  :: has been  processed
```

We introduce generalized coordinates compliant with contraints: φ, and x.

[3]:
```
f(x) = sin(x)
x2u = {x1:x,x2:x+l*sin(phi),y2:-l*cos(phi)+sin(x),y1:f(x)}
showmath(x2u)
```

[3]:

$$\{x_1 : x, y_1 : \sin(x), x_2 : l\sin(\phi) + x, y_2 : -l\cos(\phi) + \sin(x)\}$$

We express virtual displacement:$\delta x_1, \ldots$ as function of virtual displacements of new coordinates: $\delta x, \delta\phi$.

[4]:
```
for w in xy:
    vars()['d'+repr(w)+'_polar']=sum([w.subs(x2u).diff(w2)*vars()['d'+repr(w2)]␣
 ↪for w2 in uv])
    showmath([vars()['d'+repr(w)],vars()['d'+repr(w)+'_polar']])
```

Now we can write d'Alembert principle:

$$\sum_i (\mathbf{F}_i - m_i\mathbf{a}_i)\cdot\delta\mathbf{r}_i = 0,$$

[5]:
```
dAlemb = (m1*x1.subs(x2u).subs(to_fun).diff(t,2)  )*dx1_polar + \
         (m1*y1.subs(x2u).subs(to_fun).diff(t,2)+m1*g)*dy1_polar + \
         (m2*x2.subs(x2u).subs(to_fun).diff(t,2)  )*dx2_polar + \
         (m2*y2.subs(x2u).subs(to_fun).diff(t,2)+m2*g)*dy2_polar
dAlemb = dAlemb.subs(to_var)
showmath(dAlemb.collect(dx).collect(dphi))
```

[5]:

$$\left(l^2 m_2\ddot{\phi}\cos(\phi)^2 + l^2 m_2\ddot{\phi}\sin(\phi)^2 - l m_2\dot{x}^2\sin(\phi)\sin(x) + l m_2\ddot{x}\cos(x)\sin(\phi) + l m_2\ddot{x}\cos(\phi) + g l m_2\sin(\phi)\right)\delta q$$

and derive equations of motion in new coordintes:

[6]:
```
r1 = dAlemb.coefficient(dx)
r2 = dAlemb.coefficient(dphi)
w1,w2 =  solve([r1,r2],[xdd,phidd])[0]
showmath(w1.trig_simplify())
```

[6]:

$$\ddot{x} = -\frac{l m_2\dot{\phi}^2\sin(\phi) + g m_2\cos(\phi)\sin(\phi) - \left(m_2\cos(\phi)\sin(\phi) - \left(m_2\cos(\phi)^2 + m_1\right)\cos(x)\right)\dot{x}^2\sin(x) - \left(lm_2\right.}{2\,m_2\cos(\phi)\cos(x)\sin(\phi) + \left(m_2\cos(\phi)^2 + m_1\right)\sin(x)^2 - 2\,m_1 -}$$

[7]:
```
showmath(w2)
```

[7]:

$$\ddot{\phi} = -\frac{l m_2\dot{\phi}^2\cos(\phi)\cos(x)^2\sin(\phi) - l m_2\dot{\phi}^2\cos(\phi)\sin(\phi) - ((m_1+m_2)\cos(\phi)\cos(x) - (m_1+m_2)\sin(\phi))\dot{x}^2}{2\,l m_2\cos(\phi)\cos(x)\sin(\phi) - l m_1\cos(\phi)^2 - \left(lm_1\sin(\phi)^2 +}$$

Special case $m_1 \to \infty, x = -\frac{\pi}{2}$

[8]:
```
showmath( limit(w1.rhs(),m1=oo).subs({xdd:0,xd:0,x:-pi/2})  )
```
[8]:

$$0$$

[9]:
```
showmath( limit(w2.rhs(),m1=oo).subs({xdd:0,xd:0,x:-pi/2}).trig_reduce() )
```
[9]:

$$-\frac{g \sin(\phi)}{l}$$

We obtain mathematical pendulum if

[10]:
```
showmath( limit(w1.rhs(),m1=0).trig_reduce() )
```
[10]:

$$-\frac{2\,l\dot{\phi}^2}{\cos(\phi+x)+\cos(-\phi+x)-2\,\sin(\phi)}+\frac{\dot{x}^2\sin(\phi+x)}{\cos(\phi+x)+\cos(-\phi+x)-2\,\sin(\phi)}+\frac{\dot{x}^2\sin(-\phi+x}{\cos(\phi+x)+\cos(-\phi+x}$$

[11]:
```
showmath( limit(w2.rhs(),m1=0).trig_reduce() )
```
[11]:

$$\frac{2\,\dot{\phi}^2\cos(\phi)}{\cos(\phi+x)+\cos(-\phi+x)-2\,\sin(\phi)}+\frac{\dot{\phi}^2\sin(\phi+x)}{\cos(\phi+x)+\cos(-\phi+x)-2\,\sin(\phi)}-\frac{\dot{\phi}^2\sin(-\phi+x)}{\cos(\phi+x)+\cos(-\phi+x)}$$

7.2 Numerical analysis of the system

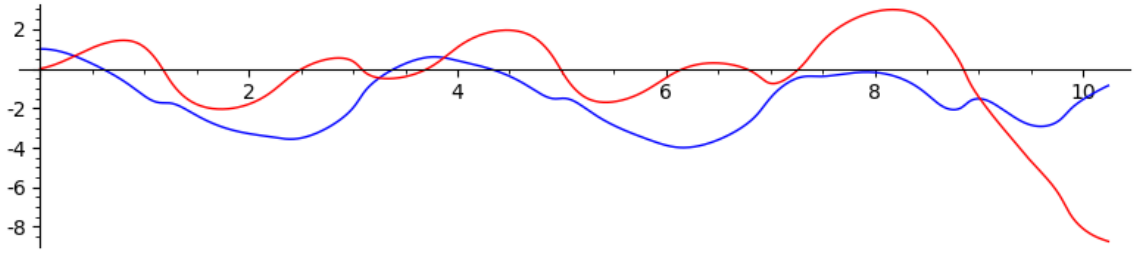
Initial conditions are four numbers: $x, \phi, \dot{x}, \dot{\phi}$.

[12]:
```
import numpy as np
```

[13]:
```
%%time
pars = {l:1,g:9.81,m1:1.,m2:1}
ode = [xd,phid,w1.rhs().subs(pars),w2.rhs().subs(pars)]
times = srange(0,10.25,0.01)
ics = [1, 0, 0, 1]
sol = desolve_odeint(ode, ics, times, [x,phi,xd,phid])
```

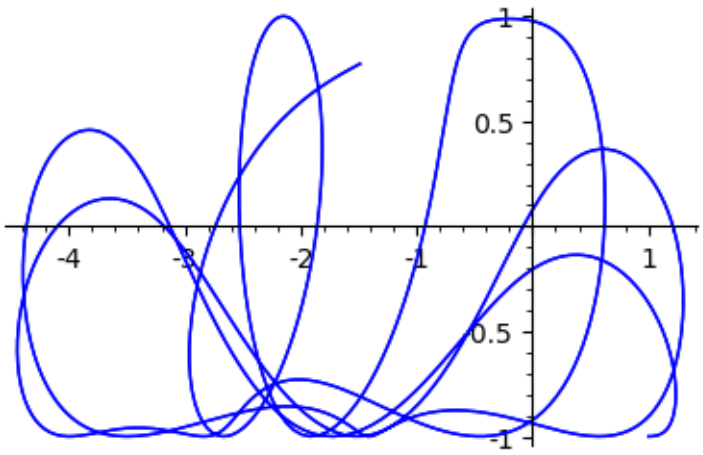
```
CPU times: user 320 ms, sys: 23.6 ms, total: 344 ms
Wall time: 339 ms
```

[14]:
```
line( zip(times,sol[::1,0]),figsize=(8,2) )+\
 line( zip(times,sol[::1,1]),color='red')
```
[14]:



[15]:
```
line( zip(times,sol[:,0]),figsize=4 )
line( zip(np.sin(sol[:,1])+sol[:,0],-np.cos(sol[:,1])),figsize=4 )+\
 line( zip(np.sin(sol[:,1])+sol[:,0],-np.cos(sol[:,1])),figsize=4 )
```

[15]:



7.2.1 Visualization

It is helpfull to write simple function displaying configuration of the system for given set of variables.

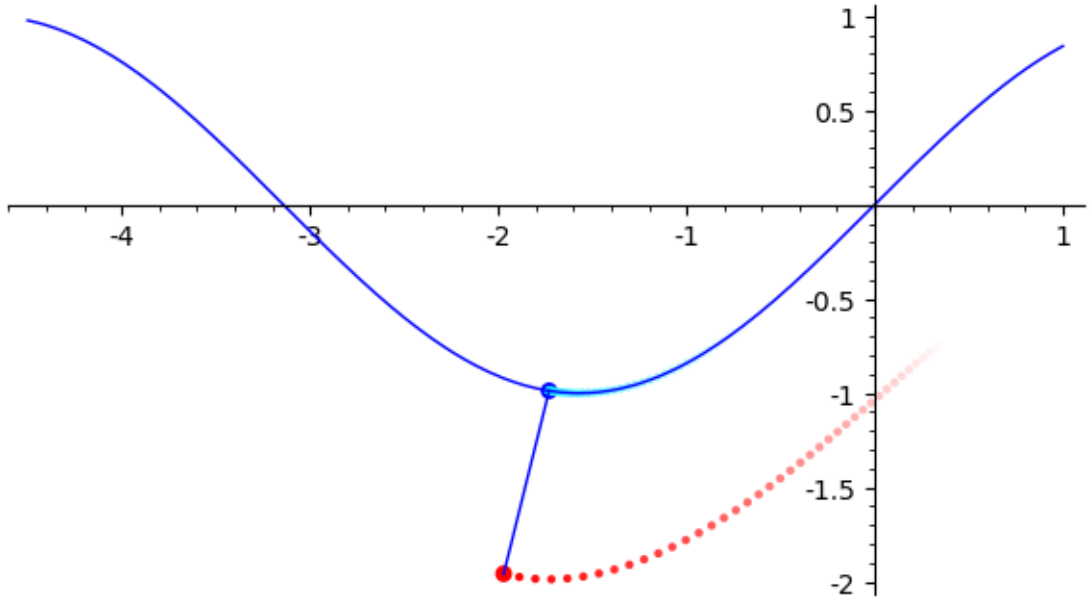
[16]:
```
def draw_system(ith=0,l=1):
    x,phi = sol[ith,:2]
    x1,y1,x2,y2 =x, f(x),  l*sin(phi) + x,f(x)-l*cos(phi)

    p = point( (x1,y1), size=40) +\
     point( (x2,y2), size=40,color='red',figsize=3) +\
     line( [(x1,y1),(x2,y2)],aspect_ratio=1)
    n=40
    i0 = max(0,ith-n)
```

```
    trace = sum([point((l*sin(phi) + x,f(x)-l*cos(phi)),hue=(0,1-(i)/n,1)) for
↪i,(x,phi) in enumerate(sol[ith:i0:-1,:2])])
    trace2 = sum([point((x,f(x)),hue=(.51,(i)/n,1)) for i,(x,phi) in
↪enumerate(sol[i0:ith,:2])])

    p += trace+trace2
    var('x_')
    p += plot(f(x_),(x_,-4.5,1),figsize=6 )
    p.set_axes_range(-4.5,1,-2,1)
    p.set_aspect_ratio(1)
    return p
```

Let's try:

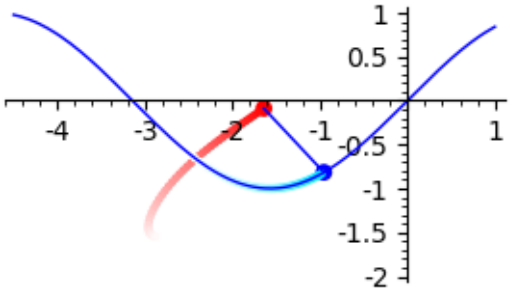
[17]:
```
draw_system(120)
```

[17]:



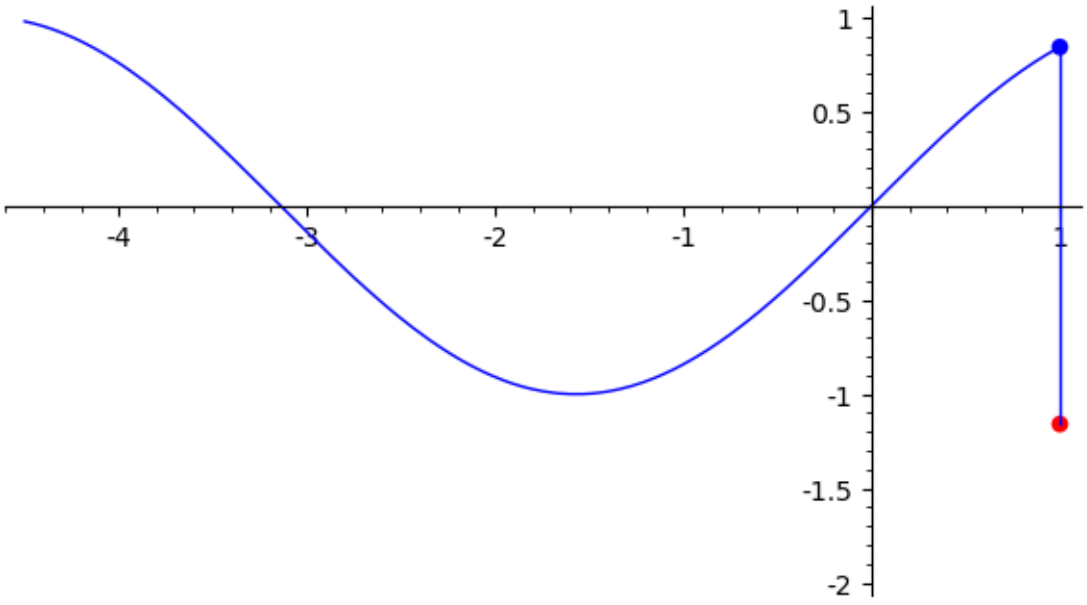
We can animate:

[18]:
```
from IPython.display import clear_output
import time
for ith in range(0,len(sol),20):
    plt = draw_system(ith=ith,l=1)
    clear_output(wait=True)
    plt.show(figsize=3)
    time.sleep(0.021)
```



Alternatively one can use slider:

[19]:
```
@interact
def _(ith=slider(range(len(sol)))):
    plt = draw_system(ith=ith,l=2)
    plt.show(figsize=6)
```



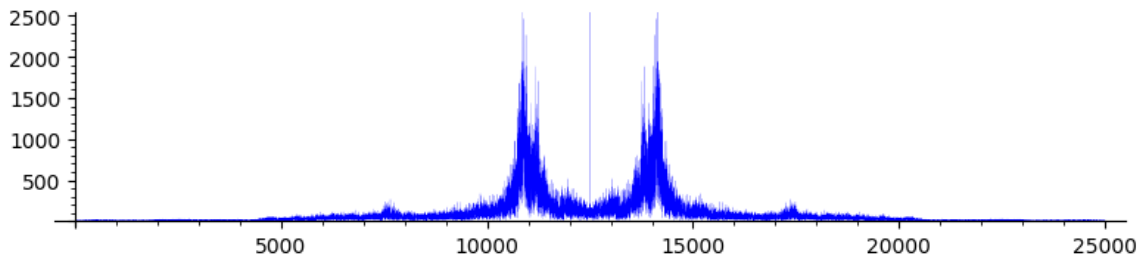
7.2.2 Chaotic properties of the solution

Spectrum The solution looks chaotic, but it can be caused by mixing of few frequencies. We can, however, calculate the Fourier transform of one of the system variables and see.

[20]:
```
%%time
pars = {l:1,g:9.81,m1:1.1,m2:1}
ode = [xd,phid,w1.rhs().subs(pars),w2.rhs().subs(pars)]
times = srange(0,5000.25,0.2)
ics = [1,0,0,0]
sol = desolve_odeint(ode,ics,times,[x,phi,xd,phid])
xfft = np.fft.fft(sol[:,0])
n1 = xfft.shape[0]
```

```
CPU times: user 8.12 s, sys: 962 ms, total: 9.08 s
Wall time: 7.98 s
```

[21]:
```
plt = line(enumerate(np.abs(np.fft.fftshift(xfft))),ymax=2500,thickness=0.1)
plt.show(figsize=(8,2))
```



Let us check for comparison that sum of two signals with different frequencies would not cause such effect

[22]:
```
expr = sin(1.2*t)+sin(sqrt(1.2)*t)
```

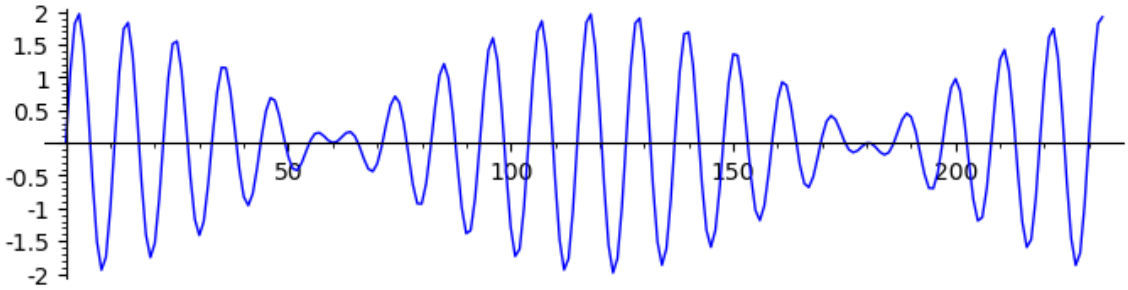
[23]:
```
import numpy as np
import sympy
expr_np = np.vectorize( sympy.lambdify(t, sympy.sympify( expr ) ) )
```

[24]:
```
nonchaotic = expr_np(np.linspace(0,5000,10000))
```

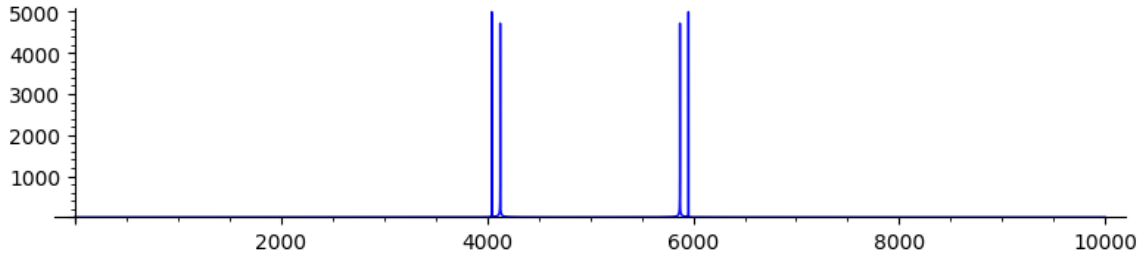
[25]:
```
%time nonchaotic_fft = np.fft.fft(nonchaotic)
```

```
CPU times: user 1.3 ms, sys: 59 µs, total: 1.36 ms
Wall time: 739 µs
```

[26]:
```
line(enumerate(nonchaotic[:234])).show(figsize=(7,2))
```



[27]:
```
plt2 = line(enumerate(np.abs(np.fft.fftshift(nonchaotic_fft))),alpha=1)
plt2.show(figsize=(8,2))
```



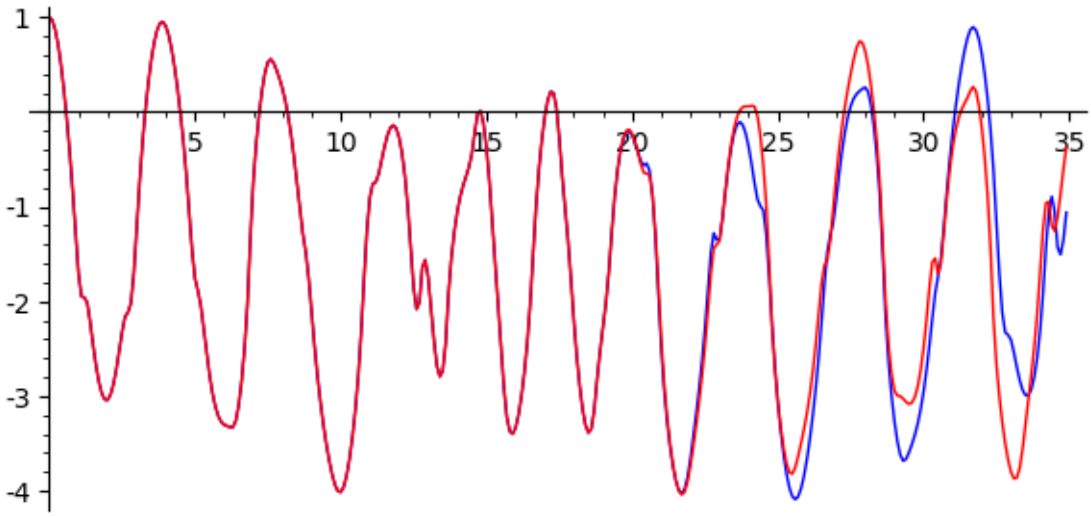
Sensitivity to initial conditions Let us compare two solution which differ by $\frac{1}{1000000}$ in initial velocity.

[28]:
```
%%time
pars = {l:1,g:9.81,m1:1.1,m2:1}
ode = [xd,phid,w1.rhs().subs(pars),w2.rhs().subs(pars)]
times = srange(0,35.,0.1)
ics = [1,0,0,0]
sol = desolve_odeint(ode,ics,times,[x,phi,xd,phid])
ics2 = [1+1e-6,0,0,0]
sol2 = desolve_odeint(ode,ics2,times,[x,phi,xd,phid])
```

```
CPU times: user 156 ms, sys: 12.2 ms, total: 168 ms
Wall time: 142 ms
```

[29]:
```
line(zip(times,sol[:,0]))+line(zip(times,sol2[:,0]),color='red',figsize=(6,3))
```

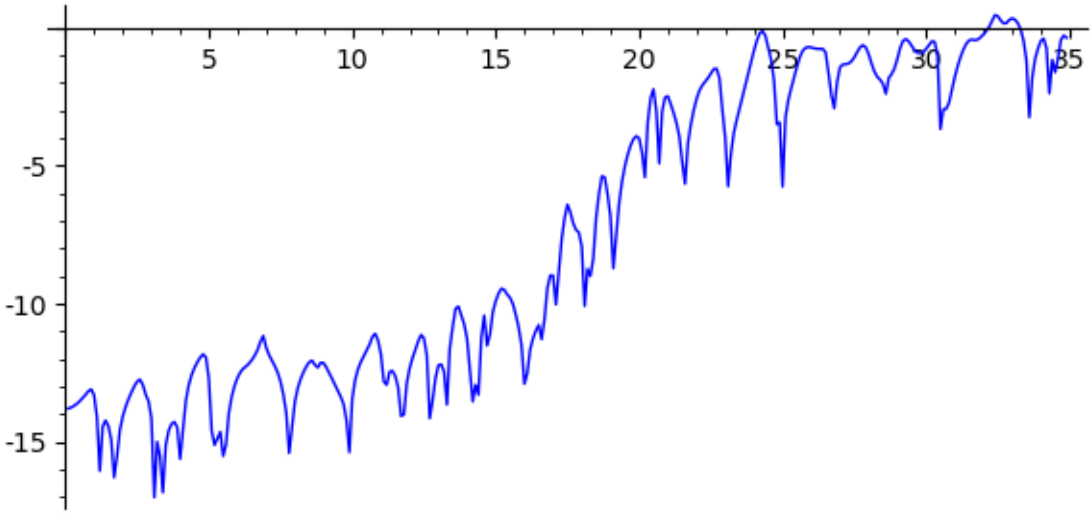
[29]:



We can have a look how the error propagetes in log-scale:

[30]:
```
line(zip(times[1:],log(abs(sol[1:,0]-sol2[1:,0]))),figsize=(6,3))
```

[30]:



8 A pendulum with a slipping suspension point

A pendulum with a slipping point of suspension

Consider a pendulum for which the suspension point can move horizontally freely.

[1]:
```
load('cas_utils.sage')
```

[2]:
```
var('t')
var('l g m1 m2')

xy_wsp = [('x1','x_1'),('y1','y_1'),('x2','x_2'),('y2','y_2')]

uv_wsp = [('phi','\phi'),('x','x')]

to_fun, to_var = make_symbols(xy_wsp, uv_wsp)
```

```
phi  :: has been  processed
x  :: has been  processed
x1  :: has been  processed
y1  :: has been  processed
x2  :: has been  processed
y2  :: has been  processed
```

8.1 Equations of motion in a Cartesian system

Let us derive the equations of motion in the Cartesian system. Let's start with the d'Alembert rule:

[3]:
```
dAlemb = (m1*x1.subs(to_fun).diff(t,2))*dx1 + \
         (m1*y1.subs(to_fun).diff(t,2)+m1*g)*dy1+\
         (m2*x2.subs(to_fun).diff(t,2))*dx2 + \
         (m2*y2.subs(to_fun).diff(t,2)+m2*g)*dy2
dAlemb = dAlemb.subs(to_var)
showmath(dAlemb)
```

[3]:

$$\delta x_1 m_1 \ddot{x}_1 + \delta x_2 m_2 \ddot{x}_2 + (g m_1 + m_1 \ddot{y}_1)\delta y_1 + (g m_2 + m_2 \ddot{y}_2)\delta y_2$$

Equations of constraints for the system are:

$-y_1 = 0$ $-(x_1 - x_2)^2 + (y_1 - y_2)^2 = l^2$

We calculate the variation of constraint equations (ie we present constraints in a differential form) using the formulas:

$$\delta f = \frac{\partial f}{\partial x_1}\delta x_1 + \frac{\partial f}{\partial x_2}\delta x_2 + \frac{\partial f}{\partial y_1}\delta y_1 + \frac{\partial f}{\partial y_2}\delta y_2$$

This difference is analogous to a general differential with the difference that time is treated as a constant.

[4]:
```
f = (x1-x2)^2+(y1-y2)^2-l^2
df = f.diff(x1)*dx1 + f.diff(x2)*dx2 + f.diff(y1)*dy1 + f.diff(y2)*dy2
showmath(df)
```

[4]:

$$2\,\delta x_1(x_1-x_2)-2\,\delta x_2(x_1-x_2)+2\,\delta y_1(y_1-y_2)-2\,\delta y_2(y_1-y_2)$$

[5]:
```
# wzor na df mozna zautomatyzowac w nastepujacy sposob
# df = sum([f.diff(w)*vars()['d'+repr(w)] for w in xy])
```

We substitute $\delta y_1 = 0$ and $y_1 = 0$ and then calculate δy_2 as a function of δx_1 and δx_2:

[6]:
```
dy2_wiezy = df.subs({dy1:0,y1:0}).solve(dy2)[0].rhs()
showmath( dy2_wiezy )
```

[6]:

$$-\frac{(\delta x_1-\delta x_2)x_1-(\delta x_1-\delta x_2)x_2}{y_2}$$

By substituting the term d'Alembert for the δy_2 expression as a function of the other shifts and $\delta y_1 = 0$, we get:

[7]:
```
showmath(  dAlemb.subs({dy2:dy2_wiezy,dy1:0}) )
```

[7]:

$$\delta x_1 m_1\ddot{x}_1+\delta x_2 m_2\ddot{x}_2-\frac{(gm_2+m_2\ddot{y}_2)((\delta x_1-\delta x_2)x_1-(\delta x_1-\delta x_2)x_2)}{y_2}$$

Moemy teraz pomnoy tak otrzyman zasad d'Alemberta przez y_2 oraz wycign przed nawias wspóczynniki przy niezalenych przesuniciach δx_1 oraz δx_2.

We can now multiply the so-obtained d'Alembert rule by y_2 and take parentheses with independent δx_1 and δx_2 offsets.

[8]:
```
showmath(  (dAlemb.subs({dy2:dy2_wiezy,dy1:0})*y2).expand().collect(dx1).
 ↪collect(dx2) )
```

[8]:

$$-(gm_2x_1-gm_2x_2-m_1\ddot{x}_1y_2+m_2x_1\ddot{y}_2-m_2x_2\ddot{y}_2)\delta x_1+(gm_2x_1-gm_2x_2+m_2\ddot{x}_2y_2+m_2x_1\ddot{y}_2-m_2x_2\ddot{y}_2)\delta x_2$$

Because the δx_1 and δx_2 virtual offsets are completely arbitrary (we have already used the dependency using constraints equations), both coefficients next to them must disappear so that the entire expression will be zeroed identically. In this way, we get two differential equations, which together with equations of constants describe the dynamics of our system:

[9]:
```
r1 = (dAlemb.subs({dy2:dy2_wiezy,dy1:0})*y2).expand().coefficient(dx1)
r2 = (dAlemb.subs({dy2:dy2_wiezy,dy1:0})*y2).expand().coefficient(dx2)
showmath( r1 )
```

[9]:

$$-gm_2x_1 + gm_2x_2 + m_1\ddot{x}_1y_2 - m_2x_1\ddot{y}_2 + m_2x_2\ddot{y}_2$$

[10]:
```
showmath( r2 )
```

[10]:

$$gm_2x_1 - gm_2x_2 + m_2\ddot{x}_2y_2 + m_2x_1\ddot{y}_2 - m_2x_2\ddot{y}_2$$

In order to be able to apply a numerical procedure to the above equations, the equation resulting from the differentiation of constraints comes:

[11]:
```
r3 = f.subs({y1:0}).subs(to_fun).diff(t,2).subs(to_var)
showmath( r3 )
```

[11]:

$$2\left(\dot{x}_1 - \dot{x}_2\right)^2 + 2\left(x_1 - x_2\right)\left(\ddot{x}_1 - \ddot{x}_2\right) + 2\,\dot{y}_2^2 + 2\,y_2\ddot{y}_2$$

The above three equations can be solved on $\ddot{x}_1$, $\ddot{x}_2$, $\ddot{y}_1$ and explicitly write a system of second degree equations that is directly applicable to numerical resolution:

[12]:
```
sol =  solve([r1,r2,r3],[x1dd,x2dd,y2dd])[0]
```

[13]:
```
showmath(sol[0])
```

[13]:

$$\ddot{x}_1 = -\frac{\left(\dot{x}_1^2 - 2\,\dot{x}_1\dot{x}_2 + \dot{x}_2^2 + \dot{y}_2^2\right)m_2x_1 - \left(\dot{x}_1^2 - 2\,\dot{x}_1\dot{x}_2 + \dot{x}_2^2 + \dot{y}_2^2\right)m_2x_2 - \left(gm_2x_1 - gm_2x_2\right)y_2}{\left(m_1 + m_2\right)x_1^2 - 2\left(m_1 + m_2\right)x_1x_2 + \left(m_1 + m_2\right)x_2^2 + m_1y_2^2}$$

[14]:
```
showmath(sol[1])
```

[14]:

$$\ddot{x}_2 = \frac{\left(\dot{x}_1^2 - 2\,\dot{x}_1\dot{x}_2 + \dot{x}_2^2 + \dot{y}_2^2\right)m_1x_1 - \left(\dot{x}_1^2 - 2\,\dot{x}_1\dot{x}_2 + \dot{x}_2^2 + \dot{y}_2^2\right)m_1x_2 - \left(gm_1x_1 - gm_1x_2\right)y_2}{\left(m_1 + m_2\right)x_1^2 - 2\left(m_1 + m_2\right)x_1x_2 + \left(m_1 + m_2\right)x_2^2 + m_1y_2^2}$$

[15]:
```
showmath(sol[2])
```

[15]:

$$\ddot{y}_2 = -\frac{\left(gm_1 + gm_2\right)x_1^2 - 2\left(gm_1 + gm_2\right)x_1x_2 + \left(gm_1 + gm_2\right)x_2^2 + \left(\dot{x}_1^2 - 2\,\dot{x}_1\dot{x}_2 + \dot{x}_2^2 + \dot{y}_2^2\right)m_1y_2}{\left(m_1 + m_2\right)x_1^2 - 2\left(m_1 + m_2\right)x_1x_2 + \left(m_1 + m_2\right)x_2^2 + m_1y_2^2}$$

8.1.1 Equations of motion in a system consistent with constraints

A much better idea is to solve the above problem in coordinates consistent with constraints. In this case, we will not have to additionally create a differential equation from the equation of constraints, the number of equations will be equal to the number of degrees of freedom (including cases 2). In addition, any solution to the system of differential equations will be

[16]:
```
x2u = {x1:x,x2:x+l*sin(phi),y2:-l*cos(phi),y1:0}
showmath(x2u)
```

[16]:

$$\{y_1 : 0, y_2 : -l\cos(\phi), x_1 : x, x_2 : l\sin(\phi) + x\}$$

To go to the description of the system in such parameterization:

- save $\ddot{x}_i - F_i$ expressions in new variables. - save $\delta x_1, \delta y_1, \delta y_2, \delta y_2$ virtual shifts as shifting functions in new $\delta x, \delta\phi$ variables using the formulas:

$$\delta x_1 = \frac{\partial x_1}{\partial x}\delta x + \frac{\partial x_1}{\partial \phi}\delta\phi$$

In order to execute the second point, let's define the 'dx1_polar' variables in Sage ... which are the expression of virtual offsets in the new parameterization:

[17]:
```
transform_virtual_displacements(xy_wsp, uv_wsp, verbose=True)
```

```
dx1_polar : is added to namespace

[dx1, dx]


dy1_polar : is added to namespace

[dy1, 0]


dx2_polar : is added to namespace

[dx2, dphi*l*cos(phi) + dx]


dy2_polar : is added to namespace

[dy2, dphi*l*sin(phi)]
```

[17]:
```
[dx, 0, dphi*l*cos(phi) + dx, dphi*l*sin(phi)]
```

The first point requires the transformation of the second Cartesian coordinate derivatives to the new parameterization. We can do this for every variable, let's take x_1 for example:

- we change coordinates to new ones - we change algebraic variables into time functions - we count the derivative over time - we are going back to algebraic variables

[18]:
```
showmath( x2.subs(x2u).subs(to_fun).diff(t,2).subs(to_var) )
```
[18]:

$$-l\dot{\phi}^2 \sin(\phi) + l\ddot{\phi} \cos(\phi) + \ddot{x}$$

Using this technique, we can rewrite the d'Alembert principle for our problem:

[19]:
```
dAlemb = (m1*x1.subs(x2u).subs(to_fun).diff(t,2)  )*dx1_polar + \
         (m1*y1.subs(x2u).subs(to_fun).diff(t,2)+m1*g)*dy1_polar + \
         (m2*x2.subs(x2u).subs(to_fun).diff(t,2)  )*dx2_polar + \
         (m2*y2.subs(x2u).subs(to_fun).diff(t,2)+m2*g)*dy2_polar
dAlemb = dAlemb.subs(to_var)
```

It looks like this:

[20]:
```
showmath(dAlemb)
```
[20]:

$$\left(\left(l\dot{\phi}^2 \cos(\phi) + l\ddot{\phi} \sin(\phi)\right)m_2 + gm_2\right)\delta\phi l \sin(\phi) - \left(l\dot{\phi}^2 \sin(\phi) - l\ddot{\phi} \cos(\phi) - \ddot{x}\right)(\delta\phi l \cos(\phi) + \delta x)m_2 + \delta x m_1 \ddot{x}$$

As in the previous case, the coefficients at δx and $\delta\phi$ must be zeroed, which implies giving us two conditions that are the equations of motion:

[21]:
```
r1 = dAlemb.expand().coefficient(dx).trig_simplify()
r2 = dAlemb.expand().coefficient(dphi).trig_simplify()
showmath(r1)
```
[21]:

$$-lm_2\dot{\phi}^2 \sin(\phi) + lm_2\ddot{\phi} \cos(\phi) + (m_1 + m_2)\ddot{x}$$

[22]:
```
showmath(r2)
```
[22]:

$$l^2 m_2 \ddot{\phi} + lm_2 \ddot{x} \cos(\phi) + glm_2 \sin(\phi)$$

Because each of these equations contains a second derivative of both variables, treating the above equations as a system of equations (linear) on $\ddot{x}$ and $\ddot{\phi}$ we solve it:

[23]:
```
sol = solve([r1,r2],[xdd,phidd])[0]
showmath(sol[0])
```
[23]:

$$\ddot{x} = -\frac{lm_2\dot{\phi}^2 \sin(\phi) + gm_2 \cos(\phi) \sin(\phi)}{m_2 \cos(\phi)^2 - m_1 - m_2}$$

[24]:
```
showmath(sol[1])
```
[24]:

$$\ddot{\phi} = \frac{lm_2\dot{\phi}^2 \cos(\phi) \sin(\phi) + (gm_1 + gm_2) \sin(\phi)}{lm_2 \cos(\phi)^2 - lm_1 - lm_2}$$

For further analysis, we can assign equations to the variables s1 and s2:

[25]:
```
s1,s2 = solve([r1,r2],[xdd,phidd])[0]
```

[26]:
```
showmath( expand( s1.rhs().denominator()/m2 ) )
```

[26]:

$$\cos(\phi)^2 - \frac{m_1}{m_2} - 1$$

[27]:
```
sol = solve([r1,r2],[xdd,phidd])[0]
```

[28]:
```
showmath(sol[0])
```

[28]:

$$\ddot{x} = -\frac{l m_2 \dot{\phi}^2 \sin(\phi) + g m_2 \cos(\phi) \sin(\phi)}{m_2 \cos(\phi)^2 - m_1 - m_2}$$

[29]:
```
showmath(sol[1])
```

[29]:

$$\ddot{\phi} = \frac{l m_2 \dot{\phi}^2 \cos(\phi) \sin(\phi) + (g m_1 + g m_2) \sin(\phi)}{l m_2 \cos(\phi)^2 - l m_1 - l m_2}$$

8.1.2 Case study $m_1 >> m_2$

One would expect that if the first mass is much larger than the other, the system of equations will strive for a mathematical pendulum. To do this, let's divide by m_1:

[30]:
```
showmath( ((s1.rhs().numerator()/m1).expand() )/( (s1.rhs().denominator()/m1).
↪expand() ) )
```

[30]:

$$-\frac{\frac{l m_2 \dot{\phi}^2 \sin(\phi)}{m_1} + \frac{g m_2 \cos(\phi) \sin(\phi)}{m_1}}{\frac{m_2 \cos(\phi)^2}{m_1} - \frac{m_2}{m_1} - 1}$$

[31]:
```
showmath( ((s2.rhs().numerator()/m1).expand() )/( (s2.rhs().denominator()/m1).
↪expand() ) )
```

[31]:

$$\frac{\frac{l m_2 \dot{\phi}^2 \cos(\phi) \sin(\phi)}{m_1} + g \sin(\phi) + \frac{g m_2 \sin(\phi)}{m_1}}{\frac{l m_2 \cos(\phi)^2}{m_1} - l - \frac{l m_2}{m_1}}$$

You can see that the first expression tends to zero and the second to

$$\frac{g \sin(\phi)}{l}$$

We can also use the function limit, which directly leads to the result:

[32]:
```
limit(s1.rhs(),m1=oo)
```

[32]: 0

[33]:
```
showmath( limit(s2.rhs(),m1=oo) )
```

[33]:

$$-\frac{g\sin(\phi)}{l}$$

8.1.3 Case study $m_2 >> m_1$

In this case, the first mass is negligible.

[34]:
```
showmath( limit(s1.rhs(),m2=oo) )
```

[34]:

$$-\frac{l\dot{\phi}^2\sin(\phi)+g\cos(\phi)\sin(\phi)}{\cos(\phi)^2-1}$$

[35]:
```
showmath( limit(s2.rhs(),m2=oo) )
```

[35]:

$$\frac{l\dot{\phi}^2\cos(\phi)\sin(\phi)+g\sin(\phi)}{l\cos(\phi)^2-l}$$

[36]:
```
showmath([s1.rhs().taylor(phi,0,1),s2.rhs().taylor(phi,0,1)])
```

[36]:

$$\left[\frac{(lm_2\dot{\phi}^2+gm_2)\phi}{m_1},-\frac{(lm_2\dot{\phi}^2+gm_1+gm_2)\phi}{lm_1}\right]$$

8.1.4 Numerical analysis of the system

The initial condition is four $x, \phi, \dot{x}, \dot{\phi}$ numbers. Consider, however, a subset of those conditions for which the total momentum of the system is zero. Note that the case of a system that has a non-zero total momentum can be reduced to a zero moment event by transformation to the center of mass system. So we have:

$$m_1\dot{x}_1+m_2\dot{x}_2=0$$

or

$$m_1\dot{x}+m_2\dot{x}+m_2l\dot{\phi}\cos(\phi)=0$$

for the case of $m_1 = m_2$ and starting from the lowest position of the second mass ($\phi = 0$) we will have:

$$2\dot{x}=-l\dot{\phi}=0.$$

So for this case, we have a one-parameter family of solutions in which the speed $\dot{\phi} = \omega_0$ is independent:

$$x = 0, \phi = 0, \dot{x} = -\frac{2}{l}\omega_0, \omega_0$$

[37]:
```
showmath( solve(m1*x+m2*x+m2*l*phid*cos(phi),x)[0].rhs() )
```

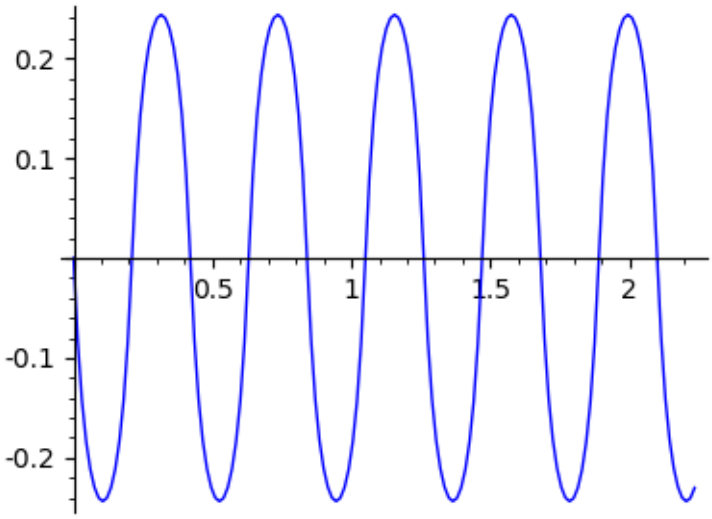
[37]:

$$-\frac{l m_2 \dot{\phi} \cos(\phi)}{m_1 + m_2}$$

[38]:
```
pars = {l:1,g:9.81,m1:2.1,m2:130}
ode=[xd,phid,s1.rhs().subs(pars),s2.rhs().subs(pars)]
times=srange(0,2.25,0.015)
ics=[0,0,-(1/2),14.]
#ics=[0,pi/2,0,3]
w0 = 6.2
ics = [0,0,(-l*m2*w0/(m1+m2)).subs(pars),w0]
sol = desolve_odeint(ode,ics,times,[x,phi,xd,phid])
```

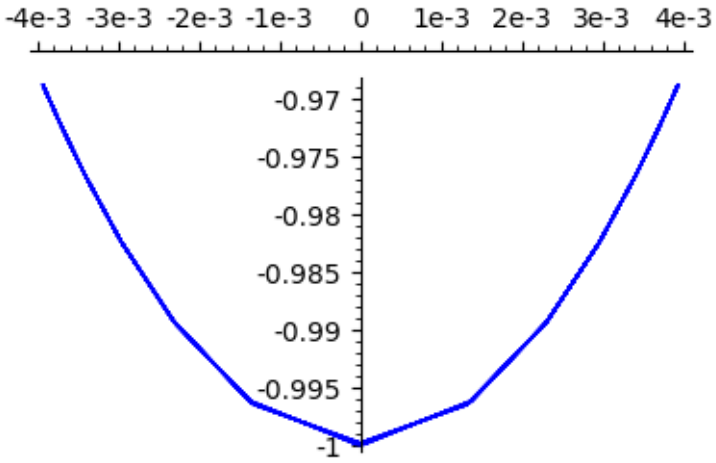
[39]:
```
line( zip(times,sol[:,0]),figsize=4 )
```

[39]:



[40]:
```
import numpy as np
line( zip(np.sin(sol[:,1])+sol[:,0],-np.cos(sol[:,1])),figsize=4 )+\
 line( zip(np.sin(sol[:,1])+sol[:,0],-np.cos(sol[:,1])),figsize=4 )
```

[40]:

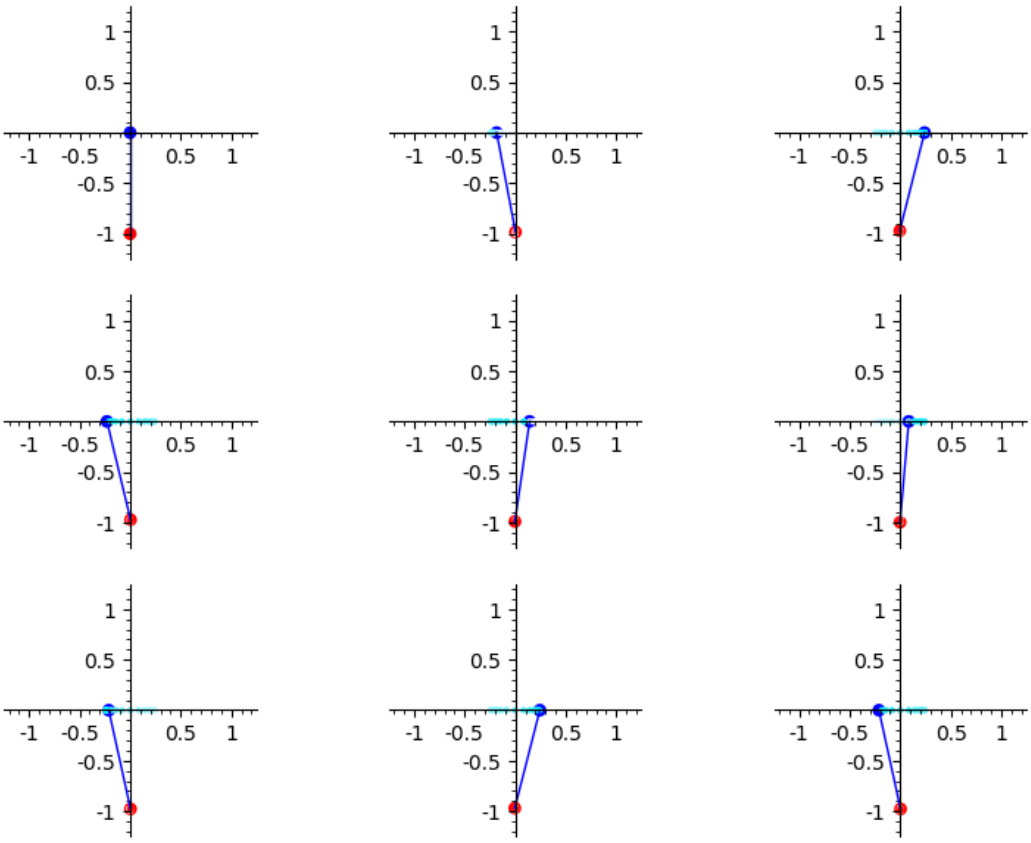


[41]:
```
def draw_system(ith=0,l=1):
    x,phi = sol[ith,:2]
    x1,y1,x2,y2 =x, 0,  l*sin(phi) + x,-l*cos(phi)

    p = point( (x1,y1), size=40) +\
     point( (x2,y2), size=40,color='red',figsize=3) +\
     line( [(x1,y1),(x2,y2)],aspect_ratio=1)
    n=20
    i0 = max(0,ith-n)
    trace = sum([point((l*sin(phi) + x,-l*cos(phi)),hue=(0,(i)/n,1)) for␣
↪i,(x,phi) in enumerate(sol[i0:ith,:2])])
    trace2 = sum([point((x,0),hue=(.51,(i)/n,1)) for i,(x,phi) in␣
↪enumerate(sol[i0:ith,:2])])
    #print i0,ith,[(i)/n. for i,(x,phi) in enumerate(sol[i0:ith,:2])]
    p += trace+trace2

    p.set_axes_range(-1.2,1.2,-1.2,1.2)
    p.set_aspect_ratio(1)
    return p
```

[42]:
```
sol.shape
```

[42]:
```
(150, 4)
```

[43]:
```
graphics_array([draw_system(i*11) for i in range(9)],ncols=3).show(figsize=8)
```




```
%%time
N = sol.shape[0]
every = int(N/25)
anim = animate([draw_system(i) for i in range(0,N,every)])
anim.show()
```

```
@interact
def _(ith=slider(range(N))):
    plt = draw_system(ith=ith,l=1)
    plt.show(figsize=6)
```

8.1.5 Problems

- Compare the layout solution in a general form to solutions of special cases: - $m_1 >> m_2$ - How does the period of movement depend on total energy? Compare the result with the mathematical pendulum.

9 Euler Lagrange - pendulum with oscillating support

We define Lagrange function as a difference between kinetic and potential energy:

$$L = E_k - E_p \tag{15}$$

Then equation of motion are given by:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\varphi}}\right) - \frac{\partial L}{\partial \varphi} = 0 \tag{16}$$

Since this formulation in invariant with respect to the change of system off coordinates, we can use it for many problems in mechanics with constraints.

9.1 System definition

Let us define a system,

[1]:
```
load('cas_utils.sage')
```

[2]:
```
var('l g w0')
xy_wsp = ['x','y']
uv_wsp = [('phi',r'\varphi')]

to_fun, to_var = make_symbols(xy_wsp, uv_wsp)
```

```
phi  :: has been  processed
x  :: has been  processed
y  :: has been  processed
```

9.1.1 Horizontal oscillations of a support point

We parametrize the system similarily to mathematical pendulum,

$$x = a\sin(\omega t) + l\sin(\varphi) \tag{17}$$
$$y = -l\cos(\varphi) \tag{18}$$

[3]:
```
# horizontal
var('a omega t')
x2u = {x:l*sin(phi)+a*sin(omega*t),y:-l*cos(phi)}
showmath(x2u)
```

[3]:

$$\{y : -l\cos(\varphi), x : a\sin(\omega t) + l\sin(\varphi)\}$$

Step 1: Kinetic energy

We have to write kinetic energy in terms of generalized coordinates:

$$E_k = \frac{1}{2}(\dot{x}^2 + \dot{y}^2) \tag{19}$$

Using transformation dictionary, to generalized coordinates we have $E_k(\varphi, , \dot{\varphi})$:

[4]:
```
Ek = 1/2*sum([x_.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2 for x_ in [x,y]])
Ek = Ek.trig_simplify()
showmath(Ek)
```

[4]:

$$\frac{1}{2}\,a^2\omega^2\cos\left(\omega t\right)^2 + al\omega\dot{\varphi}\cos\left(\omega t\right)\cos\left(\varphi\right) + \frac{1}{2}\,l^2\dot{\varphi}^2$$

Step 2: Potential energy

Similarily we have to express potential energy:

$$E_p = gy \tag{20}$$

as a function of $E_P(\varphi)$

[5]:
```
Ep = g*y.subs(x2u)
showmath(Ep)
```

[5]:

$$-gl\cos\left(\varphi\right)$$

Step 3: Langranian

Now we have Lagrangian $L(\varphi, \dot{\varphi})$:

[6]:
```
L = Ek - Ep
showmath(L)
```

[6]:

$$\frac{1}{2}\,a^2\omega^2\cos\left(\omega t\right)^2 + al\omega\dot{\varphi}\cos\left(\omega t\right)\cos\left(\varphi\right) + \frac{1}{2}\,l^2\dot{\varphi}^2 + gl\cos\left(\varphi\right)$$

9.2 Derivation of equations of motion

Using Euler-Lagrange formulas 16 we write equation of motion in generalized coordinate φ. Note, we can differentiate over φ and $\dot{\varphi}$. However to perform time derivative we first replace symbols representing variables with functions (i.e. Sage symbolic functions) of time. We have `to_fun` dictionary which automatizes this step. Then we use symbolic differenctiation `diff`. After this operation we bring thhe result back to symbolic variable φ and $\dot{\varphi}$ with `to_var` dictionary.

[7]:
```
EL1 = L.diff(phid).subs(to_fun).diff(t).subs(to_var) - L.diff(phi)
```

[8]:
```
showmath(EL1)
```

[8]:

$$-al\omega^2\cos(\varphi)\sin(\omega t) + l^2\ddot{\varphi} + gl\sin(\varphi)$$

9.3 Analysis

9.3.1 Small angle approximation

Let see what happens if oscillations are small. We can expand in Taylor seried the equations of motion:

[9]:
```
eq_lin = EL1.taylor(phi,0,1)
showmath(eq_lin)
```

[9]:

$$-al\omega^2\sin(\omega t) + gl\varphi + l^2\ddot{\varphi}$$

[10]:
```
var('alpha,omega0')
eq_lin2  = (eq_lin/l^2).expand().subs({a:l*alpha,g:l*omega0^2})
showmath(eq_lin2)
```

[10]:

$$-\alpha\omega^2\sin(\omega t) + \omega_0^2\varphi + \ddot{\varphi}$$

We see that the equations are essentially equivalent to forced harmonic oscillator. The difference might be that the effective amplitude of forcing depends on forcing frequency.

[11]:
```
assume(g>0)
assume(omega0>0)
phi_anal = desolve((eq_lin/l^2).expand()\
                   .subs({a:l*alpha,g:l*omega0^2})\
                   .subs(to_fun).subs({l:1}),\
                   dvar=Phi,ivar=t,contrib_ode=True)
showmath(phi_anal)
```

[11]:

$$-\frac{\alpha\omega^2\sin(\omega t)}{\omega^2-\omega_0^2} + K_2\cos(\omega_0 t) + K_1\sin(\omega_0 t)$$

[12]:
```
showmath(eq_lin2)
```

[12]:

$$-\alpha\omega^2\sin(\omega t) + \omega_0^2\varphi + \ddot{\varphi}$$

9.3.2 Numerical integration

We can numerically compare if the linear approximation works for selected initial conditions and parameters. For this purpose we need to solve Euler-Lagrange equation for $\ddot{\varphi}$, and for following system od 1st order ODEs:

$$\frac{d\varphi}{dt} = \dot{\varphi} \tag{21}$$

$$\frac{d\dot{\varphi}}{dt} = \frac{a\omega^2 \cos(\varphi)\sin(\omega t)}{l} - \frac{g\sin(\varphi)}{l} \tag{22}$$

Note that we threat φ and $\dot{\varphi}$ as independent variables. Since in Sage we use formulas where there are represented by different symbolic variables: `phi` and `dphi`, there will be no confusion of "dot" and derivative operator.

[13]:
```
rhs = EL1.solve(phidd)[0].rhs()
showmath(rhs().expand())
```

[13]:

$$\frac{a\omega^2 \cos(\varphi)\sin(\omega t)}{l} - \frac{g\sin(\varphi)}{l}$$

Linear system can be derived in similar way:

[14]:
```
rhs_lin = eq_lin.solve(phidd)[0].rhs()
showmath(rhs_lin)
```

[14]:

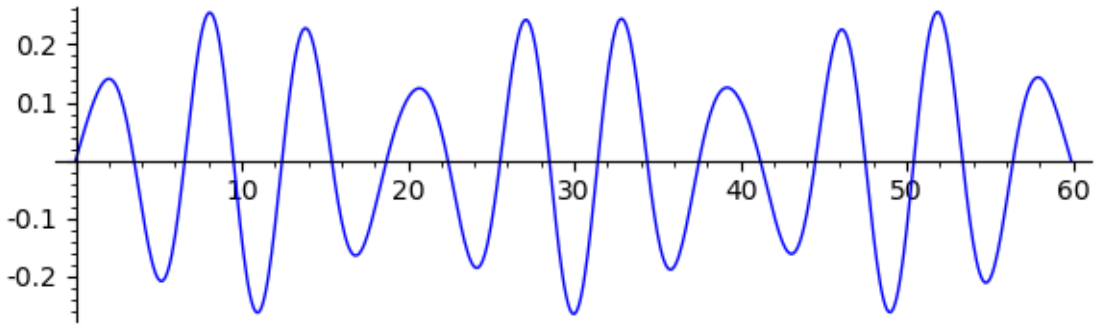
$$\frac{a\omega^2 \sin(\omega t) - g\varphi}{l}$$

[15]:
```
pars = {l:1,g:1,a:.03,omega:1.31}
t_end = 60
w0 = sqrt(g/l).subs(pars)
```

Now we can plug the system of ODE into `desolve_odeint` solver:

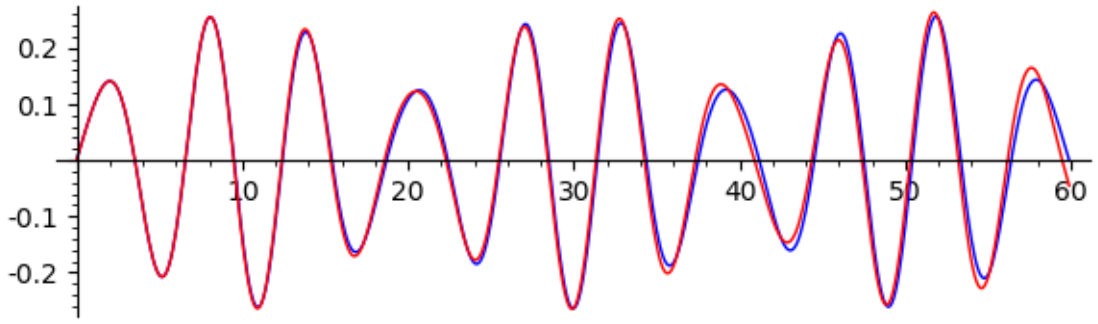
[16]:
```
ode = [phid, rhs.subs(pars)]
times = srange(0,t_end,0.1)
ics = [0.0, 0.1]
sol = desolve_odeint(ode, ics, times, [phi, phid])
line( zip(times,sol[::1,0]),figsize=(6,2), )
```

[16]:



[17]:
```
ode_lin = [phid, rhs_lin.subs(pars)]
times = srange(0,t_end,0.1)
ics = [0.0, 0.1]
sol_lin = desolve_odeint(ode_lin, ics, times, [phi,phid])

line( zip(times[0:],sol[0:,0]),figsize=(6,2) )\
 +line( zip(times[0:],sol_lin[0:,0]),color='red')
```

[17]:



We see that for small oscillations the for some time. Then they diverge. One can experiment and simulate both systems for longer times to see that the divergence grows. Also larger amplitudes of driving will make them differ significantly.

9.3.3 Vertical oscillations

In the case of vertical oscillations of a support point, the transformation to generalized coordinates reads:

$$x = l \sin(\varphi) \tag{23}$$
$$y = -a \cos(\omega t) - l \cos(\varphi), \tag{24}$$

[18]:
```
# vertical
var('a omega t')
x2u = {x:l*sin(phi), y:-l*cos(phi)-a*cos(omega*t)}
showmath(x2u)
```

[18]:

$$\{y : -a\cos(\omega t) - l\cos(\varphi), x : l\sin(\varphi)\}$$

Let us once again calculate Lagrangian and derive symbolically equations of motion:

[19]:
```
Ek = 1/2*sum([x_.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2 for x_ in [x,y]])
Ek = Ek.trig_simplify()
Ep = g*y.subs(x2u)
L = Ek - Ep
EL1 = L.diff(phid).subs(to_fun).diff(t).subs(to_var) - L.diff(phi)
showmath(EL1)
```

[19]:

$$al\omega^2\cos(\omega t)\sin(\varphi) + l^2\ddot{\varphi} + gl\sin(\varphi)$$

[20]:
```
rhs = EL1.solve(phidd)[0].rhs()
showmath( rhs.expand().collect(sin(phi)) )
```

[20]:

$$-\left(\frac{a\omega^2\cos(\omega t)}{l} + \frac{g}{l}\right)\sin(\varphi)$$

Let's try to obtain a linear approximation for small φ, as in previous case:

[21]:
```
showmath(EL1.taylor(phi,0,1) )
```

[21]:

$$l^2\ddot{\varphi} + \left(al\omega^2\cos(\omega t) + gl\right)\varphi$$

We see that in this case we do not obtain a harmonic oscillator. Forcing term is multiplied by φ, i.e. the linearized equation is fundamentally different.

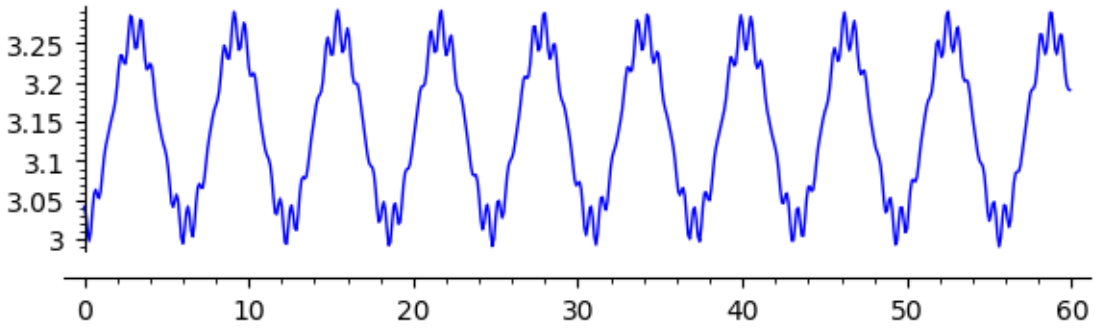
9.3.4 Stable inverted pendulum

Vertical driving of a support point as a remarkable property - under some conditions the upper steady state can become a stable one.

For example for following parameters and initial condition:

[22]:
```
pars = {l:1,a:.2,g:1,omega:10.}
w0 = sqrt(g/l).subs(pars)

ode = [phid, rhs.subs(pars) ]
times = srange(0, 60, 0.1)
ics = [pi-1e-1, .0]
```

```
sol = desolve_odeint(ode, ics, times, [phi,phid])

#plt = line( zip(times,sol[::1,0]),figsize=(8,3), ticks=[None,pi/4],\
#     tick_formatter=[None,pi],gridlines=[[],[pi.n()*i for i in␣
 ↪range(-100,100,1)]])
plt = line( zip(times,sol[::1,0]),figsize=(6,2) )

plt.show()
```



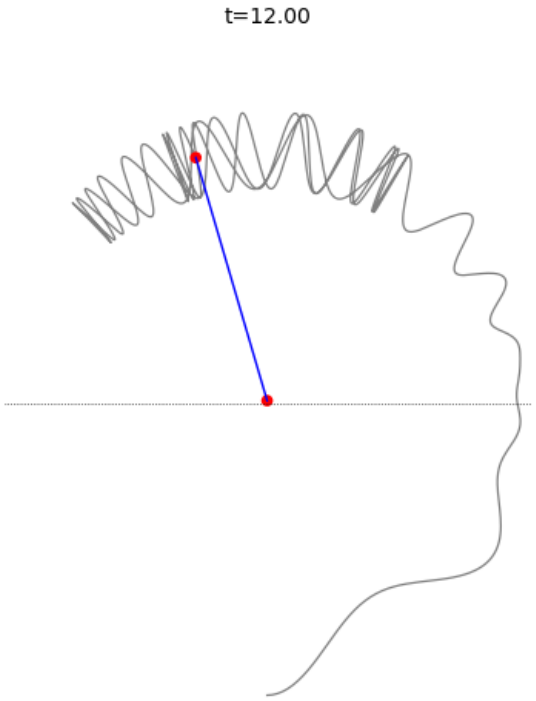
We observe that for above inital conditions the pendulum oscillates around $x = \pi$ **state** which is normally unstable fix point.

[23]:
```
pendulum = [vector([x,y]).subs(x2u).subs(pars).subs({phi:phi_,t:t_})\
            for t_,phi_ in zip(times,sol[:,0])]
o_point = [vector([x,y]).subs(x2u).subs(l==0).subs(pars).subs(t==t_)\
           for t_ in times ]
```

[24]:
```
#@interact
def draw_pendulum(ith = slider(0, len(pendulum)-1,1)):
    p1,p2 = pendulum[ith], o_point[ith]
    plt = line( [p1,p2],xmin=-1, xmax=1, ymin=-1.4, ymax=1.4,\
                aspect_ratio=1,figsize=2,axes=False, title='t=%0.2f'%times[ith])
    plt += points([p1,p2],color='red',size=30,gridlines=[None,[0]],\
                  figsize=3,axes=False)
    plt += line(pendulum[:ith],thickness=0.9,color='gray',zorder=-10)
    return plt
```

[25]:
```
#draw_pendulum(1200).save('inverted_pend.png',figsize=8)
```

Time evolution of the inverted stable pendulum:



Inverted pendulum stalibized by oscillations of a support point

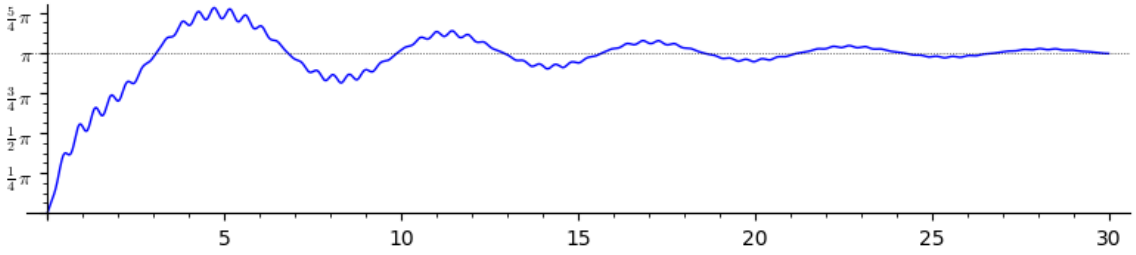
9.3.5 System with damping

Adding damping to the system will make inverted state an stable atractor.

$$\ddot{\varphi} = -2\gamma\dot{\varphi} + (-\omega_0^2 - \frac{a}{l}\omega^2\cos(\omega t))\sin(\varphi)$$

[26]:
```
var('omega,omega0,gama,t,a')
pars = {l:1,a:0.152,omega0:1,omega:14.,gama:.1}
ode = [phid,\
       (-2*gama*phid+(-omega0^2-a/l*omega^2*cos(omega*t))*sin(phi)).subs(pars)]
times = srange(0,30,0.01)
ics = [0,2.1]
sol = desolve_odeint(ode,ics,times,[phi,phid])
line( zip(times,sol[::1,0]),figsize=(8,2), ticks=[None,pi/4],\
     tick_formatter=[None,pi],gridlines=[[],[pi.n()*i for i in␣
 ↪range(-100,100,1)]])
```

[26]:


$\frac{5}{4}\pi$
π
$\frac{3}{4}\pi$
$\frac{1}{2}\pi$
$\frac{1}{4}\pi$
5
10
15
20
25
30

10 Point particle on rotating curve

10.1 Point particle on parabola

[1]:
```
load('cas_utils.sage')
```

[2]:
```
var('t')
var('w0 g')
xy_names = [('x','x'),('y','y')]
```

[3]:
```
to_fun, to_var = make_symbols(xy_names)
```

```
x  :: has been  processed
y  :: has been  processed
```

[4]:
```
dAlemb = (X.diff(t,2)-x*w0^2)*dx + (Y.diff(t,2)+g)*dy
f = 1/2*x^2 - y
```

[5]:
```
df = diff(f,x)*dx+diff(f,y)*dy
```

[6]:
```
eq1 = dAlemb.subs(df.solve(dx)).coefficient(dy).subs(to_var)
show(eq1)
```

```
g + ydd - (w0^2*x - xdd)/x
```

[7]:
```
eq2 = f.subs(to_fun).diff(t,2).subs(to_var)
show(eq2)
```

```
xd^2 + x*xdd - ydd
```

[8]:
```
sol = solve( [eq1,eq2],[xdd,ydd])
show( sol[0] )
```

```
[xdd == (w0^2 - xd^2 - g)*x/(x^2 + 1),
 ydd == ((w0^2 - g)*x^2 + xd^2)/(x^2 + 1)]
```

```
dxy = [vars()['d'+repr(zm)] for zm in xy]
constr = sum([dzm*f.diff(zm) for zm, dzm in zip(xy,dxy)])
show(f)
show(constr)
```

```
rown1=(dAlemb.subs(constr.solve(dx)[0])*x).expand().coeff(dy).subs(to_var)
show(rown1)
rown2 = f.subs(to_fun).diff(t,2).subs(to_var)
show(rown2)

sol = solve( [rown1,rown2],[xdd,ydd])
show( sol[0] )
```

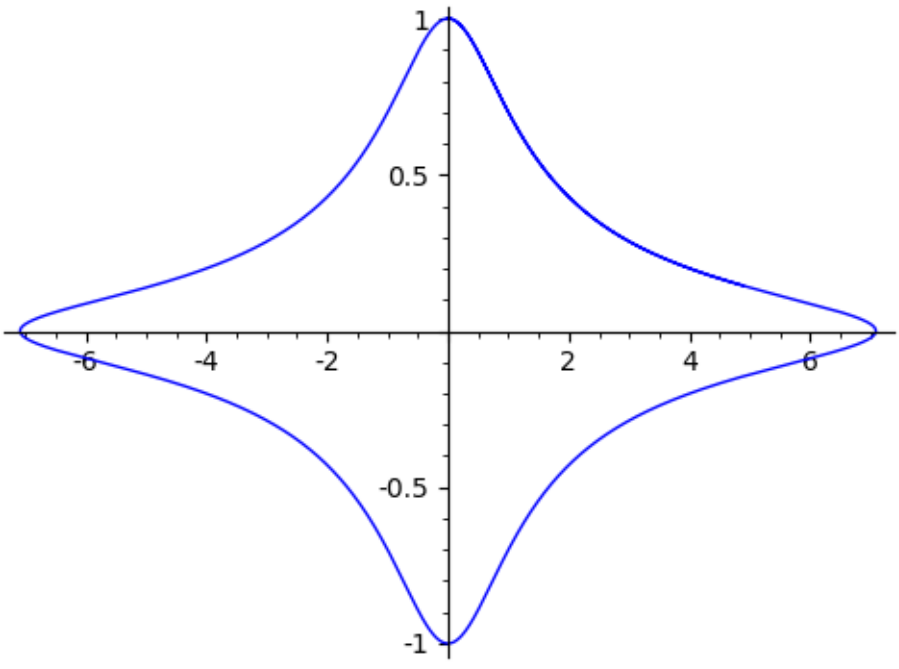
[9]:
```
show( sol[0][0])
```

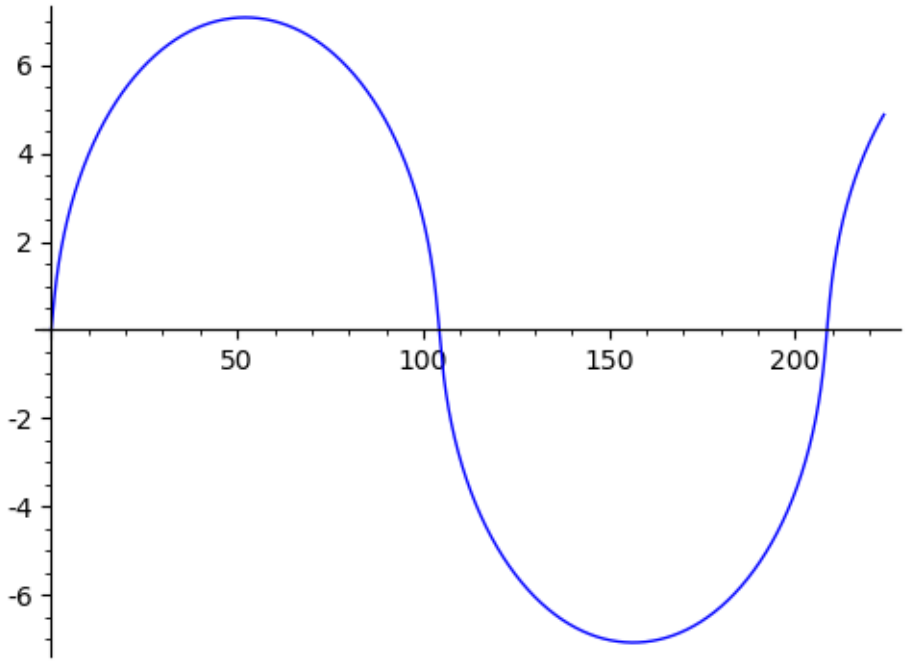
```
xdd == (w0^2 - xd^2 - g)*x/(x^2 + 1)
```

[10]:
```
ode=[xd,sol[0][0].rhs().subs({w0:1-0.01,g:1})]
show(ode)
```

```
[xd, -(xd^2 + 0.0199000000000000)*x/(x^2 + 1)]
```

[11]:
```
times = srange(0,224,0.01)
numsol = desolve_odeint(ode,[0,1],times,[x,xd])
p = line(zip(numsol[:,0],numsol[:,1]),figsize=5)
p2 = line(zip(times,numsol[:,0]),figsize=5)

p.show()
p2.show()
```





10.2 Point particle on rotating circle

[12]:
```
var('x y t')
var('w0 l g')

xy_names = ['x','y']
uv_names = [('phi','\phi')]

to_fun, to_var = make_symbols(xy_names,uv_names)
```

```
phi  :: has been  processed
x  :: has been  processed
y  :: has been  processed
```

[13]:
```
x2u = {x:-l*sin(phi),y:-l*cos(phi)}
```

[14]:
```
transform_virtual_displacements(xy_names,uv_names,verbose=True)
```

```
dx_polar : is added to namespace
```

```
[dx, -dphi*l*cos(phi)]
```

```
dy_polar : is added to namespace
```

```
[dy, dphi*l*sin(phi)]
```

[14]:
```
[-dphi*l*cos(phi), dphi*l*sin(phi)]
```

[15]:
```
x.subs(x2u).subs(to_fun)
```

[15]:
```
-l*sin(Phi(t))
```

[16]:
```
dAlemb = (x.subs(x2u).subs(to_fun).diff(t,2)-w0^2*x.subs(x2u))*dx_polar + \
         (y.subs(x2u).subs(to_fun).diff(t,2)+g)*dy_polar
dAlemb = dAlemb.subs(to_var)
show(dAlemb)
```

```
-(l*phid^2*sin(phi) + l*w0^2*sin(phi) - l*phidd*cos(phi))*dphi*l*cos(phi) + (l*phid^2*cos(ph
```

[17]:
```
sol = dAlemb.expand().coefficient(dphi).trig_simplify().solve(phidd)
show( sol)
```

```
[phidd == (l*w0^2*cos(phi)*sin(phi) - g*sin(phi))/l]
```

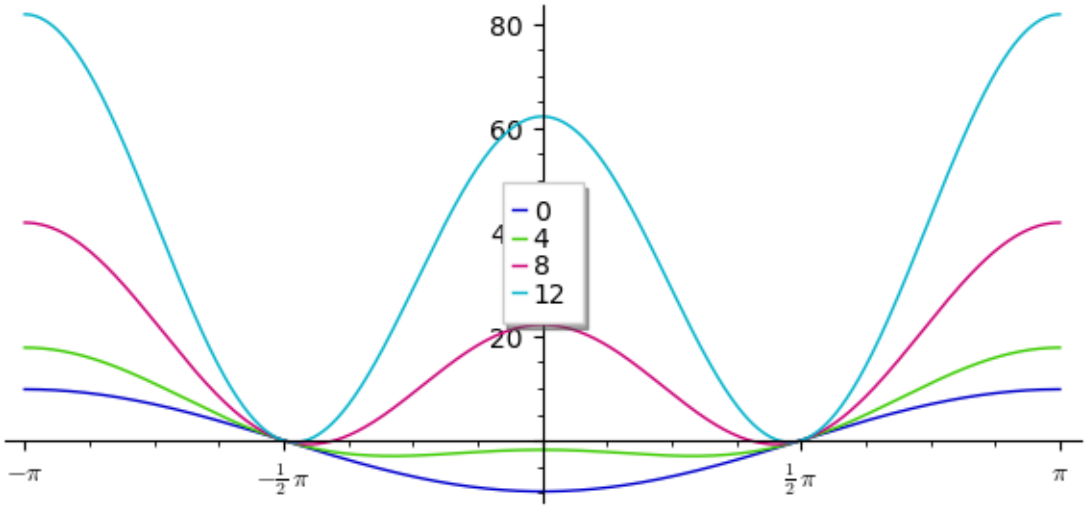
10.2.1 Efffective potential

[18]:
```
Ueff = -sol[0].rhs().expand().subs({g:9.81,l:1}).integrate(phi)
Ueff
```

[18]:
```
1/2*w0^2*cos(phi)^2 - 9.81*cos(phi)
```

[19]:
```
plot( [Ueff.subs(w0==w0_) for w0_ in [0,4,8,12]], (phi,-pi,pi),\
     legend_label=[0,4,8,12], figsize=(6,3),\
     tick_formatter=[pi,None],ticks=[pi/2,None])
```

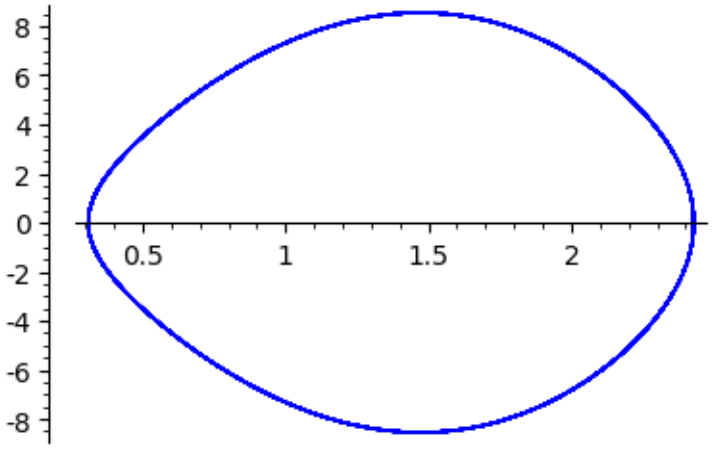
[19]:

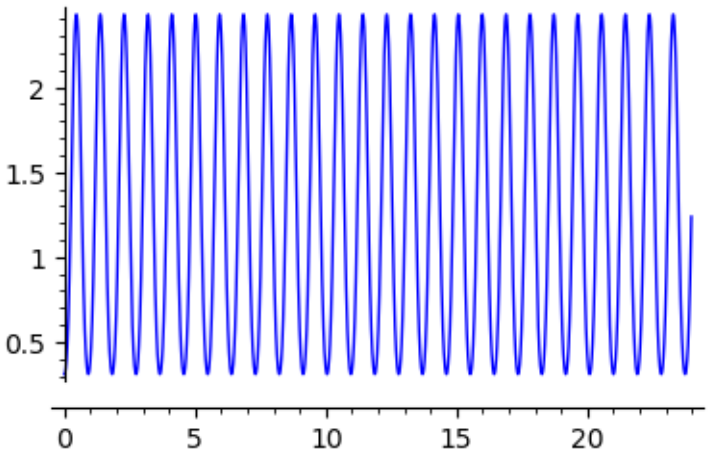


10.2.2 Numerical solutions

[20]:
```
times = srange(0,24,0.01)
ode = [phid,sol[0].rhs().subs({l:1,w0:9.81+.2,g:9.81})]
show(ode)
numsol = desolve_odeint(ode,[0.31,0],times,[phi,phid])
p = line(zip(numsol[:,0],numsol[:,1]),figsize=4)
p2 = line(zip(times,numsol[:,0]),figsize=4)

p.show()
p2.show()
```

```
[phid, 100.200100000000*cos(phi)*sin(phi) - 9.81000000000000*sin(phi)]
```





10.2.3 Code generation

We can readily generate code which can be used in external programs.

En example can be found in simulation and 3d vis

[21]:
```
from sympy import ccode
```

[22]:
```
oderhs = sol[0].rhs()
```

[23]:
```
ccode(oderhs._sympy_())
```

[23]:
```
'(-g*sin(phi) + l*pow(w0, 2)*sin(phi)*cos(phi))/l'
```

11 Double pendulum

Consider the pendulum suspended on the pendulum. We weill use dAlembert principle to derive eqaution of motion in generalized coordinaes. Naturally, we choose two angles as coordinates which comply with contraints.

[1]:
```
var('t')
var('l1 l2 m1 m2 g')

xy_names = [('x1','x1'),('y1','y1'),('x2','x2'),('y2','y2')]
uv_names = [('phi1','\\varphi_1'),('phi2','\\varphi_2')]
```

[2]:
```
load('cas_utils.sage')
```

[3]:
```
to_fun, to_var = make_symbols(xy_names,uv_names)
```

```
phi1  :: has been  processed
phi2  :: has been  processed
x1  :: has been  processed
y1  :: has been  processed
x2  :: has been  processed
y2  :: has been  processed
```

[4]:
```
x2u = {x1:l1*sin(phi1),\
       y1:-l1*cos(phi1),\
       x2:l1*sin(phi1)+l2*sin(phi2),\
       y2:-l1*cos(phi1)-l2*cos(phi2)}
```

[5]:
```
transform_virtual_displacements(xy_names,uv_names,verbose=True)
```

```
dx1_polar : is added to namespace

[dx1, dphi1*l1*cos(phi1)]


dy1_polar : is added to namespace

[dy1, dphi1*l1*sin(phi1)]


dx2_polar : is added to namespace

[dx2, dphi1*l1*cos(phi1) + dphi2*l2*cos(phi2)]
```

```
dy2_polar : is added to namespace

[dy2, dphi1*l1*sin(phi1) + dphi2*l2*sin(phi2)]
```

[5]:
```
[dphi1*l1*cos(phi1),
 dphi1*l1*sin(phi1),
 dphi1*l1*cos(phi1) + dphi2*l2*cos(phi2),
 dphi1*l1*sin(phi1) + dphi2*l2*sin(phi2)]
```

[6]:
```
dAlemb = (m1*x1.subs(x2u).subs(to_fun).diff(t,2))*dx1_polar + \
         (m1*y1.subs(x2u).subs(to_fun).diff(t,2)+m1*g)*dy1_polar+\
         (m2*x2.subs(x2u).subs(to_fun).diff(t,2))*dx2_polar + \
         (m2*y2.subs(x2u).subs(to_fun).diff(t,2)+m2*g)*dy2_polar
dAlemb = dAlemb.subs(to_var)
```

[7]:
```
showmath(dAlemb)
```

[7]:

$$-\left(l_1\dot{\varphi}_1^2\sin\left(\varphi_1\right)-l_1\ddot{\varphi}_1\cos\left(\varphi_1\right)\right)\delta\varphi_1 l_1 m_1\cos\left(\varphi_1\right)+\left(\left(l_1\dot{\varphi}_1^2\cos\left(\varphi_1\right)+l_1\ddot{\varphi}_1\sin\left(\varphi_1\right)\right)m_1+gm_1\right)\delta\varphi_1 l_1\sin\left(\varphi_1\right)-\left(l$$

[8]:
```
eq1 = dAlemb.expand().coefficient(dphi1).trig_simplify()
eq2 = dAlemb.expand().coefficient(dphi2).trig_simplify()
showmath(eq1)
```

[8]:

$$\left(l_1 l_2 m_2\cos\left(\varphi_2\right)\sin\left(\varphi_1\right)-l_1 l_2 m_2\cos\left(\varphi_1\right)\sin\left(\varphi_2\right)\right)\dot{\varphi}_2^2+\left(l_1^2 m_1+l_1^2 m_2\right)\ddot{\varphi}_1+\left(l_1 l_2 m_2\cos\left(\varphi_1\right)\cos\left(\varphi_2\right)+l_1 l_2 m_2\sin\right.$$

[9]:
```
showmath(eq2)
```

[9]:

$$l_2^2 m_2\ddot{\varphi}_2-\left(l_1 l_2 m_2\dot{\varphi}_1^2\sin\left(\varphi_1\right)-l_1 l_2 m_2\ddot{\varphi}_1\cos\left(\varphi_1\right)\right)\cos\left(\varphi_2\right)+\left(l_1 l_2 m_2\dot{\varphi}_1^2\cos\left(\varphi_1\right)+l_1 l_2 m_2\ddot{\varphi}_1\sin\left(\varphi_1\right)+gl_2 m_2\right)\sin$$

[10]:
```
sol =  solve([eq1,eq2],[phi1dd,phi2dd])[0]
```

[11]:
```
showmath(sol)
```

[11]:

$$\left[\ddot{\varphi}_1=\frac{l_1 m_2\dot{\varphi}_1^2\cos\left(\varphi_1\right)\cos\left(\varphi_2\right)^2\sin\left(\varphi_1\right)-\left(l_1 m_2\dot{\varphi}_1^2\cos\left(\varphi_1\right)+gm_2\right)\sin\left(\varphi_1\right)\sin\left(\varphi_2\right)^2+\left(l_2 m_2\cos\left(\varphi_2\right)\sin\left(\varphi_1\right)\right.}{l_1 m_2\cos\left(\varphi_1\right)^2\cos\left(\varphi_2\right)^2+2\,l_1 m_2\cos\left(\varphi_1\right)c}\right.$$

[12]:
```
showmath( sol[0].rhs().denominator() )
```

[12]:

$$l_1 m_2\cos\left(\varphi_1\right)^2\cos\left(\varphi_2\right)^2+2\,l_1 m_2\cos\left(\varphi_1\right)\cos\left(\varphi_2\right)\sin\left(\varphi_1\right)\sin\left(\varphi_2\right)+l_1 m_2\sin\left(\varphi_1\right)^2\sin\left(\varphi_2\right)^2-l_1 m_1-l_1 m_2$$

[13]:
```
(l1*(2*m1+m2-m2*cos(2*phi1-2*phi2))).expand_trig().expand_trig().expand().show()
```

```
-l1*m2*cos(phi1)^2*cos(phi2)^2 + l1*m2*cos(phi2)^2*sin(phi1)^2 - 4*l1*m2*cos(phi1)*cos(phi2)
```

[14]:
```
bool ( -2*sol[0].rhs().denominator()==(l1*(2*m1+m2-m2*cos(2*phi1-2*phi2))).
↪expand_trig().expand_trig().expand() )
```

[14]:
```
True
```

Since the "textbook" solution contains a slightly different form, let's check if we have these formulas:

$$T(\varphi_1, \varphi_2, \dot{\varphi}_1, \dot{\varphi}_2) = \frac{m_1}{2} l_1^2 \dot{\varphi}_1^2 + \frac{m_2}{2} \left(l_1^2 \dot{\varphi}_1^2 + l_2^2 \dot{\varphi}_2^2 + 2 l_1 l_2 \dot{\varphi}_1 \dot{\varphi}_2 \cos(\varphi_1 - \varphi_2) \right)$$

$$V(\varphi_1, \varphi_2) = -(m_1 + m_2) g l_1 \cos \varphi_1 - m_2 g l_2 \cos \varphi_2$$

$$m_2 l_2 \ddot{\varphi}_2 \cos(\varphi_1 - \varphi_2) + (m_1 + m_2) l_1 \ddot{\varphi}_1 + m_2 l_2 \dot{\varphi}_2^2 \sin(\varphi_1 - \varphi_2) + (m_1 + m_2) g \sin \varphi_1 = 0$$

$$l_2 \ddot{\varphi}_2 + l_1 \ddot{\varphi}_1 \cos(\varphi_1 - \varphi_2) - l_1 \dot{\varphi}_1^2 \sin(\varphi_1 - \varphi_2) + g \sin \varphi_2 = 0$$

[15]:
```
rown_wiki = [m2*l2*cos(phi1-phi2)*phi2dd+(m1+m2)*l1*phi1dd+m2*l2*phi2d^2 *␣
↪sin(phi1-phi2)+ (m1+m2)*g*sin(phi1),\
l2*phi2dd+l1*cos(phi1-phi2)*phi1dd-l1*phi1d^2*sin(phi1-phi2)+g*sin(phi2)]
```

[16]:
```
showmath(rown_wiki[0])
```

[16]:

$$l_2 m_2 \dot{\varphi}_2^2 \sin(\varphi_1 - \varphi_2) + l_2 m_2 \ddot{\varphi}_2 \cos(\varphi_1 - \varphi_2) + l_1 (m_1 + m_2) \ddot{\varphi}_1 + g (m_1 + m_2) \sin(\varphi_1)$$

[17]:
```
showmath(rown_wiki[1])
```

[17]:

$$-l_1 \dot{\varphi}_1^2 \sin(\varphi_1 - \varphi_2) + l_1 \ddot{\varphi}_1 \cos(\varphi_1 - \varphi_2) + l_2 \ddot{\varphi}_2 + g \sin(\varphi_2)$$

[18]:
```
rown_wiki[0].show()
```

```
l2*m2*phi2d^2*sin(phi1 - phi2) + l2*m2*phi2dd*cos(phi1 - phi2) + l1*(m1 + m2)*phi1dd + g*(m
```

[19]:
```
(eq1/l1).reduce_trig().show()
```

```
-l2*m2*phi2d^2*sin(-phi1 + phi2) + l2*m2*phi2dd*cos(-phi1 + phi2) + l1*m1*phi1dd + l1*m2*phi
```

[20]:
```
rown_wiki[0].show()
```

```
l2*m2*phi2d^2*sin(phi1 - phi2) + l2*m2*phi2dd*cos(phi1 - phi2) + l1*(m1 + m2)*phi1dd + g*(m
```

[21]:
```
bool((eq1/l1) == rown_wiki[0] )
```

[21]:
```
True
```

[22]:
```
(eq2/l2/m2).reduce_trig().show()
rown_wiki[1].show()
```

```
l1*phi1d^2*sin(-phi1 + phi2) + l1*phi1dd*cos(-phi1 + phi2) + l2*phi2dd + g*sin(phi2)
```

```
-l1*phi1d^2*sin(phi1 - phi2) + l1*phi1dd*cos(phi1 - phi2) + l2*phi2dd + g*sin(phi2)
```

[23]:
```
bool((eq2/l2/m2) == rown_wiki[1] )
```

[23]:
```
True
```

11.1 Euler -Langrange

[24]:
```
Ekin = 1/2*(m1*x1.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2+\
m1*y1.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2+\
m2*x2.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2+\
m2*y2.subs(x2u).subs(to_fun).diff(t).subs(to_var)^2 )

Epot = m1*g*y1.subs(x2u)+m2*g*y2.subs(x2u)
```

[25]:
```
showmath( Epot.collect(cos(phi1)) )
```

[25]:

$$-gl_2m_2\cos\left(\varphi_2\right)-\left(gl_1m_1+gl_1m_2\right)\cos\left(\varphi_1\right)$$

[26]:
```
showmath( Ekin.trig_simplify() )
```

[26]:

$$\frac{1}{2}\,l_2^2m_2\dot{\varphi}_2^2+\frac{1}{2}\left(l_1^2m_1+l_1^2m_2\right)\dot{\varphi}_1^2+\left(l_1l_2m_2\dot{\varphi}_1\cos\left(\varphi_1\right)\cos\left(\varphi_2\right)+l_1l_2m_2\dot{\varphi}_1\sin\left(\varphi_1\right)\sin\left(\varphi_2\right)\right)\dot{\varphi}_2$$

[27]:
```
L = Ekin - Epot
```

[28]:
```
EL1 = L.diff(phi1d).subs(to_fun).diff(t).subs(to_var) - L.diff(phi1)
EL2 = L.diff(phi2d).subs(to_fun).diff(t).subs(to_var) - L.diff(phi2)
```

[29]:
```
EL1 = (EL1/l1).trig_reduce()
EL2 = (EL2/l2).trig_reduce()
showmath(EL1)
```

[29]:

$$-l_2 m_2 \dot{\varphi}_2^2 \sin\left(-\varphi_1+\varphi_2\right)+l_2 m_2 \ddot{\varphi}_2 \cos\left(-\varphi_1+\varphi_2\right)+l_1 m_1 \ddot{\varphi}_1+l_1 m_2 \ddot{\varphi}_1+g m_1 \sin\left(\varphi_1\right)+g m_2 \sin\left(\varphi_1\right)$$

[30]:
```
sol =  solve([EL1,EL2],[phi1dd,phi2dd])[0]
show(sol)
```

```
[phi1dd == -(l1*m2*phi1d^2*cos(-phi1 + phi2)*sin(-phi1 + phi2) + l2*m2*phi2d^2*sin(-phi1 + p
 phi2dd == (l2*m2*phi2d^2*cos(-phi1 + phi2)*sin(-phi1 + phi2) - (g*m1 + g*m2)*cos(-phi1 + ph
```

[31]:
```
expr = sol[0].rhs()
```

[32]:
```
for ex_ in expr.factor().numerator().operands():
    show(ex_)
```

```
-l1*m2*phi1d^2*cos(-phi1 + phi2)*sin(-phi1 + phi2)
```

```
-l2*m2*phi2d^2*sin(-phi1 + phi2)
```

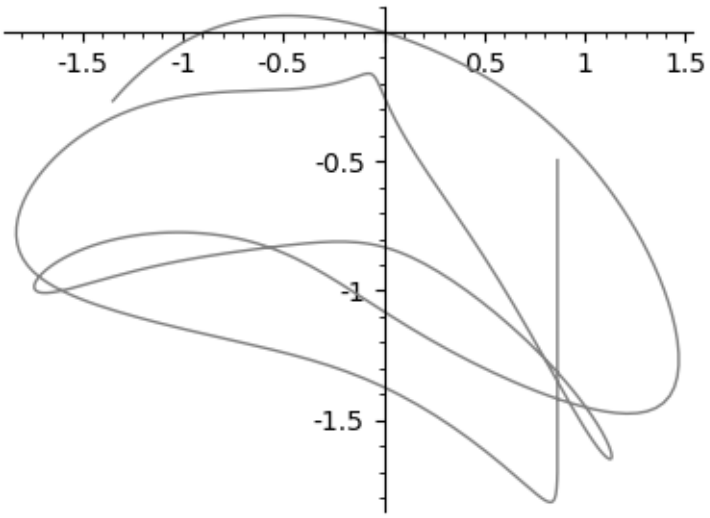
```
-g*m2*cos(-phi1 + phi2)*sin(phi2)
```

```
g*m1*sin(phi1)
```

```
g*m2*sin(phi1)
```

11.2 Numerical analysis

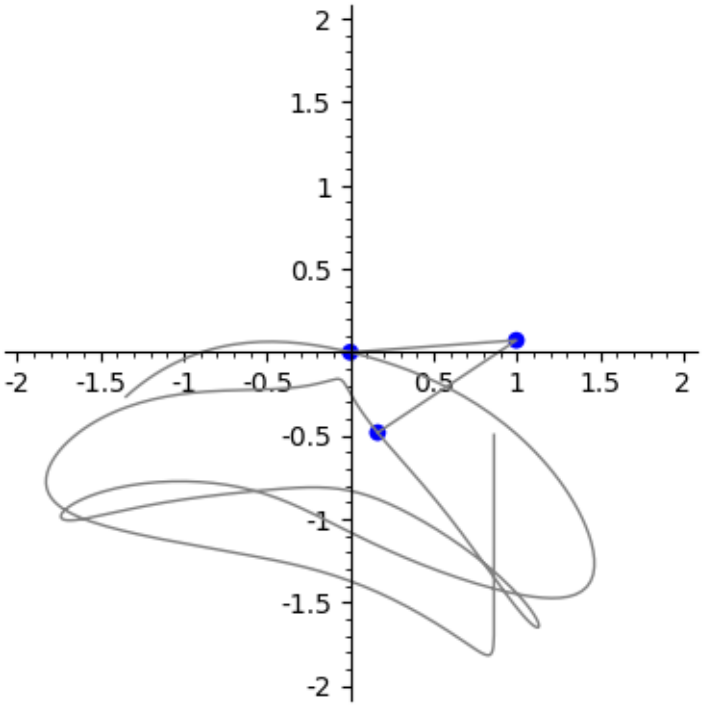
[33]:
```
import numpy as np

ode =  [phi1d,phi2d]+[sol[0].rhs(),sol[1].rhs()]
ode = map(lambda x:x.subs({l1:1,l2:1,m1:1,m2:1,g:9.81}),ode)

times = srange(0,5,.01)
numsol = desolve_odeint(ode,[2.1,0,0,0],times,[phi1,phi2,phi1d,phi2d])
p = line ( zip(np.sin(numsol[:,0])+np.sin(numsol[:,1]),\
               -np.cos(numsol[:,0])-np.cos(numsol[:,1])), color='gray' )
p.show(figsize=4)
```



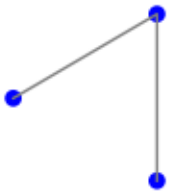
[34]:
```
def plot_dp(f1,f2,pars):
    mass1 = vector([x1,y1]).subs(x2u).subs(pars).subs({phi1:f1,phi2:f2})
    mass2 = vector([x2,y2]).subs(x2u).subs(pars).subs({phi1:f1,phi2:f2})
    plt = point([(0,0),mass1],aspect_ratio=1,size=40)
    plt += point(mass2,xmin=-2,xmax=2,ymin=-2,ymax=2,size=40)
    plt += line([(0,0),mass1,mass2],color='gray')

    return plt
```

[35]:
```
plot_dp(numsol[213,0],numsol[213,1],{l1:1,l2:1})+p
```

[35]:



[36]:
```
@interact
def _(ith=slider(0,numsol.shape[0]-1)):
    f1,f2 = numsol[ith,:2]
    plot_dp(f1,f2,{l1:1,l2:1}).show(axes=False)
```



11.3 Triple pendulum

We can easily generalize our symbolic scheme to cases where the number of points is arbitrary.

[37]:
```
N = 4
var('t g')
for i in range(1,1+N):
    var('l%d m%d'%(i,i))
```

[38]:
```
xy_names = [('x%d'%i,'x_%d'%i) for i in range(1,1+N)]
xy_names += [('y%d'%i,'y_%d'%i) for i in range(1,1+N)]
uv_names = [ ('phi%d'%i,'\\varphi_%d'%i) for i in range(1,1+N)]
```

[39]:
```
load('cas_utils.sage')
```

[40]:
```
to_fun, to_var = make_symbols(xy_names,uv_names)
```

```
phi1  :: has been  processed
phi2  :: has been  processed
phi3  :: has been  processed
phi4  :: has been  processed
x1  :: has been  processed
```

```
x2  :: has been  processed
x3  :: has been  processed
x4  :: has been  processed
y1  :: has been  processed
y2  :: has been  processed
y3  :: has been  processed
y4  :: has been  processed
```

[41]:
```
ls = [vars()['l%d'%i] for i in range(1,1+N)]
xs = [vars()['x%d'%i] for i in range(1,1+N)]
ys = [vars()['y%d'%i] for i in range(1,1+N)]
phis = [vars()['phi%d'%i] for i in range(1,1+N)]
ms = [vars()['m%d'%i] for i in range(1,1+N)]

show(phis)
```

```
[phi1, phi2, phi3, phi4]
```

[42]:
```
x2u = {x1:l1*sin(phi1),\
       y1:-l1*cos(phi1) }

for x_prev,x_,y_prev,y_,l_,phi_ in zip(xs[:-1],xs[1:],ys[:-1],ys[1:],ls[1:
 ↪],phis[1:]):
    x2u[x_] = x2u[x_prev] + l_*sin(phi_)
    x2u[y_] = x2u[y_prev] - l_*cos(phi_)
```

[43]:
```
transform_virtual_displacements(xy_names,uv_names,verbose=False)
```

[43]:
```
[dphi1*l1*cos(phi1),
 dphi1*l1*cos(phi1) + dphi2*l2*cos(phi2),
 dphi1*l1*cos(phi1) + dphi2*l2*cos(phi2) + dphi3*l3*cos(phi3),
 dphi1*l1*cos(phi1) + dphi2*l2*cos(phi2) + dphi3*l3*cos(phi3) +
dphi4*l4*cos(phi4),
 dphi1*l1*sin(phi1),
 dphi1*l1*sin(phi1) + dphi2*l2*sin(phi2),
 dphi1*l1*sin(phi1) + dphi2*l2*sin(phi2) + dphi3*l3*sin(phi3),
 dphi1*l1*sin(phi1) + dphi2*l2*sin(phi2) + dphi3*l3*sin(phi3) +
dphi4*l4*sin(phi4)]
```

[44]:
```
dxs = [vars()['dx%d_polar'%i] for i in range(1,1+N) ]
dys = [vars()['dy%d_polar'%i] for i in range(1,1+N) ]
```

[45]:
```
dAlemb = sum( (m_*x_.subs(x2u).subs(to_fun).diff(t,2))*dx_ for m_,x_,dx_ in␣
 ↪zip(ms,xs,dxs) )
dAlemb += sum( (m_*x_.subs(x2u).subs(to_fun).diff(t,2) + m_*g)*dx_ for m_,x_,dx_␣
 ↪in zip(ms,ys,dys) )
```

```
dAlemb = dAlemb.subs(to_var)
```

[46]:
```
#showmath(dAlemb)
```

[47]:
```
dphis = [vars()['dphi%d'%i] for i in range(1,1+N) ]
```

[48]:
```
eqs = [dAlemb.expand().coefficient(dphi_).trig_simplify() for dphi_ in dphis]
```

[49]:
```
showmath(eqs[2])
```

[49]:

$$(l_3 l_4 m_4 \cos(\varphi_4)\sin(\varphi_3) - l_3 l_4 m_4 \cos(\varphi_3)\sin(\varphi_4))\dot{\varphi}_4^2 + (l_3^2 m_3 + l_3^2 m_4)\ddot{\varphi}_3 + (l_3 l_4 m_4 \cos(\varphi_3)\cos(\varphi_4) + l_3 l_4 m_4 \sin$$

[50]:
```
phidds = [vars()['phi%ddd'%i] for i in range(1,1+N) ]
```

[51]:
```
sol =  solve(eqs,phidds)[0]
```

[52]:
```
len(sol[1].rhs().trig_reduce().operands())
```

[52]:
```
62
```

[53]:
```
pars= {m_:1 for m_ in ms}

for i,l_ in enumerate(ls):
    pars[l_]  = 1/ (i+1)
pars[g] = 9.81
```

[ ]:
```

```

[54]:
```
phidds
```

[54]:
```
[phi1dd, phi2dd, phi3dd, phi4dd]
```

[55]:
```
import numpy as np

phids = [vars()['phi%dd'%i] for i in range(1,1+N) ]

ode =  phids + [sol_.rhs() for sol_ in sol]
ode = map(lambda x:x.subs(pars),ode)

times = srange(0,5,.01)

ics = [0]*(N*2)
ics[-1] = 33.01
ics[:N]= [0*pi.n()]*N
```

```
numsol = desolve_odeint(ode,ics,times, phis + phids)
```

[ ]:

[56]:
```
phi_subs = lambda ith: {phi_:numval_ for phi_,numval_ in zip(phis,numsol[ith,:N])}

@interact
def _(ith=slider(0,numsol.shape[0]-1,step_size=10)):
    xnum = [0]+[x_.subs(x2u).subs(pars).subs(phi_subs(ith)) for x_ in xs]
    ynum = [0]+[x_.subs(x2u).subs(pars).subs(phi_subs(ith)) for x_ in ys]
    plt = line(zip(xnum,ynum),\
        xmin=-N,xmax=N,ymin=-N,ymax=N,marker='o')
    plt.show(axes=False,figsize=5,aspect_ratio=1)
```

12 Spherical pendulum

[1]:
```
import numpy as np
```

[2]:
```
var('t')
var('l g')
xy_names = [('x','x'),('y','y'),('z','z')]
uv_names = [('r','r'),('phi',r'\phi'),('theta',r'\theta')]

load('cas_utils.sage')
```

[3]:
```
to_fun, to_var = make_symbols(xy_names,uv_names)
```

```
r  :: has been  processed
phi  :: has been  processed
theta  :: has been  processed
x  :: has been  processed
y  :: has been  processed
z  :: has been  processed
```

[4]:
```
x2u = {x:l*sin(theta)*cos(phi),y: l*sin(theta)*sin(phi),z: l*cos(theta)}
```

[5]:
```
transform_virtual_displacements(xy_names,uv_names,suffix='_uv')
```

[5]:
```
[dtheta*l*cos(phi)*cos(theta) - dphi*l*sin(phi)*sin(theta),
 dtheta*l*cos(theta)*sin(phi) + dphi*l*cos(phi)*sin(theta),
 -dtheta*l*sin(theta)]
```

[6]:
```
dx_uv
```

[6]:
```
dtheta*l*cos(phi)*cos(theta) - dphi*l*sin(phi)*sin(theta)
```

[ ]:
```

```

[7]:
```
dAlemb = (x.subs(x2u).subs(to_fun).diff(t,2))*dx_uv + \
         (y.subs(x2u).subs(to_fun).diff(t,2))*dy_uv + \
         (z.subs(x2u).subs(to_fun).diff(t,2)+g)*dz_uv
dAlemb = dAlemb.subs(to_var)
```

[8]:
```
show(dAlemb)
```

```
(l*thetad^2*cos(theta) + l*thetadd*sin(theta) - g)*dtheta*l*sin(theta) - (2*l*phid*thetad*co
```

[9]:
```
sol = solve(\
    [dAlemb.expand().coefficient(dtheta).trig_simplify(),\
      dAlemb.expand().coefficient(dphi).trig_simplify()],\
            [phidd,thetadd])[0]
```

[10]:
```
show(sol)
```

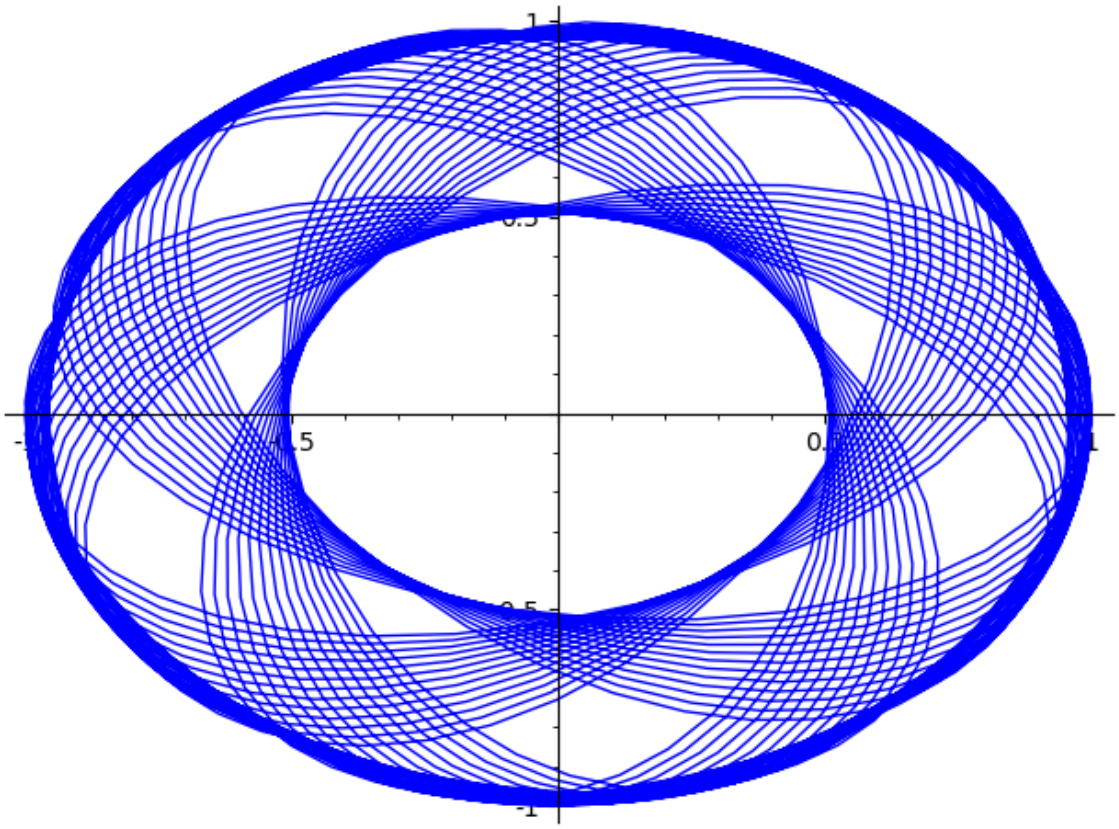
```
[phidd == -2*phid*thetad*cos(theta)/sin(theta),
 thetadd == (l*phid^2*cos(theta) + g)*sin(theta)/l]
```

[11]:
```
ode = [phid,thetad]+[s.rhs() for s in sol]
show(ode )
```

```
[phid,
 thetad,
 -2*phid*thetad*cos(theta)/sin(theta),
 (l*phid^2*cos(theta) + g)*sin(theta)/l]
```

[12]:
```
ode=map(lambda x:x.subs({l:1,g:1}),ode)
show(ode)
times = srange(0,237,.1)
numsol=desolve_odeint(ode,[0,pi/2-0.3,1,0],times,[phi,theta,phid,thetad])
#p=point(zip(np.fmod(numsol[:,0],(2*pi).n())-pi,numsol[:
 ↪,1]),figsize=5)#,aspect_ratio=1)
p = line ( zip(np.sin(numsol[:,1])*np.cos(numsol[:,0]),np.sin(numsol[:,0])*np.
 ↪sin(numsol[:,1])) )
p.show()
```

```
[phid,
 thetad,
 -2*phid*thetad*cos(theta)/sin(theta),
 (phid^2*cos(theta) + 1)*sin(theta)]
```

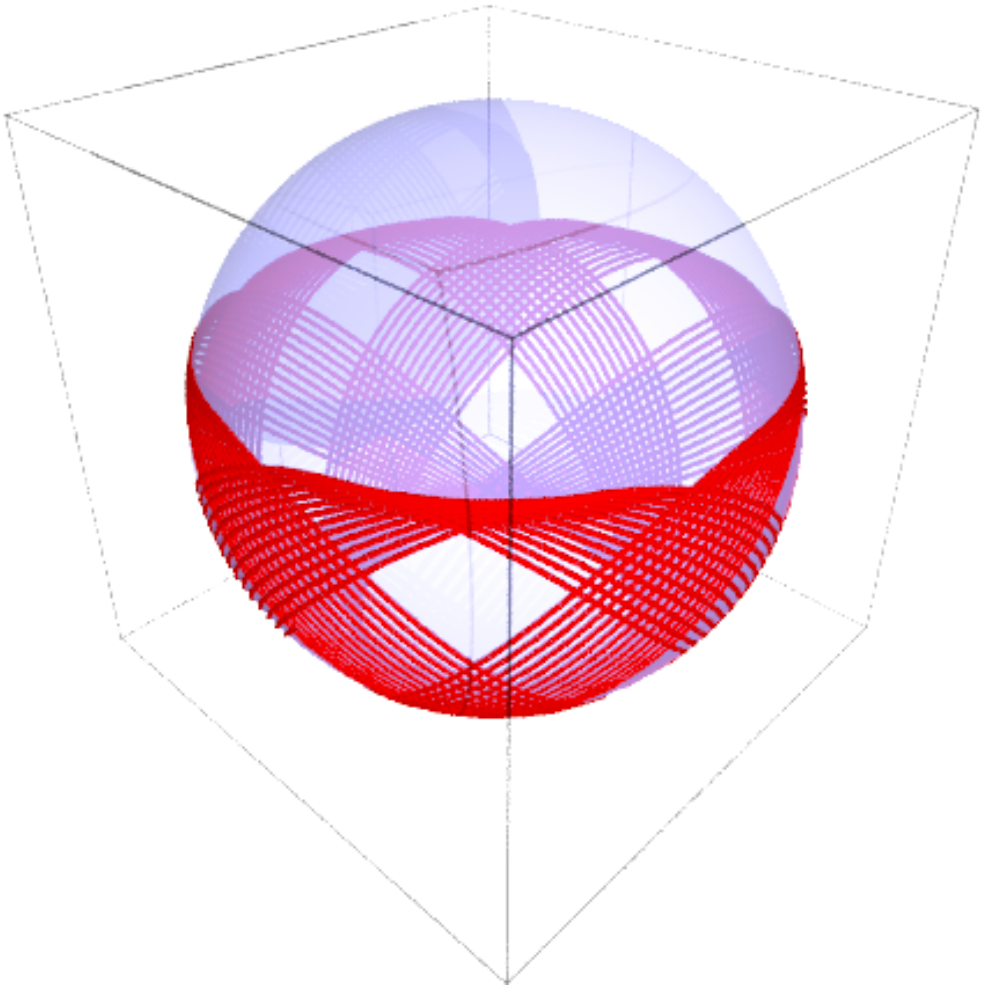


$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\phi = \operatorname{arctg} \frac{y}{x}$$

$$\theta = \arcsin \frac{z}{r}$$

[13]:
```
p3d = line3d( zip(np.sin(numsol[:,1])*np.cos(numsol[:,0]),np.sin(numsol[:,0])*np.
 ↪sin(numsol[:,1]),np.cos(numsol[:,1])),thickness=2,color='red')
p3d += sphere(opacity=0.5)
p3d.show(viewer='tachyon')
```



13 Paraboloidal pendulum

13.1 System definition

[1]:
```
import numpy as np
```

[2]:
```
var('t')
var('l g')
xy_names = [('x','x'),('y','y'),('z','z')]
load('cas_utils.sage')
to_fun, to_var = make_symbols(xy_names)
xy = [vars()[v] for v,lv in xy_names]
```

```
x  :: has been  processed
y  :: has been  processed
z  :: has been  processed
```

[3]:
```
dAlemb = (x.subs(to_fun).diff(t,2))*dx + \
         (y.subs(to_fun).diff(t,2))*dy + \
         (z.subs(to_fun).diff(t,2)+g)*dz
dAlemb = dAlemb.subs(to_var)
```

[4]:
```
showmath(dAlemb)
```

[4]:

$$\delta z(g+\ddot{z})+\delta x\ddot{x}+\delta y\ddot{y}$$

[5]:
```
f = 1/2*(x^2+y^2)-z
dxy = [vars()['d'+repr(zm)] for zm in xy]

constr =sum([dzm*f.diff(zm) for zm,dzm in zip(xy,dxy)])
showmath(constr)
```

[5]:

$$\delta xx+\delta yy-\delta z$$

[6]:
```
eq1=(dAlemb.subs(constr.solve(dz)[0])*x).expand().coefficient(dx).subs(to_var)
eq2=(dAlemb.subs(constr.solve(dz)[0])*x).expand().coefficient(dy).subs(to_var)

showmath([eq1,eq2])
```

[6]:

$$\left[gx^2+x^2\ddot{z}+x\ddot{x},gxy+xy\ddot{z}+x\ddot{y}\right]$$

[7]:
```
sol = solve([f.subs(to_fun).diff(2).subs(to_var),eq1,eq2],[xdd,ydd,zdd])[0][:2]
```

[8]:
```
showmath(sol)
```

[8]:

$$\left[\ddot{x} = -\frac{x\dot{x}^2 + x\dot{y}^2 + gx}{x^2 + y^2 + 1}, \ddot{y} = -\frac{y\dot{y}^2 + (\dot{x}^2 + g)y}{x^2 + y^2 + 1}\right]$$

[9]:
```
ode = [xd,yd] + [s_.rhs() for s_ in sol]
```

13.2 Numerical analysis

[10]:
```
ode = map(lambda x:x.subs({l:1,g:1}),ode)
showmath(ode)
```

[10]:

$$\left[\dot{x}, \dot{y}, -\frac{x\dot{x}^2 + x\dot{y}^2 + x}{x^2 + y^2 + 1}, -\frac{y\dot{y}^2 + (\dot{x}^2 + 1)y}{x^2 + y^2 + 1}\right]$$

[11]:
```
times = srange(0,237,.1)
```

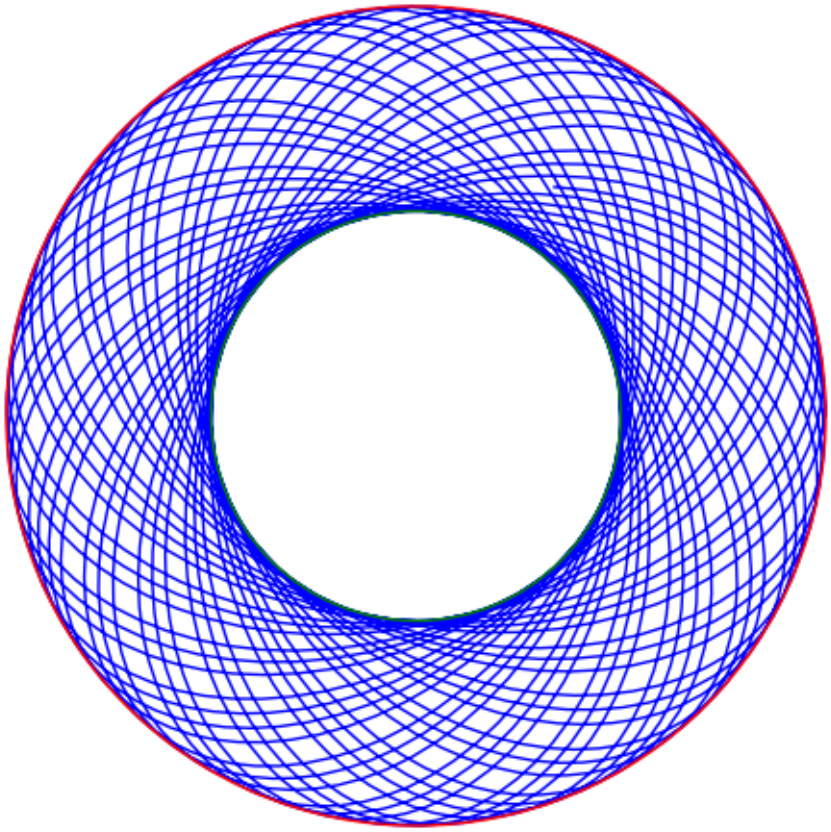
[12]:
```
z_prime = (1/2*(x^2+y^2)).subs(to_fun).diff(t).subs(to_var)
z_prime.show()
```

```
x*xd + y*yd
```

[13]:
```
Ekin = (1/2*(xd^2+yd^2+z_prime^2))
```

[ ]:
```

```

[14]:
```
numsol = desolve_odeint(ode,[1.0,0,0.,.5],times,[x,y,xd,yd])
p = line ( zip(numsol[:,0],numsol[:,1]) )
Epot = 1/2*(numsol[:,0]^2+numsol[:,1]^2)
p += circle( (0,0), sqrt(np.max(2*Epot)),color='red' )
p += circle( (0,0), sqrt(np.min(2*Epot)),color='green' )
p.show(aspect_ratio=1,axes=False)
```



[15]:
```
Ekin_num = [Ekin.subs({x:d[0],y:d[1],xd:d[2],yd:d[3]}) for d in numsol]
Epot_num  = 1/2*(numsol[:,0]^2+numsol[:,1]^2) # g=1!
```

[16]:
```
Ekin_num + Epot_num
```

[16]:
```
array([0.625, 0.6249999943305325, 0.6249999979623393, ...,
       0.625004240168609, 0.6250042526868687, 0.6250042649374952],
      dtype=object)
```

13.3 Angular momentum

We can calculate symbolically angular momentum and chech if its z-component is conserved in time.

[17]:
```
var('x y', domain='real')

#e_r = vector([x,y,0])
#v = vector([xd,yd,0])
```

```
e_r = vector([x,y,1/2*(x^2+y^2)])
v = vector([xd,yd,z_prime])
```

[18]:
```
p = e_r.cross_product(v)
```

[19]:
```
showmath(p[0].full_simplify())
```

[19]:

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

[20]:
```
Ekin_num = [Ekin.subs({x:d[0],y:d[1],xd:d[2],yd:d[3]}) for d in numsol]
```

[21]:
```
P_num = [(p[2]).subs({x:d[0],y:d[1],xd:d[2],yd:d[3]}) for d in numsol]
```

[22]:
```
P_num[0]
```

[22]:
```
0.5
```

[23]:
```
Ekin.subs( (p[2]*x*y).expand().solve(xd)[0]*x).expand().show()
```

```
x^2*yd^2 + 1/2*x^4*yd^2/y^2 + 1/2*y^2*yd^2 + 1/2*xd^2 + 1/2*yd^2
```

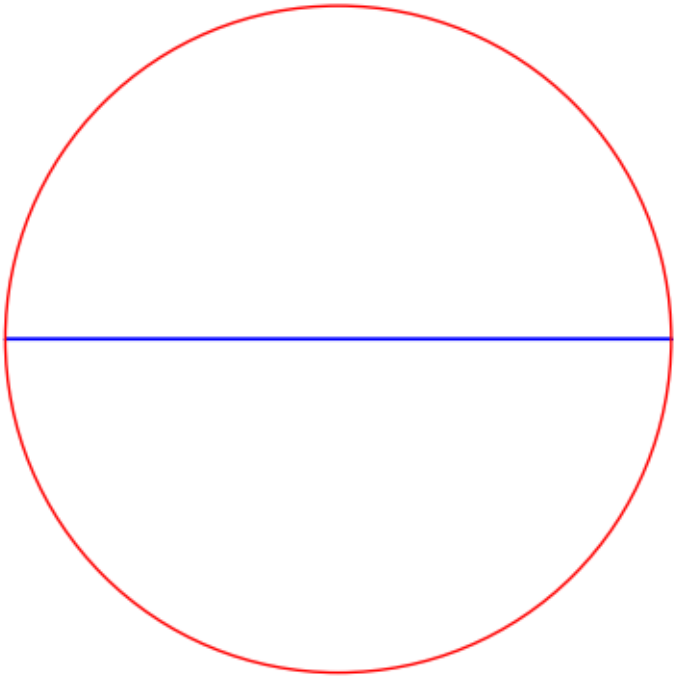
[24]:
```
Ekin.show()
```

```
1/2*(x*xd + y*yd)^2 + 1/2*xd^2 + 1/2*yd^2
```

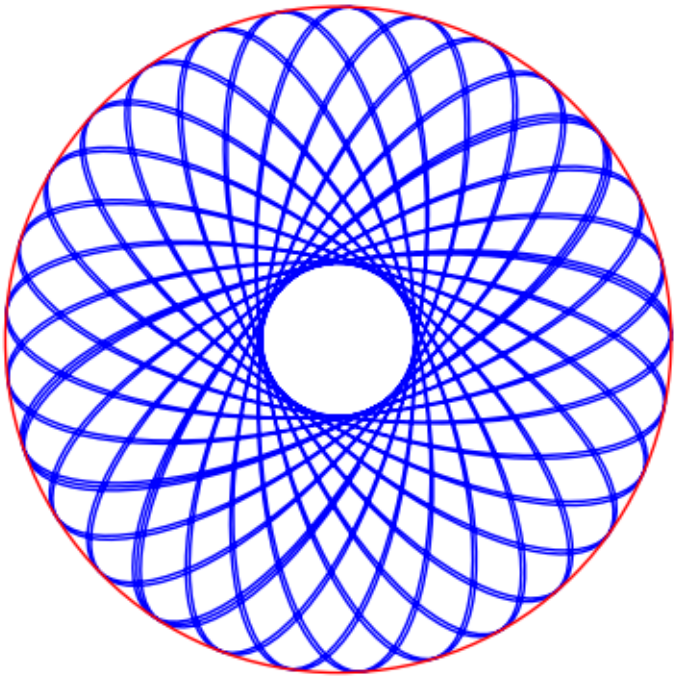
[25]:
```
P_num[:4]
```

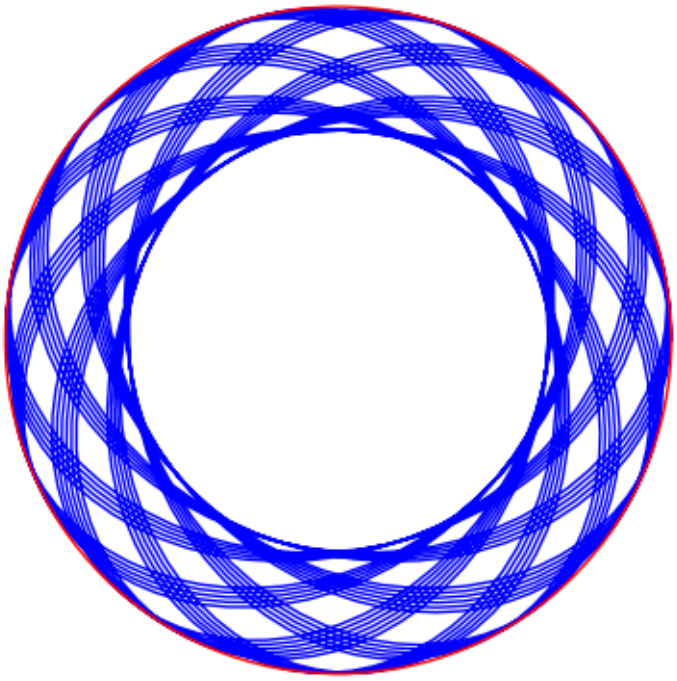
[25]:
```
[0.5, 0.4999999909879266, 0.49999999698576963, 0.4999999949420133]
```

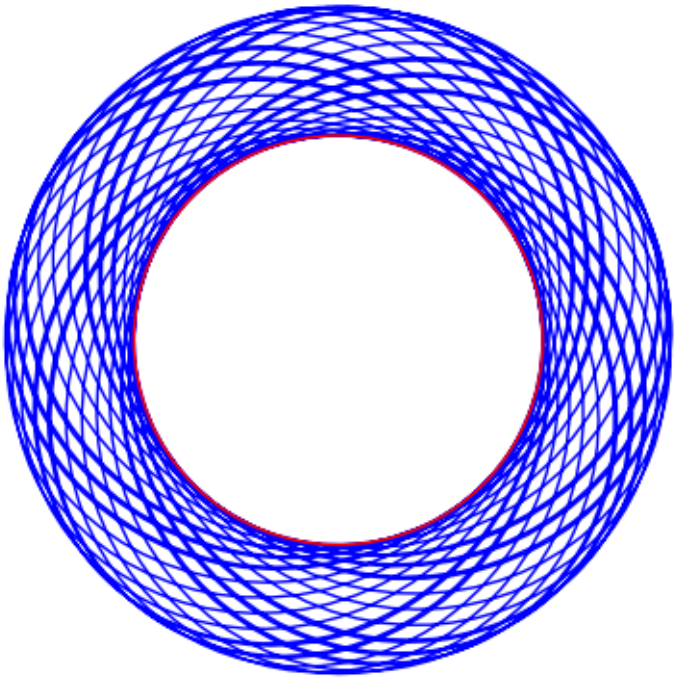
[26]:
```
@interact
def plot2d_traj(v0=slider(0,3,0.001)):
    numsol = desolve_odeint(ode,[1,0,0,v0],times,[x,y,xd,yd])
    p = line( zip(numsol[:,0],numsol[:,1]), aspect_ratio=1)
    p += circle( (0,0),1,color='red')
    p.show(axes=False)
```



[27]:
```
plot2d_traj(v0=0.23),plot2d_traj(v0=0.63),plot2d_traj(v0=1.63),
```

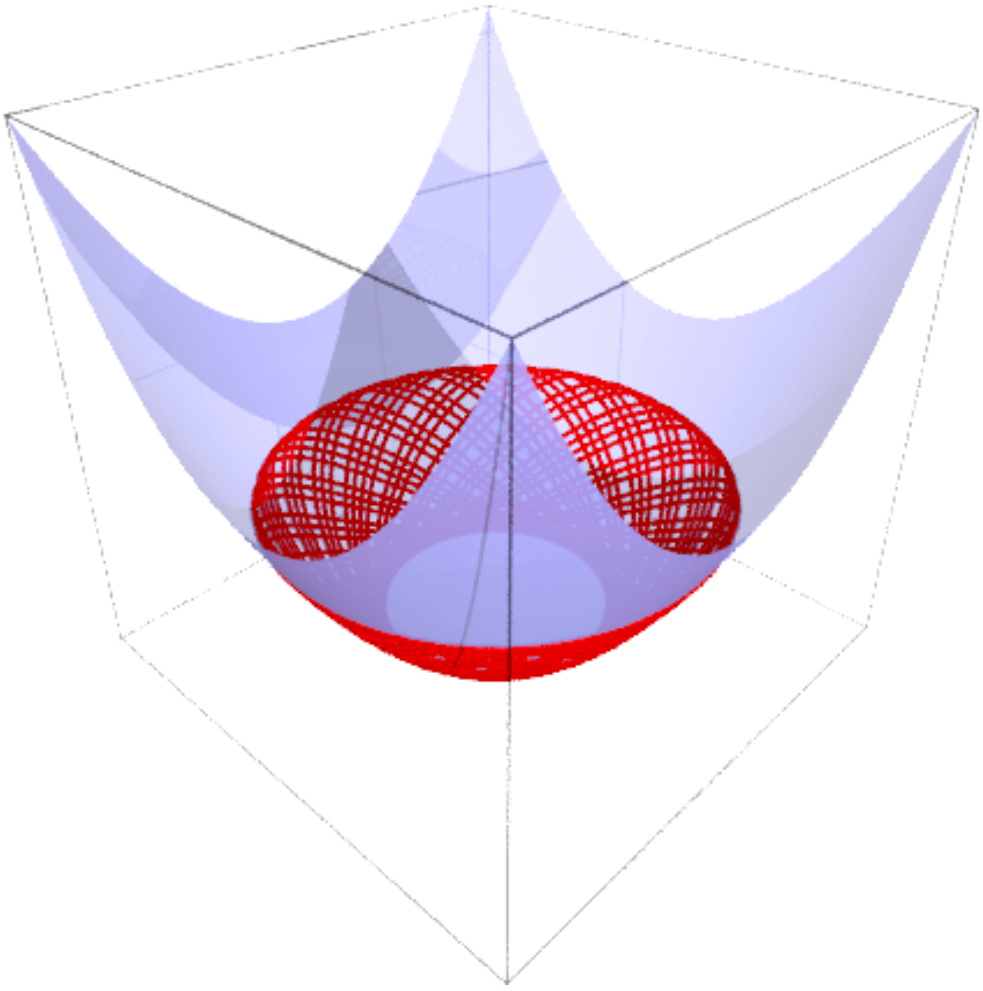






[27]:
```
(None, None, None)
```

[28]:
```
p3d = line3d( zip(numsol[:,0],numsol[:,1],(1/2*(numsol[:,0]**2+numsol[:
↪,1]**2))),thickness=2,color='red')
p3d += plot3d(1/2*(x^2+y^2),(x,-1.2,1.2),(y,-1.2,1.2),opacity=0.8)
p3d.show(viewer='tachyon')
```



14 Point particle on the cone

Consider the movement of the material point on the surface of the cone.

[1]:
```
var('t')
var('a g')
xy_wsp = [('x','x'),('y','y'),('z','z')]
uv_wsp = [('r','r'),('phi','\phi'),('z','z')]

for v,lv in uv_wsp+xy_wsp:
    var("%s"%v,latex_name=r'%s'%lv)
    vars()[v.capitalize()] = function(v.capitalize())(t)
    var("%sdd"%v,latex_name=r'\ddot %s'%lv)
    var("%sd"%v,latex_name=r'\dot %s'%lv)
    var("d%s"%v,latex_name=r'\delta %s'%lv)


xy = [vars()[v] for v,lv in xy_wsp]
uv = [vars()[v] for v,lv in uv_wsp]

to_fun=dict()
for v,lv in xy_wsp+uv_wsp:
    to_fun[vars()[v]]=vars()[v.capitalize()]
    to_fun[vars()[v+"d"]]=vars()[v.capitalize()].diff()
    to_fun[vars()[v+"dd"]]=vars()[v.capitalize()].diff(2)

to_var = dict((v,k) for k,v in to_fun.items())
```

[2]:
```
dAlemb = (X.diff(t,2))*dx +(Y.diff(t,2))*dy+ (Z.diff(t,2)+g)*dz
f = x^2 + y^2 - tan(a)^2*z^2

dxy = [vars()['d'+repr(zm)] for zm in xy]
constr =sum([dzm*f.diff(zm) for zm,dzm in zip(xy,dxy)])
show(f)
show(constr)
```

$$-z^2\tan(a)^2 + x^2 + y^2$$

$$-2*dz*z*\tan(a)^2 + 2*dx*x + 2*dy*y$$

[3]:
```
f.solve(z)
```

[3]:
```
[z == -sqrt(x^2 + y^2)/tan(a), z == sqrt(x^2 + y^2)/tan(a)]
```

[4]:
```
dAlemb_xy = dAlemb.subs(constr.solve(dz)[0]).subs(f.solve(z)[1])
```

[5]:
```
show(dAlemb_xy.subs(to_var).coeff(dx))
show(dAlemb_xy.subs(to_var).coeff(dy))
```

```
/usr/local/lib/SageMath/local/lib/python2.7/site-
packages/sage/repl/ipython_kernel/__main__.py:1: DeprecationWarning: coeff is
deprecated. Please use coefficient instead.
See http://trac.sagemath.org/17438 for details.
  from ipykernel.kernelapp import IPKernelApp
```

```
xdd + (g + zdd)*x/(sqrt(x^2 + y^2)*tan(a))
```

```
/usr/local/lib/SageMath/local/lib/python2.7/site-
packages/sage/repl/ipython_kernel/__main__.py:2: DeprecationWarning: coeff is
deprecated. Please use coefficient instead.
See http://trac.sagemath.org/17438 for details.
  from sage.repl.ipython_kernel.kernel import SageKernel
```

```
ydd + (g + zdd)*y/(sqrt(x^2 + y^2)*tan(a))
```

[ ]:
```

```

Let's solve the problem in cylindrical coordinates. Because these coordinates are not consistent with constraints, but the cone has a particularly simple form in them. Therefore, we will have to perform the step of removing the dependent virtual offsets after moving to the cylindrical coordinate system.

[6]:
```
x2u = {x:r*cos(phi),y:r*sin(phi),z:z}
for w in xy:
    vars()['d'+repr(w)+'_polar']=sum([w.subs(x2u).diff(w2)*vars()['d'+repr(w2)]␣
 ↪for w2 in uv])
show(dx_polar)
show(dy_polar)
show(dz_polar)
```

```
-dphi*r*sin(phi) + dr*cos(phi)
```

```
dphi*r*cos(phi) + dr*sin(phi)
```

```
dz
```

First, we write the d'Alembert's rule in cylindrical coordinates:

[7]:
```
dAlemb = (x.subs(x2u).subs(to_fun).diff(t,2))*dx_polar + \
         (y.subs(x2u).subs(to_fun).diff(t,2))*dy_polar+\
         (z.subs(x2u).subs(to_fun).diff(t,2)+g)*dz_polar
dAlemb = dAlemb.subs(to_var)
show(dAlemb)
```

```
-(phid^2*r*sin(phi) - phidd*r*cos(phi) - 2*phid*rd*cos(phi) - rdd*sin(phi))*(dphi*r*cos(phi)
```

[8]:
```
f_uv  = f.subs(x2u).trig_simplify()
```

[9]:
```
duv = [vars()['d'+repr(zm)] for zm in uv]
constr =sum([dzm*f_uv.diff(zm) for zm,dzm in zip(uv,duv)])
```

[10]:
```
show(constr.solve(dz)[0])
```

```
dz == dr*r*cos(a)^2/(z*sin(a)^2)
```

[11]:
```
dAlemb.subs( constr.solve(dz)[0] )
```

[11]:
```
-(phid^2*r*sin(phi) - phidd*r*cos(phi) - 2*phid*rd*cos(phi) -
rdd*sin(phi))*(dphi*r*cos(phi) + dr*sin(phi)) + (phid^2*r*cos(phi) +
phidd*r*sin(phi) + 2*phid*rd*sin(phi) - rdd*cos(phi))*(dphi*r*sin(phi) -
dr*cos(phi)) + dr*(g + zdd)*r*cos(a)^2/(z*sin(a)^2)
```

[12]:
```
zdd_rdd = (f_uv.solve(z)[1]).subs(to_fun).diff(t,2).subs(to_var)
zdd_rdd
```

[12]:
```
zdd == rdd*cos(a)/sin(a)
```

[13]:
```
dAlemb_uv = dAlemb.subs( constr.solve(dz)[0] ).subs(f_uv.solve(z)[1]).
 ↪subs(zdd_rdd)
```

[14]:
```
r1 =  dAlemb_uv.coeff(dr)
r2 =  dAlemb_uv.coeff(dphi)

show(r1)
show( r2)
```

```
/usr/local/lib/SageMath/local/lib/python2.7/site-
packages/sage/repl/ipython_kernel/__main__.py:1: DeprecationWarning: coeff is
deprecated. Please use coefficient instead.
See http://trac.sagemath.org/17438 for details.
  from ipykernel.kernelapp import IPKernelApp
/usr/local/lib/SageMath/local/lib/python2.7/site-
```

```
packages/sage/repl/ipython_kernel/__main__.py:2: DeprecationWarning: coeff is
deprecated. Please use coefficient instead.
See http://trac.sagemath.org/17438 for details.
  from sage.repl.ipython_kernel.kernel import SageKernel
```

```
-(phid^2*r*cos(phi) + phidd*r*sin(phi) + 2*phid*rd*sin(phi) - rdd*cos(phi))*cos(phi) - (phi
```

```
-(phid^2*r*sin(phi) - phidd*r*cos(phi) - 2*phid*rd*cos(phi) - rdd*sin(phi))*r*cos(phi) + (ph
```

[15]:
```
table([rown.trig_simplify() for rown in solve([r1,r2],[rdd,phidd])[0]])
```

[15]:

rdd == phid^2*r*sin(a)^2 - g*cos(a)*sin(a)	phidd == -2*phid*rd/r

[ ]:

[ ]:

15 Paraglider flight mechanics in 2d

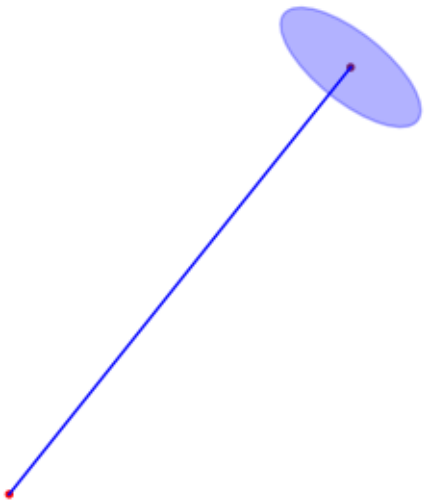
(m_1, x_1) pilot

(m_2, x_2) wing

F_x, F_y, aerodynamic force in the rest system

[1]:
```
def make_fig(x1,y1,x2,y2,CM=True,m1=1,m2=1):
    x1 = x1 - (m1*x1+m2*x2)/2
    y1 = y1 - (m1*y1+m2*y2)/2
    plts = []
    plts +=[ point((x1,y1),color='red')]
    plts +=[ point((x2,y2),color='brown')]
    plts +=[ line ([(x1,y1),(x2,y2)])]
    plts += [ellipse((x2,y2),1,.4,arctan2(y1-y2,x1-x2)-pi/2,fill=True,alpha=0.3)]
    plt = sum(plts)

    return plt
```

[2]:
```
plt = make_fig(1,2,3,4)
plt.show(figsize=4,aspect_ratio=1,axes=false)
```



[3]:
```
import numpy as np
var('t')
var('l g Fx Fy m1 m2')
xy_wsp = [('x1','x1'),('y1','y1'),('x2','x2'),('y2','y2')]

uv_wsp = [('phi','\phi'),('x','x'),('y','y')]

for v,lv in uv_wsp+xy_wsp:
    var("%s"%v,latex_name=r'%s'%lv)
    vars()[v.capitalize()] = function(v.capitalize())(t)
    var("%sdd"%v,latex_name=r'\ddot %s'%lv)
    var("%sd"%v,latex_name=r'\dot %s'%lv)
    var("d%s"%v,latex_name=r'\delta %s'%lv)

uv = [vars()[v] for v,lv in uv_wsp]

xy = [vars()[v] for v,lv in xy_wsp]
to_fun=dict()
for v,lv in uv_wsp:
    to_fun[vars()[v]]=vars()[v.capitalize()]
    to_fun[vars()[v+"d"]]=vars()[v.capitalize()].diff()
    to_fun[vars()[v+"dd"]]=vars()[v.capitalize()].diff(2)
to_var = dict((v,k) for k,v in to_fun.items())

x2u = {x2:x,y2:y, x1:x+l*sin(phi),y1:y-l*cos(phi)}

for w in xy:
    vars()['d'+repr(w)+'_polar']=sum([w.subs(x2u).diff(w2)*vars()['d'+repr(w2)]␣
 ↪for w2 in uv])
    #show(vars()['d'+repr(w)+'_polar'])

dAlemb = (m1*x1.subs(x2u).subs(to_fun).diff(t,2))*dx1_polar + \
         (m1*y1.subs(x2u).subs(to_fun).diff(t,2)+m1*g)*dy1_polar+\
         (m2*x2.subs(x2u).subs(to_fun).diff(t,2)-Fx)*dx2_polar + \
         (m2*y2.subs(x2u).subs(to_fun).diff(t,2)+m2*g-Fy)*dy2_polar
dAlemb = dAlemb.subs(to_var)
```

[4]:
```
show(dAlemb)
```

```
-(l*phid^2*sin(phi) - l*phidd*cos(phi) - xdd)*(dphi*l*cos(phi) + dx)*m1 + (dphi*l*sin(phi) -
```

[5]:
```
rown=[]
for v in uv:
    rown.append(dAlemb.expand().coefficient(vars()['d'+repr(v)]).trig_simplify())
drugie =   [vars()[repr(v)+'dd'] for v in uv]
table(rown)
```

[5]:
```
  l^2*m1*phidd + l*m1*xdd*cos(phi) + g*l*m1*sin(phi) + l*m1*ydd*sin(phi)
-l*m1*phid^2*sin(phi) + l*m1*phidd*cos(phi) + (m1 + m2)*xdd - Fx
l*m1*phid^2*cos(phi) + l*m1*phidd*sin(phi) + g*m1 + g*m2 + (m1 + m2)*ydd - Fy
```

[6]:
```
#drugie =   [vars()[repr(v)+'dd'] for v in uv]
sol = solve( rown,drugie)
show( map(lambda x:x.trig_simplify(),sol[0]) )
```

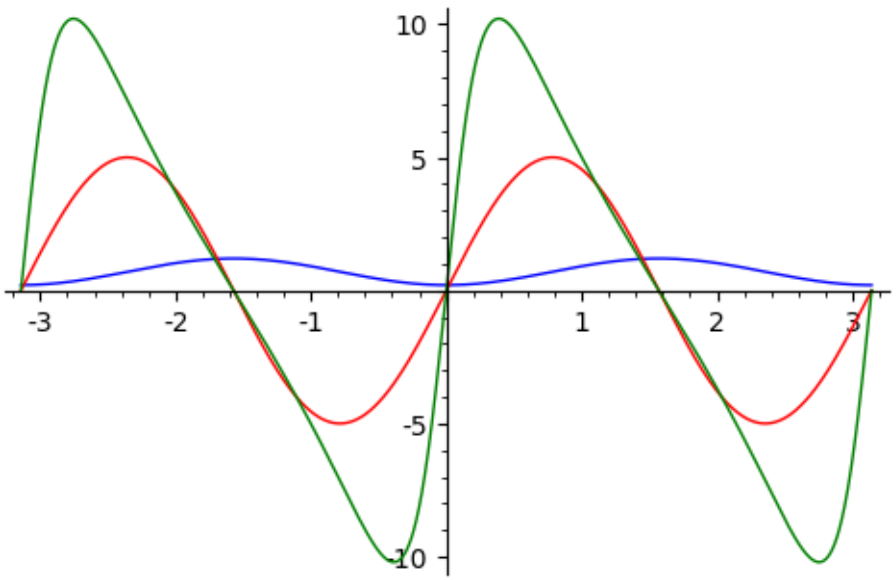
```
[phidd == -(Fx*cos(phi) + Fy*sin(phi))/(l*m2),
 xdd == (l*m1*m2*phid^2*sin(phi) + Fx*m1*cos(phi)^2 + Fy*m1*cos(phi)*sin(phi) + Fx*m2)/(m1*m
 ydd == -(l*m1*m2*phid^2*cos(phi) - Fx*m1*cos(phi)*sin(phi) - Fy*m1*sin(phi)^2 + g*m2^2 + (g
```

[7]:
```
rhs = map(lambda x:x.rhs().trig_simplify(),sol[0])
show(rhs)
```

```
[-(Fx*cos(phi) + Fy*sin(phi))/(l*m2),
 (l*m1*m2*phid^2*sin(phi) + Fx*m1*cos(phi)^2 + Fy*m1*cos(phi)*sin(phi) + Fx*m2)/(m1*m2 + m2
 -(l*m1*m2*phid^2*cos(phi) - Fx*m1*cos(phi)*sin(phi) - Fy*m1*sin(phi)^2 + g*m2^2 + (g*m1 -
```

[8]:
```
Cx(x) = .2+sin(x)^2
Cz(x) = 5*sin(x*2)
plot(Cx(x),(x,-pi,pi),figsize=5)+\
 plot(Cz(x),(x,-pi,pi),color='red')+\
 plot(Cz(x)/Cx(x),(x,-pi,pi),color='green')
```

[8]:

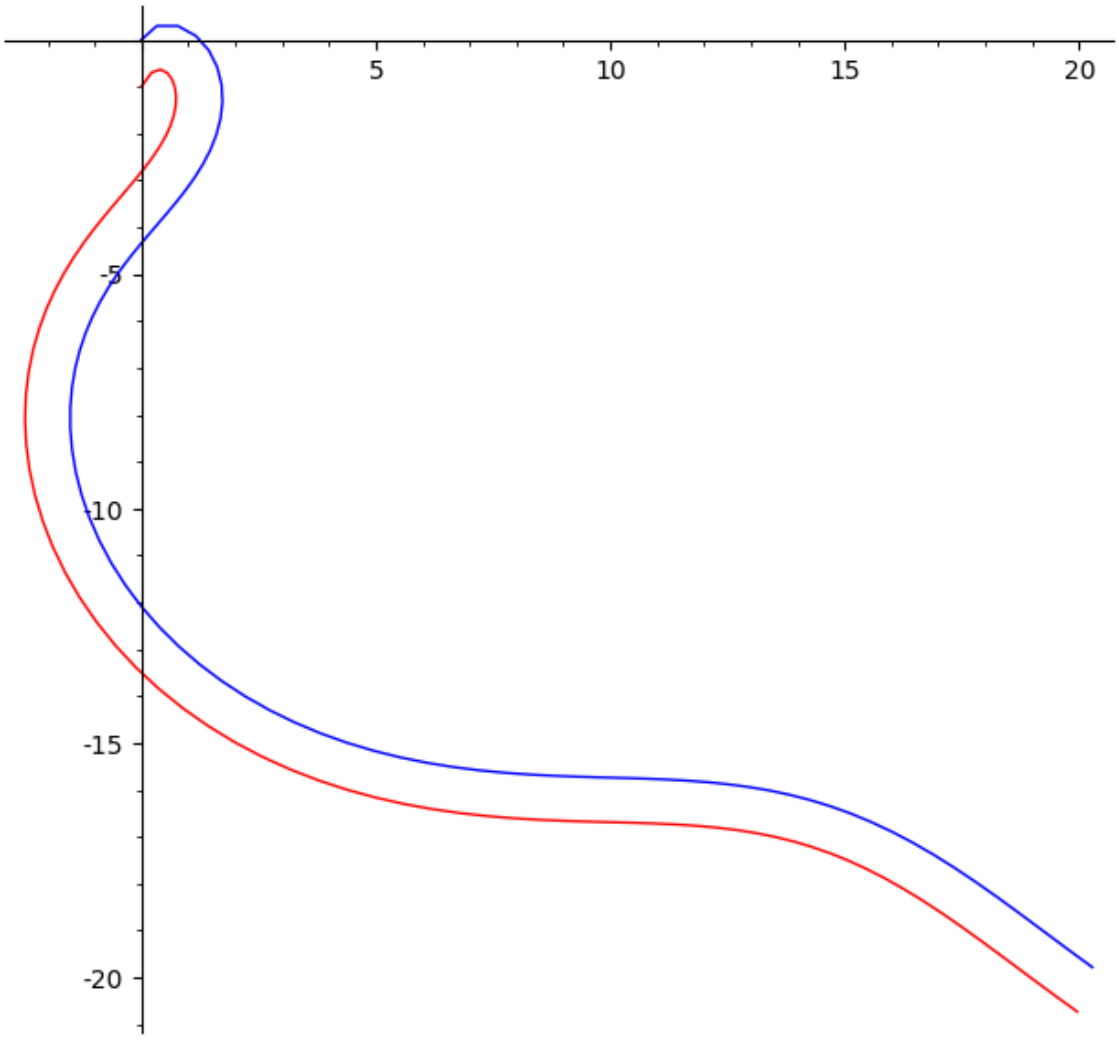


[9]:
```
#Cx = lambda x:1.0
#Cz = lambda x:1.0
trim = -0.24
alpha = arctan2(yd,xd)
forces={Fx:(-Cx(phi-alpha+trim)*cos(alpha) - Cz(phi-alpha+trim)*sin(alpha)␣
 ↪)*(xd^2+yd^2),Fy:(-Cx(phi-alpha+trim)*sin(alpha) +␣
 ↪Cz(phi-alpha+trim)*cos(alpha))*(xd^2+yd^2)}
#show( map(lambda x:x.subs(forces),rhs) )
dof3 = [phid,xd,yd] + map(lambda x:x.subs(forces),rhs)
```

[10]:
```
%%time
params = {m1:1.3,m2:2,l:1}
ode = map(lambda x:x.subs(params), dof3)
times = srange(0,8,0.1)
numsol = desolve_odeint(ode,[0,0,0,.0,2.21,4], \
                        times, [phi,x,y,phid,xd,yd])
```

```
CPU times: user 1.03 s, sys: 25.9 ms, total: 1.06 s
Wall time: 1.05 s
```

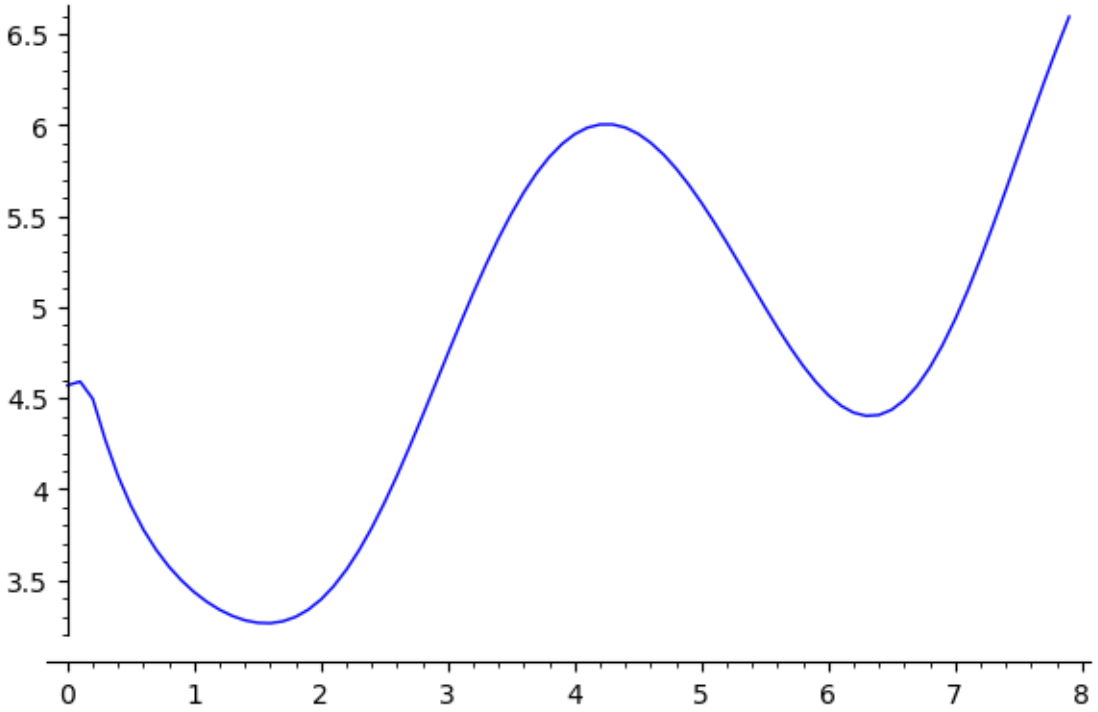
[11]:
```
p = line(zip(numsol[:,1],numsol[:,2]),figsize=9)#,marker='o')
p += line(zip(numsol[:,1]+np.sin(numsol[:,0]),numsol[:,2]-np.cos(numsol[:
 ↪,0])),color='red',aspect_ratio=1)#,marker='o')
p.show()
```



[ ]:

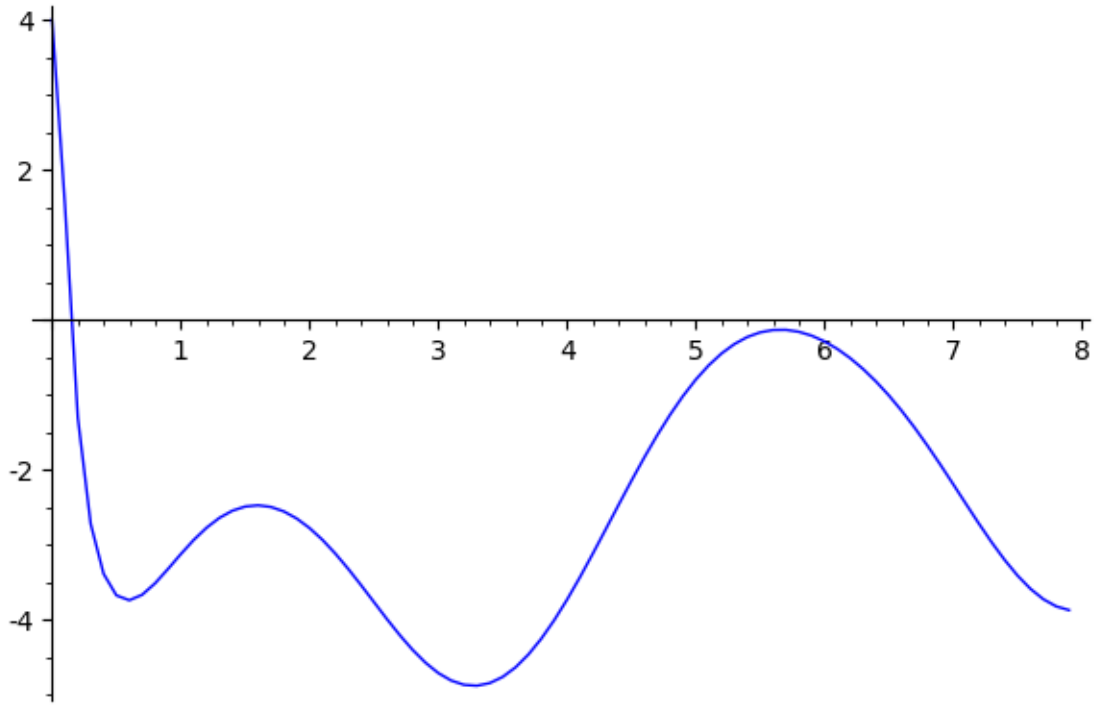
[12]:
```
line(zip(times,np.sqrt(numsol[:,4]**2+numsol[:,5]**2)) )
```

[12]:



[13]:
```
line(zip(times,numsol[:,5]) )
```

[13]:



[14]:
```
every = max(int(numsol.shape[0]/25),1)
every
```

[14]:
```
3
```

[15]:
```
def plot_system(data,l=1):
    phi,x,y = data[0:3]
    w,vx,vy = data[3:]
    x2,y2 = x+l*sin(phi),y-l*cos(phi)
    p =  point((x,y),size=40,color='blue',figsize=4)
    p += point((x2,y2),size=40,color='red')
    p += line([(x,y),(x2,y2)])
    v = vector([vx,vy])
    if v.norm()>0:
        v = v/v.norm()
        p += arrow((x,y),(x+v[0],y+v[1]),aspect_ratio=1)
    return p
```

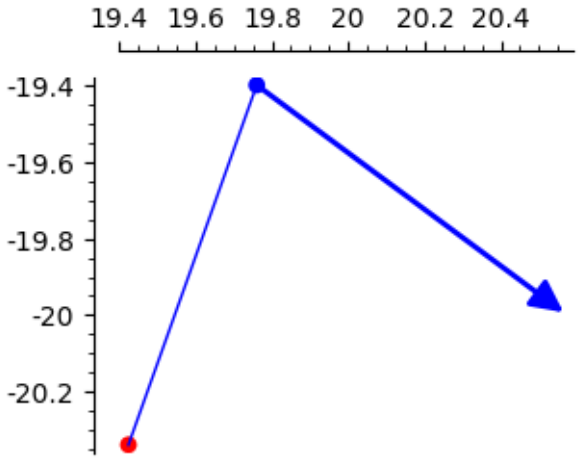
[16]:
```
plts = [p+plot_system(data,l=1) for data in numsol[::every,:]]
```

[17]:
```
%%time
anim = animate(plts,xmax=125)
anim.show()
```

```
/usr/local/lib/SageMath/local/lib/python2.7/site-
packages/sage/repl/rich_output/display_manager.py:592: RichReprWarning:
Exception in _rich_repr_ while displaying object:
Error: Neither ImageMagick nor ffmpeg appears to be installed. Saving an
animation to a GIF file or displaying an animation requires one of these
packages, so please install one of them and try again.

See www.imagemagick.org and www.ffmpeg.org for more information.
  RichReprWarning,


Animation with 27 frames



CPU times: user 86 µs, sys: 3.78 ms, total: 3.87 ms
Wall time: 4.11 ms
```

[18]:
```
plot_system(data,l=1)
```

[18]:

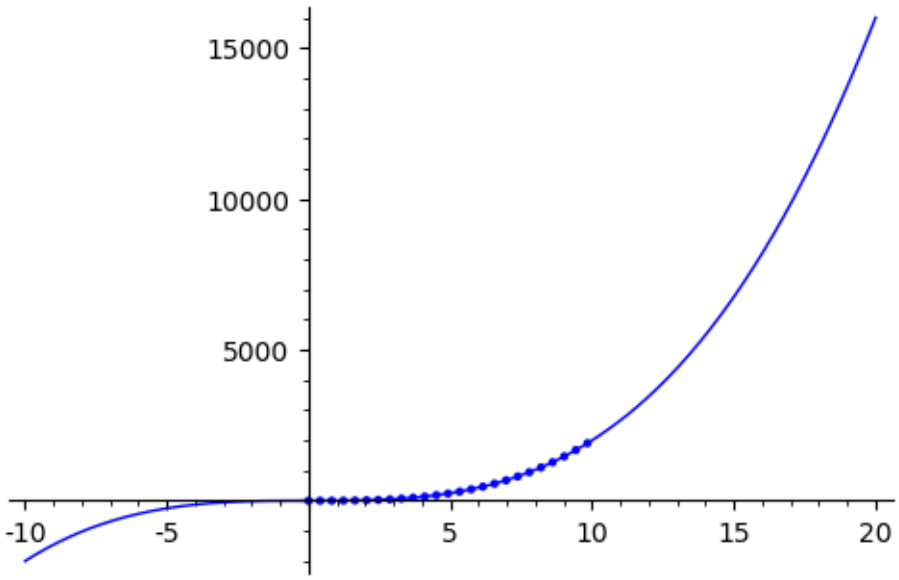


[ ]:

[ ]:

15.1 C_L from data

[19]:
```
data = [(x,2*x^3+random()*0.61) for x in srange(0.,10,0.41)]
plt = point(data)
var('c b')
model(x) = c*x^3 + b
fit = find_fit( data, model, solution_dict=True)
plt_m = plot(model(x).subs(fit),(x,-10,20))
(plt_m+plt).show(figsize=5)
model(x).subs(fit)
```



[19]: `1.9999647259094098*x^3 + 0.3487558068404126`

16 Appendix: a gentle introduction to differential equations

16.1 What is the differential equation?

The differential equation is the relationship between the function sought and its function derivative.

In the general case, we are talking about the equation of the nth rank if we have relations:

$$F(y^{(n)}(x), y^{(n-1)}(x), \ldots, y(x), x) = 0,$$

where:

- $y^{(n)}(x) = \frac{d^n f(x)}{dx^n}$.
- $y(x)$ is the function you are looking for, or a dependent variable,
- x is called an independent variable

We may also have a situation that we have m n equations on m y_i. function A special case is the m system of the first equations m degree of function. It turns out that it is possible from the n equation of this degree, create an equivalent n system of first order equations.

16.2 Example: Newton equation for one particle in one dimension

The particle's motion is described by:

$$ma = F$$

Acceleration is the second derivative of the position over time, and the force is in generality some function of x position and time. So we have:

$$m\ddot{x} = F(\dot{x}, x, t)$$

Let's introduce the new $v = \frac{dx(t)}{dt}$ function now. Substituting in the previous equation can be written:

$$\begin{cases} \dot{x} = v \\ \dot{v} = -\frac{F(\dot{x},x,t)}{m} x. \end{cases}$$

We see that we have received two systems from one equation of the second degree first degree equations.

First order equations are often presented in a form in which after the right side of the equal sign stands for the derivative and the left for the expression depending on the function:

$$\underbrace{\frac{dx}{dt}}_{\text{derivative}} = \underbrace{f(x,t)}_{\text{Right Hand Side}}$$

16.3 Geometric interpretation of differential equations.

Consider the system of two equations:

$$\begin{cases} \dot{x} = f(x,y) \\ \dot{y} = g(x,y) \end{cases}.$$

This is the so-called two-dimensional autonomous system of differential equations ordinary. Autonomy means the independence of right parties from time (ie, independent variable). An example of such a system can be traffic particles in one dimension with forces independent of time.

Equations from the above system can be approximated by substituting derivatives differential quotient:

$$\begin{cases} \frac{x(t+h)-x(t)}{h} = f(x,y) \\ \frac{y(t+h)-y(t)}{h} = g(x,y) \end{cases},$$

multiplying each equation by h and transferring a member from value dependent variables at the moment t to the right page we get:

$$\begin{cases} x(t+h) = x(t) + h \cdot f(x,y) \\ y(t+h) = y(t) + h \cdot g(x,y). \end{cases}$$

According to the definition of a derivative, in the $h \to \infty$ border of the $f(x,y)$ expression can be taken at the time between t and $t+h$. Let's assume for ease, that we will take a moment t.

Comment: this choice leads to the so-called overt algorithm if we would take a moment eg $t+h$ it would be an entangled algorithm and execution the step would be related to the solution of algebraic equations.

This system has an interesting interpretation:

- firstly, notice that the pair of functions determines the vector field on plane
- secondly, these equations give us a recipe like the value of the function in of the moment t get the value from the "next" moment $t+h$ which can be useful for recreating the $(x(t), y(t))$ curve.

16.4 Vector field

A vector field is a function that gives each point of space assigns a certain vector size. If the space will be e.g. $\mathbb{R}^2$ is a function that will consist of two functions scalar:

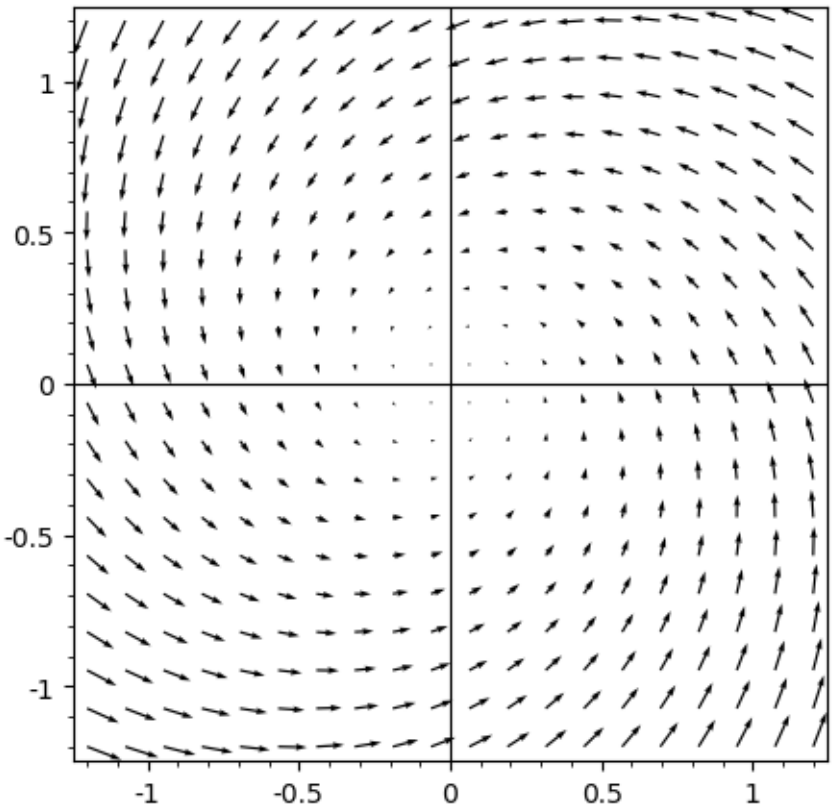
$$\vec{F}(x,y) = \begin{cases} f(x,y) \\ g(x,y) \end{cases}$$

This vector field can be visualized by drawing arrows for a certain one number of points on the plane. An example of widespread use of such vector field is wind speed field.

In Sage, we can draw a vector field with `plot _vector_ field`

[1]:
```
var('x y')
f(x,y) = -0.4*x - 0.9*y
g(x,y) = 0.9*x - 0.4*y
plt_v = plot_vector_field((f(x,y),g(x,y)),(x,-1.2,1.2),(y,-1.2,1.
↪2),figsize=6,aspect_ratio=1)
plt_v
```

[1]:



16.5 Graphical solution of the system of two differential equations

Using the derived approximated formulas to allow calculating solution of the system of differential equations at the moment $t + h$ knowing them in At the moment of Mathath2 we can try to sketch a solution based on the chart vector field. It is enough to move in small steps according to the local direction of the arrows.

Let's try to do it with the help of the algorithm:

1. we take the starting point in t
2. we calculate the point at the moment $t + h$

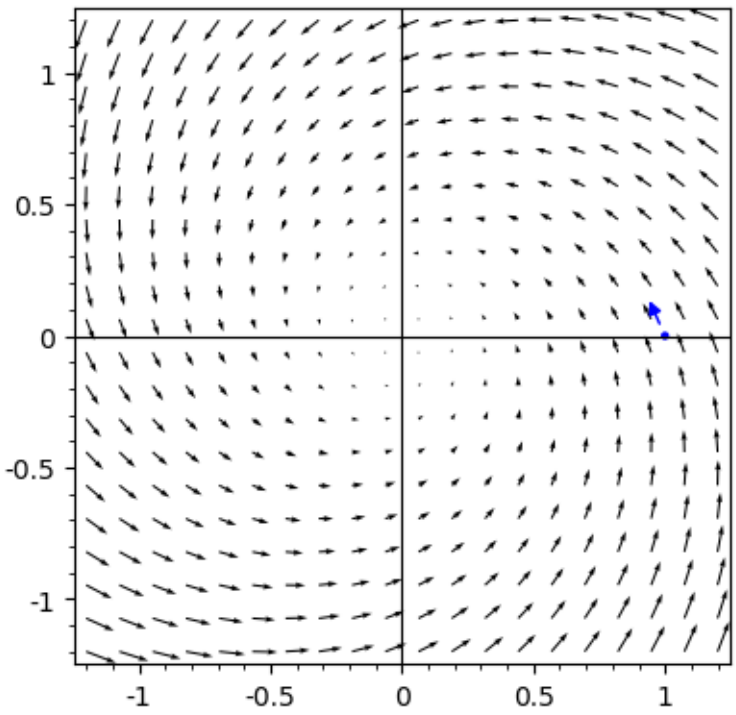
3. we draw the end point on the graph
4. we take the final point as the starting point
5. we go back to 1.

[2]:
```
x0,y0 = (1,0)
h = 0.2
```

By executing this cell many times we get the next steps of the algrorytm:

[3]:
```
x1,y1 = x0+h*f(x0,y0),y0+h*g(x0,y0)
plt_v = plt_v + point((x0,y0)) +  arrow2d( (x0,y0), (x0+h*f(x0,y0),y0+h*g(x0,y0) ␣
 ↪  ),width=1,arrowsize=2,arrowshorten=-10,aspect_ratio=1 )
x0,y0  = x1,y1
plt_v
```

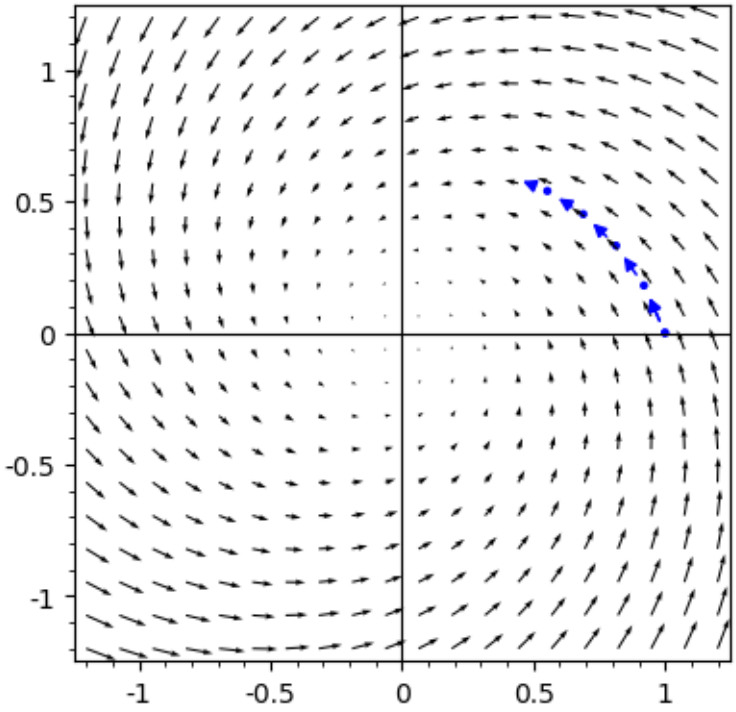
[3]:



[4]:
```
h = 0.2
x0,y0 = (1,0)
plts = [plot_vector_field((f(x,y),g(x,y)),\
                          (x,-1.2,1.2),(y,-1.2,1.2),\
                          figsize=(3,3),aspect_ratio=1)]
for i in range(5):
    x1,y1 = x0+h*f(x0,y0),y0+h*g(x0,y0)
    plt_v = plt_v + point((x0,y0)) +  arrow( (x0,y0),␣
 ↪(x0+h*f(x0,y0),y0+h*g(x0,y0)) ,width=1,arrowsize=2,arrowshorten=-10  )
```

```
    x0,y0 = x1,y1
    plts.append(plt_v)
```

[5]:
```
plts[-1].show()
# animate(plts).show()
```



We have the following conclusions:

1. The solution of the system of 2 first order equations is the curve w $\mathbb{R}^2$. space
2. The curve depends on the selection of the starting point.
3. Two solutions coming from different starting points may be go down to one point, but **can not intersect!**
4. Because we have an unlimited selection of starting points and there is (3) the solution of the system of two equations is two-parameter family of flat curves.

The differential equation (or system of equations) with the initial condition is called in the mathematics of Cauchy. Point (3) is known as Piccard's theorem on the existence and uniqueness of solutions to the problem Cauchy and it's worth noting that it imposes some restrictions on variability of the right sides of the system of equations.

16.6 Analytical solutions of differential equations

Differential equations can be analyzed using the graphical method a numeric values can be obtained with any accuracy using approximate method. These methods do not limit the form in any way right sides of the layout.

Is it possible to obtain an analytical formula for the family of functions being the solution of the differential equation?

This is difficult in the general case, however there are several forms of equations differentials in which we can always find an analytical solution. One of such cases is one separable equation of the first degree. Separability means that the right side is the product of the function x and t:

$$\frac{dx}{dt} = f(x,t) = a(x) \cdot b(t).$$

In this case, we can write the equation, treating the derivative as product of differentials:

$$\frac{dx}{dt} = a(x) \cdot b(t)$$

and integrate the above expression on both sides. Because the left side is not it explicitly includes time integration after x we are doing as if x was independent variable.

16.6.1 Example:

$$\frac{dx}{dt} = -kx$$

$$\frac{dx}{x} = -kdt$$

$$\log(x(t)) = -kt + C$$

assuming that $x > 0$. When we solve x we have:

$$x(t) = e^{-kt+C}$$

Let's see how the integration constant depends on the initial condition. Let $x(0) = x_0$, we have:

$$x(t=0) = e^{-k0+C} = e^C.$$

So we can save the solution with the initial condition $x(0) = x_0$ as:

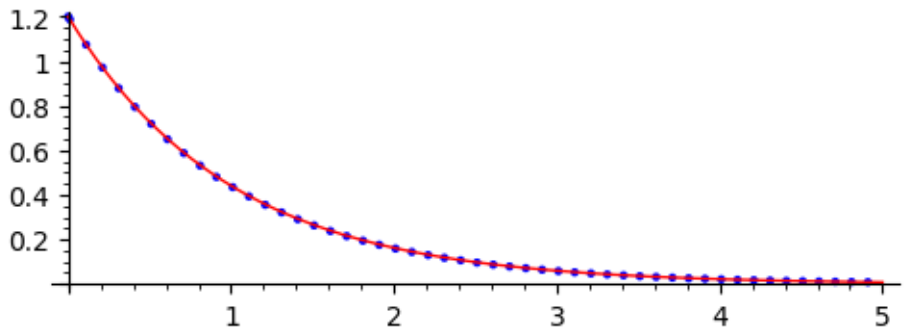
$$x(t) = x_0 e^{-kt}.$$

Let's check if this solution agrees with the obtained approximate method:

[6]:
```
L = []
k = 1.0
```

```
dt = 0.01
x0 = 1.2
X = x0
czas = 0
xt = [X]
ts = [0]
for i in range(500):
    X = X + dt*(-k*X)
    czas = czas + dt
    if not i%10:
        xt.append(X)
        ts.append(czas)

var('t')
p1 = plot( x0*exp(-k*t) ,(t,0,5),color='red',figsize=(5,2) )
p2 = point(zip(ts,xt))
p1 + p2
```

[6]:



16.7 Solving ODEs using `desolve_odeint`

There are several algorithms built in to the Sage system that significantly more accurately and more efficiently solve differential equations. Without going into the details of their implementation, it is worth learning them use.

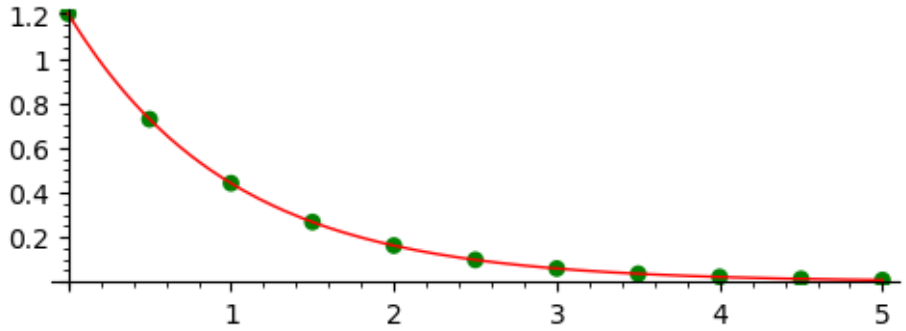
A good choice in general case is the function `desolve_ system`:

`desolve_odeint (right sides of differential equations, initial conditions, times, searched)`

For our example, we have the use of this procedure looks like the following way:

[7]:
```
f = -k*x
ic = 1.2
t = srange(0,5.01,0.5)
sol = desolve_odeint(f,ic,t,x)
```

```
p = points(zip(t,sol[:,0]),size=40,color='green')
(p1 + p).show()
print k, t
```




```
1.00000000000000 [0.000000000000000, 0.500000000000000, 1.00000000000000,
1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000,
3.50000000000000, 4.00000000000000, 4.50000000000000, 5.00000000000000]
```

The solution is passed in the form of a matrix (in fact, type np.array from the numpy package) in which for every n equations each row contains n variable values in subsequent time periods. In our case, we have one equation:

[8]:
```
sol.shape
```

[8]:
```
(11, 1)
```

[9]:
```
type(sol)
```

[9]:
```
<type 'numpy.ndarray'>
```

16.7.1 Example: harmonic oscillator

A system of two differential equations corresponding to the motion of a particle in Potential (1d)

$$U(x) = \frac{1}{2}kx^2$$

Newton's equation:

$$m\ddot{x} = ma = F = -U'(x) = -kx$$

what can you save:

$$\begin{cases} \dot{x} = v \\ \dot{v} = -kx \end{cases}$$

[10]:
```
var('t')
var('x, v')
k = 1.2
times = srange(0.0, 11.0, 0.025, include_endpoint=True)
sol = desolve_odeint([v, -k*x], [1,0], times, [x,v])
```

The solution is a numpy array (see [Introduction to numpy] (https://sage2.icse.us.edu.pl/home/pub/114/)), which can be conveniently and efficiently searched by the "slicing" technique, e.g.

[11]:
```
sol [::200, :]
```

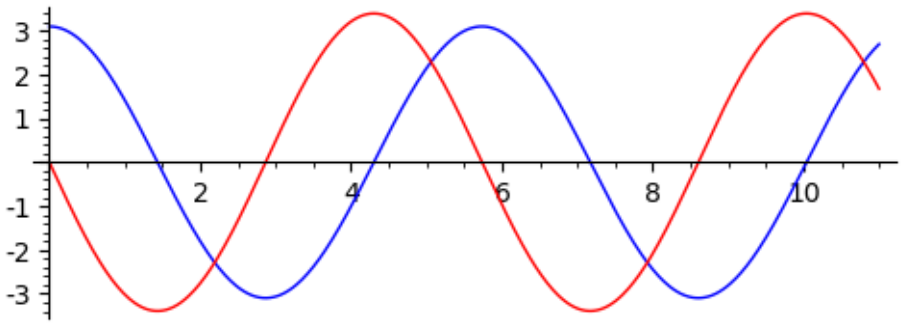
[11]:
```
array([[ 1.        ,  0.        ],
       [ 0.69241901,  0.7903589 ],
       [-0.0411118 ,  1.09451919]])
```

The parametric dependence of $(x(t), v(t))$ can be presented on the plane (X, v):

The dependencies on time, speed and position are given by functions periodic:

[12]:
```
var('x v')
k = 1.2
sol = desolve_odeint([v, -k*x], [3.1,0], times, [x,v])
px = line(zip(times,sol[:,0]),figsize=(5,2))
pv = line(zip(times,sol[:,1]),figsize=(5,2),color='red')
px+pv
```

[12]:



Because this system is known as a harmonic oscillator and we know that solution for the initial condition $x(0) = 1, v(0) = 0$ is in the form:

$$x(t) = \cos(\sqrt{k}t), v(t) = -\sin(\sqrt{k}t).$$

therefore, we can compare the result of the approximate method and the solution Analytical.

An analytical solution can also be obtained using the Sage function desolve, which solves the differential equations symbolically:

[13]:
```
sage: var('t k')
sage: assume(k>0)
sage: x = function('x')(t)
sage: de = diff(x,t,2) == -k*x
sage: desolve(de, x,ivar=t)
```

[13]:
```
_K2*cos(sqrt(k)*t) + _K1*sin(sqrt(k)*t)
```

Even if we know the form of the differential equation solution, then we can always use desolve, this is the correct application of the condition initial. Take, for example, the harmonic oscillator in which at the moment the initial $x(0) = x_0$ and $v(0) = v_0$:

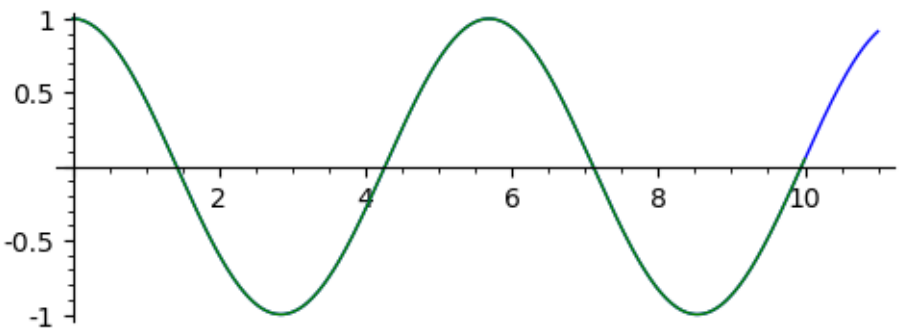
[14]:
```
var('t k')
assume(k>0)
x = function('x')(t)
de = diff(x,t,2) == -k*x
var('v0,x0')
show( desolve(de, x,ics=[0,x0,v0],ivar=t))
```

```
x0*cos(sqrt(k)*t) + v0*sin(sqrt(k)*t)/sqrt(k)
```

Let's compare the numerical and analytic solution for the condition initial $x_0, v_0 = 0, 1$:

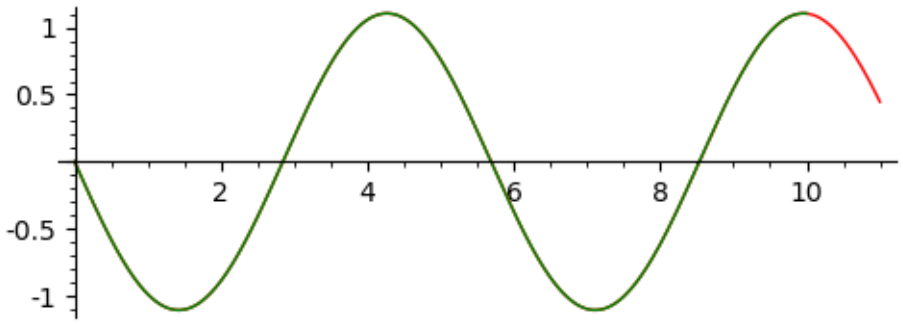
[15]:
```
sage: var('t x v')
sage: k=1.22
sage: sol = desolve_odeint([v, -k*(x)], [1.,0], times, [x,v])
sage: px = line(zip(times,sol[:,0]),figsize=(5,2))
sage: px+plot(cos(sqrt(k)*t),(t,0,10),color='green')
```

[15]:



[16]:
```
sage: var('t')
sage: pv = line(zip(times,sol[:,1]),figsize=(5,2),color='red')
sage: pv+plot(-sqrt(k)*sin(sqrt(k)*t),(t,0,10),color='green')
```

[16]:



16.7.2 Example 2: mathematical pendulum:

Newton's equation:

$$m\ddot{x} = ma = F = -U'(x) = -k\sin(x)$$

can be written in a form of a system od two ODEs:

$$\begin{cases} \dot{x} = v \\ \dot{v} = -k\sin(x) \end{cases}$$